# 第4章　函数与程序结构

函数用于把较大的计算任务分解成若干个较小的任务，使程序人员可以在其他函数的基础上构造程序，而不需要从头做起。一个设计得当的函数可以把具体操作细节对程序中不需要知道它们的那些部分隐藏掉，从而使整个程序结构清楚，减轻了因修改程序所带来的麻烦。

C语言在设计函数时考虑了效率与易于使用这两个方面。一个 C程序一般都由许多较小的函数组成，而不是只由几个比较大的函数组成。一个程序可以驻留在一个文件中，也可以存放在多个文件中。各个文件可以单独编译并与库中已经编译过的函数装配在一起。但我们不打算详细讨论这一编译装配过程，因为具体编译与装配细节在各个编译系统中各不相同。

ANSI C标准对C语言所做的最显著的修改是在函数说明与定义这两个方面。正如第 1章所述，C语言现在已经允许在说明函数时说明变元的类型。为了使函数说明与定义匹配， ANSI C标准对函数定义的语法也做了修改。故编译程序可以查出比以前更多的错误。而且，如果变元说明得当，那么程序可以自动地进行适当的类型强制转换。

ANSI C标准进一步明确了名字的作用域规则，尤其是它要求每一个外部变量只能有一个定义。初始化做得更一般化了：现在自动数组与结构都可以初始化。

C的预处理程序的功能也得到了增强。新的预处理程序所包含的条件编译指令（一种用于从宏变元建立带引号字符串的方法）更为完整，对宏扩展过程的控制更严格。

## 4.1　函数的基本知识

下面首先设计并编写一个程序，用于把输入中包含特定的“ 模式 ”或字符串的各行打印出来（这是UNIX程序grep的特殊情况）。例如，对如下一组文本行查找包含字母字符串“ ould ”的行：

```
Ah Love! could you and I with Fate conspire
To grasp this sorry Scheme of Things entire,
Would not we shatter it to bits -- and then
Re-mould it nearer to the Heart's Desire!
```

可以产生如下输出：

```
Ah Love! could you and I with Fate conspire
Would not we shatter it to bits -- and then
Re-mould it nearer to the Heart's Desire!
```

这个程序段可以清楚地分成三部分：

```
while  ( 还有未处理的行 )
    if  ( 该行包含指定的模式 )
        打印该行
```

虽然可以把所有这些代码都放在主程序 main中，但一个更好的方法是把每一部分设计成一

个独立的函数。分别处理三个较小的部分要比处理一个大的整体容易，因为这样可以把不相关的细节隐藏在函数中，从而减少了不必要的相互影响的机会。而且这些函数也可以在其他程序中使用。

我们用函数getline来实现“还有未处理的行”，这个函数已在第1章介绍过；用printf函数来实现“打印该行”，这是一个别人早就为我们提供的函数，这意味着我们只需编写一个判定“该行包含指定的模式”的函数。

我们可以通过编写一个函数strindex(s, t)来解决这个问题，该函数返回字符串t在字符串s中出现的开始位置或位标，但当s中不包含t时，返回值为-1。由于C语言数组的下标从0开始，下标的值为0或正数，故用-1之类的负数作为失败信号是比较方便的。若以后需要更复杂的模式匹配，只需替换掉strindex函数即可，程序的其余部分可保持不动。（标准库中提供的库函数strstr的功能类似于strindex函数，只是该库函数返回的是指针而不是下标值。

在做了这样的设计后，填写程序的细节就比较简单了。下面即整个程序，读者可以看看各个部分是怎样组合在一起的。我们现在所要查找的模式是字面值字符串，它不是一种最通用的机制。我们将简单讨论一下字符数组的初始化方法，第5章将介绍如何在模式中加入可在程序运行时设置的参数。getline函数的版本也稍有不同，读者可将其与第1章所介绍的版本进行比较。

```
#include  <stdio.h>
#define  MAXLINE  100   /*最大输入行长度 */

int getline (char line[ ], int max);
int strindex(char source[ ], char searchfor[ ]);

char pattern[] = "ould";     /*要查找的模式 */

/* 找出所有与模式匹配的行 */
main ( )
{
   char  line[MAXLINE];
   int  found = 0;

   while  ( getline(line, MAXLINE) > 0 )
      if  ( strindex(line, pattern) >= 0 )  {
            printf( "%s", line);
            found++;
      }
   return  found;
}
/* getline：取一行放到s中，并返回该行的长度 */
int getline(char s[ ], int lim)
{
   int  c, i;

   i = 0;
   while  ( -- lim > 0&& ( c = getchar() ) != EOF && c != '\n' )
```

```
        s[i++] = c;
    if  (c == '\n' )
        s[i++] = c;
    s[i] = '\0';
    return  i;
}

/* strindex：返回t在s中的位置，若未找到则返回-1 */
int strindex(char s[], char t[] )
{
    int  i, j, k;
    for  ( i = 0; s[i] != '\0'; i++ )  {
           for  ( j =i, k = 0; t[k] != '\0' && s[j] ==t[k]; j++, k++ )
                ;
    if  ( k > 0 && t[k] == '\0' )
          return  i;
    }
    return  -1;
}
```

每一个函数定义均具有如下形式：

```
返回类型 函数名（变元说明表）
{
    说明序列与语句序列
}
```

函数定义的各个部分都可以缺省。最简单的函数结构如下：

```
dummy( ) { }
```

这个函数什么也不做、什么也不返回。像这种什么也不做的函数有时很有用，它可以在程序开发期间用做占位符。如果在函数定义中省略了返回类型，则缺省为 int。

程序是变量定义和函数定义的集合。函数之间的通信可以通过变元、函数返回值以及外部变量进行。函数可以以任意次序出现在源文件中。源程序可以分成多个文件，只要不把一个函数分在几个文件中就行。

return语句用于从被调用函数向调用者返回值， return之后可以跟任何表达式：

```
return 表达式；
```

在必要时要把表达式转换成函数的返回类型（结果类型）。表达式两边往往要加一对圆括号，但不是必需的，而是可选的。

调用函数可以随意忽略掉返回值。而且， return之后也不一定要跟一个表达式。在 return之后没有表达式的情况下，不向调用者返回值。当被调用函数因执行到最后的右花括号而完成执行时，控制同样返回调用者（不返回值）。如果一个函数在从一个地方返回时有返回值而从另一个地方返回时没有返回值，那么这个函数不一定非法，但可能存在问题。在任何情况下，如果一个函数不能返回值，那么它的“值”肯定是没有用的。

上面的模式查找程序从主程序 main中返回一个状态，即所匹配的字符串的数目。这个值可

以在调用该程序的环境中使用。

在不同系统上对驻留在多个源文件中的 C程序的编译与载入机制有很大的区别。例如，在UNIX系统上是用在第1章中已提到过的cc命令来完成这一任务的。假定有三个函数分别存放在名为main.c、getline.c与strindex.c的三个文件中，那么命令

```
cc  main.c getline.c strindex.c
```

用于编译这三个文件，并把目标代码分别存放在文件 main.o、getline.o与strindex.o中，然后再把这三个文件一起载入到可执行文件a.out中。如果源程序中出现了错误（比如文件 main.c中出现了错误），那么可以用命令

```
cc  main.c getline.o strindex.o
```

对main.c文件重新编译，并将编译的结果与以前已编译过的目标文件 getline.o和strindex.o一起载入。cc命令用 .c 与 .o 这两种扩展名来区分源文件与目标文件。

练习4-1　编写一个函数strrindex(s, t)，用于返回字符串t在s中最右出现的位置，如果s中不包含t，那么返回-1。

## 4.2 返回非整数值的函数

到目前为止，我们所讨论的函数均是不返回任何值（ void）或只返回int类型的值。假如一个函数必须返回其他类型的值，那么该怎么办呢？许多数值函数（如 sqrt、sin与cos等函数）返回的是double类型的值，另一些专用函数则返回其他类型的值。

为了说明让函数返回非整数值的方法，编写并使用函数 atof(s)，它用于把字符串 s转换成相应的双精度浮点数。 atof函数是atoi函数的扩充，第2章与第3章已讨论了 atoi函数的几个版本。atof函数要处理可选的符号与小数点以及整数部分与小数部分。我们这个版本并不是一个高质量的输入转换函数，它所占用的空间比我们可以使用的要多。标准库中包含了具有类似功能的 atof函数，它在头文件<stdlib.h>中说明。

首先，由于atof函数返回值的类型不是int，因此在该函数中必须说明它所返回值的类型。返回值类型的名字要放在函数名字之前：

```
#include <ctype.h>

/* 把字符串s转换成相应的双精度浮点数 */
double atof( char s[ ])
{
    double  val, power;
    int  i, sign;

    for  ( i = 0; isspace(s[i]); i++ )     /* 跳过空白 */
        ;
    sign = (s[i] == '-' ) ? -1 : 1;
   if ( s[i] == '+' || s[i] == '-' )
        i++;
    for  (val = 0.0; isdigit(s[i]); i++)
```

```
        val = 10.0 * val +(s[i] -'0' );
    if  (s [i] ] = = '.')
        i++;
    for  ( power = 1.0; isdigit(s[i]); i++)  {
        val = 10.0 * val +(s[i] -'0' );
        power *= 10.0;
    }
    return  sign * val / power;
}
```

其次，也是比较重要的，调用函数必须知道 atof函数返回的是非整数值。为了保证这一点，一种方法是在调用函数中显式说明atof函数。下面所示的基本计算器程序（仅适用于支票簿计算）中给出了这个说明，程序一次读入一行数（一行只放一个数，数的前面可能有一个正负号），并把它们加在一起，在每一次输入后把这些数的连续和打印出来：

```
#include  <stdio.h>

#define  MAXLINE  100

/* 基本计算器程序 */
main ( )
{
    double  sum, atof ( char [ ] );
    char  line[MAXLINE];
    int  getline(char line[], int max);

    sum =0;
    while  ( getline(line, MAXLINE) > 0 )
        printf( "\t%g\n", sum += atof(line) );
    return  0;
}
```

其中，说明语句

```
double  sum, atof ( char [ ] );
```

表明sum是一个double类型的变量，atof是一个具有char[ ]类型的变元且返回值类型为double的函数。

函数atof的说明与定义必须一致。如果 atof函数与调用它的主函数 main放在同一源文件中，并且具有不一致的类型，那么编译程序将会检测出这个错误。但是，如果 atof函数是独立编译的（这是一种更可能的情况），那么这种不匹配的错误就不会被检测出来， atof函数将返回double类型的值，而main函数则将之处理为int类型，从而这样所求得的结果毫无意义。

按照上述说明与定义匹配的讨论，这似乎很令人吃惊。发生不匹配现象的一个原因是，如果没有函数原型，则该函数在第一次出现的表达式中隐式说明，例如下面的表达式：

```
sum += atof(line)
```

如果在前面已经说明过的某个名字出现在某个表达式中并且左边跟一个左圆括号，那么就根据上下文认为该名字是函数名字，该函数的返回值类型为 int，但对变元没有给出上面信息。而且，

如果一个函数说明中不包含变元，比如

```
double atof ( );
```

那么也认为没有给出atof函数的变元信息，所有参数检查都被关闭。对空变元表做这种特殊的解释是为了使新的编译程序能编译比较老的C程序。但是，在新程序中也如此做是不明智的。如果一个函数有变元，那么说明它们；如果没有变元，那么使用 void。

借助恰当说明的atof函数，可以编写出函数atoi（将字符串转换成整数）：

```
/* atoi：利用atof函数把字符串s转换成整数 */
int atoi( char s[ ] )
{
   double  atof(char s[ ]);

   return  (int) atof ( s );
}
```

请注意其中说明和return语句的结构。在return语句：

```
return  表达式；
```

中的表达式的值在返回之前被转换成所在函数的类型。因此，如果对 atof函数的调用直接出现在atoi函数中的return语句中，如

```
return  atof ( s );
```

那么，由于函数atoi的返回值类型为int，系统要把atof函数的double类型的结果返回值自动转换成int类型。然而，这种操作可能会丢失信息，有些编译程序可能会为此给出警告信息。在此函数中由于采用了强制转换的方法显式地表明了所要做的转换操作，可以屏蔽有关警告信息。

练习4-2　对atof函数进行扩充，使之可以处理形如

```
123.45e-6
```

一类的科学表示法，即在浮点数后跟e或E与一个（可能有正负号的）指数。

## 4.3　外部变量

C程序由一组外部对象（外部变量或函数）组成。形容词 external与internal相对，internal用于描述定义在函数内部的函数变元以及变量。外部变量在函数外面定义，故可以在许多函数中使用。由于C语言不允许在一个函数中定义其他函数，因此函数本身是外部的。在缺省情况下，外部变量与函数具有如下性质：所有通过名字对外部变量与函数的引用（即使这种引用来自独立编译的函数）都是引用的同一对象（标准中把这一性质叫做外部连接）。在这个意义上，外部变量类似于FORTRAN语言的COMMON块或Pascal语言中在最外层分程序中说明的变量。后面将介绍如何定义只能在某个源文件使用的外部变量与函数。

由于外部变量是可以全局访问的，这就为在函数之间交换数据提供了一种可以代替函数变元与返回值的方法。任何函数都可以用名字来访问外部变量，只要这个名字已在某个地方做了说明。

如果要在函数之间共享大量的变量，那么使用外部变量要比使用一个长长的变元表更方便、

有效。然而，正如在第1章所指出的，这样使用必须充分小心，因为这样可能对程序结构产生不好的影响，而且可能会使程序在各个函数之间产生太多的数据联系。

外部变量的用途还表现在它们比内部变量有更大的作用域和更长的生存期。自动变量只能在函数内部使用，当其所在函数被调用时开始存在，当函数退出时消失。而外部变量是永久存在的，它们的值在从一次函数调用到下一次函数调用之间保持不变。因此，如果两个函数必须共享某些数据，而这两个函数都互不调用对方，那么最为方便的是，把这些共享数据作成外部变量，而不是作为变元来传递。

下面通过一个更大的例子来说明这个问题。问题是要编写一个具有加（+）、减（-）、乘（*）、除（/）四则运算功能的计算器程序。为了更易于实现，在计算器中使用逆波兰表示法来代替普通的中辍表示法（逆波兰表示法用在某些袖珍计算器中，诸如Forth与Postscript等语言也使用了逆波兰表示法）。

在使用逆波兰表示法时，所有运算符都跟在其运算分量的后面。诸如

```
(1 - 2) * (4 + 5)
```

一类的中辍可用逆波兰表示法表示成：

```
1 2 - 4 5 + *
```

在使用逆波兰表示法时不再需要圆括号，只需知道每一个运算符需要几个运算分量。

计算器程序的实现很简单。每一个运算分量都被依次下推到栈中；当一个运算符到达时，从栈中弹出相应数目的运算分量（对二元运算符是两个运算分量），把该运算符作用于所弹出的运算分量，并把运算结果再下推回栈中。例如，对上面所述逆波兰表达式，首先把1与2下推到栈中，再用两者之差-1来取代它们；然后，把4与5下推到栈中，再用两者之和9来取代它们。最后，从栈中取出栈顶的-1与9，把它们的积-9下推到栈顶。当到达输入行的末尾时，把栈顶的值弹出并打印出来。

这样，该程序的结构是一个循环，每一次循环对一个运算符及相应的运算分量执行一次操作：

```
while  ( 下一个运算符或运算分量不是文件结束指示符 )
    if  ( 数 )
        将该数下推到栈中
    else if  ( 运算符 )
        弹出所需数目的运算分量
        执行运算
        将结果下推到栈中
    else if  ( 换行符 )
        弹出并打印栈顶的值
    else
        错误
```

栈的下推与弹出操作比较简单，但是，如果把错误检测与恢复操作都加进去，那么它们就会显得很长，最好把它们设计成独立的函数，而不要把它们作为在这个程序中重复的代码段。另外还需要一个单独的函数来取下一个输入运算符或运算分量。

到目前为此还没有讨论的主要设计决策是，把栈放在哪里？即哪些函数可以直接访问它？

一种可能是把它放在主函数 main中，把栈及其当前位置作为传递给要对它进行下推或弹出弹出操作的函数。但是， main函数不需要知道控制该栈的变量信息，它只进行下推与弹出操作。因此，可以把栈及其相关信息放在外部变量中，并只供 push与pop函数访问，而不能为main函数则所访问。

把上面这段话翻译成代码很容易。如果把这个程序放在一个源文件中，那么它为如下形式：

```
#include …
#define …

用于main的函数说明

main() { … }

用于push与pop的外部变量

void push ( double f ) { … }
double pop(void) { … }

int getop(char s[ ]) { … }

被getop调用的函数
```

我们在以后将讨论怎样把这个程序分割成两个或多个源文件。

main函数主要由一个循环组成，该循环中包含了一个对运算符与运算分量进行分情形操作的switch语句，这里对switch语句的使用要比3.4节所示的例子更为典型。

```
#include  <stdio.h>
#include  <stdlib.h>       /* 供atof()函数使用 */

#define  MAXOP   100       /* 运算分量或运算符的最大大小 */
#define  NUMBER  '0'       /* 表示找到数的信号 */

int getop ( char [ ] );
void push ( double f );
double pop(void);

/* 逆波兰计算器 */
main ( )
{
   int  type;
   double  op2;
   char  s[MAXOP];

   while  ( ( type = getop(s) ) != EOF )  {
      switch  ( type )  {
      case  NUMBER:
         push(atof(s));
         break;
```

```
        case  '+':
           push ( pop() + pop());
           break;
        case  '*':
           push ( pop() * pop());
           break;
        case  '-':
           op2 = pop( );
           push ( pop() - op2);
           break;
        case  '/':
           op2 = pop( );
           if  ( op2 != 0 )
               push ( pop() / op2 );
        else
               printf ( "error: zero divisor\n" );
           break;
        case  '\n':
           printf ( "\t%.8g\n", pop( ) );
           break;
        default:
           printf ( "error: unknown command %s\n", s);
           break;
        }
     }
     return  0;
}
```

**由于 + 与 * 是两个满足交换律的运算符，因此弹出的两个运算分量的次序无关紧要，但是，- 与 / 的左右运算分量的次序则是必需的。在如下所示的函数调用中：**

```
push ( pop() - pop() );     /* 错 */
```

**对pop函数的两次调用的次序没有定义。为了保证正确的次序，必须像在 main函数中一样把其第一个值弹出到一个临时变量中。**

```
#define  MAXVAL  100      /* 栈val的最大深度  */

int  sp = 0;              /* 下一个自由栈元素位置 */
double  val[MAXVAL];      /* 值栈 */

/* push：把f下推到值栈中 */
void push ( double f )
{
   if  ( sp < MAXVAL )
       val[sp++] = f;
   else
       printf ( "error: stack full, can't push %g\n", f);
}
```

```
/* pop：弹出并返回栈顶的值 */
double pop(void);
{
   if ( sp > 0 )
       return val[--sp];
   else  {
       printf ( "error: stack empty\n" );
       return  0.0;
   }
}
```

一个变量如果在函数的外面定义，那么它就是外部变量。因此，我们把必须为 push和pop函数共享的栈和栈顶指针定义在这两个函数的外面。但 main函数本身并没有引用该栈或栈顶指针，因此将它们对它隐藏。

下面讨论getop函数的实现，它用于取下一个运算符或运算分量。这一任务比较容易。跳过空格与制表符。如果下一个字符不是数字或小数点，那么返回；否则，把这些数字字符串收集起来（其中可能包含小数点），并返回NUMBER，用这个信号表示数已经收集起来了。

```
#include  <ctype.h>

int getch(void);
void ungetch(int);

/* getop：取下一个运算符或数值运算分量 */
int getop(char s[ ] )
{
   int  i, c;

   while  ( (s[0] = c = getch() ) == ' ' || c == '\t' )
    ;
   s[1] = '\0';
   if  ( !isdigit( c ) && c != '.' )
       return c;     /* 不是数 */
       i = 0;
   if  ( isdigit( c ) )  /* 收集整数部分*/
       while  ( isdigit(s[++i] = c = getch( ) ) )
           ;
   if  (c == '.' )     /* 收集小数部分 */
       while  ( isdigit(s[++i] = c = getch( ) ) )
           ;
   s[i] = '\0';
   if  (c != EOF )
       ungetch( c );
   return  NUMBER;
}
```

这段程序中的getch与ungetch是两个什么样的函数呢？在程序中经常会出现这样的情况，一个程序在读进过多的输入之前不能确定它已经读入的输入是否足够。一个例子是在读进用于组

成数的字符的时候：在看到第一个非数字字符之前，所读入数的完整性是不能确定的。由于程序要超前读入一个字符，最后有一个字符不属于当前所要读入的数。

如果能“反读”不需要的字符，那么这个问题就能得到解决。每当程序多读进一个字符时，就可以把它推回到输入中，对代码其余部分而言就像这个字符并没有读过一样。我们可以通过编写一对相互配合的函数来比较方便地模拟反取字符操作。getch函数用于读入下一个待处理的字符，而ungetch函数则用于把字符放回到输入中，使得此后对getch函数的调用将在读新的输入之前先返回经ungetch函数放回的那些字符。

把这两个函数放在一起配合使用很简单。ungetch函数把要推回的字符放到一个共享缓冲区（字符数组）中，而getch函数在该缓冲区不空时就从中读取字符，在缓冲区为空时调用getchar函数直接从输入中读字符。为了记住缓冲区中当前字符的位置，还需要一个下标变量。

由于缓冲区与下标变量是供getch与ungetch函数共享的，在两次调用之间必须保持值不变，它们必须作成这两个函数的外部变量。这样，可以如下编写getch与ungetch函数及其共享变量：

```
#define  BUFSIZE  100

char  buf[BUFSIZE];      /* 用于unget函数的缓冲区 */
int  bufp = 0;           /* buf中下一个自由位置 */

int getch(void)          /* 取一个字符（可能是推回的字符） */
{
return  ( bufp > 0 ) ? buf[--bufp] : getchar( );
}

void ungetch(int c)       /* 把字符推回到输入中 */
{
   if  ( bufp >= BUFSIZE )
       printf ( "ungetch: too many characters\n" );
   else
       buf[bufp++] = c;
}
```

标准库中提供了函数ungetc，用于推回一个字符，第7章将对它进行讨论。为了说明更一般的方法，我们这里使用了一个数组而不是一个字符用于推回字符。

练习4-3　在有了基本框架后，对计算器程序进行扩充就比较简单了。在该程序中加入取模（%）运算符并注意负数的情况。

练习4-4　在栈操作中添加几个命令分别用于在不弹出时打印栈顶元素、复制栈顶元素以及交换栈顶两个元素的值。再增加一个命令用于清空栈。

练习4-5　增加对诸如sin、exp与pow等库函数的访问操作。有关这些库函数参见附录B.4节中的头文件<math.h>。

练习4-6　增加处理变量的命令（提供26个由单字母变量很容易）。增加一个变量存放用于最近打印的值。

**练习4-7　编写一个函数ungets(s)，用于把整个字符串推回到输入中。ungets函数要使用buf与bufp吗？它可否仅使用ungetch函数？**

**练习4-8　假定最多只要推回一个字符。请相应地修改getch与ungetch这两个函数。**

**练习4-9　上面所介绍的getch与ungetch函数不能正确地处理推回的EOF。决定当推回EOF时应具有什么性质，然后再设计实现。**

**练习4-10　另一种组织方法是用getline函数读入整个输入行，这样便无需使用getch与ungets函数。运用这一方法修改计算器程序。**

## 4.4 作用域规则

**用以构成C程序的函数与外部变量完全没有必要同时编译，一个程序可以放在几个文件中，可以从库中调入已编译过的函数。我们比较感兴趣的问题主要有：**

- **怎样编写说明才能使所说明的变量在编译时被认为是正确的？**
- **怎样安排说明才能保证在程序载入时各部分能正确相连？**
- **怎样组织说明才能使得只需一份拷贝？**
- **怎样初始化外部变量？**

**为了便于讨论这些问题，我们把计算器程序组织在若干个文件中。从实用角度看，计算器程序比较小，不值得分几个文件存放，但通过它可以很好地说明在较大的程序中所遇到的有关问题。**

**一个名字的作用域指程序中可以使用该名字的部分。对于在函数开头说明的自动变量，其作用域是说明该变量名字的函数。在不同函数中说明的具有相同名字的各个局部变量毫不相关。对于函数的参数也如此，函数参数实际上可以看作是局部变量。**

**外部变量或函数的作用域从其说明处开始一直到其所在的被编译的文件的末尾。例如，如果main、sp、val、push与pop是五个依次定义在某个文件中的函数与外部变量，即：**

```
main( )  { … }

int  sp = 0;
double  val[MAXVAL];

void push( double f )  { … }

double pop( void )  { … }
```

**那么，在push与pop这两个函数中不需做任何说明就可以通过名字来访问变量sp与val，但是，这两个变量名字不能用在main函数中，push与pop函数也不能用在main函数中。**

**另一方面，如果一个外部变量在定义之前就要使用到，或者这个外部变量定义在与所要使用它的源文件不相同的源文件中，那么要在相应的变量说明中强制性地使用关键词extern。**

**将对外部变量的说明与定义严格区分开来很重要。变量说明用于通报变量的性质（主要是变量的类型），而变量定义则除此以外还引起存储分配。如果在函数的外部包含如下说明：**

```
int  sp;
double  val[MAXVAL];
```

那么这两个说明定义了外部变量 sp与val，并为之分配存储单元，同时也用作供源文件其余部分使用的说明。另一方面，如下两行:

```
extern int  sp;
extern double  val[MAXVAL];
```

为源文件剩余部分说明了 sp是一个int类型的外部变量， val是一个double数组类型的外部变量(该数组的大小在其他地方确定)，但这两个说明并没有建立变量或为它们分配存储单元。

在一个源程序的所有源文件中对一个外部变量只能在某个文件中定义一次，而其他文件可以通过extern说明来访问它（在定义外部变量的源文件中也可以包含对该外部变量的 extern说明)。在外部变量的定义中必须指定数组的大小，但在 extern说明中则不一定要指定数组的大小。

外部变量的初始化只能出现在其定义中。

假定函数push与pop在一个文件中定义，变量 val与sp在另一个文件中定义并初始化（虽然一般不可能这样组织程序)。这些定义与说明必须把这些函数和变量捆在一起：

在文件file1中：

```
   extern int  sp;
   extern double  val [ ];

   void push( double f )  { … }

   double pop( void )  { … }
```

在文件file2中：

```
   int  sp = 0;
   double  val[MAXVAL];
```

由于文件 file1中的extern说明不仅放在函数定义的外面而且还放在它们前面，故它们适用于所有函数，这一组说明对文件 file1已足够了。如果 sp与val的定义跟在对它们的使用之后，那么也要这样来组织文件。

## 4.5 头文件

下面考虑把计算器程序分成若干个源文件。主函数 main单独放在文件main.c中，push与pop函数及它们所使用的外部变量放在第二个文件 stack.c中，getop函数放在第三个文件 getop.c中，getch与ungetch函数放在第四个文件 getch.c中。之所以把它们分开，是因为在实际程序中它们来自于一个独立编译的库。

还有一个问题需要考虑，即这些文件之间的定义与说明的共享问题。我们将尽可能使所要共享的部分集中在一起，以使得只需一个拷贝，当要对程序进行改进时也能保证程序的正确性。我们将把这些公共部分放在头文件 calc.h中，在需要使用该头文件时可以用 #include指令引入(#include指令将在4.11节介绍)。如此得到的程序形式如下所示：

头文件calc.h：

```
#define  NUMBER  '0'

void push( double );
double pop( void );
int getop( char [ ] );
int getch( void);
void ungetch( int );
```

文件main.c：

```
#include  <stdio.h>
#include  <stdlib.h>
#include  "calc.h"
#define  MAXOP  100

main( )
{
    …
}
```

文件getop.c：

```
#include  <stdio.h>
#include  <ctype.h>
#include  "calc.h"
getop( )
{
    …
}
```

文件getch.c：

```
#include  <stdio.h>
#define  BUFSIZE  100
char  buf[BUFSIZE];
int  bufp = 0;
int getch( void )
{
    …
}
void ungetch( int)
{
    …
}
```

文件stack.c：

```
#include  <stdio.h>
#include  "calc.h"
#define  MAXVAL  100
int  sp = 0;
```

```
double  val[MAXVAL];

void push( double )
{
    …
}

double pop( void)
{
    …
}
```

我们对如下两个方面做了折衷：一方面是对每一个文件只能访问它完成任务所需要的信息的要求，另一方面是维护较多的头文件比较困难的现实。对于某些中等规模的程序，最好是只使用一个头文件来存放程序中各个部分需要共享的实体，这是我们在这里所做的结论。对于比较大的程序，需要做更精心的组织，使用更多的头文件。

## 4.6 静态变量

stack.c文件中定义的变量sp与val以及getch.c文件中定义的变量buf 与bufp仅供它们各自所在的源文件中的函数使用，不能被其他函数访问。 static说明适用于外部变量与函数，用于把这些对象的作用域限定为被编译源文件的剩余部分。通过外部 static对象，可以把诸如buf与bufp一类的名字隐藏在getch-ungetch组合中，使得这两个外部变量可以被 getch与ungetch函数共享，但不能被getch与ungetch函数的调用者访问。

可以在通常的说明之前前缀以关键词 static来指定静态存储。如果把上述两个函数与两个变量放在一个文件中编译，如下：

```
static char  buf[BUFSIZE];     /* 供ungetch函数使用的缓冲区 */
static int  bufp = 0;          /* 缓冲区buf的下一个自由位置 */

int getch( void )  { …}

void ungetch( int c)  { … }
```

那么其他函数不能访问变量 buf与bufp，做这两个名字不会和同一程序中其他文件中的同名名字相冲突。基于同样的理由，可以通过把变量 sp与val说明为静态的，使这两个变量只能供进行栈操作的push与pop函数使用，而对其他文件隐藏。

外部static说明最常用于说明变量，当然它也可用于说明函数。通常情况下，函数名字是全局的，在整个程序的各个部分都可见。然而，如果把一个函数说明成静态的，那么该函数名字就不能用在除该函数说明所在的文件之外的其他文件中。

static说明也可用于说明内部变量。内部静态变量就像自动变量一样局部于某一特定函数，只能在该函数中使用，但与自动变量不同的是，不管其所在函数是否被调用，它都是一直存在的，而不像自动变量那样，随着所在函数的调用与退出而存在与消失。换而言之，内部静态变量是一种只能在某一特定函数中使用的但一直占据存储空间的变量。

练习4-11　修改getop函数，使之不再需要使用 ungetch函数。提示：使用一个内部静态变量。

## 4.7 寄存器变量

register说明用于提醒编译程序所说明的变量在程序中使用频率较高。其思想是，将寄存器变量放在机器的寄存器中，这样可以使程序更小、执行速度更快。但编译程序可以忽略此选项。

register说明如下所示：

```
register int  x;
register char  c;
```

寄存器说明只适用于自动变量以及函数的形式参数。对于后一种情况，例于如下：

```
f( register unsigned m, register long n )
{
    register int  i;
    …
}
```

在实际使用时，由于硬件环境的实际情况，对寄存器变量会有一些限制。在每一个函数中只有很少的变量可以放在寄存器中，也只有某些类型的变量可以放在寄存器中。然而，过量的寄存器说明并没有什么害处，因为对于过量的或不允许的寄存器变量说明，编译程序可以将之忽略掉。另外，不论一个寄存器变量实际上是不是存放在寄存器中，它的地址都是不能访问的（关于这一问题将在第5章讨论）。对寄存器变量的数目与类型的具体限制视不同的机器而有所不同。

## 4.8 分程序结构

C语言不是Pascal等语言意义上的分程序结构的语言，因为它不允许在函数中定义函数。但另一方面，变量可以以分程序结构的形式在函数中定义。变量的说明（包括初始化）可以跟在用于引入复合语句的左花括号的后面，而不是只能出现在函数的开始部分。以这种方式说明的变量可以隐藏在该分程序外面说明的同名变量，并在与该左花括号匹配的右花括号出现之前一直存在。例如，在如下程序段中：

```
if  ( n > 0 )  {
   int  i;     /* 说明一个新的i */

   for  ( i = 0; i < n; i++ )
    …
}
```

变量i的作用域是if语句的“真”分支，这个i与在该分程序之外说明的i无关。在分程序中说明与初始化的自动变量每当进入这个分程序时就被初始化。静态变量只在第一次进入分程序时初始化一次。

自动变量（包括形式参数）也隐藏同名的外部变量与函数。对于如下说明：

```
int  x;
```

```
int  y;

f ( double x )
{
    double  y;
    …
}
```

在函数f内，所出现的x引用的是参数，其类型为double，而在函数f之外，引用的是类型为int的外部变量。对变量y也如此。

就风格而言，最好避免出现变量名字隐藏外部作用域中同名名字的情况，否则可能会出现大量混乱与错误。

## 4.9 初始化

前面已多次提到初始化的概念，但一直没有认真讨论它。这一节在前面讨论了各种存储类的基础上总结一些初始化规则。

在没有显式初始化的情况下，外部变量与静态变量都被初始化为 0，而自动变量与寄存器变量的初值则没有定义（即，其初值是“垃圾”）。

在定义纯量变量时，可以通过在所定义的变量名字后加一个等号与一个表达式来进行初始化：

```
int   x = 1;
char  squote = '\'';
long  day = 1000L * 60L *60L * 24L;   /* 每天的毫秒数 */
```

对于外部变量与静态变量，初始化符必须是常量表达式，初始化只做一次（从概念上讲是在程序开始执行前进行初始化）。对于自动变量与寄存器变量，则在每当进入函数或分程序时进行初始化。

对于自动变量与寄存器变量，初始化符不一定限定为常量：它可以是任何表达式，其中可以包含前面已定义过的值甚至可以包含函数调用。对 3.3节介绍的二分查找程序的初始化可以用如下形式：

```
int binsearch( int x, int v[ ], int n )
{
   int  low = 0;
   int  high = n - 1;
   int  mid;
   …
}
```

来代替原来的形式：

```
int low, high, mid;

low = 0;
high = n - 1;
```

实际上，自动变量的初始化部分就是赋值语句的缩写。到底使用哪一种形式还是一个尚待

尝试的问题，我们一般使用显式的赋值语句，因为说明中的初始化符比较难以为人们发现，并且距使用点比较远。

数组的初始化也是通过说明中的初始化符完成的。数组初始化符是用花括号括住并用逗号分隔的初始化符序列。例如，当要用每一个月的天数来初始化数组 days时，可用如下变量定义：

```
int days[ ] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

当数组的大小缺省时，编译程序就通过统计花括号中初始化符的个数作为数组的长度，本例中数组的大小为12个元素。

如果初始化符序列中初始化符的个数比数组元素数少，那么对于没有得到初始化的数组元素在该数组为外部变量、静态变量与自动变量时被初始化为 0。如果初始化符序列中初始化符的个数比数组元素数多，那么就是错误的。我们既无法一次性地为多个数组元素指定一个初始化符，也不能在没有指定前面数组元素值的情况下初始化后面的数组元素。

字符数组的初始化比较特殊，可以用一个字符串来代替用花括号括住并用逗号分隔的初始化符序列：

```
char  pattern [] = "ould";
```

它是如下虽然长些但却与之等价的定义的缩写：

```
char  pattern [] = { 'o', 'u', 'l', 'd' , '\0' };
```

在此情况下，数组的大小是5（4个字符外加一个字符串结束符 '\0'）。

## 4.10 递归

C函数可以递归调用，即一个函数可以直接或间接调用自己。考虑把一个数作为字符串打印的情况。如前所述，数字是以相反的次序生成的：低位数字先于高位数字生成，但它们必须以相反的次序打印出来。

对这一问题有两种解决方法。一种方法是将生成的各个数依次存储到一数组中，然后再以相反的次序把它们打印出来，正如3.6节对itoa函数所做的那样。另一种方法是使用递归解法，用于完成这一任务的函数 printd首先调用自身处理前面的（高位）数字，然后再把后面的数字打印出来。这个版本不能处理最大的负数。

```
#include  <stdio.h>

/* printd：以十进制打印数n */
void printd(int n )
{
   if  ( n < 0 )  {
       putchar( '-' );
       n = -n;
   }
   if  ( n / 10 )
       printd( n / 10 );
   putchar( n % 10 + '0' );
}
```

当一个函数递归调用自身时，每一次调用都会得到一个与以前的自动变量集合不同的新的自动变量集合。因此，在调用 printd (123) 时，第一次调用printd的变元n = 123。它把12传递给对printd的第二次调用，后者又把 1传递给对printd的第三次调用。第三次对 printd的调用将先打印1，然后再返回到第二次调用。从第三次调用返回后的第二次调用同样先打印 2，然后再返回到第一次调用。后者打印出3后结束执行。

另一个用于说明递归的一个例子是快速排序。快速排序算法是 C. A. R. Hoare于1962年发明的。对于一个给定的数组，从中选择一个元素（叫做分区元素），并把其余元素划分成两个子集合 —— 一个是由所有小于分区元素的元素组成的子集合，另一个是由所有大于等于分区元素的元素组成的子集合。对这样两个子集合递归应用同一过程。当某个子集合中的元素数小于两个时，这个子集合不需要再排序，故递归停止。

下面这个版本的快速排序函数可能不是最快的一个，但它是最简单的一个。在每一次划分子集合时都选取各个子数组的中间元素。

```
/* qsort：以递增顺序对v[left] … v[right] 进行排序 */
void qsort( int v[], int left, int right )
{
   int  i, last;
   void swap( int v[], int i, int j );

   if  ( left >= right )    /* 若数组所包含的元素数少于两个，则什么也不做 */
       return;
   swap( v, left, (left + right)/2 );    /* 把分区元素移到v[0] */
   last = left;
   for  ( i = left+1; i <= right; i++ )    /* 分区 */
        if  ( v[i] < v[left] )
              swap( v, ++last, i );
   swap( v, left, right );    /* 恢复分区元素 */
   qsort( v, left, last-1 );
   qsort( v, last+1, right );
}
```

之所以把数组元素交换操作作为一个独立的函数 swap，是因为它在qsort函数中要使用三次。

```
/* swap：交换v[i]与v[j]的值 */
void swap( int v[], int i, int j )
{
   int  temp;

   temp = v[i];
   v[i] = v[j];
   v[j] = temp;
}
```

标准库中包含了qsort函数的一个版本，它可用于对任何类型的对象排序。

递归的中间结果不需在内存特别保存，因为在某处有一个被处理值的栈。递归的执行速度并不快，但递归代码比较紧致，要比相应的非递归代码易于编写与理解。在描述诸如树等递归

定义的数据结构时使用递归尤其方便，6.5节将给出这方面的一个例子。

练习4-12　运用printd函数的思想编写一个递归版本的itoa函数，即通过递归调用把整数转换成字符串。

练习4-13　编写一个递归版本的reverse( s )函数，把字符串s颠倒过来。

## 4.11　C预处理程序

C语言通过预处理程序提供了一些语言功能，预处理程序从理论上讲是编译过程中单独进行的第一个步骤。其中两个最常用的预处理功能是#include指令（用于在编译期间把指定文件的内容包含进当前文件中）与#define指令：（用任意字符序列取代一个标记）。本节还将介绍其他功能，如条件编译与带变元的宏。

### 4.11.1　文件包含

文件包含指令，即#include指令，使我们比较容易处理一组#define指令以及说明等。在源程序文件中，任何形如：

```
#include  "文件名"
```

或

```
#include  <文件名>
```

的行都被替换成由文件名所指定的文件的内容。如果文件名用引号括起来，那么就在源程序所在位置查找该文件；如果在这个位置没有找到该文件，或者如果文件名用尖括号<与>括起来，那么就按实现定义的规则来查找该文件。被包含的文件本身也可包含#include指令。

在源文件的开始处一般都要有一些#include指令，或包含#define语句与extern说明，或访问诸如<stdio.h>等头文件中库函数的函数原型说明。（严格地说，这些没有必要做成文件。访问头文件的细节依赖于实现。）

对于比较大的程序，#include指令是把各个说明捆在一起的优选方法。它使所有源文件都被提供以相同的定义与变量说明，从而可避免发生一些特别讨厌的错误。自然地，如果一个被包含的文件的内容做了修改，那么所有依赖于这个被包含文件的源文件都必须重新编译。

### 4.11.2　宏替换

宏定义，即#define指令，具有如下形式：

```
#define  名字  替换文本
```

它是一种最简单的宏替换——出现各个的名字都将被替换文本替换。#define指令中的名字与变量名具有相同的形式，替换文本可以是任意字符串。正常情况下，替换文本是#define指令所在行的剩余部分，但也可以把一个比较长的宏定义分成若干行，这时只需在尚待延续的行后加上一个反斜杠 \ 即可。#define指令所定义的名字的作用域从其定义点开始到被编译的源文件的结束。在宏定义中也可以使用前面的宏定义。替换只对单词进行，对括在引号中的字符串不起作用。例如，如果YES是一个被定义的名字，那么在printf ( "YES" ) 或YESMAN中不能进行替换。

用替换文本可以定义任何名字，例如：

```
#define  forever  for ( ; ; )    /* 无限循环 */
```

为无限循环定义了一个新的关键词forever。

在宏定义中也可以带变元，这样可以对不同的宏调用使用不同的替换文本。例如，可通过：

```
#define  max( A, B )  ( ( A ) > ( B ) ? ( A ) : ( B ) )
```

定义一个宏max。对max的使用看起来很像是函数调用，但宏调用是直接将替换文本插入到代码中。形式参数（在此为A与B）的每一次出现都被替换成对应的实在变元。于是，语句

```
x = max( p+q, r+s );
```

将被替换成

```
x = ( ( p+q ) > ( r+s ) ? ( p+q ) : ( r+s ) );
```

只要变元能得到一致的处理，宏定义可以用于任何数据类型。没有必要像函数那样为不同数据类型定义不同的max。

如果仔细检查一下max的展开式，那么你将注意到它存在某些缺陷。其中作为变元的表达式要重复计算两次，当表达式中会带来副作用（如含有加一运算符或输入输出）时，会出现很坏的情况。例如，

```
max( i++, j++ )   /* 错 */
```

将对每一个变元做两次加一操作。同时也必须小心，为了保证计算次序的正确性要适当使用圆括号。请读者考虑一下，对于宏定义

```
#define  square( x )  x * x     /* 错 */
```

当用square( z + 1 )调用它时会出现什么情况。

然而，宏还是很有价值的。在<stdio.h>头文件中有一个很实用的例子，getchar与putchar函数在实际上往往被定义为宏，这样可以避免在处理字符时调用函数所需的运行时开销。在<ctype.h>头文件中定义的函数也常常用宏来实现。

可以用#undef指令取消对宏名字的定义，这样做通常是为了保证一个调用所调用的是一个实际函数而不是宏：

```
#undef  getchar
int  getchar( void )  { … }
```

形式参数不能用带引号的字符串替换。然而，如果在替换文本中，参数名以#作为前缀，那么它们将被由实际变元替换的参数扩展成带引号的字符串。例如，可以将其与字符串连接运算结合起来制作调试打印宏：

```
#define  dprint( expr )  printf( #expr " = %g\n", expr )
```

当用诸如

```
dprint( x/y );
```

调用该宏时，该宏就被扩展成

```
printf( "x/y" " = %g\n", x/y );
```

其中的字符串被连接起来，即这个宏调用的效果是：

```
printf( "x/y = %g\n", x/y );
```

在实际变元中，双引号 " 被替换成 \"、反斜杠 \ 被替换成 \\，故替换后的字符串是合法的字符串常量。

预处理运算符 ##为宏扩展提供了一种连接实际变元的手段。如果替换文本中的参数用 ##相连，那么参数就被实际变元替换， ##与前后的空白字符被删除，并对替换后的结果重新扫描。例如，下面定义的宏paste用于连接两个变元：

```
#define  paste( front, back )  front ## back
```

从而宏调用paste(name, 1)的结果是建立单词name1。

关于##嵌套使用的规则比较难以掌握，详细细节请参阅附录 A。

练习4-14　定义宏swap(t, x, y)，用于交换t类型的两个变元（使用分程序结构）。

### 4.11.3　条件包含

在预处理语句中还有一种条件语句，用于在预处理中进行条件控制。这提供了一种在编译过程中可以根据所求条件的值有选择地包含不同代码的手段。

#if语句中包含一个常量整数表达式（其中不得包含sizeof、类型强制转换运算符或枚举常量），若该表达式的求值结果不等于 0时，则执行其后的各行，直到遇到 #endif、#elif或#else语句为止（预处理语句#elif类似于if语句的else if结构）。在#if语句中可以使用一个特殊的表达式 defined(名字)：当名字已经定义时，其值为 1；否则，其值为0。

例如，为了保证 hdr.h文件的内容只被包含一次，可以像下面这样用条件语句把该文件的内容包围起来：

```
#if  !defined( HDR )
#define  HDR

/* hdr.h文件的内容*/

#endif
```

被#if与#endif包含的第一行定义了名字HDR，其后的各行将会发现该名字已有定义并跳到#endif。还可以用类似的样式来避免多次重复包含同一文件。如果连续使用这种，那么每一个头文件中都可以包含它所依赖的其他头文件，而不需要它的用户去处理这种依赖关系。

下面的预处理语句序列用于测试名字 SYSTEM以确定要包含进哪一个版本的头文件：

```
#if SYSTEM == SYSV
    #define HDR  "sysv.h"
#elif SYSTEM == BSD
    #define HDR  "bsd.h"
#elif SYSTEM == MSDOS
    #define HDR  "msdos.h"
#else
    #define HDR  "default.h"
```

```
#endif
# include  HDR
```

当需要测试一个名字是否已经定义时，可以使用两个特殊的预处理语句： #ifdef与#ifndef。可以使用#ifdef将上面第一个关于#if的例子改写如下：

```
#ifdef  HDR
#define  HDR

/* hdr.h文件的内容*/

#endif
```