6.2 函数的声明和定义

实际上，在前面的章节中，读者已经接触到了函数，例如 main()和 printf()等。那么，什么是函数呢？C 语言的函数（*function*）是一个由多条语句聚合而成的、在功能上相对独立的语法单位，具有高度可重用的特性。很多情况下，一个 C 函数要完成相关的运算，并得到运算结果。在这一点上，C 函数与数学上的函数非常相似。不过，C 函数的功能不止于此。除了运算，C 函数还可以处理一些事务性工作，从而支持更复杂的应用需求。

C 函数不仅在功能上是独立的，同时在书写形式上也是独立的，有特定的语法支持。下面就来详细讨论一些关于函数的话题。

6.2.1 函数的分类

在深入讨论函数之前，首先来看看 C 函数的分类情况。根据函数的提供者不同，C 函数主要分为**库函数**（*library functions*）和**用户自定义函数**（*user defined functions*）两类。

1．库函数

前面学习的 printf()、scanf()等都是库函数。这类函数是 ANSI C 标准中预定义的库函数，完成一些程序员们最常用到的功能。任何一个 C 编译器的供应商都会预先编写好这类函数，并将它们分门别类地封装在一些库文件（后缀常为.lib）中，然后作为 C 语言产品的标准配件随编译器一起提供给程序员。程序员可以直接使用库函数而不必关心这些函数是如何实现的。

为了使程序员能够使用库函数，编译器产品提供了一些预先编写好的标准头文件，用来描述库函数的外部特性。而在程序员自己编写的程序中，可以根据需要用#include 指令包含相应的标准头文件。例如：printf()和 scanf()的外部特性描述是放在标准头文件 stdio.h 中的，所以很多 C 程序的第一行都是#include <stdio.h>。

2．用户自定义函数

库函数只能完成一些常用功能，但不能完成所有的功能。因此，程序员常常根据需求去编写那些能完成特定功能的函数。这些函数就是“用户自定义函数”。可以这么说，编写一个 C 程序实际上就是在编写一系列用户自定义函数。

接下来的重点是学习如何自定义函数。

6.2.2 函数原型的声明

C 语言中的变量必须先声明后使用，否则编译器不知道这个变量是什么类型，可以用在哪里。函数也是一样，必须在使用前声明其**原型**（*prototype*）。

函数的**原型声明**（*prototype declaration*）具有如下形式：

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

通过这个声明，程序员告知编译器以下事宜：

（1）函数的名字。函数名是函数的唯一标识，它的命名规则和变量的一样。函数名后必须跟上一对圆括号，因为那是函数的标志。为函数取一个有意义的名字对程序的理解有很大的帮助。C 语言不允许有两个及两个以上的函数重名。

（2）函数的参数（6.3.1）。原型中明确说明了该函数有几个参数，每个参数的类型又是什么。参数类型可以是任意合法类型。如果函数没有参数，那么圆括号中的内容可以空缺，或者写上 void。参数的名字在原型声明中是一个可选项。但笔者强烈建议读者为参数起一个有意义的名字，以增加程序的可读性。

（3）函数的返回值类型（6.3.2）。函数一般都会有一个计算结果，而这个结果就是函数的返回值，其类型就是函数的返回值类型，可以是任意合法的类型。如果函数没有返回值，那么返回类型应该设定为 void。

旧式的 C 程序允许函数返回值类型空缺。在这种情况下，编译器规定该函数的返回值类型是 int。旧式 C 程序的 main()函数的返回值类型也常是 void。而现代的 C 程序要求每一个函数都必须指明其返回值类型。特别地，main()函数应该返回 int 型值。

例如一个用于求两数之和的加法函数的原型可能是这样的：

```
int add(int a, int b);
```

有了上述声明，那么编译器在其后的代码中会根据这个声明严格检查该函数的使用情况。一旦发现有使用不对的地方，编译器就会报出警告或错误，提醒程序员去核查代码。

前向引用概念：函数的参数（6.3.1），函数的返回值（6.3.2）

6.2.3 函数的定义

函数原型声明只是给出了函数的规格说明，没有给出函数的实现部分，因此还不能使用。给出函数实现的过程就是函数定义（*function definition*），其语法如下：

```
返回值类型  函数名(类型 参数名1[, 类型  参数名2,…]) //函数头部
{
    函数体
}
```

函数定义与函数声明不同之处在于：

（1）函数定义必须给出实现部分，也就是函数体。函数体是语句的组合，它们被放在一对{}之间。而函数声明只有函数头部信息而没有实现部分。

（2）函数定义时必须给出参数的名字（如果有的话），否则该参数将无法使用。而函数声明中的参数名是可省略的（虽然不建议这么做）。

（3）函数定义可以放在该函数的使用函数（称为“函数调用者 *caller*”）之后。在这种情况下，该函数的声明就必须放在 caller 之前。

（4）实际上，函数定义包含了函数声明。因此，如果函数定义放在 caller 之前，那么函数声明是可以省略的。

函数定义的头部信息必须与该函数的原型声明完全一致。否则，编译器会认为程序员在试图设计两个不同的函数，这会引起一些错误。例如，编译器会认为以下两个函数不同。

```
int f(int x); //声明
int f() { … } //定义
```

如果没有给出第一个函数的函数体，那么在链接时将会出现错误。

【例 6-1】函数的声明和定义：求两个数的平均值。

```
//6-1.c
#include <stdio.h>

double average(double x, double y); //函数声明

int main()
{
    double a = 4.0, b = 7.0;
    double avg;

    avg = average(a, b);

    printf("%.2lf\n", avg);

    return 0;
}
```

```
double average(double x, double y) //函数定义
{
    return (x + y) / 2.0;
}
```
程序的输出是：
```
5.50
```
C 语言规定，一个函数的定义必须放在其他所有函数之外，就像[例 6-1]中所示那样。而在函数中定义另一个函数的语法是错误的。

下面的代码就是错误的：
```
void f()
{ int g() {…} }//error。g()的定义必须放在 f()之外
```
虽然函数的定义不能嵌套，但是在函数中却可以包含另一些函数的原型声明。例如：函数 g()的声明可以放在 f()内部。
```
void f()
{ int g(); } //ok，函数原型声明
int g() {…}
```

6.2.4 函数类型

一旦在程序中声明或定义了一个函数，那么同时也定义了一种函数类型。例如有声明：
```
double average(double x, double y);
```
那么这条声明不仅说明了一个函数的名字，同时也说明 average()函数属于以下类型：
```
double (double, double)
```
函数类型是一种派生类型，由函数的返回值类型和参数个数及类型来决定。属于本例中的函数类型的函数要带两个 double 类型的参数，以及返回一个 double 类型的值。

如果再有声明：
```
double abs(double x);
```
那么函数 abs()和函数 average()的类型是不同的，因为 abs()的类型是：
```
double (double)
```
显然二者的类型是不一样的。

函数类型的定义一般伴随函数的声明或函数的定义完成，不需要从中单独分离出来。以下的函数声明是错误的：
```
double (double x) abs;
```

6.3 函数的调用

一旦定义好一个函数，那么就可以在需要的地方使用它。

函数的使用称为**“函数调用**（*function call*）**”**；函数的使用者（也是函数）称为**“主调函数**（*caller*）**”**；被调用的函数称为**“被调函数**（*callee*）**”**。例如在[例 6-1]中，
```
int main() //在主调函数中调用 average()
{   …
    avg = average(a, b); //callee
    …
}
```
此例中，callee（即 average）要计算出一个结果，此结果被返回到 caller(即 main)中，

并被保存到了 caller 中的一个变量 avg 中。

在 caller 中调用 callee 的地方称为**调用点**，同时也是 callee 的**返回点**。

函数调用时有一些需要关注的地方，在下面的内容中将会详细讲解。

6.3.1 函数的参数

函数往往需要从 caller 那里接收一些待加工的数据。这些数据就是函数的**参数**（*parameter/argument*）。参数可分为**形式参数**（简称形参，*formal parameter*）和**实际参数**（简称实参，*actual parameter*）。

1．形参

形参是指函数声明或定义时出现的参数，是 caller 和 callee 之间数据传递的媒介。例如，在函数头部

```
double average(double x, double y)
```

中，x 和 y 就是形参。形参具有以下特点：

（1）每一个形参都必须指定其数据类型。

（2）在函数的定义中，形参只是一些符号，没有具体的值。形参的作用是告诉编译器，该形参出现在函数中的什么地方，可以参与哪些运算，起到一种占位的作用。一但发生函数调用，那么形参占据的位置（的值）将会被实际的值取代，从而完成值运算，得到确切的结果。

类似于如下的函数声明

```
double average(double x, y);
double average(x, y);
```

在语法上是错误的。要知道，每一个形参都必须指定其数据类型。

2．实参

实参是指函数调用时 caller 传递给 callee 的真实数据。例如，在 main()中的函数调用

```
avg = average(a, b)
```

中，变量 a 和 b 就是实参。实参的特点是：

（1）实参可以是常量、变量或表达式，甚至是另一次函数调用的返回值；

（2）实参无论是数量、数据类型、出现顺序都必须与形参的严格一致。例如

```
函数声明：double average(double  x, double y)
                         ↓          ↓         ↓
调用时：     avg = average(         a,        b);
```

其中，变量 avg、a、b 的类型都是 double。

（3）在调用时不需要指定实参的类型。例如，调用 average(double a, double b)是错误的。不过，实参的类型应该与形参的相同，至少是相容。例如，如果形参的类型是 double，那么实参的类型可以是 double、float 以及所有的整数类型。

3．实参与形参的结合

实参向形参传递值的过程称为实参和形参的结合。C 语言采用传值调用（*call by value*）的结合方式，其含义是：实参只将其内容复制给形参；在函数中，形参代替实参参与运算。也就是说，在函数调用时，实参和形参是两个不同的内存单元，形参的改变不会影响到实参。

【例 6-2】实参与形参的结合：形参的改变不影响实参。

```
//6-2.c
#include <stdio.h>

void f(int x)
{
    ++x;
    printf("in f(): x=%d\n", x);
```

```
}

int main()
{
    int a = 1;

    printf("before: a=%d\n", a);
    f(a);
    printf("after: a=%d\n", a);

    return 0;
}
```

程序的结果如下。

```
before: a=1
in f(): x=2
after: a=1
```

从结果中可以清楚地看到，实参 a 将其值传递给了形参 x，因此 f()内部的运算结果为 2。而在函数调用前后，实参 a 的值保持不变，如图 6-1 所示。

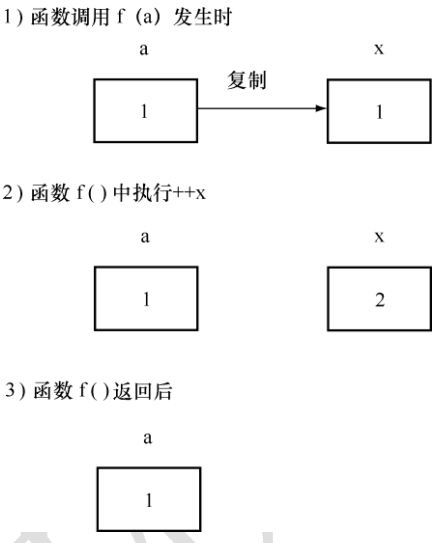

图 6-1　函数形参的改变不影响实参

6.3.2　函数的返回值和 return 语句

与数学上的函数一样，多数的 C 函数也会计算出一个结果。这个结果就是函数的返回值。这个返回值是有类型的，可以是任意合法的 C 数据类型。而这些信息都必须在函数头部进行说明。例如：

```
double average(double x, double y)
```

说明 average()会得到一个 double 类型的结果。这个运算结果一般使用 return 语句返回。return 语句的语法为：

```
return [表达式];
```

return 语句是一种无条件转移语句，其功能是立即结束函数的执行，并将表达式的结果返回到 caller 中。

当然，不是所有的 C 函数都会进行有结果的运算，函数该不该有返回值应视具体的需要来定。在无返回值的情况下，return 语句后面不应该有任何类型的表达式出现。

按函数的运算结果，可以有无返回、有一个或多个返回结果的情况。

（1）无返回结果

如果一个 C 函数完成的是一般事务处理，不需要有结果；或者结果已在这个函数中处理，那么这个函数也可以不需要返回值。这种情况下，函数的返回值类型应设为 void。例如【例 6-2】中的

```
void f(int x)
```

无返回值的函数可以没有 return 语句，编译器会自动在函数的尾部添加一条无表达式的 return 语句。例如【例 6-2】中的函数 f()。

（2）有一个待返回结果

多数情况下函数会得到一个结果，用于指示函数的运行状况。此时，需要用 return 语句将这个结果返回到 caller 中。在这种情况下，caller 中会有一条赋值语句来接收函数的返回值，然后根据这个返回值来进行下一步的处理。在【例 6-1】中可以看到这种情形。

（3）有多个待返回结果

return 语句只能向 caller 返回一个结果。如果一个函数因特殊原因产生了多个待返回结果，那么就不能使用 return 语句来完成这项工作。在这种情况下，可以采用特殊的数据类型使形参具有携带计算结果的能力的方法。后面章节中要讲到的数组和指针都具有这样的能力。

return 语句标志着函数的出口。很多情况下，函数只有一个出口，并且是在函数所有的代码执行完以后。但在某些特殊的场合，函数已经得到一个确定的结论，不需要再执行下去，那么就可以在适当的位置用 return 语句返回。这样一来，函数就有可能存在多个出口。例如

```
int max(int x, int y) //返回 x 和 y 中最大的那个
{
    if (x > y)
        return x;
    else
        return y;

    return 0;
}
```

在上面的代码中，函数 max()有 3 个出口。一旦 x>y 成立，那么其后的语句都不会被执行，函数立即返回。由于 if 语句的各个分支覆盖了所有可能的情况，因此 return 0；这条语句永远也不会被执行到。

> 一般情况下，函数中段的 return 语句都会包含在一条条件语句当中，否则函数会无条件返回，这可能并不符合程序员的初衷。在编写多出口函数的时候应该注意如下事项。

> 1. 一个函数的每一条执行路径都应该有出口；
> 2. 一个函数中的每一条 return 语句都应该有返回表达式，或者都没有。一些有而一些没有是错误的。

6.3.3 函数的调用过程

当调用一个函数的时候，要在 caller 中向 callee 传递参数，如[例 6-2]中的做法

```
f(a);
```

这里，a 是实参，其值为 1。函数 f()的形参是 x，在开始前值未定，而一旦调用开始，那么 x 将获得 a 的值，也就是 1。此后，f()函数正式开始执行。

C 函数的调用过程是一种“停–等”过程。当 caller 执行到调用 callee 的地方（称为“调用点”），caller 就会暂停执行，流程控制权将会转移到 callee 的入口处，开始执行 callee 的代码。一旦 callee 执行完毕，或者执行到了一条 return 语句，那么 callee 执行结束，流程控制权再次转移，回到 caller 的调用点（也就是返回点），caller 恢复执行。此时，caller 的执行将从调用点开始继续往下。图 6-2 示意了这种调用过程，其中的箭头指明了程序的流向，序号是执行顺序。

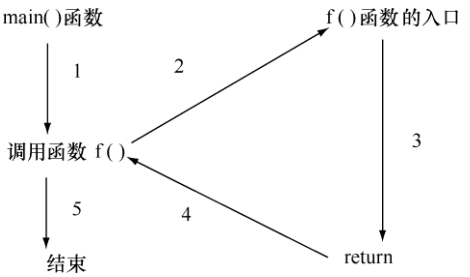

图 6-2　函数的调用过程

6.3.4 函数的嵌套调用

每一个 C 函数都完成相对独立的功能，这样一来，程序员可能会编写较多的函数来完成不同的功能。这就有可能形成这样的格局：函数 a 调用函数 b，而函数 b 又调用函数 c。这种格局称为“函数的嵌套调用”。

函数的嵌套调用也遵循“停–等”协议。图 6-3 示意了这种情况。

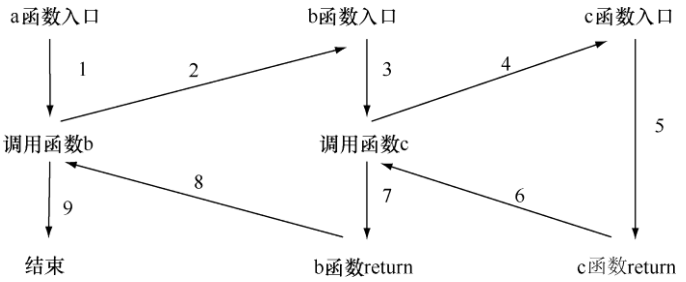

图 6-3　函数嵌套调用的流程

函数嵌套调用的一种非常特殊的形式是递归（*recursion*）（6.7），也就是函数自己调用自己。在 6.7 中将会详细讨论这种情况。

> 前向引用概念：函数递归（6.7）

6.4　函数的设计

每一个复杂的 C 程序都会遇到如何设计函数的问题。那么，何时该使用函数呢？有一种简单的规则，即如果在主函数中发现：

（1）一段语句序列构成了一个功能相对独立的模块

（2）这个模块会被重复使用。

那么，这些语句就应该构成一个函数。

一旦一个函数被确定下来，那么在设计这个函数的实现时，需要弄清楚以下几件事：

（1）函数要实现什么功能？

（2）函数有哪些输入数据（形参）？

（3）函数有哪些输出数据（返回值）？

（4）函数功能怎样实现？

首先来简要回答第一个问题。在前面已经提到过，结构化程序设计的模式是自顶向下、逐步求精。当设计一个程序时，需要将求解问题逐步分解成能解决局部问题的更小的功能模块，直至每一个局部模块都不能再分为止。而这些最小化的模块就对应一个个的函数。这里有一个重点，就是功能最小化。尽量不要设计一些大而全的函数，让每一个函数仅担负起最小的责任是明智的，因为这非常有助于程序的维护。

在明确了做什么以后，接下来就是确定函数的参数和返回值，即确定 caller 和 callee 之间要交换的数据。一旦解决这个问题，就可以为函数设计算法了。关于算法设计的问题这里就不再赘述了，这些内容会在《数据结构与算法》这门课中学习到。

一旦功能模块从主函数中分离，那么主函数所完成的工作一般就是输入、输出，并调用相应的函数对数据进行加工。

这里用一个实例来说明如何设计函数。

【例 6-3】计算 $1^k+2^k+\ldots+n^k$ 的值，其中 *n* 和 k 都是不太大的正整数。

【解题思路】很明显，从题目中可以看到两个明显的功能：累加和乘方。因此，应该设计两个函数来分别完成这两项功能。接下来分析两个函数该如何设计。

（1）累加

首先，为这个函数命名为 sigma。其次，这个函数需要从主函数那里接收两个参数：累加项数 n 和每项的幂次 k。这个函数要向主函数返回累加结果。因此，函数的原型应该是这样的。

```
long sigma(int n, int k);
```

累加的算法比较简单，就是使用一个循环：

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

其中 a_i 是级数中的第 i 项，它是 i 和 k 的函数（即 $a_i=i^k$）。当循环结束后，应该向主函数返回 sum 的值。

（2）乘方

首先，为乘方函数命名为 power。其次，该函数需要从 caller（即 sigma）中接收两个参数：底数 i 和次数 k。这个函数要向 sigma 返回乘方结果。因此，函数的原型应该是这样的：

```
long power(int i, int k);
```

乘方的算法也比较简单：

```
for (p = 1, j = 0; j < k; ++j) p *= i;
```

循环结束后，power()应该向 sigma()返回计算结果 p。

至此，两个最重要的函数设计完成，而剩下的输入、输出和初始化工作就交给主函数 main()来完成。main()的工作步骤是：

（1）输入 n 和 k；

（2）调用 sigma()；

（3）输出计算结果。

最后的工作就是为设计编码，以下是完整的代码。

```
//6-3.c
#include <stdio.h>

long power(int i, int k)            //计算 i 的 k 次方
{
    long p = 1;
    int j;
    for (j = 0; j < k; ++j) p *= i;
    return p;
}

long sigma(int n, int k)          //计算 n 的 k 次方之和
{
    long sum = 0;
    int i;
    for (i = 1; i <= n; ++i) sum += power(i, k);
    return sum;
}

int main()
{
    int n, k;
    scanf("%d,%d", &n, &k);
    printf("1 至%d 的%d 次方之和=%d\n", n, k, sigma(n,k));
    return 0;
}
```

运行结果如下：

```
9,6↙
1 至 9 的 6 次方之和=978405
```

现在要对程序进行测试。以下是对边界数据的测试结果：

```
5,0↙
1 至 5 的 0 次方之和=5
```

显然这个结果是正确的。

然而，这并不表明代码是完全正确的。在输入 n 和 k 的时候，用户可能不清楚对二者的限定

范围，可能输入的数据是负值，或者太大。因此，一个常用的方法就是在输入时检测用户的输入，以免出现异常。可以把 main()函数改为如下：

```
int main()
{
    int n, k;

    do
    {
            printf("请输入项数 n(1<=n<=9)：");
            scanf("%d", &n);
    } while (n < 1 || n > 9);

    do
    {
            printf("请输入幂次 k(0<=k<=6)：");
            scanf("%d", &k);
    } while (k < 0 || k > 6);

    printf("1 至%d 的%d 次方之和=%d\n", n, k, sigma(n,k));
    return 0;
}
```

当用户输入不在范围之内时，do 循环确保用户再次输入，直到正确为止。

6.5 存储分类

在程序中，变量一经定义，系统就会在内存中为其分配一定大小的空间，并在其中保存其值。同时为了提高内存的使用效率，系统还必须及时回收那些不再使用的数据所占的内存单元。这涉及变量的存储分类（*storage-class*）问题。

C 语言的存储分类修饰符有 4 种。

- auto
- register
- static
- extern

用上述修饰符修饰的变量具有不同的特性。为了了解变量的特性，这里先引入几个与变量相关的概念。

- 生命期（*lifetime*）：指变量在程序运行时其存储能够保留的一段时间。这个定义的言外之意就是，有些变量在生命期结束后，其存储不再被保留，而是被系统自动回收了。
- 作用域（*scope*）：指变量在其生命期中所能覆盖的程序文本范围（即该变量可以使用的范围）。有 4 种作用域，分别是函数（*function*）、函数原型（*function prototype*）、语句块（*block*）和文件（*file*）。
- 存储持续（*storage duration*）：变量的存储持续决定了它的生命期。有 3 种存储持续，分别是静态的（*static*）、自动的（*automatic*）和分配的（*allocated*）。
- 可见性（*visibility*）：指变量在其所属作用域中能被访问的能力。如果变量能被其他语法单位访问，那么称其为对该语法单位是可见的（*visible*）。

一般地，变量根据其所处的作用域可分为局部变量和全局变量；根据存储持续可分为自动变量和静态变量。具有分配持续属性的变量将在 8.7 节中讨论。

6.5.1 局部变量和全局变量

变量在程序中声明的位置在很大程度上决定了它们对其他语法单位的可见性。一般地，变量可以声明在所有函数之外、某函数之内、语句块之内或函数的参数列表中。据此，变量可以分为局部变量和全局变量两类。

1. 局部变量

被声明在如下作用域中的变量都是局部（*Local*）的：

- 在函数体中声明的；
- 在函数原型中参数列表中出现的；
- 在语句块中声明的。

局部变量只在其所属作用域内起作用。具体一点，就是：

- 在函数原型中，参数变量只在括起它们的一对()中起作用；
- 在函数和块作用域中，变量只在直接包含它们的一对{}中起作用。

例如：

```
double a(double x); //x 只在这个函数声明中起作用
void f()
{
     int i = 0; //函数作用域中的变量

     if (i == 0) //i 在 if 语句中是可见的
     {
          int k; //块作用域中的变量，超出该范围 k 就是不可见的
          ++i;   //ok
          …
     }//k 的生命期结束
     --k; //error
}    //i 的生命期结束。
```

局部变量有如下的特性。

（1）在函数定义中，形参是典型的局部变量，它们的生命期和作用域都被局限在包含它们的函数内。调用一结束，形参变量就不能在别处使用。

（2）一旦变量的生命期结束后，就可以在别处重新声明这个变量。例如：

```
if (...)
{
     int i; //块中的变量
     ...
} //i 失效

while (...)
{
     int i; //仍是块中的变量，但与 if 块中的 i 无关
     ...
} //i 失效
```

（3）在嵌套的语句块中，内部块中的变量可以与外部块中的变量重名，但内部变量的作用域将屏蔽外部同名变量的作用域。例如：

```
{
     char c = 'A'; //outer c

     {
          char c = 'B'; //inner c，外层的 c 被内层声明的 c 屏蔽
          putchar(c); //输出‘B’
```

```
    } //end of inner c

    putchar(c); //输出'A'
} //end of outer c
```

而在同一个作用域中，变量是不能重名的。

2．全局变量

定义在文件作用域中的变量属于全局（*global*）变量，其特点是定义在所有函数之外。全局变量的生命周期从程序开始直到程序结束；它对自声明点起以后的所有语法单位都是可见的。

全局变量主要用于多个函数需要共享数据的场合。例如：

```
int shared = 0; //全局变量

void f()
{
    ++shared;
}

void g()
{
    ++shared;
}
```

如果有调用序列：

```
f();
g();
```

那么 shared 的值等于 2。

过多地使用全局变量会降低函数间的模块化程度。因此，程序员应该非常谨慎地使用。一个好的建议是：尽量避免使用全局变量，函数间只通过参数传递数据。

6.5.2 自动变量和静态变量

根据变量在程序中的作用，可以使用不同的存储分类修饰符去修饰它们。这决定了变量的生命期长短。据此，变量可以分为自动变量和静态变量两类。

1．自动变量

绝大多数的局部变量都属于自动变量，其特点是在进入作用域时其占据的内存被自动分配，在作用域结束时其内存被自动回收。

说明一个自动变量的语法为：

```
[auto] 类型名  变量名;
```

自动变量使用频率非常高，几乎无处不在，于是 C 语言很贴心地将 auto 关键字设计成可省略，使得它成为使用最少的关键字。

另一个用于修饰自动变量的关键字是 register。例如：

```
register int i;
```

register 关键字暗示变量 i 会被非常频繁地使用（例如作为循环控制变量），因此建议编译器对它的使用进行优化以尽可能快地访问它。现代的编译器在优化方面做得很好，所以 register 修饰基本上被忽略了。

2．静态变量

自动变量在其作用域结束后会被自动释放，因此不能再次使用。可实际情况是有时希望再进入作用域时，其值不会消失，继续保持上一次的值。C 语言提供的静态类型可以解决这个问题。

静态变量的定义格式如下：

```
static 类型名  变量名;
```

被 static 关键字约束的局部变量称为**静态变量**。与自动变量不同，静态变量的内存一旦分

配后，就不再释放，直到整个程序结束，所以其值能够始终保存。

静态变量在定义时一般要被初始化。这个初始化工作编译器只做一次，并且是在程序启动之前。

【例 6-4】静态变量的用法。

```
//6-4.c
#include <stdio.h>
int fun(int a)
{
    auto int b=0;
    static int c=3;
    ++b;
    ++c;
    return a + b + c;
}

int main()
{
    int i;
    for (i=0; i < 3; ++i)
            printf("第%d 次调用，结果是%d\n", i+1, fun(2));
    return 0;
}
```

运行结果如下：

```
第 1 次调用，结果是 7
第 2 次调用，结果是 8
第 3 次调用，结果是 9
```

分析其运行过程，b 和 c 的值见表 6-1。b 是自动变量，进入函数时它才被分配内存空间，调用一结束就自动释放，其值当然也随之消失，所以每次进入函数时的初值都始终为 0。而 c 是静态变量，其值一直保存，故每次执行++c 都是在前次操作的基础上自增。

表 6-1　　运行时 b、c 的值

第 i 次调用 fun	调用时初值		调用结束时的值		
	b	c	b	c	a+b+c
第 1 次	0	3	1	4	7
第 2 次	0	4	1	5	8
第 3 次	0	5	1	6	9

静态变量有如下特点。

（1）静态变量的初始化非常特殊。虽然看似每次调用时都对变量 c 进行了初始化操作，按理说其值都应该恢复为 3，但结果却不是。因为静态变量是在编译时初始化的，即只被初始化一次，程序运行时它已有初值，以后每次调用时都不再重新初始化而只是保留上次调用结束时的值。

（2）静态变量如果没有显式初始化，则编译时自动将其初始化为 0。而对于自动变量，如果没有初始化，那么编译器不会去自动初始化它，因此其值是未知的。

（3）虽然静态变量的生命期和全局变量的类似，其值在函数调用结束后仍然保存，但不同的是，静态变量如果在函数或块中定义，那么它就是一个局部变量，因此在其他函数里是不能引用的，而全局变量则可在程序中的任何函数中访问。

6.6 外部声明

到目前为止，书中所有例子程序都存放在一个 C 源文件中，因为这些程序都比较简单。而在实际工作中，一个应用往往比较复杂，它的开发由多位程序员完成，而每位程序员都会将自己编写的代码存放在一个源文件中。这样一个应用就包含了多个源文件。这样的格局会带来这样的问题：程序员 A 的源代码存放在编译单元（*translation unit*）f1.c 中；程序员 B 的源代码存放在编译单元 f2.c 中，其中声明了全局变量 i 和函数 f()；而程序员 A 要在自己编写的 f1.c 中共享 f2.c 中的 i 和 f()。那么这个问题该如何处理呢？

> 一般情况下，一个 C 源程序就是一个编译单元；而头文件.h 不是，因为它们根本不会被单独编译。

解决方案就是在 f1.c 中将 i 和 f()声明为外部的（*external*），其语法如下：

```
extern int i;
extern int f();
```

关键字 extern 告诉编译器：变量 i 和函数 f()具有外部链接（*external linkage*）属性，是在另一个编译单元中定义的，在编译的时候不用去考虑它们的存储或者实现，只要按规定使用它就好了。

具有外部属性的变量或函数虽然会在不同的编译单元中被多次声明或定义，但它们引用的是相同的变量或函数。因此，当 f1.c 和 f2.c 被分别编译并正确生成目标代码后，链接器在链接这些目标代码时，会将有外部链接属性的变量或函数合并，只生成一份实例。这样就确保了使用外部变量和函数的正确性。

【例 6-5】外部声明的应用。以下是两个源文件的代码。

```
//6-5-part1.c
#include <stdio.h>

extern int i;
extern int f();

int main()
{
    printf("i=%d,f()=%d", i, f());

    return 0;
}

//6-5-part2.c
int i = 99;

int f()
{
    return 999;
}
```

- 如果读者要验证这个例子，那么请将代码分别存放在两个不同的源文件中。
- 如果读者使用 VS 2010，那么请将以上两个源文件添加到一个解决方案中，然后启动“生成解决方案”，两个源文件会被分别编译并将目标代码链接在一起生成可执行代码。
- 如果使用 gcc，那么请使用如下命令：

```
gcc 6-5-part1.c 6-5-part2.c
```

gcc 会分别编译两个源文件并将目标代码链接在一起生成 a.exe 或者 a.out。

程序的运行结果是：

```
i=99,f()=999
```

所有的全局变量和函数都可能成为外部的，而局部变量因只具有内部链接属性（*internal linkage*）而不能被声明为外部的。有的时候为了阻止全局变量或函数外部化，可以使用 static 关键字将它们的链接属性说明成内部的。例如在 f2.c 中插入全局变量声明

```
static int j = 0;
```

那么变量 j 只在包含它的编译单元（即 f2.c）中起作用。在 f1.c 中使用外部声明

```
extern int j;
```

并在代码中使用变量 j 将会引起一个链接错误。

6.7 函数递归

函数嵌套调用中有一种特殊的方式，即一个函数在它的函数体内直接或间接地调用它本身。这种嵌套称为**函数递归**（*function recursion*），这类函数叫**递归函数**（*recursive functions*），例如：

```
void f()
{   …
    f(); //直接递归
    …
}
```

或者：

```
void g() //g()和 h()形成间接递归
{   …
    h();
    …
}
void h()
{   …
    g();
    …
}
```

可以看出，递归有点类似于之前讲的循环结构，都是在不停地反复执行同一段代码。

实际上，在日常生活中经常遇到递归问题。例如小时候听到的儿歌：“从前有座山，山上有座庙，庙里有个和尚在讲故事。什么故事呢？从前有座山，山上有座庙……”就是一个典型的递归问题。在数学领域，递归问题更是比比皆是。那么，C 语言又是怎样来处理递归问题的呢？先从一个实例说起吧。

【例 6-6】求 n!

【解题思路】这个题目可以用循环来完成。不过，这里采用另外一种方法。

读者都知道，阶乘的定义为：

```
n! = 1×2×3×…×n    (0! = 1)
```

那么，根据上面的公式，可以其将定义改写为：

```
n! = (1×2×3×…×(n-1)) ×n = (n-1)! ×n
```

显然，这是一种用阶乘自身来定义自身的方式，这就是典型的递归思想。现在来将递归问题模型化。

假设函数 *f*(*n*)用于求 *n*!，那么有

f(*n*) = *f*(*n*-1) * *n*

如果函数 f 被编码，那么计算 $f(n)$和计算 $f(n\text{-}1)$的代码肯定是一样的，不需要重新编码。这正好符合函数递归的要求。

然而一个问题是，如果直接使用递归公式，那么这个递归将没有终点，从而形成一种死循环。这显然不是想要的结果。不过，细心的读者可能已经发现，递归是一种倒推模式，将 n 阶的问题降到 n-1 阶，再如法炮制，直至降为 1 阶或 0 阶为止。而 0/1 阶问题的解都会直接给定，这些给定的值理所当然地应该成为递归的终点。

综合上面的分析，可以写出如下代码。

```
//6-6.c
#include <stdio.h>

int fact(int n)
{
    if (n == 1) return 1;

    return fact(n-1) * n;
}

int main()
{
    int n = 4, f;

    f = fact(n);
    printf("%d!=%d", n, f);

    return 0;
}
```

程序的运行结果为：

```
4!=24
```

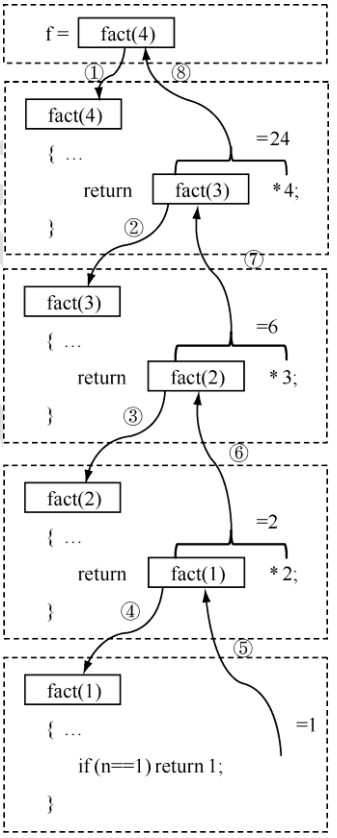

图 6-4　函数的递归过程

图 6-4 是函数调用 fact(4)递归过程的展开图。从图中可以清楚地看到，在没有到达递归终点之前，由于 return 语句中含有函数调用，因此它并不立即返回，而是一再进入函数（读者可以看到递归进入函数时参数的变化），直到到达终点后才一层一层地返回。

由以上程序可以总结出解决 n 阶递归问题的三步骤。假设解决 n 阶问题的函数是 $f(n)$，那么：

（1）确定递归结束条件。一般说来，是 n=1 附近的给定值 v。

（2）找出递归公式，并根据公式确定 $f(n)$和 $f(n\text{-}1)$的关系。

（3）按以下模式编写递归函数。

```
f(参数 n)
{
  if (n == 递归终点)
    return v;

  return 含有 f(n-1)的表达式;
}
```

下面再用一个经典的问题来进一步说明函数递归。

【例 6-7】Hanoi 塔问题。据说在某地的一个寺庙中，有一个称为 Hanoi 塔的装置，该装置由一个平板上的 3 根宝石柱构成，其中一根柱从上到下叠放了由小到大的 64 片金片。一位僧侣借助于第 3 根柱子的辅助，将金片从一根柱子上移到另一根柱子上。移动的规则是：

- 一次只能移动一片；

- 大片不能盖在小片上面。

传说如果有一天所有 64 片金片都成功转移，那么世界将到尽头。

这真是耸人听闻。如果用较科学的方式来计算一下，那么这位僧侣到底要花多长的时间才能成功呢？

首先将问题做一个模型化处理。图 6-5 示意了 Hanoi 塔的结构。

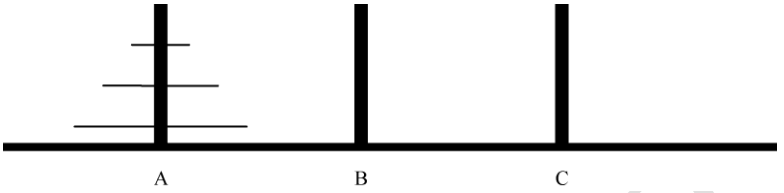

图 6-5　Hanoi 塔

这里假设金片数为 n，要从 A 移到 B，以 C 作为辅助。

如果读者从移动单片的角度出发，那么在片数很少的情况下，可以考虑得比较周全。但一旦片数较多，就会发现无从下手。现在不妨换个角度思考，从整体的角度出发来发现问题的解决方案。仔细分析移动规则，可以发现，如果要成功移动所有金片，那么就必须将除了最大片以外的所有的金片全部转移到 C 上，然后才能直接将这张最大片一次移动到 B 上，否则一定会违背规则。据此，可以设计出如下算法。

假设移动金片的算法名为 Hanoi，那么它将有如下格式：

```
Hanoi(片数，源，目的，辅助);
```

那么，将 n 片金片从 A 移动到 B，以 C 作为辅助的实现就是：

```
Hanoi(n, A, B, C)
```

具体的移动步骤是：

（1）将 A 上除了最大片外的 n-1 片以 B 作为辅助移动到 C。可以看到，移动 n-1 片和移动 n 片的规则和过程是一模一样的，只不过作为源、目的和辅助的柱子有所不同，所以其过程应该就是：

Hanoi(n-1, A, C, B)。这是一次明显的递归。

（2）将剩下的最大片一次从 A 移动到 B，这里写成 A→B；

（3）再将 C 上的 n-1 片以 A 作为辅助移动到 B，也就是

Hanoi(n-1, C, B, A)。这也是一次递归。移动完成。

根据上述算法，设 a_n 为移动 n 片的次数，那么有

$a_n = a_{n-1} + 1 + a_{n-1} = 2a_{n-1} + 1$

成立。这是个典型的两项递推关系，其通解为 $a_n=2^n-1$。如果僧侣一秒钟移动一片（这已经很快了），并且用最佳移动方案，那么他需要花 $2^{64}-1$ 秒，也就是近似于 1.6×10^{19} 秒，或者说 5×10^{11} 年。大家知晓答案了，这个数字确实惊人。

现在深入一点，来求解如何移动那些金片。对于 3 片的情况，给金片从小到大编号为 1、2、3，那么最少需要 7 步完成移动。

（1）1 from A to B
（2）2 from A to C
（3）1 from B to C
（4）3 from A to B
（5）1 from C to A
（6）2 from C to B
（7）1 from A to B

多于 3 片的情况就复杂得多了。不过，可以用递归函数来解决问题。

```
//6-7.c
#include <stdio.h>
```

```
void Hanoi(int num, char src, char dest, char aux)
{
    if (num == 1)
    {
        printf("%c--->%c\n", src, dest);
        return;
    }

    Hanoi(num - 1, src, aux, dest);
    printf("%c--->%c\n", src, dest);
    Hanoi(num - 1, aux, dest, src);
}

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

    return 0;
}
```

程序的运行结果是：

```
A--->B
A--->C
B--->C
A--->B
C--->A
C--->B
A--->B
```

至此，读者已经知道，递归从根本上来讲是一种循环结构，但它与循环又有很大的区别。递归是一种函数调用，需要系统资源的支持。这里用到的系统资源主要是堆栈，用于保存每一次递归调用时的现场，以便递归返回后能够正确地回到调用前的环境。而循环没有用到系统资源，它只是一种简单的重复。

需要指出的是，递归非常消耗堆栈资源，特别是在递归深度较大的时候。

总的来说，递归是一项很有用的编程技术，它在处理有递推关系、回溯等问题方面有得天独厚的优势。不过，正如指出的那样，递归对于初学者来说是一个难点，需要花较多的时间去理解和应用。