第4章 函 数

在学习C语言函数以前，我们需要了解什么是模块化程序设计方法。

人们在求解一个复杂问题时，通常采用的是逐步分解、分而治之的方法，也就是把一个大问题分解成若干个比较容易求解的小问题，然后分别求解。程序员在设计一个复杂的应用程序时，往往也是把整个程序划分为若干功能较为单一的程序模块，然后分别予以实现，最后再把所有的程序模块像搭积木一样装配起来，这种在程序设计中分而治之的策略，被称为模块化程序设计方法。

在C语言中，函数是程序的基本组成单位，因此可以很方便地用函数作为程序模块来实现C语言程序。

利用函数，不仅可以实现程序的模块化，程序设计得简单和直观，提高了程序的易读性和可维护性，而且还可以把程序中普通用到的一些计算或操作编成通用的函数，以供随时调用，这样可以大大地减轻程序员的代码工作量。

函数是C语言的基本构件，是所有程序活动的舞台。函数的一般形式是：

```
type-specifier function_name(parameter list)
parameter declarations
{
 body of the function
}
```

类型说明符定义了函数中return语句返回值的类型，该返回值可以是任何有效类型。如果没有类型说明符出现，函数返回一个整型值。参数表是一个用逗号分隔的变量表，当函数被调用时这些变量接收调用参数的值。一个函数可以没有参数，这时函数表是空的。但即使没有参数，括号仍然是必须要有的。参数说明段定义了其中参数的类型。

4.1 函数说明与返回值

当一个函数没有明确说明类型时，C语言的编译程序自动将整型（int）作为这个函数的缺省类型，缺省类型适用于很大一部分函数。当有必要返回其它类型数据时，需要分两步处理：首先，必须给函数以明确的类型说明符；其次，函数类型的说明必须处于对它的首次调用之前。只有这样，C编译程序才能为返回非整型的值的函数生成正确代码。

4.1.1 函数的类型说明

可将函数说明为返回任何一种合法的C语言数据类型。

类型说明符告诉编译程序它返回什么类型的数据。这个信息对于程序能否正确运行关系极大，因为不同的数据有不同的长度和内部表示。

返回非整型数据的函数被使用之前，必须把它的类型向程序的其余部分说明。若不这样做，C语言的编译程序就认为函数是返回整型数据的函数，调用点又在函数类型说明之前，编译程序就会对调用生成错误代码。为了防止上述问题的出现，必须使用一个特别的说明语句，

通知编译程序这个函数返回什么值。下例示出了这种方法。

[例4-1]

```
float sum( );    /*函数说明 */
main ( )
{
 float first, second;
 first =123.23;
 second=99.09  ;
 printf ("%f", sum (first, second)) ;
}
float sum (a, b)  /*函数定义*/
float a, b;
{
return a+b;
}
```

第一个函数的类型说明sum( )函数返回浮点类型的数据。这个说明使编译程序能够对sum( )的调用产生正确代码。

函数类型说明语句的一般形式是:

```
type_specifier function_name (;)
```

即使函数使用形参，也不要将其写入说明句。若未使用类型说明语句，函数返回的数据类型可能与调用者所要求的不一致，其结果是难以预料的。如果两者同处于一个文件中，编译程序可以发现该错误并停止编译。如果不在同一个文件中，编译程序无法发现这种错误。类型检查仅在编译中进行，链接和运行时均不检查。因此，必须十分细心以确保绝不发生上述错误。

当被说明为整型的函数返回字符时，这个字符值被转换为整数。因为 C语言以不加说明的方式进行字符型与整型之间的数据转换，因而多数情况下，返回字符值的函数并不是说明为返回字符值，而是由函数的这种字符型向整型的缺省类型转换隐含实现的。

4.1.2 返回语句

返回语句return有两个重要用途。第一，它使得内含它的那个函数立即退出，也就是使程序返回到调用语句处继续进行。第二，它可以用来回送一个数值。本章将说明这两个用途。

1. 从函数返回

函数可以用两种方法停止运行并返回到调用程序。第一种是在执行完函数的最后一个语句之后，从概念上讲，是遇到了函数的结束符“ }”(当然这个花括号实际上并不会出现在目标码中，但我们可以这样理解)。例如，下面的函数在屏幕上显示一个字符串。

[例4-2]

```
pr_reverse ()
{
 char s[80];     /*定义一个字符数组*/
 scanf("%s"  , s) ;            /*输入一个字符串，其长度不超过79个字符*/
 printf("%s\n"  , s) ;
}
```

一旦字串显示完毕，函数就没事可做了，这时它返回到被调用处。

在实际情况中，没有多少函数是以这种缺省方式终止运行的。因为有时必须送回一个值，大多数函数用return语句终止运行，有时在函数中设立了多个终止点以简化函数、提高效率。切记，一个函数可以有多个返回语句。如下所示，函数在s1、s2相等时返回1，不相等时返回-1。

[例4-3]

```
find_char(s1 , s2)
char s1, s2 ;
{
  if(s1==s2)
      return  1;
    else
       return -1;
}
```

2. 返回值

所有的函数，除了空值类型外，都返回一个数值（切记，空值是 ANSI建议标准所做的扩展，也许并不适合读者手头的 C编译程序）。该数值由返回语句确定。无返回语句时，返回值是0。这就意味着，只要函数没有被说明为空值，它就可以用在任何有效的 C语言表达式中作为操作数。这样下面的表达式都是合法的 C语言表达式。

```
x = power (y);
if  (max (x, y) >100) printf("greater");
for (ch=getchar( ); isdigit (ch); )... ;
```

可是，函数不能作为赋值对象，下列语句是错误的：

```
swap(x , y) =100;
```

C编译程序将认为这个语句是错误的，而且对含有这种错误语句的程序不予编译。

所有非空值的函数都会返回一个值。我们编写的程序中大部分函数属于三种类型。第一种类型是简单计算型 —— 函数设计成对变量进行运算，并且返回计算值。计算型函数实际上是一个“纯”函数，例如sqr( )和sin( )。第二类函数处理信息，并且返回一个值，仅以此表示处理的成功或失败。例如write( )，用于向磁盘文件写信息。如果写操作成功了，write( )返回写入的字节数，当函数返回- 1时，标志写操作失败。最后一类函数没有明确的返回值。实际上这类函数是严格的过程型函数，不产生值。如果读者用的是符合 ANSI建议标准的C编译程序，那么所有这一类函数应当被说明为空值类型。奇怪的是，那些并不产生令人感兴趣的结果的函数却无论如何也要返回某些东西。例如 printf( )返回被写字符的个数。然而，很难找出一个真正检查这个返回值的程序。因此，虽然除了空值函数以外的所有函数都返回一个值，我们却不必非得去使用这个返回值。有关函数返回值的一个常见问题是：既然这个值是被返回的，我是不是必须把它赋给某个变量？回答是：不必。如果没有用它赋值，那它就被丢弃了。请看下面的程序，它使用了 mul( )函数。mul( )函数定义为：int mul(int x, int y){.....}

[例4-4]

```
main( )
{
int x, y, z;
x=10 ; y=20 ;
```

```
z=mul(x , y) ;                    /* 1  */
printf("%d"  , mul(x , y)) ;      /* 2  */
mul(x , y) ;                      /*  3  */
}
```

在第一行，mul( )的返回值被赋予 z，在第二行中，返回值实际上没有赋给任何变量，但被printf( )函数所使用。最后，在第三行，返回值被丢弃不用，因为既没有把它赋给第一个变量，也没有把它用作表达式中的一部分。

4.2 函数的作用域规则

“语言的作用域规则”是一组确定一部分代码是否“可见”或可访问另一部分代码和数据的规则。

C语言中的每一个函数都是一个独立的代码块。一个函数的代码块是隐藏于函数内部的，不能被任何其它函数中的任何语句（除调用它的语句之外）所访问（例如，用 goto语句跳转到另一个函数内部是不可能的）。构成一个函数体的代码对程序的其它部分来说是隐蔽的，它既不能影响程序其它部分，也不受其它部分的影响。换言之，由于两个函数有不同的作用域，定义在一个函数内部的代码数据无法与定义在另一个函数内部的代码和数据相互作用。

C语言中所有的函数都处于同一作用域级别上。这就是说，把一个函数定义于另一个函数内部是不可能的。

4.2.1 局部变量

在函数内部定义的变量成为局部变量。在某些 C语言教材中，局部变量称为自动变量，这就与使用可选关键字 auto定义局部变量这一作法保持一致。局部变量仅由其被定义的模块内部的语句所访问。换言之，局部变量在自己的代码模块之外是不可知的。切记：模块以左花括号开始，以右花括号结束。

对于局部变量，要了解的最重要的东西是：它们仅存在于被定义的当前执行代码块中，即局部变量在进入模块时生成，在退出模块时消亡。

定义局部变量的最常见的代码块是函数。例如，考虑下面两个函数。

[例4-5]

```
func1()
        {
                int x;    /* 可定义为 auto int x;  */
                x=10;
        }
func2()
        {
                int x;    /* 可定义为 auto int x;  */
                x=-1999;
        }
```

整数变量x被定义了两次，一次在 func1()中，一次在func2()中。func1()和func2()中的x互不相关。其原因是每个 x作为局部变量仅在被定义的块内可知。

语言中包括了关键字auto，它可用于定义局部变量。但自从所有的非全局变量的缺省值假定为auto以来，auto就几乎很少使用了，因此在本书所有的例子中，均见不到这一关键字。

在每一函数模块内的开始处定义所有需要的变量，是最常见的作法。这样做使得任何人读此函数时都很容易，了解用到的变量。但并非必须这样做不可，因为局部变量可以在任何模块中定义。为了解其工作原理，请看下面函数。

[例4-6]

```
f()
 {
        int t;
        scanf("%d",&t);
        if(t==1){
              char s[80];            此变量仅在这个块中起作用*/
              printf("enter name:");
              gets(s);               /* 输入字符串*/
              process(s);            /* 函数调用*/
        }
   }
```

这里的局部变量s就是在if块入口处建立，并在其出口处消亡的。因此 s仅在if块中可知，而在其它地方均不可访问，甚至在包含它的函数内部的其它部分也不行。

在一个条件块内定义局部变量的主要优点是仅在需要时才为之分配内存。这是因为局部变量仅在控制转到它们被定义的块内时才进入生存期。虽然大多数情况下这并不十分重要，但当代码用于专用控制器（如识别数字安全码的车库门控制器）时，这就变得十分重要了，因为这时随机存储器（RAM）极其短缺。

由于局部变量随着它们被定义的模块的进出口而建立或释放，它们存储的信息在块工作结束后也就丢失了。切记，这点对有关函数的访问特别重要。当访问一函数时，它的局部变量被建立，当函数返回时，局部变量被销毁。这就是说，局部变量的值不能在两次调用之间保持。

4.2.2 全局变量

与局部变量不同，全局变量贯穿整个程序，并且可被任何一个模块使用。它们在整个程序执行期间保持有效。全局变量定义在所有函数之外，可由函数内的任何表达式访问。在下面的程序中可以看到，变量count定义在所有函数之外，函数main()之前。但其实它可以放置在任何第一次被使用之前的地方，只要不在函数内就可以。实践表明，定义全局变量的最佳位置是在程序的顶部。

[例4-7]

```
int count;    /*count 是全局变量 */
main()
  {
       count = 100;
       func1();
  }
func1()
```

```
{
     int temp;
     temp = count;
     func2();
     printf("count is %d",count);     /* 打印 100 */
}
func2()
{
     int count;
     for(count = 1; count < 10; count++)
        putchar('.');                  /*  打印出"。"  */
}
```

仔细研究此程序后，可见变量 count既不是main()也不是func1()定义的，但两者都可以使用它。函数func2()也定义了一个局部变量 count。当func2访问count时，它仅访问自己定义的局部变量count，而不是那个全局变量 count。切记，全局变量和某一函数的局部变量同名时，该函数对该名的所有访问仅针对局部变量，对全局变量无影响，这是很方便的。然而，如果忘记了这点，即使程序看起来是正确的，也可能导致运行时的奇异行为。

全局变量由C编译程序在动态区之外的固定存储区域中存储。当程序中多个函数都使用同一数据时，全局变量将是很有效的。然而，由于三种原因，应避免使用不必要的全局变量：①不论是否需要，它们在整个程序执行期间均占有存储空间。②由于全局变量必须依靠外部定义，所以在使用局部变量就可以达到其功能时使用了全局变量，将降低函数的通用性，这是因为它要依赖其本身之外的东西。③大量使用全局变量时，不可知的和不需要的副作用将可能导致程序错误。如在编制大型程序时有一个重要的问题：变量值都有可能在程序其它地点偶然改变。

结构化语言的原则之一是代码和数据的分离。 C语言是通过局部变量和函数的使用来实现这一分离的。下面用两种方法编制计算两个整数乘积的简单函数 mul()。

通用的

```
mul(x,y)
int x,y;
 {
   return(x*y);
 }
```

专用的

```
int x,y;
mul()
  {
   return(x*y);
  }
```

两个函数都是返回变量x和y的积，可通用的或称为参数化版本可用于任意两整数之积，而专用的版本仅能计算全局变量 x和y的乘积。

4.2.3 动态存储变量

从变量的作用域原则出发，我们可以将变量分为全局变量和局部变量；换一个方式，从变量的生存期来分，可将变量分为动态存储变量及静态存储变量。

动态存储变量可以是函数的形式参数、局部变量、函数调用时的现场保护和返回地址。这些动态存储变量在函数调用时分配存储空间，函数结束时释放存储空间。动态存储变量的定义形式为在变量定义的前面加上关键字“ auto ”，例如：

```
auto int a, b, c;
```

“auto”也可以省略不写。事实上，我们已经使用的变量均为省略了关键字“ auto”的动态存储变量。有时我们甚至为了提高速度，将局部的动态存储变量定义为寄存器型的变量，定义的形式为在变量的前面加关键字“ register”，例如：

```
register int x, y, z;
```

这样一来的好处是：将变量的值无需存入内存，而只需保存在 CPU内的寄存器中，以使速度大大提高。由于CPU内的寄存器数量是有限的，不可能为某个变量长期占用。因此，一些操作系统对寄存器的使用做了数量的限制。或多或少，或根本不提供，用自动变量来替代。

4.2.4 静态存储变量

在编译时分配存储空间的变量称为静态存储变量，其定义形式为在变量定义的前面加上关键字“static”，例如：

```
static int a=8;
```

定义的静态存储变量无论是做全程量或是局部变量，其定义和初始化在程序编译时进行。作为局部变量，调用函数结束时，静态存储变量不消失并且保留原值。

[例 4-8]

```
main( )
{
inf f( ); /*函数声明*/
int j;
for (j=0; j<3; j++)
printf ("%d\n", f( ));
}
int f( )  /*无参函数*/
{
static int x=1;
x++;
return x;
}
```

运行程序:

```
RUN
2
3
4
```

从上述程序看，函数 f( )被三次调用，由于局部变量 x是静态存储变量，它是在编译时分配存储空间，故每次调用函数 f( )时，变量x不再重新初始化，保留加 1后的值，得到上面的输出。

4.3 函数的调用与参数

如果一个函数要使用参数，它就必须定义接受参数值的变量。

4.3.1 形式参数与实际参数

函数定义时填入的参数我们称之为形式参数，简称形参，它们同函数内部的局部变量作用相同。形参的定义是在函数名之后和函数开始的花括号之前。

调用时填入的参数，我们称之为实际参数，简称实参。

必须确认所定义的形参与调用函数的实际参数类型一致，同时还要保证在调用时形参与实参的个数出现的次序也要一一对应。如果不一致，将产生意料不到的结果。与许多其它高级语言不同，(是健壮的，它总要做一些甚至你不希望的事情，几乎没有运行时错误检查，完全没有范围检测。作为程序员，必须小心行事以保证不发生错误，安全运行。

4.3.2 赋值调用与引用调用

一般说来，有两种方法可以把参数传递给函数。第一种叫做“赋值调用”(call by value)，这种方法是把参数的值复制到函数的形式参数中。这样，函数中的形式参数的任何变化不会影响到调用时所使用的变量。

把参数传递给函数的第二种方法是“引用调用”(call by reference)。这种方法是把参数的地址复制给形式参数，在函数中，这个地址用来访问调用中所使用的实际参数。这意味着，形式参数的变化会影响调用时所使用的那个变量(详细内容请参见后续章节)。

除少数情况外，C语言使用赋值调用来传递参数。这意味着，一般不能改变调用时所用变量的值。请看例4-9。

[例4-9]

```
main ( )
{
 int t =10;
 printf("%d %d "; sqr(t) , t) ;    /*  sqr(t)是函数调用，t是实参*/
}
int sqr(x)   /*  函数定义，x是形式参数*/
    int x;
{
 x=x*x ;
 return (x);
}
```

在这个例子里，传递给函数sqr( )的参数值是复制给形式参数x的，当赋值语句x=x*x执行时，仅修改局部变量x。用于调用sqr()的变量t，仍然保持着值10。

执行程序：

```
RUN
100  10
```

切记，传给函数的只是参数值的复制品。所有发生在函数内部的变化均无法影响调用时使用的变量。

4.4 递归

C语言函数可以自我调用。如果函数内部一个语句调用了函数自己，则称这个函数是“递

归”。递归是以自身定义的过程。也可称为“循环定义”。

递归的例子很多。例如定义整数的递归方法是用数字1，2，3，4，5，6，7，8，9加上或减去一个整数。例如，数字15是7+8；数字21是9+12； 数字12是9+3。

一种可递归的计算机语言，它的函数能够自己调用自己。一个简单的例子就是计算整数阶乘的函数factor( )数N的阶乘是1到N之间所有数字的乘积。例如3的阶乘是1×2×3，即是6。factor( )和其等效函数fact( )如例4-10所示。

[例4-10]

```
factor(n)                    /* 递归调用方法 */
int n;
{
  int answer;
  if (n==1)
       return (1);
  answer=factor(n-1) * n; /* 函数自身调用 */
  return(answer)   ;
}
```

[例4-11]

```
fact(n)       /* 非递归方法 */
int n;
{
  int t, answer ;
  answer=1  ;
  for (t=1;  t<=n ;  t++)
     answer = answer * t;
  return(answer)   ;
 }
```

非递归函数fact( )的执行应该是易于理解的。它应用一个从1开始到指定数值结束的循环。在循环中，用“变化”的乘积依次去乘每个数。

factor( )的递归执行比fact( )稍复杂。当用参数1调用factor( )时，函数返回1；除此之外的其它值调用将返回factor(n-1) * n这个乘积。为了求出这个表达式的值，用（n-1）调用factor( )一直到n等于1，调用开始返回。

计算2的阶乘时对factor( )的首次调用引起了以参数1对factor( )的第二次调用。这次调用返回1，然后被2乘（n的初始值），答案是2（把printf( )语句插入到factor ( )中，察看各级调用及其中间答案，是很有趣的）。

当函数调用自己时，在栈中为新的局部变量和参数分配内存，函数的代码用这些变量和参数重新运行。递归调用并不是把函数代码重新复制一遍，仅仅参数是新的。当每次递归调用返回时，老的局部变量和参数就从栈中消除，从函数内此次函数调用点重新启动运行。可递归的函数被说成是对自身的“推入和拉出”。

大部分递归例程没有明显地减少代码规模和节省内存空间。另外，大部分例程的递归形式比非递归形式运行速度要慢一些。这是因为附加的函数调用增加了时间开销（在许多情况下，速度的差别不太明显）。对函数的多次递归调用可能造成堆栈的溢出。不过溢出的可能性

不大，因为函数的参数和局部变量是存放在堆栈中的。每次新的调用就会产生一些变量的复制品。这个堆栈冲掉其它数据和程序的存储区域的可能性是存在的。但是除非递归程序运行失控，否则不必为上述情况担心。

递归函数的主要优点是可以把算法写的比使用非递归函数时更清晰更简洁，而且某些问题，特别是与人工智能有关的问题，更适宜用递归方法。递归的另一个优点是，递归函数不会受到怀疑，较非递归函数而言，某些人更相信递归函数。编写递归函数时，必须在函数的某些地方使用if语句，强迫函数在未执行递归调用前返回。如果不这样做，在调用函数后，它永远不会返回。在递归函数中不使用 if语句，是一个很常见的错误。在开发过程中广泛使用printf( )和getchar( )可以看到执行过程，并且可以在发现错误后停止运行。

4.5 实现问题

在编写C语言的函数时，有几个要点需要我们牢记，因为它们影响到函数的效率和可用性。

4.5.1 参数和通用函数

通用函数是指能够被用在各种情况下，或者是可被许多不同程序员使用的函数。我们不应该把通用函数建立在全局变量上（不应该在通用函数中使用全局变量）。函数所需要的所有数据都应该用参数传递（在个别难以这样做的情况下，可以使用静态变量）。使用参数传递，除了有助于函数能用在多种情况下之外，还能提高函数代码的可读性。不用全局变量，可以使得函数减少因副作用而导致错误的可能性。

4.5.2 效率

函数是C语言的基本构件。对于编写简单程序之外的所有程序来说，函数是必不可少的。但在一些特定的应用中，应当消除函数，而采用内嵌代码。内嵌代码是指一个函数的语句中不含函数调用语句。仅当执行速度是很关键的场合下，才用内嵌代码而不用函数。

有两个原因使得内嵌代码的执行速度比函数快。首先，调用需要花费时间；其次，如果有参数需要传递，就要把它们放在堆栈中，这也要用时间。在几乎所有的应用中，执行时间上的这些微小开销是微不足道的。不过当时间开销至关重要时，使用内嵌代码消除函数调用，可以把每次函数调用的开销节省下来。下面的两个程序都是打印从 1到10的数的平方。由于函数调用需要花费时间，所以内嵌代码版本运行的比另一个要快。

内嵌

```
main (   )
{
int x;
for (x=1, x<11 ; ++x)
     printf ("%d", x*x) ;
}
```

函数调用

```
 main (   )
{
    int x;
    for (xx=1; x<11 ; ++x)
          printf ("%d", sqr(x)) ;
   }
sqr(a) ;
int a;
{
   return a*a;
   }
```

4.6 函数库和文件

一个函数设计完后，我们可以用三种方法处理它： 1) 把它放在main( )函数的同一个文件中；2) 把它和写好的其它函数一起放在另一个文件中； 3) 把它放在函数库中。下面分别讨论这三种方法。

4.6.1 程序文件的大小

因为C语言允许分别编译，很自然就会提出这样的问题：一个文件的最适宜的规模是多大？这规模很重要，因为编译时间与被编译文件的大小直接相关。一般说来，链接处理的时间比编译处理的时间短得多，且不需要经常去重新编译已经运行过的代码；另一方面，不得不同时处理多个文件也确实是件厌烦的事。

问题的答案是，每个用户、每个编译程序、每个操作系统环境都是不同的。可是对大部分微型机和一般的C编译程序来说。源程序文件不应长于 10000个字节，建立短于5000个字节的文件，可以避免不少麻烦。

4.6.2 分类组织文件

在开发一个大型程序时，最令人烦恼的而又是最常遇到的工作之一就是需要检查每个文件，以确定某个函数的存放。在程序开发的早期做一点文件组织工作就可以避免这一问题。

首先可以把概念上有关的函数组织到一个文件中。如果在编写正文编辑程序时，把删除正文所用的所有函数放进另一个文件，等等。

第二，把所有的通用函数放在一起。例如，在数据库程序中，输入／输出格式编排函数是被其它函数调用的通用函数，应把它们放进一个单独的文件里。

第三，把最高层函数放进一个单独的文件中，如果空间允许，就和 main ( )放在一起。最高层函数被用来启动程序的总体活动。这些例程从本质上定义了程序的操作。

4.6.3 函数库

从技术上讲，函数库与分别编译的函数文件不同。当库中例程被链接到程序中，或当使用一个分别编译的文件时，文件中的所有函数都被装入和链接到程序中去。对自己创建的函数文件中的大多数文件来说，文件中所有的函数都是要用到的。而对 C的标准函数库，永远也无法把所有的函数都连接到自己的程序中去，因为目的码会大得吓人！

有时候我们需要建立一个函数库，例如，假定已经完成了一套专门的统计函数，如果当前开发的某个程序仅仅需要求出一批数值的均值，我们就不必把这些函数全部装入。在这种情况下，函数库是很有用的。

大部分C语言的编译程序都有建立函数库的指令。操作过程因编译程序不同而异，可从用户手册中寻找建库的具体步骤。

4.7 C语言的预处理程序与注释

C程序的源代码中可包括各种编译指令，这些指令称为预处理命令。虽然它们实际上不是C语言的一部分，但却扩展了 C程序设计的环境。本节将介绍如何应用预处理程序和注释简化

程序开发过程，并提高程序的可读性。

4.7.1 C语言的预处理程序

ANSI 标准定义的C语言预处理程序包括下列命令：

```
#define
#error
#include
#if
#else
#elif
#endif
#ifdef
#ifndef
#undef
#line
#pragma
```

非常明显，所有预处理命令均以符号＃开头，下面分别加以介绍。

4.7.2 #define

命令＃define定义了一个标识符及一个串。在源程序中每次遇到该标识符时，均以定义的串代换它。ANSI标准将标识符定义为宏名，将替换过程称为宏替换。命令的一般形式为：

```
#define  identifier string
```

注意，该语句没有分号。在标识符和串之间可以有任意个空格，串一旦开始，仅由一新行结束。

例如，如希望TURE取值１，FALSE 取值０，可说明两个宏#define

```
#define TURE  1
#define FALSE 0
```

这使得在源程序中每次遇到TURE或FALSE就用０或１代替。

例如，在屏幕上打印“０１２”:

```
printf("%d %d %d", FALSE , TRUE , TRUE+1) ;
```

宏名定义后，即可成为其它宏名定义中的一部分。例如，下面代码定义了 ONE、TWO及THREE的值。

```
#define        ONE          1
#define        TWO          ONE+ONE
#define        THREE        ONE+TWO
```

懂得宏替换仅仅是以串代替标识符这点很重要。因此，如果希望定义一个标准错误信息，可编写如下代码：

```
#define  E_MS "standard error on input\n"
printf(E_MS)  ;
```

编译程序遇到标识符 E_MS时，就用“ standard error on input\n ”替换。对于编译程序，printf()语句实际是如下形式：

```
printf("standard error on input\n");")
```

如果在串中含有标识符，则不进行替换。例如：

```
#define XYZ this is a test
.
.
.
printf("XYZ")  ;
```

该段不打印"this is a test"而打印"XYZ"。

如果串长于一行，可以在该行末尾用一反斜杠续行，例如：

```
#define  LONG_STRING  "this is a very long \
string that is used as an example"
```

C语言程序普遍使用大写字母定义标识符。这种约定可使人读程序时很快发现哪里有宏替换。最好是将所有的#define放到文件的开始处或独立的文件中(用#include访问)，而不是将它们分散到整个程序中。

宏代换的最一般用途是定义常量的名字和程序中的“游戏数”。例如，某一程序定义了一个数组，而它的几个子程序要访问该数组，不应直接以常量定数组大小，最好是用名字定义之（需改变数组大小时）。

```
#define  MAX_SIZE  100
float  balance[MAX_SIZE];
```

#define命令的另一个有用特性是，宏名可以取参量。每次遇到宏名时，与之相连的形参均由程序中的实参代替。例如：

[例4-12]

```
#define MIN(a, b)  (a<b) ? a : b
main()
{
  int x,  y;
        x = 10;
        y = 20;
        printf("the  minimum  is: %d",MIN(x , y)) ;
}
```

当编译该程序时，由MIN(a，b)定义的表达式被替换，x和y用作操作数，即printf()语句被代换后取如下形式：

```
printf("the  minimum  is: %d",(x<y) ? x : y);
```

用宏代换代替实在的函数的一大好处是宏替换增加了代码的速度，因为不存在函数调用的开销。但增加速度也有代价：由于重复编码而增加了程序长度。

4.7.3 #error

处理器命令#error强迫编译程序停止编译，主要用于程序调试。

4.7.4 # include

命令#include 使编译程序将另一源文件嵌入带有#include的源文件，被读入的源文件必须

用双引号或尖括号括起来。例如：

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

这两行代码均使用C编译程序读入并编译用于处理磁盘文件库的子程序。

将文件嵌入#include命令中的文件内是可行的，这种方式称为嵌套的嵌入文件，嵌套层次依赖于具体实现。

如果显式路径名为文件标识符的一部分，则仅在哪些子目录中搜索被嵌入文件。否则，如果文件名用双引号括起来，则首先检索当前工作目录。如果未发现文件，则在命令行中说明的所有目录中搜索。如果仍未发现文件，则搜索实现时定义的标准目录。

如果没有显式路径名且文件名被尖括号括起来，则首先在编译命令行中的目录内检索。如果文件没找到，则检索标准目录，不检索当前工作目录。

4.7.5 条件编译命令

有几个命令可对程序源代码的各部分有选择地进行编译，该过程称为条件编译。商业软件公司广泛应用条件编译来提供和维护某一程序的许多顾客版本。

1. #if、#else，#elif及#endif

#if 的一般含义是如果#if 后面的常量表达式为true，则编译它与#endif之间的代码，否则跳过这些代码。命令#endif 标识一个#if 块的结束，参见例4-13。

```
 #if  constant-expression
      statement sequence
#endif
```

[例4-13]

```
# define MAX 100
main(  )
{
# if  MAX>99
  printf("compiled for array greater than 99\n")
# endif
}
```

由于MAX大于99，以上程序在屏幕上显示一串消息。该例说明了一个重点：跟在 # if 后面的表达式在编译时求值，因此它必须仅含常量及已定义过的标识符，不可使用变量。表达式不许含有操作符sizeof。

else 命令的功能有点象C语言中的else； #else建立另一选择（在# if失败的情况下）。因而上面的例子可扩充，参见例4-14。

[例4-14]

```
# define  MAX  10
main ( )
{
# if  MAX>99
     printf("compiled for array greater than 99\n")
  #else
```

```
        printf("compiled for small array \ n");
    #endif
}
```

在此例中，因为MAX小于99，所以，不编译#if块，而是编译# else块，因此，屏幕上显示"compiled for small array"这一消息。

注意，# else 既是# if 块又是#else 块头。这是因为任何#if 仅有一个#endif。

#elif命令意义与ELSE IF 相同，它形成一个if else-if阶梯状语句，可进行多种编译选择。#elif 后跟一个常量表达式。如果表达式为true，则编译其后的代码块，不对其它#elif表达式进行测试。否则，顺序测试下一块。

```
#if expression
 statement sequence
#elif expression1
 statement sequence
#elif expression2
 statement sequence
#elif expression3
statement sequence
#elif expression4
#elif expression3N
 statement sequence
#endif
```

例如：下面程序利用ACTIVE_COUNTRY定义货币符号。

```
 #define  US     0
 #define  ENGLAND1
 #define  FRANCE 2
 # define  ACTIVE_COUNTRY  US
 #if  ACTIVE_COUNTRY = = US
  char currency[ ]="dollar ;"
      #elif  ACTIVE_COUNTRY= =ENGLAND
       char  currency[ ]="pound ;"
#else
   char currency[ ]="franc;"
 #endif
```

#if与#elif命令可能一直嵌套到实现规定的权限，其中#endif、#else或#elif与最近#if或#elif关联。例如，下面程序是完全有效的。

```
#if  MAX>100
  #if SERIAL_VERSION
   int port=198;
  #elif
   int port=200;
#elif
#else
 char out_buffer[100];
 #endif
```

2. # ifdef 和# ifndef

条件编译的另一种方法是用 #ifdef与#ifndef命令，它们分别表示“ 如果有定义 ”及“ 如果无定义 ”。

ifdef的一般形式是：

```
# ifdef macroname
 statement sequence
#endif
```

如果宏名在前面#define语句中已定义过，则该语句后的代码块被编译。

#ifndef的一般形式是：

```
#ifndef macroname
   statement sequence
 #endif
```

如果宏名在#define 语句中无定义，则编译该代码块。

#ifdel 与#ifndef可以用于#else 语句中，但#elif 不行。参见4-15。

[例4-15]

```
    #define  TED 10
   main ()
   {
   #ifdef TED
    printf("Hi Ted\n");
#else
  printf("Hi anyone\n");
  #endif
  #ifndef  RALPH
   printf ("RALPH not defined\n");)
  #endif
 }
```

上述代码打印“ Hi Ted ”及“ RALPH not defined ”。如果TED没有定义，则显示“ Hi anyone ”，后面是“ RALPH not defined ”。

可以像嵌套#if 那样将#ifdef 与＃ifndef 嵌套至任意深度。

4.7.6 #undef

命令 #undef 取消其后那个前面已定义过有宏名定义。一般形式为：

```
#undef macroname
```

例如：

```
# define LEN 100
#difine WIDTH 100
char array[LEN][WIDTH];
# undef  LEN
# undef  WIDTH
/ *at this point both LEN and WIDTH are undefined * /
```

直到遇到#undef 语句之前，LEN与WIDTH均有定义。

undef 的主要目的是将宏名局限在仅需要它们的代码段中。

4.7.7 #line

命令#line改变_LINE_ 与_FILE_的内容，它们是在编译程序中预先定义的标识符。

命令的基本形式如下：

```
# line number["filename"]
```

其中的数字为任何正整数，可选的文件名为任意有效文件标识符。行号为源程序中当前行号，文件名为源文件的名字。命令 #line主要用于调试及其它特殊应用。

例如，下面说明行计数从100开始；printf( ) 语句显示数102，因为它是语句#line 100后的第3行。

```
#line 100                       /* 初始化行计数器 */
main ( )                        /* 行号 100 */
{                               /*  行号101 */
  printf("%d\n"  , _LINE_) ;    /* 行号 102 */
}
```

4.7.8 #pragma

命令#pragma 为实现时定义的命令，它允许向编译程序传送各种指令。例如，编译程序可能有一种选择，它支持对程序执行的跟踪。可用 #pragma语句指定一个跟踪选择。

4.7.9 预定义的宏名

ANSI标准说明了五个预定义的宏名。它们是：

```
_LINE_
_FILE_
_DATE_
_TIME_
_STDC_
```

如果编译不是标准的，则可能仅支持以上宏名中的几个，或根本不支持。记住编译程序也许还提供其它预定义的宏名。

_LINE_及_FILE_宏指令在有关#line的部分中已讨论，这里讨论其余的宏名。

_DATE_宏指令含有形式为月/日/年的串，表示源文件被翻译到代码时的日期。

源代码翻译到目标代码的时间作为串包含在 _TIME_中。串形式为时：分：秒。

如果实现是标准的，则宏 _STDC_含有十进制常量1。如果它含有任何其它数，则实现是非标准的。

注意：宏名的书写由标识符与两边各二条下划线构成。

4.7.10 注释

在C语言中，所有的注释由字符 /*开始，以*/结束。在星号及斜杠之间不允许有空格。编译程序忽略注释开始符到注释结束符间的任何文本。例如，下面程序在屏幕上只打印“hello”。

```
 main ()
{
printf("hello")   ;
```