# COMP1511/1911 Programming Fundamentals

Week 7
Lecture 1

# Previously On...

---

- Angela was here
- Introduction to pointers

# Today

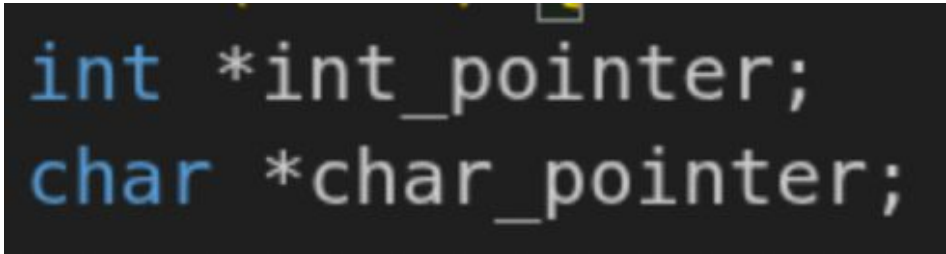
---

- Recap pointers
- Pointers with structs
- Malloc and the heap
- Multi-file projects

# Pointers

---

- Variables that store the memory address of a variable
  - Very powerful
  - Means we can modify things at their source
- To declare a pointer, you specify what type you're pointing to with a *



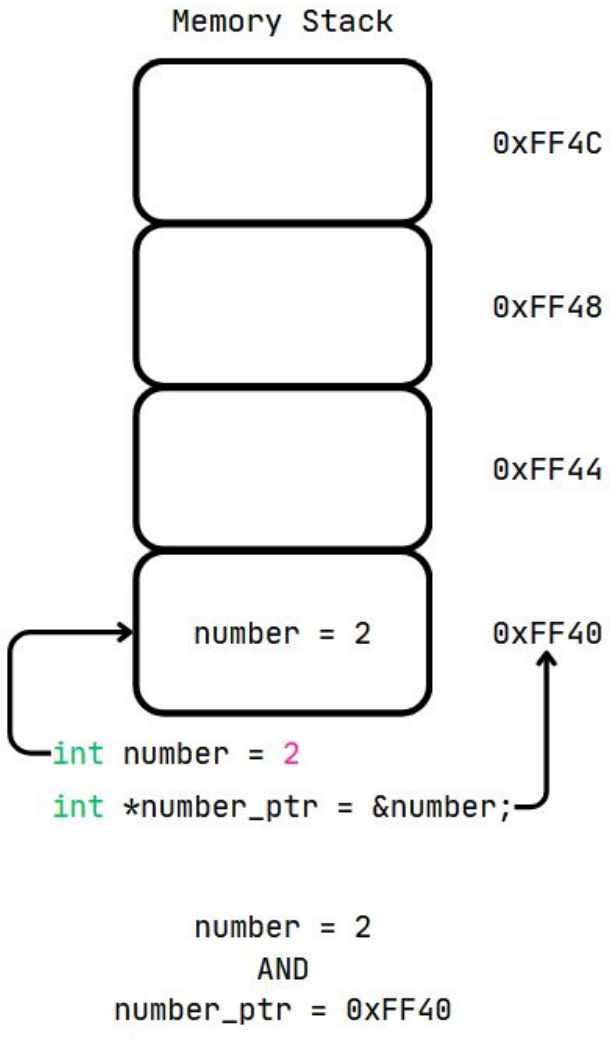
# Visually What is Happening?

---

- We are storing the memory address
  - Where the variable is stored in memory

```c
#include <stdio.h>

int main(void) {
    int number = 2;
    int *number_ptr = &number;
    printf("number = %d and number_ptr = %p\n", number, number_ptr);
    return 0;
}
```



# Pointers

1. Declare a pointer with *
    - This specifies what type the pointer points to
    - Pointer that stores the address of an int type variable?
        - int *number_ptr;
2. Initialise a pointer
    - Assign the address of a variable to your new pointer variables
        - number_ptr = &number;
3. Dereference the pointer
    - Using a *
        - Go to the address that the pointer variable is assigned, and find what is stored there
            - *number_ptr

# Putting it Together

```c
#include <stdio.h>

int main(void) {
    int *box_ptr;
    int box = 12;
    box_ptr = &box;
    printf("The variable box is stored at %p and has a value of %d\n", box_ptr, *box_ptr);
    return 0;
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
z3548950@vx16:~/public_html/COMP151126T2/code/week5$ ./prog
The variable box is stored at 0x7ffedf74d3c0 and has a value of 12
```

# Let’s Play Around

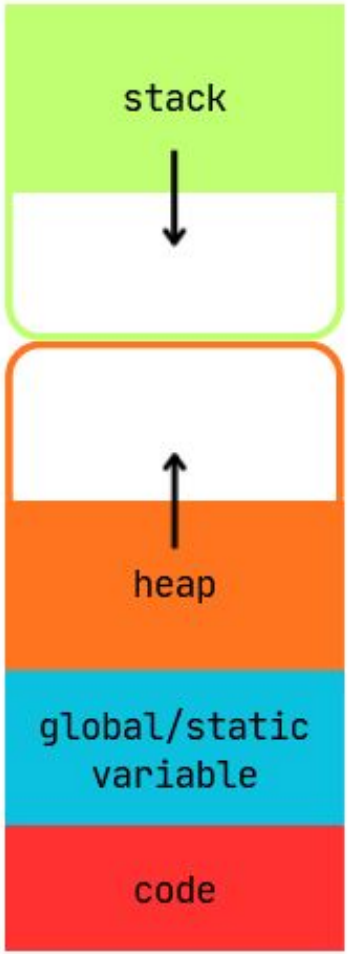
---

- Code is both swag and epic

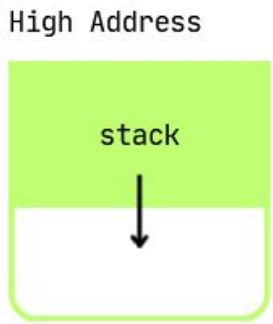
# Revisiting Memory

---

- Code
  - Your program's instructions
- Global / Static
  - Variables that exist for the entire program
- Heap
  - Dynamically allocated memory (malloc, calloc, realloc, free).
- Stack
  - Function calls, parameters, and local variables



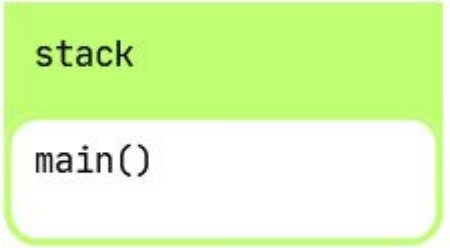
# Stack Memory




- Where relevant information about your program goes
  - Which functions are called
  - What variables are created
- Once your block of code is finished running ({}), the function calls and variables will be removed from the stack
- It means at compile time, we can allocate stack memory space
  - Not at run time
- The stack is controlled by the program
  - NOT THE DEVELOPER

# Stack Example

- Starting with the main

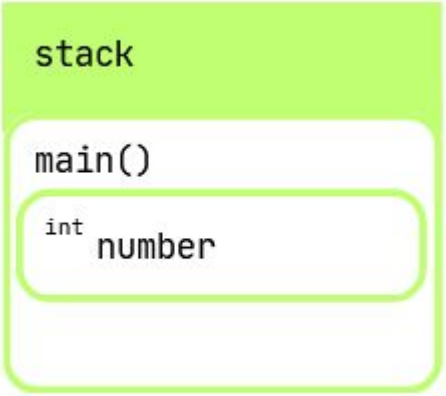



```c
int sum(void) {
    int number = 10;
    number += 5;
    return number;
}

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

# Stack Example

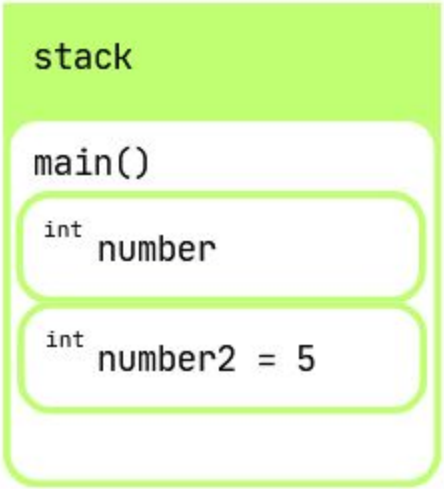
- Allocate space for the variable number in main




```
1 int sum(void) {
2     int number = 10;
3     number += 5;
4     return number;
5 }
6
7 int main(void) {
8     int number;
9     int number2 = 5;
10    int number3 = -1;
11    int *number_ptr = &number2;
12    int number4 = sum();
13    return 0;
14 }
```

# Stack Example

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




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

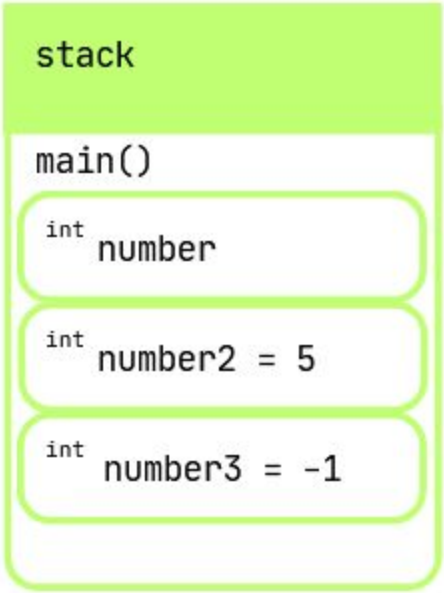
int main(void) {
    int number;
    int number2 = 5;
    int number3 = -1;
    int *number_ptr = &number2;
    int number4 = sum();
    return 0;
}
```

# Stack Example

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

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

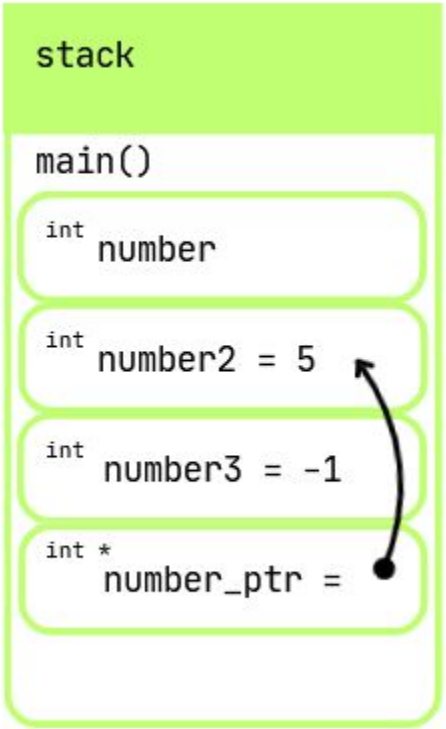
int main(void) {
    int number;
    int number2 = 5;
    int number3 = -1;
    int *number_ptr = &number2;
    int number4 = sum();
    return 0;
}
```



# Stack Example

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

```
1 int sum(void) {
2     int number = 10;
3     number += 5;
4     return number;
5 }
6
7 int main(void) {
8     int number;
9     int number2 = 5;
10    int number3 = -1;
11    int *number_ptr = &number2;
12    int number4 = sum();
13    return 0;
14 }
```

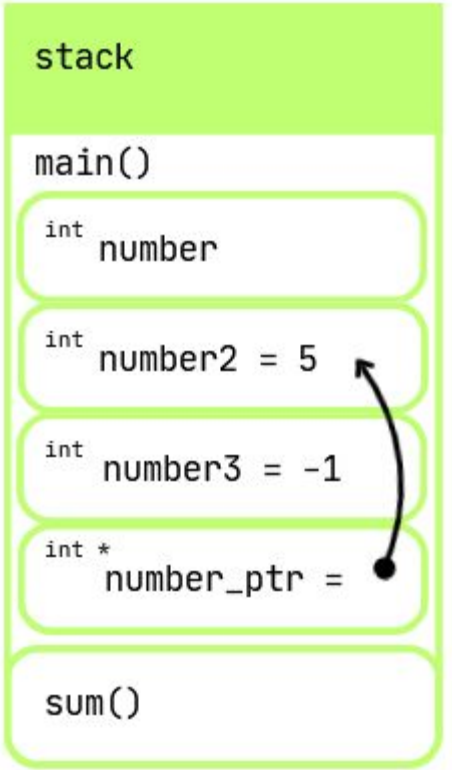


# Stack Example

- Call the function sum()
  - Remember, we go on the right first and assign the result to the left
- Allocate space for sum on the memory 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;
}
```

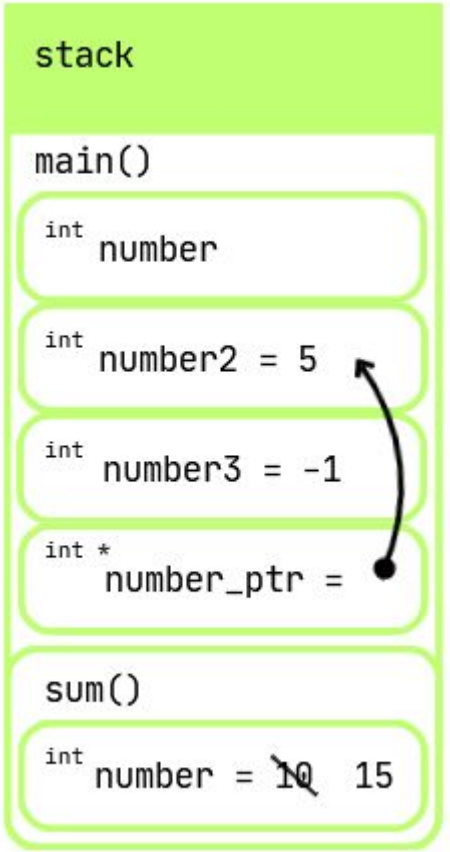


# Stack Example

- 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;
}

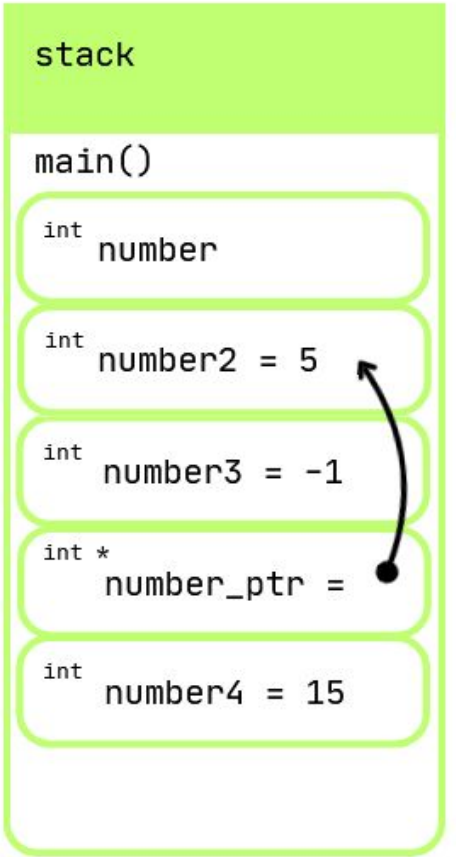
int main(void) {
    int number;
    int number2 = 5;
    int number3 = -1;
    int *number_ptr = &number2;
    int number4 = sum();
    return 0;
}
```



# Stack Example

- Deallocate the stack memory of sum() and return 15 to the main function
  - Allocate space for number4 and assign 15 to it.

```
1 int sum(void) {
2     int number = 10;
3     number += 5;
4     return number;
5 }
6
7 int main(void) {
8     int number;
9     int number2 = 5;
10    int number3 = -1;
11    int *number_ptr = &number2;
12    int number4 = sum();
13    return 0;
14 }
```



# Stack Example

- Deallocate the stack memory for main and return 0 to finish

```
1 int sum(void) {
2     int number = 10;
3     number += 5;
4     return number;
5 }
6
7 int main(void) {
8     int number;
9     int number2 = 5;
10    int number3 = -1;
11    int *number_ptr = &number2;
12    int number4 = sum();
13    return 0;
14 }
```



# Quick Recap

---

- So far we have discussed how variables are stored in memory
  - They live in their {} world on the stack memory
  - If we create data in a function, it will die when the function finishes
  - This is memory allocated by the compiler at compile time

# Quick Recap

---

- That means this cannot work

```c
#include <stdio.h>

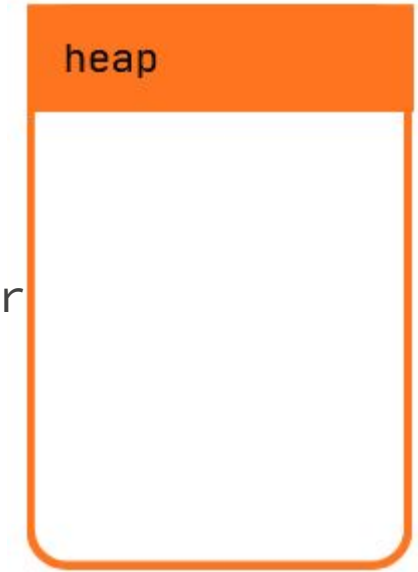
int *create_array(void) {
    int numbers[10] = {0};
    // Return pointer to the array
    return numbers;
}

int main(void) {
    int *array = create_array();
    printf("%d\n", array[0]);
    return 0;
}
```

# Revisiting Memory

---

- A helper function cannot return a pointer to a stack variable
  - How can we deal with this?
    - We can copy it to more permanent storage!
      - The heap! Yay!
- Unlike stack memory, heap memory is allocated by the programmer, and won't be deallocated until it is explicitly freed by the programmer
  - You are in control now! Mwahaha



# malloc()

---

- Function to allocate memory onto the heap
  - Takes how many bytes of memory we need
  - Returns pointer to the piece of memory we created
  - We can now dynamically create memory
    - We are in control now

# free()

---

- You can’t just go sicko mode and ask for memory over and over again
  - If we don’t kill it, it eats up all our computer’s memory
    - Often called a memory leak
  - If we malloc without freeing, bad things happen
- Use free() on the memory we malloc to prevent memory leaks

# sizeof()

- We need to know how much memory we want when we malloc
- Use sizeof()
  - Gives us the exact size of the data type we try to malloc
    - Even if it's a struct!

```c
#include <stdio.h>

int main(void) {

    int array[10] = {0};

    // Display the size of various data types and variables.
    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 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));

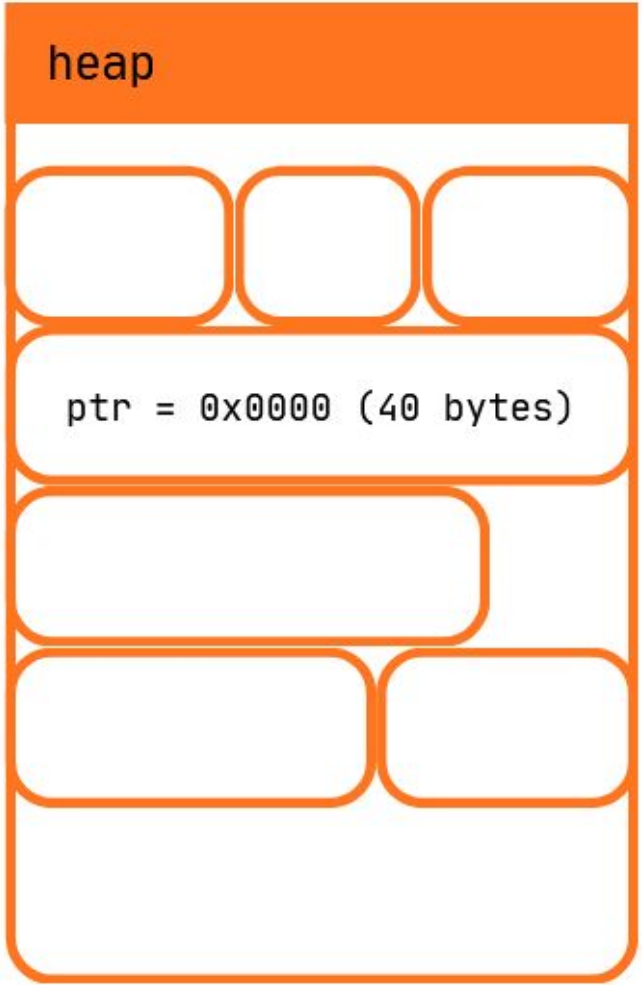
    return 0;
}
```

# malloc() Example

- Using the malloc() function example
  - How much memory will we get?

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

int main(void) {
    int x = 10;
    int *ptr = malloc(x * sizeof(int));
    printf("%p\n", ptr);
    return 0;
}
```



# Putting it all Together

---

- Simple example of malloc, sizeof, and free!

```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?

---

- We’ve talked before about how arrays are containers of certain data types
  - They’re also contiguous in memory
- What does this mean?
- What does this look like?

# Functions and Arrays

---

- We can pass arrays into functions, but we cannot return them
  - Why is that?
- We don’t actually send all the data to the function
  - We only send the address of the first value
  - Because arrays are contiguous in memory, we can alter the array within a function

# Functions and Arrays

```c
#include <stdio.h>

void increment_all(int nums[], int length) {
    for (int i = 0; i < length; i++) {
        nums[i]++;
    }
}

int main(void) {
    int array[5] = {1, 2, 3, 4, 5};
        printf("%d\n", array[3]);
    increment_all(array, 5);
    printf("%d\n", array[3]);
    return 0;
}
```

# Structs and Pointers

---

- When I have a struct, how do I access its elements?
  - I use .
    - player.x
      player.y

# Structs and Pointers

---

- What if I have a pointer of type struct?
  - How do we access those?

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

#define MAX 35

// 1. Define the struct
struct pet {
    char name[MAX];
    char species[MAX]; // "Dog" or "Cat" or ???
    int age;
    double weight;
};

int main(void) {
    struct pet my_pet;

    struct pet *pet_ptr = &my_pet;

    strcpy((*pet_ptr).name, "Buddy");
    strcpy((*pet_ptr).species, "Dog");
    (*pet_ptr).age = 3;
    (*pet_ptr).weight = 12.50;

    printf("--- Pet Record ---\n");
    printf("Name:    %s\n", (*pet_ptr).name);
    printf("Species: %s\n", (*pet_ptr).species);
    printf("Age:     %d years old\n", (*pet_ptr).age);
    printf("Weight:  %.2lf kg\n", (*pet_ptr).weight);

    return 0;
}
```

# Structs and Pointers

---

- All these brackets can get a bit confusing
  - No need to use (*player).x = 6
  - We can just use player->x = 6 instead

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

#define MAX 35

// 1. Define the struct
struct pet {
    char name[MAX];
    char species[MAX]; // "Dog" or "Cat" or ???
    int age;
    double weight;
};

int main(void) {
    struct pet my_pet;

    struct pet *pet_ptr = &my_pet;

    strcpy(pet_ptr->name, "Buddy");
    strcpy(pet_ptr->species, "Dog");
    pet_ptr->age = 3;
    pet_ptr->weight = 12.50;

    printf("--- Pet Record ---\n");
    printf("Name:    %s\n", pet_ptr->name);
    printf("Species: %s\n", pet_ptr->species);
    printf("Age:     %d years old\n", pet_ptr->age);
    printf("Weight:  %.2lf kg\n", pet_ptr->weight);

    return 0;
}
```

# Why Are We Doing This?

---

- All of this pointer stuff is essential for our new favourite data type!
- Introducing...
  - Linked lists!!!

# Linked Lists

---

- Like an array, linked lists store data of the same type
- So why bother?
  - Linked lists are dynamically sized
  - Elements of a linked list do not need to be stored contiguously in memory
  - Linked lists are NOT random access data structure
    - You have to go through them sequentially

# What We Learnt

---

- Memory and pointers
  - Ways to reference addresses
- Arrays as pointers
- Structs as pointers

# Break Time

...probably

# Multi-File Projects

---

- Big problems are often spread across multiple files
  - Improve readability
  - Separate problems into different spaces
  - Written and tested individually

# Multi-File Projects

---

- We've already been using them!
  - Whenever you #include, you are using content from another file
    - #include <stdio.h>
    - #include <strings.h>
- We can #include our own files!

# Why Bother?

---

- #include-ing our own files allows us to:
  - Join projects together
  - Have multiple people working on the same project
  - Reuse code in other projects
    - Like the C standard libraries

# .h and .c

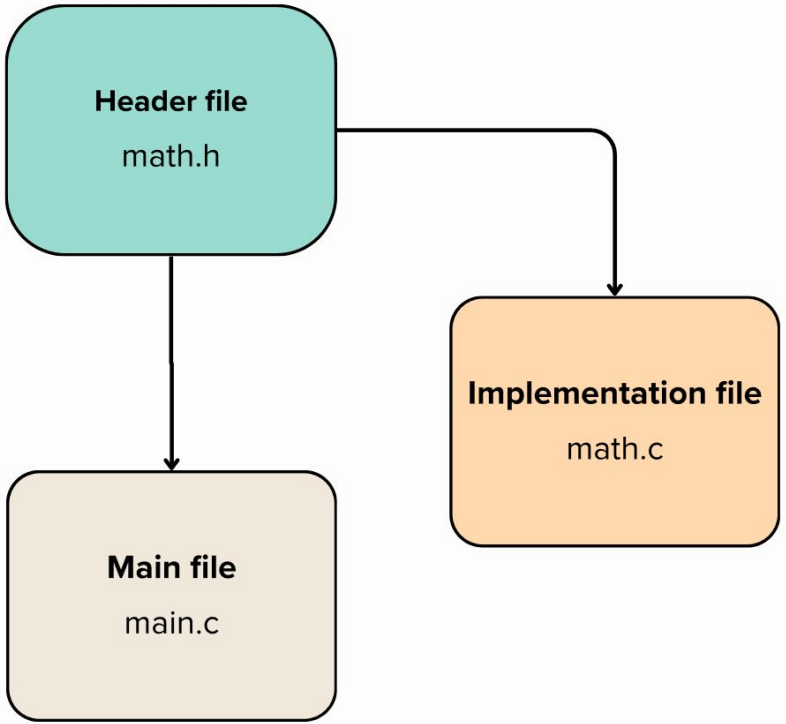
---

- In a multi-file project we may have
  - Header files
    - This is the .h file that you've already been using from standard libraries
  - Implementation files
    - This is the .c file which implements what is in the header file
- Each header file has its own implementation file
- We will also have a main .c file
  - This is the entry to our program
  - Contains the main function
  - We try and have as little code here as possible

# .h and .c

---

- The .h file serves to link together the main and the implementation



# The Header File

---

- Typically contains
  - Function prototypes for all functions in the implementation file
  - Comments describing what these functions do
  - #defines
  - enum definitions
- The file SHOWS the programmer all they need to know to use the code
  - It doesn’t actually implement anything

# The Implementation File

---

- This is where you implement what you define in the header file

# main.c

---

- Where you call the code from other files

# Compilation

---

- Must include all .c files when compiling

```
/week7/maths$ dcc main.c maths.c -o maths
/week7/maths$ ./maths
```

# What Does this Look Like?

---

- Let’s explore a multi-file project together and make our own!

# What We Learnt

---

- Memory and pointers
  - Ways to reference addresses
- Arrays as pointers
- Structs as pointers
- Multi-file projects

# Next Time

---

- Linked lists!!!!!!!
  - 🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉
  - 🥳🥳🥳🥳🥳🥳🥳🥳🥳🥳🥳🥳🥳🥳🥳🥳🥳

# Reach Out

---

- Check the course forum for questions!
  - Post there too!
- Admin questions
  - cs1511@unsw.edu.au

# Thank You

Questions?