02_13_ C Preprocessor Directives

- You learned how to use the #include preprocessor directive to include C header files.

- Since then, the **`#include`** directive has been used in every program in this class.

- In this lesson you'll learn more about the C preprocessor and making macro definitions with the preprocessor directives.

02_13_ C – Preprocessor Directives

- Before a C program is compiled in a compiler, source code is processed by a program called preprocessor. This process is called preprocessing.

- Commands used in preprocessor are called **preprocessor directives** and they begin with "**#**" symbol.

What Is the C Preprocessor?

- If there is a constant appearing in several places in your program, it's a good idea to associate a symbolic name to the constant, and then use the symbolic name to replace the constant throughout the program.

- There are two advantages to doing so.
  1. First, your program will be more readable.
  2. Second, it's easier to maintain your program.

- For instance, if the value of the constant needs to be changed, find the statement that associates the constant with the symbolic name and replace the constant with the new one.

- Without using the symbolic name, you have to look everywhere in your program to replace the constant. Sounds great, but can we do this in C?

What Is the C Preprocessor?

- Well, C has a special program called the **C preprocessor** that allows you to define and associate symbolic names with constants.
- In fact, the C preprocessor uses the terminology macro names and macro body to refer to the symbolic names and the constants.
- The C preprocessor runs before the compiler. During preprocessing, the operation to replace a macro name with its associated macro body is called **macro substitution** or **macro expansion**.
- You can put a macro definition anywhere in your program.
- However, a macro name has to be defined before it can be used in your program.

What Is the C Preprocessor?

- In addition, the C preprocessor gives you the ability to include other source files.
- For instance, we've been using the preprocessor directive #include to include C header files, such as **`stdio.h`**, **`stdlib.h`**, and **`string.h`**, in the programs throughout this class.
- Also, the C preprocessor enables you to compile different sections of your program under specified conditions.

The C Preprocessor Versus the Compiler

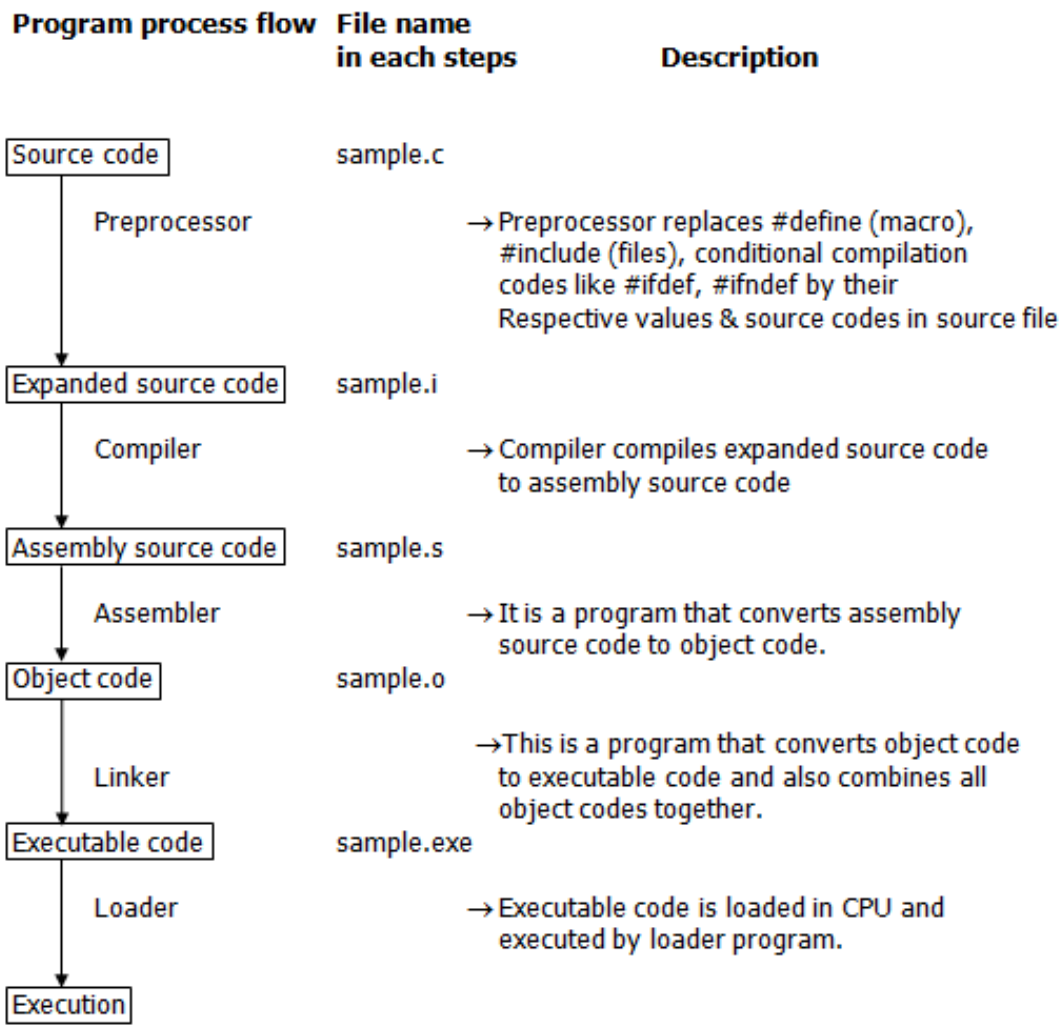
- One important thing you need to remember is that the C preprocessor is not part of the C compiler.
- The C preprocessor uses a different syntax.
- All directives in the C preprocessor begin with a **`pound sign`** (**`#`**).
- In other words, the pound sign denotes the beginning of a preprocessor directive, and it must be the first non-space character on the line.

The C Preprocessor Versus the Compiler

- The C preprocessor is line oriented.
- Each macro statement ends with a newline character, not a semicolon.
- (Only C statements end with semicolons.)
- One of the most common mistakes made by the programmer is to place a semicolon at the end of a macro statement. Fortunately, many C compilers can catch such errors.
- **NOTE:**
  - Macro names, especially those that will be substituted with constants, are normally represented with uppercase letters so that they can be distinguished from other variable names in the program.

02_22_ C – Preprocessor Directives

S.no	Preprocessor	Syntax	Description
1	Macro	`#define`	This macro defines constant value and can be any of the basic data types.
2	Header file inclusion	`#include <file_name>`	The source code of the file "`file_name`" is included in the main program at the specified place
3	Conditional compilation	`#ifdef, #endif, #if, #else, #ifndef`	Set of commands are included or excluded in source program before compilation with respect to the condition
4	Other directives	`#undef, #pragma`	`#undef` is used to undefine a defined macro variable. `#Pragma` is used to call a function before and after main function in a C program


Program process flow
File name in each steps
Description
Source code
sample.c
Preprocessor
→ Preprocessor replaces #define (macro), #include (files), conditional compilation codes like #ifdef, #ifndef by their Respective values & source codes in source file
Expanded source code
sample.i
Compiler
→ Compiler compiles expanded source code to assembly source code
Assembly source code
sample.s
Assembler
→ It is a program that converts assembly source code to object code.
Object code
sample.o
Linker
→ This is a program that converts object code to executable code and also combines all object codes together.
Executable code
sample.exe
Loader
→ Executable code is loaded in CPU and executed by loader program.
Execution

`#define` with Strings

- **Is it possible to #define strings?**

- Sure:
- `#define MY_PATH "/path/to/file"`
- That defines a macro named `MY_PATH` which gets replaced during preprocessing by the string literal `"/path/to/file"`.
- The main disadvantage of the `#define` method is that the string is duplicated each time it is used, so you can end up with lots of copies of it in the executable, making it bigger.
- The compiler or linker will optimize them out at least as long as they're used in the same file and collapses the copies into one. This feature is also sometimes called "**string pooling**“.

#define, #include preprocessors in C

- **#define** - This macro defines constant value and can be any of the basic data types.
- **#include <file_name>** - The source code of the file "**file_name**" is included in the main C program where "**#include <file_name>**" is mentioned.

```
#include <stdio.h>

#define HEIGHT 100
#define NUMBER 3.14
#define LETTER 'A'
#define LETTER_SEQUENCE "ABC"
#define BACKSLASH_CHAR '\?'

void main()
{
   printf("value of HEIGHT    : %d \n", HEIGHT );
   printf("value of NUMBER : %f \n", NUMBER );
   printf("value of LETTER : %c \n", LETTER );
   printf("value of LETTER_SEQUENCE : %s \n", LETTER_SEQUENCE);
   printf("value of BACKSLASH_CHAR  : %c \n", BACKSLASH_CHAR);

}
```

```
value of height : 100
value of number : 3.140000
value of letter : A
value of letter_sequence : ABC
value of backslash_char : ?
```

#ifdef, #else and #endif in C

- " **#ifdef** " directive checks whether particular macro is defined or not.
- If it is defined, "**If**" clause statements are included in source file.
- Otherwise, "**else**" clause statements are included in source file for compilation and execution.

```
#include <stdio.h>
#define RAJU 100

int main()
{
   #ifdef RAJU
   printf("RAJU is defined. So, this line will be added in this C file\n");
   #else                    /* else is optional */
   printf("RAJU is not defined\n");
   #endif                   /* endif is obligatory*/
   return 0;
}
```

```
RAJU is defined. So, this line will be added in this C file
```

#ifndef and #endif in C

- **#ifndef** exactly acts as reverse as **#ifdef** directive.

- If particular macro is not defined, "**If**" clause statements are included in source file.
- Otherwise, else clause statements are included in source file for compilation and execution.

```
#include <stdio.h>
#define RAJU 100
int main()
{
   #ifndef SELVA
   {
      printf("SELVA is not defined. So, now we are going to define here\n");
      #define SELVA 300
   }
   #else
   printf("SELVA is already defined in the program");

   #endif
   printf("value of SELVA    : %d \n", SELVA );
   return 0;
}
```

```
SELVA is not defined. So, now we are going to define here
```

#if, #else and #endif in C

- "If" clause statement is included in source file if given condition is true.
- Otherwise, else clause statement is included in source file for compilation and execution.

```
#include <stdio.h>
#define A 100
int main()
{
   #if (A==100)
   printf("This line will be added in this C file since A = 100\n");

   #else
   printf("This line will be added in this C file since A is not equal to 100\n");

   #endif
   return 0;
}
```

```
This line will be added in this C file since a = 100
```

undef in C

- This directive undefines existing macro in the program.

```
#include <stdio.h>

#define HEIGHT 100
void main()
{
   printf("First defined value for HEIGHT: %d\n",HEIGHT);
   #undef HEIGHT           // undefining variable
   #define HEIGHT 600      // redefining the same for new value
   printf("value of HEIGHT after undef & redefine: %d\n",HEIGHT);
}
```

```
First defined value for HEIGHT: 100
value of HEIGHT after undef & redefine: 600
```

```c
// Using #if, #elif, and #else */
#include <stdio.h>

#define C_LANG    'C'
#define B_LANG    'B'
#define NO_ERROR  0

int main(void)
{
        #if C_LANG == 'C' && B_LANG == 'B'
                #undef C_LANG
                #define C_LANG "I know the C language.\n"
                #undef B_LANG
                #define B_LANG "I know BASIC.\n"
                printf("%s%s", C_LANG, B_LANG);
        #elif C_LANG == 'C'
                #undef C_LANG
                #define C_LANG "I only know C language.\n"
        printf("%s", C_LANG);
        #elif B_LANG == 'B'
                #undef B_LANG
                #define B_LANG "I only know BASIC.\n"
                printf("%s", B_LANG);
        #else
                printf("I don't know C or BASIC.\n");
        #endif
        return NO_ERROR;
}
```

```
I know the C language.
I know BASIC.
```

```c
#include <stdio.h>

#define C_LANG    'C'
#define B_LANG    'B'
#define NO_ERROR  0

int main(void)
{
   #if C_LANG == 'D' && B_LANG == 'B'
     #define C_LANG_VALUE "I know the C language.\n"
     #define B_LANG_VALUE "I know BASIC.\n"
     printf("%s%s", C_LANG_VALUE, B_LANG_VALUE);
   #elif C_LANG == 'C'
     #define C_LANG_VALUE "I only know C language.\n"
     printf("%s", C_LANG_VALUE);
   #elif B_LANG == 'B'
     #define B_LANG_VALUE "I only know BASIC.\n"
     printf("%s", B_LANG_VALUE);
   #else
     printf("I don't know C or BASIC.\n");
   #endif

   return NO_ERROR;
}
```

```
I only know C language.
```

Defining Function-Like Macros with `#define`

- You can specify one or more arguments to a macro name defined by the `#define` directive, so that the macro name can be treated like a simple function that accepts arguments.

- For instance, the following macro name, `MULTIPLY`, takes two arguments:

```
#define MULTIPLY(val1, val2)  ((val1) * (val2))
```

- When the following statement:

```
result = MULTIPLY(2, 3) + 10;
```

- is preprocessed, the preprocessor substitutes the expression `2` for `val1` and `3` for `val2`, and then produces the following equivalent:

```
result = ((2) * (3)) + 10;
```

```
1:  /* 22L01.c:  Using #define */
2:  #include <stdio.h>
3:
4:  #define METHOD       "ABS"
5:  #define ABS(val)     ((val) < 0 ? -(val) : (val))
6:  #define MAX_LEN      8
7:  #define NEGATIVE_NUM -10
8:
9:  main(void)
10: {
11:    char *str = METHOD;
12:    int array[MAX_LEN];
13:    int i;
14:
15:    printf("The orignal values in array:\n");
16:    for (i=0; i<MAX_LEN; i++){
17:       array[i] = (i + 1) * NEGATIVE_NUM;
18:       printf("array[%d]: %d\n", i, array[i]);
19:    }
20:
21:    printf("\nApplying the %s macro:\n", str);
22:    for (i=0; i<MAX_LEN; i++){
23:       printf("ABS(%d): %3d\n", array[i], ABS(array[i]));
24:    }
25:
26:    return 0;
27: }
```

```
The orignal values in array:
array[0]: -10
array[1]: -20
array[2]: -30
array[3]: -40
array[4]: -50
array[5]: -60
array[6]: -70
array[7]: -80

Applying the ABS macro:
ABS(-10):  10
ABS(-20):  20
ABS(-30):  30
ABS(-40):  40
ABS(-50):  50
ABS(-60):  60
ABS(-70):  70
ABS(-80):  80
```

Nested Macro Definitions

- A previously defined macro can be used as the value in another #define statement. The following is an example:

```
#define ONE     1
#define TWO     (ONE + ONE)
#define THREE   (ONE + TWO)
result = TWO * THREE;
```

- Here the macro **ONE** is defined to be equivalent to the value **1**, and **TWO** is defined to be equivalent to **(ONE + ONE)**, where **ONE** is defined in the previous macro definition. Likewise, **THREE** is defined to be equivalent to **(ONE + TWO)**, where both **ONE** and **TWO** are previously defined.

- Therefore, the assignment statement following the macro definitions is equivalent to the following statement:

```
result = (1 + 1) * (1 + (1 + 1));
```

Warning

- When you are using the **#define** directive with a macro body that is an expression, you need to enclose the macro body in parentheses. For example, if the macro definition is

```
#define SUM  12 + 8
result = SUM * 10;
```

- becomes this:

```
result = 12 + 8 * 10;
```

- which assigns **92** to result.

- However, if you enclose the macro body in parentheses like this:

```
#define SUM  (12 + 8)
result = (12 + 8) * 10;
```

- and produces the result **200**, which is likely what you want.

`pragma` in C

- Pragma is used *to call a function before and after main* function in a C program.

S.no	Pragma command	Description
1	`#pragma startup <function_name> [priority]`	This directive executes function named "function_name_1" before
2	`#pragma exit <function_name> [priority]`	This directive executes function named "function_name_2" just before termination of the program.

`pragma` in C

```
0     = Highest priority
0-63  = Used by C libraries
64    = First available user priority
100   = Default priority
255   = Lowest priority
```

- The optional priority parameter should be an integer in the range 64 to 255.
- The highest priority is 0.
- Otherwise; Functions with higher priorities are called first at startup and last at exit.
- If you don't specify a priority, it defaults to 100.
- **Warning:** Do not use priority values less than 64. Priorities from 0 to 63 are reserved for ISO startup and shutdown mechanisms.

`pragma` in C

- **Is #pragma directive compiler dependent?**

- All **`#pragma`** directives are compiler-dependent, and a compiler is obliged to ignore any it does not recognize.
- Pragmas are not just compiler vendor specific, they're also version specific.
- **How to know?** Read compiler's user manual.
- (ISO-9899:2011, s6.10.6: “Any such pragma that is not recognized by the implementation is ignored.”).

pragma in C

```
#include<stdio.h>

void School();
void College() ;

#pragma startup School 105
#pragma startup College 110

#pragma exit College 110
#pragma exit School 105

void main(){
printf("I am in main \n");
}

void School(){
printf("I am in School \n ");
}

void College(){
printf("I am in College \n ");
}
```

```
I am in School
I am in College

I am in main

I am in College
I am in School
```

pragma in C

```
#include<stdio.h>

void School();
void College() ;

#pragma startup School 105
#pragma startup College       // this has a default priority value of 100

#pragma exit College          // this has a default priority value of 100
#pragma exit School 105

void main(){
printf("I am in main \n");
}

void School(){
printf("I am in School \n ");
}

void College(){
printf("I am in College \n ");
}
```

```
I am in College
I am in School

I am in main

I am in School
I am in College
```

gcc does not support pragma (Web link)

- Instead, use the two following definitions of constructor and destructor.

- With this feature, the functions defined as constructor function would be executed before the function main starts to execute, and the destructor would be executed after the main has finished execution.

```
1  __attribute__((constructor)) void begin (void)
2  {
3   /* Function Body */
4  }
5  __attribute__((destructor)) void end (void)
6  {
7   /* Function Body */
8  }
```

- The constructors with *lower priority* value would be executed first.
- The destructors with *higher priority* value would be executed first.

```
#include <stdio.h>

void school (void) __attribute__((constructor (101)));
void college (void) __attribute__((constructor (102)));

void school (void) __attribute__((destructor (101)));
void college (void) __attribute__((destructor (102)));

int main (void)
{
  printf ("Inside main ()\n");
}

void school (void)
{
  printf ("In school ()\n");
}

void college (void)
{
  printf ("In college ()\n");
}
```

```
In school ()
In college ()

Inside main ()

In college ()
In school ()
```

To `main` or not to `main` !!

- Note that the function main () is not the first function/code block to execute in your code there are a lot of code already executed before main starts to execute.

- The function main is the user's code entry point, but the program entry point is not the main function.

- There is a startup function which prepares the environment for the execution.

    1. The startup functions first call the functions declared as constructors
    2. The startup functions calls the main, when main returns the control to the startup function
    3. The startup functions then calls those functions which you have declared as the destructors.

End of 02_13