COMP1511 PROGRAMMING FUNDAMENTALS

LECTURE 15

Abstract Data Types: Stacks

LAST WEEK...

- Returned to the ice-cream shop example - rehashed and built on it (putting together a linked list example with inserting, traversing and deleting)
- Talked about using leakcheck command
- Speed ran through multi-file projects

TODAY...

- Multi File Projects - at slower speed with an actual example
- Basic Command Line Arguments
- Abstract Data Types: Stack

WHERE IS THE CODE?

Live lecture code can be found here:

HTTPS://CGI.CSE.UNSW.EDU.AU/~CS1511/22T1/LIVE/WEEK09/

LINKED LIST - ALWAYS THINK ABOUT!

- Some special boundary conditions that you need to consider when you manipulate lists:
  - Empty list
  - List with 1 element
  - Something happening at the beginning of the list
  - Something happening at the end of the list
  - Something will not occur, the item is not in the list (inserting after a number that doesn't exist etc)

MULTI FILE PROJECT

WHAT ARE THEY?

- Big programs are often spread out over multiple files. There are a number of benefits to this:
  - Improves readability (reduces length of program)
  - You can separate code by subject (modularity)
  - Modules can be written and tested separately
- So far we have already been using the multi-file capability. Every time we #include, we are actually borrowing code from other files
- We have been only including C standard libraries

MULTI FILE PROJECT

WHAT ARE THEY?

- You can also #include your own! (FUN!)
- This allows us to join projects together
- It also allows multiple people to work together on projects out in the real world
- We will also often produce code that we can then use again in other projects (that is all that the C standard libraries are - functions that are useful in multiple instances)

MULTI FILE PROJECT INCLUDES

.H FILE
.C FILE (MAYBE MULTIPLES)

- In a multi file project we might have:
- (multiple) header file - this is the .h file that you have been using from standard libraries already
- (multiple) implementation file - this is a .c file, it implements what is in the header file.
- Each header file that you write, will have its own implementation file
- a main.c file - this is the entry to our program, we try and have as little code here as possible

HEADER FILE

#INCLUDE "SOMETHING.H"

Typically contains:

- function prototypes for the functions that will be implemented in the implementation file
- comments that describe how the functions will be used
- #defines
- the file basically SHOWS the programmer all they need to know to use the code
- NO RUNNING CODE
- This is like a definition file

IMPLEMENTATION FILE

SOMETHING.C

This is where you implement the functions that you have defined in your header file

IMPLEMENTATION FILE

MAIN.C

This is where you call functions from that may exist in other modules.

AN EXAMPLE

A MATHS

- We will have three files:
  - header file - maths.h
  - implementation file - maths.c
    - #include ”maths.h”
  - main file - main.c
    - #include ”maths.h”

*maths.h

```c
// This is the header file for the maths module example
// The header file will contain:
//      - any #define
//      - function prototypes and any comments

#define PI 3.14

//Function prototype for a function that calculates
//square of a number
int square(int number);

//Function prototype for a function that calculates
//sum of two numbers
int sum(int number1, int number2);
```

main.c

```c
//This is the main file in our program
//This is where we drive the program from and where we
//make calls to our modules. We need to include the
//header file for each module that we want to use functions
//from

#include <stdio.h>
//Include the header file:
#include "maths.h"

int main (void) {
    int number = 13;
    int number2 = 10;

    printf("The square of the number %d is %d\n", number, square(number));
    printf("The sum of %d and %d is %d\n", number, number2, sum(number, number2));
    return 0;
}
```

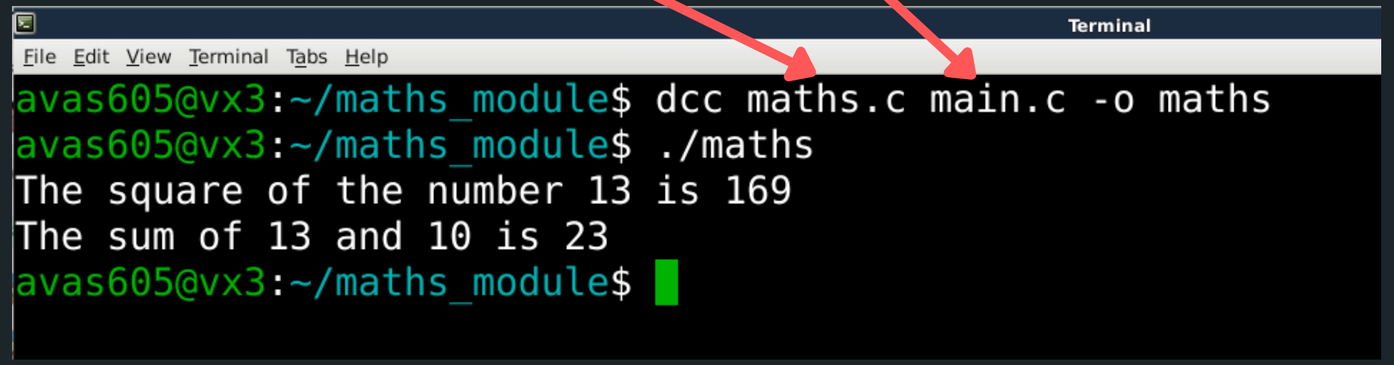
maths.c

```c
//This is the implementation file of maths.h
//We defined two functions in the header file,
//and this is where we will implement these two
//functions

//Include your header file in the implementation file
//by using the below syntax

#include "maths.h"

int square(int number) {
    return number * number;
}

int sum(int number1, int number2) {
    return number1 + number2;
}
```

COMPILING A MULTI FILE

COMPILE ALL C FILES IN THE PROJECT

- To compile a multi file, you basically list any .c files you have in your project
    - In the case of our example, we have a maths.c and a main.c file):



- The program will always enter in main.c, so there should only be one main.c when compiling

ABSTRACT DATA TYPES

WHAT ARE THEY?

- Abstract Data Types (ADT's) are data types whose implementation details are hidden from the user
  - What does this mean?
- A common example of an ADT is something called a Stack - it has set ways in which it works but it can implemented using a number of different ways (for example, using linked lists or using arrays)
- Whoever uses our code doesn't need to see how it was made
- They only really want to know how to use it

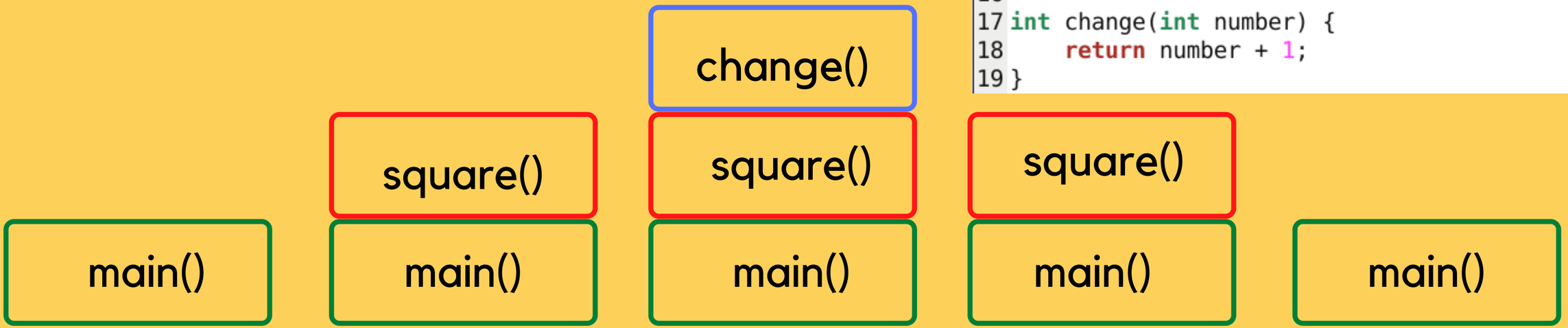
SO WHAT IS A STACK?

THINK A STACK OF DISHES, OR A STACK OF BOOKS

- A Stack is a Last In, First Out structure (LIFO)
- So you can put something on top of a stack and you can take something off the top of the stack, you cannot remove things from underneath (think of your dish stack toppling down!)

THIS IS HOW OUR MEMORY STACK WORKS FOR FUNCTIONS

```
 3 int main (void) {
 4     int number = 13;
 5     int new_number = 0;
 6
 7     new_number = new_number + square(number);
 8
 9     return 0;
10 }
11
12 int square(int number) {
13     int changed_number = change(number);
14     return changed_number * changed_number;
15 }
16
17 int change(int number) {
18     return number + 1;
19 }
```



the main() starts running on Line 3

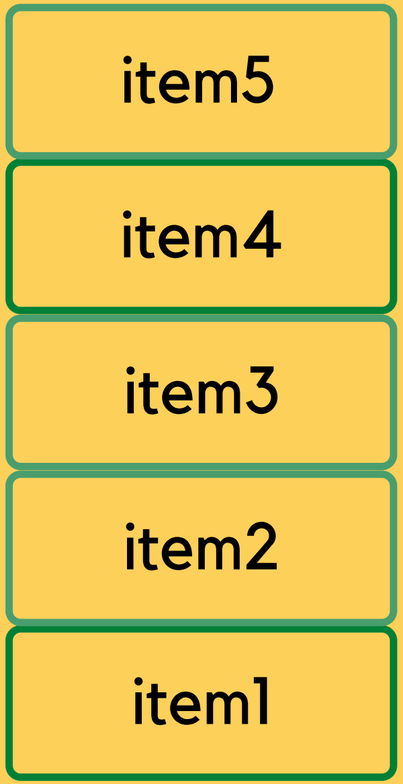
main() calls square() on Line 7 - only square() now accessible

square() calls change() on Line 13 - only change() now accessible

return from change() on Line 18 to square() - only square() now accessible

return from square() on Line 14 to main()

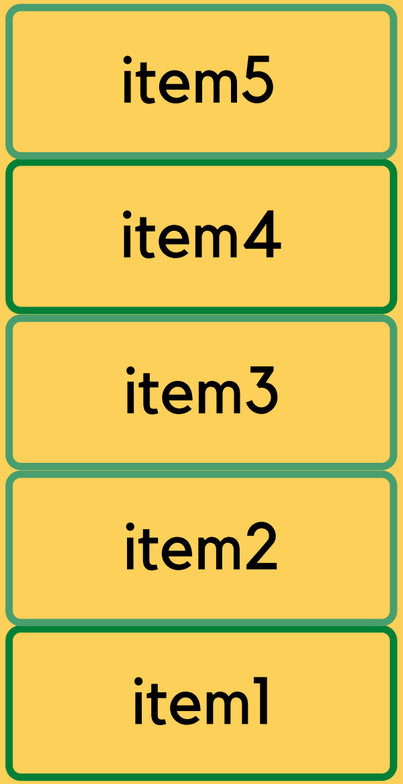
WHERE IS THE ABSTRACT PART?



- The idea of a stack is just that - an idea!
- Can you think of anywhere a Stack is applied in our everyday interactions with computers?

- A stack behaves in a certain way defined by a set of rules
- I am not given an implementation for this stack
  - I can do it using arrays
  - I can do it using linked lists
- So we could have a header file that just defines how the stack is used, but it could be implemented using arrays or linked lists and we would be none the wiser - doesn't matter as long as it follows the rules of a Stack!
-

SO WHAT ARE THE RULES OF A STACK?



- The Stack has two special terms:
  - push (onto the stack, so add the element to the top of the Stack)
  - pop (off the stack, take the top element off the Stack)
- Let's look at a few functions:
  - Create a Stack
  - Add to the Stack (push)
  - Take from the Stack (pop)
  - Count how many things are in the Stack
  - Destroy the Stack
- One header file (stack.h), and we will try two different implementations one with arrays and one with lists
  - stack_list.c & stack_array.c

HOW WILL THE HEADER FILE DEFINE THINGS FOR US?

- A stack is a structure, which we will not define in the header file, as our array and linked list files may use slightly different definitions of the same structure
- We will then define our functions in the header file:

```c
//This is the header file for the Stack
//This file describes the functions that should be implemented for the stack
//Sasha Vassar Week09 Lecture 15

#define MAX 100

//This function creates the initial stack, so it will return a pointer to the
//stack it has created, and we input nothing into it, as we are just creating
//an empty stack
struct stack *create_stack(void);

//This function pushes an item onto the stack - the function does not return
//anything, but is given the stack onto which the item is being pushed and the
//item to be pushed
void push_stack(struct stack *s, int item);

//This function pops an item off the stack - the function returns an
//int because it returns the value of the item it popped off and is given
//the stack from which they will be removing the item
int pop_stack(struct stack *s);

//This function returns the size of the stack (so how many items are there
//in this stack) - this means we are returned an int. And we give the
//function the stack that we want the size of.
int size_stack(struct stack *s);

//This function destroys the whole stack and will free the space that
//was allocated initially - the function is given the stack to destroy
//and does not return anything
void destroy_stack(struct stack *s);
```

LET'S DO A LINKED LIST IMPLEMENTATION FIRST

STACK: DEFINING A LINKED LIST STACK

```c
// Define the stack structure itself, the stack structure in this case will
// have a size and a top node (which is the head)
struct stack {
    struct node *top;
    int size;
};
// Define each element of a stack as a node
struct node {
    int data;
    struct node *next;
};
```

LET'S DO A LINKED LIST IMPLEMENTATION FIRST

STACK

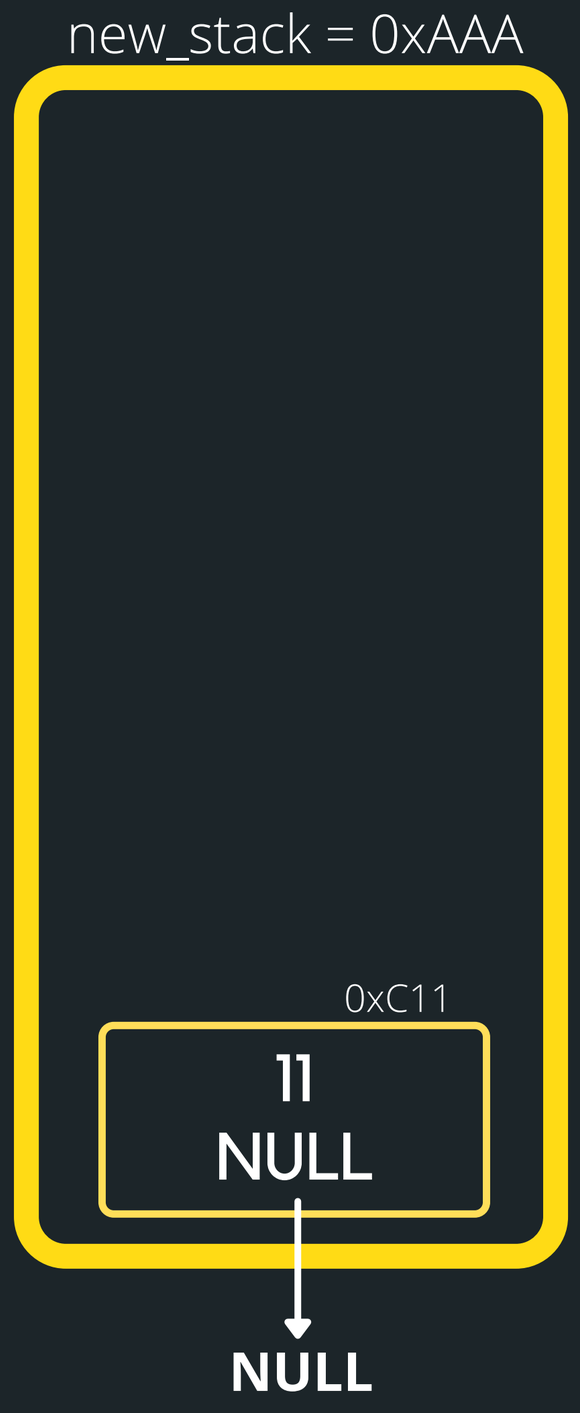
```c
#include <stdio.h>
#include "stack.h"

int main(void) {

    struct stack *new_stack = create_stack();

    push_stack(new_stack, 11);
    push_stack(new_stack, 12);
    push_stack(new_stack, 13);
    push_stack(new_stack, 14);

    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    return 0;
}
```

new_stack = 0xAAA

LET'S DO A LINKED LIST IMPLEMENTATION FIRST

STACK

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

int main(void) {

    struct stack *new_stack = create_stack();

    push_stack(new_stack, 11);
    push_stack(new_stack, 12);
    push_stack(new_stack, 13);
    push_stack(new_stack, 14);

    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    return 0;
}
```



LET'S DO A LINKED LIST IMPLEMENTATION FIRST

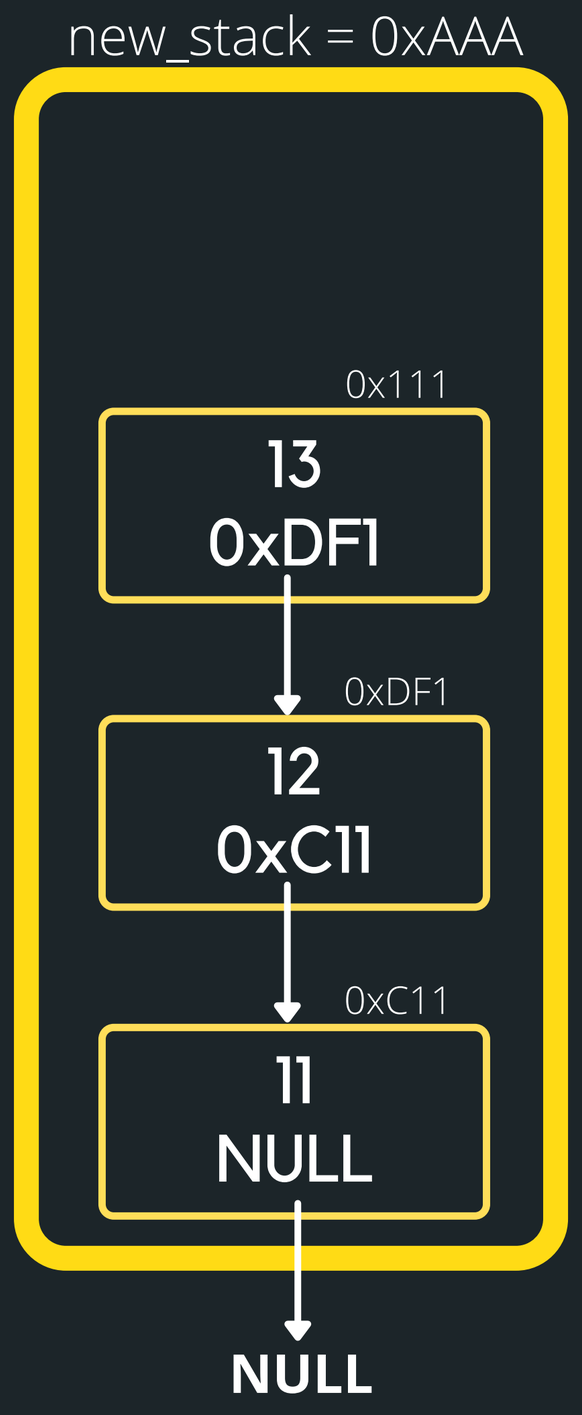
STACK

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

int main(void) {

    struct stack *new_stack = create_stack();

    push_stack(new_stack, 11);
    push_stack(new_stack, 12);
    push_stack(new_stack, 13);
    push_stack(new_stack, 14);

    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    return 0;
}
```

LET'S DO A LINKED LIST IMPLEMENTATION FIRST

STACK

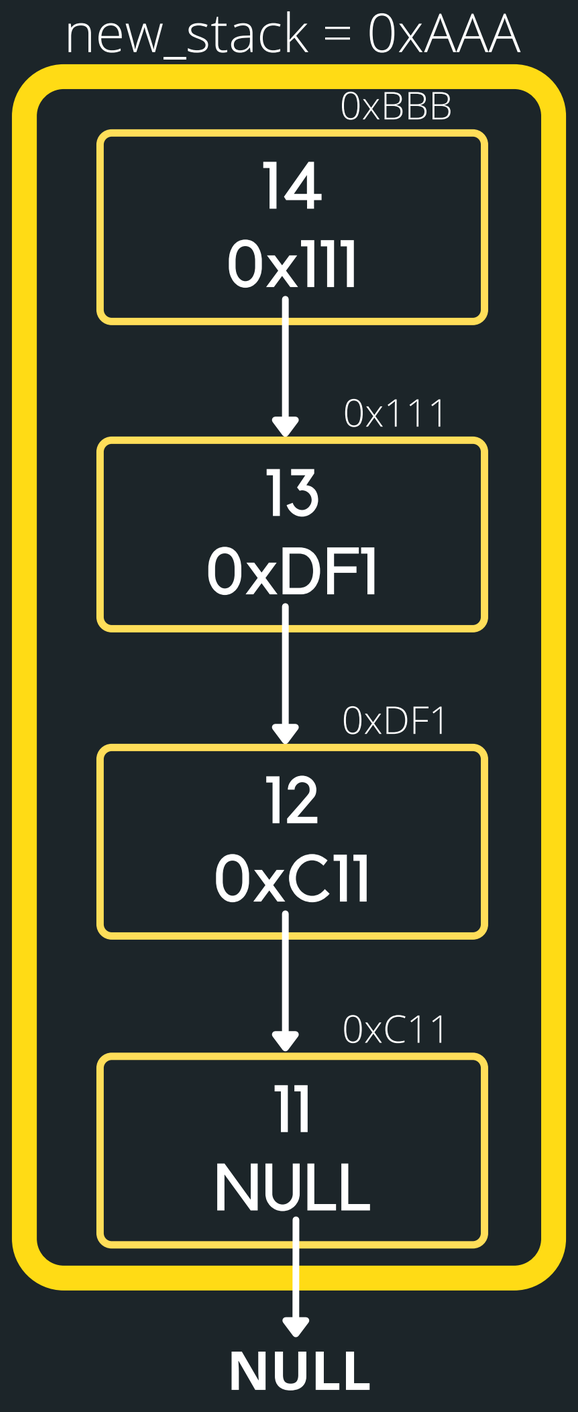
```c
#include <stdio.h>
#include "stack.h"

int main(void) {

    struct stack *new_stack = create_stack();

    push_stack(new_stack, 11);
    push_stack(new_stack, 12);
    push_stack(new_stack, 13);
    push_stack(new_stack, 14);

    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    return 0;
}
```



LET'S DO A LINKED LIST IMPLEMENTATION FIRST

STACK

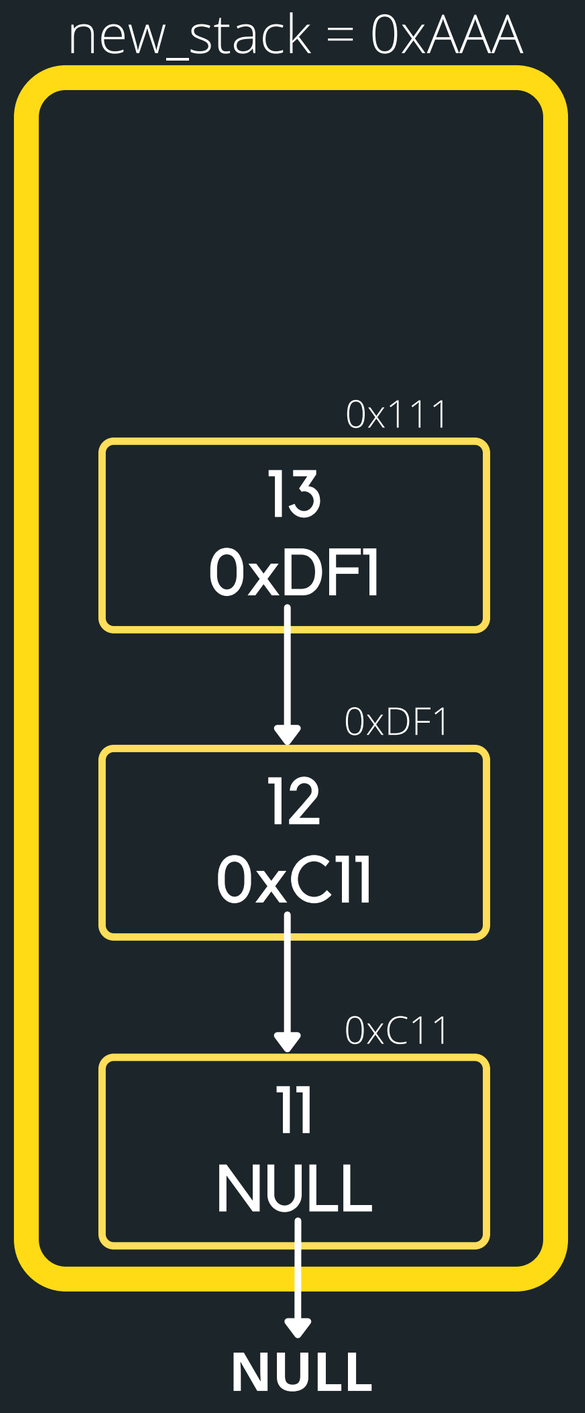
```c
#include <stdio.h>
#include "stack.h"

int main(void) {

    struct stack *new_stack = create_stack();

    push_stack(new_stack, 11);
    push_stack(new_stack, 12);
    push_stack(new_stack, 13);
    push_stack(new_stack, 14);

    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    return 0;
}
```



LET'S DO A LINKED LIST IMPLEMENTATION FIRST

STACK

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

int main(void) {

    struct stack *new_stack = create_stack();

    push_stack(new_stack, 11);
    push_stack(new_stack, 12);
    push_stack(new_stack, 13);
    push_stack(new_stack, 14);

    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    return 0;
}
```



LET'S DO A LINKED LIST IMPLEMENTATION FIRST

STACK

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

int main(void) {

    struct stack *new_stack = create_stack();

    push_stack(new_stack, 11);
    push_stack(new_stack, 12);
    push_stack(new_stack, 13);
    push_stack(new_stack, 14);

    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    return 0;
}
```

BREAK TIME...

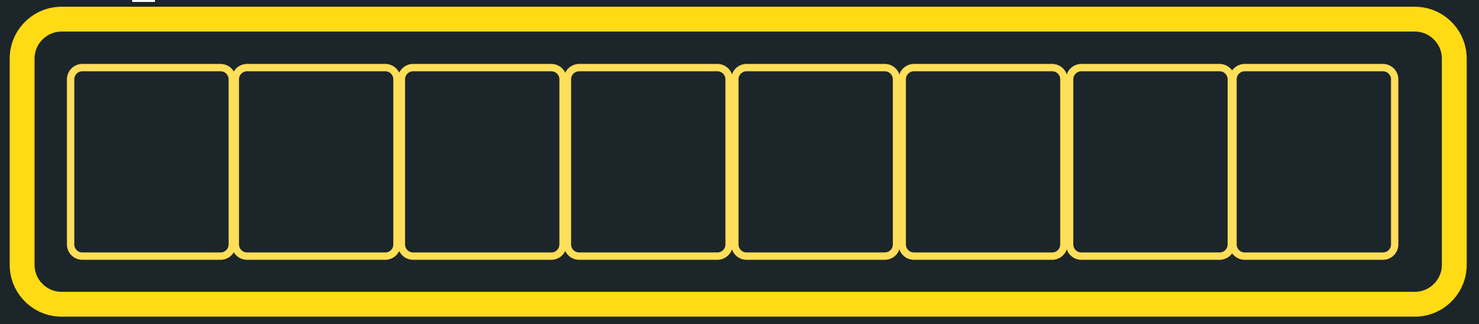
We would like to find the three fastest horses from a group of 25. We have no stopwatch and our race track has only 5 lanes. No more than 5 horses can be raced at once. How many races are necessary to evaluate the 3 fastest horses?

WHAT ABOUT AN ARRAY IMPLEMENTATION?

STACK

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

int main(void) {

    struct stack *new_stack = create_stack();

    push_stack(new_stack, 11);
    push_stack(new_stack, 12);
    push_stack(new_stack, 13);
    push_stack(new_stack, 14);

    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    return 0;
}
```

new_stack = 0xCCC

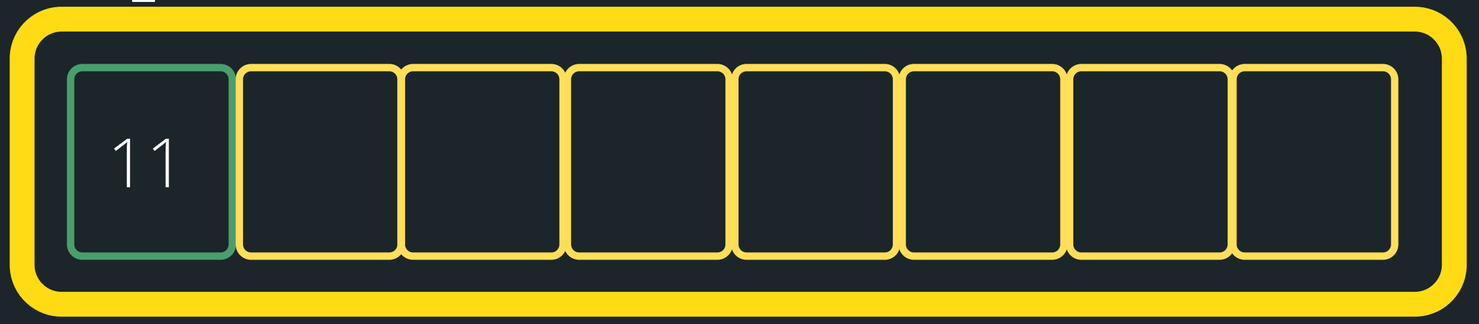


WHAT ABOUT AN ARRAY IMPLEMENTATION?

STACK

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

int main(void) {

    struct stack *new_stack = create_stack();

    push_stack(new_stack, 11);
    push_stack(new_stack, 12);
    push_stack(new_stack, 13);
    push_stack(new_stack, 14);

    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    return 0;
}
```

new_stack = 0xCCC

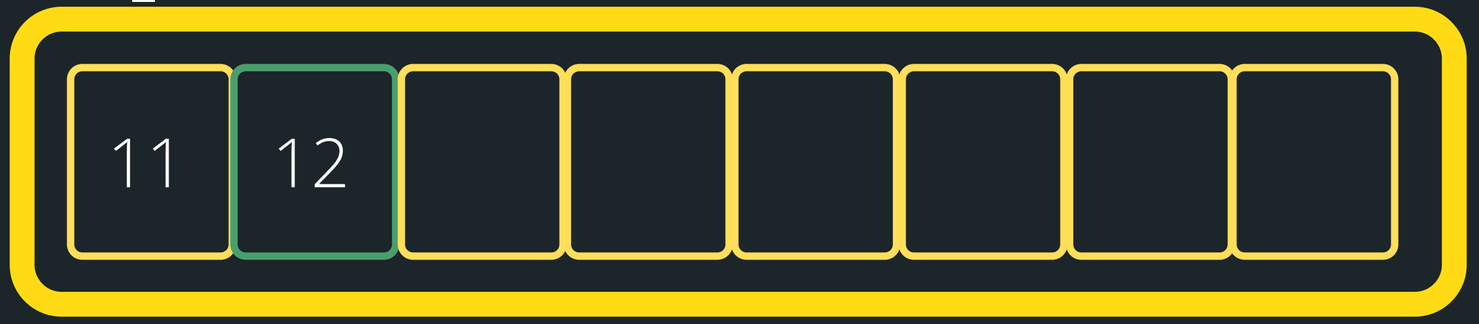


WHAT ABOUT AN ARRAY IMPLEMENTATION?

STACK

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

int main(void) {

    struct stack *new_stack = create_stack();

    push_stack(new_stack, 11);
    push_stack(new_stack, 12);
    push_stack(new_stack, 13);
    push_stack(new_stack, 14);

    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    return 0;
}
```

new_stack = 0xCCC

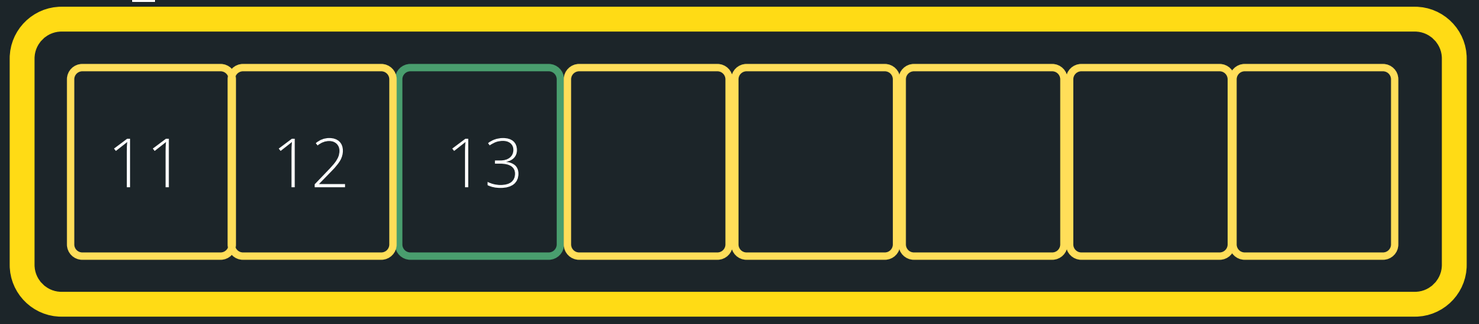


WHAT ABOUT AN ARRAY IMPLEMENTATION?

STACK

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

int main(void) {

    struct stack *new_stack = create_stack();

    push_stack(new_stack, 11);
    push_stack(new_stack, 12);
    push_stack(new_stack, 13);
    push_stack(new_stack, 14);

    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    return 0;
}
```

new_stack = 0xCCC

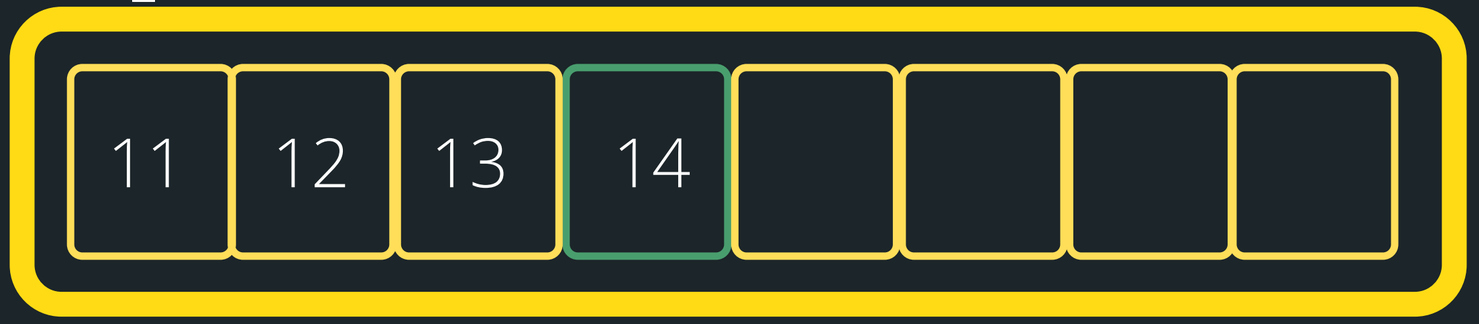


WHAT ABOUT AN ARRAY IMPLEMENTATION?

STACK

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

int main(void) {

    struct stack *new_stack = create_stack();

    push_stack(new_stack, 11);
    push_stack(new_stack, 12);
    push_stack(new_stack, 13);
    push_stack(new_stack, 14);

    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    return 0;
}
```

new_stack = 0xCCC



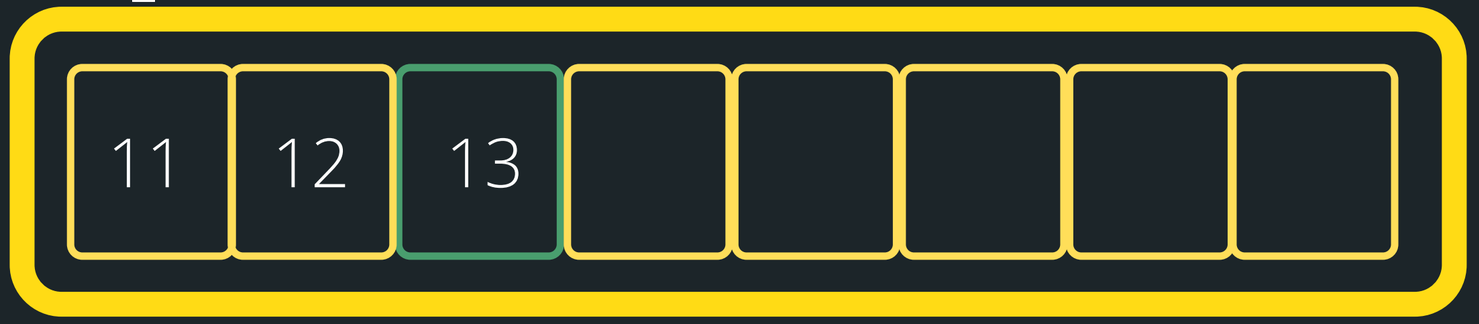
WHAT ABOUT AN ARRAY IMPLEMENTATION?

STACK

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

int main(void) {

    struct stack *new_stack = create_stack();

    push_stack(new_stack, 11);
    push_stack(new_stack, 12);
    push_stack(new_stack, 13);
    push_stack(new_stack, 14);

    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    return 0;
}
```

14

new_stack = 0xCCC

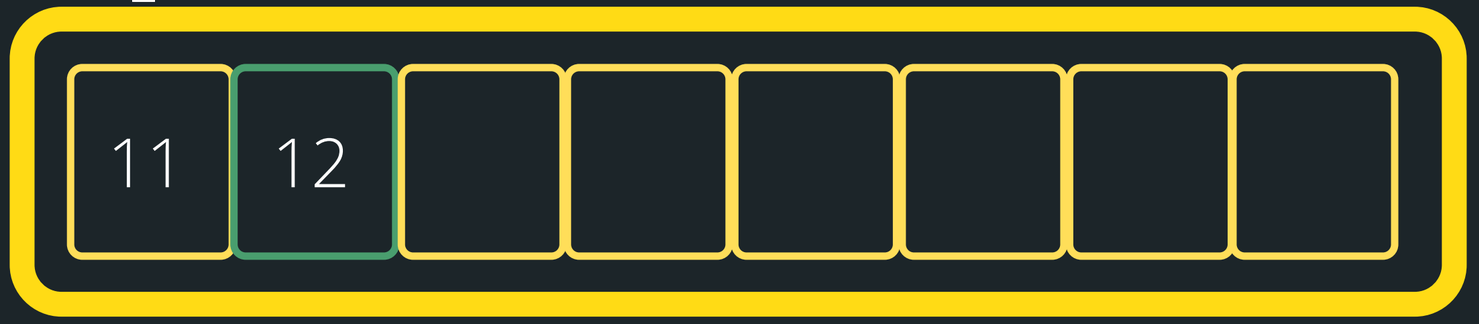


WHAT ABOUT AN ARRAY IMPLEMENTATION?

STACK

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

int main(void) {

    struct stack *new_stack = create_stack();

    push_stack(new_stack, 11);
    push_stack(new_stack, 12);
    push_stack(new_stack, 13);
    push_stack(new_stack, 14);

    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    printf("Popping the top of the stack - %d\n", pop_stack(new_stack));
    print_stack(new_stack);

    return 0;
}
```

13



OTHER ABSTRACT DATA TYPES

QUEUES

- There other abstract data types,
  - one that works in the opposite way to a Stack is a Queue
- A queue works just like a physical queue at the shops (or when you line up to get some great tickets for a music festival)
- So a Queue operates on First In, First Out principle - if you get in a queue first, you will be served first...
- To get into the queue, you enqueue, and to get out of the queue, dequeue.
- There are of course other possibilities for abstract data types!

Feedback please!

I value your feedback and use to pace the lectures and improve your overall learning experience. If you have any feedback from today's lecture, please follow the link below. Please remember to keep your feedback constructive, so I can action it and improve the learning experience.

https://www.menti.com/ym1pro5428

WHAT DID WE LEARN TODAY?

MULTI-FILE PROJECTS

maths.c

main.c

maths.h

ABSTRACT DATA TYPES

Stack:

stack.h

stack.c

main.c

REACH OUT

CONTENT RELATED QUESTIONS

Check out the forum

ADMIN QUESTIONS

cs1511@cse.unsw.edu.au