**Lecture 37 – CONCRETE STACK & STACK TESTS**

Types and Pointers

Pointers should be in this format in C:

* int \*pointer 🡪 “pointer is a pointer to an int”
* int \*p1 🡪 “p1 is a pointer to an int”

Stack Type

Implementing a type called a STACK, in 3 steps:

1. Design the type
2. Write tests for the type
3. Implement the type

We have a single type called STACK, so we can call variables of type stack and do things to it that you would expect to do to a stack

**Writing the stack**, we’ll have the following files:

* **Stack.c** 🡪 Implementation of the stack type
  + Defines the type and the functions + Implements the functions (actual code to do stuff we want to do on the stack)
  + Doesn’t contain a main
* **Stack.h** 🡪 Interface of the stack (things that other things need to know to be able to interact with the stack)
  + Doesn’t contain a main
* **testStack.c** 🡪 test stack.c
  + Written to test Stack.c
  + Contains a main, because when we compile testStack.c to run against Stack.c/Stack.h we get a little program that runs tests on them.
  + A program whose sole job is to test the stack.  
    E.g. an assembly line for seatbelts, Stack.h tells us how seatbelts behave, Stack.c is building the seatbelts, testStack.c tests the seatbelts and tries to break them. When you put the seatbelts into the car, you don’t include the testStack.c test program into the car too.
* **THE OTHER IN WHICH WE WRITE THE STACK GOES FROM: (1) STACK.H 🡪 (2) TESTSTACK.C 🡪 (3) STACK.C**

Designing the stack type (stack.h) (interface)

Design: What should be the roles and responsibilities of various software and how should they all fit together?

* There are many decisions to make when coding, of which the most important are made during the design phase, since they are harder to change once implemented.

Example STACK DECLARATION:

* #define STACK\_SIZE 1000
* **Typedef char stack [STACK\_SIZE];** 🡪 “Typedef’ing stack to be an array of characters, of size 1000”
* We want to use this type over and over again, so we need to typedef it.

We design our stack with our basic functions:

* FN1: PUSH/ADD 🡪 **Stack add (stack s, char elt);** 🡪 adds an element to the top of the stack
* FN2: TOP 🡪 **Char top (stack s);** 🡪 returns what exists at the top of the stack
* FN3: POP 🡪 **Stack pop (stack s);** 🡪 will take the top element OFF the stack

Writing tests for the functions (testStack.c)

Using asserts, you test the functions you want to make and that will be the implementation.

* It would be unfair to test functions that don’t exist or the creator wouldn’t have prior knowledge of.
* Need to test for risks that might occur, e.g. when creating the stack, make sure that it doesn’t overflow 🡪 knowing how big the array is
* You can use the test function to “police” the function
* But we have no way of knowing how big an array is if only a pointer to the first element is passed into the function.

To get around this:

* + Pass in the size of the array (when arrays are passed around, they are just pointers to the array)
  + Pass in the pointer to the final element in the array
  + Have the size of certain arrays #define-d so everyone can refer to it easily
  + Have asserts to prevent overflows
  + Use structs
  + Have a function which asks how big the stack is
  + Notify the user whether they are going to overflow an array. Return an error message if this occurs.
* Example test:
  + s = push (s, ‘a’); // perform a function that alters the stack 🡪 In this case adding an element to top of stack
  + assert (top(s) == ‘a’); // now use another FN to check that both functions worked
  + We use the test FN’s to police the functions of the implementation

Implement the stack functions (Stack.c)

Make the stack work and interact with memory.

Stack.c is where the functions are stored / called. Example functions:

* Stack push (stack s, char element) { // add element functions
* Char top (stack s) // find the top element
* Stack pop (stack s) // takes an element off the stack

With a stack, you can either ADD/PUSH (insert operation) or POP (delete operation) something.

**Lecture 38 – TOWARDS ABSTRACTION (moving from concrete tests 🡪 abstract tests)**

REMINDER: (1) Design the stack / stack.h (2) Write tests / testStack.c (3) Implement it / stack.c

DESIGNING

Bad Example:

* Typedef struct \_stack {
  + Char stack[STACK\_SIZE];
  + Int size;
  + Int items;
* } stack;

Problem: everything in this stack is about a stack, so having “char stack[STACK\_SIZE]” is not very useful, because it is not clear what is being stored in the array.

Better Example:

* Typedef struct \_stack {
  + Char item[STACK\_SIZE];
  + Int size;
  + Int numItems;
* } stack;

TESTING

To test the correctness of the stack, all we have to test is that the interface functions work. In particular, the sequence of the functions is correct.

* Write ASSERTS that are based off interface functions etc., rather than based off your own inner-secret functions

In the long-run, you only want to have ABSTRACT TESTS.

* Instead of asserting exact values, you assert the whole functions for **future-proof testing**.

**Bad/ Old Method/ Concrete Testing:**



* Intimately links with what the stack looks like as you need to know what’s actually inside the stack, such as the element numbers and variable names.
* There is a possibility for changes in the design of the struct, therefore this type of assert is not good because it is prone to breaking, since you are not testing the interface function.

**Good/ New Method/ Abstract Testing:**



* The assert doesn’t depend on how you design the struct.
* For a type like this, we are trying to test that the functions work well together.
  + The sequence of your operations count, but not what you’re operating on.

By abstracting our code, debugging becomes a “divide and conquer” process, where we can continually zoom into the code, finding where the problem is. The bad implementation is our process of testing, whereas the good implementation is our process of debugging.

* E.g. USB’s can only be interacted through the interface, so we only care about what’s in the standard. If you rely on anything else that’s not in the standard, then things will break.
* E.g. Microsoft’s code had a standard for everyone to use, but it also had hidden functions that were not part of the standard. People started using these functions because they were more efficient, but when MS updated their systems, they only worried about the standards and changed or removed their hidden functions in the update. As a result, a lot of people’s programs broke
* Similarly, when we are testing game.c, because everyone’s implementation may be different, we can only rely on the interface functions.
* Your testGame.c will defend you against bad game.c files, because your AI can only be matched up with games that pass the tests. To not be paired with a crap game, you write really good tests/ asserts in your testGame.c file which will fail the broken games and wont pair your AI with them.

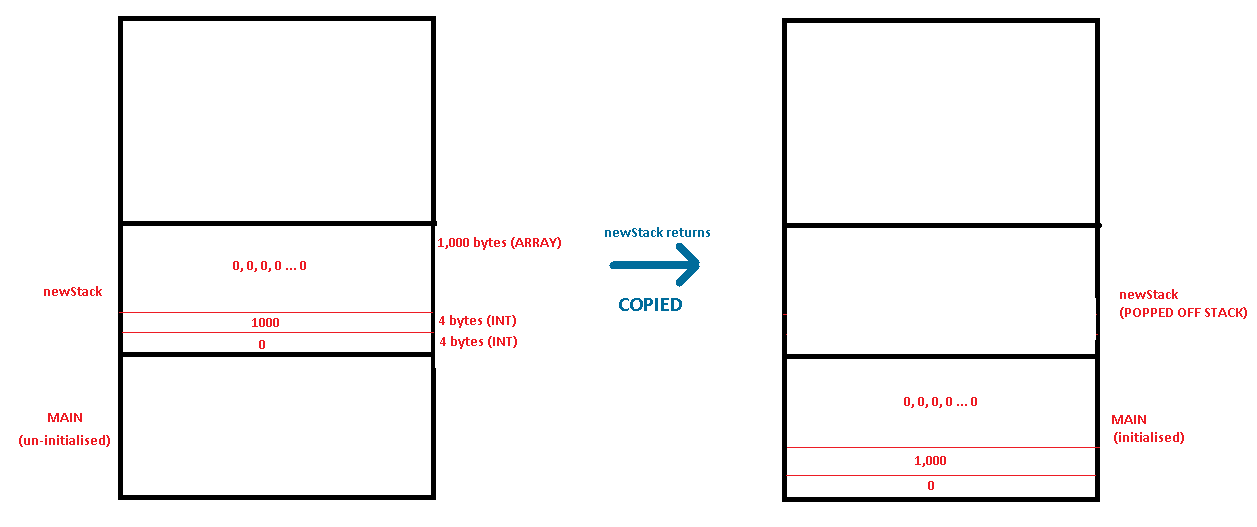
**This applies for the rest of your computing course. Always write tests first, instead of trying to do the program in one go, because you will definitely waste time going back and fixing things after you make it. You write one function 🡪 write tests for it 🡪 Write your second function 🡪 write tests for it 🡪 Write your third function etc. etc.** **DO LITTLE BABY STEPS AND TEST AS YOU GO!!!**

Extra info about the AI

* The idea of making types and accessing them through an interface is very powerful.
* The only way you ever access the game type is by using the game interface functions, then you can change the game type as much as you like in the future and everything will still work 🡪 An ABSTRACT TYPE
* **ABSTRACT TYPE** is a type where you ONLY access it through the interface functions.
  + You don’t know what the structs are inside it
  + E.g. for the HEXAGONAL GAME BOARD: Everyone will represent their struct differently, but if we do it abstractly, it will work with everyone else’s game because we are accessing their structs only through the interface functions.

**Lecture 39 – INTERFACES & ADTs**

* Stacks contain **2 x ints** and **1 x array**
* Does it matter that a stack contains an array?
  + Will it set up 2 x ints and 1 x POINTER to the array? OR 2 x ints and 1 x the WHOLE array?
  + Structs are returned by copy, so we don’t need to worry about memory that will be overridden (like in pointers).
  + ANS:  = 1008



Designing the Interface and Planning

Keep it simple stupid.

* The simpler the interface, the more universal it is for interacting with other code.
* You don’t need a function to do every single thing possible. Just the important stuff is enough.
* ASK YOURSELF: Is this function useful? What is the cost of putting it in VS cost of not putting it in?
* For every “cool (ahem useless)” function you make, you have to code it, which will increase the chance of errors coming up and breaking the whole program.
* We want to be quick, fast and agile so **design the interface to be as small as possible**. If we are given a huge complex system, we will have less incentive to use it.
* You can have lots of functions in the interface, but then the functions become very specific and also time consuming to make.
  + Many functions may also have to interact with each other in the tests, so if one function fails, lots of tests won’t be passed.
* Think of a function as **something that is incredibly useful to have right now**, NOT its potential in the future.

To **get/ retrieve information** from an abstract data type, the name of the function follows:

* <return type> get<what you’re getting> (<from what type>);
* E.g. int getSize (stack s, index)

To **set/ place information** into a data type:

* E.g. int setSomething (stack s. index)

Testing and Implementation

Start with easy functions before creating the harder functions.

* After creating the easy functions, the harder functions may seem less difficult and possibly even unnecessary.
* If we start at the harder functions, we spend too much time worrying about them which will slow down progress.

When writing functions, make sure you always check for errors that can be produced through user input.

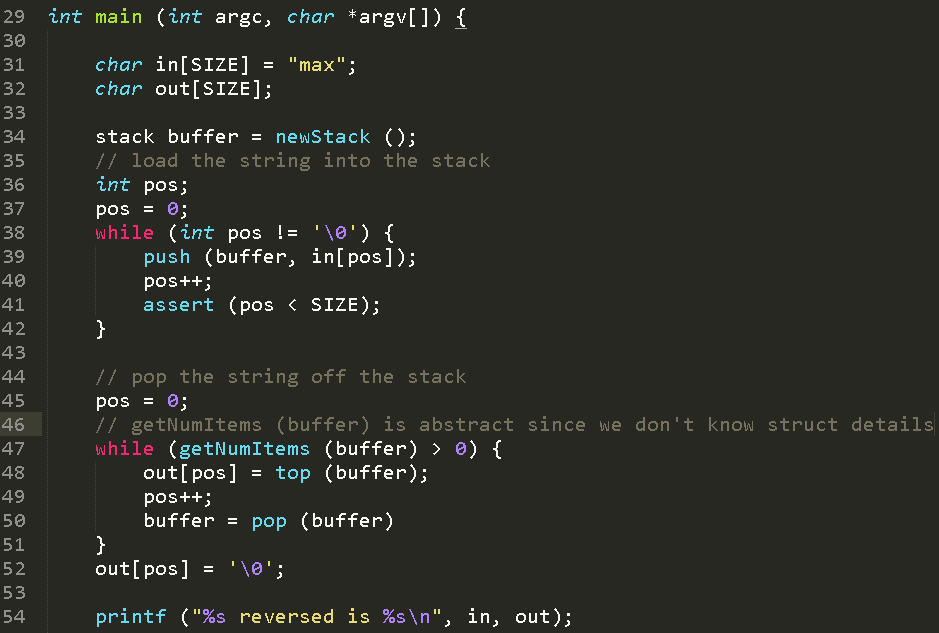
* E.g. Check for **Buffer Overflows** and that you have **spare capacity to add/remove/alter information** in the data type.

**Lecture 39 – USING ADTs**

EXAMPLE: Reversing a string using structs

* We need a program that does this:
  + IN: |’M’|’A’|’X’|’\0’| 🡨 Needs the null at the end to become a string, otherwise it is just 3 characters.
  + OUT: |’X’|’A’|’M’|’\0’|
* We PUSH ‘M’ 🡪 ‘A’ 🡪 ‘X’ into the stack, then POP ‘X’ 🡪 ‘A’ 🡪 ‘M’ off the stack and write it into the array.

CODE EXAMPLE: Reversing a string using structs



* **BUFFER** is a temporary array, used for storing things, and then taking out later.
* Users should never know what is inside a data structure, because as soon as they can, you will have to support all the undocumented functions/ features that they are using or else, otherwise things will mysteriously break.

Using **\* Game** means that it is a pointer to a struct \_game.

* In Game.c we say:
  + Typedef struct \_game { /\* code \*/ } game;
* In game.h we say:
  + Typedef struct \_game \* Game { /\* code \*/ } game;
* In runGame.c we say:
  + Game
* Concrete types are defined starting with lowercase letter (e.g. **g**ame)
* Abstract types are defined starting with an uppercase letter (e.g. **G**ame)
* In Game.c we have the “gut” details of the struct, all the secrets which we don’t want to show people.
* In game.h we only have the “name” of the struct.  
  **“Let everyone know the name, but don’t tell anyone the guts”**