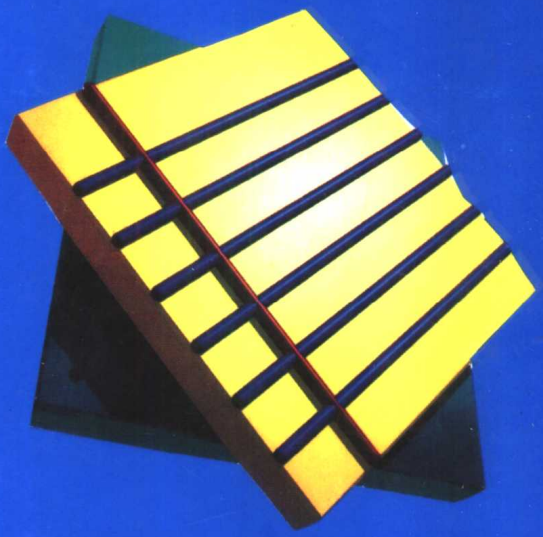
计算机知识普及系列丛书



C

# Quick C、Microsoft C 高级编程指南

李振格
王　江
罗　克　编写
汤建平

学苑出版社

计算机知识普及系列丛书

# *Quick C*、*Microsoft C* 高级编程指南

李振格　王　江
罗　克　汤建平　编写
管叶茂　审校

学苑出版社
1993.

(京)新登字 151 号

## 内 容 提 要

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 Microsoft C、Quick C 语言和 Quick C 集成环境的参考资料。

欲购本书的用户，请直接与北京 8721 信箱联系，电话 2562329，邮码 100080。

计算机知识普及系列丛书

Quick C、Microsoft C 高级编程指南

编　　写：李振格　王　江
　　罗　克　汤建平
审　　校：管叶茂
责任编辑：甄国宪
出版发行：学苑出版社　　邮政编码：100032
社　　址：北京市西城区成方街 33 号
印　　刷：施园印刷厂
开　　本：787×1092　1/16
印　　张：38　　字　数：901 千字
印　　数：1～3000 册
版　　次：1993 年 12 月北京第 1 版第 1 次
ISBN7－5077－0821－7/TP・19
本册定价：30.00 元

学苑版图书印、装错误可随时退换

# 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，克服现实生活中的问题。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

# 目　录

# 目　录

# 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 dbl_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 */
}
```