COMP1511 - Programming Fundamentals

Term 3, 2019 - Lecture 10

What did we learn yesterday?

Assignment 1 - CS Paint

- Assessment and some details

Functions and Libraries

- Including other C Libraries

Characters and Strings

- Letters and words in C

What are we covering today?

Memory and Pointers

- We're going to take one step closer to the memory we've been using
- Pointers allow us to access memory directly

Halfway point of COMP1511

- Let's make a program that uses everything we've learnt so far

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)
- Every house has a unique address that we can use to find it

Arrays work a bit like this . . .

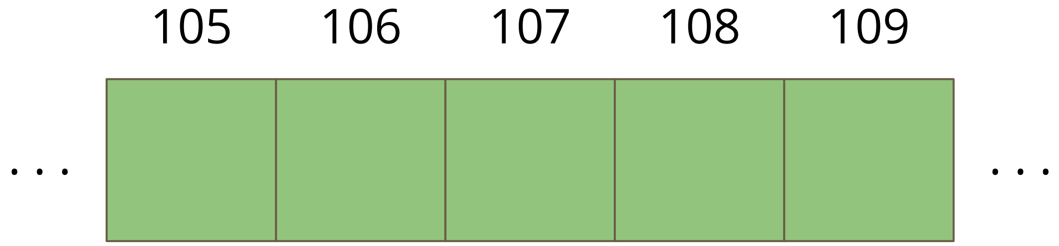
- We've already seen indexing into arrays to find elements
- We could think of our entire computer's memory as a big array of bytes

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

Somewhere in memory:



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 memory addresses
- They are created to point at the location of variables

- If a variable was a house, the pointer would be the address of that house
- In C, the pointer is like an integer 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
// an integer in the location of i
int *ip = &i;
```

Pointer Types

Different pointers to point at different variables

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

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

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 stdio.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 asking what’s inside the house at that address

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

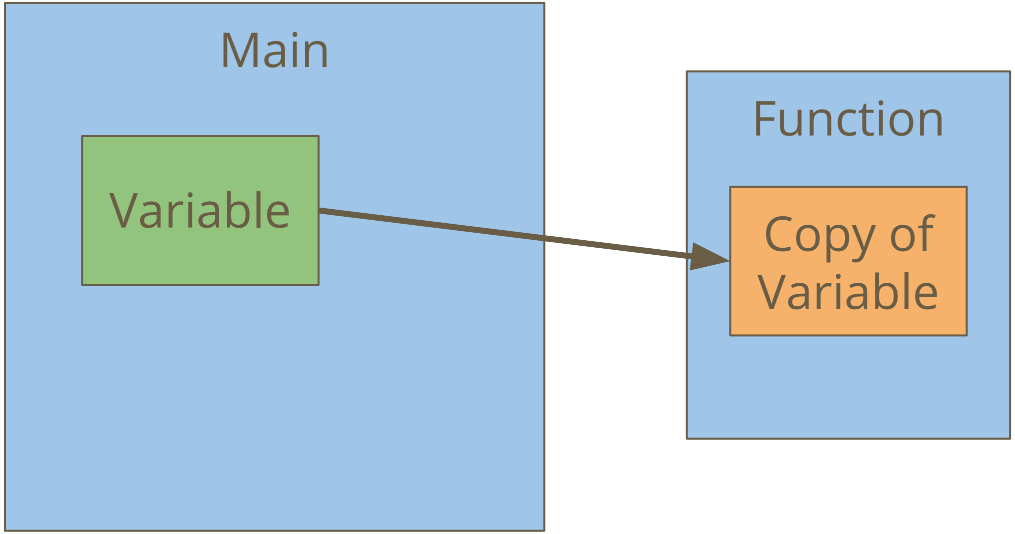
- `%p` in `printf` will print the address of a pointer

Pointers and Functions

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

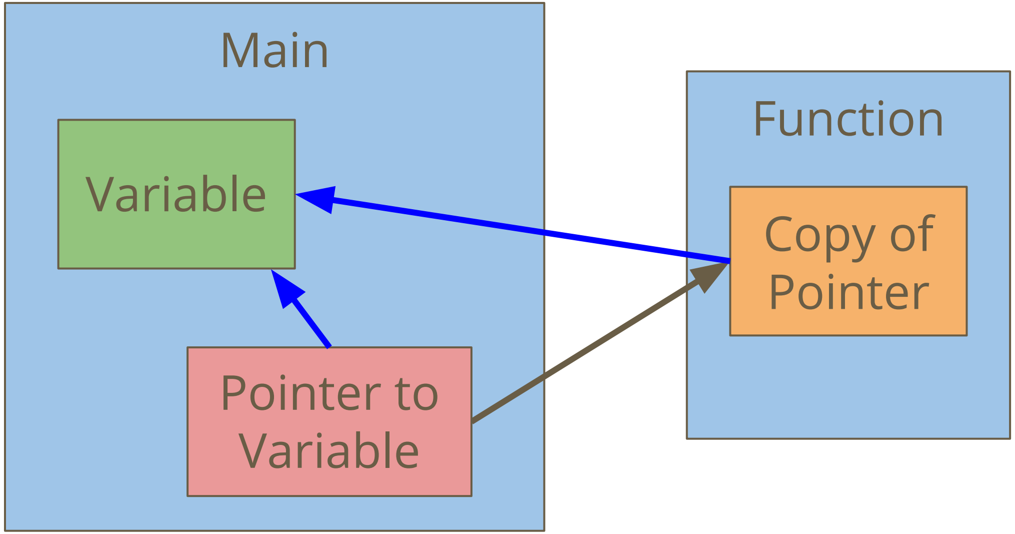
- We can create functions that take pointers as input
- All function inputs are always passed in "by value" which means they're copies, not the same variable
- But if I have a copy of the address of a variable, I can still find exactly the variable I'm looking for

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 incrementInt(int n) {
    n = n + 1;
}
// this function will affect whatever n is pointing at
void incrementPointer(int *n) {
    *n = *n + 1;
}
```

Pointers and Functions

We can now do more with functions

- Pointers mean we can give multiple variables to a function
- 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

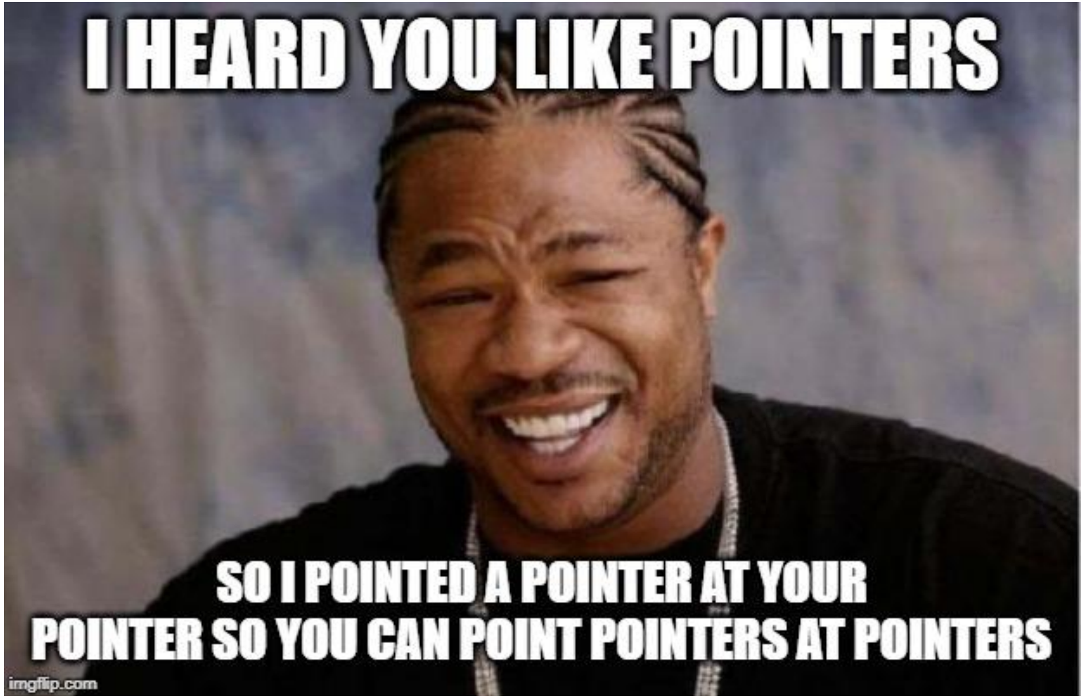
Arrays are blocks of memory

- The array variable is actually a pointer to 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);
```

Break Time

- Pointers are variables
- Pointers can point at variables
- uh oh . . .


I HEARD YOU LIKE POINTERS
SO I POINTED A POINTER AT YOUR POINTER SO YOU CAN POINT POINTERS AT POINTERS
imgflip.com

Whooaaah We're Halfway There ...

We're going to use a bit of everything we've seen so far in COMP1511

This program is a word game

- It will read in a string from the user
- It will then read in another string from the user and tell us how many of the letters from the second appear in the first
- This will use if, while, arrays (of characters), functions and pointers

Where will we start?

A simple version to begin with

- Let's read in a line of characters
- Then read in a single character and see whether it's in the line or not

Read in a line of characters (a string)

We can use a nice library function here

- **`fgets()`** will grab an entire line from standard input
- We can set up a maximum line size as well

```
#define MAX_LINE_LENGTH 100

int main(void) {
    char line[MAX_LINE_LENGTH];
    fgets(line, MAX_LINE_LENGTH, stdin);
```

Read in a single character

Starting simple, we can take a character as input

- **`getchar()`** will read a single character from standard input
- Remember that we'll be using int as our type for individual characters
- Here we can loop and continually get characters until input ends

```
int inputChar;
inputChar = getchar();
while (inputChar != EOF) {
    inputChar = getchar();
}
```

A Function to find a character in a string

Loop through the string, testing for a character

- We've done this kind of loop before with other types!

```
int testChar(char c, char *line) {
    int charCount = 0;
    int i = 0;
    while (i < MAX_LINE_LENGTH && line[i] != '\0') {
        if (line[i] == c) {
            charCount++;
        }
        i++;
    }
    return charCount;
}
```

Simple functionality... how well is it working?

What tests should we run at this point?

- Look for syntax errors using our compiler (dcc)
- Look for logical errors by testing with different inputs

We might need to add in some extra outputs

- If we're getting strange behaviour, we can confirm our guesses
- We might learn more about what's going on in our program

What are these extra characters?

Maybe we need to check what those characters are

- Some print statements can help here

```
    int inputChar;
    inputChar = getchar();
    while (inputChar != EOF) {
        printf("Main loop running, readChar is %c.\n", inputChar);
        printf("%d\n", testChar(inputChar, line));
        inputChar = getchar();
    }
```

Dealing with little issues

We're reading newlines (`\n`) as characters!

- Let's remove the newlines from both our line and our inputs
- We'll use a library function, **`strlen()`** to find the end of a string
- To use **`strlen()`**, we will need the string.h library, which we will include
- We'll then replace the **`\n`** with **`\0`** which will end the string early

Removing newlines

Removing a `\n` at the end of a string:

```
int main(void) {
    char line[MAX_LINE_LENGTH];
    fgets(line, MAX_LINE_LENGTH, stdin);
    int length = strlen(input);
    input[length - 1] = '\0';
```

Ignoring the `\n` while reading input:

```
    inputChar = getchar();
    if (inputChar == '\n') {
        inputChar = getchar();
    }
```

Expanding on the functionality

Our first attempt just checked for single letters

- Now we expand to words!
- Read in another word
- Check every letter in the word for whether it appears in the phrase
- Then report back how many letters matched

Some good reasons to use functions!

- Reading in words is now duplicated
- We can reuse our testChar() function to see if letters match

A function to read a line

This function also removes the `\n` that fgets will give us

```c
void readString(char *input) {
    fgets(input, MAX_LINE_LENGTH, stdin);
    int length = strlen(input);
    input[length - 1] = '\0';
}
```

A function to count letters

Counts how many letters from one string appear in the other

This function also uses another function!

```
int numLetterMatches(char *word, char *line) {
    int i = 0;
    int matchCount = 0;
    while (i < MAX_LINE_LENGTH && word[i] != '\0') {
        if (testChar(word[i], line)) {
            matchCount++;
        }
        i++;
    }
    return matchCount;
}
```

A simple word game

What coding concepts have we used there that might come in handy?

- Characters and Strings (note that we'll never need the ASCII table to work with characters)
- Using libraries and provided functions
- Loops on strings (using the Null Terminator `\0`)
- Writing multiple functions and using functions within functions
- A lot of our basic C concepts like if, while and array indexing

What did we learn today?

Memory and Pointers

- All variables exist at some address in memory
- A pointer is a copy of an address that allows us to access memory

Coding using everything we've learnt so far

- A single program that tries to use most concepts we've covered in the first half of this course