File IO

- File is collection of data and information on storage device.
- Each file have data (contents) and metadata (information).
- File IO can enable read/write file data.
- File Input Output
  - Low Level File IO
    - Use File Handle.
  - High Level File IO
    - Use File Pointer.
    - Formatted (Text) IO
      - fprintf(), fscanf()
    - Unformatted (Text) IO
      - fgetc(), fputc(), fgets(), fputs()
    - Binary File IO
      - fread(), fwrite()
- File must be opened before read/write operation and closed after operation is completed.
  - FILE * fp = fopen("filepath", "mode"); – to open the file
    - File open modes:
      - w: open file for write. If exists truncate. If not exists create.
      - r: open file for read. If not exists, function fails.
      - a: open file for append (write at the end). If not exists create.
      - w+: Same as "w" + read operation.
      - r+: Same as "r" + write operation.
      - a+: Same as "a" + append (write at the end) operation.
    - File can be opened as text file (default or suffix "t") or binary (suffix "b").
    - Return FILE* when opened successfully, otherwise return NULL.
  - fclose(fp);
    - Close file and release resources.
- Character IO
  - fgetc(), fputc()
- String (Line) IO
  - fgets(), fputs()
- Formatted IO
  - fscanf(), fprintf()
- Binary (record) IO
  - fread(), fwrite()
- File position
  - fseek(), ftell()

Preprocessor Directives

- Preprocessor is part of C programming toolchain/SDK.
  - Removes comments from the source code.
  - Expand source code by processing all statements starting with #.
  - Executed before compiler

- All statements starting with # are called as preprocessor directives.
  - Header file include
    - #include
  - Symbolic constants & Macros
    - #define
  - Conditional compilation
    - #if, #else, #elif, #endif
    - #ifdef #ifndef
  - Miscellaneous
    - #pragma, #error

#include

- #include includes header files (.h) in the source code (.c).
- #include <file.h>
  - Find file in standard include directory.
  - If not found, raise error.
- #include "file.h"
  - File file in current source directory.
  - If not found, find file in standard include directory.
  - If not found, raise error.

#define (Symbolic constants)

- Used to define symbolic constants.
  - #define PI 3.142
  - #define SIZE 10
- Predefined constants
  - **LINE**
  - **FILE**
  - **DATE**
  - **TIME**
- Symbolic constants and macros are available from there declaration till the end of file. Their scope is not limited to the function.

#define (Macro)

- Used to define macros (with or without arguments)
  - #define ADD(a, b) (a + b)
  - #define SQUARE(x) ((x) * (x))
  - #define SWAP(a,b,type) { type t = a; a = b; b = t; }
- Macros are replaced with macro expansion by preprocessor directly.
  - May raise logical/compiler errors if not used parenthesis properly.
- Stringizing operator (#)
  - Converts given argument into string.
  - #define PRINT(var) printf(#var " = %d", var)
- Token pasting operator (##)
  - Combines argument(s) of macro with some symbol.

  - #define VAR(a,b) a##b

Differance between Function and Macro

Functions

- Function have declaration, definition and call.
- Functions are called at runtime by creating FAR on stack.
- Functions are type-safe.
- Functions may be recursive.
- Functions called multiple times doesn't increase code size.
- Functions execute slower.
- For bigger reusable code snippets, functions are preferred.

Macros

- Macro definition contain macro arguments and expansion.
- Macros are replaced blindly by the processor before compilation
- Macros are not type-safe.
- Macros cannot be recursive.
- Macros (multi-line) called multiple times increase code size.
- Macros execute faster.
- For smaller code snippets/formulas, macros are preferred.

Conditional compilation

- As preprocessing is done before compilation, it can be used to control the source code to be made available for compilation process.
- The condition should be evaluated at preprocessing time (constant values).
- Conditional compilation directives
  - #if, #elif, #else, #endif
  - #ifdef, #ifndef
  - #undef

```
#define VER 1
int main() {
    #ifndef VER
        #error "VER not defined"
    #endif
    #if VER == 1
         printf("This is Version 1.\n");
    #elif VER == 2
         printf("This is Version 2.\n");
    #else
        printf("This is 3+ Version.\n");
     #endif
     return 0;
}
```

Function Pointer

- function pointer is used to store address of function
- function address is address of first instruction of that function
- function name indicates address of that function
- to store address of function, we need pointer of same type
- Function Declaration/prototype/signature

```
<return type> <function name>([List of types of arguments]);
eg int fun(int, int);
    // fun indicates address of function
```

- Function Pointer

```
<return type> (*<pointer name>)([List of types of arguments]);
eg int (*ptr)(int, int);
    // ptr is a pointer '*' of collection '()' of statemets
    // which takes two arguments of type integer '(int, int)'
    // which return integer 'int'
```

- Few examples

```
    int fun(int);
    int (*ptr)(int);

    void fun(int);
    void (*ptr)(int);

    void fun(int, int);
    void (*ptr)(int, int);

    void fun(int, char);
    void (*ptr)(int, char);

    void fun(char, int);
    void (*ptr)(char, int);
```