**MODULE 1 – INTRODUCTION**

**INTRODUCTION – C STYLE GUIDE**

HEADER COMMENT (Final Exam)

// Real name and Student Number  
// Final Exam Name + QN  
// Date  
// What this file is for (one line summary).

**Bits & Bytes**

Bits and Bytes

A byte is an area of memory that is addressable or referred to by name.

* A byte contains 8 bits
* A bit (binary digit) has only 2 values: 0 or 1.
* Each byte can contain a number between 0 and 255 (encoded in 8 binary digits i.e. 255 = 11111111)

Binary counting system

For a chip that has **4-bits** in each byte (e.g. 400x processors), there are 16 different combinations (16 permutations)

* There are ALSO 16 POSSIBLE NUMBERS in decimal notation (0-15 can all be constructed using 4 bits)
* Examples: 0000 (0 in decimal notation), 0001 (1), 0010 (2), 0011 (3), 0100 (4), 0101 (5), 0111 (7), 1111 (15)
* Incrementing the maximum value 1111(15) by 0001(1) would cause the stored value to wrap around to 0000(0)

For a chip that has **8-bits** in each byte, each byte has a much larger range of possible values that can be assigned into that memory location:

* Examples: 00000000 (0) all the way to 11111111 (255).

**C Program & Compiling**

Arguments

Things you pass into functions are called arguments.

* Some functions require multiple arguments, separated by commas. E.g. int main(int argc, char \*argv[])

Variables

A variable (a placeholder) stores a piece of information that we can manipulate.

* The computer assigns variables a specific amount of memory addresses inside the memory table, depending on the variable type.
* We can access VALUES STORED in variables by referring to the variable by NAME or ACCESSING ITS ADDRESS (pointers)

Constants and EXIT\_SUCCESS

Constants can be thought of as a storage of data, which are #define’d at the beginning of the file and don’t change throughout.

* Constants are written in uppercase letters separated by underscores e.g. EXIT\_SUCCESS

EXIT\_SUCCESS is a constant, with the value of 0, which is the integer value that tells the computer that the program ran successfully.

* What is the difference between **return 0** vs. return **EXIT\_SUCCESS**?
  + Main returning 0 simply tells the system that the program ran without errors
  + EXIT\_SUCCESS is defined in <stdlib.h> so the compiler will change EXIT\_SUCCESS to a zero wherever it occurs.
  + EXIT\_SUCCESS is better because if we were using a non-compliant standard machine which had a different return value to indicate that a program ran successfully, then the constant stored in EXIT\_SUCCESS can be changed such that the program still returns the correct value to say that the program ran with no errors.

Compiling

EXAMPLE: gcc –Wall –Werror –O –o hello HelloWorld.c

* **-Wall** = tell me all the warnings
* **-Werror**  = turn the warning into an error and abort compiling process (so we can fix the code)
* **-O**  = terminal optimiser (more warnings and errors)
* **-o** **hello**  = name the program (in this case the program’s name is “hello”)
* **HelloWorld.c** = name of the C source code to compile

**Variables – Declaration, Initialisation, Assignment**

Variables

A variable assigns a name (an **identifier**) to a specific memory location, allowing you to access / change data stored at that location by referring to the identifier, rather than the memory location manually.

* **DECLARATION**: States the TYPE of a variable, along with its NAME/ IDENTIFIER
* **INITIALISATION**: A special type of assignment, THE FIRST. Before initialisation, vars have a NULL value.
* **ASSIGNMENT**: Throwing away the old value of a variable and REPLACING WITH A NEW VALUE.

**Big Ideas**

Concurrency

The property of algorithms is to do many things at once, independent of each other, so they can be run across cores or even run across entire computers.

Preconditions & Asserts

**Preconditions** are conditions which an algorithm assumes to be true.

**Assert** is a very simple way to abort a program if a precondition is not upheld, which means the program will generate error while compiling, as the program is only guaranteed to be correct within an assertion that the programmer made.

* **#include <assert.h>** to have inserts in your program
* You should place asserts right after an input line (e.g. scanf) or at a location where the asserted condition is liable to change, to ensure that the preconditions are met. Placing an assert beforehand will not affect code that appears after.

Irrationality

Computers generally store numbers as fixed-width binary numbers, its often too computationally expensive to accurately represent **irrational numbers** (number that can’t be expressed as a fraction / ratio of integers), so instead, most software just uses a **floating point approximation** (approximation of a real number, to support a trade-off between range and precision).

* However, it is important to note that **rounding errors can add up** and impact results if a programmer isn’t careful.

**Functions**

Church Turing Hypothesis

A function is a **computable function** (ignoring time and resource constraints), if it is computable by a **Turing Machine**

* A Turing Machine is an abstract model of a computer, which can determine a result from a predefined set of rules + set of input variables.

Function structure

A function is structured as follows:

* functionType functionName( input1Type intput1Name, input2Type input2Name ) {

// function code  
return outputValue;  
}

**Bits, Binary and Hexadecimal**

Binary

We need 2 different digits for Binary (**Base 2**)

* 0 and 1

Hexadecimal

We need 16 different digits for Hexadecimal (**Base 16**)

* DECIMAL: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
* HEX: 0 1 2 3 4 5 6 7 8 9 A B C D E F
* For HEX, 2 hexadecimal digits can be stored in ONE BYTE (4 bits per hex digit)

Converting FROM Base 10 (Decimal)

EXAMPLE: Convert the number 1337 from decimal to hexadecimal (base 16)

* The largest exponent of 16 that’s smaller than 1337: **162 = 256**
  + Divide 1337 by 256 = **5** remainder 57
  + Divide 57 by 16 = **3 r**emainder 9
  + Remainder **9**
* Therefore 1337 in HEXADECIMAL = **5 3 9**

Converting TO Base 10 (Decimal)

EXAMPLE: Convert 1010 from binary (base 2) to decimal

* Write up the powers from the original base:  
  23 = **8** 22 = **4** 21 = **2** 20 = **1**

**1 0 1 0**

* Multiply them with the corresponding ONE’s or ZERO’s then SUM EVERYTHING UP:

(8 x 1) + (4 x 0) + (2 x 1) + (1 x 0) = **10**

**Memories and Types**

Storing values in Microprocessors

* Each value is stored as a **binary number** **in a “byte”**
* The **40XX series has 4 bits** (binary digits) to a byte. So the byte can store numbers from **0-15**
* The **80XX series has 8 bits** to a byte. So the byte can store numbers from **0-255**
* The numbers do not always literally mean that number:
  + E.g. The number 1 could either be the number 1 OR an instruction to do something.

Data Types

An **INT** is an **integer number**

|  |  |
| --- | --- |
| **INT TYPE** | **PROPERTIES** |
| **INT** | * Generally 4 bytes * Range from ??? Or does it depend on unsigned / signed |
| **UNSIGNED INT** | * Range from **0 to 232 - 1** |
| **SIGNED INT** | * Range from **- 231 to 231 - 1** |
| **SHORT** | * Generally2 bytes * Range from **0 to 215 - 1** |
| **LONG** | * Generally same size as a standard int: 4 bytes |
| **LONG LONG** | * Generally 8 bytes * Range from **0 to 263 - 1** |

* The only guarantee with these is that *short ≤ int ≤ long ≤ long long*
* Generally ONLY USE INT unless one of the others is specifically needed
  + Microprocessors can generally only work with an int at a time so using another size results in either multiple operations needed or useless data being moved.

A **FLOAT** is any **real number**

* Float is short for floating-point, which is a number with a “floating point . “
  + The point “floats around”:
  + E.g. 1234 🡪 1.234 🡪 123.4
* A float is stored in **32 bits**
  + E.g. 1.234 = 1234 x 10-3
  + **8 bits** are used to store the EXPONENT (position of the decimal point)
  + **1 bit** is used to store the SIGN
  + The **leftover bits** are used to store the number

A **DOUBLE** uses double the number of bytes as a float.

* It is generally preferred to be used over a float, as precision is important.
* **PUT INFO ABOUT USING DOUBLE VS FLOAT**

Numbers can be SIGNED or UNSIGNED

* **Signed numbers** can use one bit for whether the number is negative or positive
  + Encoded using “2s complement”:
  + To get –x, take the binary version of x, invert all 0’s and 1’s then add 1.
* **Signed:** +ve or –ve **Unsigned:** only +ve
* Unsigned numbers use all of the bits for the number itself, so can store larger values than signed, but cannot store negative numbers.
* The default is SIGNED for most numbers, but you can add SIGNED or UNSIGNED before the normal type to force it to be signed or unsigned. This only applies to INTEGER TYPES, not floats and doubles.

A **CHAR** is an ASCII character (letter, number, punctuation etc.).

* It’s also defined to be **exactly one byte**, which has a **range from 0 to 255 values** on almost every computer.

**NO BOOLEAN TYPE** **(true/false)** in C.

* INT is used instead, where 0 = false, any other number = true.

**Memory Types #2**

Void

A **VOID TYPE** is a type that is essentially nothing. It means that the function will not return anything.

* A void function means that the function will not return anything to it’s caller.
* A function with void in the brackets tells C not to take in anything, or no inputs.

EXAMPLE:  // A declaration of functionName, which has no inputs or outputs

Ints

Most computers store int in 4 bytes, using **base 256**.

* E.g. For the number 1234, 256 goes into it 4 times, which gives us 1024 r. 210.
* Therefore, it will store |0|0|4|210| (cells 1 to 4)
* **Each cell is** **256^n, where it starts at n = 0 from the right.**

**% Format Specifiers**

Format specifiers are operators used in printf() function to retrieve data/values stored in variables.

* Format specifiers start with a **%** symbol and follows a **special character to identify the** **type of data**.
* There are basically six types of format specifiers that are available in C:

|  |  |
| --- | --- |
| **Format Specifier** | **Description** |
| **%d** | * **Signed integer value** * Printf(“%d“, <variable name>); |
| **%f** | * **Fractional values / Floating point numbers** * Printf(“%f”, <variable name>); |
| **%c** | * **Single characters** * Printf(“%c”, <variable name>); |
| **%s** | * **Strings** (a sequence / string of single characters) * Printf(“%s”, <variable name>); |
| **%u** | * **Unsigned integer value** * Printf(“%u”, <variable name>); |
| **%ld** | * **Long integer value (similar to just int)** * Printf(“%ld”, <variable name>); |
| **%p** | * **Pointer address** * Printf(“%p”, <variable name>); |
| **%x** | * **Hexadecimal value** * Printf(“%x”, <variable name>); |

**Variable Scope**

The scope of the variable is the area where a variable exists.

* Variables only exist inside the function or main that they are declared within.
* Those variables can only be be manipulated / called in that specific function
* When that function ends, all the variables in the scope are thrown away.

In C, the two main restrictions that concern variables are:

* Variables are bounded by scope (curly braces create new scopes)
* Variables must be declared before being used

**Local variables**: A variable declared inside functions or within a block of code e.g. *while loop* or an *if statement.*

* Local variables can only be used by statements that are also within the function / block

**Global variables**: A variable declared outside of every function.

* Global variables can be accessed by ANY function or block of code within the program.

**Scanf** can read in a value within main and pass it into a LOWER FUNCTION or it can read in the function and use the value directly.

Consider the scope of variables: think about if they only have meaning INSIDE a particular f’n vs. USED SEVERAL TIMES?

**Side Effects**

A function is said to have side-effects if it **takes an input other than the inputs given to it as parameters**, or **produces output other than the output it gives to the caller.**

**Function Design**

**Top-Down Design** is defining the outer structure/ system of the program first and breaking it down to smaller problems which gives insights into the sub-parts. This process is iterated until eventually you’re able to solve each of them with little difficulty.

* Make all the initial functions that need to be done
* Create the next set of sub-functions and anything that follows.

**Advantages** of breaking down a big problem into smaller parts:

* Improves legibility of code for humans
* Better reliability compared to a big junk of complex code
* Easier to check for errors / maintenance 🡪 You can test each smaller part separately + easier to identify code issues

Things you can do:

* If you have written a long function, go back to it and break it down into smaller functions
* It is important to find the easiest method to solve something. Complexity can cause error in code
* If your function doesn’t fit the screen, it should be split.
* Every function should do one thing and this should be specialised.
* Get a big picture of the program then relax 🡪 Break it into smaller problems + keep doing it and eventually the problem will be small enough to solve it right away.

**Bitmap Images**

Bits can be interpreted in various ways, from ints, to characters, to hexadecimals or even COLOURS.

They have no initial meaning.

* A **bitmap graphic** is composed of many tiny PIXELS.
* Each of these pixels are represented by BLUE, GREEN and RED intensities.
* It is possible to edit each individual pixel to change its colour.

Bitmap Files

A bitmap is simply a collection of 0’s and 1’s, which are grouped into EIGHT BITS (each group is = one byte).

* In a hex editor, it displays the bytes as their numeric value (in hex notation) rather than characters.

A bitmap file structure:

* **HEADER (14 bytes)**
  + Altering bytes *0 – 13* may **corrupt the file** or cause unexpected errors
  + Bytes *0 and 1* correspond to the **file type** (BM)
  + Bytes *2 – 5* refer to the **size of the file**
  + Bytes *6 – 9* are reserved for the **application that generates the Bitmap file**
  + Bytes *10 – 13* store the **location of the start of data** (where the Pixel Array begins)
* **DIB HEADER (Size varies: Richard’s was 40 bytes)**

Data here contains various information about the image:

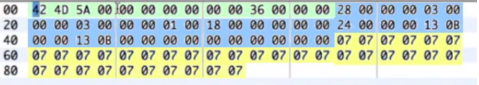
* + *First 4 bytes* are assigned to the **size of the header**
  + *Next 8 bytes* correspond to the **bitmap dimensions in pixels (4 bytes for width / 4 bytes for height)**
  + *Next 2 bytes* correspond to the **colour plane**
  + *Next 2 bytes* correspond to **bit depth**. E.g. if bit depth = 18 (hex 🡪 dec) 24 bits = 3x8 bit colour. (RGB)
  + *Next 4 bytes* are for **compression**
  + *Next 4 bytes* correspond to **size of the pixel array data, including padding** (padding = make image rectangular)
  + *Next 8 bytes* are for **resolution dimensions (4b for horizontal / 4b for vertical)**

NOTE: bitmap / resolution dimension are different.  
🡪 Bitmap dimension = # of pixels width/height of the entire image  
🡪 Resolution dimension = # of pixels width/height per unit of length. E.g. pixels per inch (ppi) measurement

Higher res = smaller pixels per unit, Smaller res = larger pixels per unit

* + *Next 4 bytes* are for the **number of colours in the palette**
  + *Next 4 bytes* are the **number of important colours**
* **DATA / PIXEL ARRAY**
  + The image’s pixels will be stored here – length depends on resolution of the image.
  + The total amount of bytes necessary to store an array of pixels is given by:  
    **ArraySize = RowSize x |Image Height|** , where image height is expressed in # of pixels.
  + Pixels are stored in values of RED, GREEN, BLUE.

Bitmap file in a HEX EDITOR



* GREEN = header / BLUE = DIB header / YELLOW= Data or Pixel Array
* First pixel **begins from the bottom left**
* A row of pixels must start on a **4 byte boundary** (convenient for memory purposes)
  + There are **3 pixels in each row** and **each pixel takes 3 bytes = 9 bytes used to store a row of pixels**
  + Since there is a 4 byte boundary, the last 3 bytes of each row are all **padding**

Storing Colours on a BMP file

Colours are represented and stored on the file using **3 bytes**

* They follow the structure BLUE, GREEN, RED (as opposed to RGB)
* Each pixel uses **24 bits** to store its **3 colours** (1 byte / 8 bits per colour)

Colouring the pixels:

* Go from values 0 – 255 in HEX
* Using | 00 00 00 |for each byte will give you **black**
* Using | FF FF FF | for each byte will give you **white**
* Lower numbers = darker shade  
  Higher numbers = lighter shade

**Addresses & Pointers**

Pointers

A pointer is a variable that contains a memory address.

Example:

int a;

int b = 10;

float c = 5.8;

printf (“address of a is at %u \n”, &a); *note: %u = unsigned integer (always shows positive interpretation, -ve🡪+ve)*

printf (“address of b is at %u \n”, &b);

printf (“address of c is at %u \n”, &c);

int a has no data, is at memory location xxxxxx1 (takes 4 bytes)

int b has value of 10, is at memory location xxxxxx2 (takes 4 bytes)

float c has value of 5.8, is at memory location xxxxxx3

Their values are actually stored as binary, in zeros/ones.

-----------------

int p1; *a variable*

vs.

*(declaration)*

**int \*p1;**  *becomes a pointer variable* *(contains a memory address)*

*(initialising)*

**p1 = &a;**  *“pointer p1 contains the address of a” the address of a = the memory location stated above*

*(declaration + initialising)*

**int \*p2 = &b;** *“pointer p2 contains the address of b”* *same as the above statement*

**float \*p3 = &c;** *“float pointer p3 contains the address of c”*

-----------------

Putting the \* asterisk before p1 means that I am trying to access the value/data from the memory location, which the address is stored in pointer variable p1.

**Therefore, if you want to access the value of a variable, simply put an \* asterisk sign before the name of the pointer.**

**printf (“the value of p1 is %u”, p1);** *prints the value of pointer p1, which is the memory address of a*

**printf (“the value of a is %d”, \*p1);**  *prints the value of a, the data value of variable a*

**printf (“the value of p3 is %u”, p3);** *prints the value of pointer p3, which is the memory address of c*

**printf (“the value of c is %f”, \*p3);**  *prints the value of c, the data value of variable c  
note the use of %f because variable c is of type float*

-----------------

Sum using pointers

printf (“the sum of a and b is %d”, a + b); *summing a and b WITHOUT pointers*

printf (“the sume of a and b is %d, \*p1 + \*p2 ); *summing a and b WITH pointers*

-----------------

Changing values using pointers

p1 = p2; *assign p1 the memory address value of p2/ p1 stores address of b*

Printf (“value of p1 is %u \n”, p1); *value of p1 = memory address p2 (after assigning p1 = p2)*

Printf (“value of b is %d \n, \*p1); *value of b = 10, using \*p1 (as p1 now has the same address as p2, storing value of b)*

-----------------

Extra notes on pointers

Pointers are variables, so space is allocated for them + they have their own memory addresses.