COMP1511 - Programming Fundamentals

Term 2, 2019 - Lecture 11

What did we learn last week?

Assignment 1

- Everything you need to know about CS Paint!

Professionalism

- The importance of skills beyond the technical

Characters and Strings

- Using letters and words in C

What are we covering today?

Command Line Arguments

- Adding information to our program when it runs

Pointers

- Directly addressing memory

Characters and Strings Recap

Our new variable type: `char`

- Represents a letter
- Is also a number, an ASCII code, and we'll often use `int`s to represent a character
- When used in arrays, they're referred to as strings
- Strings often end before the end of the array they're stored in
- When they do, we store a null terminator `'\0'` after the last character

Characters in code

```c
#include <stdio.h>

int main (void) {
    // we're using an int to represent a single character
    int character;
    // we can assign a character value using single quotes
    character = 'a';
    // This int representing a character can be used as either
    // a character or a number
    printf("The letter %c has the ASCII value %d.\n", character,
character);
    return 0;
}
```

Note the use of %c in the printf will format the variable as a character

Strings in Code

Strings are arrays of type char, but they have a convenient shorthand

```c
// a string is an array of characters
char word1[] = {'h','e','l','l','o'};
// but we also have a convenient shorthand
// that feels more like words
char word2[] = "hello";
```

Both of these strings will be created with 6 elements. The letters **`h,e,l,l,o`** and the null terminator **`\0`**

h	e	l	l	o	\0

Command Line Arguments

Sometimes we want to give information to our program at the moment when we run it

- The **"Command Line"** is where we type in commands into the terminal
- **Arguments** are another word for input parameters

```
$ ./program extra information 1 2 3
```

- This extra text we type after the name of our program can be passed into our program as strings

Main functions that accept arguments

`int main` doesn't have to have `void` input parameters!

```
int main(int argc, char* argv[]) {
}
```

- **argc** will be an "argument count"
- This will be an integer of the number of words that were typed in (including the program name)
- **argv** will be "argument values"
- This will be an array of strings where each string is one of the words

An example of use of arguments

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int i = 1;
    printf("Well actually %s says there's no such thing as ", argv[0]);
    while (i < argc) {
        fputs(argv[i], stdout);
        printf(" ");
        i++;
    }
    printf("\n");
}
```

Arguments in argv are always strings

But what if we want to use things like numbers?

- We can read the strings in, but we might want to process them

```
$ ./program extra information 1 2 3
```

- In this example, how do we read 1 2 3 as numbers?
- We can use a library function to convert the strings to integers!
- **`strtol()`** - "string to long integer" is from the stdlib.h

Code for transforming strings to ints

Adding together the command line arguments

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

int main(int argc, char *argv[]) {
    int total = 0;

    int i = 1;
    while (i < argc) {
        total += strtol(argv[i], NULL, 10);
        i++;
    }
    printf("Total is %d.\n", total);

}
```

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 . . .

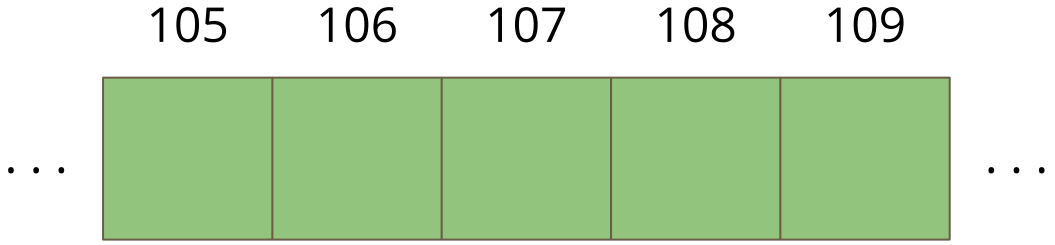
- We've already seen indexing into arrays to find elements
- We could think of our entire computer's memory as a big array of bytes

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

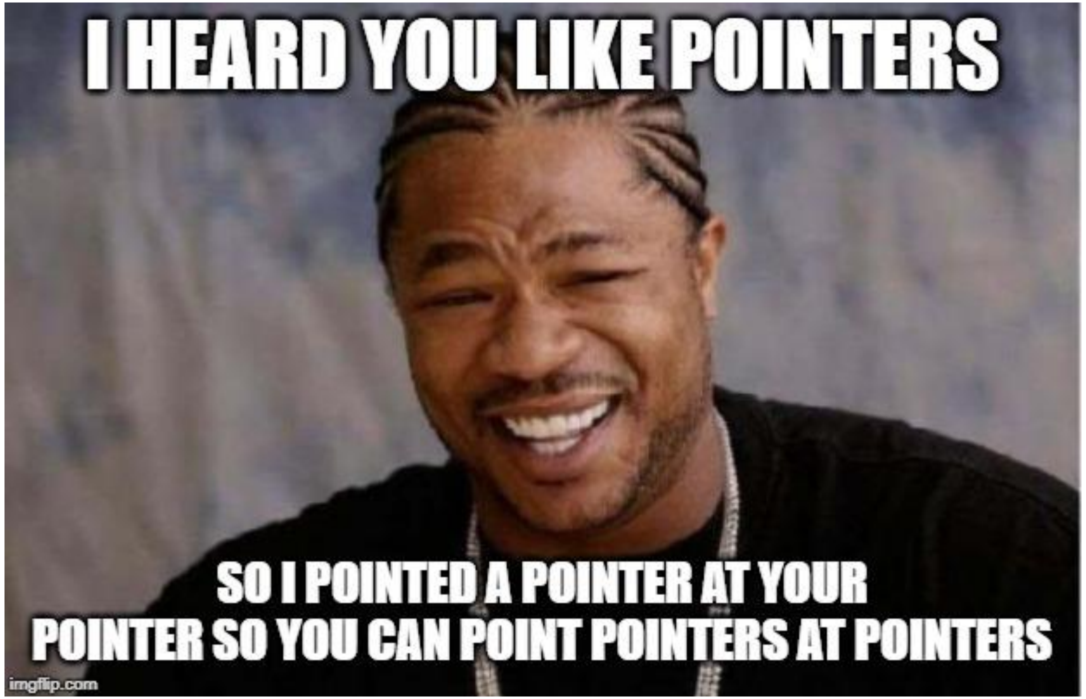
A New Variable Type - Pointers

- 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)

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

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
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);
```

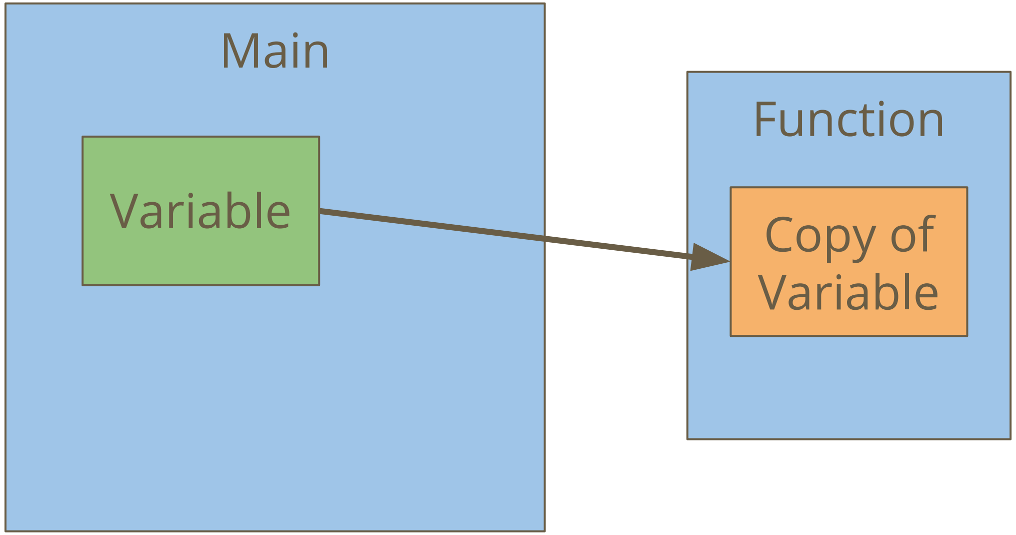
- `%p` in printf will print the address of a pointer

Pointers and Functions

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

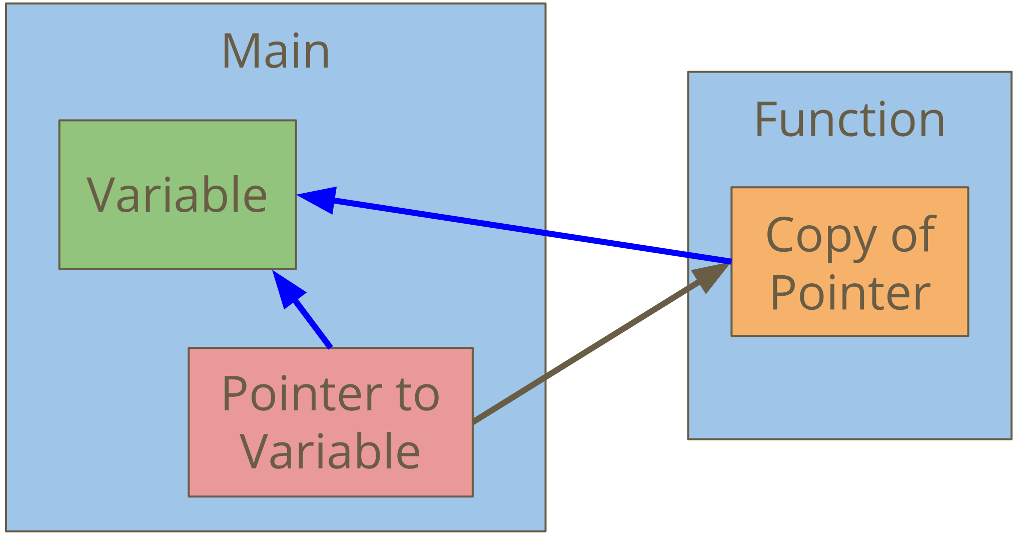
- We can create functions that take pointers as input
- All function inputs are always passed in "by value" which means they're copies, not the same variable
- But if I have a copy of the address of a variable, I can still find exactly the variable I'm looking for

Variables pass "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 "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);
```

Ok let's make a simple program

This program is called the Jumbler

- It will take some numbers as command line arguments
- It will jumble them a little, changing their order
- Then it will print them back out

Converting our Command Line Arguments

We'll read the command line arguments and convert them to ints

- Note that we're ignoring the first element of arguments because we know that it's the name of the program and not one of our numbers

```c
void read_args(int nums[MAX_NUMS], char *arguments[], int argCount) {
    int i = 0;
    while (i < MAX_NUMS && i < argCount - 1) {
        nums[i] = strtol(arguments[i + 1], NULL, 10);
        i++;
    }
}
```

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 numCount to stop us early if necessary

```
void print_nums(int nums[MAX_NUMS], int numCount) {
    int i = 0;
    while (i < MAX_NUMS && i < numCount) {
        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;
}
```

Jumble 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 jumble(int nums[MAX_NUMS], int numCount) {
    int i = 0;
    while (i < MAX_NUMS && i < numCount) {
        int j = i * 2;
        if (j < MAX_NUMS && j < numCount) {
            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];
    read_args(numbers, argv, argc);
    jumble(numbers, argc - 1);
    print_nums(numbers, argc - 1);
    return 0;
}
```

What did we learn today?

Command Line Arguments

- We can take input from the terminal as extra arguments typed in after the program name

Pointers

- Memory addresses in variables
- We can pass pointers to functions and they will have access to our memory
- Arrays are organised like pointers