COMP1511 PROGRAMMING FUNDAMENTALS

Lecture 11

Multi-File Projects
More memory and the start of something beautiful (linked lists)

ONE WEEK AGO...

- Pointers (fun?!)
- A whole bunch of useful functions with characters and strings: getchar(), putchar(), fgets(), etc.
- Strings
- Command Line Arguments

TODAY...

- We will ease back into it (promise!)
- Multi File Projects
- A bit more about memory
- Starting to think about another data structure - linked list

WHERE IS THE CODE?

LIVE LECTURE CODE CAN BE FOUND HERE:

https://cgi.cse.unsw.edu.au/~cs1511/21T3/live/Week07/

QUICK REVISIT OF THE LAST BIT OF CODE WE WROTE IN WEEK 5

COMPARE.C

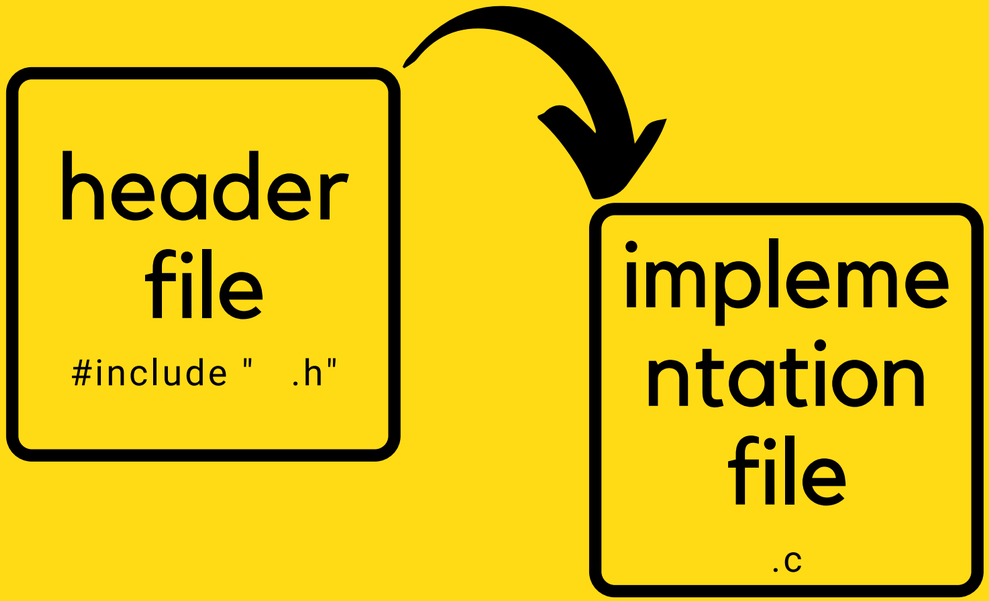
MULTI FILE PROJECTS

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
  - 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 AND A .C FILE



- 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 " .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

.c

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

MAIN

MAIN.C

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

AN EXAMPLE

SOME SORT OF MATHS MODULE

- 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

```
// 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);
```

maths.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;
}
```

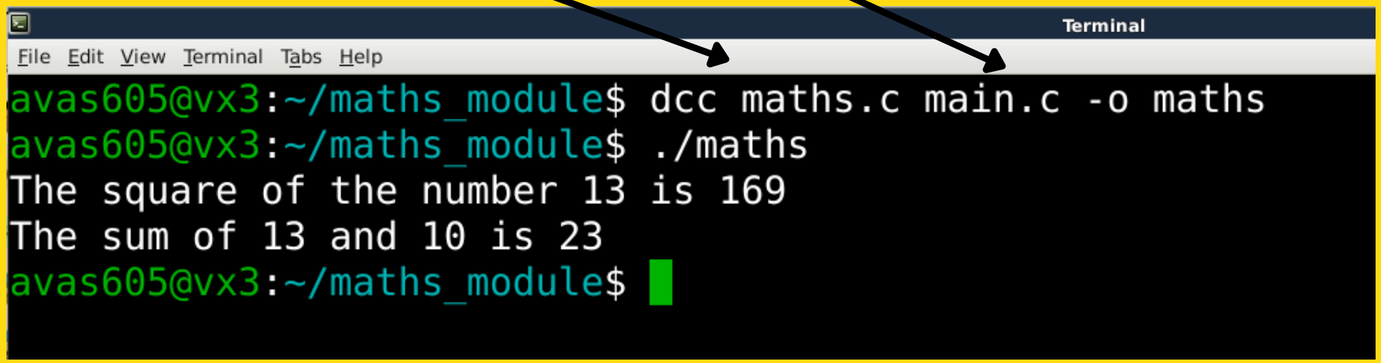
main.c

```
//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;
}
```

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

BREAK TIME (5 MINUTES)

You have 8 sticks; 4 of them are of one length and the four remaining ones are a different length. Arrange the sticks so that they form 3 identical squares.

QUICK REHASH

MEMORY

- 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):

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...

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

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(bytes)
  - 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

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... 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()

IF I RUN MALLOC, HOW DO I KNOW HOW MANY BYTES I WANT TO HOLD?

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 type
//We use format specifier %lu because

#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 ints (array[10]) 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));
    printf("etc\n");
    return 0;
}
```

sizeof_eg.c

PUTTING IT ALL TOGETHER:

MALLOC(SIZEOF()) FREE()

Allocate memory as needed with malloc() and using sizeof() to determine bytes needed. Remember malloc() returns a pointer to that memory

Free the memory that you had reserved with free(), once you are done using it.

- Using all of these together in a simple example:

```c
#include <stdio.h>
//malloc() and free() live inside the <stdlib.h>
#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 memory\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;
}
```

malloc_eg.c

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 members with .)
    //Remember we can't just do jax.name = "Jax"
    //So we will 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 that points to the variable jax of type struct dog
    struct dog *jax_ptr = &jax;

    //3. Initialise struct (access members with .)
    //Remember we can't just do jax.name = "Jax"
    //So we will use the function strcpy() in <string.h> to copy the string over
    //strcpy(jax.name, "Jax");
    //jax.age = 6;

    //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 ->

```
27      strcpy((*jax_ptr).name, "Jax");
28      (*jax_ptr).age = 6;
29
30      printf("%s is an awesome dog, who is %d years old\n", (*jax_ptr).name,
31                                                            (*jax_ptr).age);
```

Note that there is no need to use *(jax_ptr) and instead can just straight jax_ptr ->

```
    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 AND WHEN WOULD THIS BE USEFUL?

INTRODUCING A NEW DATA STRUCTURE

- Now that you have become comfortable with arrays, we are going to become acquainted with another important data structure: linked list
- What is a linked list?
  - Like an array, it 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.

LET'S VISUALISE IT...

WHAT DOES A LINKED LIST LOOK LIKE IN MEMORY?

SO WHAT DOES THAT MEAN A NODE IS?

A COLLECTION OF SOME DATA AND ALSO A POINTER TO THE NEXT NODE

- Since a node is a collection of two things: some data and a pointer, it is a great contender for a struct to hold this collection:

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

- int data - is just some information we are going to store of type int
- struct node *next - is a pointer called next of type struct node, this basically means that the pointer holds the address to the next node

TOMORROW WE WILL START TO PUT THIS INFORMATION TOGETHER

WHAT HAPPENS WHEN WE HAVE MANY NODES?
HOW DO WE GROW OUR LIST?
HOW DO WE SHRINK OUR LIST? (NEXT WEEK)

FEEDBACK?

PLEASE LET ME KNOW ANY FEEDBACK FROM TODAY'S LECTURE!

www.menti.com

Code: 8852 4616

WHAT DID WE LEARN TODAY?

MULTI FILE PROJECTS

maths.c
maths.h
main.c

MALLOC(), SIZEOF(), FREE()

sizeof_eg.c
malloc_eg.c

STRUCTS AND POINTERS

struct_ptr.c

FIRST GLIMPSE AT LINKED LIST

ANY QUESTIONS?

DON'T FORGET YOU CAN ALWAYS EMAIL US ON CS1511@CSE.UNSW.EDU.AU FOR ANY ADMIN QUESTIONS

PLEASE ASK IN THE FORUM FOR CONTENT RELATED QUESTIONS