COMP1511 - Programming Fundamentals

Week 4 - Lecture 8

What did we learn last lecture?

Functions and Libraries

- Using functions we haven't written
- `#include` to use C standard libraries

Multi-Dimensional Arrays

- Arrays of arrays
- Like a 2D (or more than 2 dimensions) map or grid

What are we covering today?

Memory

- Looking at computer memory in more detail

Pointers

- A C variable that lets us access memory directly

Libraries recap

We can use functions from the C standard libraries

- These are added to our code using **`#include`**
- Once they're included, they provide access to their functions
- An example is **`<stdio.h>`** giving access to **`printf`** and **`scanf`**
- We'll see more of these as we add new techniques!

Multi Dimensional Arrays Recap

We can store any variables in arrays

- Arrays are variables!
- We can store arrays in arrays
- In 2 dimensions, this can build a grid

Indexes	0	1	2	3	4
0	63	88	43	55	67
1	54	52	91	21	32
2	77	58	1	61	79

A 2D Array

Accessing 2D Arrays

Two coordinates to access single elements

- We use two dimensions to create the 2D array
- We also use two coordinates to get access to a single element

```
int main (void) {
    // declare a 2D Array
    int grid[4][4] = {0};

    // test a value
    if (grid[2][0] < 1) {
        // print out a value
        printf("The bottom left square is: %d", grid[3][0]);
    }
```

Memory and addressing

More detail about how memory works in our computer

- Let's start with an idea of a neighbourhood
- Each house is a piece of memory (a byte or more, depending)
- Every house has a unique address that we can use to find it

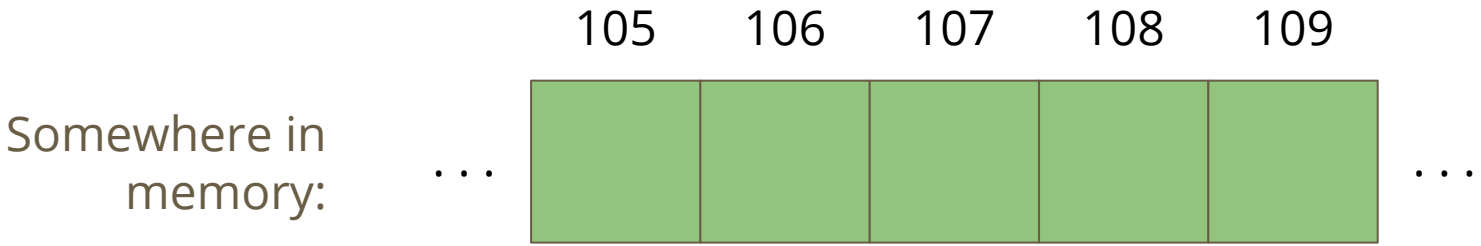
Arrays work a bit like this . . .

- We've already seen indexing into arrays to find elements
- We could think of our entire computer's memory as a big array of bytes

A neighbourhood of memory

Every block of memory has an address

- The address is actually an integer
- If I have that address, it means I can find the variable wherever it is in memory
- Just like if I have an address to a house, I'll be able to find it



Houses and addresses

Continuing the idea . . .

- A variable is a house
- That house is in a certain location in memory, its address
- The house contains the bits and bytes that decide what the value of the variable is

The address is an integer

- In a 64 bit system, we'll usually use a 64 bit integer to store an address
- We can address 2^{64} bytes of memory

Introducing Pointers

A New Variable Type - Pointers

- Pointers are variables that hold memory addresses
- They are created to point at the location of other variables

- If a variable was a house, the pointer would be the address of that house
- In C, the pointer is like an integer variable that stores a memory address
- Pointers are usually created with the intention of "aiming at" a variable (storing a particular variable's address)

Pointers in C

Pointers can be declared, but slightly differently to other variables

- A pointer is always aimed at a particular variable type
- We use a ***** to declare a variable as a pointer
- A pointer is most often "aimed" at a particular variable
- That means the pointer stores the address of that variable
- We use **&** to find the address of a variable

```
int i = 100;
// create a pointer called ip that points at i
// it stores the address of i
int *ip = &i;
```

Pointer Types

Different pointers to point at different variables

```
// some variables
int i;
double d;

// some pointers to particular variables
// * declares a pointer variable
// & finds the address of a variable
int *ip = &i;
double *dp = &d;
```

Initialising Pointers

Pointers should be initialised like other variables

- Generally pointers will be initialised by pointing at a variable
- "`NULL`" is a `#define` from most standard C libraries (including `stdlib.h`)
- If we need to initialise a pointer that is not aimed at anything, we will use `NULL`

Using Pointers

If we want to look at the variable that a pointer "points at"

- We use the **`*`** on a pointer to access (dereference) the variable it points at
- Using the address analogy, this is like following the address to actually get to the house, then looking inside

```
int i = 100;
// create a pointer called ip that points at i
// it stores the address of i
int *ip = &i;
printf("The value of the variable at %p is %d", ip, *ip);
```

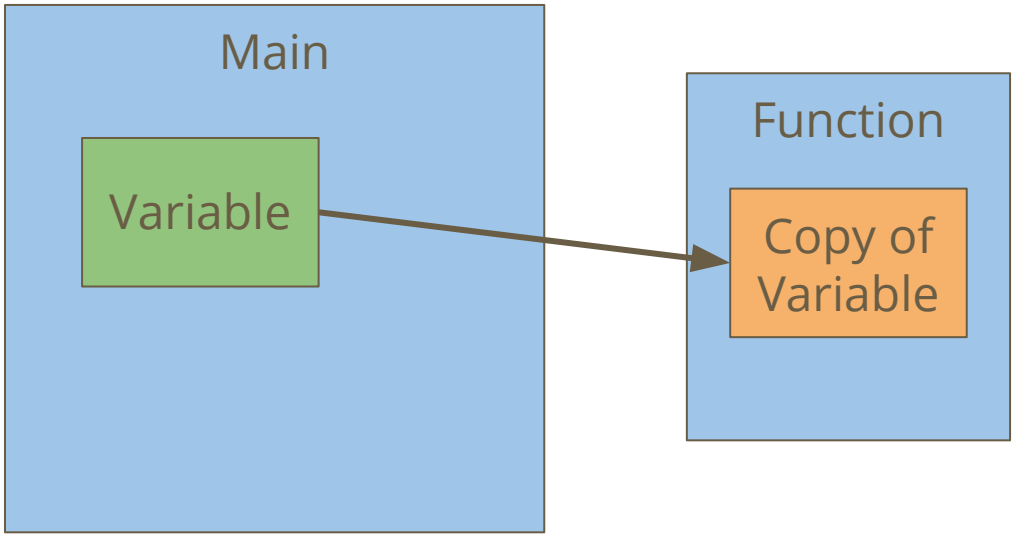
- **`%p`** in **`printf`** will print the address stored in a pointer

Pointers and Functions

Pointers allow us to pass around an address instead of a variable

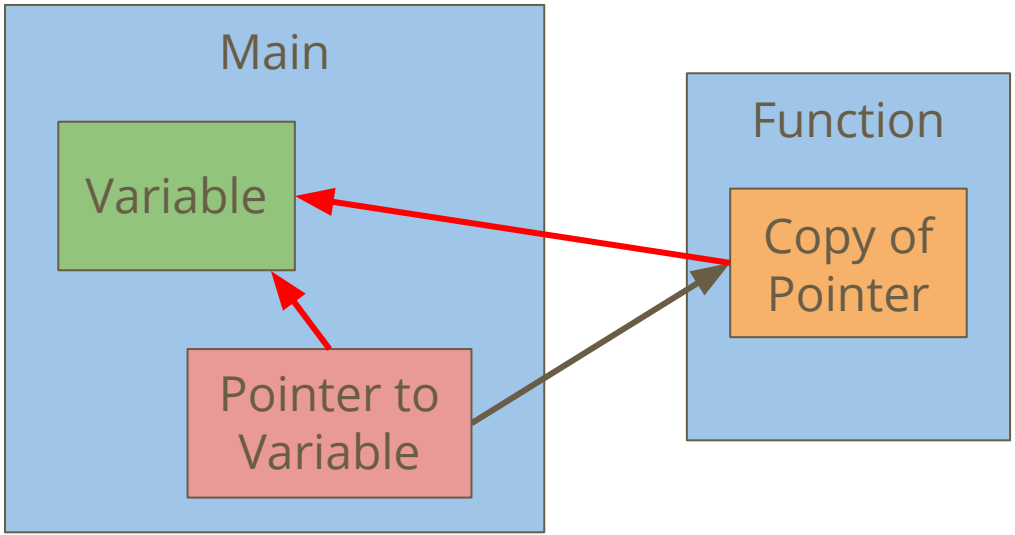
- We can create functions that take pointers as input
- All function inputs are always passed in "by value" which means they're copies, not the same variable
- But if I have a copy of the address of a variable, I can still find exactly the variable I'm looking for

Function variables pass in "by value"



In this case, the copy of the variable can't ever change the value of the variable, because it's just a copy

Pointers pass in "by value" also



The function has a copy of the pointer.

However, even a copy of a pointer contains the address of the original variable, allowing the function to access it.

Pointers and Functions in code

The following code illustrates the two examples

- A variable passed to a function is a copy and has no effect on the original
- A pointer passed to a function gives us the address of the original

```
// this function will have no effect!
void increment_int(int n) {
    n = n + 1;
}
// this function will affect whatever n is pointing at
void increment_ptr(int *n) {
    *n = *n + 1;
}
```

Pointers and Functions

We can now do more with functions

- Pointers mean we can give a function access to multiple variables
- This means one function can now change multiple variables at once

```
// This function is now possible!
void swap(int *n, int *m) {
    int tmp;
    tmp = *n;
    *n = *m;
    *m = tmp;
}
```

Pointers and Arrays

Arrays are blocks of memory

- An array variable actually stores the memory address of the start of the array!
- This is why arrays as input to functions let you change the array

```
int numbers[10];
// both of these print statements
// will print the same address!
printf("%p\n", &numbers[0]);
printf("%p\n", numbers);
```

Pointers to Pointers

- Pointers are variables
- Pointers can point at variables
- uh oh . . .
- For now, we will not use pointers aimed at other pointers, but in the future you may find uses for them

Break Time

Making Games with Computers

- Computers and games have a strong history (since the 1950s)
- The theme of our first assignment (Slide)
- What follows from something simple like moving blocks
- Ends up with interactive 3D worlds and deep immersion
- You can run a realtime version of Slide in a CSE terminal by using the command: `1511 arcade solution`

Let's make a program using functions and pointers

This program is called The Shuffler

- It will take some numbers as inputs
- It will shuffle them a little, changing their order
- Then it will print them back out

- We'll make some use of functions to separate our code
- We'll show how pointers let us access memory in our program

What functions do we want?

Deciding how to split up our functionality

- A function that reads the inputs as integers
- A function that swaps two numbers
- A function that swaps several numbers
- A function that prints out our numbers

Reading Input

A function to read inputs into an array

- We're also going to want to know how many numbers are being entered!

```
int read_inputs(int nums[MAX_NUMS]) {
    int i = 0;
    int input_count = 0;
    printf("How many numbers? ");
    scanf("%d", &input_count);
    while (i < MAX_NUMS && i < input_count) { // have processed i inputs
        scanf("%d", &nums[i]);
        i++;
    } // have processed i inputs in total
    return input_count
}
```

Printing our numbers

This is a trivial function

- The only issue is that we might have to work with an array that isn't full
- So we use **`num_count`** to stop us early if necessary

```
void print_nums(int nums[MAX_NUMS], int num_count) {
    int i = 0;
    while (i < MAX_NUMS && i < num_count) {
        printf("%d ", nums[i]);
        i++;
    }
}
```

Using Pointers to swap variable values

A simple swap function

- This function doesn't even know whether the ints are in arrays or not
- It sees two memory locations containing ints
- and uses a temporary int variable to swap them

```
void swap_nums(int *num1, int *num2) {
    int temp = *num1;
    *num1 = *num2;
    *num2 = temp;
}
```

Shuffle performs some swaps

This function just loops through and swaps a few numbers

- This is a good candidate for a function that could be changed or written differently and just used by our main without thinking about it

```
void shuffle(int nums[MAX_NUMS], int num_count) {
    int i = 0;
    while (i < MAX_NUMS && i < num_count) {
        int j = i * 2;
        if (j < MAX_NUMS && j < num_count) {
            swap_nums(&nums[i], &nums[j]);
        }
        i++;
    }
}
```

Using all the functions in the main

A nice main makes use of its functions

- It's very easy to read this main!
- It shows its steps using its function names
- There isn't much code to dig through

```
int main(int argc, char *argv[]) {
    int numbers[MAX_NUMS];
    int numInputs = read_args(numbers);
    shuffle(numbers, numInputs);
    print_nums(numbers, numInputs);
    return 0;
}
```

It's a simple program, but what's different?

Using functions, we have much more readable code

- Large sections of code are outside of the main
- The main itself is now very readable
- Each separate piece of functionality is on its own

Pointers give us access to other parts of memory

- We can give access to our variables via pointers

What did we learn today?

Memory and Pointers

- Pointers are variables that contain memory addresses
- We can use them to get access to variables anywhere in our program
- Functions operate in their own memory "space"

Using Functions

- A practical example of how functions can separate code
- Makes our code very readable
- Also means that all of the code for a specific purpose is collected together