Microsoft C6.0 · Quick C2.50系列之一

HOPE

（共二册）

中国科学院希望高级电脑技术公司

最新Quick C · Microsoft C

高级编程指南

- 使用Quick C和Microsoft C的数据结构和编程技巧。
- 开发通用的应用程序的方法。

最新 Quick C、MicroSoft C

高级编程指南(上册)

李振格
王 江 罗 克 汤建平 编
管叶茂 校

中国科学院希望高级电脑公司

一九九一年一月

最新 Quick C、MicroSoft C

高级编程指南(下册)

李振格
王 江 罗 克 汤建平 编
管叶茂 校

中国科学院希望高级电脑公司

一九九一年一月

前 言

不管是正式出版，还是非正式出版，用户手册、操作手册、参考手册之类的书都已很多了。每当一个“强大”的软件推出时，都产生一种使用手册；当一个现有软件由于功能或性能更新或提高而出现新版本时，也会产生新手册。用户手册乃软件与用户界面(典型的是中心控制的集成环境)的用法的全面而简单的介绍，参考手册却是软件的增强手段即库函数包中子程序的用法说明，如参数的个数、类型，函数的返回值等。软件的用法也可以说是语法，它们解释得虽然全面，但略显简单和单调。

对于一个真正的软件开发者来说，掌握一个应用软件(如 AutoCad)或一种语言软件(如 MicroSoft C)只是一种手段，其真正的目的是利用这些软件，设计数据结构去实现一种算法，克服现实生话中的问题。因此追踪软件的版本是徒劳的。最关键的是通晓相对稳定的数据结构和实现算法模型。

《Quick C MicroSoft C 高级编程指南》(上、下册)就是介绍使用 Quick C 和 MicroSoft C 的数据结构和编程技巧的书。

本书的前几章讨论非常有用的数据结构：栈、队列，链接表和树。其后介绍开发通用的应用程序的方法，如语法分析，排序和数据压缩、加密，与 DOS 的接口，利用随机数生成器进行模型仿真，以及图中关键路径，迷宫的求解等。最后在附录中给出了一些 MMicroSoft C、Quick C 语言和 Quick C 集成环境的参考资料。

并不是说不需要用户手册、参考手册，相反，它们是基础。因此在阅读本书前，希望对 MicroSoft C 和 Quick C 有一些了解。另外，为了让读者更好地使用 Quick C，我们将很快继续推出《使用 Quick C》。

译者

1991 年 1 月

第一章　栈与队列

在本书中你会看到许多有趣的实用程序，这些程序有很大的部分都依赖于很少量的策略与数据结构。本章你将学习两种广泛使用的数据结构。

这些程序大都用于处理由单词、数或者由单词和数的组合所构成的表格和序列。插入和删除是表处理的两个最简单最基本的操作。在本章所考察的那些表中，这两种操作总是在表的某一端即表头或表尾上进行。

可以用两种基本的策略来对这种表或序列进行处理，它们的差别在于元素被加入或被除去的顺序不同。一种策略是总是除去最近加入表中的元素，即最后加入表中的元素最先被删去，这种表叫做 LIFO 表（后进先出表），举例说明 LIFO 表：

- 考古学中的层。即代表特定时期的沉积物。沉积物逐层垒迭，于是最近的沉积物在最顶层。这些层是以后进先出的顺序被挖掘出来的，因为最近的沉积物将会首先被挖掘出来。
- 函数调用。程序在执行一个给定的函数（如 main()）时，调用另一个函数（如 first ()），则在执行 first()时，暂停执行 main()；如果在执行 first( )时，再调用函数 second ()，则 first()又被暂时停止执行，同时将执行 second( )。一旦函数 second()执行完毕，控制就将返回至最近被暂停的函数 first()。假设在下表中的每个函数在执行时都要调用下一个函数，那么这此函数将会以从函数 main()至函数 fifth(的顺序被依次暂停执行。但是，这些函数将会以相反的次序被重新激活。即在执行完 sixth( )后，fifth( )将会被重新激活；在执行完 fifth ( )后，fourth ( )将会被重新激活，并且依次类推。

```
main( )
first( )
second( )
third( )
fourth( )
fifth( )
sixth( )
```

- 自助食堂中堆积起来的盘子和碟子。人们总是取走最顶上的盘子和碟子，并且人们总是将盘子或碟子加到最顶上。

另一种策略是总是除去表中“最老”的元素，即删去最先加入表中元素（也就是表中的第一个元素）。这种策略通常称为“先进先出”。即 FIFO。例如：

- 在银行或自助食堂里的排队。

程序调度者，它将按照收到的控制请求的次序来将控制转交给任务。

- 按照严格的顺序来处理的等待表。

§1.1 栈

栈是一种 LIFO 表。可分两部分来描述栈：存贮栈元素的空间和存取栈顶元素的方法。

§1.1.1 栈的描述

可以用如下的数据结构来描述一个存贮有 double 型值的栈。

```
#define MAX_VALS  50       /* maximum number of elements in stack */
struct db1_stack {
     /* vals is a MAX_VALS element array of double */
        double        vals [ MAX_VALS]:
        int           top;
};
```

这一结构包含了栈的两个部分：top 成员总是指明了在栈的数组（即 vals[ ]成员）中的下一个可供使用的单元。

类似地，可以使用下面的数据结构来表示一个字符串的栈。

```
#define MAX_VALS 50    /* maximum number of elements in stack */
#define SHORT_STR 15   /* maximum length for a stack entry */
struct stack {
    /* vals is a MAX_VALS element array of strings */
       char     vals [ MAX_VALS] [ SHORT_STR];
       int      top;     /* indicates the next stack cell to be filled */
};
```

与 double 型值的栈一样，这个结构也包含了栈的两个部分：（字符串）元素的 vals[ ]数组的 top，top 指明了在 vals[ ]成员中的下一个可供使用的单元。此时，下一个可供使用的元素本身也是一个数组。因此 top 在实际上是说明了这个字符数组的第一个单元。例如，当 top=10 时，下一个被加入到数组中的字符将会被存贮在单元 Vals[10][0]中。

§1.1.2 栈的操作

在本章的后几节里，你将编制一些处理字符串的栈的函数。在设计计算器时，将要用到这种表示法所提供的一种灵活性。为此，我们将利用在本章中的第二个栈例子 stack 结构，而不是 dbl_stack 结构来进行工作。你还可以同样容易地编写出处理其它类型的栈的函数。在读者利用这些例子来进行工作的同时，应该想一下你自己将会怎样来完成这些工作。在后续章节里，你将会使用其它的方式来定义栈元素，并将会对栈函数进行相应的修改。

需要在栈上执行的唯一的破坏性的操作是插入和删除。读者将会发现对一个栈是否为空可是否已满进行检查，以及对诸如栈的大小和栈顶槽中的内容等其它的方面进行考察都是很有用的。此外，一个用于初始化一个栈的通用过程也是很有用的。

§1.1.3 初始化和考察一个栈

下述宏和函数实现了栈的初始化、测试栈是否为空栈或满栈，等等。你应输入这些程序，并根据实际情况进行修改。有关 Quick C 的编辑命令参见本书附录 C（或查阅《Quick

C 的使用》第二章）。

```
/* Macros and functions for manipulating stacks.
   File assumes defs.h has been read.
*/
/* For all three macros, A should be a pointer to stack */
#define STACK_EMPTY(A)  (!((A)→top))                  /* is stack empty? */
#define STACK_FULL(A)   ((A)→top == MAX_VALS)  /* is stack full? */
#define STACK_SIZE(A)   ((A)→top)                     /* nr elements in stack */
struct stack {
    /* vals is a MAX_VALS element array of strings */
      char vals [ MAX_VALS] [ SHORT_STR];
      int top;      /* indicates the next stack cell to be filled */
};
/* ***********
   stack function DECLARATIONS
**********    */
void          disp_stack ( struct stack *);
void          init_stack ( struct stack *);
void          show_stack_top ( struct stack *);
int           stack_height ( struct stack *);
int           stack_is_empty (struct stack *);
int           stack_is_full ( struct stack *);
/* ************
   stack function DEFINITIONS
************ */
/*
   void init_stack ( struct stack *stack_ptr)
   ***********************************************
   Initialize each cell in the stack to NULL_STR;
   set top of stack (next element to add) to 0.
   CALLS : strcpy ();
   GLOBALS : MAX_VALS
   PARAMETERS :
         struct stack *stack_ptr : pointer to stack being initialized.
   RETURN : <none>
   USAGE : init_stack ( &st);
   ***********************************************
*/
void init_stack ( struct stack *stack_ptr)
{
```

```
        int index;
        for ( index = 0; index < MAX_VALS; index++)
                strcpy (stack_ptr → vals [ index], null_str);
        stack_ptr → top = 0;
}
/*
      int stack_is_empty ( struct stack *stack ptr)
      ************************************************
      Test whether stack_ptr is empty.
      If the_stack.top == 0, then stack is empty,
      so negating value returns the correct answer.
      CALLS :
      GLOBALS :
      PARAMETERS :
            struct stack *stack_ptr : pointer to stack being checked.
      RETURN : TRUE if stack is empty; FALSE otherwise.
      USAGE : result = stack_is_empty ( &st);
      ******************************************
*/
int stack_is_empty ( struct stack *stack_ptr)
{
          return ( !(stack_ptr → top)); /* if zero, return true */
}
/*
   int stack_is_full ( struct stack *stack_ptr)
   ****************************************************
   Test whether stack_ptr is full.
   If stack_ptr → top == MAX_VALS, then stack is full.
   CALLS :
   GLOBALS : MAX_VALS
   PARAMETERS :
         struct stack *stack_ptr : pointer to stack being checked.
   RETURN : TRUE if stack is full; FALSE otherwise.
   USAGE : result = stack_is_full ( &st);
   ************************************************
*/
int stack_is_full ( struct stack *stack_ptr)
{
        return ( stack_ptr → top == MAX_VALS); /* if equal, return true */
}
```

```
/*
   void show_stack_top ( struct stack *stack_ptr)
   ************************************************************
   Display the top stack element, but DO NOT pop the element.
   CALLS : printf (); STACK_EMPTY ()
   GLOBALS :
   PARAMETERS :
         struct stack *stack_ptr : pointer to stack being checked.
   RETURN : <none>
   USAGE : show_stack_top ( &st);
   ********************************************
*/
void show_stack_top ( struct stack *stack_ptr)
{
        if ( !STACK_EMPTY ( stack_ptr))
                printf ( "Top stack element: %s\n",
                        stack_ptr → vals [ stack_ptr_top - 1]);
}
/*
   int stack_height ( struct stack *stack_ptr)
   ************************************************
   Return the number of elements currently in the stack
   CALLS :
   GLOBALS :
   PARAMETERS :
         struct stack *stack_ptr : pointer to stack being checked
   RETURN : number of elements currently in stack.
   USAGE : size = stack_height ( &st);
   ******************************************
*/
int stack_height ( struct stack *stack_ptr)
{
         return ( stack_ptr → top);
}
/*
   void disp_stack ( struct stack *st)
   ***************************************
   Display the contents of the stack.
   CALLS : printf ();
   GLOBALS :
```

```
   PARAMETERS :
         struct stack *stack_ptr : pointer to stack being displayed.
   RETURN : <none>
   USAGE : disp_stack ( &st);
   ****************************************
*/
void disp_stack ( struct stack *st)
{
         int index;
         for ( index = st → top - 1; index >= 0; index--)
                  printf ( "stack.top == %d : value == %s\n",
                           index, st → vals [ index]);
}
```

文件defs.h中包含了以上这些函数以及其它函数所用到的常数定义。下面列出了在文件defs.h里栈函数所要用到的那部分内容。

```
/* Contents of file defs.h, as required for stack and other functions. */
#define INVALID VAL -99999.999
#define MAX_VALS    50    /* maximum stack size */
#define MAX_STR     100
#define SHORT_STR   15    /* length for a short string, used in stacks */
#define BLANK_CHAR' '
#define NULL_CHAR  '\0'
#define FALSE       0
#define TRUE        1
char      *null_str = "";
char      *blank_str = " ";
```

init_stack( )

函数init_stack( )把数组成员的每个单元都初始化为NULL_STR，并把（*stack_ptr）.top置为零。栈的top员的值代表了在栈的下一个单元的索引。因此，如果这个值为0，则说明该栈为空栈。

请回想一下，结构指针运算符（→）是在利用一个指向结构的指针来工作的时候，对结构中的一个成员的访问的速记方法。下面的两条语句是等效的，并且表明了所做的工作：

```
/* the following two assignment statements are equivalent */
stack_ptr → vals [ 12] = val;
/* dereference the stack pointer, to get at the stack itself
   --- (*stack_ptr) ---
   then access the structure's vals member --- (*stack_ptr).vals [ 12]
   The parentheses are necessary because of the relative precedence of
```

```
   . and *
*/
(*stack_ptr).vals [ 12] = val;
```

stack_is_empty( )与 STACK_EMPTY( )

函数 stack_is_empty( )检查栈的 top 成员的值，并且这个值是作为一个自变量来传递的。如果这个值等于 0，则栈为空。为了返回适当的真值，为零的结果被取反以产生一个非零（或真）的结果。请注意这里的参数实际上是一个指向栈的指针而不是一个栈；这是为了节省内存，因为当 stack_empty( )被调用时，系统将需要去制作栈的一个局部的拷贝。制作这样的拷贝经常会需要增加程序所用的栈空间；同时拷贝还需要占一定的时间。

STACK_EMPTY( )宏可以更快地完成同样的任务。这个宏将使用栈的 top 成员的否定去替换由宏的自变量所指向的那个值。在本章里的栈函数所使用的是 STACK_EMPTY( )而不是 stack_is_empty( )。

stack_is_full 与 STACK_FULL

函数 stack_is_full 检查栈的 top 成员是否等于预先定义的栈容量极限 MAX_VALS。如果等于条件成立，则说明栈已满。（因为下一个将要被填充的单元的索引超出了该栈数组所能允许的最大的索引）。同样，该函数也使用了一个指向栈的指针来节省内存的时间。

STACK_FULL( )宏以更快速度完成了同样的任务。这个宏将去替换一个测试 top 成员和 MAX_VALS 的相等性的表达式。这个宏取一个指向栈的指针来作为它的自变量。在本章的栈处理函数中所使用的是 STACK_FULL，而不是 stack_is_full( )。

show_stack_top( ), disp_stack( ),
stack_height( ), 和 STACK_SIZE( )

函数 show_stack_top( )显示当前位于栈顶的值。函数 disp_stack( )以每行显示一个元素的方式显示整栈。函数 stack_height( )返回了在栈中的元素的个数。最后，宏 STACK_SIZE( )与 stack_height( )的功能相同，只是宏的速度要快一点，因为在预处理的过程中，这个宏只是简单地替换了在一个表达式中栈的 top 成员的值，而不是在运行的时刻要求一次函数调用。

§1.1.4 栈的插入与删除

根据栈的定义，只能在栈顶维护栈。要想插入一个元素，就将它加入支栈的顶部——亦即在第一个可供使用的单元中。要想删去一个元素，则应除去在数组中最顶部的那个值，并且相应地调整栈顶指示器。栈的插入与删去这两种动作分别被称为栈的压入和弹出。下列函数能够压入和弹出所说明的栈。

```
/* Functions for pushing and popping a stack.
   Add to file stack.c.
*/
/* Add to function declaration section of stack.c */
char          *pop_stack ( struct stack *);
```

```
int             push_stack ( char *, struct stack *);
/* function definitions */
/*
    int push_stack ( char *val, struct stack *stack_ptr)
   ****************************************************
   Add an element to a stack, first checking whether the stack is full.
   If the stack is full, the function does nothing.
   CALLS : STACK_FULL (); strcpy ()
   GLOBALS : TRUE, FALSE
   PARAMETERS :
          char *val : string being added to stack;
          struct stack *stack_ptr : pointer to stack being augmented.
   RETURN : int specifying whether something was added to stack
   USAGE : did_add = push_stack ( str, &st);
   ****************************************************
*/
int push_stack ( char *val, struct stack *stack_ptr)
{
       /* if stack is not full, add the element,
          AND increment the top of stack marker.
       */
         if ( !STACK_FULL ( stack_ptr))
         {
               strcpy ( stack_ptr -> vals [ (stack_ptr -> top)++], val);
               return ( TRUE);
         }
         else    /* if stack is full, indicate that push was unsuccessful */
               return ( FALSE);
}
/*
   char *pop_stack ( struct stack *stack_ptr)
   ****************************************************
   Remove an element from a stack, if possible;
   return the element to the calling function;
   adjust the top of the stack to reflect the pop.
   CALLS : STACK_EMPTY ()
   GLOBALS :
   PARAMETERS :
          struct stack *stack_ptr : pointer to stack being popped.
   RETURN : string popped from stack.
```

```
   USAGE : strcpy ( str, pop_stack ( &st));
   ***********************************************
*/
char *pop_stack ( struct stack *stack_ptr)
{
        if ( !STACK_EMPTY ( stack_ptr))
                return ( stack_ptr → vals [ --stack_ptr → top]);
        else
                return ( null_str);
}
```

push_stack( )

函数 push_stack( )有两个自变量：字符串的值与栈指针。第二个参变量必须为指针以使得在 push_stack( )已完成了它的工作之后，新的元素还能保留在栈中。

请回忆一下，在 C 语言中参数是传值的；也就是说，只有保存在一个自变量中的值的一个拷贝被正常地传递了。这就意味着无法直接改变实际的变量，除非所传递的是该变量的地址。

函数 push_stack( )首先检测栈是否已满；如果栈已满，则该函数不改变栈的内容，而返回值 FALSE(0)以表明这次压栈没有成功。

在函数 push_stack( )中，结构指针运算符的二次使用使得中心的表达式非常紧凑。这个表达式指明数组的下标，它等效于说明了“取 stack_prt 所指的那个栈的 top 成员，使用 top 的当前值来完成这项任务，然后用来增加 top 成员”。在 stack_ptr→top 两端的园括号是为了提高清晰性而加入的;它们不是必须的，因为结构指针运算符的优先级较增量运算符的优先级高。因此这条赋值语句（指调用函数的那条语句——译者注）是说“把 Val 的值赋给在 stack_ptr 所指向的栈的栈顶单元；在执行完此赋值后，再调整栈顶指示器。

pop_stack( )

在函数 pop_stack( )里用类似的语法来特定的栈单元和调节栈顶指示器。但是，在说明将要弹出的单元的索引的时候，该函数所使用的是减量运算符（－－）的前缀版本（即前置－－———译者注）。top 成员总是指明下一个被填充的单元。它表明最后被压入到栈中的那个值应位于其索引等于当前的栈顶值减 1 的那个单元中。一旦这个元素被弹出了，top 的新值就会被保留下来以作为新的栈顶指示器，因为刚才被这次弹出所空出来的那个单元现在还是下一次将要被填入的单元。

§ 1.1.5 栈的练习

你可利用下面的程序来测试栈函数。其中 main( )是前述程序中唯一没有涉及到的函数。因此如果你已在磁盘上建立了其它函数，则你只需输入 main( )。在文件 defs.h 和文件 stack.c 中包括了数据结构，预处理器常数以及早些时候给出的函数定义。

```
/* Program to exercise stack functions.
   Program reads stack routines and definitions from file stack.c
```

```
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/* read files containing preprocessor directives and function definitions */
#include "defs.h"
#include "stack.c"
main ()
{
        struct stack st;
        int          index;
        char         *outcome, message [ SHORT_STR];
    /* show how much storage the stack requires. */
        printf ( "%d\n", sizeof (st));
        init_stack ( &st);          /* initialize the stack */
/*  get values for the stack */
   do
   {
          printf ( "? ");
          gets ( message);
          push_stack ( message, &st);
}
while ( strcmp( message, null_str));
/* display the values by looping through the stack */
   for ( index = st.top; index >= 0; index--)
   {
          outcome = pop_stack ( &st);
          if ( outcome != null_str)
                  printf ( "%d: %s\n", st.top, outcome);
          else
                  printf ( "nothing left in stack\n");
    }
/* get more values for the stack */
   do
   {
         printf ( "? ");
         gets ( message);
         push_stack ( message, &st);
   }
   while ( strcmp ( message, null_str));
```

```
/* display the values by calling disp_stack () */
   disp_stack ( &st);
}
```

你可以在 Quick C 环境中把这个程序编译到内存中，你也可以利用 Quick C 或 Microsoft C5.0 或其它的遵守 Draft Procesed ANSI Standard 的 C 编译器来产生一个可执行文件。假设你给这个程序命名为 xstack.C，则在 Quick C 中将此文件编译成可执行的形式的步骤如下：

1. 在 DOS 命令行键入 qc xstack。
2. 进入 Quick C 后，先按 ALT-R，再按 C，即可得到编译器选项菜单。
3. 在该菜单的对话框中，按 X 键来使 Quick C 编译出一个 exe 文件。

通过调试程序，可以了解栈的工作原理，并能测试出这些函数的局限。例如，可以去试着输入一个长于 SHORT_STR 个字符的串，在压入了一个这样的长串之后，如果再将一个串压入到栈中，将会出现什么现象？

在 stack.c 中所定义的 MAX.VALS 的值 50 是任意定义的。但是，如果你要把它定义得大一些，那么就将会需要在 Quick C 运行时刻的选项中增大栈的容量。为此你需按下 ALT-R-O（即先同时按下 ALT 和 R 键，然后再按下字母 O 键）。在所产生的对话框中，按下 ALT-S 来进入到 Stack Size 正文框中。把它设置成一个较大的值（如 4096 或 8192），并回车。然后按正常的步骤来编译和运行程序。

§1.2 栈的应用：一个简单的后缀式计算器

在本节你可编制一个程序来作为简单的四功能计算器。这个程序在计算的过程中将使用栈来保存值。该计算器将使用一种后缀式或逆波兰表达式的记号法。

§1.2.1 中缀记号法

你已习惯于以如下的方式来书写数学表达式：

7.5＋39.3－16.2

让我们简要地看一下这个表达式的结构。象＋和－这样的元素被称为操作符，它们确定了将要对操作符两侧的数进行何种操作。而把 7.5 和 16.2 这两个元素称为操作数，它们代表了被操作符用来产生结果或用作其它的操作符的新的操作数的那些值。因此上述表达式的结构为：

7.5	＋	39.3	－	16.2
<操作数>	<操作符>	<操作数>	<操作符>	<操作数>

操作符＋和－都要求有两个操作数，并返回单值。加法运算符把运算数 7.5 与 39.3 相加，返回和 46.8；类似地减法运算符把所产生的结果 46.8 与 16.2 相减返回差 30.6。

要求带有两个操作数并返回单一的结果的操作符称为二元操作符。在前面的记号法中，每个二元操作符都被放在它的两个操作数之间，这种记号法称为中缀记号法。它是在

学校中所教的方法。

在数学表达式中，如没有乘法和除法，则加减法是依从左至右的顺序进行的。在代数中先计算乘除法，后计算加减法。因此下列表达式的值为 17.5，而不是 22.5。因为乘法优先于加法。

2.5＋5＊3 要想使此式的结果等于 22.5，必须用括号来改变运算顺序，即：

(2.5＋5)＊3 这个例子指出了中缀记号法的一个缺点：需要用括号来控制计算的次序。

§1.2.2 后缀记号法

在本世纪五十年代，波兰逻辑学家 Janlukasiewicz 发明了两种用于书写数学表达式的不用括号的记号系统。在这两种系统里，操作符或者是放它的操作数之前，（前缀，或波兰记号法），或者是放在它的操作数之后（后缀，或逆波兰记号法）。这两种系统的优点在于无须使用括号来控制运算的次序。

在本章中所构造的这个计算器将要使用后缀记号法。但是，首先让我们简要地看一看后缀记号法是如何工作的。下面分别给出二个表达式的中缀的和后缀的版本

```
3  +  7  /* infix notation: <left operand>  <operator>  <right operand>  */
3  7  +  /* postfix notation: <left operand>  <right operand>  <operator>  */
2.5  +  5  *  3   /* infix notation */
2.5  5  3  *  +   /* postfix notation */
```

第一对表达式表明了中缀式和后缀式的主要区别。在第一个表达式中操作符位于左右操作数之间。而在第二个表达式中左右操作数都在操作符之前。

让我们仔细地看一下第二对表达式。这两个表达式说明了如何确定运算次序。对于中缀式你可以确定其结果为 17.5，因为此表达式将会演变成 2.5+15；即由乘法产生的结果 15 是加操作符的右操作数。在中缀式中执行过程为：

```
  5   *   3      /* == 15 */
2.5   +  15      /* == 17.5 */
```

你也可以把后缀式分写成如下形式，来产生相同的运算结果。

```
5      3     *      /* == 15 */
2.5    15    +      /* == 17.5 */
```

通常你是从左至右地计算表达式的。可以使用这种方法来看看后缀式表示法是如何进行计算的。计算步骤如下：

1. 如果表达式中只有一个操作数，那么求值已经完成；否则执行步骤 2。
2. 找出最左边的操作符。上例中为*。
3. 找出紧挨在这个操作符前面的两个操作数。在上例中为 3 和 5。
4. 把紧靠所选操作符的操作数看作该算子的右操作数，而另一个则为左操作数。并实现这次运算。在上列中为 5*3
5. 用第四步中的结果取代上次运算的三个项。在上例中用 15 代替 5，3 以及*。
6. 重复步骤 1 至 5，直至只剩下一个单个的值为止。在上例中把 2.5 和 15 相加后就得出最终结果。

为了看清在后缀式中无需使用括号的原因，让我们来考察一下这个表达式的另一个版本。

下面是一个中缀和后缀的版本，在这个表达式中加法先于乘法完成：

```
(2.5    +   5)    *   3          /* infix notation */
2.5     5   +     3   *          /* postfix notation */
```

利用该算法，经下列步骤可解决上述表达式：

1. 找到＋作为最左边的操作符。
2. 计算 2.5+5。
3. 用上述计算结果代替 2.5，5 和+，剩下 7.5 3 和*。
4. 确定*为第二次运算的最左操作符。
5. 计算 7.5*3。
6. 用计算结果取代 7.5 3 和*，剩下 22.5。
7. 因只有一个操作数而无操作符，故终止计算。

在后缀式中，操作符总是以它们将会被施用的顺序来出现的。即使操作数很复杂，包含了其它的表达式，也还是要放在操作符的前面。

§1.2.3 建立后缀表达式

让我们看一个例子，随后你将编写一个自动形成后缀表达式的函数。现在，要想去生成一个后缀表达式，只需试着去确定每个操作符的操作数是什么。可简单地把操作数放在操作符之前来形成一个后缀式。有时必须通过一定的计算才能获得一个操作数。例如上例中必须先计算 5*3，才能得到操作符＋的右操作数。

试着建立下式的后缀表达式：

7.5＋35＊(7-3.5)＋6/3

显然第一个＋操作符的左操作数为 7.5，而其右操作数尚不清楚，只知其为乘法的运算结果。因此为了确定第一个＋操作符的右操作数，需先计算包含一个乘法算子的表达式。因此在解决乘法之前，先暂不考虑这个加法问题。

乘操作符的左操作数也很清楚，它是 3.5。而其右操作数为括号内的减法的运算结果。因此要想得到乘法的右操作数，首先要计算这个减法。减法的左操作数为 7，右操作数为 3.5，故减法的后缀表达式为：

```
7   3.5   -      /* postfix form of term involving subtraction operator */

L   R            /* L == left operand; R == right operand */
```

这个式子就变成了乘法的右操作数，因此乘法表达式变为：

```
35   7   3.5   -   *

L        R
```

接下来上述表达式又变成第一个加法的右操作数，于是我们有：