HW 5 - Part I

1A: Understanding Stacks and Abstract Data Types in C

1B: writing a symbol-matching program using a stack.

Key concepts: stack operations, header files, pointers to structures, separate compilation, information hiding, dynamic memory allocation.

A stack is a Last-In-First (LIFO) out data structure. The main operations are push (add new item on top of stack) and pop (remove top item from stack).

Data Abstraction: the main idea here is to separate ***what*** operations can be performed on a data type (in this case a stack) from ***how*** those operations are implemented. In this lab, a header file is used to create this separation.

Part 1A:

Complete the following exercises (1)-(5). Give you answers by editing your own copy of this document.

(1) UNDERSTANDING STACK OPERATIONS

Starting from an empty stack, suppose we perform the operations below. In the right column of the table show the state of the stack with the TOP of the stack on the right.

push 5	5
push 3	3,5
push 7	7,3,5
pop	3,5
push 3	3,3,5
push 4	4,3,3,5
pop	3,3,5
pop	3,5

(2) COMPILING A STACK LIBRARY AND CLIENT PROGRAM:

Download the files **`stack.c`**, **`stack.h`** and **`main.c`** (from the src subdirectory or by downloading and unpacking the `.zip` file). The code is made up of three files: the "library" files **`stack.c`** and **`stack.h`**, and the main ("client") program **`main.c`**.

Browse the source files. Notice that `stack.c` does not have a main function and that `main.c` calls functions defined in `stack.c` (and declared in `stack.h`) Try to compile `main.c` as you normally would (with `gcc` or `clang`):

```
gcc main.c
```

What happens? Take a guess as to why it happens?

Here, main.c is fully dependent on stack.h and stack.c. As of now, main.c doesn't know what stk_push, stk_pop and other structs are. To solve this problem i think we should link the the stack files during compilation. After linking,the main.c will know what those structs are and the output of it.

So, how do we compile this program? We first need to compile the stack "module" into an "object file":

```
gcc -c stack.c
```

List the files in the directory. There should be a new file `stack.o` there. Let's find out something about this file; execute the following:

```
file stack.o
```

Now let's compile `main.c`. Since main uses functions from the stack module, we need to tell the compiler to "link" with the stack object file `stack.o`.

```
gcc stack.c stack.o -o stk_tst
```

Go ahead and run the program to see if it behaves as you expect.

(3) CODE OVERVIEW

The files `stack.c` and `stack.h` specify an implementation of a stack in C. Together they give us a general purpose library which can be used in multiple programs that need a stack.

Think of the header file `stack.h` as the "*interface specification*" -- the operations that a client can perform on a stack.

Think of the file `stack.c` as the *"implementation"* of those operations.

A client uses the library by first calling the `stk_create()` function to dynamically allocate and initialize a stack "object". From that point onward, the client uses the other `stk_xxx` functions to perform standard operations on the stack.

The client calls `stk_free` when done with the stack so memory can be reclaimed.

(4) UNDERSTANDING TYPEDEFS AND VISIBILITY BETWEEN FILES:

Go back to the header file `stack.h` where you'll find some typedefs.

- The type **`StackStruct`** is essentially an alias for a structure `struct stack_struct`
- The type **`StackPtr`** is a pointer to the same kind of structure (`struct stack_struct`)

Now try this: in **`main.c`** declare a variable of type **`StackStruct`** (instead of a pointer to such a struct) like this:

```
StackStruct ss;
```

Try to compile `main.c` again. What happens? It should fail; what message do you get? Observations/comments?

```
Message : error: variable has incomplete type 'StackStruct' (aka
    'struct stack_struct')
  StackStruct ss;
./stack.h note: forward declaration of 'struct stack_struct'
typedef struct stack_struct *StackPtr;
```

The compilation fails.
The Variable StackStruct ss has incomplete data types. According to Stack.h,It's a name for stack_struct struct.

Ok, where the heck is this thing "`struct stack_struct`"? What fields does it have?

It's in the Stack.c file. It contains int Elemtype items[CAPACITY] and int top in it.

Did you find it in `main.c`? Aha, the struct contains two fields: an array of elements **`items[]`** and an integer **`top`** (which stores the index of the top of the stack; by convention, if the stack is empty, top is set to -1). There is a reason for this organization which hopefully will be clear by the end of this lab.

(5) ACCESSING (AND HIDING) DATA FROM THE CLIENT:

Ok, remove the offending line from `main.c` to restore it to its initial form. `StackPtr` is a *pointer* to a struct, let's try to access the fields inside the struct (you found the struct in `stack.c` right?)

Recall that to access a structure field through a pointer, you can use the * operator like this (for example):

```
(*ptr).field_name = new_value;
```

Or you can equivalently use the -> operator (read "follow"):

```
ptr->field_name = new_value
```

So, in **`main.c`**, insert a statement that tries to set the top field to some number. Save and recompile. What happens. If it fails, what error message do you get? Is this behavior good/ useful? Imagine that the stack code is written by a different person (maybe at a different company and at a different time) than the client code.

It's good because the client won't be able to see the actual code. It gives an error,

It's the same thing as (4), it needs proper datatype.

TAKEAWAYS:

- Encapsulation: bundling up everything we need (for a stack in this case) into a single data structure (and type)
- Interface: the operations that can be performed on this new type (specified in a .h file in C).
- Data hiding: using language features to make sure that *only* these operations are performed.
- Using a pointer to a struct as a "handle" to such a data structure.

Part 1B: