1. **Abstract idea of a stack:**

The *stack* is a very common data structure used in programs. By *data structure*, we mean something that is meant to *hold* data and provides certain *operations* on that data.

One way to describe how a stack data structure behaves is to look at a physical analogy, a stack of books...

Now, a *minimal* set of things that we might do with the stack http://www.cs.bu.edu/teaching/c/stack/array/books1.gif are the following:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Place a book on the top...** | http://www.cs.bu.edu/teaching/c/stack/array/books2.gif | **Take one off the top...** | http://www.cs.bu.edu/teaching/c/stack/array/books1.gif | **See if the stack is empty...** | http://www.cs.bu.edu/teaching/c/stack/array/books1.gif | **NOT Empty**, there are books on the stack! |

We'll consider other, more complex operations to be inappropriate for our stack. For example, pulling out the 3rd book from the top cannot be done *directly* because the stack might fall over.

*How might you get the 3rd book from the top using only the simple operations?*

**Abstraction**

Now, let's think about a stack in an abstract way. I.e., it doesn't hold any particular kind of thing (like books) and we aren't restricting ourselves to any particular programming language or any particular implementation of a stack.

Stacks hold objects, usually all of the same type. Most stacks support just the simple set of operations we introduced above; and thus, the main property of a stack is that objects go on and come off of the *top* of the stack.

Here are the minimal operations we'd need for an abstract stack (and their typical names):

* + Push: Places an object on the *top* of the stack.
  + Pop: Removes an object from the *top* of the stack and produces that object.
  + IsEmpty: Reports whether the stack is empty or not.

Because we think of stacks in terms of the physical analogy, we usually draw them vertically (so the **top** is really *on top*).

1. ***Order* produced by a stack:**

Stacks are linear data structures. This means that their contexts are stored in what looks like a line (although vertically). This linear property, however, is not sufficient to discriminate a stack from other linear data structures. For example, an array is a sort of linear data structure. However, you can access any element in an array--not true for a stack, since you can only deal with the element at its top.

One of the distinguishing characteristics of a stack, and the thing that makes it useful, is *the order* in which elements come out of a stack. Let's see what order that is by looking at a stack of letters...

Suppose we have a stack that can hold letters, call it stack. What would a particular sequence of Push and Pops do to this stack?

We begin with stack empty:

-----

stack

Now, let's perform Push(stack, A), giving:

-----

| A | <-- top

-----

stack

Again, another push operation, Push(stack, B), giving:

-----

| B | <-- top

-----

| A |

-----

stack

Now let's remove an item, letter = Pop(stack), giving:

----- -----

| A | <-- top | B |

----- -----

stack letter

And finally, one more addition, Push(stack, C), giving:

-----

| C | <-- top

-----

| A |

-----

stack

You'll notice that the stack enforces a certain *order* to the use of its contents, i.e., the *Last* thing *In* is the *First* thing *Out*. Thus, we say that a stack enforces **LIFO** order.

*Now we can see one of the uses of a stack...To reverse the order of a set of objects.*

1. **Implementing a stack with an array:**

Let's think about how to implement this stack in the C programming language.

First, if we want to store letters, we can use type char. Next, since a stack usually holds a bunch of items with the same type (e.g., char), we can use an array to hold the contents of the stack.

Now, consider how we'll use this array of characters, call it contents, to hold the contents of the stack. At some point we'll have to decide how big this array is; keep in mind that a normal array has a fixed size.

Let's choose the array to be of size 4 for now. So, an array getting **A**, then **B**, will look like:

-----------------

| A | B | | |

-----------------

0 1 2 3

contents

*Is this array sufficient, or will we need to store more information concerning the stack?*

**Answer:** We need to keep track of the *top* of the stack since not all of the array holds stack elements.

*What****type****of thing will we use to keep track of the top of the stack?*

**Answer:** One choice is to use an integer, top, which will hold the array index of the element at the top of the stack.

**Example:**

Again suppose the stack has (A,B) in it already...

stack (made up of 'contents' and 'top')

----------------- -----

| A | B | | | | 1 |

----------------- -----

0 1 2 3 top

contents

Since **B** is at the top of the stack, the value *top* stores the index of **B** in the array (i.e., 1).

Now, suppose we push something on the stack, Push(stack, 'C'), giving:

stack (made up of 'contents' and 'top')

----------------- -----

| A | B | C | | | 2 |

----------------- -----

0 1 2 3 top

contents

(Note that both the *contents* and *top* part have to change.)

So, a sequence of pops produce the following effects:

* + letter = Pop(stack)
  + stack (made up of 'contents' and 'top')
  + ----------------- ----- -----
  + | A | B | | | | 1 | | C |
  + ----------------- ----- -----
  + 0 1 2 3 top letter
  + contents
  + letter = Pop(stack)
  + stack (made up of 'contents' and 'top')
  + ----------------- ----- -----
  + | A | | | | | 0 | | B |
  + ----------------- ----- -----
  + 0 1 2 3 top letter
  + contents
  + letter = Pop(stack)
  + stack (made up of 'contents' and 'top')
  + ----------------- ----- -----
  + | | | | | | -1| | A |
  + ----------------- ----- -----
  + 0 1 2 3 top letter
  + contents

so that you can see what value *top* should have when it is empty, i.e., -1.

Let's use this implementation of the stack with **contents** and **top** fields.

*What happens if we apply the following set of operations?*

* + Push(stack, 'D')
  + Push(stack, 'E')
  + Push(stack, 'F')
  + Push(stack, 'G')

giving:

stack (made up of 'contents' and 'top')

----------------- -----

| D | E | F | G | | 3 |

----------------- -----

0 1 2 3 top

contents

*and then try to add****H****with Push(stack, 'H')?*

*Thus, what is the disadvantage of using an array to implement a stack?*

1. **Dynamically-sized stack:**

Now, we will add one more *choice* to how we'll implement our stack. We want to be able to decide the maximum size of the stack at run-time (not compile-time).

Thus, we cannot use a regular array, but must use a pointer to a dynamically-allocated array.

*Now, will we need to keep track of any more information besides the****contents****and****top****?*

**Answer:** Yes! We'll need to keep the size of this array, i.e., the *maximum size* of the stack. We'll see why this is necessary as we write the code.

1. **C code:**

Now, let's think about how to actually code this *stack of characters* in C.

It is usually convenient to put a data structure in its own module, thus, we'll want to create files stack.h and stack.c.

Now, there are 2 main parts to a C data structure: the *data types* needed to keep track of a stack and the *functions* needed to implement stack operations.

* + The main *data type* we need is a type that people can use to declare new stacks, as in:
  + *type-of-a-stack* s1, s2;
  + Some of the functions we'll need come directly from the operations needed for an abstract stack, like:
  + StackPush(*ref-to-s1*, 'A');
  + ch = StackPop(*ref-to-s2*);

Notice how each stack operation needs some way to refer to a specific stack (so that we can have more than one stack at a time).

We may need to add a few other operations to help implement a stack. These will become apparent as we start to implement the stack. Remember that we need to put prototypes for each stack function instack.h and put the function definitions (bodies) in stack.c.

Before we ponder the details of the stack functions, we should decide on the types we need...

1. **Data types for a stack:**

For the array implementation, we need to keep track of (at least) the array **contents** and a **top** index. *How could we combine these 2 into a single C construct of type****stackT****?*

**Answer:** Put them into a struct.

So, our stack types become:

typedef char stackElementT; /\* Give it a generic name--makes \*/

/\* changing the type of things in \*/

/\* the stack easy. \*/

typedef struct {

stackElementT \*contents;

int top;

/\* Other fields? \*/

} stackT;

Note that the *contents* is a pointer since it will be dynamically-allocated.

*Are any other fields needed?* Well, remember that the maximum size of the array is determined at run-time...We'll probably need to keep that value around so that we can tell when the stack is full... The final type, thus, is:

typedef struct {

stackElementT \*contents;

int top;

int maxSize;

} stackT;

These types should go in the interface, stack.h.

1. **Filling in stack functions:**

Now that we've decided on the data types for a stack, let's think about the functions we need...

First, we need the standard stack operations:

StackPush()

StackPop()

StackIsEmpty()

We'll use the convention of placing the data structure name at the beginning of the function name (e.g., *Stack*IsEmpty). That will help us down the line. For example, suppose we use 2 different data structures in a program, both with *IsEmpty* operations--our naming convention will prevent the 2 different IsEmpty functions from conflicting.

We'll also need 2 extra operations:

StackInit()

StackDestroy()

They are not part of the *abstract* concept of a stack, but they are necessary for setup and cleanup when writing the stack in C.

Finally, while the array that holds the contents of the stack will be dynamically-allocated, it still has a maximum size. So, this stack is unlike the abstract stack in that it can get full. We should add something to be able to test for this state:

StackIsFull()

1. **StackInit():**

The first function we'll implement is StackInit(). It will need to set up a stackT structure so that it represents an *empty stack*.

Here is what the prototype for StackInit() looks like...

void StackInit(stackT \*stackP, int maxSize);

It needs to change the stack passed to it, so the stack is passed by reference (stackT \*). It also needs to know what the maximum size of the stack will be (i.e., maxSize).

Now, the body of the function must:

* + Allocate space for the contents.
  + Store the maximum size (for checking fullness).
  + Set up the top.

Here is the full function:

void StackInit(stackT \*stackP, int maxSize)

{

stackElementT \*newContents;

/\* Allocate a new array to hold the contents. \*/

newContents = (stackElementT \*)malloc(sizeof(stackElementT)

\* maxSize);

if (newContents == NULL) {

fprintf(stderr, "Insufficient memory to initialize stack.\n");

exit(1); /\* Exit, returning error code. \*/

}

stackP->contents = newContents;

stackP->maxSize = maxSize;

stackP->top = -1; /\* I.e., empty \*/

}

Note how we make sure that space was allocated (by testing the pointer against NULL). Also, note that if the stack was not passed by reference, we could not have changed its fields.

1. **StackDestroy():**

The next function we'll consider is the one that cleans up a stack when we are done with it. It should get rid of any dynamically-allocated memory and set the stack to some *reasonable state*.

This function only needs the stack to operate on:

void StackDestroy(stackT \*stackP);

and should reset all the fields set by the initialize function:

void StackDestroy(stackT \*stackP)

{

/\* Get rid of array. \*/

free(stackP->contents);

stackP->contents = NULL;

stackP->maxSize = 0;

stackP->top = -1; /\* I.e., empty \*/

}

1. **StackIsEmpty() and StackIsFull():**

Let's look at the functions that determine emptiness and fullness. Now, it's not necessary to pass a stack by reference to these functions, since they do not change the stack. So, we could prototype them as:

int StackIsEmpty(stackT stack);

int StackIsFull(stackT stack);

However, then some of the stack functions would take pointers (e.g., we need them for StackInit(), etc.) and some would not. It is more *consistent* to just pass stacks by reference (with a pointer) **all the time**. Furthermore, if the struct stackT is large, passing a pointer is more efficient (since it won't have to copy a big struct).

So, our prototypes will be:

int StackIsEmpty(stackT \*stackP);

int StackIsFull(stackT \*stackP);

**Emptiness**

Now, testing for emptyness is an easy operation. We've said that the *top* field is -1 when the stack is empty. Here's a simple implementation of the function...

int StackIsEmpty(stackT \*stackP)

{

return stackP->top < 0;

}

**Fullness**

Testing for fullness is only slightly more complicated. Let's look at an example stack.

Suppose we asked for a stack with a maximum size of 1 and it currently contained 1 element (i.e., it was full)...

stack (made up of 'contents' and 'top')

----- -----

| A | | 0 |

----- -----

0 top

contents

We can see from this example that when the *top* is equal to the *maximum size minus 1* (e.g., 0 = 1 - 1), then it is full. Thus, our fullness function is...

int StackIsFull(stackT \*stackP)

{

return stackP->top >= stackP->maxSize - 1;

}

This illustrates the importance of keeping the maximum size around in a field like maxSize.

1. **StackPush():**

Now, pushing onto the stack requires the stack itself as well as *something to push*. So, its prototype will look like:

void StackPush(stackT \*stackP, stackElementT element);

The function should place an element at the correct position in the *contents* array and update the *top*. However, before the element is placed in the array, we should make sure the array is not already full...Here is the body of the function:

void StackPush(stackT \*stackP, stackElementT element)

{

if (StackIsFull(stackP)) {

fprintf(stderr, "Can't push element on stack: stack is full.\n");

exit(1); /\* Exit, returning error code. \*/

}

/\* Put information in array; update top. \*/

stackP->contents[++stackP->top] = element;

}

Note how we used the *prefix* ++ operator. It increments the *top* index **before** it is used as an index in the array (i.e., where to place the new element).

Also note how we just reuse the StackIsFull() function to test for fullness.

1. **StackPop():**

Finally, popping from a stack only requires a stack parameter, but the value popped is typically returned. So, its prototype will look like:

stackElementT StackPop(stackT \*stackP);

The function should return the element at the top and update the *top*. Again, before an element is removed, we should make sure the array is not empty....Here is the body of the function:

stackElementT StackPop(stackT \*stackP)

{

if (StackIsEmpty(stackP)) {

fprintf(stderr, "Can't pop element from stack: stack is empty.\n");

exit(1); /\* Exit, returning error code. \*/

}

return stackP->contents[stackP->top--];

}

Note how we had the sticky problem that we had to update the *top* before the function returns, but we need the *current value of top* to return the correct array element. This is accomplished easily using the *postfix*-- operator, which allows us to use the current value of *top* before it is decremented.

1. **Stack module:**

Finally, don't forget that we are putting this stack in its own module. The stack types and function prototypes should go in stack.h. The stack function definitions should go in stack.c.

People that need to use the stack must include stack.h and link their code with stack.c (really, link their code with its object file, stack.o).

Finally, since we wrote the types and functions for a stack, we know how to use a stack. For example, when you need stacks, declare stack variables:

stackT s1, s2;

When you pass a stack to stack functions, pass it by reference:

StackInit(&s1, 10);

StackPush(&s1, 'Z');

1. **Testing the implementation:**

We've written the complete stack module for you ([stack.h](http://www.cs.bu.edu/teaching/c/stack/array/download/stack.h) and [stack.c](http://www.cs.bu.edu/teaching/c/stack/array/download/stack.c)). Also, here's a sample main program [stacktest.c](http://www.cs.bu.edu/teaching/c/stack/array/download/stacktest.c) and a [Makefile](http://www.cs.bu.edu/teaching/c/stack/array/download/Makefile).

The program demonstrates one use of the ordering produced by a stack...*What is it?*