ISBN 0 [illegible]

THE COMPUTER JOURNAL®

Programming - Applications - User Support

Issue Number 25 $3.00U.S.

Repairing and Modifying Printed Circuits page 4

Z-Notes

Z-Com Versus the 'Hacker' Version of Z-System page 10

The C Column

Exploring Single Linked Lists page 14

Adding a Serial Port to the AMPRO Little Board page 19

The SCSI Interface

Building a SCSI Adapter page 23

NEW-DOS

Part 4: The CCP Internal Commands page 30

AMPRO 186 Column

Networking With Super DUO page 35

ZSIG page 43

The Computer Corner page 48

The C Column

Exploring Singly Linked Lists

By Donald Howes

Exploring Singly Linked Lists

This time around, I'll be looking at the use of linked lists. This type of data structure has many applications, such as stacks, queues and dynamic tables. The inherently dynamic structure of a linked list lends itself well to use in data base applications, or any code where the amount of data is not known in advance, or can vary within large limits. By dynamically allocating and releasing memory in a linked list, the programmer can avoid the memory overhead of allocating an array large enough to handle the maximum amount of data thought likely to be manipulated by the program.

Structure Allocation

As an example of a singly linked list, I'll be developing the code for a stack. The dynamic capabilities of this application are achieved through the use of pointers to structures as the elements of each list node. The stack nodes consist of a pair of pointers to an associated data structure and to the following node. All the "useful" information, that is, the actual data being placed on the stack, is contained in the separately defined data structure. This allows the stack code to be used in all circumstances, without regard for the type of data being pushed on the stack.

To create any linked list, three things are needed (Figure 1). First, the list needs a header, which will minimally tell the length of the list and the location of the first node (for other than stack applications, it is a good idea to include a pointer to the last item on the list as well). In addition, each node of the list must be declared, with a pointer to the data structure for that node and a pointer to the location of the next node on the list. Last, the tail of the list must be marked. This can be accomplished most easily by the use of a standard list node, where the pointer to the following node is declared as a NULL pointer.

Listing 1 gives the structures which are used to declare the stack header and the stack nodes. Both of them are very simple. The only thing to note is the declaration of the data pointer in the stack node. As stated above, this data structure can be anything (simple or complex) which is required to get the job done. The structure itself is declared in the program using the stack code.

A Look At Stacks

The usual way to give a visualization of how a stack operates is to draw the analogy with the type of plate dispenser usually seen in cafeterias. These little devices are spring loaded, so that, as the plates are loaded, the level of the stack of plates drops. This makes only the top plate available for use.

The analogy is a good one, since this is exactly the way a software stack operates. Only the uppermost value which has been placed on the stack is accessible, and data can only be added, or removed, from the top of the stack. A stack is, therefore, what is technically known as a LIFO (Last In, First Out) data structure. Of course, there is also a FIFO (First In, First Out) data structure, which is commonly known as a queue (in analogy to lines at bank windows or cashiers, although these queues generally move a lot faster).

Listing 1. Header File for Stack Code

```
/*****************************************************
*                    stack.h                         *
*            Header file for stack code.             *
*****************************************************/

#define   MALLOC(x)  ((x *)malloc(sizeof(x)))
#define   NULL    0

/****** typedefs for stack data structures *****/

typedef struct node{      /* structure for stack node */
  struct info *data;
  struct node *next_node;
}stk_node;

typedef struct head{      /* structure for stack header */
  int length;
  struct node *top;
}stk_head;
```

To manage a stack, the programmer has to be able to accomplish four things. First, allocate space for the stack header to start the stack. Second, allocate space for a stack node and push the node and associated data onto the stack. Third, pop data from the stack when needed and deallocate the space for the popped node. Fourth, deallocate the space for the stack header (and any left over stack nodes) after all access to the stack has been completed. This will free up all space used by the stack for other purposes.

The Code

The code for the stack functions are found in Listing 2. In the listing, the code actually relating to stack management is restricted to the last five functions. These are; crt_stk(), crt_node(), push(), pop() and del_stk(). The rest of the code in the listing is a short demonstration program to illustrate the use of the stack functions.

Starting at the top, there is the structure definition for the data structure used in the application. In this case, the structure contains only a single integer value, but could be made as complicated as needed, with a variety of variables and variable types. Following the structure is the list of error messages which will be used by the p_error() function. Following those are declarations for four of the stack functions. These functions

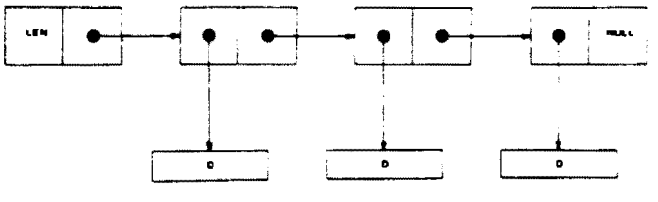

Figure 1

must be declared externally to the main() function for two reasons. First, if functions are not explicitly declared as to their return value (if that value is other than an int), then the compiler will assume that the return value for the function is an interger and make the appropriate changes to the actual return value to conform with those conditions. In other words, if you are using a function (say sqrt(), which returns a double) and do not declare that function to return a double, then an int will be returned. This can lead to some very interesting and hard to track down bugs, and is one of the common programming mistakes made by most, if not all, C programmers at one time or another. The second reason that these functions are declared outside of main() is related to scoping. The declaration outside of main() allows the functions to be scoped globally to all functions within the same file. If the functions had been declared inside of main(), then their use in functions other than main() [such as the call to crt_node() from inside of push()] would result in the same problem just described. The actual return value of the function wouldn't be known and a default value of int would be returned.

Moving on to main() itself, space for the stack header is created using crt_stk(). This function uses the macro MALLOC() which was defined in the header file (Listing 1). This provides a nice shorthand way to invoke the function malloc(), which is used to allocate memory (see your compiler documentation for a longer explanation). If the space for the header is not available, the function will return a NULL pointer and the program will terminate with an error message. If everything goes according to plan, the function will return a pointer to the location of the stack header, and we're in business. Although I didn't do it here, there is nothing to prevent the programmer from creating multiple stacks with this code, each stack being handled with a separate header variable. That is the nice thing about this type of stack building using pointer referencing, it allows the programmer to make maximum use of the memory available on the computer, without having to make decisions about memory allocation prior to run time.

Once the header has been allocated, the guts of the program run inside a single switch() function located inside of a while statement (note that braces are not necessary here, since only a single logical statement follows the while, even though that statement is relatively complex). The while calls the function menu() which displays a menu of valid operations. Since the while statement continues to loop until an 'S' is entered, the code is self checking in the case of invalid input. In the switch(), each case statement only looks for uppercase letters, since menu() translates input to uppercase before passing the value back to main().

Working through the switch() will take us through the rest of the program. Entering a 'P' from the menu will call the function get_data(), which allocates space for a data structure. If this space is available, the function input() is called, with an appropriate input request string and the MIN and MAX values for the input range. Input() does appropriate range checking, not returning a value until one within the declared range is entered. If no space was available, a NULL pointer is returned immediately to the function call in the switch() and the program is terminated with an error message. If everything works, then push() is called, being passed the new data structure and the stack header. One important thing to note here is that I am passing the actual structures to the function. This is in accordance with one of the changes being proposed by the ANSI C committee, but older versions of a compiler will not support this

feature. If you try to compile this code and it doesn't work properly, you will probably have to modify the function calls to pass pointer to the structures, rather than the actual structure. An example of this would be changing the call to push() from push(new,stk) to push(&new,&stk), everything should work properly then.

Once push() is called, it immediately calls crt_node(), which allocates space for the stack node and assigns the pointers for the location of the data structure and the next node in the stack. If a NULL pointer is not returned, the pointer for the top of the stack is set to that of the newly created node and the counter for the length of the stack is incremented. If a NULL pointer is returned, the value FALSE is returned to get_data(), which returns a zero (0) to the switch and causes the program to terminate with an error message (yeah, I know, it seemed a lot easier when I wrote the code). In (hopefully) simpler terms, what we have just done can be visualized using those little pop-together plastic beads. In inserting the new node at the head of the stack, we detached the node following the header (which is pointed to by the value of "top" in the header) and assigned that pointer to "next_node" in the new node structure. Once that was done, we assigned the pointer returned for the new node to "top", and everything was done.

Popping data from the stack is even easier, since there is less fiddling and error checking required (everything is already present, all we're trying to do is get at it). Entering "O" from the menu calls disp_data(). This function determines if anything is on the stack, then calls pop() and displays an appropriate message to explain the output and waits for the user to

Listing 2. Source Code for Stack Application.

```
/*
  stack.c

  Copyright 1986 Donald Howes
    All rights reserved.

  This program may be copied for personal, non-commercial use
  only, provided that the copyright notice is included in all
  copies.

  Possible compiler dependent functions are:
  1. cursor()
  2. clrscr()
*/

#include  <stack.h>
#include  <ctype.h>
#include  <stdio.h>
#include  <malloc.h>

#define MIN -100
#define MAX  100

/***** structure for data to be placed on stack *****/

typedef struct info{
  int data;
}stk_info;

static int *err_mess[3] = {
    'Can't create the stack.',
    'Could not push data onto the stack.',
    'Stack is empty, can't display anything.'
};

stk_head *crt_stk(), del_stk();
stk_node *crt_node();
stk_info *pop();

main()
{
  int c, test;
  stk_head *stack;

  stack = crt_stk();
  if (stack == = NULL)
  {
    p_error(0);
    exit();
  }

  while ((c = menu()) ! = 'S')
     switch(c){
      case 'P':
         if (!(test = get_data(stack)))
         {
           p_error(1);
           exit();
         }
         break;
      case 'O':
         disp_data(stack);
         break;
      case 'Q':
         query_stk(stack);
         break;
      default:
         putchar(' \ 007');
     }
  clrscr();
  del_stk();
} /* end of main */
```

```
int p__error(erno)
int erno;
{
  printf(err__mess[erno]);
} /* end of p__error */

int menu()
{
  int c;

  clrscr();
  cursor(10,20);
  printf('[P]ush data on the stack \ n');
  cursor(11,20);
  printf('p[O]p data from the stack \ n');
  cursor(12,20);
  printf('[Q]uery stack items \ n');
  cursor(13,20);
  printf('[S]top \ n');
  cursor(15,20);
  printf('select function letter : ');
  c = toupper(getch());

  return(c);
} /* end of menu */

void query__stk(stk)
stk__head *stk;
{
  clrscr();
  cursor(10,10);
  printf('%d items are on the stack. \ n',stk->length);
  cursor(12,10);
  printf('press any key to continue.');
  getch();
} /* end of query__stk */

int get__data(stk)
stk__head *stk;
{
  stk__info *new;

  clrscr();

  if (new = MALLOC(stk__info))
  {
    cursor(10,10);
    new->data = input('enter data value : ',MIN,MAX);
  }
  else
    return(new);

  return(push(new,stk));
} /* end of get__data */

int input(prompt,low,high)
char *prompt;
int  low;
int  high;
{
  int ival;
  printf('%s',prompt);

  while(1)
  {
    scanf('%d',&ival);
    getch();

    if((ival > = low) && (ival < = high))
      break;
    cursor(10,29);
  }

  return(ival);
} /* end of input */

void disp__data(stk)
stk__head *stk;
{
  stk__info *new;

  clrscr();

  if (stk->length)
  {
    cursor(10,10);
    new = pop(stk);
    printf('the value is %d',new->data);
    free((char *)new);
    cursor(12,10);
    printf('press any key to continue.');
    getch();
  }
  else
  {
    cursor(10,10);
    p__error(2);
    cursor(12,10);
    printf('press any key to continue.');
    getch();
  }
} /* end of disp__data */

stk__head *crt__stk() /* create a stack */
{
  stk__head  *new;

  if (new = MALLOC(stk__head))
  {
    new->length = 0;
    new->top = NULL;
  }
  return(new);
} /* end of crt__stk */

stk__node *crt__node(val,ptr) /* create a node on the stack */
  stk__info *val;
  stk__node *ptr;
{
  stk__node *new;
```

```
  if(new = MALLOC(stk_node))
  {
    new->data = val;
    new->next_node = ptr;
  }
  return(new);
} /* end of crt_node */

int push(data,stk) /* push a node onto the stack */
  stk_info *data;
  stk_head *stk;
{
  stk_node *new;

  if(new = crt_node(data,stk->top))
  {
    stk->top = new;
    stk->length++;
    return(TRUE);
  }
  else
    return(FALSE);
}  /* end of push */

stk_info *pop(stk)  /* pop data off the stack */
  stk_head *stk;
{
  stk_node *temp;
  stk_info *data;

  temp = stk->top;
  data = temp->data;
  stk->top = temp->next_node;
  stk->length--;
  free((char *)temp);
  return(data);
} /* end of pop */

stk_head del_stk(stk) /* delete the stack header */
  stk_head *stk;
{
  stk_info *data;

  while(stk->length > 0)
  {
    data = pop(stk);
    free((char *)data);
  }
  free((char *)stk);
} /* end of del_stk */
```

press a key before returning to the menu. Pop() requires that a temporary node be created before the top of the stack is popped. What is done is the reverse of pushing a node onto the stack. The values of the first node of the stack are assigned to the temporary node and the value of "next_node" of the temporary structure is assigned to "top" in the header. A pointer to the data structure associated with temp is returned to the calling function and the space allocated to the node is freed.

The third menu option is to query the size of the stack. A call to query_stk() displays a message giving the length of the stack, which is determined from the value of "length" in the stack header. Finally, entering 'S' from the menu will stop the program, calling del_stk(), which deallocates the storage for any nodes remaining on the stack [by calling pop()] and that of the stack header. Under normal circumstances, there shouldn't be anything remaining on the stack when del_stk() is called, and you may wish to add some error checking code which will flag this condition if it occurs.

Erratum

I'd like to correct a mistake I made in my last column. In talking about how the program "strip" worked, I said that the bit pattern for 0x7F was "10000000". Anyone who added up the value of that pattern found it was equivalent to 0x80. The real bit pattern for the 0x7F mask is "01111111" and the masking process is a bitwise AND, not an add. Don't worry, the code works alright, I just didn't explain it very well. I guess it's a mistake to write these columns late at night. ■

* These listings are available on disk, or on the TCJ BBS (406) 752-1038.