```
/************************************************************
 *
 *                      stack.h
 *
 *  Include file for the simple stack module.  This example
 *  illustrates how modules are implemented.
 *
 ************************************************************/

#ifndef STACK_H
#define STACK_H

/*
 *  The Stack type is used as a handle to a stack object.
 *  User programs don't know the stack data structure, but
 *  only a pointer is passed, so they don't need to know.
 */

typedef struct stack_struct* Stack;

/*
 *  Declarations of the functions in the stack module.
 */

Stack stack_create( int size );
Stack stack_destroy( Stack );
void stack_push( Stack, int item );
int stack_pop( Stack );
int stack_peek( Stack );
int stack_full( Stack );
int stack_empty( Stack );

#endif
/*****************************************************************/
```



```
/************************************************************
 *
 *                      stack.c
 *
 *  Implementation of the stack module.  This file contains
 *  the C code for the procedures that manipulate the stack
 *  data structure.
 *
 ************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

/*
 *  Declare the data structure that is used for the stack
 *  module.  This data structure isn't visible outside of
 *  this file.  A simple array is used to store the data
 *  in the stack.  The stack structure also contains the
 *  size of the stack, and the index of the current stack
 *  top.
 */

struct stack_struct {
     int       size;     /* The size of the stack */
     int       top;      /* Index of the top of the stack */
     int*      dataptr;  /* The actual stack data */
};


/*
 *  The stack_create procedure creates a new instance of
 *  the stack data structure.  The parameter to this procedure
 *  is the maximum number of elements in the stack.  The
 *  value returned is a Stack value that can be passed to
 *  the other functions in this module.
 */
```



```
Stack stack_create( int size ) {
     Stack result;

     result = (Stack) malloc( sizeof *result );
     if ( result == NULL ) {
          printf( "out of memory in stack module\n" );
          abort(  );
     }

     result->size = size;
     result->top = 0;
     result->dataptr = ( int * ) calloc( size, sizeof(int) );
     if ( result->dataptr == NULL ) {
          printf( "out of memory in stack module\n" );
          abort(  );
     }

     return( result );

}


/*
 *  The stack_destroy function returns the resources that have
 *  been allocated to a stack instance.  it is careful to set
 *  all freed pointers to NULL so they can't be dereferenced
 */

Stack stack_destroy( Stack old ) {

     free( old->dataptr );
     old->dataptr = NULL;
     free( old );

     return( NULL );

}


/*
 *  The stack_push procedure add an item to the top of the stack.
 *  If the stack is full, this procedure returns without adding
 *  the item, keeps the stack in a valid state.
 */
```



```
void stack_push( Stack s, int item ) {

     if ( s->top == s->size )
          return;

     s->dataptr[s->top] = item;
     s->top++;

}


/*
 *  The stack_pop procedure removes the top item from the
 *  stack and returns it.  If the stack is already empty
 *  it returns the value 0 without making any changes to
 *  the stack.  This preserves the valifity of the stack.
 */

int stack_pop( Stack s ) {

     if ( s->top == 0 )
          return( 0 );

     s->top--;
     return( s->dataptr[s->top] );

}


/*
 *  The stack_peek function returns the top item of the stack
 *  without removing it from the stack.  If the stack is already
 *  empty this procedure returns 0.
 */

int stack_peek( Stack s ) {

     if ( s->top == 0 )
          return( 0 );

     return( s->dataptr[s->top-1] );

}
```

```
/*
 *  The stack_full function returns true if the stack is full,
 *  otherwise it returns false.
 */

int stack_full( Stack s ) {

     if ( s->top == s->size )
          return( 1 );
     else
          return( 0 );

}


/*
 *  The stack_empty procedure returns true if the stack is
 *  empty, otherwise it returns false.
 */

int stack_empty( Stack s ) {

     if ( s->top == 0 )
          return( 1 );
     else
          return( 0 );

}
/************************************************************/
```



```
/************************************************************
 *
 *                    test.c
 *
 *  Simple test program for the stack module.  This program
 *  test the basic functionality of the stack module, and
 *  detects the most serious bugs.
 *
 ************************************************************/

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

int main( void ) {
     Stack s1;
     int i;

     /*
      *  first create a new stack, and then see if stack_empty
      *  and stack_full produce the correct values for an
      *  empty stack
      */

     s1 = stack_create( 10 );

     if ( stack_empty( s1 ) )
          printf( "stack_empty correct on new stack\n" );
     else
          printf( "new stack should be empty, but isn't\n" );

     if ( stack_full( s1 ) )
          printf( "a new stack shouldn't be full\n" );
     else
          printf( "new stack isn't full\n" );

     /*
      *  Now fill the stack with integers, on exit from the
      *  for loop the stack should be full.  This gives us
      *  another opportunity to test stack_empty and stack_full
      */

     for ( i = 0; i < 10; i++ )
          stack_push( s1, i );
```



```
     if ( stack_full( s1 ) )
          printf( "stack_full is true for a full stack\n" );
     else
          printf( "stack_full doesn't return true for a full stack\n");

     if ( stack_empty( s1 ) )
          printf( "a full stack shouldn't be empty\n" );
     else
          printf( "stack_empty is false for a full stack\n" );

     /*
      *  Now check stack_peek to see if it returns the
      *  top item on the stack
      */

     if ( stack_peek( s1 ) == 9 )
          printf( "stack_peek returns the correct value\n" );
     else
          printf( "stack_peek returns an incorrect value\n" );

     /*
      *  Pop the values of the stack, see if the correct
      *  sequence of values is produced
      */

     printf( "This should print: 9 8 7 6 5 4 3 2 1 0\n" );
     for ( i = 0; i < 10; i++ )
          printf( "%2d", stack_pop( s1 ) );
     printf( "\n" );

     /*
      *  At this point the stack should again be empty, test
      *  stack_empty and stack_full to see if they produce
      *  the correct values
      */

     if ( stack_empty( s1 ) )
          printf( "stack_empty correct for empty stack\n" );
     else
          printf( "stack should be empty now\n" );
```



```
/*
      *  Finally destroy the stack, and see if stack_destroy
      *  returns the correct value
      */

     s1 = stack_destroy( s1 );

     if ( s1 == NULL )
          printf( "stack_destroy returns the correct value\n" );
     else
          printf( "stack_destroy doesn't return NULL\n" );
    return 1;
}

/*
stack_empty correct on new stack
new stack isn't full
stack_full is true for a full stack
stack_empty is false for a full stack
stack_peek returns the correct value
This should print: 9 8 7 6 5 4 3 2 1 0
 9 8 7 6 5 4 3 2 1 0
stack_empty correct for empty stack
stack_destroy returns the correct value
*/
```