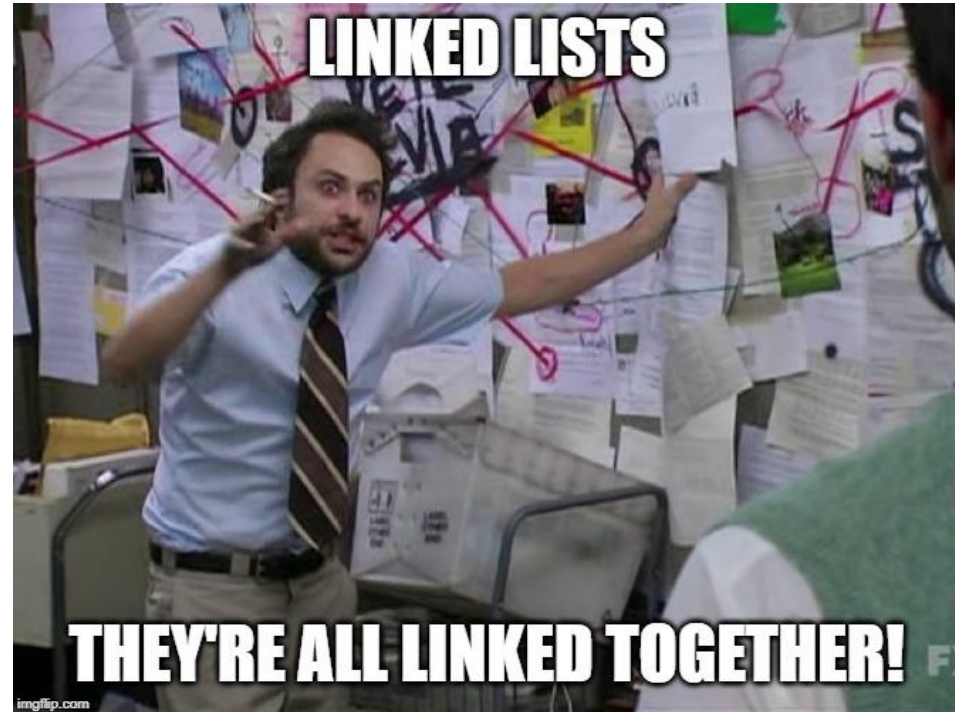

COMP1511 - Programming Fundamentals

— Week 9 - Lecture 15 —

What did we learn last week?

Linked Lists

- A complete working implementation of Linked Lists
- Inserting nodes
- Removal of nodes
- Cleaning our memory



What are we covering today?

Abstract Data Types

- A recap of Multiple File Projects
- More detail on things like typedef
- The ability to present capabilities of a type to us . . .
- . . . without exposing any of the inner workings

Recap - Multiple File Projects

Separating Code into Multiple files

- Header file (* .h) - Function Declarations
- Implementation file (* .c) - Majority of the running code
- Other files - can include a Header to use its capabilities

Separation protects data and makes functionality easier to read

- We don't have access to internal information we don't need
- We can't accidentally change something important
- We have a simple list of functions we can call

Using Multiple Files

Linking the Files

- A file that `#includes` the Header (`*.h`) file will have access to its functions
- It's own implementation (`*.c`) file will always `#include` it
- Implementation files are never included!

Compilation

- All Implementation files are compiled
- Header files are never compiled, they're included

An Example - CSpotify

Assignment 2 - CSpotify is a nice example

`cspotify.h`

- Contains only defines, typedefs and function declarations
- Is commented heavily so that it's easy to know how to use it

`cspotify.c`

- Contains actual structs
- Contains implementation of `cspotify.h`'s functions (once we've written them)

An Example - CSpotify

How some of the other files interact ...

`main.c`

- `#includes cspotify.h`
- Uses the functions in `cspotify.h`

`test_cspotify.c` with `test_main.c`

- `#includes cspotify.h`
- Is mutually exclusive with `main.c` because they both have main functions

Abstract Data Types

Types we can declare for a specific purpose

- We can name them
- We can fix particular ways of interacting with them
- This can protect data from being accessed the wrong way

We can hide the implementation

- Whoever uses our code doesn't need to see how it was made
- They only need to know how to use it

Typedef

Type Definition

- We declare a new Type that we're going to use
- **typedef** <original Type> <new Type Name>
- Allows us to use a simple name for a possibly complex structure
- More importantly, hides the structure details from other parts of the code

```
typedef struct library *Library;
```

- We can use **Library** as a Type without knowing anything about the struct underlying it

Typedef in a Header file

The Header file provides an interface to the functionality

- We can put this in a **header** (*.h) file along with functions that use it
- This allows someone to see a Type without knowing exactly what it is
- The details go in the *.c file which is not included directly
- We can also see the functions without knowing how they work

- We are able to see the **header** and use the information
- We hide the **implementation** that we don't need to know about

An Example of an Abstract Data Type - A Stack

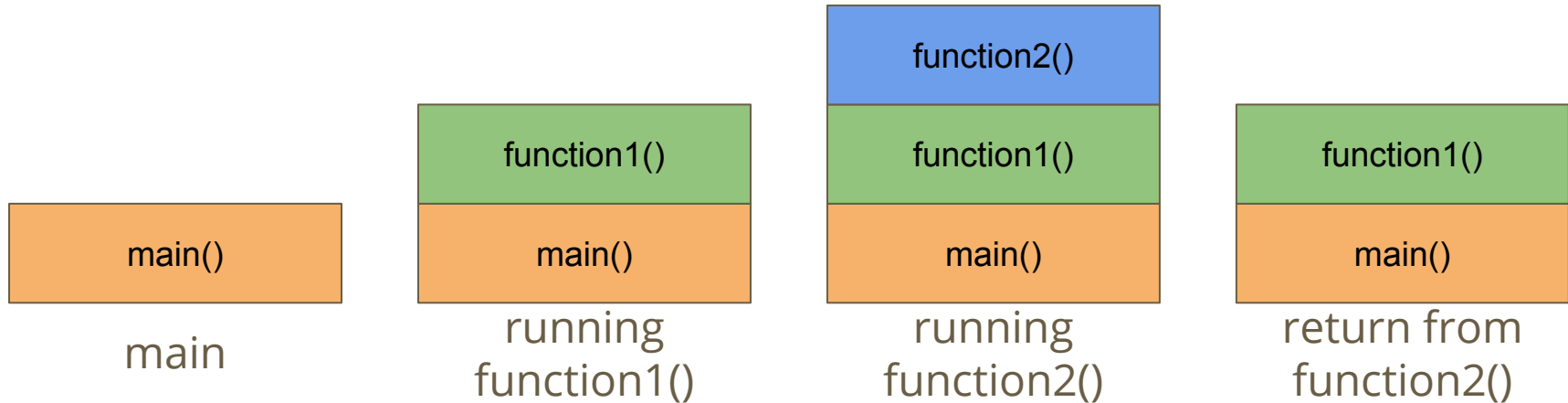
A stack is a very common data structure in programming

- It is a "Last in first out" structure
- You can put something on top of a stack
- You can take something off the top of a stack
- You can't access anything underneath

This is actually how functions work!

The currently running code is on the top of the stack

- `main()` calls `function1()` - only `function1()` is accessible
- `function1()` calls `function2()` - only `function2()` is accessible
- control returns to `function1()` when `function2()` returns



What makes it Abstract?

A Stack is an idea

- An Array or a Linked List is a very specific implementation
- A Stack is just an idea of how things should be organised
- There's a structure, but there's no implementation!

Abstract Data Type for a Stack

- We can have a header saying how the Stack is used
- The Implementation could use an Array or a Linked List to store the objects in the Stack, but we wouldn't know!

Break Time

Programming Languages

- C++, Java, C# and many others are based on C
- There are too many programming languages to count or learn!
- Remember the fundamentals!
- C syntax is not as important as your plans and thinking
- You will encounter many programming languages, some will feel very different from C in their approach
- But if you learn how you want to communicate with computers, the actual language you use will never be a barrier for you

Let's build a Stack ADT

We're only concerned with how we'll use it, not what it's made of

- Our user will see a "Stack" rather than an Array or Linked List
- We will start with a Stack of integers
- We will provide access to certain functions:
 - Create a Stack
 - Destroy a Stack
 - Add to the Stack (known as "push")
 - Remove from the Stack (known as "pop")
 - Count how many things are in the Stack

A Header File for a Stack

```
// stack type hides the struct that it is implemented as
typedef struct stackInternals *Stack;

// functions to create and destroy stacks
stack stackCreate(void);
void stackFree(Stack s);

// Push and Pop items from stacks
// Removing the item returns the item for use
void stackPush(Stack s, int item);
int stackPop(Stack s);

// Check on the size of the stack
int stackSize(Stack s);
```


What does our Header (not) Provide?

Standard Stack functions are available

- We can push or pop an element onto or off the Stack
- We are not given access to anything else inside the Stack!
- We cannot pop more than one element at a time
- We aren't able to loop through the Stack

The power of Abstract Data Types

- They stop us from accessing the data incorrectly!

Stack.c

Our *.c file is the implementation of the functionality

- The C file is like the detail under the "headings" in the header
- Each declaration in the header is like a title of what is implemented
- Let's start with a Linked List as the underlying data structure
- A Linked List makes sense because we can grow it and shrink it easily
- We can also look at how to implement this with arrays . . .

The implementation behind a type definition

We can create a pair of structs

- stackInternals represents the whole Stack
- stackNode is a single element of the list

```
// Stack internals holds a pointer to the start of a linked list
struct stackInternals {
    struct stackNode *head;
};

struct stackNode {
    struct stackNode *next;
    int data;
};
```

Creation of a Stack

If we want our struct to be persistent, we'll allocate memory for it

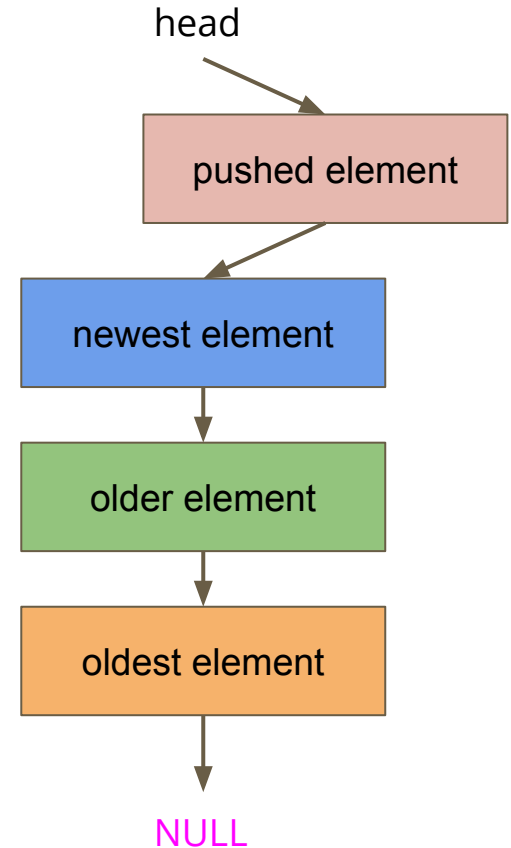
We create our Stack empty, so the pointer to the head is NULL

```
// Create an empty Stack
Stack stackCreate(void) {
    Stack newStack = malloc(sizeof(struct stackInternals));
    newStack->head = NULL;
    return newStack;
}
```

Pushing items onto the Stack

We push items onto the head of the Stack

- We can insert the new element at the head
- All the other elements will stay in the same order they were in



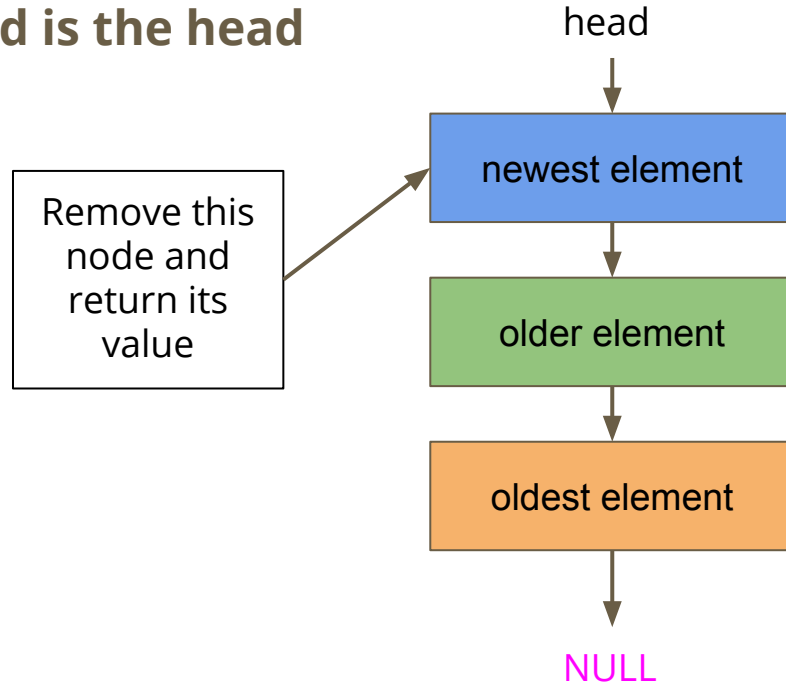
Code for Pushing

Adding to the head of a linked list is something we've done before

```
void stackPush(Stack s, int item) {  
    struct stackNode *newNode = malloc(sizeof(struct stackNode));  
    newNode->data = item;  
  
    // Attach newNode to the old head and make it the new head  
    newNode->next = s->head;  
    s->head = newNode;  
}
```

Popping (removing) a Node

The only node that can be popped is the head
(the top of the stack)



Code for Popping

```
// Remove the head from the list and free the memory used
int stackPop(Stack s) {
    if (s->head == NULL) {
        printf("Attempt to pop an element from an empty stack.\n");
        exit(1);
    }
    // Read the value from the head
    int returnData = s->head->data;
    struct stackNode *remNode = s->head;

    // move the stack head to the new head and free the old
    s->head = s->head->next;
    free(remNode);

    return returnData;
}
```


Testing Code in our Main.c

```
int main(void) {
    printf("Creating a deck of cards.\n");
    Stack deck = stackCreate();

    int card = 7;
    printf("Putting %d on top of the deck!\n", card);
    stackPush(deck, card);
    card = 10;
    printf("Putting %d on top of the deck!\n", card);
    stackPush(deck, card);

    printf("Card %d just got removed from the deck!\n", stackPop(deck));

    id = 3;
    printf("Putting %d on top of the deck!\n", card);
    stackPush(deck, card);
}
```

Other Functionality

There are some functions in the header we haven't implemented

- **Destroying and freeing the Stack**
- We're still at risk of leaking memory because we're only freeing on removal
- **Checking the Number of Elements**
- This would be very handy because it would allow us to tell how many elements we can pop before we risk errors
- You could even store an int in the Stack struct that increments every time you push and decrements every time you pop . . .

Different Implementations

Stack.c doesn't have to be a linked list . . . so long as it implements the functions in Stack.h

- We could use an array instead
- Our data can be stored in an array with a large maximum size
- We'll keep track of where the top is with an int

stack.c

```
// Struct representing the stack using an array
struct stackInternals {
    int stack[MAX_STACK_SIZE];
    int top;
};

// create a new stack
stack stackCreate() {
    stack s = malloc(sizeof(struct stackInternals));
    s->top = 0;
    return s;
}
```

Push and Pop

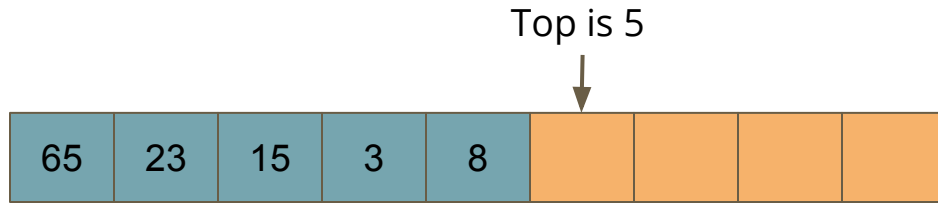
These should only interact with the top of the stack

- **Push** should add an element after the end of the stack
- It should then move the top index to that new element

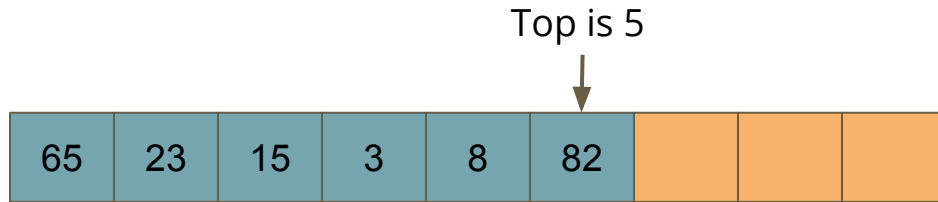
- **Pop** should return the element on the top of the stack
- It should then move the top index down one

Push

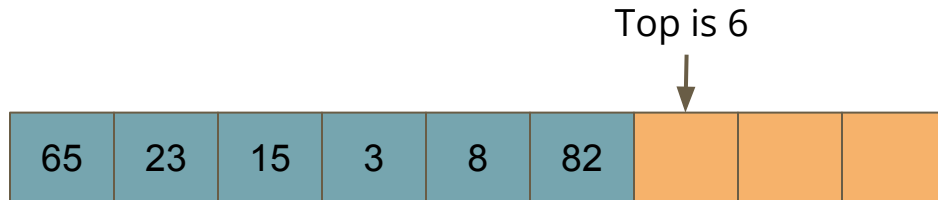
Push a new element "82" onto the stack



The stack starts like this



82 is added at top's index



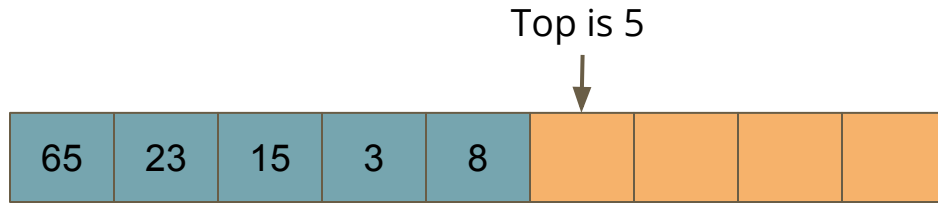
Top then moves up one

Push code

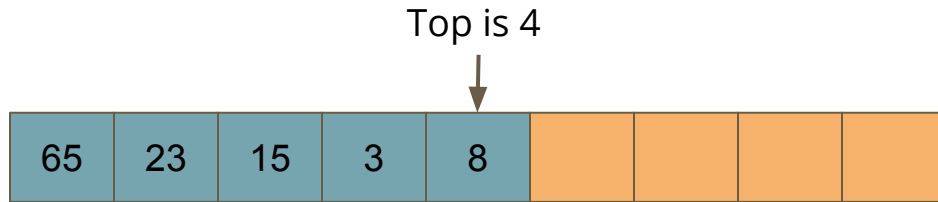
```
// Add an element to the top of the stack
void stackPush(stack s, int item) {
    // check to see if we've used up all our memory
    if(s->top == MAX_STACK_SIZE) {
        printf("Maximum stack size reached, cannot push.\n");
        exit(1);
    }
    s->stackData[s->top] = item;
    s->top++;
}
```


Pop

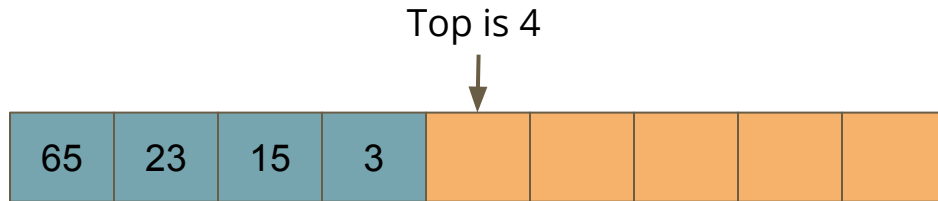
Pop removes the top element from the stack



The stack starts like this



Top moves down one



Read the element at top and return it

Pop code

```
// Remove an element from the top of the stack
int stackPop(stack s) {
    // check to see if the stack is empty
    if(s->top <= 0) {
        printf("Stack is empty, cannot pop.\n");
        exit(1);
    }
    s->top--;
    return s->stackData[s->top];
}
```

Hidden Implementations

Neither Implementation needs to change the Header

- The main function doesn't know the difference!
- The structures and implementations are hidden from the header file and the rest of the code that uses it
- If we want or need to, we can change the underlying implementation without affecting the main code

Other Abstract Data Types

Stacks are obviously not the only possibility here

- If we simply change the rules (last in, first out), we can make other structures
- A Queue is "first in, first out", and could be created using similar techniques
- There are many possibilities that we can create!

What did we cover today?

Abstract Data Types

- Makes use of Multi-file projects we discussed earlier
- `typedef` to protect a struct from open access
- Using multiple files to control how a type is used
- Hiding the implementation
- Providing a fixed interface
- Showing that different implementations can work with the same ADT