第5章 函 数

学习导读

在编写复杂程序的时候，往往会有成百上千条语句以实现较多的功能。如果所有的代码都集中在一个主函数中（main 函数），会面临诸多问题。首先，代码可读性差。如果复杂程序的语句全挤在主函数里，那么程序就像一本没有目录和章节的书，很难让人快速理解程序的逻辑和功能。其次，编写效率低。由于代码量大，常需多人编写，但主函数代码混乱，后续编写者得花大量时间理解前面的代码，才能保证逻辑正确，这大大拖慢了编写速度。最后，调试维护难。程序出错时，在冗长的主函数里找问题，如同大海捞针。而且修改代码时，牵一发而动全身，容易引发新的错误，增加了调试和维护的难度。

为了解决以上问题,C 语言中通过函数实现程序的模块化。以建造一座大型商场为例，不同的施工团队就像 C 语言函数。结构施工队搭框架，水电安装队铺管道线路，各团队功能独立。以往项目施工方案可复用，如同函数多次调用。项目经理统筹，团队内再分工，类似函数嵌套调用。各团队施工细节保密，体现模块化与封装性，保障项目高效推进。本章介绍 C 语言程序的函数设计。

内容导学

（1）函数的基本概念。
（2）函数调用和参数传递。
（3）函数嵌套调用和递归调用的编写方法。
（4）局部变量与全局变量的概念
（5）静态变量与动态变量的概念。
（6）预处理命令的使用方法。

教学目标

知识目标：

（1）了解函数设计思想。
（2）掌握函数定义，明确函数名、参数、返回值等构成要素。
（3）掌握嵌套与递归调用方法。
（4）了解局部变量、全局变量、动态存储变量、静态存储变量的特性。

能力目标：

（1）能够根据具体问题需求，合理设计函数，确定函数的功能、参数和返回值。

（2）掌握函数调用的规则和方法。

育人目标

“能用众力，则无敌于天下矣；能用众智，则无畏于圣人矣”出自《三国志·吴书·孙权传》，意思是能运用众人力量，便天下无敌，能运用众人智慧，便无惧圣人，深刻地诠释了团队协作的重要性。

个体的力量是有限的，无论个人多么强大，都有其能力边界。但当众人的力量汇聚在一起时，就能够产生巨大的合力。这种合力不是简单的个体力量相加，而是会产生乘数效应，形成一股远超个体总和的强大力量。同样，C 语言函数设计强调模块化，将复杂程序分解为多个独立功能函数，这就如同团队项目分工，每个成员负责独立模块，共同完成复杂任务。在团队中，明确各自职责，每个成员专注自身模块，如同函数专注单一功能，最终整合实现项目整体目标。

5.1　函数的基本概念

在 C 语言中，函数是完成特定任务的独立代码单元，如同程序中的“小工具”。它能将程序模块化，将复杂任务分解为多个小功能，增强代码可读性与可维护性。还能提高代码复用率，避免重复编写相同代码。同时，它利于团队协作开发，不同成员负责不同函数，提高开发效率。

本章主要介绍函数的定义、调用、函数中参数的传递、函数的嵌套调用、递归调用，通过学习了解函数在程序编写中带来的优势。

5.1.1　函数的引例

【例 5.1】从键盘输入 x 的值，计算 x 的绝对值。

C 语言中没有绝对值运算符，所以不能用绝对值的形式计算|x|的值。但 C 语言提供的库函数中的 fabs(x)的功能是计算 x 的绝对值，因而我们可以调用库函数 fabs()来求解。

```
#include<stdio.h>
#include<math.h>     /*fabs()函数来自库函数中的数学函数，必须加此行*/
int main()
{   double  x=0,z=0;
    printf("Input data:");
    scanf("%lf",&x);
    z=fabs(x);
    printf("%lf 的绝对值是%lf\n", x,z);
    return 0;
}
```

C 语言库函数中的 fabs()函数大幅降低了计算 x 绝对值的难度，我们不需要设计算法来实现该功能，现有的函数已经将该功能模块化，供我们直接使用。当然，我们也可以自己定义一个函数来实现此功能。

```
#include<stdio.h>
double myabs(double x)
{   if(x>=0)
       return x;
    else
       return -x;
}
int main()
{   double  x=0,z=0;
    printf("Input data:");
    scanf("%lf",&x);
    z=myabs(x);
    printf("%lf 的绝对值是%lf\n", x,z);
    return 0;
 }
```

无论是 C 语言中的库函数还是自定义函数，都属于函数的范畴。以上的引例让我们看到函数的使用优化了主函数中的程序代码，使其运行效率更高、可读性更强。

5.1.2 函数的定义形式

函数的一般定义形式如下：

```
[返回值类型] 函数名（[类型名 形式参数 1，类型名 形式参数 2，……]）
{
      定义部分；
      语句部分；
}
```

在定义函数的时候，我们应确定以下几部分：

- 返回值类型：它指定了函数执行完毕后返回结果的数据类型。这个类型可以是 C 语言中的任意基本数据类型（如 int、float、double、char 等），也可以使用 void 表示函数不返回任何值。
- 函数名：函数名是用来标识函数的，遵循 C 语言标识符的命名规则，即由字母、数字和下画线组成，且第一个字符必须是字母或下画线。函数名应该具有一定的描述性，以便清晰地表达函数的功能。
- 函数的形式参数列表：形式参数列表指定了函数在调用时需要接收的输入值。可以包含一个或多个参数，每个参数由数据类型和参数名组成，多个参数之间用逗号分隔；也可以为空，表示函数不需要任何输入。

5.1.3　函数的定义方法

根据函数定义时不同的返回值类型以及参数数量，可以将函数的定义分为无参函数、有参函数、有返回值函数、无返回值函数。

1. 定义无参函数

【例 5.2】 定义无参函数 printStars，功能为打印 10 个“*”。

```
1   void printStars()
2   {printf("**********\n");
3   }
```

该函数并没有返回值，因此返回值类型为 void。又因该函数是无参函数，所以函数名后面的括号内可省略参数或者写 void 表示“空”。

2. 定义有参函数

【例 5.3】 定义有参函数 printNStars，使其能够按照要求打印 n 个“*”。

```
1   void printNStars(int n)
2   {   int i=0;
3       for(i=1;i<=n;i++)
4           printf("*");
5       printf("\n");
6   }
```

printNStars()函数在 printStars()函数的基础上多了一个整型参数 n，在调用函数时根据传递给形参 n 的数值，可打印相应数量的“*”。

3. 定义无返回值函数

无返回值的函数是指函数执行完特定任务后不会向调用者返回一个具体的值，其返回值类型为 void。printStars()函数既是无参函数，也是无返回值的函数。

4. 定义有返回值函数

【例 5.4】 定义有返回值的函数 myfac，计算 n!（n>0）。

```
1   int myfac(int n)
2   {
3       int i=0, y=1;
4       for(i=1;i<=n;i++)
5           y=y*i;
6       return y;
7   }
```

程序说明：

（1）第 1 行 int myfac(int n) 是函数首部。int 表明函数返回值为整型，myfac 是函数名，(int n) 表示函数接收一个整型的参数 n，此函数用于计算 n 的阶乘。

（2）第 3～5 行是函数体中的循环部分。第 3 行定义并初始化两个整型变量 i 为 0、y 为 1；第 4、第 5 行通过 for 循环，让 i 从 1 递增到 n，每次循环将 y 乘以 i，以此实现阶乘的累乘计算。

（3）第 6 行“return y;”是返回语句。将循环计算得到的阶乘结果 y 返回给调用该函数的地方，供后续程序使用。

【例 5.5】定义函数 countdigit，它的功能是统计某正整数 number 中数字 digit 的个数。

例题解析：先确定该函数的返回值类型以及参数列表。根据题意，该函数有返回值，并且返回值是整数，因此该函数的返回值类型为 int。该函数接收两个参数 number 和 digit，均为整数，由此我们可以写出 countdigit 函数的部首：int countdigit（int number，int digit）。

```
1   int countdigit(int number, int digit) /*定义 countdigit 函数*/
2   {   int count=0;
3       while (number>0) {
4           if(number%10==digit)    /*判断 number 的个位数与 digit 是否相等*/
5               count++;
6           number=number/10; }     /*去除 number 的个位数*/
7       return count;
8   }
```

C 语言中，函数的定义恰似团队中的个人工作的分配。函数有明确功能，个人有清晰职责，为整体目标发力。函数可复用来提升效率，个人经验也能帮助团队进步。若函数出错影响程序，个人工作失误也危及项目。每个成员做好本职工作，发挥个人优势，团队才能高效运转，实现最终目标。

5.2 函数的调用

在编程的世界里，函数如同一个个精巧的工具，各自承担着特定的任务。而函数调用则是将这些工具运用起来，让程序真正运转的关键环节。本节我们将深入探讨函数调用的相关知识，涵盖函数调用形式（即如何正确地发起对函数的调用）、函数调用过程（了解程序在调用函数时究竟发生了什么）、函数返回值（明白如何获取并处理函数执行后的结果），以及函数原型说明（掌握其在函数调用中的重要作用）。

5.2.1 函数调用形式

函数调用的一般形式为：

函数名（实际参数 1，实际参数 2，……）

其中函数名为被调用函数的名称，在调用时该函数必须存在；实参列表是在函数调用时传递给函数的具体数据。实参列表是由一个或多个实参组成的序列，这些实参将为函数执行提供必要的输入信息。实参的数量必须与函数定义时的形式参数数量一致、类型一致。

不同的函数在调用时会有形式上的差异，下面根据 5.1.3 节中函数定义的分类一一阐

述函数的调用形式。

1. 无参函数的调用

调用无参函数时，实参列表可为空，但是括号不可省略。如例 5.2 中的函数，可将函数单独作为一个语句，“printStars();”即可完成函数功能。

2. 有参函数的调用

有参函数括号内包含实际参数列表，实参之间用逗号分隔，即“**函数名(实参 1, 实参 2, …)**;”。以例 5.3 中的 printNStars()函数为例，如果需要完成打印 10 个“*”的功能，调用函数时需要在括号中填入实参“10”，即“printNStars(10);”。

3. 无返回值函数的调用

可参照 printNStars()函数的调用。

4. 有返回值函数的调用

有返回值的函数可赋值给一个变量，也可以作为表达式的一部分参与各种运算，例如算术运算、逻辑运算等。

【例 5.6】 调用例 5.4 中的 myfac 函数，计算 n 的绝对值和 n!(n>0)。

```
#include<stdio.h>
#include<math.h>        /*使用库函数 fabs 需要使用指令*/
int myfac(int n)        /*定义 myfac 函数，功能为计算阶乘*/
{   int i=0, y=1;
    for(i=1;i<=n;i++)
    y=y*i;
    return y;
}
int main()
{
    int  n=0,z=0;  double  y=0;
    scanf("%d",&n);
    y = fabs(n);        /*调用库函数 fabs*/
    z = myfac(n);       /*调用自编函数 myfac*/
    printf("n=%d,y=%lf,z= %d\n",n,y,z);
    return 0;
}
```

程序说明：

（1）在 main()函数中，读取用户输入的整数 n，调用 fabs 函数计算 n 的绝对值存入 y，调用 myfac 函数计算 n 的阶乘存入 z，最后输出 n、y、z 的值。

（2）myfac 函数用于计算一个整数的阶乘。它接收一个整数参数 n，通过 for 循环从 1 到 n 进行累乘操作，最终得到 n 的阶乘结果并返回。

（3）调用 myfac 函数时，只需将需要计算阶乘的整数作为参数传递给它。这里将变量 n 的值作为参数传递给 myfac 函数，函数计算 n 的阶乘后将结果返回，赋值给变量 z。

5.2.2 函数调用的过程

在函数调用过程中，实参和形参的传递是一个重要环节，它涉及数据如何从调用函数传递到被调用函数。

实参是在函数调用时传递给函数的具体数据，它可以是常量、变量、合法表达式等。实参位于调用函数中，用于提供函数执行所需的输入信息。

形参是函数定义时声明的参数，它是函数接收外部数据的占位符。形参位于被调用函数中，在函数调用时接收实参传递过来的值。

实参与形参可以同名，但占不同的存储单元。打个比方，不同的楼可以有相同编号的房间，这些房间各自占用不同的空间，它们之间互不相干。函数比喻成楼，各函数中的变量（包括形参）想象成各楼中的房间，变量名如同房间号。不同函数中的同名变量占用不同的存储空间，因此，实参与对应形参可以同名。

【例 5.7】 输入 3 个整数，要求用函数找到最小值。

```
#include<stdio.h>
int findMin(int x, int y, int z)
{  int min = x;
   if (y < min)  min = y;
   if (z < min)  min = z;
   return min;
}
int main()
{   int a, b, c, minimum;
    printf("请输入 3 个整数，以空格分隔：");
    scanf("%d %d %d", &a, &b, &c);
    minimum = findMin(a, b, c);     /*调用 findMin 函数，找出最小值*/
    printf("这 3 个整数中的最小值是：%d\n", minimum);
    return 0;
 }
```

以上函数调用的示例可分为 4 步，注意所有的程序入口都是主函数。

（1）实参的输入。

在主函数中，首先定义了 3 个整型变量 a、b、c，这 3 个变量将作为调用 findMin() 函数时的实参。通过 scanf()函数读取用户输入的内容，并将输入的 3 个整数分别存储到变量 a、b、c 中。

（2）调用函数时参数传递。

执行“minimum = findMin(a, b, c);”语句时，发生函数的调用，此时实参 a、b、c 的数值会传递给 findMin 函数中的形参 x、y、z。

（3）函数内部执行。

在 findMin 函数内部，形参 x、y、z 接收到数值后进行依次比较，并将最小值存储在 min 中，作为函数的返回值。

（4）返回结果。

函数调用完毕后，回到主函数，函数返回值赋值给变量 minimum，输出实参 a、b、c 中的最小值。

5.2.3　函数的原型声明

在之前涉及函数定义与调用的程序里，我们不难留意到，所有自编函数都被放置在主函数之前。这是由于 C 语言有明确的规则：函数必须先完成定义，之后才能被使用。但从实际的函数编写和阅读角度来看，主函数作为程序的入口，往往是我们关注的重点。将自编函数都置于主函数之前，会干扰我们对程序整体逻辑的把握，在理解程序如何启动和运行时造成不便，也不符合我们从程序入口开始梳理流程的习惯。函数原型声明就很好地解决了这一问题。

函数的原型声明能让函数的定义和调用顺序更加灵活，我们不必再把函数定义都前置。编译器依据函数原型声明就能对函数调用进行检查，判断参数类型、数量以及返回值的使用是否正确，保障程序编译顺利进行。同时，函数原型声明提高了代码的可读性与可维护性，方便开发者快速了解函数信息，还支持模块化编程，实现代码的复用。

函数原型声明的一般形式如下：

[返回值类型] 函数名（[类型名 形式参数 1，类型名 形式参数 2，……]）;

可以看到函数原型声明相比函数定义中的函数首部多了一个分号。如果我们在主函数前已经对自编函数进行了声明，那么自编函数的定义就可以放在主函数之后。

【例 5.8】 编写程序，通过调用函数，计算两个数 a、b 的平方和（a^2+b^2）与平均数。

```
#include<stdio.h>
double square_sum(double a, double b);  /*计算平方和函数原型声明*/
double average(double a, double b);     /*计算平均值函数原型声明*/
int main()
{   double a, b, sum_of_squares,avg;
    printf("请输入两个数，用空格分隔：");
    scanf(%d%d,&a,&b);
    sum_of_squares = square_sum(a, b);  /*调用函数计算平方和*/
    avg = average(a, b);                /*调用函数计算平均数*/
    printf("这两个数的平方和是：%.2f\n", sum_of_squares);
    printf("这两个数的平均数是：%.2f\n", avg);
    return 0;
}
double square_sum(double a, double b)
{ return  a * a + b * b; }
double average(double a, double b)
{ return  (a + b) / 2; }
```

程序说明：

（1）square_sum()函数与 average()函数可以分别实现两数的平方和与平均数。

（2）之前的自编函数都定义在主函数的前面，因为函数必须存在，我们才可以使用它。这个程序在主函数前做了函数的原型声明，因此可以定义在主函数之后。

当程序中需要调用多个函数的时候，函数原型声明可以逐一列在主函数之前。这样，在阅读程序时可以提前知晓函数信息，函数原型声明就像函数的“说明书”，阅读代码的人在查看函数调用时，无须在代码中四处查找函数的具体定义，就能快速了解该函数的基

本信息。

5.2.4 函数的应用

在深入学习函数的定义和调用之后，我们会发现，对于很多原本复杂的程序，如今可以采用一种更为高效、清晰的方式来编写。

以一个需要完成多种计算任务的程序为例，若不使用函数，所有的计算逻辑都会堆积在主函数中，这会使主函数变得冗长且复杂，代码的可读性和可维护性也会大打折扣。但当我们把不同的计算任务封装成独立的函数后，主函数就只需要负责调用这些函数，就如同按照流程依次使用不同的工具一样。

下面用一个例题对比不使用函数和使用函数在编写程序上的不同之处。

【例 5.9】 计算 $s=1+\frac{1}{2\times3}+\frac{1}{3\times4\times5}+\cdots+\frac{1}{n\times(n+1)\times\cdots\times(2n-1)}$（$n$ 为正整数）。

例题解析：

此数列是由多个分式项相加组成，第 n 项的分母是从 n 开始连续 n 个自然数的乘积。为了计算总和，可以采用循环的方式依次计算每一项的值，再将每一项的值累加起来。如果不使用函数，结合之前学习过的内容，这是一个嵌套循环的问题。外层循环用于控制求和项数，从 1 到 n 遍历，确定当前要计算的是第几项。对于每一项，使用内层循环计算其分母。内层循环从当前项对应的起始数字开始，累乘连续的自然数，直到达到 $2n-1$，得到该项分母的值。

```
#include<stdio.h>
int main()
{   int i,j,n;
    double s = 0, term = 1;
    do
    {printf("请输入 n 的值: ");        /*提示用户输入 n 的值*/
     scanf("%d", &n);}while(n<=0);
     for (int i = 1; i <= n; i++)    /*外层循环控制项数*/
       { for (int j = i; j <= 2 * i - 1; j++) /*内层循环计算每一项的分母*/
           {term *= j;
           }
        s += 1.0 / term;             /*将每一项的倒数累加到 s 中*/
        }
     printf("s 的值为: %.10lf\n", s);
     return 0;
}
```

此问题也可通过定义函数与调用函数解决。创建 f 函数，其接收一个整数参数 n 代表当前项序号。在函数内部，利用循环从 i 累乘到 2i−1 得到该项分母，再取倒数得到该项的值并返回。此函数将计算单项的逻辑封装，提高了代码复用性。

```
#include<stdio.h>
double f(int n)
{ int i;
```

```
    double s=1;
for(i=n;i<=2*n-1;i++)
    s*=i;
return s;
}
int main()
{   int i,n;
    double s = 0;
    do
    { printf("请输入 n 的值: ");       /*提示用户输入 n 的值*/
      scanf("%d", &n);}while(n<=0);
      for (i = 1; i <= n; i++)         /*循环控制项数*/
         s += 1.0 / f(i);              /*将每一项的倒数累加到 s 中*/
      printf("s 的值为: %.10lf\n", s);
      return 0;
}
```

不使用函数和使用函数的两个程序都能计算 $1+\frac{1}{2\times3}+\frac{1}{3\times4\times5}+\cdots+\frac{1}{n\times(n+1)\times\cdots\times(2n-1)}$。不使用函数的程序将所有逻辑写在主函数中，代码集中但逻辑较混乱，若要修改单项计算逻辑，要在主函数中直接改动。使用函数的程序把计算单项的逻辑封装在 f()函数中，主函数更简洁，提高了代码复用性和可维护性，修改单项计算只需要调整对应的函数。

函数就像是团队中的成员，各自有着明确的分工。函数的调用顺序恰似项目的执行流程，需要合理安排、有序推进。函数原型声明则是成员间沟通的规范，确保信息能够准确无误地传递。就像一个优秀的团队，只有成员分工协作、有序沟通，才能高效地完成任务。同样，一个功能完善的程序，也离不开各个函数之间的默契配合。

5.3　函数的嵌套调用

在 C 语言中，函数的调用场景十分灵活。函数不仅能在主函数中被调用，在我们自己编写的函数中同样可以被调用。当调用一个函数时，若这个函数在执行过程中又调用了其他函数，这种调用方式就被称作函数的嵌套调用。

如图 5.1 所示，程序在 main()函数运行，首先调用了 a 函数，在运行 a 函数时调用了 b

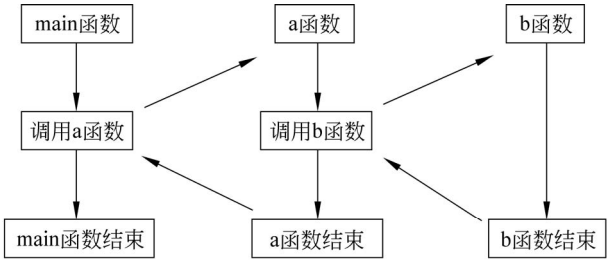

图 5.1　函数嵌套调用的流程图