8 函 数

1.1 概述

在前面已经介绍过，C源程序是由函数组成的。虽然在前面各章的程序中大都只有一个主函数 main()，但实用程序往往由多个函数组成。函数是C源程序的基本模块，通过对函数模块的调用实现特定的功能。C语言中的函数相当于其它高级语言的子程序。C语言不仅提供了极为丰富的库函数(如 Turbo C，MS C 都提供了三百多个库函数)，还允许用户建立自己定义的函数。用户可把自己的算法编成一个个相对独立的函数模块，然后用调用的方法来使用函数。可以说C程序的全部工作都是由各式各样的函数完成的，所以也把C语言称为函数式语言。

由于采用了函数模块式的结构，C语言易于实现结构化程序设计。使程序的层次结构清晰，便于程序的编写、阅读、调试。

在C语言中可从不同的角度对函数分类。

1. 从函数定义的角度看，函数可分为库函数和用户定义函数两种。
   1) 库函数：由C系统提供，用户无须定义，也不必在程序中作类型说明，只需在程序前包含有该函数原型的头文件即可在程序中直接调用。在前面各章的例题中反复用到 printf、scanf、getchar、putchar、gets、puts、strcat 等函数均属此类。

2) 用户定义函数：由用户按需要写的函数。对于用户自定义函数，不仅要在程序中定义函数本身，而且在主调函数模块中还必须对该被调函数进行类型说明，然后才能使用。

2. C语言的函数兼有其它语言中的函数和过程两种功能，从这个角度看，又可把函数分为有返回值函数和无返回值函数两种。
   1) 有返回值函数：此类函数被调用执行完后将向调用者返回一个执行结果，称为函数返回值。如数学函数即属于此类函数。由用户定义的这种要返回函数值的函数，必须在函数定义和函数说明中明确返回值的类型。
   2) 无返回值函数：此类函数用于完成某项特定的处理任务，执行完成后不向调用者返回函数值。这类函数类似于其它语言的过程。由于函数无须返回值，用户在定义此类函数时可指定它的返回为“空类型”， 空类型的说明符为“void”。
3. 从主调函数和被调函数之间数据传送的角度看又可分为无参函数和有参函数两种。
   1) 无参函数：函数定义、函数说明及函数调用中均不带参数。主调函数和被调函数之间不进行参数传送。此类函数通常用来完成一组指定的功能，可以返回或不返回函数值。
   2) 有参函数：也称为带参函数。在函数定义及函数说明时都有参数，称为形式参数(简称为形参)。在函数调用时也必须给出参数，称为实际参数(简称为实参)。进行函数调用时，主调函数将把实参的值传送给形参，供被调函数使用。
4. C语言提供了极为丰富的库函数，这些库函数又可从功能角度作以下分类。
   1) 字符类型分类函数：用于对字符按ASCII码分类：字母，数字，控制字符，分隔符，大小写字母等。
   2) 转换函数：用于字符或字符串的转换；在字符量和各类数字量(整型，实型等)之间进行转换；在大、小写之间进行转换。
   3) 目录路径函数：用于文件目录和路径操作。
   4) 诊断函数：用于内部错误检测。
   5) 图形函数：用于屏幕管理和各种图形功能。
   6) 输入输出函数：用于完成输入输出功能。
   7) 接口函数：用于与DOS，BIOS和硬件的接口。
   8) 字符串函数：用于字符串操作和处理。
   9) 内存管理函数：用于内存管理。
   10) 数学函数：用于数学函数计算。
   11) 日期和时间函数：用于日期，时间转换操作。
   12) 进程控制函数：用于进程管理和控制。
   13) 其它函数：用于其它各种功能。

以上各类函数不仅数量多，而且有的还需要硬件知识才会使用，因此要想全部掌握则需要一个较长的学习过程。应首先掌握一些最基本、最常用的函数，再逐步深入。由于课时关系，我们只介绍了很少一部分库函数，其余部分读者可根据需要查阅有关手册。

还应该指出的是，在C语言中，所有的函数定义，包括主函数main在内，都是平行的。也就是说，在一个函数的函数体内，不能再定义另一个函数，即不能嵌套定义。但是函数之间允许相互调用，也允许嵌套调用。习惯上把调用者称为主调函数。函数还可以自己调用自己，称为递归调用。

main 函数是主函数，它可以调用其它函数，而不允许被其它函数调用。因此， C程序的执行总是从main函数开始，完成对其它函数的调用后再返回到main函数，最后由main函数结束整个程序。一个C源程序必须有，也只能有一个主函数main。

1.1 函数定义的一般形式

1. 无参函数的定义形式

```
类型标识符 函数名()
{声明部分
 语句
}
```

其中类型标识符和函数名称为函数头。类型标识符指明了本函数的类型，函数的类型实际上是函数返回值的类型。 该类型标识符与前面介绍的各种说明符相同。函数名是由用户定义的标识符，函数名后有一个空括号，其中无参数，但括号不可少。

{}中的内容称为函数体。在函数体中声明部分，是对函数体内部所用到的变量的类型说明。

在很多情况下都不要求无参函数有返回值，此时函数类型符可以写为 void。

我们可以改写一个函数定义：

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

这里，只把 main 改为 Hello 作为函数名，其余不变。Hello 函数是一个无参函数，当被其它函数调用时，输出 Hello world 字符串。

2. 有参函数定义的一般形式

```
类型标识符 函数名(形式参数表列)
{声明部分
 语句
}
```

有参函数比无参函数多了一个内容，即形式参数表列。在形参表中给出的参数称为形式参数，它们可以是各种类型的变量，各参数之间用逗号间隔。在进行函数调用时，主调函数将赋予这些形式参数实际的值。形参既然是变量，必须在形参表中给出形参的类型说明。

例如，定义一个函数，用于求两个数中的大数，可写为：

```
int max(int a, int b)
{
  if (a>b) return a;
  else return b;
}
```

第一行说明 max 函数是一个整型函数，其返回的函数值是一个整数。形参为 a,b,均为整型量。a,b 的具体值是由主调函数在调用时传送过来的。在{}中的函数体内，除形参外没有使用其它变量，因此只有语句而没有声明部分。在 max 函数体中的 return 语句是把 a(或 b)的值作为函数的值返回给主调函数。有返回值函数中至少应有一个 return 语句。

在 C 程序中，一个函数的定义可以放在任意位置，既可放在主函数 main 之前，也可放在 main 之后。

例如：

可把 max 函数置在 main 之后，也可以把它放在 main 之前。修改后的程序如下所示。

【例 8.1】

```
int max(int a,int b)
{
    if(a>b)return a;
    else return b;
}
main()
{
    int max(int a,int b);
    int x,y,z;
    printf("input two numbers:\n");
    scanf("%d%d",&x,&y);
    z=max(x,y);
    printf("maxmum=%d",z);
}
```

现在我们可以从函数定义、函数说明及函数调用的角度来分析整个程序，从中进一步了解函数的各种特点。

程序的第 1 行至第 5 行为 max 函数定义。进入主函数后，因为准备调用 max 函数，故先对 max 函数进行说明(程序第 8 行)。函数定义和函数说明并不是一回事，在后面还要专门讨论。 可以看出函数说明与函数定义中的函数头部分相同，但是末尾要加分号。程序第 12 行为调用 max 函数，并把 x, y 中的值传送给 max 的形参 a, b。max 函数执行的结果(a 或 b)将返回给变量 z。最后由主函数输出 z 的值。

1.1 函数的参数和函数的值

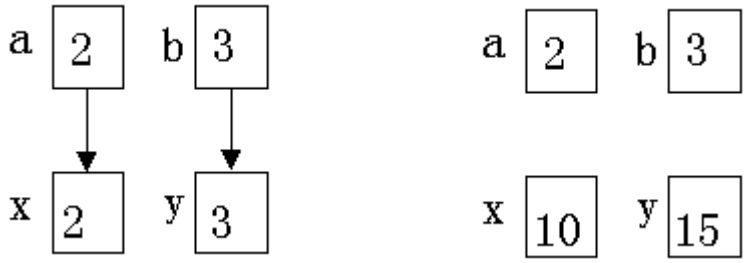
1.1.1 形式参数和实际参数

前面已经介绍过，函数的参数分为形参和实参两种。在本小节中，进一步介绍形参、实参的特点和两者的关系。形参出现在函数定义中，在整个函数体内都可以使用，离开该函数则不能使用。实参出现在主调函数中，进入被调函数后，实参变量也不能使用。形参和实参的功能是作数据传送。发生函数调用时，主调函数把实参的值传送给被调函数的形参从而实现主调函数向被调函数的数据传送。

函数的形参和实参具有以下特点：

1. 形参变量只有在被调用时才分配内存单元,在调用结束时,即刻释放所分配的内存单元。因此，形参只有在函数内部有效。函数调用结束返回主调函数后则不能再使用该形参变量。
2. 实参可以是常量、变量、表达式、函数等，无论实参是何种类型的量，在进行函数调用时，它们都必须具有确定的值，以便把这些值传送给形参。因此应预先用赋值，输入等办法使实参获得确定值。
3. 实参和形参在数量上，类型上，顺序上应严格一致，否则会发生类型不匹配”的错误。
4. 函数调用中发生的数据传送是单向的。即只能把实参的值传送给形参，而不能把形参的值反向地传送给实参。 因此在函数调用过程中，形参的值发生改变，而实参中的值不

会变化。



【例 8.2】可以说明这个问题。

```
main()
{
    int n;
    printf("input number\n");
    scanf("%d",&n);
    s(n);
    printf("n=%d\n",n);
}
int s(int n)
{
    int i;
    for(i=n-1;i>=1;i--)
      n=n+i;
    printf("n=%d\n",n);
}
```

本程序中定义了一个函数 s，该函数的功能是求 $\sum n_i$ 的值。在主函数中输入 n 值，并作为实参，在调用时传送给 s 函数的形参量 n(注意，本例的形参变量和实参变量的标识符都为 n，但这是两个不同的量，各自的作用域不同)。在主函数中用 printf 语句输出一次 n 值，这个 n 值是实参 n 的值。在函数 s 中也用 printf 语句输出了一次 n 值，这个 n 值是形参最后取得的 n 值 0。从运行情况看，输入 n 值为 100。即实参 n 的值为 100。把此值传给函数 s 时，形参 n 的初值也为 100，在执行函数过程中，形参 n 的值变为 5050。返回主函数之后，输出实参 n 的值仍为 100。可见实参的值不随形参的变化而变化。

1.1.1 函数的返回值

函数的值是指函数被调用之后，执行函数体中的程序段所取得的并返回给主调函数的值。如调用正弦函数取得正弦值，调用例 8.1 的 max 函数取得的最大数等。对函数的值(或称函数返回值)有以下一些说明：

1)　函数的值只能通过 return 语句返回主调函数。

return　语句的一般形式为：

```
return 表达式;
```

或者为：

return (表达式);

该语句的功能是计算表达式的值，并返回给主调函数。在函数中允许有多个 return 语句，但每次调用只能有一个 return 语句被执行，因此只能返回一个函数值。

2) 函数值的类型和函数定义中函数的类型应保持一致。如果两者不一致，则以函数类型为准，自动进行类型转换。

3) 如函数值为整型，在函数定义时可以省去类型说明。

4) 不返回函数值的函数，可以明确定义为“空类型”，类型说明符为“void”。如例 8.2 中函数 s 并不向主函数返函数值，因此可定义为：

```
void s(int n)
{ ……
 }
```

一旦函数被定义为空类型后，就不能在主调函数中使用被调函数的函数值了。例如，在定义 s 为空类型后，在主函数中写下述语句

```
sum=s(n);
```

就是错误的。

为了使程序有良好的可读性并减少出错， 凡不要求返回值的函数都应定义为空类型。

1.1 函数的调用

1.1.1 函数调用的一般形式

前面已经说过，在程序中是通过对函数的调用来执行函数体的，其过程与其它语言的子程序调用相似。

C 语言中，函数调用的一般形式为：

函数名(实际参数表)

对无参函数调用时则无实际参数表。实际参数表中的参数可以是常数，变量或其它构造类型数据及表达式。各实参之间用逗号分隔。

1.1.1 函数调用的方式

在 C 语言中，可以用以下几种方式调用函数：

1. 函数表达式：函数作为表达式中的一项出现在表达式中，以函数返回值参与表达式的运算。这种方式要求函数是有返回值的。例如：z=max(x,y)是一个赋值表达式，把 max 的返回值赋予变量 z。
2. 函数语句：函数调用的一般形式加上分号即构成函数语句。例如： printf("%d",a);scanf ("%d",&b);都是以函数语句的方式调用函数。
3. 函数实参：函数作为另一个函数调用的实际参数出现。这种情况是把该函数的返回值作为实参进行传送，因此要求该函数必须是有返回值的。例如：printf("%d",max(x,y)); 即是把 max 调用的返回值又作为 printf 函数的实参来使用的。在函数调用中还应该注意的一个问题是求值顺序的问题。所谓求值顺序是指对实

参表中各量是自左至右使用呢，还是自右至左使用。对此，各系统的规定不一定相同。

介绍 printf 函数时已提到过，这里从函数调用的角度再强调一下。

【例 8.3】

```
main()
{
    int i=8;
    printf("%d\n%d\n%d\n%d\n",++i,--i,i++,i--);
}
```

如按照从右至左的顺序求值。运行结果应为：

8
7
7
8

如对 printf 语句中的++i，--i，i++，i--从左至右求值，结果应为：

9
8
8
9

应特别注意的是，无论是从左至右求值， 还是自右至左求值，其输出顺序都是不变的，即输出顺序总是和实参表中实参的顺序相同。由于 Turbo C 现定是自右至左求值，所以结果为 8，7，7，8。上述问题如还不理解，上机一试就明白了。

1.1.1 被调用函数的声明和函数原型

在主调函数中调用某函数之前应对该被调函数进行说明（声明），这与使用变量之前要先进行变量说明是一样的。在主调函数中对被调函数作说明的目的是使编译系统知道被调函数返回值的类型，以便在主调函数中按此种类型对返回值作相应的处理。

其一般形式为：

```
类型说明符 被调函数名(类型 形参，类型 形参…)；
```

或为：

```
类型说明符 被调函数名(类型，类型…)；
```

括号内给出了形参的类型和形参名，或只给出形参类型。这便于编译系统进行检错，以防止可能出现的错误。

例 8.1 main 函数中对 max 函数的说明为：

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

或写为:

```
int max(int,int);
```

C 语言中又规定在以下几种情况时可以省去主调函数中对被调函数的函数说明。

1) 如果被调函数的返回值是整型或字符型时，可以不对被调函数作说明，而直接调用。这时系统将自动对被调函数返回值按整型处理。例 8.2 的主函数中未对函数 s 作说明而直接调用即属此种情形。

2) 当被调函数的函数定义出现在主调函数之前时，在主调函数中也可以不对被调函数

再作说明而直接调用。例如例 8.1 中，函数 max 的定义放在 main 函数之前，因此可在 main 函数中省去对 max 函数的函数说明 int max(int a,int b)。

3) 如在所有函数定义之前，在函数外预先说明了各个函数的类型，则在以后的各主调函数中，可不再对被调函数作说明。例如：

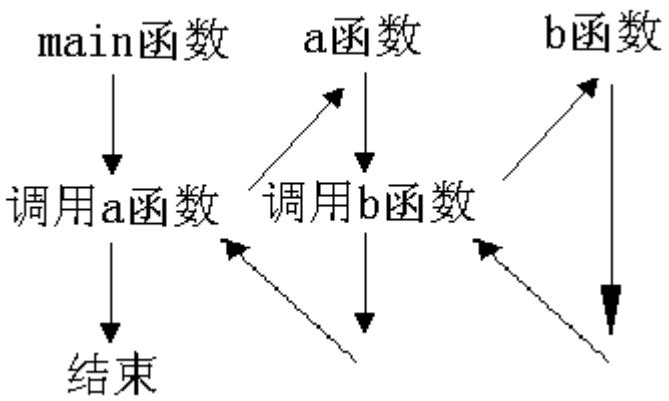
```
char str(int a);
float f(float b);
main()
{
 ……
}
char str(int a)
{
 ……
}
float f(float b)
{
 ……
}
```

其中第一，二行对 str 函数和 f 函数预先作了说明。因此在以后各函数中无须对 str 和 f 函数再作说明就可直接调用。

4) 对库函数的调用不需要再作说明，但必须把该函数的头文件用 include 命令包含在源文件前部。

1.1 函数的嵌套调用

C 语言中不允许作嵌套的函数定义。因此各函数之间是平行的，不存在上一级函数和下一级函数的问题。但是 C 语言允许在一个函数的定义中出现对另一个函数的调用。这样就出现了函数的嵌套调用。即在被调函数中又调用其它函数。这与其它语言的子程序嵌套的情形是类似的。其关系可表示如图。



图表示了两层嵌套的情形。其执行过程是：执行 main 函数中调用 a 函数的语句时，即转去执行 a 函数，在 a 函数中调用 b 函数时，又转去执行 b 函数，b 函数执行完毕返回 a 函数的断点继续执行，a 函数执行完毕返回 main 函数的断点继续执行。

【例 8.4】计算 s=$2^2!+3^2!$

本题可编写两个函数，一个是用来计算平方值的函数 f1，另一个是用来计算阶乘值的函数 f2。主函数先调 f1 计算出平方值，再在 f1 中以平方值为实参，调用 f2 计算其阶乘值，然后返回 f1，再返回主函数，在循环程序中计算累加和。

```
long f1(int p)
{
    int k;
    long r;
    long f2(int);
    k=p*p;
    r=f2(k);
    return r;
}
long f2(int q)
{
    long c=1;
    int i;
    for(i=1;i<=q;i++)
      c=c*i;
    return c;
}
main()
{
    int i;
    long s=0;
    for (i=2;i<=3;i++)
      s=s+f1(i);
    printf("\ns=%ld\n",s);
}
```

在程序中，函数 f1 和 f2 均为长整型，都在主函数之前定义，故不必再在主函数中对 f1 和 f2 加以说明。在主程序中，执行循环程序依次把 i 值作为实参调用函数 f1 求 i^2 值。在 f1 中又发生对函数 f2 的调用，这时是把 i^2 的值作为实参去调 f2，在 f2 中完成求 $i^2!$ 的计算。f2 执行完毕把 C 值(即 $i^2!$)返回给 f1，再由 f1 返回主函数实现累加。至此，由函数的嵌套调用实现了题目的要求。由于数值很大，所以函数和一些变量的类型都说明为长整型，否则会造成计算错误。

1.1 函数的递归调用

一个函数在它的函数体内调用它自身称为递归调用。这种函数称为递归函数。C 语言允许函数的递归调用。在递归调用中，主调函数又是被调函数。执行递归函数将反复调用其自身，每调用一次就进入新的一层。

例如有函数 f 如下：

```
int f(int x)
{
  int y;
  z=f(y);
  return z;
}
```

这个函数是一个递归函数。但是运行该函数将无休止地调用其自身，这当然是不正确的。为了防止递归调用无终止地进行，必须在函数内有终止递归调用的手段。常用的办法是加条件判断，满足某种条件后就不再作递归调用，然后逐层返回。下面举例说明递归调用的执行过程。

【例 8.5】用递归法计算 n!

用递归法计算 n!可用下述公式表示：

n!=1　　(n=0,1)

n×(n-1)!　　(n>1)

按公式可编程如下：

```
long ff(int n)
{
    long f;
    if(n<0) printf("n<0,input error");
    else if(n==0||n==1) f=1;
    else f=ff(n-1)*n;
    return(f);
}
main()
{
    int n;
    long y;
    printf("\ninput a inteager number:\n");
    scanf("%d",&n);
    y=ff(n);
    printf("%d!=%ld",n,y);
}
```

程序中给出的函数 ff 是一个递归函数。主函数调用 ff 后即进入函数 ff 执行，如果 n<0,n==0 或 n=1 时都将结束函数的执行，否则就递归调用 ff 函数自身。由于每次递归调用的实参为 n-1，即把 n-1 的值赋予形参 n，最后当 n-1 的值为 1 时再作递归调用，形参 n 的值

也为 1，将使递归终止。然后可逐层退回。

下面我们再举例说明该过程。设执行本程序时输入为 5，即求 5!。在主函数中的调用语句即为 y=ff(5),进入 ff 函数后，由于 n=5,不等于 0 或 1，故应执行 f=ff(n-1)*n,即 f=ff(5-1)*5。该语句对 ff 作递归调用即 ff(4)。

进行四次递归调用后，ff 函数形参取得的值变为 1，故不再继续递归调用而开始逐层返回主调函数。ff(1)的函数返回值为 1，ff(2)的返回值为 1*2=2，ff(3)的返回值为 2*3=6，ff(4)的返回值为 6*4=24，最后返回值 ff(5)为 24*5=120。

例 8.5 也可以不用递归的方法来完成。如可以用递推法，即从 1 开始乘以 2，再乘以 3…直到 n。递推法比递归法更容易理解和实现。但是有些问题则只能用递归算法才能实现。典型的问题是 Hanoi 塔问题。

【例 8.6】Hanoi 塔问题

一块板上有三根针，A，B，C。A 针上套有 64 个大小不等的圆盘，大的在下，小的在上。如图 5.4 所示。要把这 64 个圆盘从 A 针移动 C 针上，每次只能移动一个圆盘，移动可以借助 B 针进行。但在任何时候，任何针上的圆盘都必须保持大盘在下，小盘在上。求移动的步骤。

本题算法分析如下，设 A 上有 n 个盘子。

如果 n=1，则将圆盘从 A 直接移动到 C。

如果 n=2，则：

1.将 A 上的 n-1(等于 1)个圆盘移到 B 上；

2.再将 A 上的一个圆盘移到 C 上；

3.最后将 B 上的 n-1(等于 1)个圆盘移到 C 上。

如果 n=3，则：

A. 将 A 上的 n-1(等于 2，令其为 n`)个圆盘移到 B(借助于 C)，步骤如下：

(1)将 A 上的 n`-1(等于 1)个圆盘移到 C 上。

(2)将 A 上的一个圆盘移到 B。

(3)将 C 上的 n`-1(等于 1)个圆盘移到 B。

B. 将 A 上的一个圆盘移到 C。

C. 将 B 上的 n-1(等于 2，令其为 n`)个圆盘移到 C(借助 A)，步骤如下：

(1)将 B 上的 n`-1(等于 1)个圆盘移到 A。

(2)将 B 上的一个盘子移到 C。

(3)将 A 上的 n`-1(等于 1)个圆盘移到 C。

到此，完成了三个圆盘的移动过程。

从上面分析可以看出，当 n 大于等于 2 时，移动的过程可分解为三个步骤：

第一步　把 A 上的 n-1 个圆盘移到 B 上；

第二步　把 A 上的一个圆盘移到 C 上；

第三步　把 B 上的 n-1 个圆盘移到 C 上；其中第一步和第三步是类同的。

当 n=3 时，第一步和第三步又分解为类同的三步，即把 n`-1 个圆盘从一个针移到另一个针上，这里的 n`=n-1。 显然这是一个递归过程，据此算法可编程如下：

```
move(int n,int x,int y,int z)
{
    if(n==1)
      printf("%c-->%c\n",x,z);
    else
```

```
    {
      move(n-1,x,z,y);
      printf("%c-->%c\n",x,z);
      move(n-1,y,x,z);
    }
}
main()
{
    int h;
    printf("\ninput number:\n");
    scanf("%d",&h);
    printf("the step to moving %2d diskes:\n",h);
    move(h,'a','b','c');
}
```

从程序中可以看出,move 函数是一个递归函数，它有四个形参 n,x,y,z。n 表示圆盘数,x,y,z 分别表示三根针。move 函数的功能是把 x 上的 n 个圆盘移动到 z 上。当 n==1 时，直接把 x 上的圆盘移至 z 上，输出 x→z。如 n!=1 则分为三步：递归调用 move 函数，把 n-1 个圆盘从 x 移到 y；输出 x→z；递归调用 move 函数，把 n-1 个圆盘从 y 移到 z。在递归调用过程中 n=n-1，故 n 的值逐次递减，最后 n=1 时，终止递归，逐层返回。当 n=4 时程序运行的结果为：

```
input number:
4
the step to moving 4 diskes:
a→b
a→c
b→c
a→b
c→a
c→b
a→b
a→c
b→c
b→a
c→a
b→c
a→b
a→c
b→c
```

1.1 数组作为函数参数

数组可以作为函数的参数使用，进行数据传送。数组用作函数参数有两种形式，一种是

把数组元素(下标变量)作为实参使用；另一种是把数组名作为函数的形参和实参使用。

1. 数组元素作函数实参

数组元素就是下标变量，它与普通变量并无区别。 因此它作为函数实参使用与普通变量是完全相同的，在发生函数调用时，把作为实参的数组元素的值传送给形参，实现单向的值传送。例 5.4 说明了这种情况。

【例 8.7】判别一个整数数组中各元素的值，若大于 0 则输出该值，若小于等于 0 则输出 0 值。编程如下：

```
void nzp(int v)
{
    if(v>0)
      printf("%d ",v);
    else
      printf("%d ",0);
}
main()
{
    int a[5],i;
    printf("input 5 numbers\n");
    for(i=0;i<5;i++)
      {scanf("%d",&a[i]);
       nzp(a[i]);}
}
```

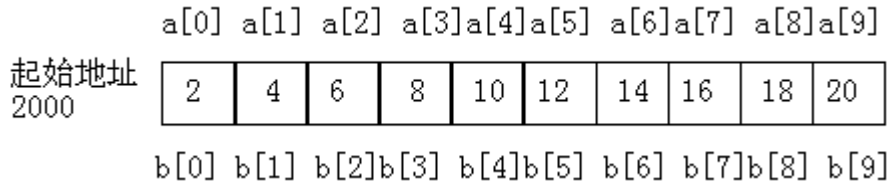
本程序中首先定义一个无返回值函数 nzp，并说明其形参 v 为整型变量。在函数体中根据 v 值输出相应的结果。在 main 函数中用一个 for 语句输入数组各元素，每输入一个就以该元素作实参调用一次 nzp 函数，即把 a[i]的值传送给形参 v，供 nzp 函数使用。

2. 数组名作为函数参数

用数组名作函数参数与用数组元素作实参有几点不同：

1) 用数组元素作实参时，只要数组类型和函数的形参变量的类型一致，那么作为下标变量的数组元素的类型也和函数形参变量的类型是一致的。因此，并不要求函数的形参也是下标变量。换句话说，对数组元素的处理是按普通变量对待的。用数组名作函数参数时，则要求形参和相对应的实参都必须是类型相同的数组，都必须有明确的数组说明。当形参和实参二者不一致时，即会发生错误。

2) 在普通变量或下标变量作函数参数时，形参变量和实参变量是由编译系统分配的两个不同的内存单元。在函数调用时发生的值传送是把实参变量的值赋予形参变量。在用数组名作函数参数时，不是进行值的传送，即不是把实参数组的每一个元素的值都赋予形参数组的各个元素。因为实际上形参数组并不存在，编译系统不为形参数组分配内存。那么，数据的传送是如何实现的呢?在我们曾介绍过，数组名就是数组的首地址。因此在数组名作函数参数时所进行的传送只是地址的传送，也就是说把实参数组的首地址赋予形参数组名。形参数组名取得该首地址之后，也就等于有了实在的数组。实际上是形参数组和实参数组为同一数组，共同拥有一段内存空间。



上图说明了这种情形。图中设 a 为实参数组，类型为整型。a 占有以 2000 为首地址的一块内存区。b 为形参数组名。当发生函数调用时，进行地址传送，把实参数组 a 的首地址传送给形参数组名 b，于是 b 也取得该地址 2000。于是 a，b 两数组共同占有以 2000 为首地址的一段连续内存单元。从图中还可以看出 a 和 b 下标相同的元素实际上也占相同的两个内存单元(整型数组每个元素占二字节)。例如 a[0]和 b[0]都占用 2000 和 2001 单元，当然 a[0]等于 b[0]。类推则有 a[i]等于 b[i]。

【例 8.8】数组 a 中存放了一个学生 5 门课程的成绩，求平均成绩。

```
float aver(float a[5])
{
    int i;
    float av,s=a[0];
    for(i=1;i<5;i++)
      s=s+a[i];
    av=s/5;
    return av;
}
void main()
{
    float sco[5],av;
    int i;
    printf("\ninput 5 scores:\n");
    for(i=0;i<5;i++)
      scanf("%f",&sco[i]);
    av=aver(sco);
    printf("average score is %5.2f",av);
}
```

本程序首先定义了一个实型函数 aver，有一个形参为实型数组 a，长度为 5。在函数 aver 中，把各元素值相加求出平均值，返回给主函数。主函数 main 中首先完成数组 sco 的输入，然后以 sco 作为实参调用 aver 函数，函数返回值送 av，最后输出 av 值。 从运行情况可以看出，程序实现了所要求的功能。

3) 前面已经讨论过，在变量作函数参数时，所进行的值传送是单向的。即只能从实参传向形参，不能从形参传回实参。形参的初值和实参相同，而形参的值发生改变后，实参并不变化，两者的终值是不同的。而当用数组名作函数参数时，情况则不同。由于实际上形参和实参为同一数组，因此当形参数组发生变化时，实参数组也随之变化。当然这种情况不能理解为发生了“双向”的值传递。但从实际情况来看，调

用函数之后实参数组的值将由于形参数组值的变化而变化。为了说明这种情况，把例 5.4 改为例 5.6 的形式。

【例 8.9】题目同 8.7 例。改用数组名作函数参数。

```
void nzp(int a[5])
{
    int i;
    printf("\nvalues of array a are:\n");
    for(i=0;i<5;i++)
    {
    if(a[i]<0) a[i]=0;
    printf("%d ",a[i]);
    }
}
main()
{
    int b[5],i;
    printf("\ninput 5 numbers:\n");
    for(i=0;i<5;i++)
      scanf("%d",&b[i]);
    printf("initial values of array b are:\n");
    for(i=0;i<5;i++)
      printf("%d ",b[i]);
    nzp(b);
    printf("\nlast values of array b are:\n");
    for(i=0;i<5;i++)
      printf("%d ",b[i]);
}
```

本程序中函数 nzp 的形参为整数组 a，长度为 5。主函数中实参数组 b 也为整型，长度也为 5。在主函数中首先输入数组 b 的值，然后输出数组 b 的初始值。然后以数组名 b 为实参调用 nzp 函数。在 nzp 中，按要求把负值单元清 0，并输出形参数组 a 的值。 返回主函数之后，再次输出数组 b 的值。从运行结果可以看出，数组 b 的初值和终值是不同的，数组 b 的终值和数组 a 是相同的。这说明实参形参为同一数组，它们的值同时得以改变。

用数组名作为函数参数时还应注意以下几点：

a. 形参数组和实参数组的类型必须一致，否则将引起错误。

b. 形参数组和实参数组的长度可以不相同，因为在调用时，只传送首地址而不检查形参数组的长度。当形参数组的长度与实参数组不一致时，虽不至于出现语法错误(编译能通过)，但程序执行结果将与实际不符，这是应予以注意的。

【例 8.10】如把例 8.9 修改如下：

```
void nzp(int a[8])
{
    int i;
    printf("\nvalues of array aare:\n");
```

```
    for(i=0;i<8;i++)
    {
      if(a[i]<0)a[i]=0;
      printf("%d ",a[i]);
    }
}
main()
{
    int b[5],i;
    printf("\ninput 5 numbers:\n");
    for(i=0;i<5;i++)
      scanf("%d",&b[i]);
    printf("initial values of array b are:\n");
    for(i=0;i<5;i++)
      printf("%d ",b[i]);
    nzp(b);
    printf("\nlast values of array b are:\n");
    for(i=0;i<5;i++)
      printf("%d ",b[i]);
}
```

本程序与例 8.9 程序比，nzp 函数的形参数组长度改为 8，函数体中，for 语句的循环条件也改为 i<8。因此，形参数组 a 和实参数组 b 的长度不一致。编译能够通过，但从结果看，数组 a 的元素 a[5]，a[6]，a[7]显然是无意义的。

c. 在函数形参表中，允许不给出形参数组的长度，或用一个变量来表示数组元素的个数。

例如，可以写为：

```
void nzp(int a[])
```

或写为

```
void nzp(int a[], int n)
```

其中形参数组 a 没有给出长度，而由 n 值动态地表示数组的长度。n 的值由主调函数的实参进行传送。

由此，例 8.10 又可改为例 8.11 的形式。

【例 8.11】

```
void nzp(int a[],int n)
{
    int i;
    printf("\nvalues of array a are:\n");
    for(i=0;i<n;i++)
      {
    if(a[i]<0) a[i]=0;
    printf("%d ",a[i]);
      }
```

```
}
main()
{
    int b[5],i;
    printf("\ninput 5 numbers:\n");
    for(i=0;i<5;i++)
      scanf("%d",&b[i]);
    printf("initial values of array b are:\n");
    for(i=0;i<5;i++)
      printf("%d ",b[i]);
    nzp(b,5);
    printf("\nlast values of array b are:\n");
    for(i=0;i<5;i++)
      printf("%d ",b[i]);
}
```

本程序 nzp 函数形参数组 a 没有给出长度，由 n 动态确定该长度。在 main 函数中，函数调用语句为 nzp(b，5)，其中实参 5 将赋予形参 n 作为形参数组的长度。

d. 多维数组也可以作为函数的参数。在函数定义时对形参数组可以指定每一维的长度，也可省去第一维的长度。因此，以下写法都是合法的。

```
int MA(int a[3][10])
```

或

```
int MA(int a[][10])。
```

1.1 局部变量和全局变量

在讨论函数的形参变量时曾经提到，形参变量只在被调用期间才分配内存单元，调用结束立即释放。这一点表明形参变量只有在函数内才是有效的，离开该函数就不能再使用了。这种变量有效性的范围称变量的作用域。不仅对于形参变量，C 语言中所有的量都有自己的作用域。变量说明的方式不同，其作用域也不同。C 语言中的变量，按作用域范围可分为两种，即局部变量和全局变量。

1.1.1 局部变量

局部变量也称为内部变量。局部变量是在函数内作定义说明的。其作用域仅限于函数内，离开该函数后再使用这种变量是非法的。

例如：

```
int f1(int a)          /*函数 f1*/
{
int b,c;
.....
}
```

```
    a,b,c 有效
    int f2(int x)          /*函数 f2*/
    {
    int y,z;
    ……
    }
    x,y,z 有效
    main()
    {
      int m,n;
      ……
      }
    m,n 有效
```

在函数 f1 内定义了三个变量，a 为形参，b,c 为一般变量。在 f1 的范围内 a,b,c 有效，或者说 a,b,c 变量的作用域限于 f1 内。同理，x,y,z 的作用域限于 f2 内。m,n 的作用域限于 main 函数内。关于局部变量的作用域还要说明以下几点：

1) 主函数中定义的变量也只能在主函数中使用，不能在其它函数中使用。同时，主函数中也不能使用其它函数中定义的变量。因为主函数也是一个函数，它与其它函数是平行关系。这一点是与其它语言不同的，应予以注意。
2) 形参变量是属于被调函数的局部变量，实参变量是属于主调函数的局部变量。
3) 允许在不同的函数中使用相同的变量名，它们代表不同的对象，分配不同的单元，互不干扰，也不会发生混淆。如在前例中，形参和实参的变量名都为 n，是完全允许的。
4) 在复合语句中也可定义变量，其作用域只在复合语句范围内。

例如：

```
    main()
    {
      int s,a;
      ……
    {
      int b;
      s=a+b;
      ……                   /*b 作用域*/
    }
    ……                     /*s,a 作用域*/
    }
```

【例 8.12】

```
main()
{
    int i=2,j=3,k;
    k=i+j;
    {
      int k=8;
      printf("%d\n",k);
```

```
    }
    printf("%d\n",k);
}
```

本程序在 main 中定义了 i,j,k 三个变量，其中 k 未赋初值。而在复合语句内又定义了一个变量 k，并赋初值为 8。应该注意这两个 k 不是同一个变量。在复合语句外由 main 定义的 k 起作用，而在复合语句内则由在复合语句内定义的 k 起作用。因此程序第 4 行的 k 为 main 所定义，其值应为 5。第 7 行输出 k 值，该行在复合语句内，由复合语句内定义的 k 起作用，其初值为 8，故输出值为 8，第 9 行输出 i，k 值。i 是在整个程序中有效的，第 7 行对 i 赋值为 3，故以输出也为 3。而第 9 行已在复合语句之外，输出的 k 应为 main 所定义的 k，此 k 值由第 4 行已获得为 5，故输出也为 5。

1.1.1 全局变量

全局变量也称为外部变量，它是在函数外部定义的变量。它不属于哪一个函数，它属于一个源程序文件。其作用域是整个源程序。在函数中使用全局变量，一般应作全局变量说明。只有在函数内经过说明的全局变量才能使用。全局变量的说明符为 extern。但在一个函数之前定义的全局变量，在该函数内使用可不再加以说明。

例如：

```
int a,b;            /*外部变量*/
void f1()           /*函数 f1*/
{
  ……
}
float x,y;          /*外部变量*/
int fz()            /*函数 fz*/
{
  ……
}
main()              /*主函数*/
{
  ……
}
```

从上例可以看出 a、b、x、y 都是在函数外部定义的外部变量，都是全局变量。但 x,y 定义在函数 f1 之后，而在 f1 内又无对 x,y 的说明，所以它们在 f1 内无效。a,b 定义在源程序最前面，因此在 f1,f2 及 main 内不加说明也可使用。

【例 8.13】输入正方体的长宽高 l,w,h。求体积及三个面 x*y,x*z,y*z 的面积。

```
int s1,s2,s3;
int vs( int a,int b,int c)
{
    int v;
    v=a*b*c;
    s1=a*b;
```

```
    s2=b*c;
    s3=a*c;
    return v;
}
main()
{
 int v,l,w,h;
 printf("\ninput length,width and height\n");
 scanf("%d%d%d",&l,&w,&h);
 v=vs(l,w,h);
 printf("\nv=%d,s1=%d,s2=%d,s3=%d\n",v,s1,s2,s3);
}
```

【例 8.14】外部变量与局部变量同名。

```
int a=3,b=5;      /*a,b 为外部变量*/
max(int a,int b) /*a,b 为外部变量*/
{int c;
 c=a>b?a:b;
 return(c);
}
main()
{int a=8;
 printf("%d\n",max(a,b));
}
```

如果同一个源文件中，外部变量与局部变量同名，则在局部变量的作用范围内，外部变量被“屏蔽”，即它不起作用。

1.1 变量的存储类别

1.1.1 动态存储方式与静态动态存储方式

前面已经介绍了，从变量的作用域（即从空间）角度来分，可以分为全局变量和局部变量。

从另一个角度，从变量值存在的作时间（即生存期）角度来分，可以分为静态存储方式和动态存储方式。

静态存储方式：是指在程序运行期间分配固定的存储空间的方式。

动态存储方式：是在程序运行期间根据需要进行动态的分配存储空间的方式。

用户存储空间可以分为三个部分：

1) 程序区；

2) 静态存储区；

3)　动态存储区；

全局变量全部存放在静态存储区，在程序开始执行时给全局变量分配存储区，程序行完毕就释放。在程序执行过程中它们占据固定的存储单元，而不动态地进行分配和释放；

动态存储区存放以下数据：

1)　函数形式参数；

2)　自动变量（未加 static 声明的局部变量）；

3)　函数调用实的现场保护和返回地址；

对以上这些数据，在函数开始调用时分配动态存储空间，函数结束时释放这些空间。

在 c 语言中，每个变量和函数有两个属性：数据类型和数据的存储类别。

1.1.1 auto 变量

函数中的局部变量，如不专门声明为 static 存储类别，都是动态地分配存储空间的，数据存储在动态存储区中。函数中的形参和在函数中定义的变量（包括在复合语句中定义的变量），都属此类，在调用该函数时系统会给它们分配存储空间，在函数调用结束时就自动释放这些存储空间。这类局部变量称为自动变量。自动变量用关键字 auto 作存储类别的声明。

例如：

```
int f(int a)              /*定义 f 函数，a 为参数*/
{auto int b,c=3;          /*定义 b，c 自动变量*/
 ……
}
```

a 是形参，b，c 是自动变量，对 c 赋初值 3。执行完 f 函数后，自动释放 a，b，c 所占的存储单元。

关键字 auto 可以省略，auto 不写则隐含定为“自动存储类别”，属于动态存储方式。

1.1.1 用 static 声明局部变量

有时希望函数中的局部变量的值在函数调用结束后不消失而保留原值，这时就应该指定局部变量为“静态局部变量”，用关键字 static 进行声明。

【例 8.15】考察静态局部变量的值。

```
f(int a)
{auto b=0;
 static c=3;
 b=b+1;
 c=c+1;
 return(a+b+c);
}
```

```
main()
{int a=2,i;
 for(i=0;i<3;i++)
 printf("%d",f(a));
}
```

对静态局部变量的说明：

1) 静态局部变量属于静态存储类别，在静态存储区内分配存储单元。在程序整个运行期间都不释放。而自动变量（即动态局部变量）属于动态存储类别，占动态存储空间，函数调用结束后即释放。
2) 静态局部变量在编译时赋初值，即只赋初值一次；而对自动变量赋初值是在函数调用时进行，每调用一次函数重新给一次初值，相当于执行一次赋值语句。
3) 如果在定义局部变量时不赋初值的话，则对静态局部变量来说，编译时自动赋初值（对数值型变量）或空字符（对字符变量）。而对自动变量来说，如果不赋初值则它的值是一个不确定的值。

【例 8.16】打印 1 到 5 的阶乘值。

```
int fac(int n)
{static int f=1;
 f=f*n;
 return(f);
}
main()
{int i;
 for(i=1;i<=5;i++)
 printf("%d!=%d\n",i,fac(i));
}
```

1.1.1 register 变量

为了提高效率，C 语言允许将局部变量得值放在 CPU 中的寄存器中，这种变量叫"寄存器变量"，用关键字 register 作声明。

【例 8.17】使用寄存器变量。

```
int fac(int n)
{register int i,f=1;
 for(i=1;i<=n;i++)
    f=f*i
 return(f);
}
main()
{int i;
 for(i=0;i<=5;i++)
```

```
 printf("%d!=%d\n",i,fac(i));
}
```

说明：

1) 只有局部自动变量和形式参数可以作为寄存器变量；
2) 一个计算机系统中的寄存器数目有限，不能定义任意多个寄存器变量；
3) 局部静态变量不能定义为寄存器变量。

1.1.1 用 extern 声明外部变量

外部变量（即全局变量）是在函数的外部定义的，它的作用域为从变量定义处开始，到本程序文件的末尾。如果外部变量不在文件的开头定义，其有效的作用范围只限于定义处到文件终了。如果在定义点之前的函数想引用该外部变量，则应该在引用之前用关键字 extern 对该变量作“外部变量声明”。表示该变量是一个已经定义的外部变量。有了此声明，就可以从“声明”处起，合法地使用该外部变量。

【例 8.18】用 extern 声明外部变量，扩展程序文件中的作用域。

```
int max(int x,int y)
{int z;
 z=x>y?x:y;
 return(z);
}
main()
{extern A,B;
 printf("%d\n",max(A,B));
}
int A=13,B=-8;
```

说明：在本程序文件的最后 1 行定义了外部变量 A，B，但由于外部变量定义的位置在函数 main 之后，因此本来在 main 函数中不能引用外部变量 A，B。现在我们在 main 函数中用 extern 对 A 和 B 进行“外部变量声明”，就可以从“声明”处起，合法地使用该外部变量 A 和 B。