

第 5 章 函数和模块化程序设计

作为模块化程序设计的基石，有关函数的语法、应用是本章的核心内容。本章首先介绍了 C 语言中用户自定义函数的定义、说明、调用等语法规则，并对函数的返回值、形参、实参及其参数传递方式做了详细说明。同时还介绍了两类常用的系统库函数：数学函数及随机函数。变量的作用域及生命期等概念也是函数设计中的重要知识点。此外还对函数的嵌套调用和递归调用进行了讨论。

5.1 模块化程序设计方法

当我们讨论一个较小的程序时，关注点可能在表达式、语句等语言成份上。当面对的程序有成千上万行代码时，我们一般首先将它划分为若干程序模块，每个模块用来实现一个特定的功能。然后再分别实现各个模块，组成一个完整的程序系统。这样的思路不仅易于理解、便于操作，而且由于“好的”模块便于重复使用，还可以大量减少编写重复代码的工作量，提高编程的效率。

在 C 语言中，最简单的程序模块就是函数。函数被视作程序设计的基本逻辑单位，一个 C 程序是由一个 `main()` 函数和若干个其他函数组成。程序执行从 `main()` 函数开始，由 `main()` 调用其他函数，函数之间可以相互调用。

图 5.1 是某个程序中若干函数之间的调用关系图。其中每个方框表示一个函数模块，箭头表示其间的调用关系。图 5.1 表明 `main` 函数调用了 `f1`、`f2`、`f3` 函数，相对而言，我们称 `main` 为主调函数，`f1`、`f2`、`f3` 为被调函数。`f1` 函数调用了 `g1`、`g2` 函数，`f3` 函数调用了 `g3`、`g4`、`g5` 函数。

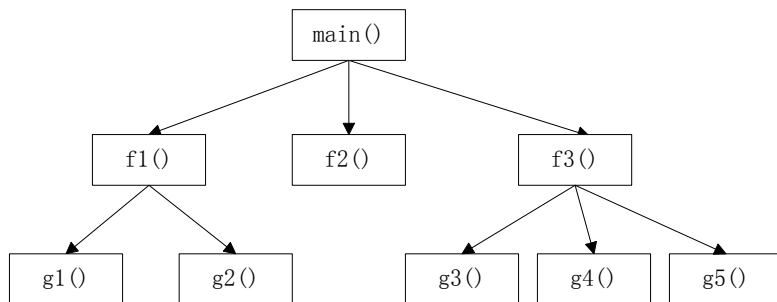


图 5.1 某程序的函数调用关系

前面的章节已经多次使用过函数，如 `printf`、`scanf` 等，只不过这些都是编译系统提供的库函数，也被称为标准函数。另一类函数是自定义函数，设计、编写它们是程序员最常见的工作。

使用 C 语言编程实际就做两件事：编制不同功能的函数；调用这些函数。本章就是在引导读者学习有关函数的语法规则的基础上，引导读者锻炼编写自定义函数的能力。

在本章的语法规则和例题讲解中，预先掌握以下模块化程序设计方法，对读者的学习是有益处的。

(1) 自顶向下对问题进行分解。程序员常常利用这个方法将大问题分解成若干小问题，直到每个小问题都便于用一个函数来实现。

(2) 尽可能地使每个函数能独立地完成一项特定的功能。虽然有时因为问题的多个方面互相牵扯,使得这个目标难以圆满实现,但这始终是程序员追求的一个目标。

(3) 尽可能地改进每个函数,使之便于将来的重复使用,即提高其复用性。

初学者若能有意识地思考、练习以上模块化程序设计方法,不仅有助于学习、理解本章的众多语法概念,更能通过对语法规则的灵活应用,达到提高编程效率的目标,为将来的程序开发工作打下坚实的基础。

5.2 用户自定义函数

函数作为程序设计的逻辑单位,相关的语法成份有:函数的定义、函数的声明、函数的调用。

【任务 5.1】用函数实现累加求和

任务描述:在任务 4.4 的基础上,设计一个函数完成对 $1+2+\cdots+n$ 求和的功能,在主程序中通过对该函数的调用实现对输入的任意整数 n 进行求和并输出。

任务分析:设计函数 `sum` 用于求和,函数输入为 n ,返回为 n 个整数的和。在主函数调用 `sum` 函数并显示结果。为此,涉及到函数的定义和声明、函数的调用,本节接下来进行介绍。

5.2.1 函数的定义与声明

1. 函数的定义

函数的定义由函数头和函数体组成,函数头是函数的接口,函数体是一个语句块,实现函数的功能。语法格式如下:

返回值类型 函数名(形式参数表)

```
{  
    函数体  
}
```

(1) 形式参数表

函数的形式参数表是一个用逗号分隔的变量名及其类型列表。根据形参的个数,函数分成无参函数和有参函数两类。

例如:

```
void show()  
{  
    printf("Hello!\n");  
}
```

这是一个无参函数,参数表是空的。在 C89 中,小括号内为空就表示参数表为空,而在 C99 中则要求在小括号内放置 `void` 关键字来明确表示无参数。

而以下函数就是一个有参函数:

```
int min(int x,int y)  
{  
    int z;
```

```

    z= x<y?x:y;
    return z;
}

```

这是一个用于求 x 和 y 二者中小者的函数。该函数有两个形式参数，定义每个形式参数时，都必须有数据类型和变量名，这一点与一般变量定义不同，读者应加以区别。

(2) 返回值类型

函数的返回值类型指定函数返回值的数据类型。函数的返回值是通过 `return` 语句获得的。`return` 语句的格式为：

```
return 表达式;
```

该语句的功能是终止当前函数的执行流程，将流程返回到主调函数，同时将表达式的值作为函数的返回值。例如，在上述 `min` 函数中，语句 `return z;` 的作用是将 z 的值作为函数结果带回到主调函数中。

在定义函数时，指定的函数返回值类型一般应该与 `return` 语句中的表达式类型一致，因此在定义 `main` 函数时，将其返回值类型定义为 `int` 型。

对于不带返回值的函数，应当用 `void` 定义函数为“空类型”（或称“无类型”），如上述 `show` 函数。此时，在函数体中可以没有 `return` 语句，例如上述 `show()` 函数，也可以有不带返回值 `return` 语句，即：

```
return;
```

但不能出现类似 `return z;` 的语句。

说明：

1) 对于 `void` 类型的函数，C89 允许函数体中包含没有返回值的 `return` 语句；但 C99 则不允许。

2) 对于非 `void` 类型的函数，C89 没有要求一定要返回一个值，如果 `return` 后没有返回值，就返回一个随机数；但 C99 要求 `return` 后必须有返回值。

3) 在 C89 中，如果函数省略了返回类型，则默认为 `int` 类型；但 C99 中要求任何函数必须有明确的返回类型，不允许默认返回类型为 `int` 的情况出现。

4) `main()` 函数也有返回值，尤其 C99 标准规定，`main()` 函数必须返回一个 `int` 值，该返回值可以反映程序执行的结果状态，以便程序调用者获知其执行状态。一般情况下，`main()` 函数的返回值是提供给操作系统使用的。

本书采用了 C99 的相关规定。

2. 函数的声明

如果在一个函数中调用另一个自定义函数，而该函数定义的位置在主调函数的后面，则应该在主调函数中对被调函数作声明。声明的作用是将函数名、函数参数个数和类型等信息通知编译系统，以便在遇到函数调用时，编译系统能够正确识别函数并检查函数调用是否合法。

函数的声明语句的书写格式如下，相比函数头，只是在末尾添加了一个分号（;）。

返回值类型 函数名(形式参数表) ;

上述 `min` 函数的声明语句如下：

```
int min(int x,int y);
```

说明：

(1) 由于函数声明与函数首部一致，因此将函数声明称为函数原型。

(2) 在函数声明中，参数名可省略，如上述函数声明可写为：

```
int min(int,int);
```

(3) 如果被调函数的定义出现在主调函数之前，可以不加以声明。

(4) 从良好的编程风格来看，将所有函数定义体一律写在 `main` 函数之后，在 `main` 函数之前，安置所有自定义函数的声明语句，不仅保证了所有函数被调用之前必有相应的函数声明，更符合自顶向下对问题分解的模块化程序设计方法。

5.2.2 函数调用

函数调用的一般形式为：

函数名(实际参数表)

如果调用无参函数，实际参数表（简称实参表）可以没有，但小括号不能省略。如果实参表包含多个实参，则各参数间用逗号隔开。实参与形参的个数应相等，类型应匹配。

按函数在程序中出现的位置来分，可以有 3 种函数调用方式：

- (1) 把函数作为一个语句。
- (2) 函数出现在一个表达式中，这种表达式称为函数表达式。
- (3) 函数调用作为另一个函数的实参。

例如：

```
#include <stdio.h>

void show();
int min(int x,int y);

int main()
{
    int min_a, min_b, x=12, y=56;
    show();                //调用方式 (1)
    min_a=min(x, y);        //调用方式 (2)
    min_b=min(36,min( x, y)); //调用方式 (3)
    printf("%d,%d\n", min_a, min_b);
}

void show()
{
    printf("Hello!\n");
}

int min(int x,int y)
{
    int z;
    z= x<y?x:y;
    return z;
}
```

【例 5.1】 简单的无参函数 `getpi`，函数功能是返回 π 值。

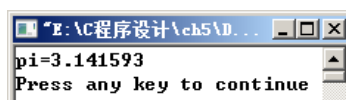
```
#include<stdio.h>
double getpi();
```

```
int main()
{
    double pi=getpi();
    printf("pi=%f\n",pi);

    return 0;
}
```

```
double getpi()
{
    double pi=3.1415926;
    return pi;
}
```

程序的运行结果截图如下：



程序分析：

以上程序中，遵循良好的程序风格，将 `getpi` 函数的说明语句置于程序首部，在 `main` 函数的定义体之后，安置 `getpi` 函数的定义体。

本例中，`getpi` 函数的返回值类型是 `double`，形参表 “()” 为空（也可以写作 “(void)”），表明函数不需要任何参数。在 `main` 函数中，调用 `getpi` 时，函数名后仍然必须带上一对括号 “()”。

因为 `getpi` 函数的返回值类型是 `double`，所以在函数体执行流程的末尾，一定是要有 `return` 语句。`return` 语句中返回值的类型应与函数类型一致。例 5.1 的执行流程见图 5.2。

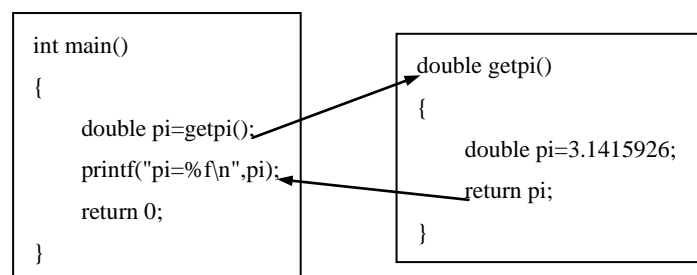


图 5.2 例 5.1 的执行流程

不是每个函数都需要返回值。C 语言规定，没有返回值的函数，它的返回值类型为 `void`，也称空类型，即该函数执行后，不会有将任何值带入主调函数函数中。请看下例。

【例 5.2】简单的有参函数 `print`，函数有一个整型参数 `n`，功能是在一行中打印 `n` 个 “*” 号。

```
#include <stdio.h>
void print(int n);

int main()
```

```

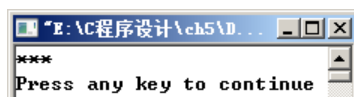
{
    print(3);
    return 0;
}

void print(int n)
{
    int i;

    for(i=0; i<n; i++)
        printf("*");
    printf("\n");
}

```

程序的运行结果截图如下：



程序分析：

以上程序中，因为函数 `print` 没有返回值，所以函数体中没有 `return` 语句。

函数调用中的“(3)”被称为实际参数表（简称为实参表），其中的“3”是实际参数（简称实参）。实参的个数、类型应与形参相同。

综合以上两例，读者应掌握函数的定义、函数的声明、函数的调用等语法规则，并对其参数表、返回值的对应关系有所认识：在函数定义中的形参表，目的是说明形参的个数、类型。在函数调用中的实参表，目的是将某个具体的数据带入函数执行流程中。二者在数量上、类型上都应该一一对应。

在编程实践中，许多初学者常常忽视这些基本语法规则。在上机编程练习时，无论是返回值类型还是参数个数、类型上的不一致，都将导致编译时的语法错误，耽误许多调试时间。

5.2.3 完成任务 5.1 的程序

参考程序：

```

#include<stdio.h>

int sum(int n);

int main()
{
    int n,s;

    printf("Please input n:");
    scanf("%d",&n);
    s=sum(n);
    printf("1+2+...+%d=%d\n",n,s);

    return 0;
}

```

```

}

int sum(int n)
{
    int i,s=0;

    for(i=1; i<=n; i++)
        s=s+i;

    return s;
}

```

程序的运行结果截图如下（假设输入：100）：



程序分析：

以上程序中，函数 `sum` 封装了求和运算的细节，主调函数 `main()` 中调用了 `sum` 函数。

5.3 系统库函数

【任务 5.2】用函数实现素数判定

任务描述：设计一个函数判断 0~999 范围内的一个随机整数是否为素数。

任务分析：

（1）考虑判断整数 `n` 是否为素数的函数实现。

在例 4.13 中，给出了一种判断素数的方法，即：将 `n` 被 2 到 `n-1` 之间的每个整数去除，

如果都不能整除，那么 `n` 就是一个素数。其实，还可以进一步简化为：如果 `n` 不能被 2 到 \sqrt{n} 之间的每个整数整除，`n` 就是素数。根据上述方法，可写出判断 `n` 是否素数的函数。

（2）在主函数中，随机生成一个整数 `n`，然后调用上述函数，以判断 `n` 是否为素数。

在实现判断素数的函数时，需要求平方根，C 语言提供了实现这一功能的库函数 `sqrt(x)`。而在主函数中需要生成随机整数，又需要借助于 C 语言提供的随机函数。

C 语言提供了大量的库函数供编程者使用，以减轻编程者的工作量。这些库函数的集合成为函数库。C 语言的库函数并不是 C 语言本身的一部分，它是由编译程序根据一般用户的需要，编制并提供用户使用的一组程序。C 的库函数极大地方便了用户，同时也补充了 C 语言本身的不足。在编写 C 语言程序时，使用库函数，既可以提高程序的运行效率，又可以提高编程的质量。

5.3.1 头文件与文件包含

文件包含是 C 语言预处理的一个重要功能。文件包含命令的作用是使编译程序将另一源文件（通常 `.h` 或 `.cpp` 等文件）嵌入带有 `# include` 的源文件，即将指定的文件插入到此命令行位置以取代该命令行，从而将指定的文件和当前的源程序文件连成一个源文件，如图 5.3

所示。

在程序设计中，文件包含很有用。一个大的程序可以分为多个模块，由多个程序员分别编写。有些公用的宏定义可单独组成一个文件，在其他文件的开头用包含命令包含该文件即可使用。这样可避免在每个文件开头都去书写那些公用量，从而便于文件共享，可以增加程序的可维护性。

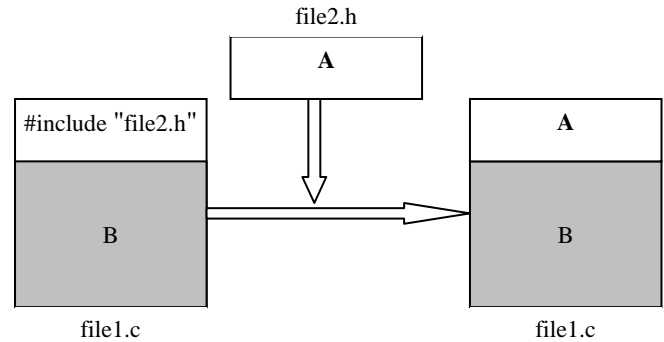


图 5.3 文件包含功能示意图

文件包含命令有两种形式：

`#include <文件名>`

`#include "文件名"`

说明：

(1) 两种形式的文件包含命令差别在于，预处理器查找要被包含的文件路径有所不同：

- 如果用尖括号标识文件名，预处理器只在系统设置的 `include` 目录(该目录可由用户在设置环境时设置的)中查找要被包含的头文件。此方法通常用来包含系统标准库的头文件。
- 如果用双引号标识文件名，预处理器先在被编译程序所在的目录中查找要被包含的文件，如果未发现文件，则再在系统设置的 `include` 目录中查找。此方法通常用在将程序员定义的头文件包含到程序中。

(2) 常见的被包含文件是以 `.h` 为后缀的头文件，包括由编译系统提供的系统头文件和用户自定义的头文件。

(3) 文件包含可以嵌套。

根据库函数的功能，C 语言编译系统将其组织在相应的头文件中，表 5-1 列出常用的头文件。

表 5-1 常用头文件

库函数类别	头文件	库函数类别	头文件
标准输入输出函数	stdio.h	字符串函数	string.h
数学函数	math.h	动态内存分配函数	malloc.h
字符函数	ctype.h	标准库函数	stdlib.h

如果使用库函数，应该在程序文件开头用 `#include` 命令，将调用相应的头文件包含到本文件中。

5.3.2 数学函数

C 语言编译系统提供的 `math.h` 头文件中, 包含了常用的数学函数, 如: 求绝对值函数 `fabs(x)`, 求平方根函数 `sqrt(x)`等。表 5-2 给出了常用数学函数。

表 5-2 常用数学函数

函数原型	功能	说明
<code>int abs(int x)</code>	求整数 x 的绝对值	
<code>double acos(double x)</code>	计算 $\cos^{-1}(x)$ 的值	$x \in [-1, 1]$
<code>double asin(double x)</code>	计算 $\sin^{-1}(x)$ 的值	$x \in [-1, 1]$
<code>double atan(double x)</code>	计算 $\tan^{-1}(x)$ 的值	向上取整
<code>double ceil(double x)</code>	求出不小于 x 的最大整数	
<code>double cos(double x)</code>	计算 $\cos(x)$ 的值	x 的单位为弧度
<code>double exp(double x)</code>	计算 e^x 的值	
<code>double fabs(double x)</code>	求 x 的绝对值	
<code>double floor(double x)</code>	求出不大于 x 的最大整数	向下取整
<code>double log(double x)</code>	计算 $\log_e x$, 即 $\ln x$	
<code>double log10(double x)</code>	计算 $\log_{10} x$	
<code>double pow(double x, double y)</code>	计算 x^y 的值	
<code>double sin(double x)</code>	计算 $\sin(x)$ 的值	x 的单位为弧度
<code>double sqrt(double x)</code>	计算 x 的平方根	
<code>double tan(double x)</code>	计算 $\tan(x)$ 的值	x 的单位为弧度

【例 5.3】从 $30^\circ \sim 180^\circ$, 每隔 30° 打印三角函数的值。

参考程序:

```
#include <stdio.h>
#include <math.h>
#define PI 3.14

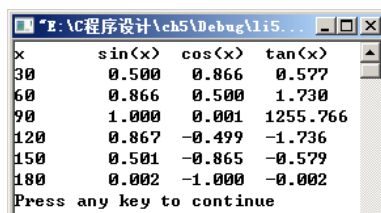
int main()
{
    int i;
    double s,c,t;

    printf("x\tsin(x)\tcos(x)\ttan(x)\n");
    for(i=30;i<=180;i=i+30)
    {
        s=sin(i*PI/180);
        c=cos(i*PI/180);
        t=tan(i*PI/180);
        printf("%d\t%6.3f\t%6.3f\t%6.3f\n",i,s,c,t);
    }

    return 0;
}
```

```
}
```

程序的运行结果截图如下：



x	sin(x)	cos(x)	tan(x)
30	0.500	0.866	0.577
60	0.866	0.500	1.730
90	1.000	0.000	1255.766
120	0.867	-0.499	-1.736
150	0.501	-0.865	-0.579
180	0.002	-1.000	-0.002

Press any key to continue

程序说明：

(1) 由于库函数中的三角函数参数必须是弧度，因此需要将整数 i 通过 $i*PI/180$ 转换成弧度。

(2) 为了显示整齐，使用了 `printf` 函数的格式输出。

5.3.3 随机函数

C 语言提供了产生随机数的函数 `rand()`，该函数会返回位于在 0 至 32767 之间的一个随机整数。如果未设随机数种子，`rand()`在调用时会自动设随机数种子为 1，每次执行时是相同的。因此要产生不同的随机数，在调用此函数前，必须先利用 `srand()`设好随机数种子。通常可调用 `time()`函数得到当前系统时间，并将该时间作为随机种子。

函数 `rand()`和 `srand()`在头文件 `stdlib.h` 中，函数 `time()`在头文件 `time.h` 中。

【例 5.4】产生 0~99 之间的随机数。

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main()
{
    int r1,r2;

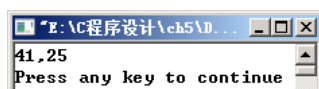
    r1=rand()%100;

    srand(time(NULL));
    r2=rand()%100;

    printf("%d,%d\n",r1,r2);

    return 0;
}
```

程序第一次的运行结果截图如下：



41,25
Press any key to continue

程序第二次的运行结果截图如下：



程序运行结果分析:

从两次结果可以看出采用 srand()函数设置随机数种子的效果。

5.3.4 完成任务 5.2 的程序

参考程序:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
int prime(int);

int main()
{
    int r;

    srand(time(NULL));
    r=rand()%1000;

    if(prime(r))
        printf("%d is prime.\n",r);
    else
        printf("%d is not prime.\n",r);

    return 0;
}

int prime(int n) //判断 n 是否为素数的函数
{
    int m;

    for(m=2;m<=sqrt(n);m++)
    {
        if(n%m==0)
            return 0;
    }
    return 1;
}
```

5.4 变量的作用域

5.4.1 局部变量

在函数内部定义的变量，包括形参，一律称为局部变量。局部变量只能在所属函数中使用，即其作用域是所属函数。

在程序设计中，局部变量是使用最多的标识符。各个函数都有若干局部变量，但不同函数之间的局部变量，即使重名，也互不影响。

以下是一个程序框架示例，其中 3 个函数各有若干局部变量。

```
int main()
{ int m,n;    // m,n 是 main 函数中的局部变量
  .....
}
int f1(int a)
{ int b,c;    // a,b,c,m 是 f1 函数中的局部变量
  int m;
  .....
}
int f2(double x,double y)
{ double z,w; // x,y,z,w,n 是 f2 函数中的局部变量
  int n;
  .....
}
```

值得注意的是，在 main 函数和 f1 函数中都有局部变量 m，但二者是相互独立的两个内存单元。main 函数的 n 和 f2 函数中的 n 也属这种情况。

在程序执行中，函数的执行状态决定于其间的所有局部变量的值，只有清晰地了解局部变量的变化，才能真正理解函数的流程。

还有一类作用域更小的局部变量，它定义在一个复合语句之中。从定义之处到复合语句结束的范围，是它的作用域。这种更“局部”的局部变量的意义在于，使结构化程序中的变量组织也结构化了。

【例 5.5】 更“局部”的局部变量。

```
#include <stdio.h>

int main()
{
    int i,s=0;        // 局部变量

    for(i=1; i<=10; i++)
    {
        int j=2*i;    // 更局部的局部变量
        s=s+j;
    }
    printf("s=%d\n",s);
}
```

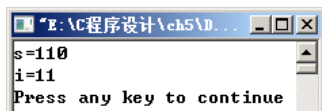
```

printf("i=%d\n",i);
// printf("j=%d\n",j);    // 注释句

return 0;
}

```

程序的运行结果截图如下：



程序分析：

以上程序中，s 和 i 是 main 函数中的局部变量，j 是循环体复合语句中的局部变量。在 main 函数的尾部，可以正常打印 s 和 i，但不能打印 j，因为 j 的作用域只是 for 语句的循环体。若删除注释句的注释符号，则将造成语法错误，j 是 “undeclared identifier”。

5.4.2 全局变量

定义在函数外部的变量，称为全局变量。从全局变量定义之处至文件末尾的所有函数都是其作用域，都可引用它。以下是程序框架，其中定义了两组全局变量 p、q 和 c1、c2。

```

int p=1,q=2;
int main()
{ ..... }
char c1,c2;
int f1(int a)
{ ..... }
int f2(double x,double y)
{ ..... }

```

以上程序框架中，全局变量 p、q 可以被所有函数读写，全局变量 c1、c2 可以被其后续的函数 f1、f2 读写。

5.4.3 重名问题

全局变量不能重名，同一个函数中的局部变量也不能重名。重名将导致 “redefinition” 的语法错误。在一个较大规模的程序中，若全局变量较多，必将增大代码的理解难度，提高程序维护的成本，所以良好的编程风格是全局变量尽可能的少用！

各函数内的局部变量有各自独立的作用域，函数之间重名的局部变量各自代表了不同的内存单元，所以它们可以重名。因此，局部变量的命名有很大的自由度，但编程者还是应遵循良好的命名风格。

若某函数中的局部变量与全局变量同名，那么局部变量在其作用域中屏蔽同名的全局变量。这种现象常被戏称为 “强龙斗不过地头蛇”。

【例 5.6】 验证局部变量与全局变量同名时的规则。

```
#include<stdio.h>
```

```

void f1();
void f2();

```

```

int n=1;    // 全局变量 n

int main()
{
    f1();
    f2();
    printf("main(): n=%d\n",n);

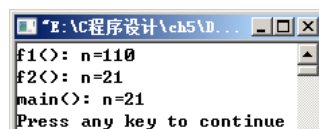
    return 0;
}

void f1()
{
    int n=100;    // 局部变量 n
    n=n+10;      // 此处 n 是局部变量
    printf("f1(): n=%d\n",n);
}

void f2()
{
    n=n+20; // 此处 n 是全局变量
    printf("f2(): n=%d\n",n);
}

```

程序的运行结果截图如下：



程序分析：

以上程序中，有全局变量 `n`，在函数 `f1` 中定义了一个重名的局部变量 `n`。在函数 `f1` 中，局部变量 `n` 屏蔽了全局变量 `n`，输出的是局部变量的值；在函数 `f2` 和 `main` 中，变量 `n` 是同一个全局变量，输出的是全局变量的值。

5.5 变量的生存期

变量的存储属性包括变量的存储位置、存在时间（生存期）等性质。了解它，有助于理解程序运行时内存的结构。

一个编译后的 C 程序一般创建并使用 4 个内存分区，如图 5.4 所示。

- 程序代码区，存放可执行代码。
- 静态数据区，存放全局变量和静态变量。
- 堆区，程序员可利用 C 语言的动态内存分配函数进行管理的自由内存区域。
- 栈区，存放函数调用时的返回地址、函数参数、局部变量以及 CPU 的当前状态。

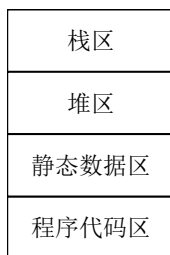


图 5.4 C 程序的内存分区

静态数据区中的数据，在程序开始执行时，被系统分配空间，在程序执行完毕之时，系统自动回收空间。

栈区中的数据，在某个函数开始执行时，被系统分配空间；在该函数执行完毕之时，系统自动回收空间。

C 语言中，依据变量的存储位置，将变量分为两大类：存储于静态数据区的变量称为静态变量；存储于栈区的变量称为动态变量。

5.5.1 动态变量

动态变量的类型说明符是 `auto`。由于在编程中，动态变量使用最多，C 语言规定函数内的局部变量如未加存储类型说明，都是动态变量。如 “`int x`” 等价于 “`auto int x`”，都是定义动态变量 `x`。静态变量有两种，一是全局变量；二是在类型定义符之前，加上 `static` 修饰的局部变量。

系统为动态变量分配内存空间的时机，是其所在函数每次开始执行之时。当所在函数执行完毕之时，系统自动回收动态变量的内存空间。由于不能预先确定函数的执行空间，所以动态变量的初值都是随机值。

5.5.2 静态变量

系统为全局变量分配内存空间的时机，是程序开始执行之时；为 `static` 修饰的局部变量分配内存空间的时机，是其所属函数第一次被调用之时。直到程序执行完毕之时，系统自动回收所有静态变量的内存空间。若在定义静态变量时未初始化其值，则它们的初值一律是 0 值。

【例 5.7】验证动态变量与静态变量的存储机制。

```
#include <stdio.h>
void f();
int main()
{
    f();
    f();
    f();
    return 0;
}

void f()
{
```

```

    auto int a=1;    // 动态变量
    static int b=1;  // 静态变量
    a++; b++;
    printf("a=%d,b=%d\n",a,b);
}

```

程序的运行结果截图如下：



程序分析：

以上程序中，函数 main 以完全相同的方式调用了三次 f 函数，结果却有差异。在函数 f 中，有动态变量 a 和静态变量 b。在每次调用函数 f 时，变量 a 被重新分配空间，重新初始化为 1。所以每次调用函数 f，a 的输出值都是 2。

静态变量 b 则不然，在第一次调用函数 f 时，b 被分配空间，并初始化为 1，经过运算，在函数 f 结束之时，b 值为 2 输出。但函数 f 运行结束后，变量 b 的空间并未释放。第二次调用函数 f 时，静态变量 b 还保留着上次函数执行完时的数值。这就使得 b 的第二次输出值为 3。第三次调用的情形与第二次调用的情形相同。

一般说来，用相同的参数调用同一个函数，其功能、返回值应相同的。但也可借助于静态变量，记忆函数上一次执行的状态，使函数的功能更加灵活。这在某些应用中显得格外有意义。

5.6 函数的嵌套调用

【任务 5.3】方程近似解

任务描述：用弦截法求方程 $f(x)=x^3-5x^2+16x-80=0$ 的近似解。

任务分析：弦截法是求解高次方程近似解的常用方法。参见图 5.5，若已知方程在区间 $[x_1, x_2]$ 内存在解，则 $f(x_1)$ 和 $f(x_2)$ 必定一正一负。连接曲线上两点 $(x_1, f(x_1))$ 和 $(x_2, f(x_2))$ 作弦，该弦与 x 轴必定有交点 x。用 x 替代 x_1 或 x_2 ，就缩小了求解近似解的区间。

在图 5.5 中， $f(x_1)<0$ ，且 $f(x)<0$ ，则将 x 替代 x_1 ，使原求解区间 $[x_1, x_2]$ 缩小为 $[x, x_2]$ 。重复以上求交、缩小区间的步骤，当求解区间的长度足够小时， x_1 或 x_2 都可视作方程的近似解。

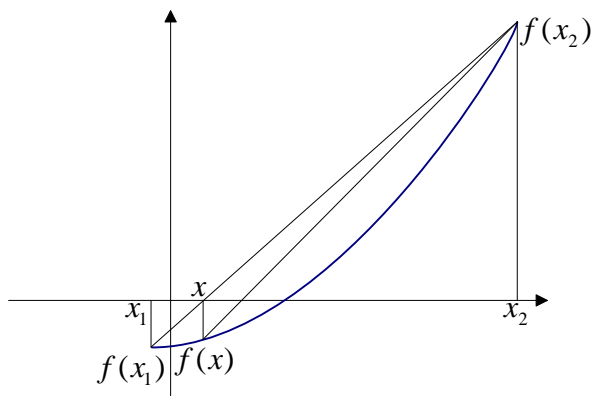


图 5.5 弦截法求方程的近似根

以下分步骤描述程序的开发过程：

(1) 在弦截法的求解过程中，需要反复计算表达式 $f(x)=x^3-5x^2+16x-80$ 的值，所以设计函数 f 完成表达式的计算。

(2) 在弦截法的求解过程中，需要多次为计算弦与 x 轴的交点，设已知区间为 $[x_1, x_2]$ ，则 $(x_1, f(x_1))$ 至 $(x_2, f(x_2))$ 的弦与 x 轴的交点处的 x 值的计算公式如下：

$$\text{弦的斜率 } k = \frac{f(x_2) - f(x_1)}{x_2 - x_1}, \text{ 交点处的 } x = x_1 - \frac{f(x_1)}{k}。$$

设计函数 getx ，完成上述计算。

(3) 为实现弦截法流程，设计函数 getroot ，完成方程近似解的计算。设当近似解 x 对应的 $|f(x)| < 10^{-6}$ 时，即可作为函数的返回值。

(4) 函数 $f(x)=x^3-5x^2+16x-80$ 可以估计的解区间大致在 $[0, 10]$ 之间。为最终求解本任务，设计函数 main 。

5.6.1 函数的嵌套调用

在 5.2 节中的自定义函数，总是被函数 main 调用。其实函数之间可以相互调用，这就是函数的嵌套调用（但函数不能嵌套定义）。它体现了自顶向下的模块化程序设计方法。

图 5.6 表示的是三层嵌套：

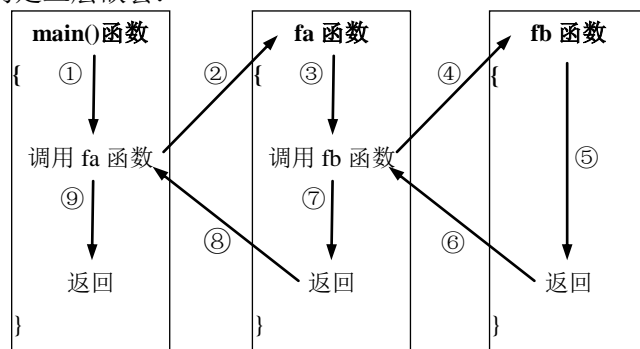


图 5.6 函数嵌套调用

其执行过程是：

- ① 执行 main 函数；
- ② 遇到调用 fa 函数的语句，流程转到 fa 函数；
- ③ 执行 fa 函数；
- ④ 遇到调用 fb 函数的语句，流程转到 fb 函数；
- ⑤ 执行 fb 函数，若 fb 函数中再未调用其他函数，则完成 fb 函数的全部操作；
- ⑥ 返回 fa 函数中调用 fb 函数的位置；
- ⑦ 继续执行 fa 函数中尚未执行完的部分，直到完成 fa 函数的全部操作；
- ⑧ 返回 main 函数中调用 fa 函数的位置；
- ⑨ 继续执行 main 函数中尚未执行完的部分，直到完成 main 函数的全部操作。

5.6.2 完成任务 5.3 的程序

参考程序：

```

#include <stdio.h>
#include <math.h>
double f(double);

```

```

double getx (double,double);
double getroot(double,double);

int main()
{
    double x1=0,x2=10;
    double x=getroot(x1,x2);
    printf("f(%f)=%f\n",x,f(x));

    return 0;
}

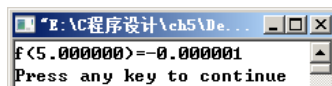
double f(double x)
{
    return ((x-5)*x+16)*x-80;
}

double getx (double x1,double x2)
{
    double k=(f(x2)-f(x1))/(x2-x1);
    return x1-f(x1)/k ;
}

double getroot(double x1,double x2)
{
    double y1=f(x1), y2=f(x2);
    while( 1 )
    {
        double x=getx(x1,x2); // 求弦与 x 轴的交点
        double y=f(x);
        if(fabs(y)<1e-6) break;
        if(y*y1>0) { y1=y; x1=x; }
        else { y2=y; x2=x; }
    }
    return x1;
}

```

程序的运行结果截图如下：



程序分析：

本程序中共 4 个函数，彼此之间的调用关系参见图 5.7。随着编程实践的逐步拓展、深入，读者编写的程序中将会有越来越多的函数、越来越复杂的调用关系。

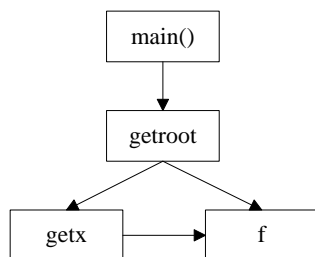


图 5.7 任务 5.3 的函数调用关系

5.7 递归函数

【任务 5.4】求阶乘

任务描述：计算 $\text{sum}=1 \times 2 \times 3 \times 4 \times \cdots \times n$ 。

任务分析：第 4 章中已经介绍过用循环实现阶乘的思想。本次任务我们仔细观察阶乘计算的特殊性。用 $\text{Fac}(n)$ 表示 $n!$ ，显然 $\text{Fac}(n)$ 的计算可依赖于对 $\text{Fac}(n-1)$ 的计算，即：

$$\text{Fac}(n) = n \times \text{Fac}(n-1)$$

这样，就将规模为 n 的问题分解成了同类的规模为 $n-1$ 的问题；同理， $\text{Fac}(n-1)$ 的计算可依赖于对 $\text{Fac}(n-2)$ 的计算，即：

$$\text{Fac}(n-1) = (n-1) \times \text{Fac}(n-2)$$

这样，就将规模为 $n-1$ 的问题分解成了同类的规模为 $n-2$ 的问题；……；这样依次类推，问题规模不断缩小，直到 $n=1$ 时，可以直接给出结果 1。因此 $n!$ 存在如下递推公式：

$$\text{Fac}(n) = \begin{cases} 1 & n=1 \\ n \times \text{Fac}(n-1) & n>1 \end{cases}$$

对于这类问题，C 语言中可以使用递归函数来实现。接下来介绍递归函数的定义与调用方法。

5.7.1 函数的递归定义与调用

函数之间不仅可以相互调用，函数也可以直接或间接调用自身，这被称为递归调用。从语法上看，递归调用与嵌套调用没有任何区别，但从逻辑上看，由于递归调用具有较高的抽象性，对它其的理解掌握有一定的难度。

递归函数的思维基础是数学归纳法。当使用数学归纳法证明某命题时，一般分为两个步骤：

- 证明 $n=1$ 的初始情形时，命题成立；
- 如果当 $n=m$ 时命题成立，那么当 $n=m+1$ 时，命题同样成立。

在递归函数的流程中，一般也包括两种处理：

- 在某种条件下的简单处理：无需递归调用，直接处理完成；
- 问题分解策略：将递归函数处理的问题转换为类型相同、但规模较小的问题，进行递归调用。

如果忽视静态变量的影响，可以认为函数的执行状态是由其中的局部变量组成的。当一个函数递归调用自身时，系统将自动生成一个全新的局部变量集合，得到该函数的一个全新的执行过程。这些不同的执行过程，对应的函数名、局部变量名、代码完全相同，但其中同

名的局部变量都对应着不同的内存单元。为避免混淆递归函数的不同的执行过程，读者可以参考附录 3，在单步调试程序时，使用“函数调用栈”、“观察变量”等手段，观察递归调用的各个层次及每个层次中的变量，加深对此的理解。

5.7.2 完成任务 5.4 的程序

参考程序：

```
#include <stdio.h>

int Fac(int);

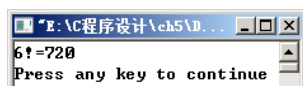
int main()
{
    int m,n=6;

    m=Fac(n);
    printf("%d!=%d\n",n,m);

    return 0;
}

int Fac(int n)
{
    if(n==1)                //递归的结束条件
        return 1;
    else
        return n*Fac(n-1); //大问题转换为小问题
}
```

程序的运行结果截图如下：



5.8 函数应用程序设计实例

【例 5.8】设计函数，计算 $x = \sqrt{a}$ 。已知计算 $x = \sqrt{a}$ 的迭代公式是 $x_{n+1} = \frac{1}{2} \left(x_n + \frac{a}{x_n} \right)$ ，要求计算精度为 10^{-6} 。

分析：

自定义一个函数用于根据迭代公式计算算术平方根，然后在主函数中调用该自定义函数。

参考程序：

```
#include <stdio.h>
#include <math.h>
double getsqrt(double a);
```

```

int main()
{
    double x[5]={1,3,5,7,9},    //计算 1、3、5、7、9 这 5 个数的算术平方根
        xsqrt[5];                //计算结果保存在 xsqrt 数组中
    int i;

    for(i=0;i<5;i++)
    {
        xsqrt[i]=getsqrt(x[i]);
        printf("%11.9f\t%11.9f\n",xsqrt[i],sqrt(x[i]));
        //将 getsqrt()函数的计算结果与直接调用 sqrt()函数的计算结果均输出，以进行比较
    }

    return 0;
}

double getsqrt(double a)
{
    double x1=a,x2;

    while(1)
    {
        x2=(x1+a/x1)/2;
        if(fabs(x1-x2)<1e-6)
            break;
        x1=x2;
    }

    return x1;
}

```

程序的运行结果截图如下：



程序分析：

以上程序中，函数 `getsqrt` 封装了开方运算的细节，主调函数在循环中，以数组元素 `x[i]` 作为实参，多次调用 `getsqrt` 函数，计算结果存于数组 `xsqrt` 之中。程序输出的第一列是函数 `getsqrt` 的计算值，第二列是利用系统库函数 `sqrt` 的计算值，二者对比可以看出计算精度确实是 10^{-6} 。

在函数调用中，实参的形式可以是常量、变量、数组元素、表达式等。以下是一些 `getsqrt` 函数的调用示例：

```
double y=getsqrt(4);
```

```
double z=getsqrt(x+y);
double w=getsqrt(getsqrt(z)+10);
```

【例 5.9】设计递归程序，将正整数的各位数字逐个打印出来。

分析：

本例可将打印 n 的各位数字问题分解为：打印 $n/10$ 的各位数字，然后打印 n 的个位数字。

参考程序：

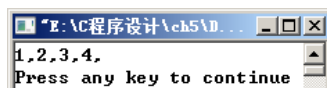
```
#include <stdio.h>
void printd(int n );

int main()
{
    printd(1234);
    printf("\n");

    return 0;
}

void printd(int n )
{
    if(n==0) return;
    printd(n/10);
    printf("%d,",n%10);
}
```

程序的运行结果截图如下：



程序分析：

实际上，实现本例更一般做法是，将正整数的各位数字取出，存于数组之中，再逆序输出。本例有意采用了递归函数的做法，也有巧妙之处。函数 `printd` 中包含两个子处理方法：①当参数 n 为 0 时，直接返回；②当参数 n 大于 0 时，将打印 n 的各位数字问题分解为：（递归调用）打印 $n/10$ 的各位数字，然后打印 n 的个位数字。

参见图 5.8，针对函数 `main` 中的函数调用语句，引发的递归调用深度竟有 4 层。函数 `main` 在执行中调用 `printd(1234)`；`printd(1234)` 在执行中调用 `printd(123)`；……；`printd(1)` 在执行中调用 `printd(0)`；之后 `printd(0)` 执行完成，`printd(1)` 执行完成，……；`printd(1234)` 执行完成；`main` 函数执行完成。

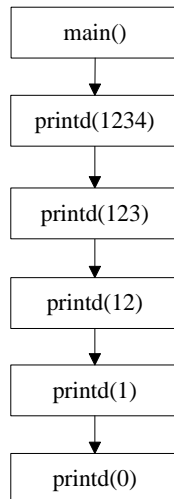


图 5.8 例 5.9 的函数调用关系

【例 5.10】 设计递归程序，计算 Fibonacci 数列第 5 项元素的值。

分析：

斐波那契数列中的各项存在如下关系：

$\text{Fib}(0)=1, \text{Fib}(1)=1,$

$\text{Fib}(n)=\text{Fib}(n-1) + \text{Fib}(n-2) \quad (n \geq 3)$

参考程序：

```
#include <stdio.h>
```

```
int Fib(int n);
```

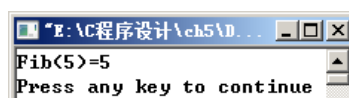
```
int main()
```

```
{
    printf("Fib(5)=%d\n",Fib(5));
    return 0;
}
```

```
int Fib(int n)
```

```
{
    if(n==1||n==2)
        return 1;
    return Fib(n-1)+Fib(n-2);
}
```

程序的运行结果截图如下：



程序分析：

(1) 以上程序中，函数 Fib 的两个子处理方法是：

- 当参数 n 为 1 或 2 时，直接返回 1；
- 当参数 n 大于 2 时，递归调用函数 Fib，分别计算第 n-2 项、第 n-1 项的值，再返

回其和。

(2) 本例中函数 `Fib` 的递归调用关系如图 5.9 所示。显然，当函数 `Fib` 的实参值较大时，该函数的被调用次数将迅速增大，频繁的函数调用，必然增加系统开销，影响程序的效率。

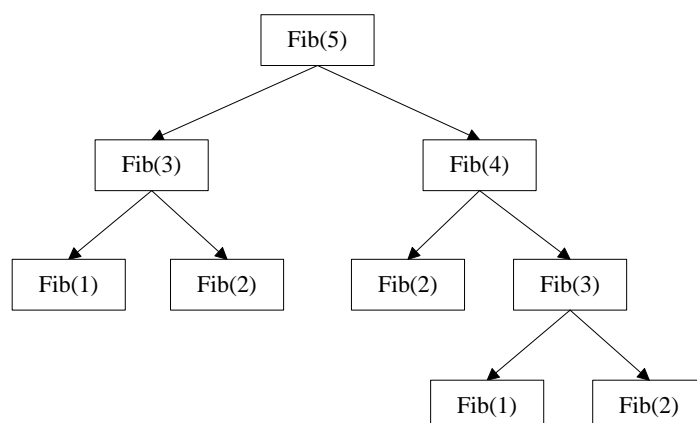


图 5.9 例 5-21 的函数调用关系

但是，笔者同意尹宝林在其《C 程序设计思想与方法》一书中的论述，即那种认为递归函数会极大地降低程序运行的效率的看法并不正确。

递归作为一种调用自身的函数，与一般函数并没有太大的区别。唯一的区别是由于递归调用，有可能使得函数嵌套调用的层次比较多。比起可能替代递归的循环计算，递归所付出的额外代价就是执行函数调用所需要执行的代码，其中主要的就是函数调用所需要的参数的传递和函数运行环境的保存。现代计算机的 CPU、内存等资源的性能比早期的计算机有了很大的提高，完全可以满足合理使用递归函数时的要求。因此，在一定范围内，函数调用的嵌套并不会显著降低递归函数的运行速度。

实际上，影响递归函数效率的不是递归本身，而是我们对于递归不正确的使用。可能影响递归函数执行效率的原因主要有以下两点：

- 函数体简单，参数传递过多，使得函数调用过程所占用的比例过大。例如，如果函数体内只有一两个简单的加减法运算和赋值语句，函数调用的开销将占到整个函数运行时间的三分之二以上。
- 对递归的不正确使用引起大量重复的计算。对于大多数效率低下的递归代码，这一原因尤为突出。

下面仍以 fibonacci 数列的递归函数实现为例，来进一步分析。

对于例 5.10 中的递归函数实现，结合图 5.9 所示，我们可以发现，在这个递归过程中，任何大于 1 的参数值，都要通过递归调用进行降解，直至递归结束，函数在计算过程中没有保留任何已被计算过的结果，因而出现了大量的重复计算，这种重复计算会随着 `n` 值的增加而成倍地增长。

对例 5.10 中的递归函数改进如下：

```
#include <stdio.h>
int Fib(int n,int a,int b);

int main()
{
    printf("Fib(5)=%d\n",Fib(5,1,1));
    return 0;
}
```



```

}

int Fib(int n, int a, int b)
{
    if(n<=2)
        return b;
    return Fib(n-1,b,a+b);
}

```

在下面的例题中我们会看到，使用递归的方法要明显强于其他非递归的方法。

【例 5.11】 设计递归程序，打印实现 Hanoi 游戏的步骤。

分析：

Hanoi 游戏是一个来自印度神庙的古老游戏。参见图 5.10，在 3 个柱子（记作 A、B、C）上的 Hanoi 游戏是这样的：

①游戏的初始状态是：A 柱上有 n 个从小到大叠在一起的圆盘，B、C 柱为空，状态图见图 5.10(a)。

②游戏的完成状态是：A、B 柱为空，C 柱上有 n 个从小到大叠在一起的圆盘，状态图见图 5.10(d)。

③游戏中，每个步骤只能移动一个圆盘，且每次移动之后，要保证每个柱上的圆盘是从小到叠在一起的。

Hanoi 游戏的标准玩法策略：若 $n=1$ ，则直接将 A 柱的 1 个圆盘，移至 C 柱，游戏完成；否则，对游戏问题分解为以下 3 个步骤：

步骤①：借助于 C 柱，将 A 柱的 $n-1$ 个圆盘，移至 B 柱，状态图见图 5.10(b)；

步骤②：将 A 柱上的唯一圆盘，移至 C 柱，状态图见图 5.10(c)；

步骤③：借助于 A 柱，将 B 柱的 $n-1$ 个圆盘，移至 C 柱，游戏完成。

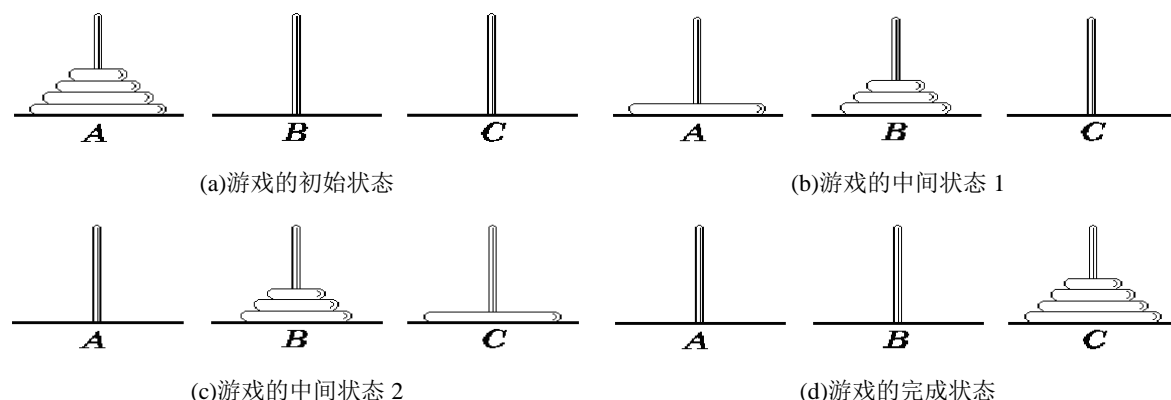


图 5.10 Hanoi 游戏

显然，以上游戏策略是一个递归策略，步骤①③只是将问题规模缩小，只有步骤②是一个简单操作。为此，设计函数 Move 输出步骤②。

为实现 Hanoi 游戏的分解策略，设计函数 `Hanoi(int n,char source,char help,char target)`，用 3 个字符参数表示 3 个柱子。source 代表初始状态时 n 个圆盘所在的柱子，target 代表完成状态时 n 个圆盘所在的柱子，help 代表起过渡作用的柱子。

在函数 main 中，给出代表 3 个柱子的字符参数，指定 n 值，调用函数 Hanoi，就可以输出游戏的详细步骤。

参考程序:

```
#include <stdio.h>

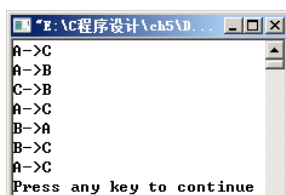
void Move(char source,char target);
void Hanoi(int n,char source,char help,char target);

int main()
{
    Hanoi(3,'A','B','C');
    return 0;
}

void Move(char source,char target)
{
    printf("%c->%c\n",source, target);
}

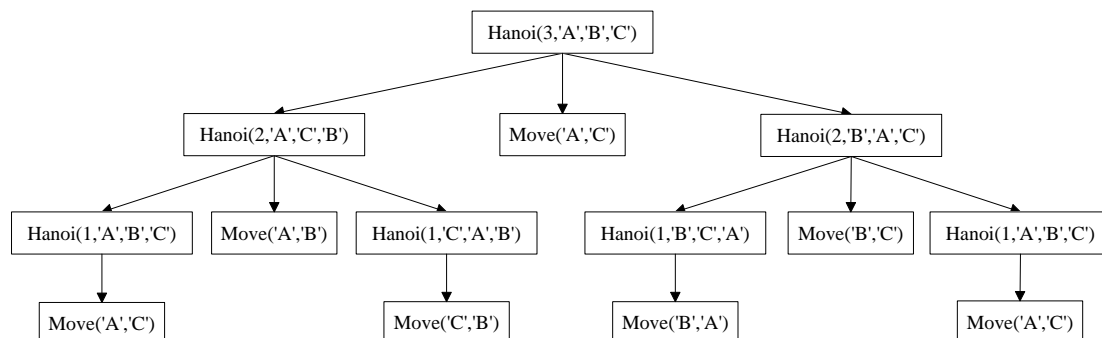
void Hanoi(int n,char source,char help,char target)
{
    if(n==1) Move(source,target);
    else
    {
        Hanoi(n-1,source,target,help);    // 步骤①
        Move(source,target);               // 步骤②
        Hanoi(n-1,help,source,target);     // 步骤③
    }
}
```

程序的运行结果截图如下:



程序分析:

由于函数 Hanoi 的参数个数较多,递归调用时,实参的变化也较多,为便于理解函数的执行过程,图 5.11 给出了函数 Hanoi 的递归调用图。希望读者结合“单步跟踪”、“观察变量”等程序调试手段,对图 5.11 有一个具体的、真切的认识。



本章小结

本章的主要内容总结如下：

(1) 将函数视作模块，模块化程序设计的基本原则有：自顶向下对问题进行分解；每个函数完成一项特定的功能；不断地改进每个函数，提高其复用性。

(2) 函数的定义体、声明语句、函数的调用的语法规则，及相关的良好程序风格。

(3) 系统库函数的使用方法。

(4) 局部变量和全局变量的定义、作用域，以及它们之间的重名处理规则。

(5) 动态变量、静态变量的概念，以及不同的生命期对函数执行功能的影响。

(6) 嵌套的函数调用，展现自顶向下的模块化程序设计方法，是解决复杂问题的必由之路。

(7) 函数的递归调用。递归体现了问题分解策略，对于哪些大问题可以转换为类型相同，但规模较小的问题，可以采用递归调用。递归调用过程中，系统为每一次的函数调用，创建一个独立的局部变量集合，得到一个全新的执行过程。

习题 5

一、选择题。

1. 已知某函数定义：void f(){ } 正确的函数调用形式是 ()。

A. f; B. f(); C. f(void); D. f(1);

2. 以下函数定义，其中正确的是 ()。

A. int fun(int a,b){} B. int fun(int a[][]){}
C. int fun(void){} D. int fun(static int a,int b){}

3. 以下定义全局变量的语句正确的是 ()。

A. auto int i=1; B. char for=1;
C. float a=1, b=0.5, c=a+b; D. static char ch;

二、读程序，写结果。

1. 以下程序的输出结果是_____。

```
#include <stdio.h>
int f(int a) { return a%2; }
int main()
{
    int s[8]={ 1,3,5,2,4,6},d=0,i;
    for(i=0; f(s[i]); i++)
        d+=s[i];
    printf("%d\n",d);
    return 0;
}
```

```
}
```

2. 以下嵌套调用程序的输出结果是_____。

```
#include <stdio.h>
int f1(int x,int y){ return x>y?x:y; }
int f2(int x,int y){ return x>y?y:x; }
int main()
{
    int a=4,b=3,c=5,d=2,e,f,g;
    e=f2(f1(a,b),f1(c,d));
    f=f1(f2(a,b),f2(c,d));
    g=a+b+c+d-e-f;
    printf("%d,%d,%d\n",e,f,g);
    return 0;
}
```

3. 以下程序的输出结果是_____。注意静态变量的用法。

```
#include <stdio.h>
int fun(int a)
{
    int b=0;
    static int c=3;
    b++; c++;
    return a+b+c;
}
int main()
{
    int i;
    for(i=0; i<3; i++)
        printf("%d %d\n",i,fun(5));
    return 0;
}
```

4. 以下递归程序的输出结果是_____。注意与例 5.9 的区别。

```
#include <stdio.h>
void printd(int n );
int main()
{
    printd(1234);
    return 0;
}
void printd(int n )
{
    if(n==0) return;
    printf("%d,",n%10);
    printd(n/10);
}
```

5. 以下递归程序的输出结果是_____。

```
#include <stdio.h>
void fun(int a[],int i,int j)
{
    if(i<j)
    {
        int t=a[i]; a[i]=a[j];a[j]=t;
        i++; j--;
        fun(a,i,j);
    }
}
int main()
{
    int x[4]={2,6,1,8},i;
    fun(x,0,3);
    for(i=0; i<4; i++)
        printf("%d,",x[i]);
    return 0;
}
```

6. 以下递归程序的输出结果是_____。

```
#include <stdio.h>
int f(int n);
int main()
{
    int i;
    for(i=1; i<=5; i++)
        printf("f(%d)=%d\n",i,f(i));
    return 0;
}
int f(int n)
{
    int s=0,i;
    if(n==0||n==1) return 1;
    for(i=n-1; i>=0; i--)
        s += f(i) * f(n-1-i);
    return s;
}
```