# Tut 1

## Topic 1: C language

### General structure of a C program

1. Documentation section -> Author of the program, description of the program, input,output etc
2. Link section -> Including Header files
3. Definition section -> Defining constants using `#define`
4. Global declaration section -> Declare global variables and user functions
5. Main function section -> Contains the main function
6. Sub program section -> Define user functions

### Headers

- A header file is a file which contains C function declarations and macros definitions
- Extension is `.h`
- Including a header file is equal to copying the content of the header file but we do not do so since:
  - Bigger code base is harder to debug in a single file
  - If you copy the header code and make changes in it, you will have to make the changes in other files as well
- Advantages of using headers:
  - Speeds us compile time - Can compile and check parts of code rather than a single huge code. Can compile different small files to object code and link them all together in the end (What object codes are will be explained in compilation process)
  - Keeps code more organized since we are going to keep code related to a single concept in a single file
  - Seperates interface from implementation - Allows you to change the implementation without affecting the entire system.
- Difference between `#include <header>` and `#include "header"`
  - `#include <header>` searches for header files in a standard list of system directories
  - `#include "header"` searches for header files in the current program's directory

#### Macros

```
#ifdef MACRO
//conditional codes
#endif
```

```
#ifndef MACRO
//conditional code
#endif
```

```
#if expression
//conditional codes
#endif
```

```
#if expression
// expression = 1 conditional codes
#else
// expression = 0 conditional codes
#endif
```

```
#if expression1
// expression1 = 1 conditional codes
#elif expression2
// expression2 = 1 conditional codes
#elif expression3
// expression3 = 1 conditional codes
#else
// all expressions = 0 conditional code
#endif
```

```
#defined MACRO
```

Checks if `MACRO` has been defined, used with `#if` statements

```
#define PI 3.14
#define circleArea(r) (PI*(r)*(r))
```

### Header example

#### hello.h

```
#ifndef HELLO_H
#define HELLO_H
void hello(char *A);
#endif
```

#### hello.c

```
#include <stdio.h>
#include "hello.h"

void hello(char *A) {

```

```
    printf("Hello %s", A);
}
```

**main.c**

```
#include "hello.h"

int main() {
    hello("World");
}
```

Compiled using `gcc -o outputname main.c hello.c`

## Compilation process

1. Preprocessor -> Expands code (MACROS, includes), removes comments (still .c)
2. Compiler -> Converts the expanded code into assembly code (syntax checking happens here) (converted to .s)
3. Assembler -> Converts assembly code to object code (converted to .o for linux systems)
4. Linker -> Links the various object codes (both user and standard library) together to form an executable

### Implications

### Case 1:

**hello.h**

```
#ifndef HELLO_H
#define HELLO_H
void hello(char *A);
#endif
```

**hello.c**

```
#include <stdio.h>
#include "hello.h"

void hello(char *A) {
    printf("Hello %s", A);
}
```

**main.c**

```
#include "hello.h"

int main() {
```

```
    hello("World");
    printf("This will compile with warnings and since it finds printf during
linking");
}
```

**Case 2:**

**hello.h**

```
#include <stdio.h>
#ifndef HELLO_H
#define HELLO_H
void hello(char *A);
#endif
```

**hello.c**

```
#include "hello.h"

void hello(char *A) {
    printf("Hello %s", A);
}
```

**main.c**

```
#include "hello.h"

int main() {
    hello("World");
    printf("This will compile");
}
```

- Dynamic linking: Rather than copying the actual object code, it loads/links it during runtime. Saves space in executables. No need for theory, just remember to add flags while compiling (such as -lm)

---

## Pointers

- Stores the memory address of another variable
- Size depends on the system architecture (32 bit/ 64 bit) and not on the type of variable the pointer is storing the address of
- Declared using `*` operator
- Access value using `*` operator

```
int *p; // declaring a pointer
int a = 10;
p = &a // p now stores the memory address of a
```

```
int b = *p // b now has value stored at the memory referenced by p, which is 10
```

- In the above example, when p was declared it was a NULL pointer. A NULL pointer does not point to any memory location. Most of the times, it has the value 0, but it is not the same as a pointer whose value is 0.

## Strucs

- Collection of different variables in a single variable
- Defined using `struct` keyword

```
struct student{
    int rollnumber;
    char *name;
    char **courses;
}; // don't forget the semicolon
```

```
struct student{
    int rollnumber;
    char *name;
    char **courses;
} s1; // s1 is a variable of type student

int main() {
    struct student s2; // s2 is also a variable of type student
}
```

- You can use structs within a struct

```
struct course{
    char *course_code;
    char **instructors;
    int students;
};

struct student{
    int rollnumber;
    char *name;
    struct course *courses; // array containing courses of a student
}
```

- Access elements inside a struct using `.` or `->` operator

```
struct student{
    int rollnumber;
```

```
    char *name;
    char **courses;
} s1, *s2;

int main() {
    s1.rollnumber = 101;
    s2->rollnumber = 102;
}
```

- Note that `s2` is a pointer
  - `s2->rollnumber` is equivalent to `(*s2).rollnumber`
  - `(&s1)->rollnumber` is equivalent to `s1.rollnumber`
  - Rule of thumb, use `->` when working with pointers to structs and `.` when working with structs, helps you keep track of which variable represents what.

---

# Topic 2: Linked list and doubly linked lists

## Linked list (specifically Singly linked list)

- Data structure used to store data in a sequential order
  - Collection of various nodes which contain some data, and a link to the next node
  - Different from arrays which store data in sequential order by their physical placement in memory
- Advantages over an array?
  - Insertion and removal of nodes is better than arrays with respect to memory (In linked lists, you just have to manipulate pointers and not move anything, while in an array you have to move elements in the memory after inserting or removing elements)
- Disadvantages compared to an array?
  - No random access, in a linked list going to any index requires you to iterate over all the elements before it. In an array, you can directly access any element at any index
- You can modify/add new links to nodes for new ways to traverse (but then it is not a singly linked list, still a linked list)

## Doubly linked list

- Similar to singly linked list with the addition of a new link which points to the previous node
- We can traverse in both front and back directions, useful in systems where we can move from a state to an adjacent state on either side (back and forward button in web browsers, basic undo/redo systems)

- Reversal is extremly fast as you need to just change the root and update a boolean that represents which link is the forward link and which is the back link.
    - In singly linked list, you would have had to iterate over all nodes and change their forward links.
    - In an array you would have to copy and replace the elements while iterating over them

---

## Topic 3: ADTs

- Abstract data types (ADTs) are a way for abstraction and encapsulation
- It seperates interface from implementation, so a user can use functionalities related to an ADT without worrying about how it is implemented.
- In C, ADTs are implemented using structs
- In C++/Java, we use classes to implement ADTs
- Note that the implementations will differ due to the language, but the operations will remain the same
- Example of polynomial ADT:
    - question: How many parameters are required to define a polynomial of 1 variable
    - Representing the coefficients of a polynomial using an array
    - Function to multiply the polynomial with a scalar
    - Function to add two polynomials
    - question: How to subtract two polynomials?
    - question: function to multiply two polynomials
    - question: function to evaluate a polynomial at a point

---