COMP1511 PROGRAMMING FUNDAMENTALS

LECTURE 10

Pointers

Memory

LAST TIME…

- For those interested, I did find churros and ate many

TODAY…

- Multi-file projects
- Pointers pointers pointers
- Memory and dynamic memory

WHERE IS THE CODE?

Live lecture code can be found here:

HTTPS://CGI.CSE.UNSW.EDU.AU/~CS1511/25T1/CODE/WEEK_5/

POINTERS

👏👏👏

- A pointer is another variable that stores a memory address of a variable
- This is very powerful, as it means you can modify things at the source (this also has certain implications for functions which we will look at in a bit)
- To declare a pointer, you specify what type the pointer points to with an asterisk:

```
type_pointing_to *name_of_ variable;
```

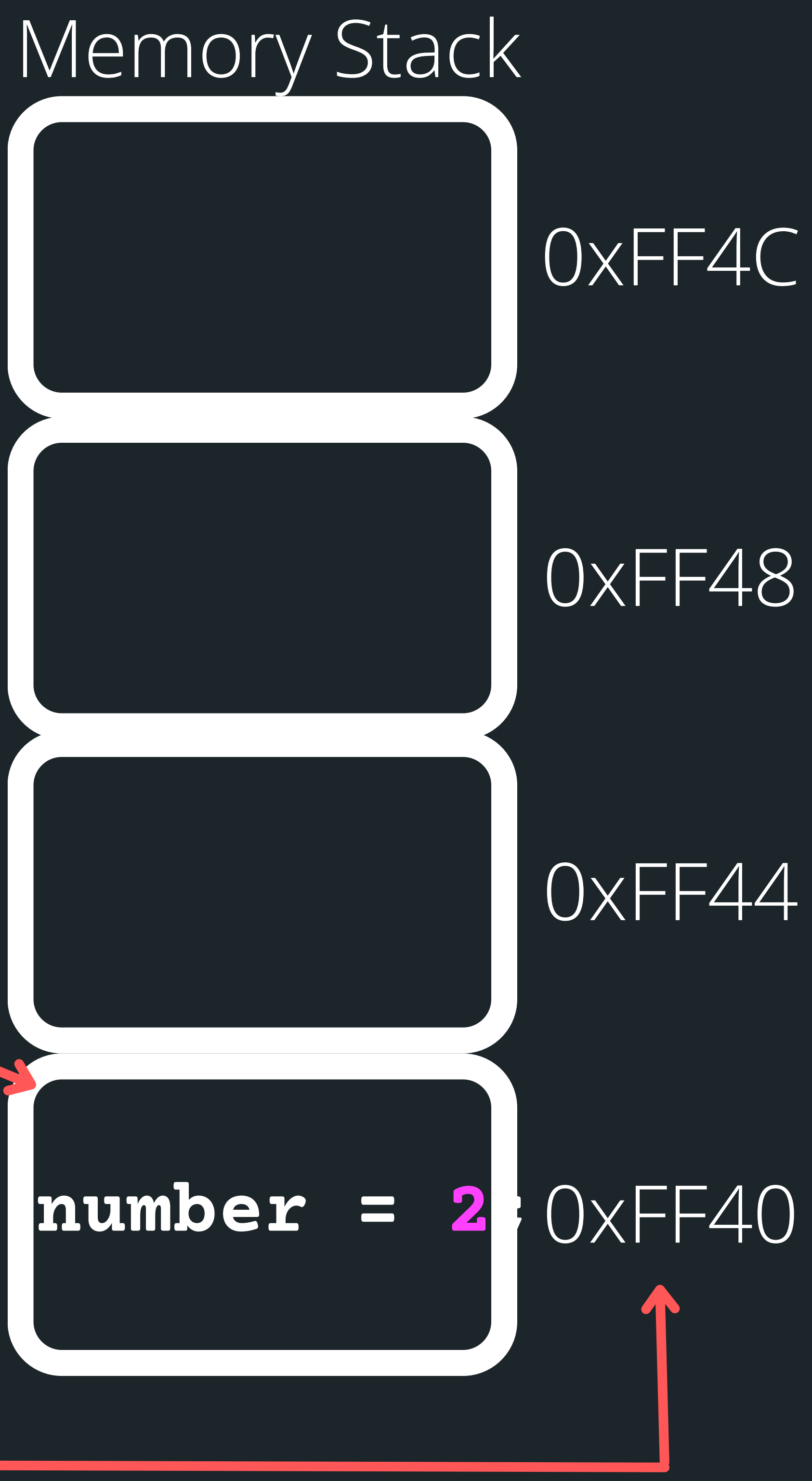
  - For example, if your pointer points to an int:

    ```
    int *pointer;
    ```

VISUALLY WHAT IS HAPPENING?

```
// Declare a variable of
// type int. called number
// Assign the value 13 to
// box
int number = 2;

// Declare a pointer
// variable that points to
// an int and assign the
// address of number to it
int *number_ptr = &number;
```




```
// So now:
number = 13
AND
number_ptr = 0xFF40
```

POINTERS

1) Declare a pointer with a * - this is where you will specify what type the pointer points to. For example, a pointer that stores the address of an int type variable:

```
int *number_ptr;
```

2) Initialise a pointer - assign the address to the variable with &

```
number_ptr = &number;
```

3) Dereference a pointer - using a * , go to the address that this pointer variable is assigned and find what is at that address

```
*number_ptr
```

POINTERS

THERE ARE THREE PARTS TO A POINTER

1. *Declare a pointer with a * - this is where you will specify what type the pointer points to*

2. *Initialise a pointer - assign the address to the variable with &*

```
#include <stdio.h>

int main (void) {

    //Declare a variable of type int, called box.
    //Assign value 6 to box
    int box = 6;
    //Declare a pointer variable that points to an int.
    //Assign the address of box to it
    int *box_ptr = &box;

    printf("The value of the variable 'box' located at address %p is %d\n"
    , box_ptr, *box_ptr);

    return 0;
}
```

3. *Dereference a pointer -Using a * , go to the address that this pointer variable is assigned and find what is at that address*

CODE CODE CODE

A SIMPLE POINTERS EXAMPLE

`howdy_pointer.c`

- A simple pointers example

CODE CODE CODE

ARRAYS AND POINTERS AND FUNCTIONS - LET'S BRING IT ALL TOGETHER...

`pointer_function.c`

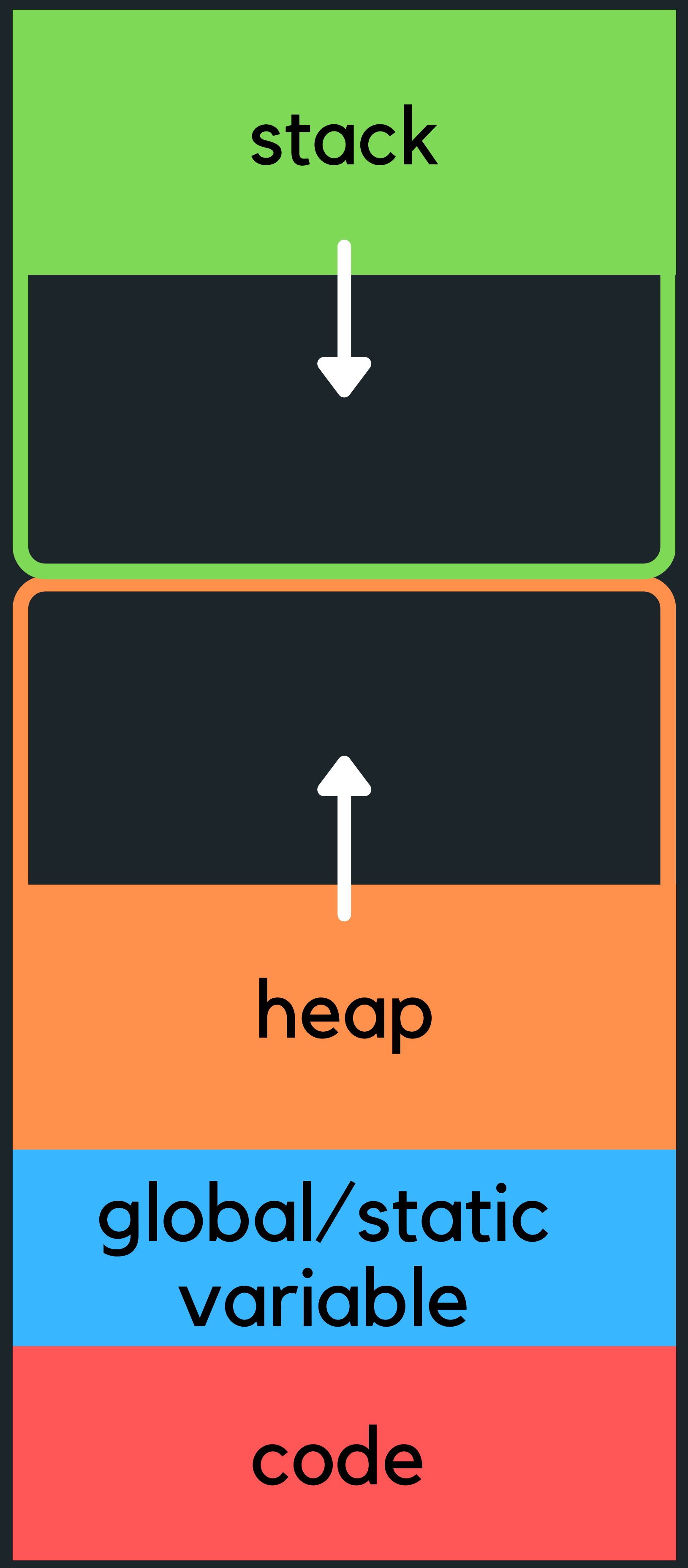
- Let’s see and use some pointers. Now remember that you can only return one thing back to main and you can’t return an array*

- The problem is this:

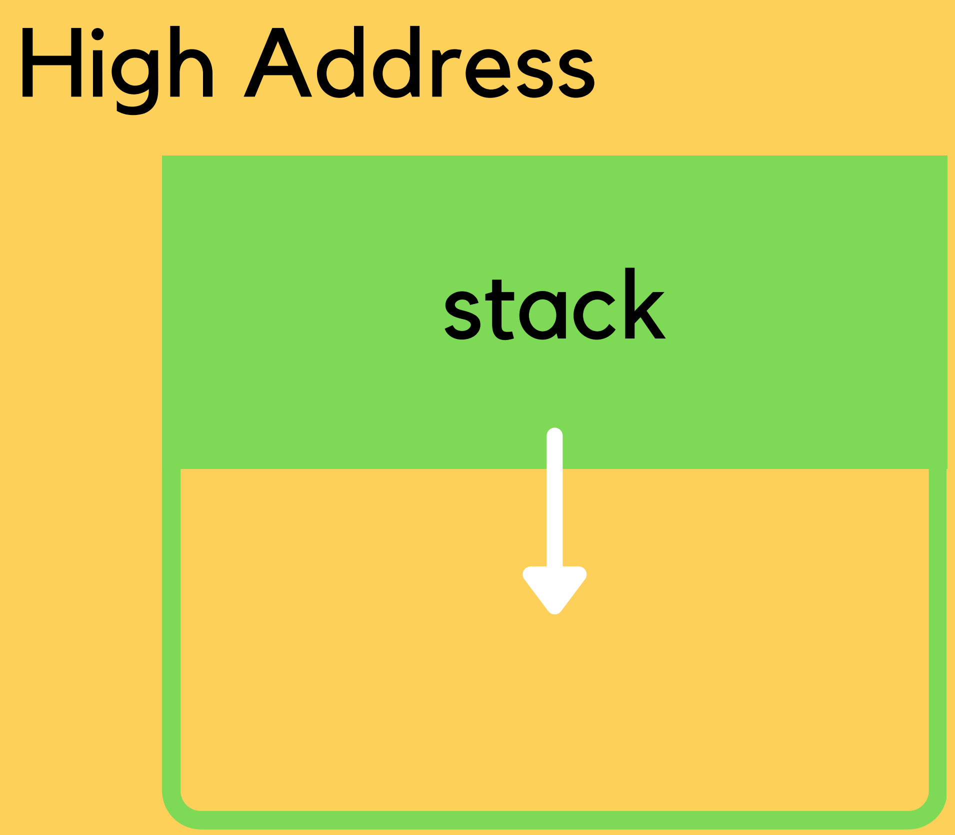
Swap two numbers in a function call

- So without using pointers, can you have a swapping function that swaps out two things? How would you return both of those things back to the main?

REVISITING MEMORY



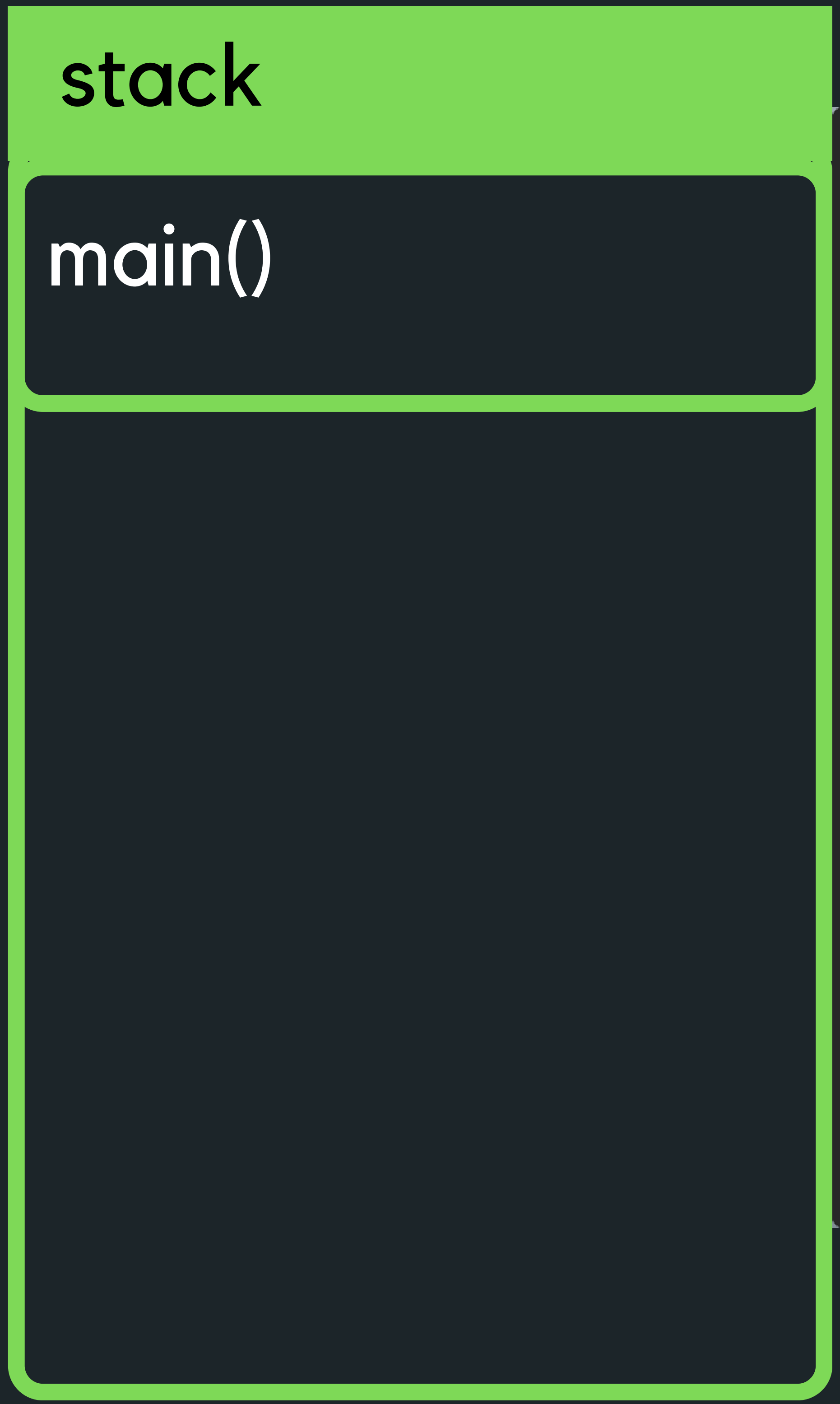
REVISITING MEMORY



- Stack memory is where relevant information about your program goes:
  - which functions are called,
  - what variables you created,
- Once your block of code finishes running {}, the function calls and variables will be removed from the stack (it's alive!)
- It means at compile time we can allocate stack memory space (not at run time)
- The stack is controlled by the program NOT BY THE developer

REVISITING MEMORY

Start running the main




```c
int sum(void) {
    int number = 10;
    number += 5;
    return number;
}

int main(void) {
    int number;
    int number2 = 5;
    int number3 = -1;
    int *number_ptr = &number2;
    int number4 = sum();
    return 0;
}
```

REVISITING MEMORY

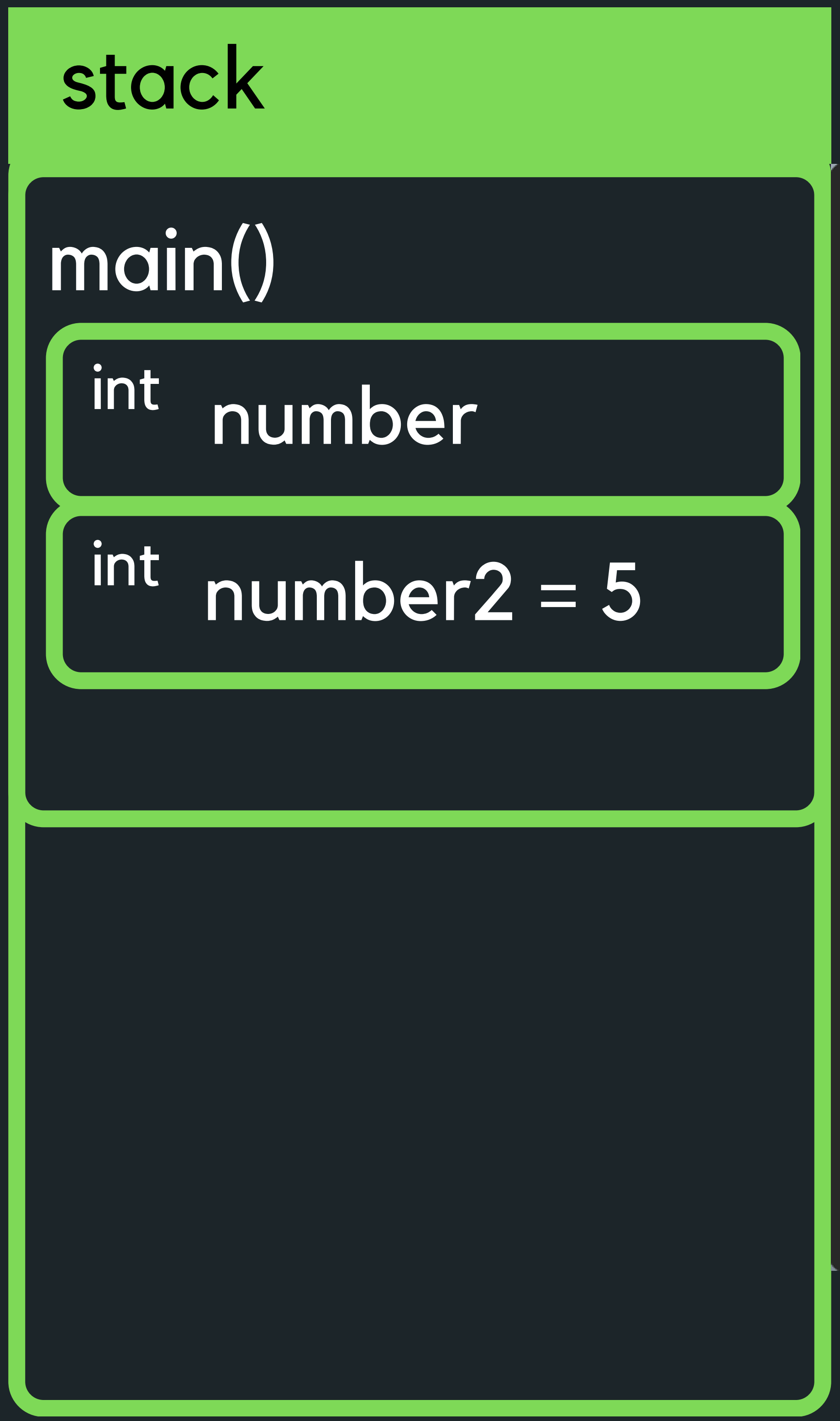
Allocate space for the variable number in main


```c
int sum(void) {
    int number = 10;
    number += 5;
    return number;
}

int main(void) {
    int number;
    int number2 = 5;
    int number3 = -1;
    int *number_ptr = &number2;
    int number4 = sum();
    return 0;
}
```

REVISITING MEMORY

Allocate space for the variable number2 in main, and assign value 5 to it

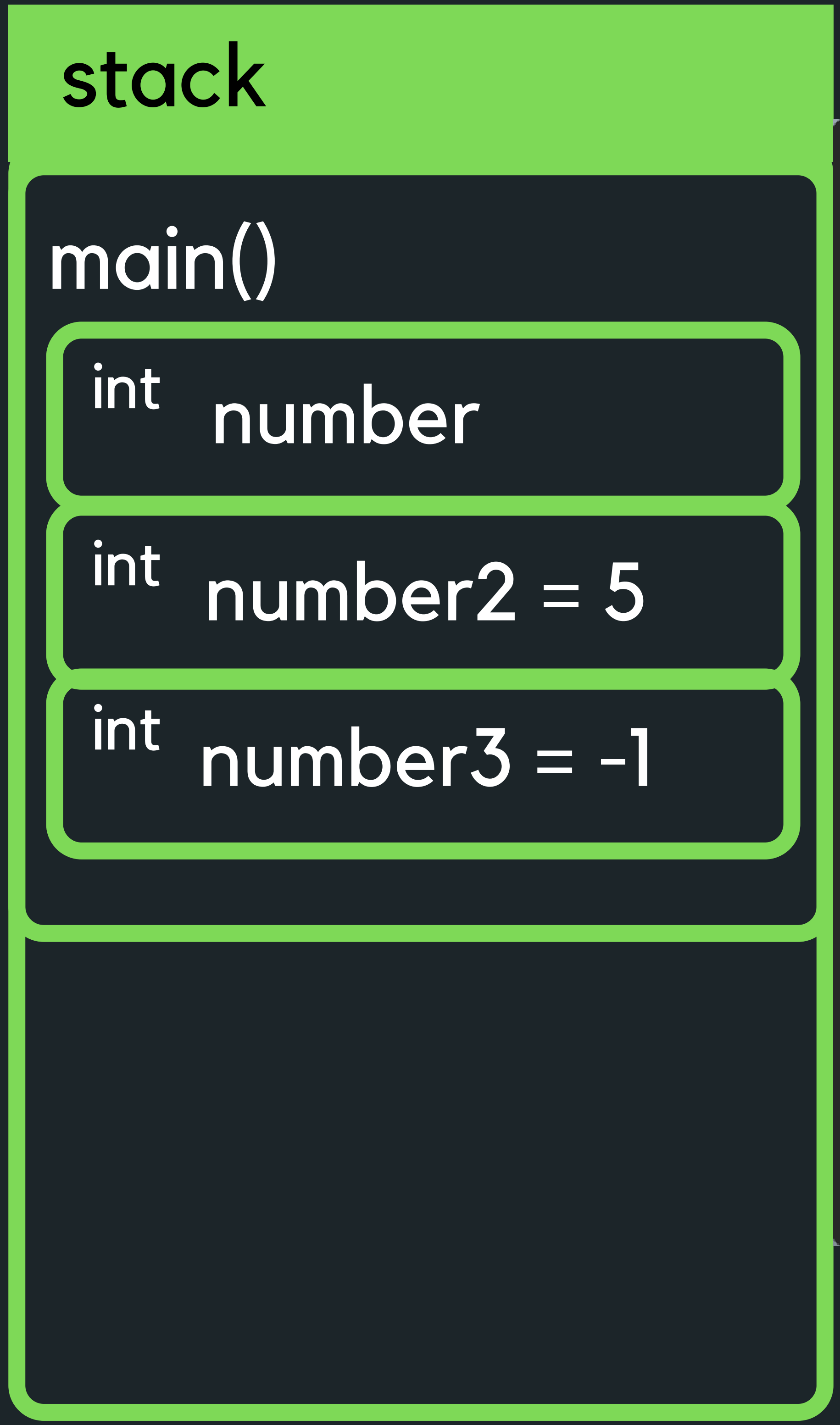



```c
int sum(void) {
    int number = 10;
    number += 5;
    return number;
}

int main(void) {
    int number;
    int number2 = 5;
    int number3 = -1;
    int *number_ptr = &number2;
    int number4 = sum();
    return 0;
}
```

REVISITING MEMORY

Allocate space for the variable number3 in main, and assign value -1 to it

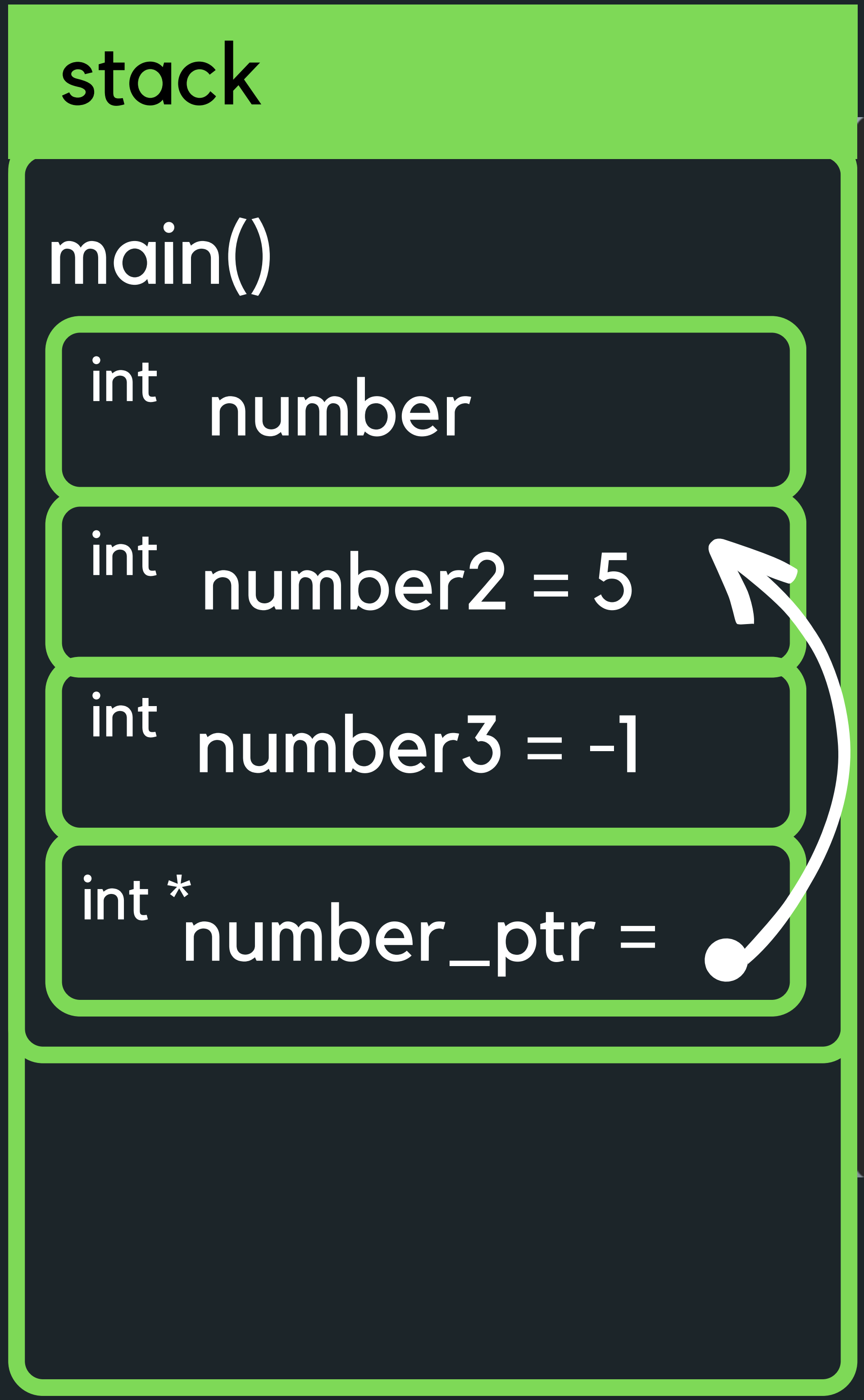



```c
int sum(void) {
    int number = 10;
    number += 5;
    return number;
}

int main(void) {
    int number;
    int number2 = 5;
    int number3 = -1;
    int *number_ptr = &number2;
    int number4 = sum();
    return 0;
}
```

REVISITING MEMORY

Allocate space for the variable number_ptr in main, and assign the address of number2 to it

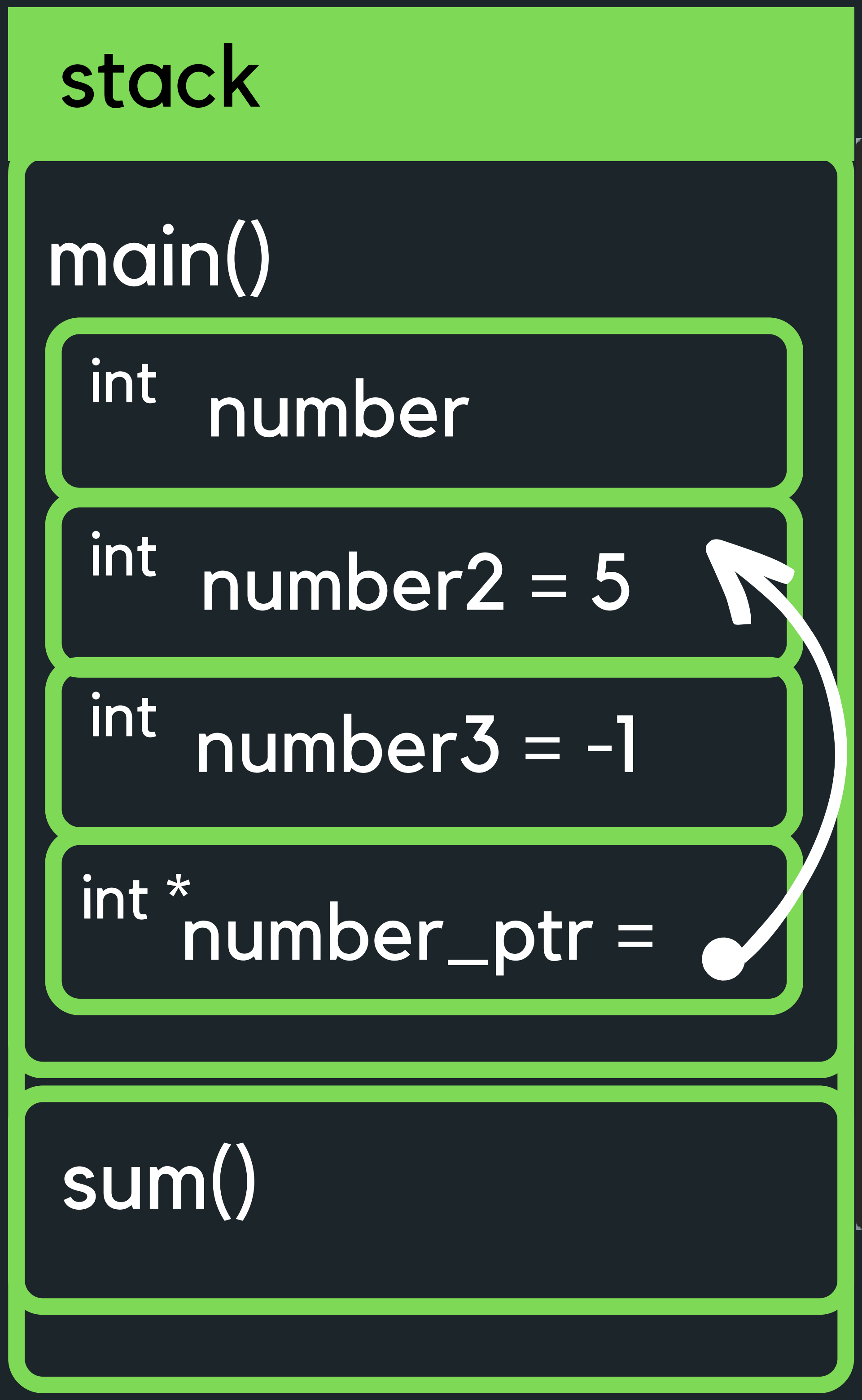



```c
int sum(void) {
    int number = 10;
    number += 5;
    return number;
}

int main(void) {
    int number;
    int number2 = 5;
    int number3 = -1;
    int *number_ptr = &number2;
    int number4 = sum();
    return 0;
}
```

REVISITING MEMORY

Call function sum() (remember we go on the right first and then assign to the left!) and allocate memory space on the stack

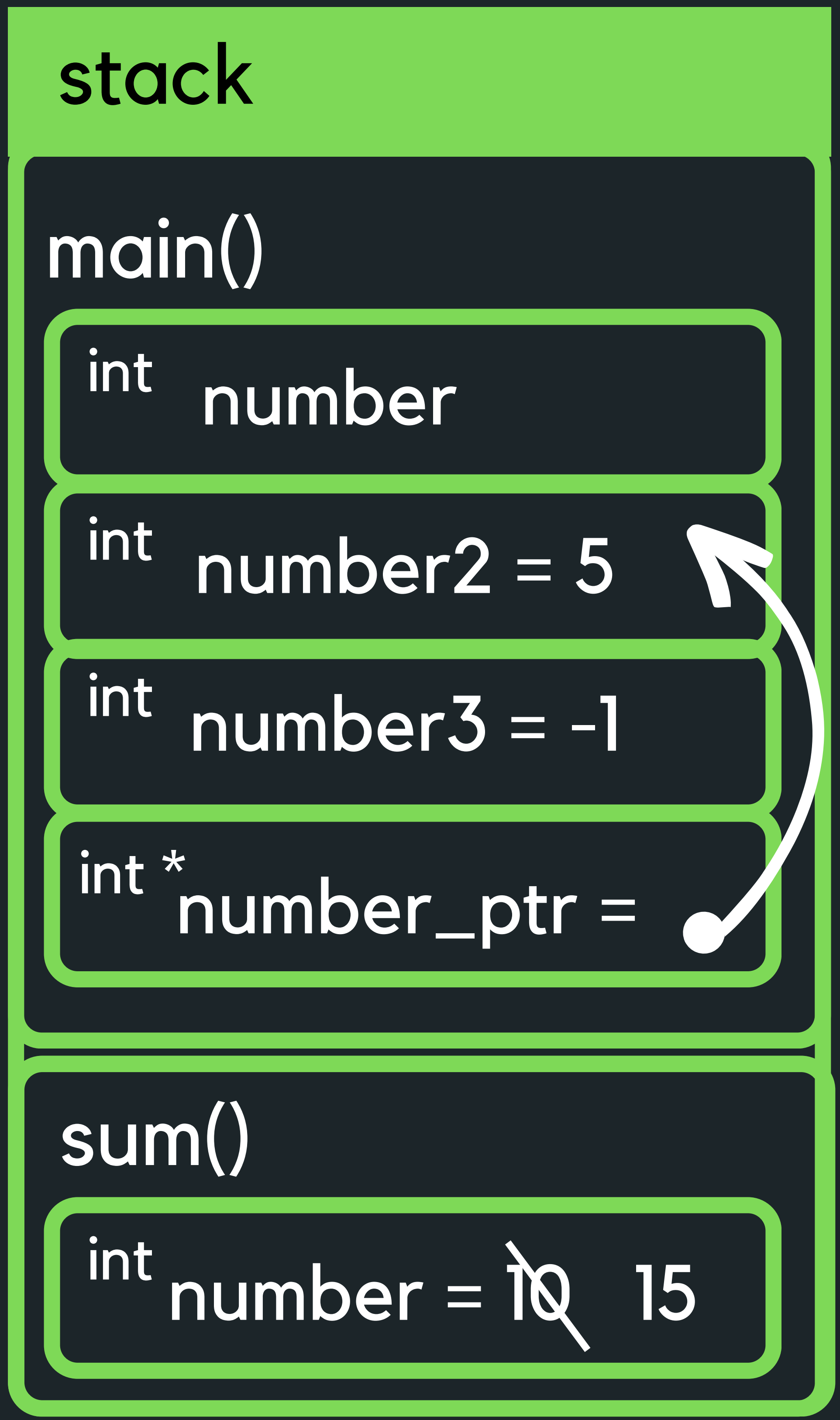



```c
int sum(void) {
    int number = 10;
    number += 5;
    return number;
}

int main(void) {
    int number;
    int number2 = 5;
    int number3 = -1;
    int *number_ptr = &number2;
    int number4 = sum();
    return 0;
}
```

REVISITING MEMORY

Allocate space for variable number in the sum function call and assign the value 10 to it. THen change the value by adding 5 to it

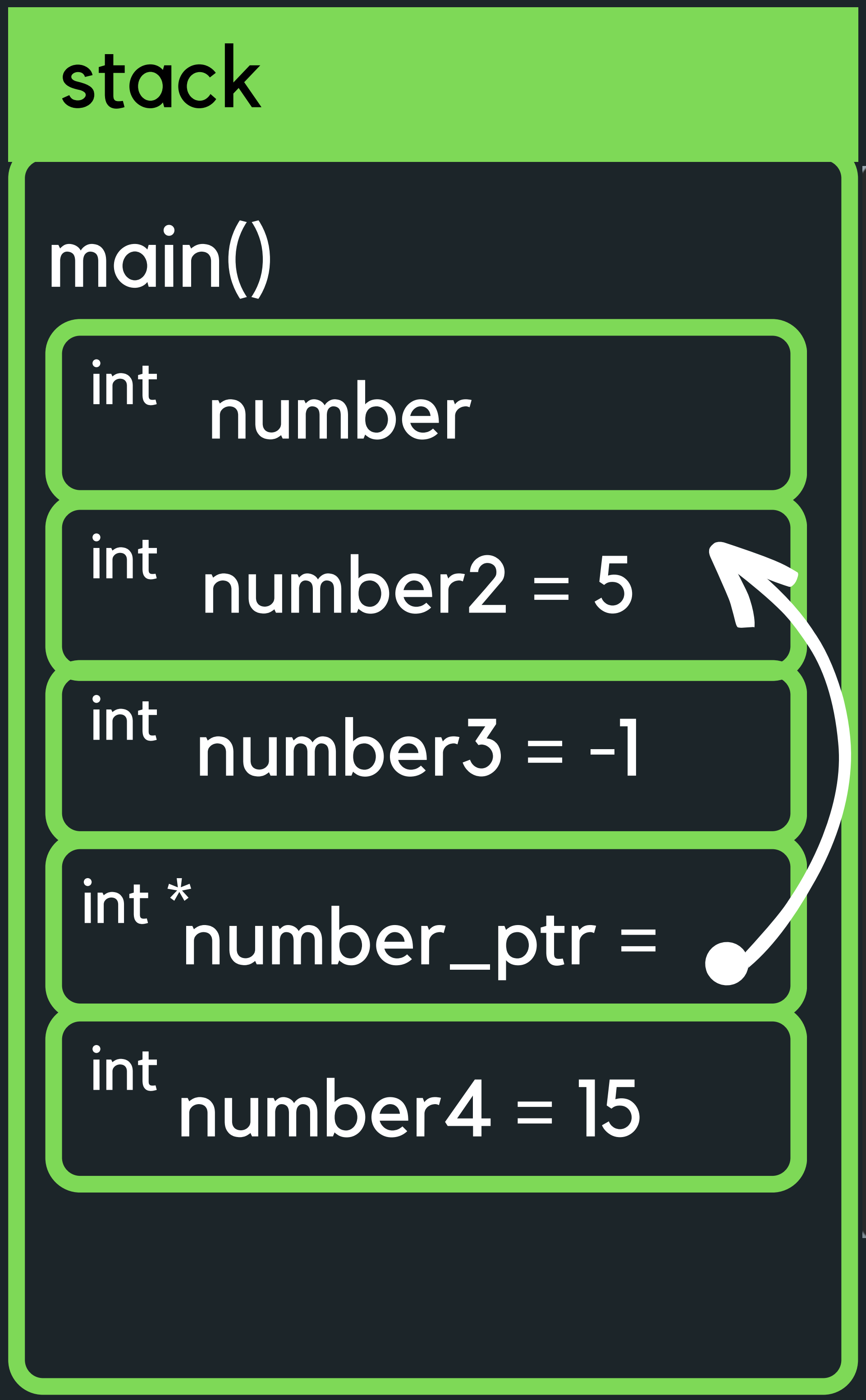



```c
int sum(void) {
    int number = 10;
    number += 5;
    return number;
}

int main(void) {
    int number;
    int number2 = 5;
    int number3 = -1;
    int *number_ptr = &number2;
    int number4 = sum();
    return 0;
}
```

REVISITING MEMORY

Deallocate the stack memory of sum() and return 15 to the main function. Allocate space for number and assign 15 to it.




```
int sum(void) {
    int number = 10;
    number += 5;
    return number;
}

int main(void) {
    int number;
    int number2 = 5;
    int number3 = -1;
    int *number_ptr = &number2;
    int number4 = sum();
    return 0;
}
```

REVISITING MEMORY

Deallocate the stack memory for main and return 0 to finish


```c
int sum(void) {
    int number = 10;
    number += 5;
    return number;
}

int main(void) {
    int number;
    int number2 = 5;
    int number3 = -1;
    int *number_ptr = &number2;
    int number4 = sum();
    return 0;
}
```

QUICK REHASH

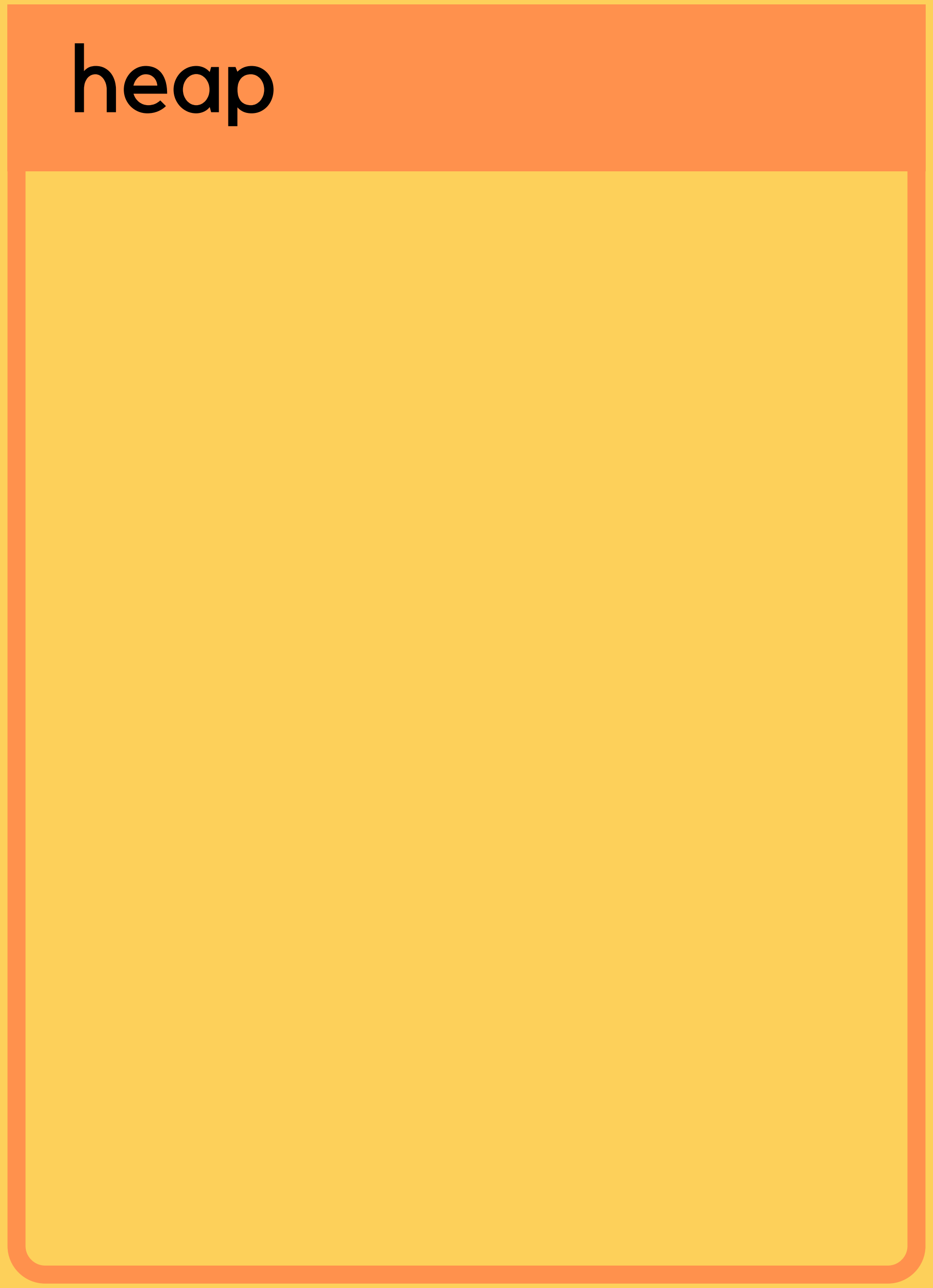
MEMORY

So far we have talked a bit about how variables are stored in memory, and live in their world {} in the stack memory

- This means that if we create data inside a function, it will die when that function finishes running
- This is memory that is allocated by the compiler at compile time…

```c
// Make an array
int *create_array(void) {
    int numbers[10] = {0};
    // Return pointer to the array
    return numbers;
}
//However, when we close the curly brakes,our
//array is killed, so we are returning a
//pointer to memory that we no longer have...
```

REVISITING MEMORY



A helper function cannot return a pointer of a stack variable! So how can be deal with this? You can return by copying it or putting it into a more permanent storage - yay the heap!

Unlike stack memory, heap memory is allocated by the programmer and won’t be deallocated until it is explicitly freed by the programmer also! You have a great power now… but with great power comes great responsibility!

BUT WHAT HAPPENS IF I WANT TO SAVE SOME MEMORY?

MALLOC()

- We do have the wonderful opportunity to allocate some memory by calling the function `malloc()` and letting this function know how many bytes of memory we want
  - this is the stuff that goes on the heap!
  - this function returns a pointer to the piece of memory we created based on the number of bytes we specified as the input to this function
  - this also allows us to dynamically create memory as we need it - neat!
  - This means that we are now in control of this memory (cue the evil laugh!)

WHAT IF I RUN WILD AND JUST KEEP ASKING FOR MEMORY?

FREE()

It would be very impolite to keep requesting memory to be made (and hog all that memory!), without giving some back…

- This piece of memory is ours to control and it is important to remember to kill it or you will eat up all the memory you computer has… slow down the machine, and often result in crashing… often called a memory leak…
- A memory leak occurs when you have dynamically allocated memory (with **`malloc()`**) that you do not free - as a result, memory is lost and can never be free causing a memory leak
- You can free memory that you have created by using the function **`free()`**

HOW DO I KNOW HOW MUCH MEMORY TO ASK FOR WHEN I USE MALLOC()

SIZEOF()

- We can use the function **`sizeof()`** to give us the exact number of bytes we need to malloc (memory allocate)

```c
// This program demonstrates how sizeof() function works
// It returns the size of a particular data type
// We use the format specified %lu with it (long unsigned)
// if we want to print out the output of sizeof()

#include <stdio.h>

int main (void) {

    int array[10] = {0};

    // Example of using the sizeof() function
    printf("The size of an int is %lu bytes\n", sizeof(int));
    printf("The size of an array of int is %lu bytes\n", sizeof(array));
    printf("The size of a 10 ints is %lu bytes\n", 10 * sizeof(int));
    printf("The size of a double is %lu bytes\n", sizeof(double));
    printf("The size of a char is %lu bytes\n", sizeof(char));

    return 0;
}

```

FORMAT

MALLOC()

- Using the **`malloc()`** function:

if you need to have space for more than one element, you multiply it by the number of elements you need

```
int *ptr = malloc(x * sizeof(int));
```

the pointer that malloc will return to indicate the start of the portion of space it has allocated

using the function

specify data type that you need

FORMAT

MALLOC()

- Using the **`malloc()`** function example

```
int *ptr = malloc(10 * sizeof(int));
```

This will create a piece of memory of 10 * 4 bytes = 40 bytes and return the address of where this memory is in ptr

PUTTING IT ALL TOGETHER:

MALLOC(SIZEOF()) FREE()

- Using all of these together in a simple example:

```c
#include <stdio.h>

// malloc() and free() are functions in the <stdlib.h> library

#include <stdlib.h>

void read_array(int *numbers, int size);
void reverse_array(int *numbers, int size);

int main (void) {
    int size;
    printf("How many numbers would you like to scan: ");
    scanf("%d", &size);

    // Allocate some memory space for my array and return a pointer
    // to the first element
    int *numbers = malloc(size * sizeof(int));

    // Check if there is actually enough space to allocate
    // memory, exit the program if there is not enough memory
    // to allocate.

    if (numbers == NULL) {
        printf("Malloc failed, not enough space to allocate memomry\n");
        return 1;
    }

    // Perform some functions here
    read_array(numbers, size);
    reverse_array(numbers, size);

    // Free the allocated memory
    // In this case, it would happen on program exit anyway
    free(numbers);

    return 0;
}
```

WHAT ARE ARRAYS?

DECAY TO POINTERS...

`array_decay.c`

- Let's see how arrays decay to a pointer...

BREAK TIME…

Jax and Juno have fallen in love (via the internet) and Jax wishes to mail her a ring. Unfortunately, they live in the country of Kleptopia where anything sent through the mail will be stolen unless it is enclosed in a padlocked box. Jax and Juno each have plenty of padlocks, but none to which the other has a key. How can Jax get the ring safely into Juno's hands?

STRUCTS AND POINTERS

-> VERSUS .

- Remember that when we access members of a struct we use a .

```c
#include <stdio.h>
#include <string.h>

#define MAX 15

// 1. Define struct
struct dog {
    char name[MAX];
    int age;
};

int main (void) {
    // 2. Declare struct
    struct dog jax;

    // 3. Initialise struct (access memebers with .)
    // Remember we can't just do jax.name = "Jax"
    // So can use the function strcpy() in <string.h>
    // to copy the string over

    strcpy(jax.name, "Jax");
    jax.age = 6;

    printf("%s is an awesome dog, who is %d years old\n", jax.name, jax.age);
    return 0;
}
```

STRUCTS AND POINTERS

-> VERSUS .

- What happens if we make a pointer of type struct? How do we access it then?

```c
#include <stdio.h>
#include <string.h>

#define MAX 15

// 1. Define struct
struct dog {
    char name[MAX];
    int age;
};

int main (void) {
    // 2. Declare struct
    struct dog jax;

    // Have a pointer to the variable jax of type struct dog
    struct dog *jax_ptr = &jax;

    // How would we initialise it using the pointer?
    // Perhaps dereference the pointer and access the member?

    strcpy((*jax_ptr).name, "Jax");
    (*jax_ptr).age = 6;

    printf("%s is an awesome dog, who is %d years old\n", (*jax_ptr).name, (*jax_ptr).age);
    return 0;
}

```

STRUCTS AND POINTERS

-> VERSUS .

- Those brackets can get quite confusing, so there is a shorthand way to do this with an ->
- There is no need to use (*jax_ptr) and instead can just straight jax_ptr ->

```c

// INSTEAD OF THIS:
//strcpy((*jax_ptr).name, "Jax");
//(*jax_ptr).age = 6;
//printf("%s is an awesome dog, who is %d years old\n", (*jax_ptr).name, (*jax_ptr).age);
// DO THIS:
strcpy(jax_ptr->name, "Jax");
jax_ptr->age = 6;

printf("%s is an awesome dog, who is %d years old\n", jax_ptr->name, jax_ptr->age);

```

WHY ARE YOU HURTING US WITH ALL THIS STUFF?

WE HAVE COME TO THE ULTIMATE REVEAL.

- Now that you have become comfortable with arrays, we are going to become acquainted with another important data structure (drum roll please 🥁):

- The one and only LINKED LIST 🥳

MULTI-FILE PROJECTS

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

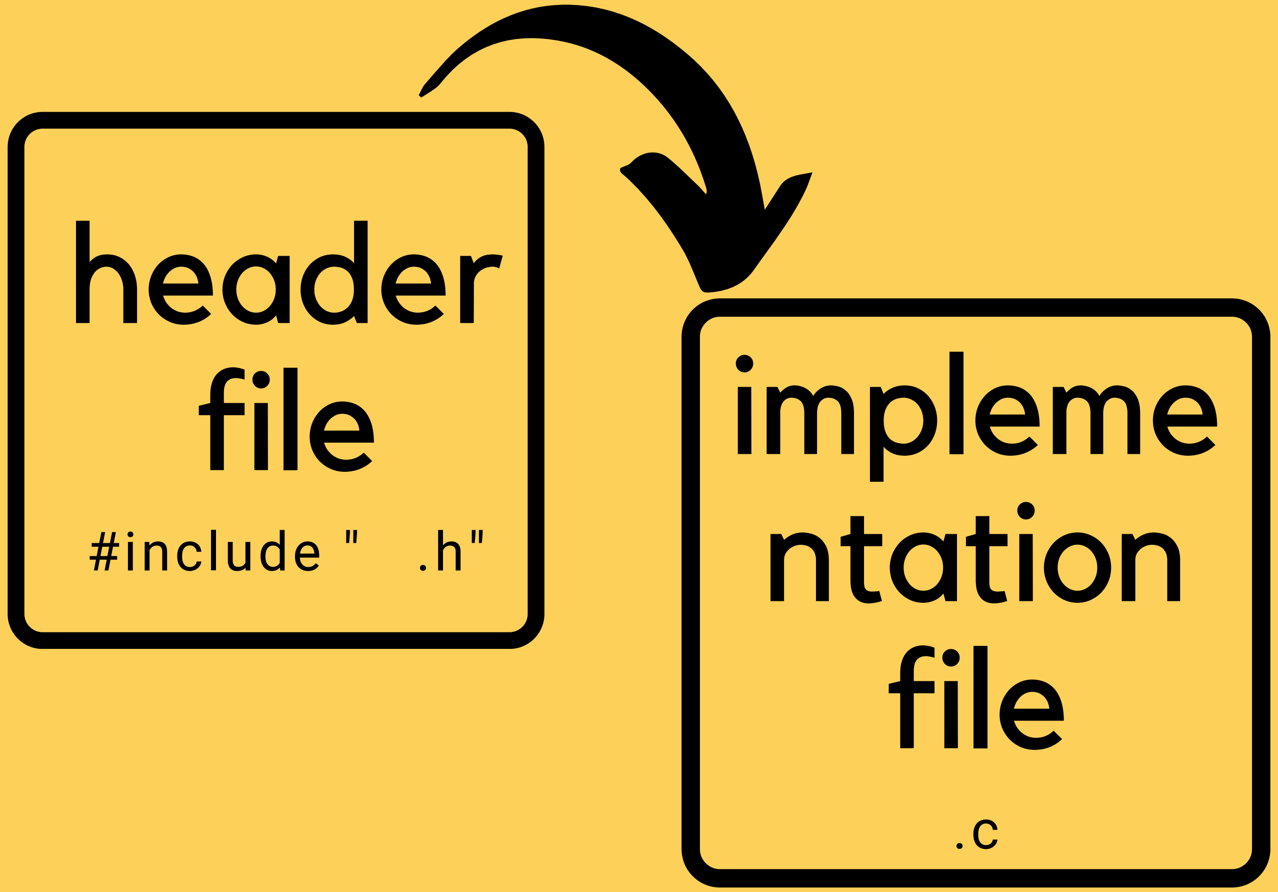
MULTI-FILE PROJECTS

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 PROJECTS

.H AND .C



- 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

MULTI-FILE PROJECTS

.H
HEADER FILE

header file

#include "   .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

MULTI-FILE PROJECTS

.C IMPLEMENTATION

implementation file

.c

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

MULTI-FILE PROJECTS

MAIN.C

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

MULTI-FILE PROJECTS

AN EXAMPLE

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

MULTI-FILE PROJECTS

AN EXAMPLE HEADER FILE

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

#define PI 3.14

// Function prototype for a function that
// calculate the square of a number:
int square(int number);

// Function prototype that calculates the sum of
// of two numbers
int sum(int number_one, int number_two);
```

MULTI-FILE PROJECTS

AN EXAMPLE IMPLEMENTATION FILE (NOTE TO INCLUDE THE HEADER THAT WE DEFINED!

```c
// This is the implementation file of maths.h
// We defined two functions in the header file (.h)
// and this is where we actually implement them

// 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 number_one, int number_two) {
    return number_one + number_two;
}
```

MULTI-FILE PROJECTS

AN EXAMPLE OF MAIN THAT DRIVES OUR PROGRAM

```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 our header file also
#include "maths.h"

int main(void) {
    int number_one = 13;
    int number_two = 10;

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

MULTI-FILE PROJECTS

COMPILING

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):

```
Terminal
File Edit View Terminal Tabs Help
avas605@vx3:~/maths_module$ dcc maths.c main.c -o maths
avas605@vx3:~/maths_module$ ./maths
The square of the number 13 is 169
The sum of 13 and 10 is 23
avas605@vx3:~/maths_module$
```

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

INTRODUCING A NEW DATA STRUCTURE

LINKED LISTS

- Like an array, a linked list is used to store a collection of the same data type
- So what's the point?
  - Linked lists are dynamically sized, that means we can grow and shrink them as needed - efficient for memory!
  - Elements of a linked list (called nodes) do NOT need to be stored contiguously in memory, like an array.
  - Unlike arrays, linked lists are not random access data structures! You can only access items sequentially, starting from the beginning of the list.

HAVE A RESTFUL FLEX WEEK!

- We hope that you all have a good rest and catch up over the Flex Week time.
  - There are no formal classes next week!
- Help Sessions are still running, please check the timetable
- Forum will be monitored closely to help you with any Assignment 1 queries

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://forms.office.com/r/SdwfGte8MK

WHAT DID WE LEARN TODAY?

POINTERS

howdy_pointer.c

array_magic.c

pointer_functions.c

pointer_struct.c

MEMORY

sizeof_demo.c

malloc.c

memory_fun.c

REACH OUT

CONTENT RELATED QUESTIONS

Check out the forum

ADMIN QUESTIONS

cs1511@unsw.edu.au