第5章 像搭积木一样搭建程序——函数

到目前为止，我们已经掌握了利用C语言编写程序的基本知识，即使不再学习后续章节的知识，我们也可以编写出解决常见问题的计算机程序。既然如此，为什么还要学习函数以及后面的数组、结构体这些知识呢？当你阅读完相关章节以后，相信你就会找到一个共同的答案——降低程序代码编写工作的复杂性，提高程序代码的开发效率。

5.1 复杂程序的开发问题

在前面的章节中，我们基本上都是在main函数中编写程序代码，代码的行数很少，程序的功能也相对简单。即便如此，当程序代码中存在错误时，往往也要花费很长时间来调试程序，找出其中的错误问题并纠正。当要编写一个复杂的大型程序时，例如开发操作系统，它的程序代码可能达到数千万行，如果将所有的程序代码都写在main函数中，那么程序调试将变得非常困难。编写和维护这样的程序将会成为程序员们的一场噩梦。因此，需要寻找一些科学的方法来指导我们高效地组织程序代码的开发工作。

5.1.1 像工业化生产一样开发程序

人类社会的进步与社会分工密不可分。社会分工是人类社会经济得以发展的基础。对人类来说，没有社会分工就没有交换，市场经济也就无从谈起。人类社会分工的优势是让擅长的人做自己擅长的事情，使平均社会劳动时间尽可能缩短，从而显著提高社会生产效率。

当今的社会是高度发达的工业化社会，工业化的生产更离不开社会分工。例如，计算机的生产就采用了社会化分工。如果要生产一台计算机，它所有的零部件并不是全部由一家工厂生产。而是先由不同的生产商制造出不同的零部件，然后由一家生产商对零部件进行选配，从而组装成性能不同的计算机，以满足不同用户对计算机的差异化使用需求。在计算机的主要配件中，CPU的生产商就有Intel(英特尔)、IBM(国际商业机器公司)、龙芯等，显卡的生产商有NVIDIA(英伟达公司)、AMD(美国超威半导体公司)等，硬盘的生产商有Seagate(希捷)、WestDigital(西部数据)等。每个公司都专注于某种特定计算机配件的研制和生产，这种配件的生产效率将会极大地提升，计算机的生产效率也会极大地提高。

与硬件产品类似，计算机程序作为一种软件产品，也可以通过社会化生产分工的方式来研发。例如，计算机操作系统Windows操作系统、Mac OS系统、鸿蒙操作系统等，数据库管理系统有Oracle、DB2、MS SQLServer和Sybase、MySQL等产品。这里提到的每一个软件都是一个大型的程序。它们都是由成千上万的程序员通过分工合作的方式共同编写完成的。因此，我们需要找到一种能够对程序代码编写进行分工合作的方法。

5.1.2 将程序代码做成积木模块的方法

微课 5.1 模块化思想与函数

在生活中，孩子们喜爱搭建积木。例如，乐高积木有很多种形状，它的每个积木块一面有凸粒，另一面有可嵌入凸粒的凹槽，通过凹凸部位的连接可以实现积木块之间的拼装。孩子们利用这些积木模块可以快速搭建各式各样的城堡、飞机、跑车，安装上各种电子器件甚至可以让它们动起来。这些积木模块的类型是有限的，很容易实现批量化的生产，却又可以快速地拼搭出各种不同的模型产品。插上想象的翅膀，你可以用这些积木模块搭建出你想创造的任意实物模型。

这种模块化的生产思想被广泛地应用到工业化生产中。例如，计算机的制造过程就是模块化的生产过程。计算机的配件是模块化的配件。这些配件一般都有卡槽，通过连接线可以很容易地将它们连接在一起。不同公司生产的计算机配件的功能和性能可能是不同的，但是这些卡槽和连接线的接口一定是相同的。这样计算机的集成商通过选择性价比不同的计算机配件就可以快速地组装一台性价比不同的计算机产品。

我们也可以借鉴模块化的思想来开发程序软件。将一段能够实现特定功能的程序代码封装成一个程序模块。每个程序模块也设置一种类似积木模块的“凹凸”接口，通过“凹凸”接口可以实现对程序模块的拼接。我们可以根据软件的功能需求来选择具有不同功能的程序模块，然后将它们拼装组合成程序，从而实现程序软件的快速开发。

1. 模块化程序设计

在程序开发过程中，对程序的功能进行模块化设计是一种重要的程序设计思想。**模块化程序设计方法是指在进行程序设计的时候将一个程序按照功能划分为若干个小的功能，每个小的功能分别由一个程序模块来实现，通过模块间的相互协作来实现整个程序功能的设计方法**。一般来说，一个复杂问题都可以逐级、逐层分解为若干个简单的问题，通过逐个解决简单问题实现对复杂问题的解决。通过模块化程序设计可以把程序要解决的复杂问题的总目标分解为若干个解决简单问题的子目标，还可以进一步把子目标再分解为更具体的小目标。将实现每一个小目标的程序代码封装为功能模块，最后对程序模块的功能逐级、逐层进行组合，从而实现构建复杂程序功能的大目标。

利用模块化思想进行程序设计，程序代码具有以下特点。

1）封装性

封装性是指模块内部的程序代码是模块私有的，不能被其他模块所访问。每个模块内部的代码是封闭的，不会对其他模块开放，其他模块无法访问该模块中的变量。每个程序模块只能通过输入输出接口与其他模块进行数据交互。这种封装性会降低程序代码的灵活性，但是它也降低了程序代码的耦合性，提高了程序模块功能的独立性，使程序

的可读性和可维护性得到提高。

假如一个模块的代码中存在错误，那么这种错误只会影响到模块自身的功能实现，并不会影响到其他模块的功能实现。在测试模块功能的时候，只需要向模块中输入数据，然后检查它的输出数据是否正确就可以检查模块的功能是否正确。如果模块的功能不正确，那么模块的代码中有可能存在错误，程序代码的错误检查与故障诊断的范围会极大地缩小。

2）重用性

重用性是指程序模块的代码能够在不做修改的情况下被其他程序所使用。模块的重用性可以提高程序的开发与维护效率。例如，如果发现程序的某个模块的性能不足，那么可以选用具有相同功能但性能更优的其他程序模块进行替换。在模块的替换过程中，不需要或者只需要对替换模块的代码进行较小的修改就可以实现模块间的替换。这种模块替换并不会对程序的整体功能造成较大的影响，可极大地提升程序代码的维护效率。

3）组装性

组装性是指通过对程序模块进行选择、组合可以构建不同功能的程序。由于程序模块具有封装性和重用性，所以程序模块也就具备了组装性。这样可以系列化地开发不同功能和性能的程序模块，通过选配不同的程序模块来组装性价比不同的程序软件。程序员可以通过分工协作的方式来开发功能更加复杂、更加强大的程序软件。每个程序员都可以专注于特定功能的程序模块的开发工作，每个程序模块的功能和性能都会有较大的提升，从而使程序软件的整体功能与性能得到较大的提升。

C 程序语言中提供了 function 机制来实现程序代码的模块化开发工作。function 翻译成中文有“功能”或者“函数”的意思。在计算机语言中习惯将它称为函数。函数就像一个能够处理数据的“黑盒子”，将数据送进去就能够得到一个想要的结果。函数内部是如何工作的，外部的程序是不需要知道的。将一个个这样的“黑盒子”有机地组合起来就可以构建一个具有特定功能的程序。

为什么函数可以使程序模块具有封装性、重用性和组合性呢？下面让我们来了解函数的结构。

2. 函数的结构

积木的每个模块都有一个名称来区分彼此。程序中的每个函数都代表了一个特定的功能，因此它也需要一个名称来表示这个功能。另外，积木模块通过它的“凹凸”结构来连接其他模块。在函数中，这种“凹凸”结构是函数的输入数据接口和输出数据接口。

例如，函数 A 要使用函数 B 的功能，它需要与函数 B 的输入和输出接口进行连接。函数 A 需要通过函数 B 的输入接口输入待处理的数据，当函数 B 处理完输入数据以后，函数 A 又需要通过函数 B 的输出接口接收函数 B 的输出数据。这样函数 A 就使用了函数 B 的功能，也实现了它与函数 B 的一次交互。函数 A 与函数 B 的接口关系参见图 5.1。

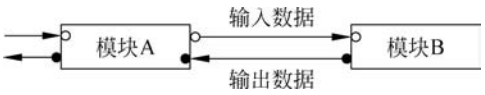

图 5.1　函数 A 与函数 B 的接口连接关系

一般情况下，函数都需要接收从外部输入的数据并对数据进行处理，然后输出结果数据，从而实现它的功能。因此，一个函数的结构包括函数的名称、输入数据接口、函数体和输出数据接口 4 个要素，其中函数体是一段能够实现函数功能的程序代码。

1）函数的名称

在 C 语言中，函数的命名规则与变量的命名规则相同。对函数的名称进行命名要尽可能地反映函数的功能。例如，要实现一个求最大值功能的函数，可以用英文单词 maximum 的简写 Max 来为函数命名。这样程序员通过函数的名称就可以大概知道这个函数的功能是求解最大值。

2）输入数据接口

函数的输入数据接口可以接收多个输入数据。输入数据接口的格式如下：

函数名(数据类型 1　变量 1,数据类型 2　变量 2,...,数据类型 n　变量 n)

在函数名后有一个小括号，在小括号里面定义了函数中负责接收输入数据的变量的名称及其数据类型。根据输入数据的数量可以确定输入数据接口中所需要定义的变量的个数。如果没有输入数据，那么输入数据接口中就不需要定义任何变量。此时，虽然小括号里面什么都没有，但是小括号还是需要保留的。

假设 Max 函数的功能是实现比较两个整数的大小并输出其中的较大值，那么 Max 函数的输入数据接口可以定义如下：

```
Max(int a, int b)
```

如果 Max 函数的输入数据是 4 和 5，即 Max(4,5)，此时整数 4 会存储到变量 a 中，整数 5 会存储到变量 b 中。在函数体中判断变量 a、b 大小的程序语句会实现对整数 4 和 5 的大小比较。

如果 Max 函数要实现判断两个浮点型数据的大小，那么需要重新定义 Max 函数的输入数据接口如下：

```
Max(float a, float b)
```

此时，Max 函数才能正确地接收所输入的两个浮点数，如 Max(4.5,5.5)。

在 Max 函数中，变量 a 和变量 b 中存储的数据的类型是预先确定的，但是数据的值是无法事先确定的，因此变量 a 和变量 b 被称为形式上的参数，简称形参。当向 Max 函数中输入具体的数据后，变量 a 和变量 b 中才有了具体的数值，输入的数据是实际上的参数，简称实参。例如，在 Max(4.5,5.5)中，4.5 和 5.5 就是实参。

函数输入数据接口中所定义的变量是形参，其他函数向该函数形参中所输入的具体数据就是实参。输入数据可以是常量、变量、表达式等，也就是说实参可以是常量、变量、

表达式等。形参和实参是函数中非常重要的概念。

3）函数体

实现函数功能的程序代码称为函数体。函数体负责完成对输入数据的处理并输出结果数据，它是一个复合语句。函数体的格式如下：

```
函数名(数据类型 1 变量 1,数据类型 2 变量 2,…,数据类型 n 变量 n)
{
    函数体
}
```

例如，Max 函数的函数体代码如下：

```
1   Max(int a,int b)
2   {
3       int MaxValue;
4       if(a > b) MaxValue = a;
5       else MaxValue = b;
6   }
```

第 2～6 行的代码是 Max 函数的函数体。第 3～5 行是 Max 函数判断变量 a 和 b 较大值的语句。在函数体执行后，变量 a 和变量 b 中较大的数值就会赋值给变量 MaxValue。通过函数体语句，Max 函数能够选择出整型变量 a 和 b 中的较大值并暂时存储到变量 MaxValue 中，这就是 Max 函数的功能。

4）输出数据接口

通过函数的输出数据接口可以输出函数中的数据，但是只能够输出一个数据。这一点与输入接口不同，输入数据接口可以输入多个数据。函数的输出数据接口的格式如下：

```
函数类型 函数名(数据类型 1 变量 1,数据类型 2 变量 2,...,数据类型 n 变量 n)
    {
        函数体
        return (变量/常量/表达式);
    }
```

函数的输出数据接口需要定义输出数据的类型和数据值。输出数据的类型又称为函数类型，函数类型需要在函数名的前面进行定义。函数输出数据的值又称为函数的返回值，通过关键字 return 输出具体的数据值。函数的返回值可以是变量、常量、表达式的值。需要注意的是，这些数值的数据类型一定要与函数的类型相匹配。如果返回值的类型与函数类型不匹配，则会将返回值的类型转换成函数类型。

【例 5.1】 编写 Max 函数实现比较两个整数的大小，并通过返回值输出最大值。

问题分析：Max 函数的输出数据是整型变量 a 或者 b 中的较大值，因此它的函数类型可以定义为 int 类型。在函数体中，定义了 float 类型的变量 MaxValue，用于存储 a、b 中的较大值，通过 return MaxValue 输出函数的值。

程序代码如下：

```
1   int Max(int a,int b)
2   {
3       float MaxValue;
4       if (a>b) MaxValue=a;
5       else MaxValue=b;
6       return MaxValue;
7   }
```

虽然变量 MaxValue 的数据类型是 float，但是函数的数据类型是 int，因此函数在执行 return 语句时会将变量 MaxValue 中的数值转换成 int 类型后再输出。假如 Max 函数的实参分别是 4 和 5，即 Max(4,5)，那么 MaxValue 的值是 5.0。当第 6 行语句执行后，函数 Max 的返回值是 int 类型的整数 5，而不是 float 类型的数值 5.0。

函数可以没有输入数据，但是一般要有输出数据，不然函数就没有意义。有的时候，函数不会通过 return 语句输出数据，它会通过 printf 等输出函数将数据显示到屏幕上，或者输出到文件中，其实这也是函数输出数据的一种方式，但是这种方式不是通过函数的输出接口输出数据。只有 return 语句才能将数据送给输出接口。当函数不通过 return 语句输出数据的时候，函数就不需要定义输出数据的类型，此时函数的类型是 void 类型，表示无返回数据。

【例 5.2】 编写一个函数 Max 实现比较两个整数的大小，并通过 printf 函数输出最大值。

在下面的代码中，Max 函数通过 printf 函数输出数据，它的类型是 void 类型，即空类型，用于表示函数没有返回数据。

程序代码如下：

```
1     void Max(int a,int b)
2     {
3         int MaxValue;
4         if(a>b) MaxValue=a;
5         else MaxValue=b;
6         printf("a,b 中较大的值是 %f\n",MaxValue);
7     }
```

3. 函数的形态

一个完整的函数一般包括函数类型、函数名、形式参数、函数体和返回值 5 个要素，但并不是每个函数都必须包含这 5 个要素。在函数中，除了函数类型、函数名和函数体 3 个要素必须有，其他两个要素可以没有。常见的函数形态可以分为以下几种。

1) 空函数

空函数是最简单的函数。它没有输入数据和输出数据，函数体里面也没有语句。它的形式为：

```
void 函数名()
{  }
```

空函数没有实际的功能，但是它是符合 C 语言语法的合法函数。空函数一般用在程序设计阶段。当我们已经划分好了一些功能模块，但是还没有想好怎么用函数实现它的具体功能时，可以先写成空函数占个位置，等具体要编写这个函数时再去修改和完善它。

2）无形参，无返回值函数

这种形态的函数没有输入数据和 return 返回值，但是它的函数体中有语句。如果一个函数没有输入数据，则不需要定义形参；如果它同时没有返回值，则可以无 return 语句。它的形式是：

```
void 函数名()
{
    语句
}
```

在下面的代码中，Hello 函数通过 printf 语句输出数据，它没有 return 语句。

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

3）无形参，有返回值函数

这种形态的函数没有输入数据，但是它有 return 语句，即函数有返回值。它的形式为：

```
函数类型  函数名()
{
    语句
    return 语句
}
```

例如，

```
int Hello()
{
    printf("Hello!\n");
    return 0;
}
```

4）有形参，无返回值函数

这种形态的函数有输入数据和函数体，但是它没有 return 语句。它的形式为：

```
void  函数名(形式参数)
{
    语句
}
```

例如，【例 5.2】中的函数 Max 没有返回值。

5）有形参，有返回值函数

这种形态的函数所包含的要素最全，例如【例 5.1】中的函数 Max。

4. 如何定义函数

变量在使用前需要先定义，函数在使用前也需要先定义。**函数定义是按照函数的结构编写代码实现函数功能的过程**。函数定义的要素主要包括定义函数的类型、名称、形参和函数体。其中**函数类型、名称和形参组合在一起又被称为函数首部或者函数头**。

1）函数定义

我们可以在函数内部定义变量，也可以在函数外部定义变量，但是**不能在一个函数的内部定义另外一个函数，即函数不能嵌套定义**。函数之间是平行的和互相独立的。

例如，Max 函数可以定义在 main 函数的前面，也可以定义在 main 函数的后面，但是不能在 main 函数中定义 Max 函数。

【例 5.3】 在 main 函数前面定义 Max 函数。

程序代码如下：

```
1   #include<stdio.h>
2   int Max(int a,int b)
3   {
4       int MaxValue;
5       if(a>b) MaxValue=a;
6       else MaxValue=b;
7       return MaxValue;
8   }
9   int main()
10  {
11      int x,y;
12      scanf("%d%d",&x,&y);
13      printf("x,y中较大的值是%d",Max(x,y));     //调用 max 函数
14      return 0;
15  }
```

在这个例子中，分别定义了 Max 函数和 main 函数。在 main 函数中，通过第 13 行语句对 Max 函数的功能进行了使用。**这种在一个函数中使用另一个函数的功能的方式称为函数调用**。此时，main 函数是调用函数又称为主调函数，Max 函数是被调用函数又称为被调函数。通过函数调用，主调函数 main 可以将它所包含的实参变量 x 和 y 中的数据传递给被调函数的形参变量 a 和 b，并执行被调函数 Max 函数体中的语句，对形参变量 a 和 b 中的数据进行处理，通过 retrun 语句返回 MaxValue 的值，最后在 main 函数中通过 printf 函数输出 Max 函数的返回值，从而实现 main 函数对 Max 函数的功能调用。

2）函数声明

如果被调函数在主调函数之后进行定义，那么需要先在主调函数中或主调函数之前对被调函数进行函数声明，然后才能够对被调函数进行调用。**函数声明是对被调函数的类**

型、函数名和形参数据类型的说明，以便编译系统能够在主调函数中正确地识别被调函数。

【例 5.4】 Max 函数定义在 main 函数后面，在 main 函数中对 max 函数进行声明。程序代码如下：

```
1   #include<stdio.h>
2   int main()
3   {
4       int x,y;
5       int Max(int a,int b);                          //对 Max 函数进行声明
6       scanf("%d%d",&x,&y);
7       printf("x,y 中较大的值是%d",Max(x,y));        //对 Max 函数进行调用
8       return 0;
9   }
10  int Max(int a,int b)                               //对 Max 函数进行定义
11  {
12      int MaxValue;
13      if (a>b) MaxValue=a;
14      else MaxValue=b;
15      return MaxValue;
16  }
```

在 main 函数中，通过第 5 行语句对 Max 函数进行了声明。如果去掉 Max 函数语句，程序在编译时会产生警告："warning C4013：'Max'未定义；假设外部返回 int"，参见图 5.2。

图 5.2 函数未声明的例子

警告产生的原因是：当编译系统由前到后逐句编译程序代码时，先遇到了函数调用语句 Max(x,y)，由于还未编译到 Max 函数的定义语句，无法判别 Max(x,y)是什么，因

此产生了“Max 未定义”的警告提示。

函数声明与函数定义是不同的。函数定义是对函数功能的描述与实现，包括函数类型、函数名、形参、函数体和返回值，它是一个完整的函数单位。函数声明只是对函数首部的函数类型、函数名和形参进行说明。如果被调函数在主调函数之前定义，那么编译系统会先编译被调函数，从而获得被调函数的类型、函数名和形参的信息，因此不需要在主调函数中对被调函数再进行声明。如果被调函数在主调函数之后定义，那么编译系统将先编译主调函数中的函数调用语句，如果没有对被调函数进行声明，那么它将无法理解主调函数中的函数调用语句。

编译系统通过函数声明可以检查被调函数是否存在，被调函数的类型、函数名、形参类型与主调函数中函数调用语句的返回值、函数名、实参类型是否一致。如果被调函数不存在或者两者出现不一致的情况，编译系统就会通过错误提示信息提醒程序员对函数定义与函数调用的程序代码进行检查。

对函数进行声明的方式一般包括以下两种：

第一种，直接复制被调函数的函数首部再加上分号，如

```
函数类型 函数名(数据类型 1 变量 1,数据类型 2 变量 2,...,数据类型 n 变量 n);
```

第二种，直接复制函数首部，可以去掉形参的名称再加上分号，如

```
函数类型 函数名(数据类型 1,数据类型 2,...,数据类型 n);
```

由于编译系统只会检查函数调用语句中的实参与函数的形参数据类型是否匹配，以达到检查主调函数输入的数据是否与函数的输入数据接口的数据类型相匹配的目的，此时，函数声明中的形参变量的名称没有任何意义，因此在函数声明的时候可以省略，甚至二者形参的名称不一致也没有关系。

5. 如何使用函数

当函数定义完成以后，该如何使用它们呢？函数的功能是接收输入数据，经过函数体代码处理后，输出结果数据。可以通过函数调用方式执行函数体中的程序代码让它的功能发挥出来。对于无返回值的函数，一般以函数调用语句的方式来使用函数的功能；对于有返回值的函数，一般以函数表达式的方式来使用它的功能。下面具体介绍函数的使用方法。

微课 5.2　函数的定义与使用

对于有返回值函数的使用，可以类比变量的使用方法。在变量中有数据，函数的返回值也是数据。之前我们已经学习了使用变量中数据的方法，那么能否像使用变量那样使用函数呢？首先要弄清函数和变量是否一样。其实它们有相同的地方，也有不一样的地方。

变量有独立的存储空间，可以用它来存储数据，也可以读取其中的数据。函数是一个功能，函数名没有独立的存储空间。函数中的数据是存储在函数所包含的变量里面。这就意味着可以对变量赋值，但是不能对函数赋值。例如，

```
int a = 2;
```

该语句可以对整型变量 a 直接赋值，但是如果对函数直接赋值就是错误的。对函数输入数据要通过形参变量，不能通过对函数直接赋值的方式输入数据。另外，函数的主要功能是处理数据，输出结果数据，不是用于存储数据。例如，

```
Max = 2;
```

上面对 Max 函数赋值的语句是错误的。大家可能会有疑问，在函数的内部不是有变量吗？难道这些变量不可以存储数据？这些变量是可以存储数据的，但是只能通过对这些变量直接赋值的方式来存储数据，而不能通过对函数赋值的方式来存储数据。

函数有返回数据，变量也有数据，可以使用它们的数据，这是二者相同的地方。因此，函数除了不能像变量那样存储数据之外，它的使用方式与变量基本是相同的。它可以像变量一样来构建函数表达式，也可以作为函数的实参，还可以单独成为一条函数调用语句。

（1）函数表达式。

有返回数值的函数可以像变量一样参与表达式的运算，这样的表达式又叫作函数表达式。例如，

```
int a = Max(4,5) + 2;
```

Max(4,5)的返回值是 5，5＋2 的结果为 7，因此变量 a 的值为 7。

（2）函数的实参。

变量可以作为函数的实参。例如，求变量 a、b、c 中的最大值并存储在变量 d 中，代码如下：

```
int a = 4,b = 5,c = 6,d;
d = Max(a,b);
d = Max(d,c);
```

也可以这样写

```
d = Max(Max(a,b),c);
```

此时，Max(a,b)函数的返回值又成为另一次 Max 函数调用的实参。

（3）函数调用语句。

对于具有返回值的函数除了不能对它赋值以外，其他使用方式与变量基本一致，它更像一个具有一定功能的变量。没有返回值的函数就不能构成函数表达式或者作为函数的实参。没有返回值的函数可以通过加上英文分号“;”构成函数语句，通过函数语句的方式来调用它的功能。

例如，对【例 5.2】中没有返回值的 Max 函数通过函数语句方式进行调用。

```
Max(4,5);
```

Max 函数将通过 printf 语句输出“a,b 中较大的值是 5”。

对于有返回值的函数也可以通过函数调用语句的方式进行调用。一般情况下，对于有返回值的函数不会通过函数语句的方式进行调用，因为通过函数语句调用无法使用函数的返回值，这样做没有太大的意义。

5.2 对程序模块进行组装

利用函数可以实现程序的模块化。当编写完一个个功能独立的函数以后，接下来就需要对函数进行组装，即通过函数模块之间的互相调用来构建一个具有特定功能，能够完成某种特定任务的程序。

5.2.1 程序模块间的组装问题

在C语言中，函数可以分为3类：main函数、库函数和用户自定义函数。这3种函数的结构都符合函数的结构要求，它们都包含函数类型、函数名、形参、函数体和返回值等基本要素。通过它们之间的互相调用可以实现程序模块间的自由组合。

1. 函数的分类

虽然这3类函数的结构相同，但是它们在程序中的地位与作用有所不同，因此它们的使用方法也存在一定的差异。

1）主函数

main函数又称为主函数，它是程序的入口，也是组装其他函数的起始点。在main函数中可以调用其他函数，其他函数又可以调用另外的函数，但是无论函数之间如何调用，最后都会在main函数内结束调用。需要注意的是，其他函数不可以调用main函数，main函数只能被操作系统调用。

2）用户自定义函数

用户自定义函数是程序员在开发某个特定程序的时候，根据程序的功能模块划分而编写的函数。用户自定义函数一般只在该程序内部使用。当然也可以根据需要将这些函数的定义代码复制到其他程序中使用。

3）库函数

库函数是一些具有通用功能的函数集合。它包括C语言标准规定的库函数和编译器特定的库函数，不同的编译器提供的库函数可能是不同的。库函数（如输入输出函数——printf函数、scanf函数，数学上的开平方函数——sqrt函数等）是一种特殊的用户自定义函数，它将用户自定义函数封装入库，提供给程序员在C源程序中进行调用。

库函数的函数定义代码一般是不可见的，库函数的头文件中包含了库函数的声明。程序员在使用库函数的时候不需要将库函数的源程序代码复制到源程序中，也不需要对库函数进行函数声明，只需要把库函数声明所在的头文件名用“#include <头文件名>”指令包含到源程序中就可以完成对库函数的声明。

在编写程序的时候要多使用库函数，这样可以提高程序的开发与执行效率。**头文件是一种包含函数、数据接口声明的文件，主要用于保存程序的声明。**

例如，printf 函数的声明包含在 stdio.h 头文件中，如下：

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

它是编译预处理命令，是程序在编译之前需要处理的内容。如果在程序中用到 C 语言标准函数库中的输入输出函数的时候，需要在程序的开头写上一行：#include "stdio.h"或者#include < stdio.h >，只有这样才能调用 C 语言标准函数库中的输入输出函数。#include<stdio.h>一般用于包含系统文件，它是从系统目录开始查找头文件，而#include "stdio.h"一般用于包含项目文件，它是先从项目目录开始查找，如果找不到该文件，再从系统目录查找。二者的区别主要是在头文件的查找效率上有所不同。

2. 函数的调用

在 main 函数、用户自定义函数和库函数之间进行调用需要遵循一定的规则。操作系统可以调用 main 函数。main 函数可以调用用户自定义函数和库函数。用户自定义函数之间可以互相调用，它也可以调用库函数。库函数只能调用库函数。函数调用关系如图 5.3 所示。

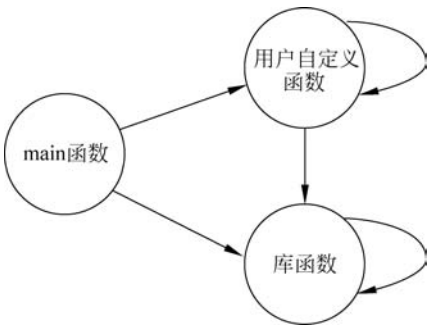

图 5.3　不同类型函数间相互调用的关系示意图

微课 5.3　函数调用与参数传递

1）函数的嵌套调用

main 函数、用户自定义函数和库函数在定义的时候是相互独立的，它们不能嵌套定义，即在一个函数内部不能定义另一个函数。但是它们之间在调用的时候可以嵌套调用，即在调用一个函数的过程中，又可以调用另一个函数。

【例 5.5】 设计一个 Max3 函数实现求 3 个整数的最大值。

问题分析： Max3 函数已经有了名称，还需要确定函数类型、形参和函数体 3 个要素。函数的功能是求 3 个整数的最大值，需要向 Max3 函数中输入 3 个整数，因此需要定义 3 个形参变量，如 int a、int b、int c。Max3 函数执行后需要返回最大值，返回值也可以定义为 int 类型。这样可以写出 Max3 的函数声明如下：

```
int Max3(int x,int y,int z);
```

有了 Max3 函数的声明就可以编写 main 函数的代码。在 main 函数中，利用 scanf 函数从键盘读入 3 个整数，然后调用 Max3 函数求得最大值。先对 main 函数进行定义。

程序代码如下：

```
#include <stdio.h>
int main()
{
    int Max3(int x,int y,int z);                          //对 Max3 函数进行声明
    int x,y,z;
    scanf("%d%d%d",&x,&y,&z);
    printf("x,y,z 中最大值是%d。\n",Max3(x,y,z)); //对 Max3 函数进行调用
    return 0;
}
```

根据 Max3 函数的声明信息，在 main 函数中就可以编写调用它的程序代码了。也就是说，在定义 main 函数的过程中并不需要关注 Max3 函数是否已经完成定义，是否正确定义，因为这些都是 Max3 函数的定义问题。如果没有定义 Max3 函数，则需要去定义它，如果 Max3 函数定义的代码有错误，则需要解决 Max3 函数代码的错误问题，而这一切都与 main 函数的定义无关。这就是函数模块化的特点，它让函数之间保持了一定的独立性。因此，在定义一个函数的时候，我们并不需要关注它所调用的函数是如何实现的，只需要关注被调函数的输入数据和输出数据就可以了，这样函数调用就变得简单明了。

定义 Max3 函数的工作只剩下最后一步——编写定义函数体的代码，实现比较 3 个整数最大值的功能。在前面的例子中，我们编写了可以实现对两个整数大小比较的 Max 函数。对于求 3 个整数的最大值问题，如果可以转换成求两个整数最大值的问题，那么 Max3 函数就可以通过调用 Max 函数来实现求 3 个整数的最大值的功能了。其实，无论求多少个整数的最大值问题都可以转换成求两个整数最大值的问题。只要先比较两个整数，然后让其中较大值的整数再与下一个整数进行比较就可以求得 3 个整数中的最大值。重复这样的过程，直到比较完最后一个整数就可以求得任意个整数中的最大值。

Max 函数的定义见【例 5.1】，函数声明如下：

```
int Max(int,int);
```

在 Max 函数的声明中只给出了形参的类型并未给出形参的名称，这是因为编译系统并不会检查函数声明中形参的名称与它定义时的名称是否一致，因此函数声明时可以不写形参的名称，但是它们的形参的数据类型、个数以及顺序一定都要相同。

根据 Max 函数的声明信息，对 Max3 函数进行定义。

程序代码如下：

```
int Max3(int x,int y,int z)
{
```

```
    int m;
    int Max(int,int);                    //对 Max 函数进行声明
    m = Max(x,y);                        //求出 x 和 y 中的较大值,并赋值给 m
    m = Max(m,z);                        //求出 m 和 z 中的较大值,并赋值给 m
    return m;                            //返回 m 中的值
}
```

在上面的例子中,main 函数调用了 Max3 函数求 3 个 int 数值中的最大值,为此 Max3 函数又通过两次调用 Max 函数实现了求 3 个整数最大值的功能。这种在函数的定义中又调用另一个函数的情况就是典型的函数嵌套调用,它是函数调用的最常见方式。

在学习函数知识的过程中,大家需要注意以下两个问题。

问题 1:实参变量与形参变量能互相改变吗? 例如,在函数调用语句"Max(x,y);"中,实参变量是 x 和 y,Max 函数的形参变量是 a 和 b。当"Max(x,y);"语句执行的时候,实参 x 和 y 的值会分别传递给形参 a 和 b,因此实参会改变形参的值。如果形参 a 和 b 的值发生了变化,那么它们会相应地改变实参 x 和 y 的值吗? 答案是不会,这是为什么呢?

问题 2:当实参变量与形参变量的名字相同的时候,它们是同一个变量吗? 例如,在 main 函数中有实参变量 x、y、z,在 Max3 函数中又有形参变量 x、y、z,它们名字相同,它们是同一组变量吗? 它们不是同一组变量。为什么在同一个函数中变量不允许重名,而在不同的函数变量又允许重名呢?

我们先来探讨第一个问题,为什么实参可以改变形参的值,而形参却无法改变实参的值。实参的数据可以传递给形参,而形参的数据却不能反向传递给实参,这是为了实现函数的封闭性要求。实参属于主调函数,形参属于被调函数,用于接收主调函数的输入数据,即实参的数据。主调函数只能够通过被调函数的返回值接收被调函数对它的影响。如果形参能够反向改变实参的值,那么主调函数和被调函数之间封闭性就会被破坏。下面介绍实参与形参之间的数据传递过程。

2) 实参与形参间的单向数据传递

函数只能通过形参变量接收输入数据,确保无法通过其他途径对函数体的功能进行干扰,目的是保持函数的封闭性。实参变量存储的是被调函数的输入数据,需要将实参变量的值传递给形参变量,因此在函数调用时实参变量可以改变形参变量的值。

在 C 程序中,经常会遇到需要交换两个变量数值的问题,那么能不能编写一个交换两个变量数值的函数呢?

【例 5.6】 在 main 函数中交换变量 x、y 的值。

```
#include <stdio.h>
int main()
{
    int x,y,z;
    scanf("%d%d",&x,&y);
    z = x;
    x = y;
```

```
8        y = z;
9        printf("x = %d,y = %d\n",x,y);
10       return 0;
11   }
```

在第6～8行代码中借助变量z交换了变量x和y的值。如果此时运行程序，通过键盘输入“4 5”，那么程序的输出结果是“x＝5，y＝4”。

【例5.7】 定义Exchange函数实现交换两个变量的值。

问题分析：函数的输入数据为两个要交换数据的整型变量，形参有两个：int a和int b。函数不需要返回值，因此Exchange的函数类型是void类型。对Exchange函数进行定义。

程序代码如下：

```
1    void Exchange(int a,int b)
2    {
3        int z;                                  //⑤
4        z = a;                                  //⑥
5        a = b;                                  //⑦
6        b = z;                                  //⑧
7        printf("a = %d,b = %d\n",a,b);
8    }                                           //⑨
```

在Exchange函数中交换变量a，b值的代码与【例5.6】中交换变量x，y值的代码的功能是相同的。下面在main函数中调用Exchange函数尝试实现对main中变量x和y数据的交换。

程序代码如下：

```
1    #include <stdio.h>
2    int main()
3    {
4        int x,y,z;                              //①
5        void Exchange(int a,int b);
6        scanf("%d%d",&x,&y);                    //②
7        Exchange(x,y);                          //③调用 Exchange 函数，交换 x，y 的值
8        printf("x = %d,y = %d",x,y);
9        return 0;                               //④
10   }
```

程序运行后，为变量x和变量y分别输入4和5，调用Exchange函数将实参x和y的值分别传递给形参变量a和变量b，此时a和b的值分别为4和5。Exchange函数交换了a和b的值，在Exchange中通过printf语句输出a和b的值，此时a和b的值分别是5和4。函数调用结束，在main函数中执行printf语句再次输出x和y的值，x和y的值仍然是4和5，并没有发生交换。程序运行的结果参见图5.4，可以发现变量a和b的值进行了交换，但变量x和y的值并没有被交换。

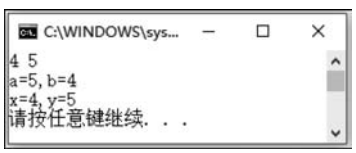

图5.4 Exchange函数执行结果

实参变量和形参变量都有自己独立的存储空间。在函数调用时只是将实参变量中存储的数据复制到形参变量的存储空间。当函数调用结束后并不会再将形参变量中存储的数据复制到实参变量中。在程序执行过程中，实参变量和形参变量中存储的数据的变化过程参见图 5.5。

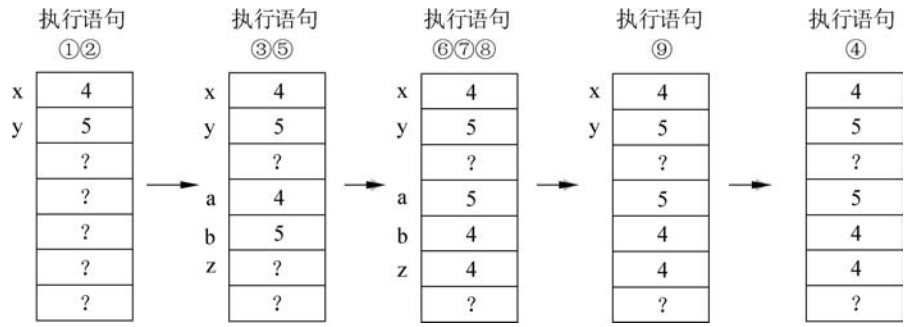

图 5.5　实参向形参传递数据

当语句①执行时，变量 x 和变量 y 被分配了存储空间，它们中存储的数值未知。继续执行语句②时，从键盘输入数值 4 和 5，变量 x 和 y 中分别存储了 4 和 5。

继续执行语句③，此时调用 Exchange 函数，形参变量 a 和 b 被分配了存储空间，实参 x 和 y 中的数值 4 和 5 会被复制到变量 a 和 b 中。执行语句⑤，变量 z 被分配了存储空间。

执行 Exhange 函数中的语句⑥⑦⑧，借助变量 z，变量 a 和 b 实现了数值交换，此时变量 z 的值为 4，变量 a 的值为 5，变量 b 的值为 4。当执行语句完⑨，Exchange 函数调用结束，变量 a、b 和 z 的存储空间被释放。虽然变量 a、b 和 z 消失了，但是它们的数值仍存储在内存中，只是无法再通过变量 a、b 和 z 访问这些数值了。

最后执行语句④，main 函数退出，变量 x 和 y 的存储空间也被释放，也无法通过变量 x 和 y 访问内存中的数据 4 和 5 了。

从上面的过程中可以看出，**实参变量与形参变量之间是单向数据传递**。实参变量与形参变量各自都有独立的存储空间。在函数调用的时候，程序会将实参变量中的数值复制到相对应的形参变量中，从而实现向函数中输入数据。在函数调用的时候，形参变量才出现并被分配存储空间。在函数体语句中，对形参变量的改变，只会对形参变量中存储的数据进行修改，不会同时修改对应实参变量中的数据，函数执行完毕后，形参变量的存储空间就被释放了。通过改变形参变量的值来改变实参变量值的想法违反了函数设计的初衷，因此函数不支持形参变量向实参变量的反向数据传递。

如果要在被调函数中改变主调函数中实参变量的值，那么可以通过被调函数的 return 语句返回形参变量的值，将返回值赋值给实参变量，但是 return 语句只能返回一个值，这种方法只能改变一个实参变量的值。

5.3.2 节将介绍如何在被调函数中通过指针改变主调函数中多个实参变量值的方法。但是这种方法破坏了函数的封闭性，因此一般不提倡这样做。

3. 函数调用举例

【例 5.8】 通过定义阶乘函数 fac 实现求 n!。要求在 main 函数中接收从键盘输入的一个自然数 n,通过调用 fac 函数的方式实现求 n!。

问题分析：fac 函数的输入数据 n 为自然数,形参变量可以定义为 int n。n! 的计算结果是整数,fac 函数的类型也可以定义为 int 类型。fac 函数的声明如下：

```
int fac(int n);
```

阶乘的数学公式为

$$n!=\begin{cases}1, & n=0,1\\ 1\times 2\times 3\times\cdots\times(n-1)\times n, & n>1\end{cases}$$

当 n 等于 0 时,0! 的值是 1。当 n 大于 0 时,计算 n! 的值可以从 1 开始,乘 2,再乘 3,一直乘到 n,可以使用循环语句来实现。

程序代码如下：

```
#include <stdio.h>
int fac(int n)
{
    int i,s;                                  //定义 s 存储阶乘值
    if (n == 0) s = 1;                        //求 0!
    else for(i = 1,s = 1;i <= n;i++) s = s * i;  //求 n!,n >= 1
    return s;                                 //返回阶乘值
}
int main()
{
    int n;
    int fac(int);                             //对 fac 函数声明
    scanf("%d",&n);
    if (n < 0) printf("n < 0,输入数据错误!\n");
    else printf("%d!= %d\n",n,fac(n));        //调用 fac 函数
    return 0;
}
```

在此之前,我们学习的函数调用都是一个函数调用另外一个函数,那么一个函数能否对自己进行调用呢？可以。当我们在程序中用递归思想来解决问题时,必须采用一个函数对自身进行直接或者间接调用的方式来解决递归问题。**一个函数对自身的直接或者间接调用称为递归调用**。下面介绍递归思想的程序实现过程。

5.2.2 递归思想的程序实现

阶乘公式还有另外一种表达方式：

$$n!=\begin{cases}1, & n=0,1\\ n\times(n-1)!, & n>1\end{cases}$$

这种表达式体现了一种递归的思想，又称为递归表达式。如果求 $n!$，只要先求出 $(n-1)!$，再乘以 n 就可以得到 $n!$。同理，如果要求 $(n-1)!$ 的阶乘，只要先求得 $(n-2)!$，再乘以 $n-1$ 就可以求得 $(n-1)!$。这样一直递推下去，直至求解 $2!$，只要求 $1!$，再乘以 2，由于已知 $1!$ 等于 1，因此可以求得 $2!$ 等于 2。如此回溯，可以求得 $3!$ 等于 6，直至 $n!$。这种求 $n!$ 的方法就是利用了递归思想。

1. 递归原理

递归是数学上一种分而治之的思想。它是把一个大型的复杂问题层层转化为一个与原问题相似、规模较小的问题来求解。随着问题规模不断变小，问题的复杂度也不断地降低，直至最小的问题被解决，然后返回来不断地解决较大的问题，直到原先的问题被解答。利用递归思想可以解决很多复杂的问题。下面通过数俄罗斯套娃数量的问题来理解递归思想。

微课 5.4　函数递归调用

俄罗斯套娃是一种俄罗斯特产的木制玩具。它一般由多个图案相同但大小不同的空心木娃娃一个套一个组成，越往里面套娃越小，套娃的数量可达十多个。

你有没有想过如果让你来数套娃的数量，你会怎么数呢？一般情况下，我们都会打开一个套娃，数一个，再打开一个，再数一个，直到打开最后一个套娃，发现里面有没有套娃了，就可以得到套娃的总数量。其实还有另外一种数法，我们很少会这样做，但是也可以数出套娃的数量。我们先一直打开套娃不计算数量，直到打开最后一个套娃，然后再把套娃盖上，盖上一个数一个，直到最大的套娃被盖上，也能数出所有套娃的个数。

大家可能会有疑惑，第一种方法很快啊，打开最后一个就能数出套娃的数量，为什么还要在一个个盖上套娃数数量？这样数不是慢吗？如果只是数出数量，不要求恢复套娃的原状，第一种方式比较快。如果不但要算出套娃的数量，还要把套娃恢复原状，那么这两种方式的效率就是一样的了。

我们也可以通过编写程序让计算机来完成数套娃的工作。打开套娃和累加计数是一种重复性活动，可以用循环语句来实现数套娃的工作。假设打开套娃的函数定义为 open 函数，盖上套娃的函数定义为 close 函数。NoTaowa 函数可以判断第 n 层套娃中是否仍存在套娃。如果 NoTaowa 函数的返回值是 1，则说明该套娃里面没有套娃；如果返回值是 0，则说明它里面还有套娃。open 函数、close 函数和 NoTaowa 函数的声明如下：

```
void open(int n);          //n 是套娃的层编号，可以任意设定最外层的套娃编号，内层比外层小 1
void close(int n);
int NoTaowa(int n);
```

定义 count 函数，它的功能是计算第 n 层套娃中套娃的个数。下面定义利用第一种方式数套娃的 count 函数。

程序代码如下：

```
1    int count(int n)
```

```
2   {
3       int no = n,number = 0;              //用 no 记住套娃的编号,number 是套娃的数量
4       do                                  //打开套娃并计数
5       {
6          open(n);
7          number++;
8          n--;
9       }while(!NoTaowa(n));                //当打开最后 1 个套娃时 NoTaowa 函数返回 1
10      do                                  //盖上套娃
11      {
12         close(n);
13         n++;
14      } while(no!= n);                    //盖上第 1 个套娃,循环结束
15      return number;
16  }
```

利用第二种方式数套娃的 count 函数。

程序代码如下：

```
1   int count(int n)
2   {
3       int no = n,number = 1;
4       do                                  //打开套娃
5       {
6           open(n);
7           n--;
8        } while(!NoTaowa(n));
9        do                                 //盖上套娃并计数
10       {
11          close(n);
12          number++;
13          n++;
14       } while(no!= n);
15       return number;
16  }
```

这两种数套娃的方法虽然不同,但是它们解决问题的思路是相同的。它们都是把数套娃的问题看作是一个完整的问题,而没有把这个问题进一步地分解成更小的问题。

我们也可以用递归的思想来解决数套娃的问题。每当我们打开一个套娃,那么剩下的套娃的数量就会减少,相当于数套娃问题的复杂性在降低。同时,数剩下套娃数量的方法与数原来套娃的方法是相同的,也就是分解的小问题与原问题的解决方法是一致的。这些条件说明数套娃的问题可以用递归方法来解决。

具体思路是：外层的套娃编号为 n,其内层的套娃为 n－1。要数出第 n 号套娃中包含的套娃的数量,只要能先数出第 n－1 号套娃中所包含的套娃的数量,再加上 1,就是它包含的套娃的个数。以此类推,直到遇到第 x 号套娃中没有套娃为止,此时 x 号套娃中包含的套娃数量为 1。用递归方法实现 count 函数。

程序代码定义如下：

```
1     int count (int n)
2     {
3         int number = 0;
4         open(n);
5         if (NoTaowa(n)) number = 1;      //如果是最后 1 个套娃,number 值是 1
6         else number = count(n - 1) + 1;  //如果不是最后 1 个,继续数里层套娃
7         close(n);
8         return number;
9     }
```

第 5 行语句是一个 if 选择语句，如果编号 n 的套娃中没有套娃了，则说明它是最后一个套娃，套娃的数量 number 为 1，通过 return 函数返回套娃数量 1。如果它里面还有编号 n－1 的套娃，则继续调用 count(n－1)函数，计算编号 n－1 的套娃中包含的套娃的数量，通过 return 函数返回 number 的值就是 n 号套娃中套娃的数量。函数 count(n－1)会继续调用 count 函数，重复 count(n－2)的过程，直至递归结束。在第 6 行代码中，count(n)函数对自身进行了调用 count(n－1)，这种函数调用就是递归调用。

递归策略只需要用少量的程序代码就可以描述出解题过程，极大地减少了程序的代码量。对比上面 3 种 count 函数的代码也可以看出来，用递归方法实现的 count 函数的代码要比前两种简洁得多。但是，如果不能够理解递归的思想，那么遇到实际的问题也很难想出递归的方法，也就无法编写出递归函数。

2. 设计递归函数

如果理解了递归思想并且掌握了利用递归思想来解决实际问题的方法，那么我们就可以来设计递归函数，让计算机也可以学会用递归方法来解决实际的问题。在构建递归函数的时候重点关注两个部分：

1）递归的关系表达式

构建递归关系表达式是非常困难的一步。只有多接触递归问题才能够掌握递归的方法，然后通过函数递归调用的表达式将要解决的问题描述出来。

例如，在用递归方法数套娃的例子中，求编号 n 套娃中所包含的套娃数量的递归关系表达式是“count(n)＝count(n－1)＋1”。在定义递归函数 count 的函数体中，不能够将递归关系表达式“count(n)＝count(n－1)＋1”直接作为函数递归调用的表达式，因为 count(n)是函数调用，无法对它进行赋值，需要再引入一个变量 number，将递归关系表达式转换成递归表达式语句“number＝count(n－1)＋1;”。

2）终止递归的条件表达式

构建终止递归的条件表达式就是构建描述解决最小问题的表达式，否则递归函数会一直调用自己进入无限递归，导致程序无法结束。虽然递归终止的条件表达式相对容易确定，但并不一定唯一，而且十分容易遗漏，导致递归无法停止。

例如，递归函数 count 的终止条件是“if (NoTaowa(n)) number＝1”。在递归函数

中，一般会使用 if 语句来构建递归的终止条件和递归调用的表达式。

3. 递归举例

【例 5.9】 利用函数递归调用实现求 n!。

问题分析：在求 n! 的问题中，根据阶乘公式“n!＝n＊(n－1)!”构建递归的关系表达式是：

$$fac(n)=n*fac(n-1)$$

在函数 fac 中，不能直接使用 fac(n)＝n＊fac(n－1)作为函数递归调用的表达式，需要引入整型变量 s 存储 fac(n)的值，并构建函数调用的递归表达式 s＝n＊fac(n－1)。函数 fac 的输出结果是整数值，可以定义 fac 的函数类型为 int。函数的输入数据是整数 n，可以定义函数 fac 的形参变量的数据类型是 int。

程序代码如下：

```
1   int fac(int n)
2   {
3       int s;
4       s = fac(n - 1) * n;                  //递归调用
5       return s;
6   }
```

这段代码中隐藏了一个很大的问题，函数 fac 的递归调用是无法停止的，也就是说，当 n 的值小于 0 的时候，函数会继续递归调用下去。这会产生错误，因为负数没有阶乘。

在递归函数 fac 中缺少了递归调用的终止条件。函数 fac 的递归终止条件是“if(n＝＝0) s＝1;”，也可以将其修改为“if(n＝＝0||n＝＝1) s＝1;”。在 fac 函数定义中加入递归终止条件。

程序代码如下：

```
1   int fac(int n)
2   {
3       int s;
4       if(n == 0) s = 1;                    //递归终止条件
5       else s = n * fac(n - 1);             //递归调用
6       return s;
7   }
```

在 main 中对 fac 函数进行调用。

程序代码如下：

```
1   #include < stdio.h >
2   int main()
3   {
4       int n;
5       int fac(int);
6       scanf("%d",&n);
```

```
7        if(n<0) printf("n<0,输入数据错误!\n");
8        else printf("%d!=%d\n",n,fac(n));  //调用 fac 函数
9    }
```

当程序运行后，从键盘输入数字 3，程序执行的结果是在屏幕上输出“3!＝6”。在函数 fac(3)递归调用的过程中，程序代码执行过程示意图参见图 5.6。

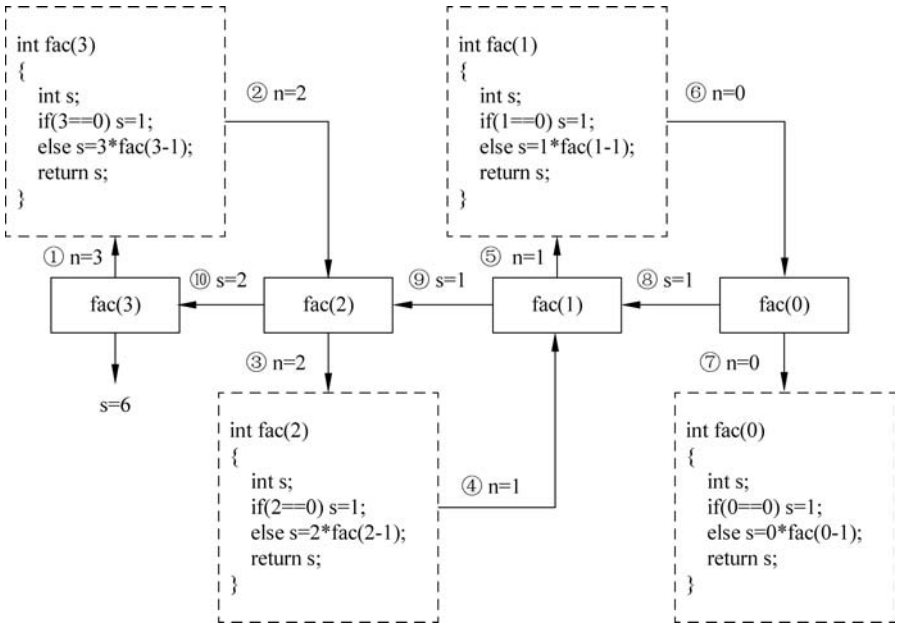

图 5.6　函数 fac(3)的递归调用过程

当 main 函数执行第 8 行语句调用 fac 函数的时候，此时 n＝3，在 fac(3)函数中所有的 n 的值都是 3，fac(3)的代码见图 5.6 中①。此时 if 语句中的条件表达式“3＝＝0”为假，因此执行表达式 s＝3＊fac(2)。接着调用 fac(2)，见图 5.6 中②。fac(2)的代码见图 5.6 中③，此时 if 语句中的条件表达式“2＝＝0”仍然是假，执行 s＝2＊fac(1)。接着调用 fac(1)，见图 5.6 中④，fac(1)的代码见图 5.6 中⑤。此时 if 语句中的条件表达式(1＝＝0)仍然是假，执行 s＝1＊fac(0)。接着调用 fac(0)，见图 5.6 中⑥，fac(0)的代码见图 5.6 中⑦。此时 if 语句中的条件表达式“0＝＝0”为真，返回 fac(0)的值 s＝1，见图 5.6 中⑧。此时函数 fac(1)中，s＝1＊fac(0)的结果是 1，fac(1)的返回值是 1，见图 5.6 中⑨。同理返回 fac(2)的值 2，见图 5.6 中⑩。最后得到 fac(3)的值 6，即 s＝3＊fac(2)。

从函数递归调用的过程来看，递归调用占用了大量的内存空间，只有当递归调用终止的时候，随着递归调用的返回，函数才会逐步地释放掉内存资源。

【例 5.10】 利用递归求斐波那契数列。

1202 年，意大利数学家斐波那契提出了一个有趣的问题：假设 1 对刚出生的小兔 1 个月后就能长成大兔子，再过 1 个月就能生下 1 对小兔子，并且此后每个月都生 1 对小兔子，假设兔子 1 年内没有发生死亡，问：1 对刚出生的兔子，1 年内能繁殖多少对兔子？

更甚者，假设兔子是永生不死的，那么第 n 个月有多少对兔子呢？

问题分析：

通过计算可以知道，第一个月的兔子数为 1 对（小兔子），第二个月的兔子数仍然为 1 对（大兔子），到第三个月一对小兔子出生，兔子总数为 2 对（一对大兔子和一对小兔子），以此类推，可以得到前面几个月的兔子数目为：1,1,2,3,5,8,...,13...。

分析上面的序列，可以发现一个规律，也就是第 n（n＞2）个月的兔子的数量是第 n－1 个月和第 n－2 个月兔子数量的和。用下面的公式可以表示这个关系：

$$\mathrm{Fib}(n)=\begin{cases}1, & n=1,2\\ \mathrm{Fib}(n-1)+\mathrm{Fib}(n-2), & n>2\end{cases}$$

观察上面的公式，可以发现斐波那契数列也可用递归方法来计算。定义存储 Fib(n) 函数的返回值的变量 int f。递归的关系表达式是“f= Fib(n－1)＋Fib(n－2)”。终止递归的条件表达式是“n==1||n==2”，当它成立时 f=1。

程序代码如下：

```
1   #include<stdio.h>
2   int main()
3   {
4       int n;
5       int Fib(int);
6       scanf("%d",&n);
7       if(n<0) printf("n<0,输入数据错误!\n");
8       else printf("Fib(%d) = %d\n",n,Fib(n));
9       return 0;
10  }
11  int Fib(int n)
12  {
13      int f;
14      if(n==1||n==2) f=1;            //递归的终止条件表达式
15      else f=Fib(n-1)+Fib(n-2);      //递归的关系表达式
16      return f;
17  }
```

求解斐波那契数列和求阶乘的函数都是利用递归方法实现的，但是两者之间还是有区别的。在求阶乘的函数定义中，递归只是调用了一次，而在求斐波那契数列函数的定义中，递归却调用了两次。如果从调用关系上来看，求斐波那契数列的递归调用会是一棵树，这会导致调用的过程更加复杂，运行的效率会更慢。大家可以在 VS2010 环境中运行感受一下这段代码的运行过程，也可以思考如何能够更快地进行求解。

5.3 人类永恒的话题“矛盾”：封闭性与开放性

虽然函数有不同的定义形式和调用的方法，但是其目的都是为了实现程序代码的模块化。通过函数的封装，可以将它内部定义的变量封闭起来，避免了其他函数访问这些

内部的变量，从而保证函数功能的独立性。因此，函数之间只能通过形参和函数的返回值交换数据。但是函数的返回值只有一个，如果函数需要返回多个数值，那又该怎么办呢？这样的封闭性是函数的优势，但同时也是它的劣势。我们需要另外一些方式来提高函数的开放性，以便在函数之间实现更灵活的数据交互。

5.3.1 不准动我的积木

一个积木模块的内部结构是固定的。积木模块之间只能够通过“凹凸”接口进行拼接，但是在拼接过程中它们是不能互相更改对方的结构的。一个函数模块的功能是也固定的。在函数之间进行调用的过程中它们也是不能相互更改对方的功能的。为了保持函数的这种功能的独立性，在一个函数内部定义的变量不允许被其他函数访问。**在函数内部定义的变量称为局部变量**。它们只能在函数的内部使用，因此被称为局部变量。变量的作用范围又称为变量的作用域，局部变量的作用域局限于函数内部。通过对函数中局部变量作用域的限制，使得函数具有了封闭性，从而保证了其功能的独立性。

1. 函数的私有财产——局部变量

在计算机中，最宝贵的资源是计算资源和存储资源。由于局部变量的作用域仅限于函数内部，因此只有当函数被调用时计算机才会为局部变量分配存储空间。当函数调用结束时，这些局部变量所占用的内存空间又会被系统收回，重新分配给其他函数的局部变量使用。这种对局部变量的存储空间进行动态分配的使用方式可以提高计算机的内存利用效率。

微课 5.5 局部变量与全局变量

在 5.2.1 节，对函数的调用我们曾经提出过两个问题，其中问题二：实参变量与形参变量的名字相同，它们是同一个变量吗？我们已经知道了答案：不是。那为什么不是呢？现在可以找到原因了，因为它们分别是不同函数的局部变量。实参变量是主调函数的局部变量，形参变量是被调函数的局部变量。它们的作用域是各自的函数内部，因此虽然它们重名但是不会产生冲突。

【例 5.11】 分析 main 函数中函数调用 fac(n)中的实参变量 n 与 fac 函数定义中的形参变量 n 是否是同一个变量。

程序代码如下：

```
#include <stdio.h>
int main()
{
    int n;                                          //局部变量 n
    int fac(int);
    scanf("%d",&n);
    if (n<0) printf("n<0,输入数据错误!\n");
    else printf("%d!= %d\n",n,fac(n));              //实参变量 n
}
```

```
10  int fac(int n)                          //形参变量 n
11  {
12      int s;
13      if (n == 0) s = 1;
14      else s = n * fac(n - 1);
15      return s;
16  }
```

在 main 函数中，第 4 行语句定义了局部变量 n。在 fac 函数中，第 10 行和第 12 行语句分别定义了局部变量 n 和 s。根据局部变量的作用域可以知道在上述代码中共有 3 个局部变量，main 函数中的变量 n，fac 函数中的变量 n 和 s。main 函数中定义的变量 n 只在 main 函数中起作用，因此第 8 行函数调用 fac(n)中的 n 是 main 函数的局部变量 n，同时它也是 fac 函数的实参。在第 13 行和第 14 行代码中的 n 是 fac 函数的局部变量 n，同时它也是 fac 函数的形参。在这个例子中，实参变量 n 和形参变量 n 是两个重名的局部变量，分别属于 main 函数和 fac 函数。

不同函数的局部变量的命名最好不要相同，因为如果对局部变量的概念不清楚，往往会认为重名的局部变量是同一个变量，其实它们是不同的变量。在使用局部变量时应重点关注以下几个问题：

(1) 不能跨函数访问局部变量。

在一个函数中不能够访问另一个函数的局部变量。例如，在 main 函数中无法对 fac 函数中的局部变量 s 进行访问。如果在 main 函数中对 fac 函数中的局部变量 s 进行引用，编译系统会给出错误提示：error C2065：“s”：未定义的标识符。

(2) 形参是局部变量。

局部变量是在函数内部定义的变量，由于函数的形参位于函数的首部，所以往往会造成形参变量不是局部变量的错觉。其实形参变量也是局部变量，只不过它是负责接收函数的输入数据。函数的形参变量可以在本函数内部引用，但是它也不能被其他函数访问。

(3) 局部变量可以重名。

既然局部变量只在函数内部起作用，不同函数中的局部变量可以重名。例如，main 函数中定义了局部变量 n，fac 函数中也定义了形参变量 n。在各自的函数内部引用变量 n，都是引用各自定义的变量 n，而不是其他函数定义的变量 n。这就像两个班级中都有一个名叫李明的同学，在各自班级里喊李明同学的名字的时候，只有本班级的李明同学才会听到，而另外一个李明同学根本听不到有人喊他的名字，因而在各自班级里面的李明同学不会因为重名而产生混淆。

(4) 局部变量的生存期是函数调用期间。

局部变量是动态存储方式。只有当函数被调用的时候，系统才会为局部变量分配内存空间，当函数调用结束后，局部变量的存储空间就会被系统回收。因此在函数没有被调用之前，局部变量是不占用内存空间的。例如，如果在 main 函数中去除掉第 8 行调用 fac 函数的语句，即使 fac 函数中定义了局部变量 n 和 s，它们也永远不会出现在内存中，

也就不会占用任何内存资源。

2. 复合语句中的局部变量

函数是程序的局部，函数中的变量是局部变量。局部变量只在函数调用时才会动态地占用内存空间。那么能不能让局部变量占用内存空间的时间再缩短一些，让它们只在函数的局部起作用呢？为此，C 语言提供了一种在函数内部通过复合语句定义局部变量的方式。只有当执行复合语句的时候，这些局部变量才会被分配存储空间。当复合语句执行结束时，系统会收回这些局部变量所占用的内存资源。从这些方面可以看出，对于程序来说内存资源的分配与使用是非常重要的。

在复合语句中定义的局部变量的作用域是复合语句内部，只能在复合语句内部使用。在复合语句外部定义的局部变量可以在复合语句内部被引用。在复合语句外部定义的变量允许与复合语句内部定义的变量重名。当发生重名的时候，在复合语句外部定义的变量在复合语句内部不起作用。

【例 5.12】 分析下面代码中局部变量 t 和重名的局部变量 s 的作用域。

程序代码如下：

```
1   #include<stdio.h>
2   int main()
3   {
4       int s=0;                                    //定义局部变量 s
5       int t=1;                                    //定义局部变量 t
6       {
7           int s=1;                                //定义局部变量 s
8           printf("s=%d,t=%d\n",s,t);
9       }
10      printf("s=%d\n",s);
11      return 0;
12  }
```

在 main 函数中，第 4 行语句中定义了局部变量 s，并赋值 0。第 5 行语句定义了局部变量 t，并赋值 1。在复合语句中，通过第 7 行语句定义了另一个局部变量 s，并赋值 1。此时在 main 函数中有两个重名的局部变量 s。

当第 8 行语句执行时，由于局部变量 s 重名，因此 printf 语句中的局部变量 s 是复合语句中定义的局部变量 s，s 的值是 1。在复合语句中没有定义与 t 重名的局部变量，因此 t 可以在复合语句中引用。printf 语句的输出结果是“s=1，t=1”。当复合语句执行结束后，复合语句中定义的局部变量 s 消失。

当执行第 10 行语句时，printf 语句中的变量 s 是函数头部定义的局部变量 s，它的值是 0，printf 语句的输出结果是“s=0”。

下面对函数内部定义的两种局部变量的区别进行小结：

(1) 局部变量定义的位置。

在函数中，可以在形参的位置、函数体的头部和复合语句中定义局部变量。除了这

3个位置以外，一般不能在函数的其他地方定义局部变量。

（2）局部变量的重名问题。

在不同函数中定义的局部变量可以重名。在一个函数中，形参变量和函数体头部定义的局部变量不可以重名，它们可以在函数的内部起作用。在复合语句内部定义的局部变量之间也不可以重名。在函数中，复合语句内部定义的局部变量可以与它外部定义的局部变量重名。

（3）局部变量的引用。

不同函数之间的局部变量不可以互相引用。在同一个函数中，形参变量和函数体头部定义的局部变量可以在整个函数中引用。在复合语句中定义的局部变量只能在复合语句中引用。

（4）局部变量的生存期。

在函数调用的整个期间内，形参变量和在函数体头部定义的局部变量都会存在，可以对它们进行访问。在复合语句中定义的局部变量只在复合语句执行期间存在，复合语句执行完成后，它们就会消失。

5.3.2 我偏要动你的积木

利用局部变量的访问控制机制，很好地保护了函数中变量数据的私密性。局部的反义词是全局。局部变量只能够在一个函数的内部使用，也可以定义一种全局变量，让它可以在函数之间使用。全局变量可以作为函数之间的“信使”来传递数据，函数之间通过访问全局变量来交互数据。全局变量可以提高函数之间数据交互的灵活性，但是它也破坏了函数的封闭性。如果对全局变量使用不当，对函数来说将是致命的打击。

1. 多个函数间的“信使”——全局变量

在函数外部定义的变量称为全局变量，全局变量可以为函数所共用。局部变量的空间是函数内部。对于函数内部的空间，我们容易理解。对于函数外部的空间，不太好理解。函数外部是指在一个程序中函数的外部空间。一个程序可以由多个源文件和头文件构成。到目前为止，我们只学习了在一个源文件中编写程序代码，还没有讨论在多个源文件中编写程序代码的问题。我们可以把所有的函数都存放于一个源文件中，但是一个源文件可以存储的数据的大小是有限的，可能无法存储所有的函数代码。如果把多个函数都存储在同一个源文件中也不利于程序员通过分工协作来编写代码。因此，大型的C语言程序一般都是由多个源文件构成的。一般情况下，一个函数的代码只会存储在一个源文件中，那么函数的外部就是指多个源文件的函数外部的空间。

局部变量的作用空间是函数内部，即使将函数分别存储于不同的源文件也不会影响它的使用。全局变量可以存储于不同的源文件中，在一个源文件中也可以存储在文件的不同位置，可以根据它们的存储位置对全局变量的作用域进行一定的设定，使得一些函数可以访问它而另外一些函数不能够访问它，让它的作用域变得更加灵活。

1）全局变量的定义与使用

在一个函数外部定义全局变量的位置有很多选择，可以在该函数的前面定义，也可以在该函数的后面定义，需要做一定的区别吗？可以做，也可以不做。为了精细控制全局变量的作用域，减少全局变量破坏函数封闭性的影响，**在C语言中规定了全局变量的作用范围为定义全局变量的位置开始到本文件结束**。这样只需要在第一个使用该全局变量的函数前面定义它就可以了。

【例5.13】 定义input函数接收从键盘输入的两个数据，定义max函数实现对输入两个数据的大小比较，并输出其中的最大值。利用全局变量实现input函数、max函数和main函数之间的数据交互。

问题分析：定义3个全局变量a、b、c，其中a、b两个变量用于存储输入的两个数据，变量c用于存储最大值数据。input函数和max函数通过访问全局变量来输入和输出数据，因此它们都不需要定义形参变量，也不需要通过return语句输出数据。input函数和max函数的声明如下：

```
void input();
void max();
```

程序代码如下：

```
#include <stdio.h>
float c;                                    //定义全局变量c
int main()
{
    void input();                           //函数声明
    void max();                             //函数声明
    input();                                //函数调用
    max();                                  //函数调用
    printf("a,b中较大值是%f\n",c);          //输出全局变量c的值
    return 0;
}
float a,b;                                  //定义全局变量a和b
void input()
{
    printf("请输入要比较的两个数字:");      //输入提示信息
    scanf("%f%f",&a,&b);                    //为全局变量a,b输入数据
}
void max()
{
    if(a>b) c=a;                            //访问全局变量c
    else c=b;
}
```

在这个例子中，全局变量a、b和c的作用域是不同的。全局变量c定义在main函数的前面，main函数以及它之后定义的所有函数都可以访问变量c。在input函数前面定义了全局变量a和b，在input函数和max函数中都可以访问变量a和b，但是在main函

数中不能够访问全局变量 a 和 b，因为它们是在 main 函数之后定义的。

全局变量是在函数外部定义的，可以在函数内部使用它，但是不能在函数外部使用它。

【例 5.14】 阅读下面的程序代码，判断 max 变量的输出值是多少。

程序代码如下：

```
1   #include <stdio.h>
2   int max = 0;
3   max = 1;                                    //存在语法错误
4   int main()
5   {
6       printf("max = %d\n",max);
7       return 0;
8   }
```

第 2 行语句定义了全局变量 max 同时初始化赋值 0。第 3 行语句对 max 赋值 1。第 6 行语句通过 printf 函数输出 max 的值。在程序编译的时候，编译系统会提示第 3 行代码中存在错误：error C2374："max"：重定义；多次初始化。假设该程序能够通过编译，输出的 max 值是 0 还是 1 呢？

我们都知道，程序是从 main 函数开始执行的，在 main 函数中再调用其他函数，直至 main 函数执行完毕后程序退出运行。第 3 行语句不在任何一个函数中，它永远不会被执行到。第 2 行语句是在定义全局变量 max 时进行初始化赋值，这是允许的，也就是说，在函数外面对全局变量进行赋值操作，只能在定义它的同时进行初始化赋值。

2）全局变量的存储方式

全局变量的内存管理是一种静态存储方式。**在程序运行的期间，系统为全局变量分配了固定的存储空间，只有当程序退出运行时系统才会收回全局变量的存储空间**。在程序中只要定义了全局变量，无论是否使用它，系统都要为它分配内存空间，并且在程序运行的整个期间内都要一直占用内存单元。

3）全局变量的重名问题

全局变量之间不可以重名，但是它与局部变量之间是可以重名的。当一个函数中的局部变量和全局变量重名时，在该函数内部重名的局部变量起作用。这一点与函数的两种局部变量的作用域相类似。在函数中，当复合语句定义的局部变量和函数体头部定义的局部变量重名时，在复合语句中，引用该变量时引用的是复合语句中定义的局部变量。这一点类似一句谚语"强龙压不过地头蛇"。

对局部变量与全局变量概念的定义，正是事物矛盾性在程序语言设计中的充分体现。我们想通过提高函数的封闭性，降低函数间的耦合性来实现函数的模块化，局部变量完美地解决了这个问题。但是，局部变量的引入导致了函数的交互性下降。我们又想要让多个函数在必要的时候能够建立某种特殊的数据通道来传递共享的数据，因此全局变量被设计出来，承担了函数之间传递数据的"信使"的职责。其实，无论定义什么样的规则，只有充分地理解规则、善用规则才能真正发挥这些规则的作用。

对于程序员来说，要尽量少用全局变量。过多地使用全局变量会丧失函数模块化带来的优越性，很难判定在某个瞬间，到底是哪个函数修改了全局变量的值，从而使程序的模块化程度变差，难以维护。

在为变量命名时，无论是全局变量还是局部变量，无论是函数的实参变量还是形参变量都尽量不要重名。重名会导致程序员对变量的识别和使用产生混淆，从而引起不必要的麻烦。当然，当你清楚地了解了函数的机理、局部变量和全局变量的用途，在提高内存使用效率的前提下，你可以按照你的意愿对全局变量和局部变量进行命名和使用。不是有那么一句话吗？“我的地盘我做主。”

2. * 不同源文件中的全局变量

在一个源文件中定义的全局变量可以被这个源文件内的函数访问，那它能否被其他源文件中的函数引用呢？这也是可以的。只要在引用函数所在的源文件中对要引用的全局变量进行声明就可以访问到该全局变量。这一点跟函数声明的作用类似。

对全局变量进行声明需要用到关键字 extern。在声明中，变量的数据类型可以省略。全局变量声明的格式如下：

extern [数据类型] 全局变量的名称;

【例 5.15】 编写程序实现对输入的两个数进行处理，并输出最大值、最小值和平均值。定义 average 函数实现求最大值、最小值和平均值。利用两个源文件分别存储 main 函数和 average 函数。

问题分析： average 函数可以返回一个数值，但是不能同时返回 3 个数值，因此至少需要定义 2 个全局变量，可以定义全局变量 max 和 min 分别存储最大值和最小值，通过 average 函数返回平均值。创建源文件 File1.c，在它里面定义 main 函数，在 main 函数中定义全局变量 max 和 min。创建源文件 File2.c，在它里面定义 average 函数。在 File2.c 中，用关键字 extern 对全局变量 max 和 min 进行声明。

File1.c 文件中的程序代码如下：

```
1   #include <stdio.h>
2   float max,min;                              //定义全局变量
3   int main()
4   {
5       float average(float,float);             //函数声明
6       float a,b;
7       printf("请输入要比较的两个数字:");
8       scanf("%f%f",&a,&b);
9       printf("max=%f,min=%f,ave=%f\n",max,min,average(a,b));
10      return 0;
11  }
```

File2.c 文件中的程序代码如下：

```
1     extern max,min;                           //也可以写成 extern int max,min;
```

```
2     float average(float a,float b)
3     {
4         if (a > b)
5         {
6             max = a;
7             min = b;
8         }
9         else
10        {
11            max = b;
12            min = a;
13        }
14       ave = (a + b)/2;
15       return ave;
16    }
```

在默认情况下,全局变量的作用域只在本源文件内部。如果不使用 extern 关键字,就不能实现对全局变量的跨源文件使用。在不同源文件中的全局变量可以重名吗?答案是否定的。虽然在默认情况下全局变量的作用域只在本源文件内部,但是全局变量统一存储在静态区域中。如果全局变量重名,那么系统无法在静态存储区中区分这两个全局变量,因此不同源文件中的全局变量不可以重名。

3. *“四不像”变量——静态局部变量

在程序中,局部变量和全局变量各有优势与劣势。局部变量的作用域局限在一个函数的内部。它是动态存储的,因此内存利用率高。但是当函数调用结束后,局部变量的存储空间就会被释放了,它里面存储的数据也就无法再访问了,这也是它的缺点。

如果在函数调用结束后仍然想保持局部变量中的数据,那么就不能释放局部变量的存储空间。我们可以将局部变量的动态存储模式改为静态存储模式,让它与全局变量的存储模式一样,但是让它的作用域仍然是在函数的内部。**这种作用域局限于函数内部,但是在函数调用结束后仍然不释放存储空间的变量就是静态局部变量**。静态局部变量既具备了局部变量的部分特点,又兼具了全局变量的部分特性,因此它更像一个“四不像”的变量。

我们可以使用关键字 static 在函数内部定义静态局部变量,它的一般格式是:

```
static 数据类型 变量名称;
```

静态局部变量具有以下两个特点:

(1) 它是局部变量,其作用域在函数内部,因此它只能在函数内部定义与使用。

(2) 它采用了静态存储模式。在函数调用的时候,系统在静态存储区为它分配存储空间。在函数调用结束后,系统并不会并收回它的存储空间。当程序运行结束的时候,系统才会收回它的存储空间,这一点与全局变量的内存管理模式相同。

静态局部变量会保存上次函数调用时的数据,在下次函数调用时可以访问该静态局部变量中的数值。至于怎么利用静态局部变量的特性来解决实际问题,那就要看程序员

如何发挥自己的聪明智慧了。

【例 5.16】 分析在函数 addOne 调用过程中静态局部变量 a 的数值变化情况。

程序代码如下：

```
#include <stdio.h>
int main()
{
    void addOne();
    addOne();                                   //第一次调用 addOne 函数
    addOne();                                   //第二次调用 addOne 函数
    addOne();                                   //第三次调用 addOne 函数
    return 0;
}
void addOne()
{
    static int a = 0;                           //定义静态局部变量 a
    a++;
    printf("addOne 函数被调用了 %d 次\n",a);  //输出变量 a 的值
}
```

在 addOne 函数中，语句 12 定义 static int a 静态局部变量。第 5 行语句对 addOne 函数进行了第一次调用。执行语句 12，静态局部变量 a 被初始化赋值 0。继续执行语句 13，变量 a 自增 1。第一次 addOne 函数调用结束后，变量 a 中的值是 1。执行第 6 行语句对 addOne 函数进行第二次调用，此时语句 12 不再执行，因为在第一次 addOne 函数调用时已经对静态局部变量 a 进行了初始化赋值。执行语句 13 后，变量 a 的值自增 1 变为 2。执行语句 7 对 addOne 函数第 3 次调用，变量 a 的值变为 3。

程序执行结果参见图 5.7。

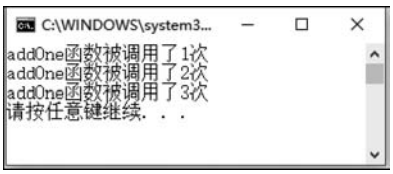

图 5.7 程序运行结果

【例 5.17】 利用静态局部变量实现求 n! 的运算。

程序代码如下：

```
#include <stdio.h>
int main()
{
    int n,i;
    void fac(int);
    scanf("%d",&n);
    if (n<0) printf("n<0,输入数据错误!\n");
    else for (i=0;i<=n;i++) fac(i);             //调用 fac 函数求阶乘
    return 0;
}
void fac(int n)
{
    static int s=1;                             //定义静态局部变量 s
    if (n>0) s=s*n;                             //求 n>1 时,n!
    printf("%d!=%d\n",n,s);
}
```

在第8行代码中有一个循环语句，通过循环调用fac函数求0！到n！。当i=0时，调用fac(0)可以求得0！=1。在fac函数中，通过语句13定义了静态局部变量s用于存储阶乘值，fac(0)调用结束后，静态局部变量s会保存0！值1。当i=1时，调用fac(1)，执行语句14，由于s中保存了0！的值，执行s=s*n，就可以求得1！的值1。也就是说，当fac(n)调用时，s中会保存fac(n-1)调用的结果(n-1)！，通过s=s*n就可以求得n！。

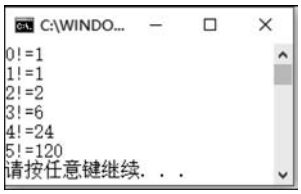

图5.8　程序运行结果

当n的输入值为5时，程序的运行结果参见图5.8。

4. *外部函数与内部函数

根据作用域的不同，变量分为局部变量和全局变量，那么函数有没有类似的作用域问题呢？函数也有作用域的概念。根据函数作用域的不同，函数分为内部函数和外部函数。

1）内部函数

如果规定一个函数只能被本文件中的其他函数所调用，那么该函数称为内部函数。在定义内部函数的时候，需要在函数的首部加入关键字static。

【例5.18】 将fac函数定义为内部函数。

程序代码如下：

```
1   static void fac(int n)
2   {
3       static int s = 1;
4       if(n > 0) s = s * n;
5       printf("%d!= %d\n",n,s);
6   }
```

内部函数fac只能在本源文件中被其他函数调用，它的作用域只限定于本文件。这样在不同的源文件中即使有同名的内部函数，它们之间也不会互相干扰，也不必担心所用的函数是否会与其他文件中的函数同名。

2）外部函数

如果一个函数能够被其他源文件中的函数调用，那么它就是外部函数。在定义外部函数的时候，需要在函数的首部加入关键字extern。

【例5.19】 将fac函数定义为外部函数。

程序代码如下：

```
1   extern void fac(int n)
2   {
3       static int s = 1;
4       if(n > 0) s = s * n;
5       printf("%d!= %d\n",n,s);
6   }
```

这样 fac 函数就可以被其他源文件的函数所调用。C 语言规定 extern 可以省略，也就是说，不加 static 或者 extern 的函数都是外部函数，可以在所有的源文件中进行调用。

在同一个源文件中，如果主调函数在调用的函数后面，则需要在主调函数中对被调用函数进行声明；如果被调用的函数在主调函数前面，则不需要进行声明。在不同的源文件中，函数的相互调用也是如此，需要在主调函数中对被调用函数用关键字 extern 进行声明。这一点与在不同源文件中通过 extern 对全局变量进行声明实现跨文件的全局变量访问的机制是类似的。

5. 两个函数间的“信使”——指针

全局变量可以在多个函数之间传递数据，它破坏了函数的封闭性，不建议过多使用。其实，还有一种方式可以实现在两个函数之间传递数据，那就是利用指针。

通过指针来访问变量是一种间接访问方式。如果要在被调函数中直接访问主调函数的局部变量，这是不可能实现的。但是，我们可以在被调函数中借助变量的地址以间接方式来访问主调函数的局部变量。将主调函数中局部变量的指针作为实参传递给被调函数的形参变量，在被调函数中通过指针运算操作——*（局部变量指针），可以访问到主调函数中的局部变量，从而实现从被调函数向主调函数传递数据。

【例 5.20】 定义 change 函数，在 change 函数中实现对 main 函数中局部变量的访问。

问题分析：在 main 函数中定义局部变量 int x。在 change 函数调用时，将变量 x 的指针值 &x 作为 change 函数的实参传递给它的形参变量，通过“*（形参变量）”的表达式间接访问 x 变量并对变量 x 进行赋值操作。change 函数的形参变量存储了变量 x 的地址，因此形参变量的数据类型是指针类型 int *。change 函数的声明如下：

```
void change(int *)
```

程序代码如下：

```
1   #include <stdio.h>
2   int main()
3   {
4       void change(int *);                    //对 change 函数声明
5       int x=1;
6       change(&x);                            //调用 change 函数
7       printf("%d\n",x);
8   }
9   void change(int* y)
10  {
11      *y=2;                                  //间接访问局部变量 x
12  }
```

下面通过程序调试的方式，跟踪局部变量 x 和 y，观察它们的值的变化过程。在第 6 行、第 7 行和第 12 行语句的前面分别插入断点，利用 debug 调试运行至第 6 行语句处，

此时 x 和 y 中的状态参见图 5.9。

图 5.9　在 change 函数调用前变量的值

从监视窗口中可以观察到变量 x 的值是 1。由于 change 函数还未调用，因此该函数中局部变量 y 还未分配存储空间，系统提示“没有找到符号'y'”。继续执行至下一个断点，程序调用 change 函数，此时 x 和 y 的状态参见图 5.10。

图 5.10　change 函数调用时变量的值

从监视窗口中可以观察到变量 y 的值是 0x0053fbb0，这是变量 x 的地址，该地址对应变量的值已经修改为 2，也就是变量 x 的值已经变为 2。当 change 函数调用完毕后，x 和 y 的状态参见图 5.11。

此时局部变量 x 的值为 2，局部变量 y 已经不可以再访问。

这个例子验证了以下两个事实：

(1) 函数只有在被调用的时候，它的局部变量才会被分配存储空间。当函数调用结束后，局部变量的存储空间会被系统收回。

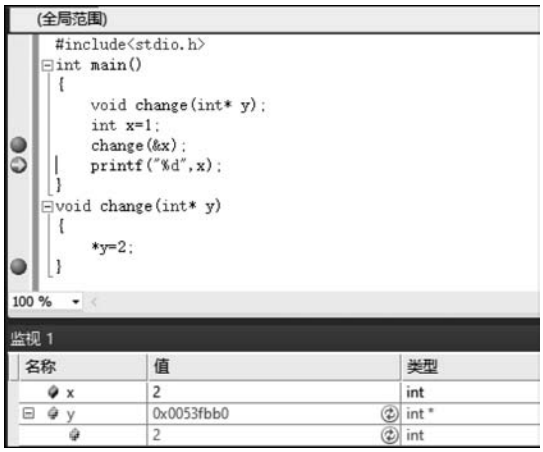

图 5.11　change 函数调用结束后变量的值

(2) 如果被调函数通过形参变量获得了主调函数的局部变量的指针,那么在被调函数内部可以通过间接访问变量的指针运算符 * 对该局部变量进行访问,从而实现被调函数向主调函数传递数据。

虽然利用指针的间接访问变量方式,可以改变主调函数中局部变量的值,但是它并没有违反函数"形参无法改变实参"的准则。在上面的例子中,形参变量 y 的数据类型是 int * ,实参是变量 x 的地址值 &x,改变 y 的值不会影响到 &x 的值。我们虽然通过局部变量 y 改变了局部变量 x 的值,但是没有违背函数的形参值无法影响实参值的规定。因为在这个例子中,形参是 y,实参是 &x,即变量 x 的地址,而不是 x。这一点,大家一定要清楚。

在 5.2.1 节中,介绍函数形参的改变无法改变实参值的时候,曾经举例说明通过 Exchange 函数交换两个形参变量的值无法交换 main 函数中两个实参变量 x 和 y 的值。现在可以通过利用指针在 exchange 函数中实现对 main 函数中变量 x 和 y 值的交换。

【例 5.21】 定义 exchange 函数,利用指针实现对 main 函数中两个局部变量 x 和 y 的值的交换。

问题分析:将变量 x 和 y 的地址作为实参传递给 exchange 函数的形参。在 exchange 函数中通过"*(指针)"运算实现对 main 函数中局部变量 x 和 y 的访问,从而交换它们的值。exchange 函数形参的类型是指针类型,它的功能是交换数据,不需要返回数据值,因此它的函数类型是 void 类型。

程序代码如下:

```
#include<stdio.h>
int main()
{
    void exchange(int*,int*);          //对 exchange 函数声明
```

```
5       int x = 1, y = 2;
6       exchange(&x, &y);                    //将 x 和 y 的地址传到 exchange 函数中
7       printf("x = %d, y = %d\n", x, y);
8   }
9   void exchange(int * a, int * b)
10  {
11      int c;
12      c = * a;                             //通过 * a 间接访问变量 x
13      * a = * b;                           //通过 * b 间接访问变量 y
14      * b = c;
15  }
```

第 4 行语句是对 exchange 函数的声明，该函数的两个形参的数据类型都是 int * 。第 6 行语句是对 exchange 函数的调用，它的实参分别是 &x 和 &y，将变量 x 和 y 的地址传递到 exchange 函数中(此处一定要注意，函数 exchange 调用时的实参并不是 x 和 y 两个变量的值)。在第 11 行语句中定义了 int 变量 c 作为交换变量 x 和 y 值的中间变量。第 12 行语句通过 * a 间接访问变量 x，并将变量 x 的值赋值给变量 c。第 13 行语句通过间接访问方式将变量 y 的值赋值给变量 x。语句 14 将 c 中存储的 x 的原值赋值给变量 y。执行第 7 行语句输出 x 的值为 2，y 的值为 1。

利用指针实现两个函数之间的数据传递是指针的重要用途之一。当然，这种方式破坏了函数的封闭性，一般不推荐使用。但是为了提高内存的利用效率，有时又不得不用，这一点在学习数组知识的时候再进行讨论。

5.4 函数举例

本节通过两个例子来深入学习，如何像搭积木一样利用函数来编程求解问题。

5.4.1 求三角形的面积

【例 5.22】 给定平面上的 3 个坐标点，求 3 点所围成的三角形的面积。

问题分析：

第 2 章中我们给出三角形 3 条边长，利用下面的海伦公式，编程求解了三角形的面积。

$$s=\sqrt{p(p-a)(p-b)(p-c)}$$

其中，p 为半周长，即 $p=(a+b+c)/2$。

如果只知道三角形的 3 个顶点坐标，也可以求出三角形的面积。3 个顶点所围成的三角形的边的长度可以通过两点间的距离公式求得。

$$d=\sqrt{(x1-x2)^2+(y1-y2)^2}$$

$(x1,y1)$和$(x2,y2)$表示两个点的坐标，d 是两点间的距离。

因此给定三角形的3个顶点($x1$,$y1$),($x2$,$y2$)和($x3$,$y3$),可以通过先求3条边的长度,再求出三角形的面积。假如把求距离和求面积的公式看成函数模块,这些函数模块该如何设计与组装呢?参见图5.12。

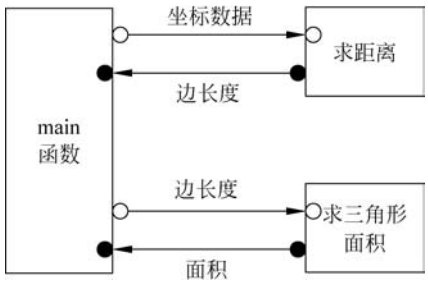

图5.12 求三角形面积的函数模块组装

下面首先实现求两点间距离的函数distance。虽然需要求3条边的长度,但是求边长的代码写一个函数就可以了,这也是代码可复用性的体现。

程序代码如下:

```
1   float distance(float x1,float y1,float x2,float y2)
2   {
3       float d = sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2));
4       return d;
5   }
```

下面根据海伦公式实现求三角形面积的函数triangleArea。

程序代码如下:

```
1   float triangleArea(float a,float b,float c)
2   {
3       if((a + b)> c && (a + c)> = b && (b + c)> a)        //判断两边之和大于第三边
4       {
5           float p = (a + b + c)/2.0;
6           float s = sqrt(p * (p - a) * (p - b) * (p - c));
7           return s;
8       }
9       else return  - 1;
10  }
```

在main函数中输入3个点的坐标,通过调用distance和triangleArea函数完成模块之间的组装,实现求一个具体的三角形面积。

程序代码如下:

```
1   int main()
2   {
3       float x1,y1,x2,y2,x3,y3;
4       float a,b,c,s;
5       scanf(" % f  % f  % f  % f  % f  % f",&x1,&y1,&x2,&y2,&x3,&y3);
6       a = distance(x1,y1,x2,y2);
7       b = distance(x1,y1,x3,y3);
8       c = distance(x2,y2,x3,y3);
9       s = triangleArea(a,b,c);
10      if (s!= - 1) printf("输入的3个点所围成的三角形面积为: % f\n",s);
11      else printf("输入的坐标不是合法的三角形的顶点坐标。\n",s);
12      return 0;
13  }
```

在上面的代码中，先分别求得三角形的 3 条边 a、b、c 的值，然后通过参数传递给 triangleArea 函数去求面积。其实也可以将 distance 函数的返回值直接作为参数传给 triangleArea 函数。

程序代码如下：

```
1     int main()
2   {
3       float x1,y1,x2,y2,x3,y3;
4       float s;
5       scanf("%f %f %f %f %f %f",&x1,&y1,&x2,&y2,&x3,&y3);
6       s=triangleArea(distance(x1,y1,x2,y2),distance(x1,y1,x3,y3), distance(x2,y2,x3,y3));
7       if (s!= -1) printf("输入的3个点所围成的三角形面积为:%f",s);
8       else printf("输入的坐标不是合法的三角形的顶点坐标。\n",s);
9       return 0;
10  }
```

从这个例子可以看出，先进行功能模块划分，然后将不同的功能用函数来封装，最后通过函数调用和参数传递等形式可以把简单的程序功能组装起来形成复杂的程序功能。

5.4.2 利用函数实现简单的文件操作

第 3 章介绍了文件的基本概念以及操作文件的一些基本函数。在一些更复杂的文件操作中，我们常常将对文件的一些操作包装成函数来使用。

【例 5.23】 统计一个文本文件里不同种类字符的数量，包括字母、数字、空格和其他字符的个数，并将统计信息写到另一个文本文件中，格式如下：

```
Letters:10
Numbers:5
Spaces:6
Others:7
```

我们可以通过调用库函数的方式，将所有的代码都写到 main 函数中，但是这样做代码不够简洁。可以利用模块化的思想，将读取源文本文件和统计文件内不同类型字符个数的功能用一个函数实现，函数命名为 fileResult。另外，将统计结果写到另一个文件的功能用 printResult 函数实现。在 main 函数内调用这两个函数，这样可以使得 main 函数的代码比较简洁。

下面首先分析这两个函数的输入输出数据。fileResult 函数需要获得文本文件的路径，除此之外不需要其他输入，它还需要统计 4 种不同类型字符的数量，因此该函数需要输出 4 个数值，而通过 return 语句无法返回 4 个数值。根据 5.3 节的内容，可以知道有两种方式解决这个问题：第一种是利用全局变量作为信使，实现在多个函数之间传递数据；第二种是使用指针作为参数，实现在两个函数之间传递数据。这里选择使用第一种

方案，定义 4 个全局变量，分别为 letters、numbers、spaces 和 others，作为记录不同类型字符数量的计数器，类型均为整型，初始值为 0。

printResult 函数同样需要获得另一个文本文件的名称，它还需要获得由 fileResult 函数统计出的不同类型字符的数量值。由于 letters、numbers、spaces 和 others 是全局变量，所以 printResult 函数也可以访问和使用这些变量，不需要通过参数传递的方式输入这些数据。

fileResult 函数和 printResult 函数的具体实现如下：

第一步，fileResult 函数需要利用获得的文件名打开文件。因为需要统计每个字符的类型，所以可以建立循环结构，利用 fgetc 函数获取文件内的每个字符，再利用分支结构判断每个字符的类型，对相应字符类型的计数器执行加 1 操作，直到读到文件尾部。

第二步，当统计结束以后，利用 printResult 函数将统计结果按照输出格式，利用 fprintf 函数写入到文件中即可。

在 main 函数内通过调用 fileResult 和 printResult 函数来组织代码。剩下的工作就是具体实现每一个函数，这里又需要用到之前学习过的文件读写、循环、分支等知识。

程序代码如下：

```
#include <stdio.h>
#include <stdlib.h>
int letters = 0,numbers = 0,spaces = 0,others = 0;      //定义全局变量
int main()
{
    void fileResult(char *fileName);
    void printResult(char *fileName);
    char *fn = "text.txt", *pn = "result.txt";
    fileResult(fn);
    printResult(pn);
    return 0;
}
void fileResult(char *fileName)                         //打开文件并进行统计
{
    char ch;
    FILE *fp;
    if ((fp = fopen(fileName,"r")) == NULL)
    {
        printf("无法打开此文件\n");
        exit(0);
    }
    while(!feof(fp))
    {
      ch = fgetc(fp);
      if(ch >= 'a' && ch <= 'z'||ch >= 'A' && ch <= 'Z') letters++;
```

```
        else if (ch >= '0' && ch <= '9') numbers++;
            else if(ch == ' ') spaces++;
                else others++;
    }
    fclose(fp);
}
void printResult(char *fileName)                    //结果写到 result.txt 中
{
    FILE *fp;
    if ((fp = fopen(fileName,"w")) == NULL)
    {
        printf("无法打开此文件\n");
        exit(0);
    }
    fprintf(fp,"letters: %d\n",letters);
    fprintf(fp,"numbers: %d\n",numbers);
    fprintf(fp,"spaces: %d\n",spaces);
    fprintf(fp,"others: %d\n",others);
    fclose(fp);
}
```

虽然上面的代码比较长,但是 main 函数的代码比较简短,只定义了两个指向文件名的字符指针,然后调用了 fileResult 和 printResult 函数。在该例中,利用全局变量作为函数之间数据传递的信使,实现了在函数之间交互多个数值。

5.5 本章小结

函数是一种有效的组织程序代码的方法。本章从模块化设计的角度重点介绍了如何定义函数和调用函数。在函数使用过程中,要重点关注函数调用过程的参数传递以及局部变量和全局变量的使用问题。

模块化思想是C语言程序设计的一种重要思想。在开发C程序的过程中,利用模块化的思想对程序的功能进行划分,通过函数的方式对模块的功能进行实现与组装,极大地提高了程序的开发和维护效率。针对代码模块的封闭性、重用性和组装性要求,对函数中可以使用的变量的作用域进行了限定,从而产生了局部变量和全局变量。局部变量只能在一个函数内部使用,它很好地体现了函数的封闭性,而全局变量则可以在多个函数中使用,它破坏了函数的封闭性,但提高了函数之间数据交互的灵活性。利用指针通过间接访问变量方式,可以实现两个函数之间的数据交互,但它破坏了函数的封闭性,也带来了一定的灵活性。只要掌握了模块化思想,学会了编写函数的本领,就可以轻松地应对大型复杂程序的开发问题!本章知识点参见图 5.13。

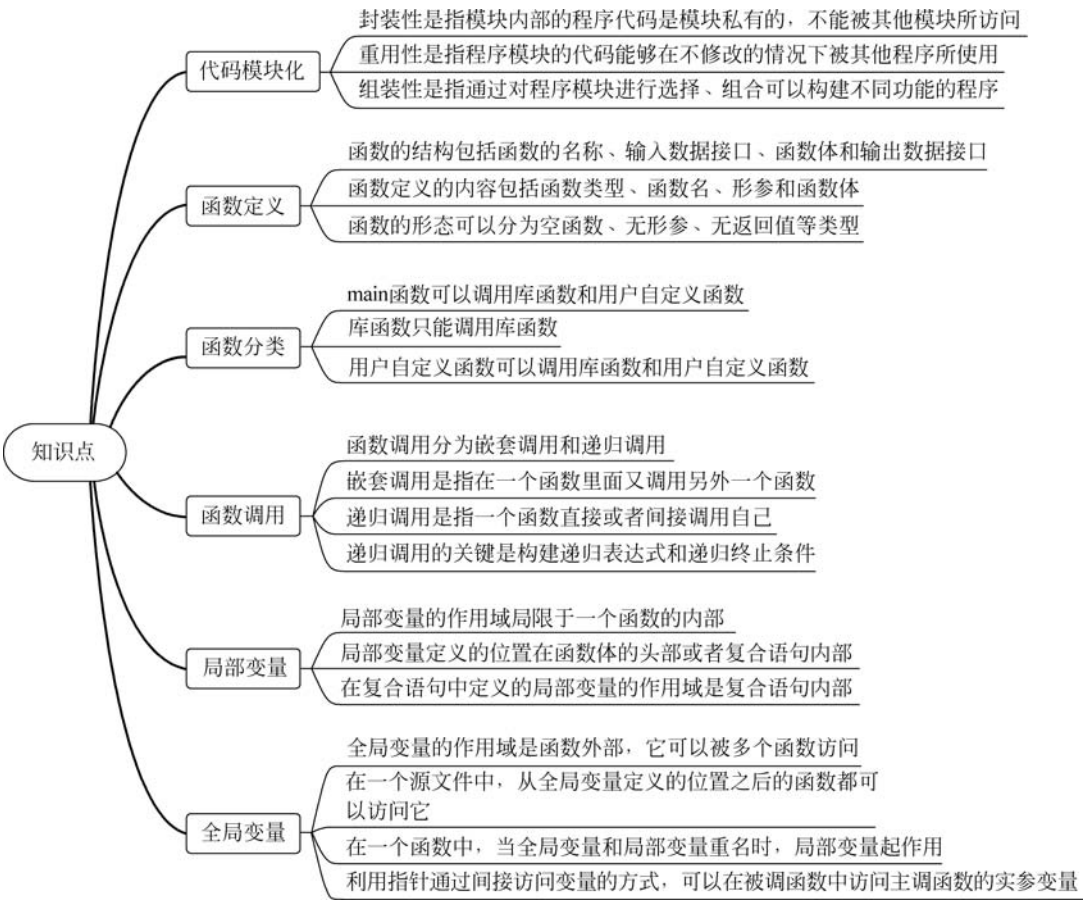

图 5.13　像搭积木一样搭建程序

5.6　习题

1. 请联系生活中的例子，描述一下模块化思想。

2. 简要介绍库函数的作用，并举例介绍库函数的使用方法。

3. 简要介绍形参与实参的概念，并说明它们之间数据的传递过程。

4. 实现一个判别输入正整数是否为素数的函数。比如当函数的输入数据是“17”时，函数的输出为“1”，表示该数为素数；如果输入“20”，则输出为“0”，表示该数为合数。

5. 根据摄氏温度和华氏温度的转换关系

$$C = \frac{5 \times (F - 32)}{9}$$

其中，C 表示摄氏度，F 表示华氏温度，实现一个从华氏温度到摄氏温度转换的函数。

6. 利用 5.4 节中求距离和三角形面积的函数，实现一个求任意凸四边形面积的函数，函数的参数为 4 个顶点的坐标值。

7. * 利用第 6 题的方法，设计求任意凸多边形面积的函数，可以将多边形的顶点坐标利用数组进行存储，函数以数组作为参数。数组相关的知识参考第 6 章。

8. 利用递归的思想实现逆序输出整数。例如，设计一个函数 reverse，函数的输入为一个正整数，比如 123456，通过 reverse 函数输出为 654321。

9. 汉诺塔问题求解。印度神话中有一个关于汉诺塔的故事，汉诺塔内有 3 个柱子 A、B、C，开始的时候 A 柱上有 64 个圆盘，盘子大小不等，大的在下，小的在上。有一个婆罗门想把圆盘从 A 柱上挪到 B 柱上，但是一次只能移动一个，并且要保证大盘在下，小盘在上，移动中可以利用 C 柱子。试编程求解移动的步骤。这是一道必须使用递归方法才能解决经典的问题。即使是用计算机来模拟移动过程，也需要很长时间。在编程的时候，可以只移动 7 个圆盘。

10. * 利用函数实现复数的简单运算。复数的形式是 a+bi 的形式，其中 a 和 b 都为实数，a 称为复数的实部，bi 称为复数的虚部。复数同样有加减乘除四则运算。现在考虑设计一个简单的复数运算的函数，比如复数的加法 ComplexAdd，假设将该函数定义为

```
double ComplexAdd(double a,double b,double c,double d)
```

其中，a 代表第一个复数的实部，b 代表第一个复数虚部的系数，c 代表第二个复数的实部，d 代表第二个复数虚部的系数。

试问：该函数的定义能否返回两个复数相加的结果？如果这种定义方式不便于实现，请参考第 6 章数组和第 7 章结构体的内容，设计一个可以满足需求的函数。

11. 设计一个函数计算两个日期相隔多少天。函数的参数可以为 6 个，分别代表起始的年、月、日和截止的年、月、日。注意，在计算的过程中，可能需要多次计算某一个年份是否为闰年，所以可以首先设计一个判断某年为闰年的函数。