**MODULE 0, 1, 2, 3 – INTRODUCTION**

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.

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 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.

**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 permutations (Numbers 0 – 15)

For a chip that has **8-bits** in each byte, there are 256 different permutations (Numbers 0 – 255)  
Incrementing maximum values by 1 will cause the stored value to wrap around to ZERO.

**C Program & Compiling**

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.

* EXIT\_SUCCESS is a constant (value of 0) which is the int value telling 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 some machines may return a different value than zero to indicate success.

Compiling

* **-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”)

**Bits, Binary and Hexadecimal**

Binary = **Base 2** (0 and 1)  
Decimal = **Base 10** (0 to 9)

Hexadecimal = **Base 16** (0 to 15)

* 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** 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**

Data Types

A **FLOAT / FLOATING POINT NUMBER** is any **real number**

* It is a number with a point that “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 (Also a long float)** uses double the number of bytes as a float: (same as a float, except double # of bytes)

* It is generally preferred to be used over a float, as precision is important.

**SIGNED** or **UNSIGNED** numbers:

* 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.
* 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.

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 Specifier** | **Description** | **Decimal / Notation Values**  **(Assuming a 32-bit processor)** |
| **%d** | * SIGNED integer value | * **From - 231 to 231 – 1** * **32 bits = 4 bytes** |
| **%u** | * UNSIGNED integer value | * **From 0 to 232 – 1** |
| (1) **%ld**  (2) **%lld** | 1. Long integer value (same as int) 2. Long Long SIGNED int | 1. **Same as signed int** 2. **64 bits = 8 bytes** |
| **%s** | * String (a sequence / string of single characters) | * **Depends on number of chars** |
| **%c** | * Single characters | * **1 char = 1 byte** |
| (1) **%f** (2) **%lf** | 1. Fractional values / Floating point numbers 2. Long floating point / Double | 1. **32 bits = 4 bytes** 2. **8 bytes / 0 to 263 - 1** |
| **%p** | * Pointer address |  |
| **%x** | * Hexadecimal value | * **0 – 255 (dec) OR 00 – FF (hex)** * **Two hex digits = 1 byte** |

**Bitmap Images (ignore if no past papers have questions on this)**

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

* 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.

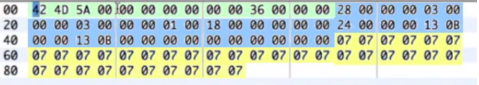
A bitmap file structure:

* **HEADER (14 bytes)**: Contains file type, size of file, generation of the Bitmap, location for start of data.
* **DIB HEADER (Size varies – Richard’s was 40 bytes)**: Contains various information about the image, such as size of header, bitmap dimensions, compression, size of pixel array data including padding, resolution dimensions
* 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

* **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 (left 🡪 right, bottom 🡪 top)**
* 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:

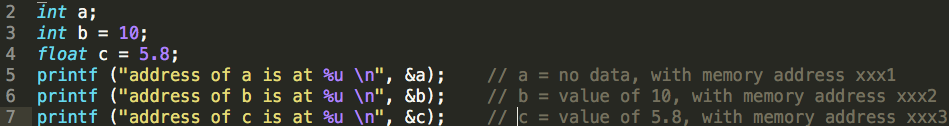
* Go from values 0 – 255 in HEX
* Using | 00 00 00 |for each byte will give you **black**
* Using | FF FF FF | for each byte will give you **white**
* Lower numbers = darker shade  
  Higher numbers = lighter shade

**Addresses & Pointers**

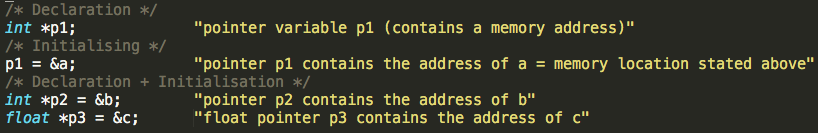
Pointers

A pointer is a variable that contains a memory address. Example:

ASSUME:

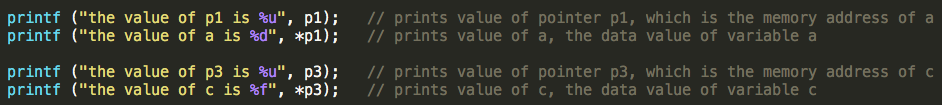


SO THAT:



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.

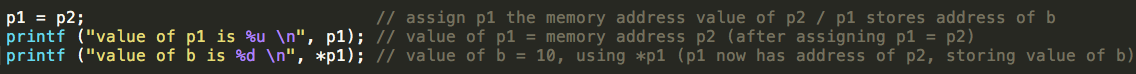
**Therefore, if you want to access the value of a variable, simply put an \* asterisk sign before the name of the pointer.**

****

Sum using pointers



Changing values using pointers



**MODULE 3, 4 – LOOPS (CHAR ENCODING, ROWS&COLS) + HTTP, STRINGS, ARRAYS**

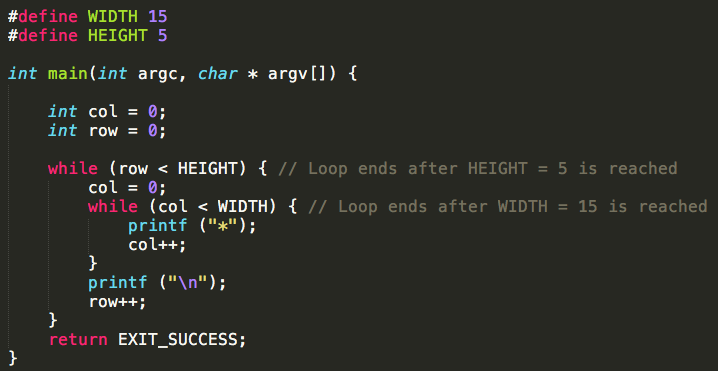
**Loops**

Char Encoding

**Nested Loops**

Rows & Columns

Printing a picture of a flag



**HTTP – Hypertext Transfer Protocol**

A web address entered into a browser can be considered a function.

* By entering this address (i.e. [www.google.com)](http://www.google.com)) a request **(“HTTP GET request”)** is sent to a web server (Google), which then processes this request and sends a response back **(“HTTP response”)** which is the output, a webpage.
* (IPv4) IP addresses range from 0.0.0.0 to 255.255.255.255 (4 byte / 32 bit addresses)

In addition to an IP address, we need to know the **port number** to use in order to communicate with a device on the internet.

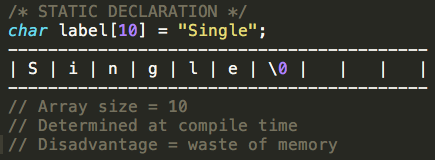
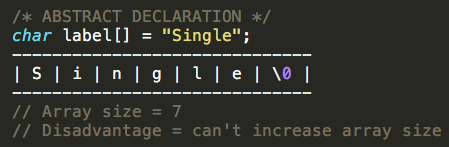
* Port numbers are **16-bit unsigned integers** = ranges from 0 – 65535
* E.g. Command to access a page: **telnet [ip number] [port number]** / telnet www.website.com 80

HTTP is a protocol designed so that browsers (clients) and servers have a standard way to transfer Hypertext from each other.

* **HTML is Hypertext Markup Language**, which adds tags to text, so that browsers can format, modify and process text and add things such as links and images.

**Strings**

Strings as ARRAYS



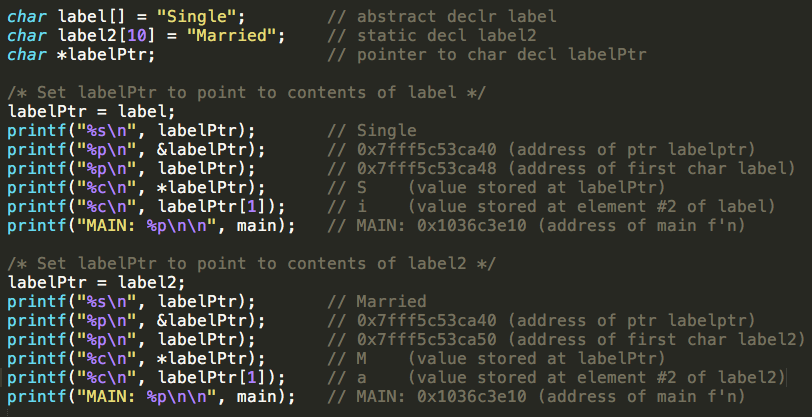
You can also initialise as **char label[7] = { ‘S’, ‘I’, ‘N’, ‘G’, ‘L’, ‘E’, ‘\0’ }**

* This “character by character” initialisation method requires inserting a **NULL TERMINATOR**
* You can initialise as a **string literal** automatically include the null terminator (see below)

Strings as POINTERS

Pointers only hold an address, they cannot hold all characters in a char array.

* For pointers to strings, they hold the address of the **first character of an array**
* When we use char \* to keep track of a string, the char array containing the string must already be declared.
* Disadvantage of using pointers rather than arrays is that you can’t change the string with just pointers.



String LITERALS

**String Literals** are when **double notations (“insert text here“)** are used to initialise a string.

* They are automatically **NULL TERMINATED**

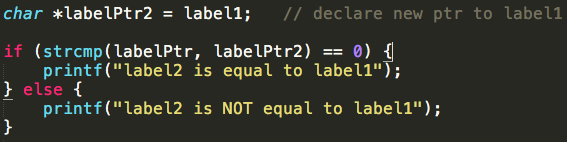
**String Function <string.h>**

strlen( )



strcmp(char \*str1, char \*str2)

**Strcmp** compares string 1 with string 2:



strcpy(char \*destination, const char \*src)

**Strcpy** copies the string source to the destination, then returns destination.



**Arrays**

When declaring + initialising arrays, you are storing the **memory address of the start of the array = array[0].**

* After initialising an array, you cannot make it bigger in C, because you may overwrite other data.
* Arrays are stored by calculating the amount of memory needed for the array, then skipping to the end for the next set of data. Example:
  + **Array Size (bytes) = [number of elements] x [# of bytes per data type]**
  + I.e. 4 BYTES / INT or 1 BYTE / CHAR

Buffer Overflow

**Buffer Overflow** is when you overwrite over the end of an array.

* There is a **5 MARK PENALTY PER BUFFER OVERFLOW IN THE FINAL EXAM**
* Protect against buffer overflows by using **#define SIZE\_OF\_ARRAY** then declare as such: **int array[SIZE\_OF\_ARRAY];**
* Use **<assert.h>** to assert that the max array size isn’t breached.

**MODULE 5, 6, 7 – STRUCTS, STACK vs HEAP, ADTs**

**Structs & Typedefs**

**Structs** are compound data types which allow storage of multiple variables under a single name – used to abstract data.

**Typedef** are used to create an alternative name for a data type.

* 
* Typedef is executed by the compiler, #define is executed by the pre-processor.

**Stack vs Heap**

**Stack** is a region of your computer’s memory that stores temp variables created by each function.

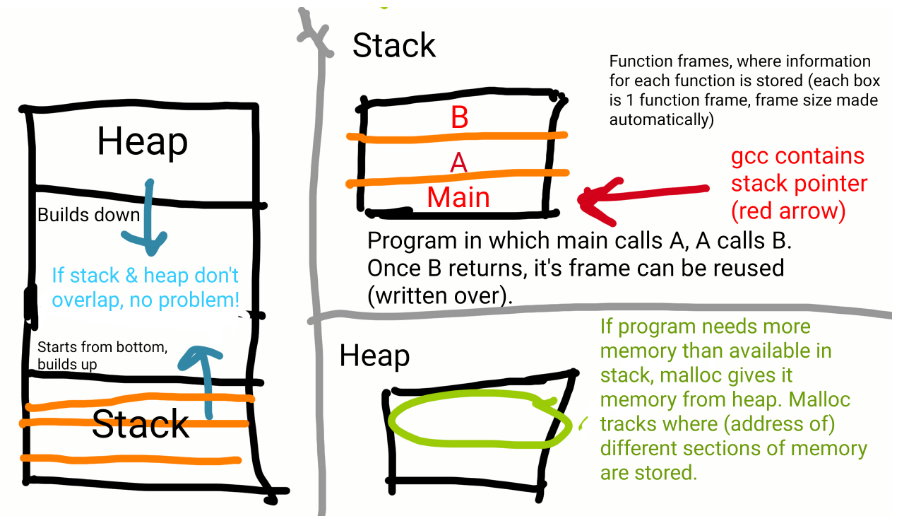
* It uses a **FILO (First In, Last Out)** data structure
* Every time a function declares a new variable, it is “pushed” onto the stack.
* Every time a function exits, all variables pushed on the stack are deleted / popped off the stack.
* Once deleted, that region of memory becomes available for other stack variables.

**Heap** is a region of your computer memory that is free-floating and larger than the stack.

* You need to allocate memory on the heap using **malloc(sizeof(xxx))**
* You then need to free the allocated memory using **free(xxx)** and set the **ptr xxx = NULL**.
* This will prevent a **memory leak**

Use the HEAP when you need to allocate a large block of memory and need to keep it around for a long time (such as a global)

|  |  |
| --- | --- |
| **STACK PROS & CONS** | **HEAP PROS & CONS** |
| * Very fast access * Don’t have to explicitly de-allocate variables * Space is managed efficiently by CPU, memory will not become fragmented * Local variables only * Limit on stack size (OS-dependent) * Variables cannot be resized | * Variables can be accessed globally * No limited on memory size * Relatively slower access * No guaranteed efficient use of space, memory may become fragmented over time as blocks of memory are allocated, then freed. * You must manage memory (you’re In charge of allocating and freeing variables) * Variables can be resized using **realloc()** |



**ADTs (Abstract Data Types)**

ADT is a logical description of how we view data and the operations that are allowed, without regard to how they will be implemented.

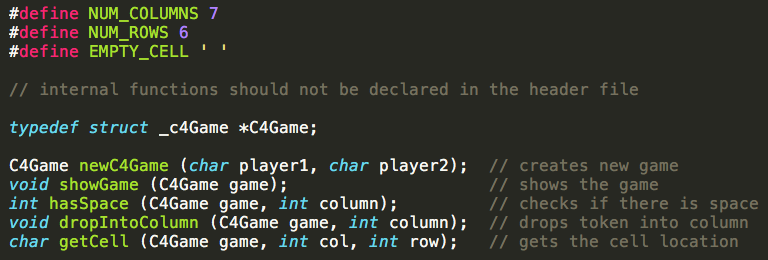
* We are only concerned with what the data is representing and not with how it will eventually be constructed.
* This level of abstraction allows an **encapsulation around the data**, **hiding it from the user’s view**.

Example of using ADTs:

* Order of creation program.h 🡪 testProgram.c 🡪 program.c
* **Program.c** 🡪 Implementation of the program
  + Defines the type and the functions + Implements the functions
  + Doesn’t contain a main
* **Program.h** 🡪 Interface of the program
  + Doesn’t contain a main
* **testProgram.c** 🡪 Test file for program.c
  + Contains a main to test program.c

**MODULE 8 – EXAMPLE GAME: CONNECT 4**

**Connect4.h INTERFACE FUNCTIONS EXAMPLE**



**Connect4.c PARTIAL EXAMPLE (new game function)**

