Week 1

C files are intended for humans to read. It doesn't require an interpreter or virtual machine.
Compilation: to convert our human readable C code to machine level code that can be fed directly to the CPU.

GCC stands for GNU Compiler Collection and is the compiler which converts our code first to assembly, then pure binary.
Note the `-o` flag to indicate that we want a binary in the end and the name of the binary `hello` after it.

Without the `-o` flag, it would produce a `.out` file by default.

- How to compile: `gcc hello.c -o hello`
- How to run: `./ hello`

*hello.c

`#include <stdio.h>

```
int main() {
printf("Hello World!\n");
return 0;
}
```

- man: view user manual. e.g. `man 3 printf`

C Types
C is a statically types language - all variables must have a type associated with it, including:

- *char and unsigned char*
- *int and unsigned int*
- *short and unsigned short*
- *long and unsigned long*
- *double*
- *float*
  Each type is defined by the CPU architecture, platform and compiler. e.g. An int may be 4 bytes on one architecture, but 8 on another. **Always use int32 (4 bytes) or a specific bit number.**
- Include *stdint.h* to use integer types having specified widths: `#include <stdint.h>`

C exposes unsigned types to eliminate the usage of the sign bit as part of the type encoding. **C provides access to memory allocations using array and pointer types. To declare an array in C you need to specify the size.**

Pointers

```c
int x = 42;
int *ptr = &x;       // ptr now holds x's address

printf("%d", x);
```

- `*` is used to **declare a pointer** (`int *ptr`)
- `*ptr` **dereferences** the pointer to access the value at that address
- `ptr` = the address, `*ptr` = the value at that address

`int*` means pointer to an int.
`**` is a double pointer

- To get the ADDRESS of the pointer, `&`
- To get the VALUE of the point, `*`

```
#include <stdio.h>

int secret_location_gold = 10;
int* secret_map = &secret_location_gold;

int main() {
    int wallet = 0;
    int** map_to_secret_map = &secret_map;

    wallet += **map_to_secret_map;
    **map_to_secret_map -= wallet;

    printf("There are %d gold pots on the island\n", secret_location_gold);

    printf("There are %d gold pots in my wallet\n", wallet);
}
```

- Arrays automatically decay to a pointer to their first element.

I/O

In C, there is no String type. This means that all strings are represented as an array of characters. The characters exist in a contiguous area of memory. We know when a string ends when there is a null terminator `\0` at the end of the array.

To be able to read input, we need to create empty values first in memory and provide the functions with a pointer to our memory. This way the functions can follow our map and modify the values for us. This is a key pattern in C that will be recurring.

The `scanf()` function reads input from the standard input stream stdin and `fscanf()` reads input from the stream pointer stream.
https://www.w3schools.com/c/ref_stdio_scanf.php

```
int scanf(const char *restrict format, ...);
       int fscanf(FILE *restrict stream,
                  const char *restrict format, ...);
```

`fgets()` reads in at most one less than size characters from stream and stores them into the buffer pointed to by s. Reading stops after an EOF or a newline. If a newline is read, it is stored into the buffer. A terminating null byte ('\0') is stored after the last character in the buffer.

```
#include <stdio.h>

int main() {
    char buff[100];
    int n = 10;

    printf("Enter a string: \n");

    // Read input from the user
    fgets(buff, n, stdin);
    printf("You entered: %s", buff);
    return 0;
}
```

```
// TODO create me!
#include <stdio.h>

int main() {
    char msg[100];
    int n = 100;

    while (fgets(msg, n, stdin)!= NULL) {
        printf("%s", msg);
    }
    return 0;
}
```

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


int main(int argc, char *argv[]) {
    char msg[100];
    int n = 100;

    printf("What is your name? ");

    fgets(msg, n, stdin); // read n characters and store in msg array.
    printf("%s ", argv[1]);
    printf("%s", msg);
    return 0;
}
```

- argc: arg count
- argv: an array

gets - Read a line from a channel

- `gets channelId ?varName?`

scanf() examples

```
// Create a string
char firstName[30];

// Ask the user to input some text
printf("Enter your first name: \n");

// Get and save the text
scanf("%s", firstName);

// Output the text
printf("Hello %s", firstName);
```

```
// Create an int and a char variable (somewhere to store the text inputs)
int myNum;
char myChar;

// Ask the user to type a number AND a character
printf("Type a number AND a character and press enter: \n");
```

```
// Get and save the number AND character the user types
scanf("%d %c", &myNum, &myChar);

// Print the number
printf("Your number is: %d\n", myNum);

// Print the character
printf("Your character is: %c\n", myChar);
```

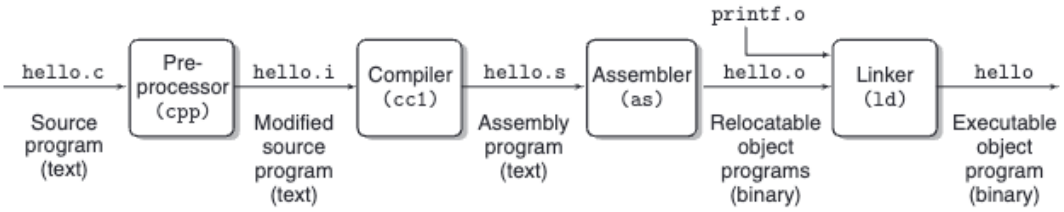
Tute notes

https://comp2017.stvya.com/wk2a/1

LECTURE NOTES

Text_Book_cs2017, page 33 **Section 1.1, 1.2 and optional 1.3 - 1.6**

hello.c is stored in a file as a sequence of bytes, which each has an integer value.



Deductions for bad practices: memory errors/leaks, Variable Length Arrays (VLA), unclean repositories, failing to compile/execute, excessive memory/processing, *fake* solutions, dependence on external non-authored code, not following other restrictions.

You may be asked to explain your code to the tutor or lecturer. AI is not a proxy for your knowledge.
What you need to consider for these assessments:

- the code must compile and execute with the simplest test cases
- you have handled the error cases presented in the description
- you have tested your own code
- invalid assumptions are void, especially where it trivialises the solution. It cannot be considered. Alway ask.
- if there are issues such as off by one in the code, logic errors. These are not assessed manually.
- there will be private test cases that will run on your code
- the code is clean, legible (refer to the coding style guidelines)

- Machine code is decoded by the cpu; the code depends on the architecture the cpu provides. So if you write it in assembly, but want to move cpus, you would have to rewrite everything in the new language --> C is agnostic to the cpu architecture.
- C compiler --> abstract syntax tree --> different back ends

History of C:

- UNIX was first written in low-level PDP-7 assembly
- Ken Thompson invented B on BCPL to overcome issues
- Dennis Ritchie build on B and called his language C.
- C-compilers employ two languages: preprocessing language: text-macro, actual c-language: high-level programming language.

C-language:

- Mainly used for: systems software (OS, embedded systems, etc.), software that needs hardware interaction, application programming, science/engineering.
- C-compilers exist for almost all cpu architectures.
- Does not have the features: objects and classes, templates, operator/function overloading
- C++ overcomes this - a successor of C
- Writing a non-optimising c-compiler is straightforward.
- printf() does not add a new line

```c
#include <stdio.h>


int main(int argc, char **argv)
{
 int      ftemp;   /* the fahrenheit temperature */


 printf("Please enter a fahrenheit temperature");
 scanf("%d", &ftemp);
 printf("%d fahrenheit is %d centigrade", ftemp,
                          (ftemp - 32) * 5 /  9);
 return 0;
}
```

- C also has header files: declaration for variables and functions
- `#include <stdio.h>` imports the standard I/O header
- Arrays can be handled with pointers
- Arrays can be created and initialised in declaration
- C strings are just arrays (with termination character)
  - no garbage function, have to clear out memory manually
- **sizeof** operator
- create dynamic data structures with malloc()
- C allows declarations only at block start - Only true for pedantic compilers these days

Functions

C Intro, page 21

- `int main()` is the first function invoked (where the program starts)