

Lecture 8

Content	Compile in CMD	Cross Compiler vs Native Compiler	Macros
	Other Preprocessor Directives and Operators		
	Peak to Compilation Process		
	Preprocessor Conditional Compilation Directives		
	Static vs Dynamic Linking	Useful Command Lines	

Some Useful Command Lines

Cross Compiler vs Native Compiler

Native Compiler

Cross Compiler

Peak to Compilation Process

Preprocessor

Compilation process Commands

Static vs Dynamic Linking

Macros

Macros Best Practices

Multi-line Macro

Preprocessor Conditional Compilation Directives

#if (condition) . . . #endif

#ifdef (Macro) . . . #endif

File Guard Preprocessor Directive

Other Preprocessor Directives

__FILE__, __LINE__, __DATE__ Preprocessors

#error "Error message"

#line "New Line Number" "New File Name"

#pragma

Preprocessor Operators

defined(MACRO)

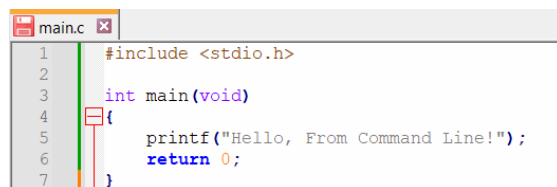
Stringize Operator

Continuation Operator

Some Useful Command Lines

- **cd** Command
 - Used to enter a directory if it is followed by a directory name `cd file`
 - Used to exit a directory if it is followed by two dots `cd ..`
- **dir** Command
 - Used to display the content of the current directory
- **cls** Command
 - Used to clear the command prompt
- `type nul > file.extension` command
 - Creates a file in the current directory.
 - We replace `file` with the file name
- Up\Down arrows

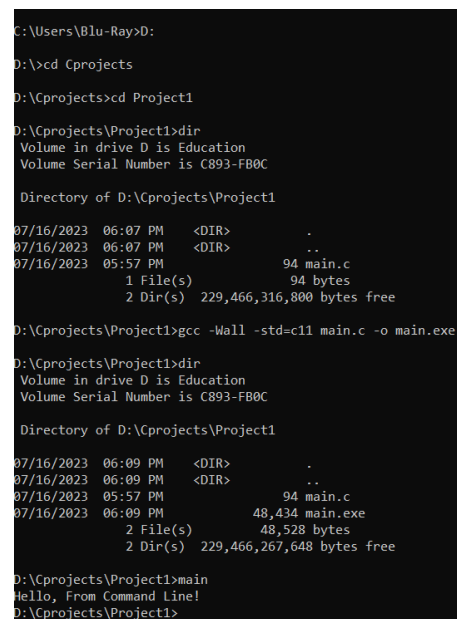
- Go back\forward to any previous commands we typed
- To Compile
 - .c file
 - `gcc -Wall -std=c11 main.c`
 - `gcc -Wall -std=c11 main.c -o main.exe` it simply set the output (.exe) file name
 - `gcc -Wall -std=c11 -c main.c -o main.o` it makes the compiler generate an object file
 - .cpp file
 - `g++ -Wall -std=c++14 main.cpp`
 - `g++ -Wall -std=c++14 main.cpp -o main.exe` it simply changes the .exe file name
- To run the .exe file We simply write the file name
- `<Partition name>:` Command
 - It takes you to another partition `D:`
- Example to compile .c file in the command prompt



```

1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Hello, From Command Line!");
6     return 0;
7 }

```



```

C:\Users\Blu-Ray>D:
D:\>cd Cprojects
D:\Cprojects>cd Project1
D:\Cprojects\Project1>dir
Volume in drive D is Education
Volume Serial Number is C893-FB0C

Directory of D:\Cprojects\Project1

07/16/2023  06:07 PM  <DIR>          .
07/16/2023  06:07 PM  <DIR>          ..
07/16/2023  05:57 PM                94 main.c
               1 File(s)                94 bytes
               2 Dir(s)  229,466,316,800 bytes free

D:\Cprojects\Project1>gcc -Wall -std=c11 main.c -o main.exe
D:\Cprojects\Project1>dir
Volume in drive D is Education
Volume Serial Number is C893-FB0C

Directory of D:\Cprojects\Project1

07/16/2023  06:09 PM  <DIR>          .
07/16/2023  06:09 PM  <DIR>          ..
07/16/2023  05:57 PM                94 main.c
07/16/2023  06:09 PM       48,434 main.exe
               2 File(s)        48,528 bytes
               2 Dir(s)  229,466,267,648 bytes free

D:\Cprojects\Project1>main
Hello, From Command Line!
D:\Cprojects\Project1>

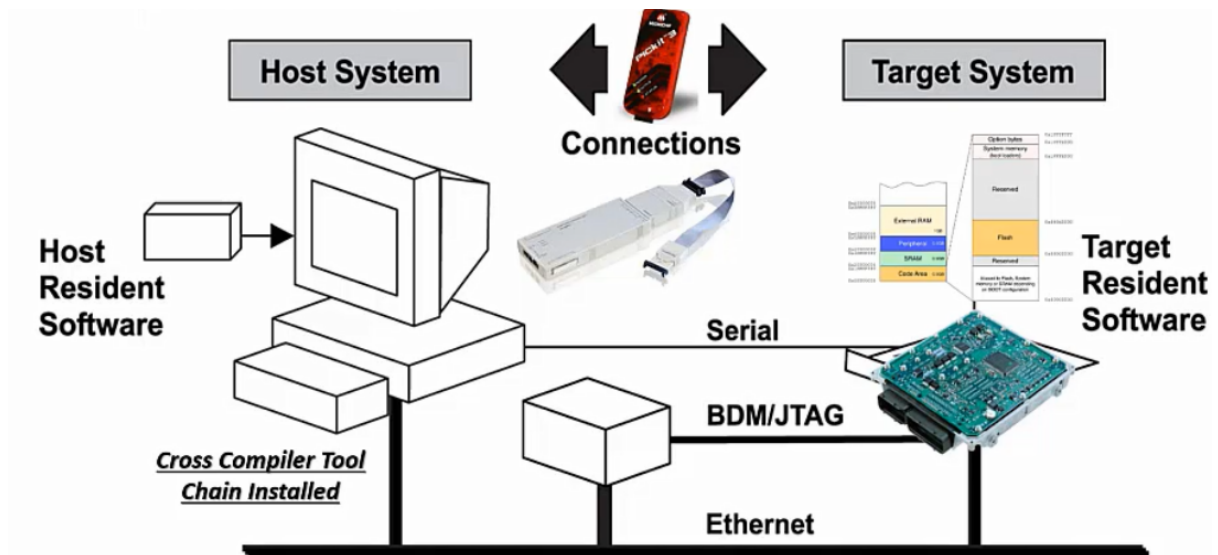
```

Cross Compiler vs Native Compiler

Native Compiler

- Compiler that generates a code for the **same platform** on which it runs
- It converts the high level code into computer's executable format
- The code generation\compilation and running the executable happened on the same platform
- Example: Turbo C and GCC compiler

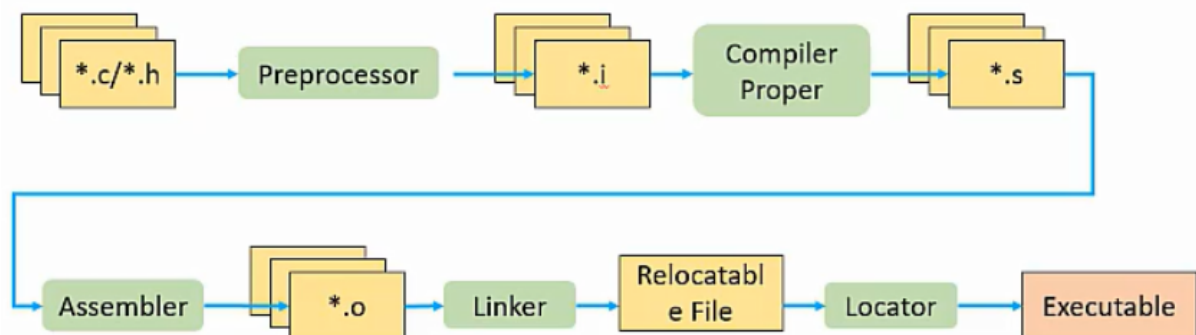
Cross Compiler



- The code will be Written in a platform (**Host**) and will be executed in another platform (**Target**)
- Compiler that generates executable code for **another platform** on which the compiler is running
- **Example**: GCC compiler for ARM Embedded Processors (GNU Arm Embedded Toolchain)
 - The output executable will run in to ARM based MCU

Peak to Compilation Process

- Tool Chain:
 - It is a set of executable files the compiler use to convert the .c files to .exe file
 - You will find the Tool Chain files in the **bin folder** in the compiler directory
 - `cpp.exe` → Preprocessor
 - `gcc.exe` → Compiler
 - `as.exe` → Assembler
 - `ld.exe` → Linker



Preprocessor

- It takes a .c file and generates a .i file (preprocessed → postprocessed)
- The preprocessor file name in the Tool Chain is `cpp.exe`
- To call the preprocessor for a .c file we use following command `cpp main.c > main.i` or `cpp -Wall -std=c11 main.c -o main.i`

```
D:\Cprojects\Project1>cpp main.c > main.i
D:\Cprojects\Project1>
```

- It does text replacement. replace each `#` with its equivalent text, and replace each comment with a single space



```
C main.c x C main.i 9+
Project1 > C main.c > main.i
1 #include <stdio.h>
2 #define NUM 5
3
4 // Comment
5
6 int main(void)
7 {
8     printf("Hello, From Command Line\NUM = %d !!!", NUM);
9     return 0;
10 }
```



```
C main.c x C main.i 9+ x
Project1 > C main.c > main.i
1398 "C:/Program Files/MinGW32/6406-w64-mingw32/include/stdio.h" 2 3
1399
1400 "C:/Program Files/MinGW32/6406-w64-mingw32/include/_mingw_print_pop.h" 1 3
1401
1402 "C:/Program Files/MinGW32/6406-w64-mingw32/include/stdio.h" 2 3
1403
1404 # 2 "main.c" 2
1405
1406
1407 # 6 "main.c"
1408 int main(void)
1409 {
1410     printf("Hello, From Command Line\NUM = %d !!!", 5);
1411     return 0;
1412 }
```

- If we have a user-defined library in a folder different from the project folder, we can include it with its path. Example: `#include "C:\Users\Username\OneDrive\Desktop\Cprojects\Project1\Libraries\myLibrary.h"`
- We cannot define a function more than one time. So, if we include the .c file that contains the functions declarations in more than one .c file we will get an error `"multi-definition error"`

Compilation process Commands

- To make the compiler takes the (.i) file and generate a (.s) file we use following command `gcc -Wall -std=c11 -S inputFile.i -o outputFile.s` OR `gcc -S inputFile.i -o outputFile.s`

```
C:\Users\Blu-Ray\OneDrive\Desktop\Cprojects\Project1>gcc -Wall -std=c11 -S main.i -o main.s
```

- To make the assembler takes the (.s) file and generate a (.o) file we use following command `as -Wall inputFile.s -o outputFile.o` or `as inputFile.s -o outputFile.o`

```
C:\Users\Blu-Ray\OneDrive\Desktop\Cprojects\Project1>as -Wall main.s -o main.o
```

- To Link the (.o) files and generate a (.exe) file we use following command `gcc -Wall -std=c11 main1.o main2.o ... -o main.exe`

```
C:\Users\Blu-Ray\OneDrive\Desktop\Cprojects\Project1>gcc -Wall -std=c11 main.o -o main.exe
```

- A step by step compilation process on command prompt

```

C main.c X C app.c C app.h
Project1 > C main.c > main(void)
1 #include <stdio.h>
2
3 #include "app.h"
4 #include "app.h"
5 #include "app.h"
6
7 #define NUM 5
8
9 // Comment
10
11 int main(void)
12 {
13     int number;
14     printf("Hello, From Command Line!\nNUM = %d !!!\n", NUM);
15     printMyName();
16     printf("Enter a number\n");
17     scanf("%d", &number);
18     printf("You entered %d\n", number);
19     return 0;
20 }

```

```

C main.c C app.c C app.h X
Project1 > C app.h > printMyName(void)
1 #include <stdio.h>
2
3 void printMyName(void);

```

```

C main.c C app.c C app.h
Project1 > C app.c > ...
1 #include "app.h"
2
3 void printMyName(void)
4 {
5     printf("Mahmoud Khaled\n");
6     return;
7 }

```

```

C:\Users\Blu-Ray\OneDrive\Desktop\Cprojects\Project1>gcc -Wall -std=c11 main.c -o main.i
C:\Users\Blu-Ray\OneDrive\Desktop\Cprojects\Project1>gcc -Wall -std=c11 app.c -o app.i
C:\Users\Blu-Ray\OneDrive\Desktop\Cprojects\Project1>gcc -Wall -std=c11 -S app.i -o app.s
C:\Users\Blu-Ray\OneDrive\Desktop\Cprojects\Project1>gcc -Wall -std=c11 -S main.i -o main.s
C:\Users\Blu-Ray\OneDrive\Desktop\Cprojects\Project1>as main.s -o main.o
C:\Users\Blu-Ray\OneDrive\Desktop\Cprojects\Project1>as app.s -o app.o
C:\Users\Blu-Ray\OneDrive\Desktop\Cprojects\Project1>gcc -Wall -std=c11 main.o app.o -o final.exe
C:\Users\Blu-Ray\OneDrive\Desktop\Cprojects\Project1>final
Hello, From Command Line!
NUM = 5 !!!
Mahmoud Khaled
Enter a number:
5000
You entered 5000

```

Static vs Dynamic Linking

- Static and dynamic linking are two different methods of linking object files together to create an executable file in a software development environment.
- Static linking involves linking the libraries used by the executable file directly into the executable itself, creating a standalone executable file that doesn't depend on external libraries at runtime.
- Dynamic linking involves linking the executable file to external libraries at runtime, resulting in a smaller executable file that depends on the libraries being available on the system.
- Static linking can result in larger executable files and may require recompiling the entire program if any of the linked libraries are updated.
- Dynamic linking allows for more efficient use of system resources but requires the libraries to be installed on the system separately and may result in version compatibility issues.
- Choosing between static and dynamic linking depends on the specific requirements of the project, such as the need for portability, performance, and ease of maintenance.

Macros

Macros Best Practices

- Macros Name is written in upper case letters

- If the macro name is a multi-word name we separate between then with `_`
- We cannot write any comments with the same line with the macro except the last line
- In function like macro we wrap each parameter with round brackets
- macro doesn't end with semi-column

```
#define MACRO_NAME (300.0) // this is a macro
```

- The preprocessor doesn't replace any text between double quotes. Example:

```
#include <stdio.h>
#define MY_NAME "Mahmoud Khaled Mohamed"

int main(void)
{
    printf("My name is MY_NAME"); // here the preprocessor doesn't replace MY_NAME
    return 0;
}
```

Multi-line Macro

- Syntax for multi-line macro

```
# \
define \
MACRO_NAME \
5000
```

- In multi-line macro if we need to write any comments it must be in the last line

```
# \
define \
SET_BIT(X, BIT_POSITION) \
(X) |= (1 << (BIT_POSITION)) // This function like macro set a bit with value 1 in variable X
```

- Example to multi-line macro. Swapping function using XOR method

```
#define SWAP(X, Y) \
{ \
    *(X) = *(X) ^ *(Y); \
    *(Y) = *(X) ^ *(Y); \
    *(X) = *(X) ^ *(Y); \
}
```

Preprocessor Conditional Compilation Directives

#if (condition) . . . #endif

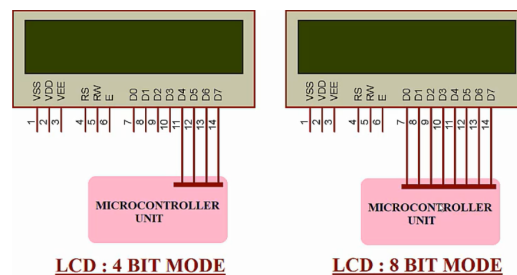
- It is used to choose between two or more codes so that one of them is compiled under a condition and the other code is neglected
- Syntax

```

#if (/* condition 1 */)
/* Code 1 to be compiled */
#elif (/* condition 2 */)
/* Code 2 to be compiled */
#else (/* condition 3 */)
/* Code 3 to be compiled */
#endif
// the #endif must end the structure of the preprocessor directive

```

- Real world example: If we have an LCD driver written. The LCD drive contains a code for 8-bit LCD and 4-bit LCD. Here, we need to choose one code of them to compile depending on our LCD.



```

C main.c  C lcd.c  X  C lcd.i  9+  C lcd.h
C lcd.c > print_data(uint8_t)
1  #include <stdio.h>
2
3  #include "lcd.h"
4
5  static void print_lcd_4bit(uint8_t data);
6  static void print_lcd_8bit(uint8_t data);
7
8  static void print_lcd_4bit(uint8_t data){
9      printf("4-bit data = %d", data);
10 }
11
12 static void print_lcd_8bit(uint8_t data){
13     printf("8-bit data = %d", data);
14 }
15
16 void print_data(uint8_t data){
17     #if LCD_MODE==4
18     print_lcd_4bit(data);
19     #elif LCD_MODE==8
20     print_lcd_8bit(data);
21     #endif
22 }

```

```

C main.c  C lcd.c  C lcd.i  9+  C lcd.h  X
C lcd.h > ...
1  #ifndef _PROJECT2_LCD_H_
2  #define _PROJECT2_LCD_H_
3
4  #define LCD_MODE 4
5
6  typedef unsigned char uint8_t;
7
8  void print_data(uint8_t data)
9
10 #endif // _PROJECT2_LCD_H_

```

```

C main.c  C lcd.c  C lcd.i  9+  X  C lcd.h
C lcd.i > ...
177
178 # 6 "lcd.h"
179 typedef unsigned char uint8_t;
180 void print_data(uint8_t data)
181 # 4 "lcd.c" 2
182
183 static void print_lcd_4bit(uint8_t data);
184 static void print_lcd_8bit(uint8_t data);
185
186 static void print_lcd_4bit(uint8_t data){
187     printf("4-bit data = %d", data);
188 }
189
190 static void print_lcd_8bit(uint8_t data){
191     printf("8-bit data = %d", data);
192 }
193
194 void print_data(uint8_t data){
195     print_lcd_4bit(data);
196 }
197
198
199
200
201
202

```

- We can eliminate a piece of code if we put it between `#if 0` and `#endif`. It removes the code in the preprocessing time

```

C main.c  C lcd.c  C lcd.i  9+  C lcd.h
C main.c > main()
1  #include <stdio.h>
2
3  #include "lcd.h"
4
5  int main()
6  {
7      #if 0
8      printf("Mahmoud khaled mohamed\n");
9      return 0;
10     #endif
11 }

```

#ifdef (Macro) . . . #endif

- Used to detect if a macro is defined or not
- Syntax

```
#ifndef (/* macro */)
/* Code 1 to be compiled */
#endif
```

File Guard Preprocessor Directive

- It is used to prevent copying the content of the (.h) file multiple times if it is included multiple times
- Syntax

```
// our .h file
#ifndef _FOLDER_FILE_H_
#define _FOLDER_FILE_H_
/* .h file content */
#endif
```

- We can use it to define a not defined macro. Example

```
#ifndef NULL
#define NULL ((void *)0)
#endif
```

Other Preprocessor Directives

__FILE__, __LINE__, __DATE__ Preprocessors

- `__FILE__`: Replaced with the file name by the preprocessor → string type
- `__LINE__`: Replaced with the line number name by the preprocessor → int type
- `__DATE__`: Replaced with the file date by the preprocessor → string type

```
C main.c x C main.c+ C lcd.c C lcd.h C lcd.c+
1 #include <stdio.h>
2
3 #include "lcd.h"
4
5 int main()
6 {
7     printf("File Name = %s\nline Number = %d\ndate = %s\n", __FILE__, __LINE__, __DATE__);
8     return 0;
9 }
```

```
C main.c C main.c+ C lcd.c C lcd.h C lcd.c+
1 #include <stdio.h>
2 #include "lcd.h"
3 void print_data(char *data)
4 {
5     #4 "main.c" 2
6 }
7
8 int main()
9 {
10    printf("File Name = %s\nline Number = %d\ndate = %s\n", "main.c", 7, "Jul 18 2023");
11    return 0;
12 }
```

```
C:\Users\Bla-Bla\Desktop\Projects\Project2\main
File Name = main.c
line Number = 6
date = Jul 18 2023
```

#error "Error message"

- Used to make the preprocessor generates an error message
- It stops the compilation process
- Syntax: `#error "Error Message"`
- Example

```
#ifdef TEST
#error "Test is defined"
#else
#error "Test is NOT defined"
#endif
```


#line "New Line Number" "New File Name"

- Used to set a line number instead of the default number
- Syntax: `#line "New Line Number" "New File Name"`
- Example

```
#include <stdio.h>

int main(void)
{
    #line 33 "new.c"
    printf("File Name = %s\tLine Number = %d\n", __FILE__, __LINE__); // File Name = new.c    line Number = 33
    printf("File Name = %s\tLine Number = %d\n", __FILE__, __LINE__); // File Name = new.c    line Number = 34
    return 0;
}
```

#pragma

- Compiler dependent. #pragma Commands differ from compiler to another. To know pragmas offered by the compiler you have to read the compiler's documentation
- It may affect the code portability. Because of that it is rarely used
- Most of the compilers define the command `#pragma once` to do the functionality of file guard. But file guard is more used.
- `#pragma GCC poison identifier` this pragma is used specially with GCC compiler. It is throw an error if the identifier specified is used in our file. Example

```
#pragma GCC poison printf
. . .
printf("Hello, World!"); // error
. . .
```

- `#pragma GCC warning "Warning message"` this pragma is used specially with GCC compiler. It is throw an warning at preprocessing time.
- `#pragma GCC error "error message"` this pragma is used specially with GCC compiler. It is throw an error at preprocessing time, and stops the compilation

Preprocessor Operators

defined(MACRO)

- Return true if the macro passed into it is defined. Otherwise, It returns false
- Syntax

```
#define TEST 2

#if defined(TEST)
/* code */
#endif
```

- We can write logical expression with it like `!defined(test1) && defined(test2)`

Stringize Operator

- Used to put the macro next to it between double quotes
- Syntax `#MACRO`
- Example

```
C:\main> #PRINT_NAME(FIRST_NAME, SECOND_NAME)
1 #ifndef _PROJECT2_FILE_H_
2 #define _PROJECT2_FILE_H_
3
4 #include <stdio.h>
5
6 #define PRINT_NAME(FIRST_NAME, SECOND_NAME) printf("#FIRST_NAME " " #SECOND_NAME " "\n")
7
8 #endif
```

```
C:\main.c> #PRINT_NAME(Mahmoud, Khaled)
1 #include "file.h"
2
3 PRINT_NAME(Mahmoud, Khaled);
```

```
C:\main.c> #PRINT_NAME(Mahmoud, Khaled)
1 #include "file.h"
2
3 PRINT_NAME(Mahmoud, Khaled);
883 # 1400 "C:/Program Files/mingw32/i686-w64-mingw32/inc
884 # 5 "file.h" 2
885 # 2 "file.c" 2
886
887 # 3 "file.c"
888 printf("Mahmoud " " "khaled" "\n");
889
890
```

Continuation Operator

- It is used to indicate that the macro is continued in the next line
- Syntax

```
#define FUN(PAR_1, PAR_2) \
{ \
    /* code */ \
}
```

- Example

```
#define SWAP(X, Y) \
{ \
    X = X ^ Y; \
    Y = X ^ Y; \
    X = X ^ Y; \
}
```