

# **COSC1101 – Programming Fundamentals**

Maham Khan

Lecture – 16

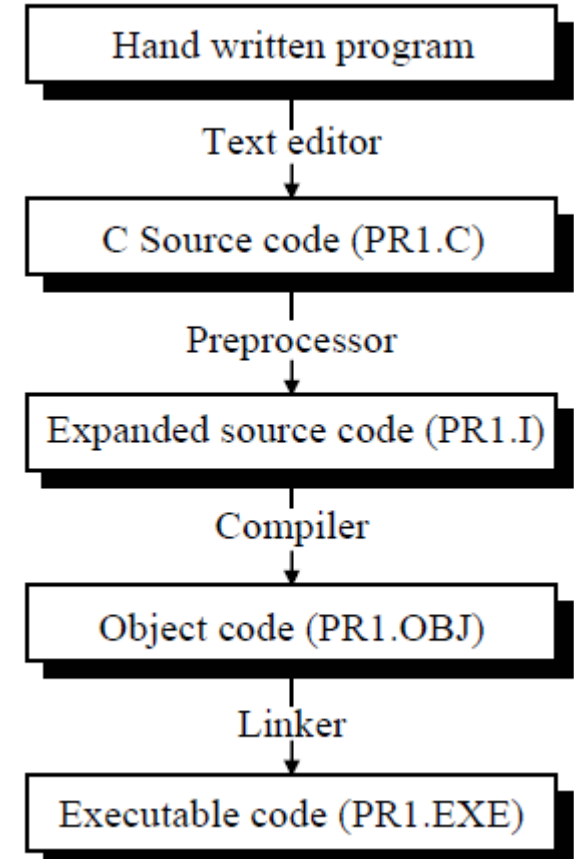
# Preprocessor Directives

# Preprocessor Directives

- A program that processes the source code before passing it to the compiler.
- Preprocessor commands (known as directives) can be considered a language within C language.
- We can write C programs even without knowing anything about the preprocessor or its facilities.

# Function of utilities

Processor	Input	Output
Editor	Program typed from keyboard	C source code containing program and preprocessor commands
Preprocessor	C source code file	Source code file with the preprocessing commands properly sorted out
Compiler	Source code file with preprocessing commands sorted out	Relocatable object code
Linker	Relocatable object code and the standard C library functions	Executable code in machine language



# Preprocessor Directives

The preprocessor offers several features called preprocessor directives.

- Each directive begin with a # symbol.
- The directives can be placed anywhere in a program but are mostly placed at the beginning of a program and before the first function definition.
- Categories of these directives are:

(a) Macro expansion	#define
(b) File inclusion	# include
(c) Conditional Compilation	#if, #elif
(d) Miscellaneous	#undef

# # define directive used for macro expansion

Example-1:

```
#define UPPER 25
main( )
{
    int i ;
    for ( i = 1 ; i <= UPPER ; i++ )
        printf ( "\n%d", i ) ;
}
```

# Explanation Macro Expansion

- Instead of writing 25 in the for loop we have defined a constantan before the start of the definition of main( ) function.

#define UPPER 25

- This above statement is called 'macro definition' or more commonly, just a 'macro'.
- During preprocessing, the preprocessor replaces every occurrence of UPPER in the program with 25.

# Macro Expansion

Example - 2:

```
#define PI 3.1415
```

```
main( )
```

```
{
```

```
    float r = 6.25 ;
```

```
    float area ;
```

```
    area = PI * r * r ;
```

```
    printf ( "\nArea of circle = %f", area ) ;
```

```
}
```

Output: Area of circle = 122.714844



# Macro Expansion and related concepts

- UPPER and PI in the above programs are often called 'macro templates', whereas, 25 and 3.1415 are called their corresponding 'macro expansions'.
- When we compile the program, before the source code passes to the compiler it is examined by the C preprocessor for any macro definitions.
- When it sees the #define directive, it goes through the entire program in search of the macro templates; wherever it finds one, it replaces the macro template with the appropriate macro expansion.

# Macro Expansion and related concepts .. contd

- After resolving preprocessor directives the code is transferred/handed over to the compiler.
- In C programming it is customary to use **capital letters** for **macro template**. That makes easy for the programmers to pick up all the macro templates when reading through the program.
- A **macro template** and its **macro expansion** both are separated by **blanks** or **tabs**.
- A **space** between **#** and **define** is optional.
- Remember that a macro definition **should never be terminated by a semicolon**.

# Why we use macros?

## 1. Improve Readability

- made the program easier to read.
- Even though 3.1415 is a common constant which is easily recognizable, yet there are many instances where a constant doesn't reveal its purpose so readily.
- For example, the string “\n” causes the cursor to move down one line, causes control transfer to newline.
- “\n” or “NEWLINE”, we may use the macro definition #define NEWLINE “\n”, wherever NEWLINE appears in the program it will automatically be replaced with “\n” before compilation begins.

# Why we use macros?

## 2. Easier to update many occurrences of a constant

- Suppose a constant like 3.1415 appears many times in your program. If required to be changed then need to change every occurrence. However, if you have defined PI in a #define directive, you only need to make one change, in the #define directive itself: #define PI 3.141592
- Beyond this the change will be made automatically to all occurrences of PI before the beginning of compilation.
- This convenience may not matter for small programs shown above, but with large programs macro definitions are almost indispensable.

# Why not variable instead macro templates?

- But the same purpose could have been served had we used a variable  $\pi$  instead of a macro template  $\text{PI}$ .
- A variable could also have provided a meaningful name for a constant and permitted one change to effect many occurrences of the constant.
- A variable can be used instead of macro templates.

Then, why not use it?

For three reasons it's a bad idea.

# Why not variable instead macro templates?

1. It is inefficient, since the compiler can generate faster and more compact code for constants than it can for variables.
2. Using a variable for what is really a constant encourages sloppy thinking and makes the program more difficult to understand: if something never changes, it is hard to imagine it as a variable.
3. There is always a danger that the variable may inadvertently get altered somewhere in the program. So it's no longer a constant that you think it is.

# Example-1 macro Template defining Operators

# define directive many be used to define operators as shown below:

```
# define AND &&
```

```
# define OR ||
```

```
main( )
```

```
{
```

```
    int f = 1, x = 4, y = 90 ;
```

```
    if ( ( f < 5 ) AND ( x <= 20 OR y <= 45 ) )
```

```
        printf ( "\n Inside the if block.." ) ;
```

```
    else
```

```
        printf ( "\n Inside the else block" ) ;
```

```
}
```

**Output:** Inside the if block..

## Example-2 Nested macro Template

A #define directive could be used even to replace a condition, as shown below.

```
# define AND &&
# define ARANGE ( a > 25 AND a < 50 )
main( )
{
    int a = 30 ;
    if ( ARANGE )
        printf ( "with in range" ) ;
    else
        printf ( "out of range" ) ;
}
```

**Output:** with in range



## Example-3: Macros with Arguments

```
#define ISDIGIT(y) ( y >= 48 && y <= 57 ) //Between '0' & '9'
main( )
{
    char ch ;
    printf ( "Enter any digit " ) ;
    scanf("%c",&ch);
    if ( ISDIGIT ( ch ) )
        printf ( "\n You entered a digit" ) ;
    else
        printf ( "\n You entered an invalid digit" ) ;
}
```

## Example-4: Macros with Arguments

```
#define AREA(x) ( 3.14 * x * x )  
main( )  
{  
    float r1 = 6.25, r2 = 2.5, a ;  
  
    a = AREA ( r1 ) ;  
    printf ( "\n Area of circle = %f", a ) ;  
    a = AREA ( r2 ) ;  
    printf ( "\n Area of circle = %f", a ) ;  
}
```

Output:

Area of circle = 122.656250

Area of circle = 19.625000

# Preprocessor output for the compiler input

- After the above source code has passed through the preprocessor, what the compiler gets to work on will be this:

```
main( )
```

```
{  
    float r1 = 6.25, r2 = 2.5, a ;  
    a = 3.14 * r1 * r1 ;           // Area (x) replaces 3.14*r1*r1  
    printf ( "Area of circle = %f\n", a ) ;  
    a = 3.14 * r2 * r2 ;           // Area (x) replaces 3.14*r2*r2  
    printf ( "Area of circle = %f", a ) ;  
}
```

## Points to Remember: (1. not to leave blanks)

while writing macros with arguments:

- Be careful not to leave a blank between the macro template and its argument. For example, there should be no blank between AREA and (x) in the definition,

`#define AREA(x) ( 3.14 * x * x ) . . . . .` Correct

`AREA -(x) ( 3.14 * x * x ) . . . . .` Incorrect

AREA will be translated as (x)

and remaining part will be ignored.

The template would be expanded

to ( r1 ) instead of expanding to ( 3.14 \* r1 \* r1 )

## Points to Remember: (2. enclose in parenthesis)

The entire macro expansion should be enclosed within parentheses. what would happen if we fail to enclose the macro expansion within parentheses:

```
#define SQUARE(n) n * n
main( )
{
int j ;
j = 64 / SQUARE ( 4 ) ;
printf ( "j = %d", j ) ;
}
```

The output of the above program would be:

j = 64

Where as, it should be j = 4. What went wrong?

The macro was expanded into j = 64 / 4 \* 4 ; which resulted 64.

# Macros versus Functions

- In the previous example a macro was used to calculate the area of the circle. As we know, even a function can be written to calculate the area of the circle. Though macro calls are 'like' function calls, they are not really the same things. Then what is the difference between the two?
- In a macro call the preprocessor replaces the macro template with its macro expansion, where as in a function call the control is passed to a function along with certain arguments, some calculations are performed in the function and a useful value is returned back from the function.

# Macros versus Functions . . . . contd

## macros with arguments

- macros make the program run faster but increase the program size.
- If we use a macro hundred times in a program, the macro expansion goes into our source code at hundred different places, thus increasing the program size.
- Passing arguments is nullified with macros since they have already been expanded and placed in the source code before compilation.

## function?

- Functions make the program smaller and compact.
- On the other hand, if a function is used, then even if it is called from hundred different places in the program, it would take the same amount of space in the program.
- Passing arguments to a function and getting back the returned value does take time and would therefore slow down the program.

# Macros versus Functions . . . . contd

## Moral of the story:

- if the macro is simple like in our examples, it makes nice shorthand and avoids the overheads associated with function calls.
- On the other hand, if we have a fairly large macro and it is used most often, then we ought to replace it with a function.



# Preprocessor Directive: # include

The second preprocessor directive is for a file inclusion.

- This directive causes one file to be included in another. The preprocessor command for file inclusion is:

✓ #include "filename" .

- It causes the entire contents of filename to be inserted into the source code at that point in the program.
- Of course it presumes that the file to be included exists.  
When and why this feature is used?  
It can be used in two cases:

# # include : When; why; and where to use?

- If we have a very large program, the code is best divided into several different files, each containing a set of related functions. It is a good programming practice to keep different sections of a large program separate. These files are `#included` at the beginning of main program file.
- There are some ***functions*** and some ***macro definitions*** that we need almost in all programs that we write. These commonly needed functions and macro definitions can be stored in a file, and that file can be ***included*** in every program we write, which would add all the statements in this file to our program as if we have typed them in.

# # include ( Header File .h extension)

- It is common for the files that are to be included to have a:  

## .h extension

 stands for 

## 'header file'

,  
possibly because it contains statements which go to  
the head of your program
- 
- The prototypes of all the library functions are grouped into different categories and then stored in different header files:
- Examples
  - prototypes of all mathematics related functions are stored in the header file 'math.h',
  - prototypes of console input/output functions are stored in the header file 'conio.h', and so on.

# # include : (How can we include files?)

- Actually there exist two ways to write #include statement.

These are:

- ✓ 1. #include "filename"
- ✓ 2. #include <filename>

- #include "goto.c" command would look for the file goto.c in the current directory as well as the specified list of directories as mentioned in the include search path that might have been set up.
- Whereas #include <goto.c> command would look for the file goto.c in the specified list of directories only.
- To setup the path: will be demonstration practically

# Conditional Compilation with #ifdef, #endif

- If required we can direct the compiler to skip over part of a source code by inserting the preprocessing commands:

`#ifdef` and `#endif`, which have the general form:

Example

```
#ifdef macroname  
statement 1 ;  
statement 2 ;  
statement 3 ;  
#endif
```

If macroname has been `#defined`, the block of code will be processed as usual; otherwise not.

# Where would `#ifdef` be useful?

When would you like to compile only a part of your program? Three cases:

## **Case-1:**

- To “comment out” obsolete lines of code. It happens that a program is changed at the last minute to satisfy a client.
- This involves rewriting some part of source code and deleting the old code for client’s satisfaction.
- The veteran programmers are familiar with the clients who change their mind and want the old code back again just the way it was.
- Now you would definitely not like to retype the deleted code again. One solution in such a situation is to put the old code within a pair of `/* */` combination.
- But we might have already written a comment in the code that we are about to “comment out”. This would mean we end up with nested comments. Obviously, this solution won’t work since we can’t nest comments in C. Therefore the solution is to use conditional compilation as shown below.

```
main( )
```

```
{
```

```
    #ifdef OKAY ✓
```

```
    statement 1 ;
```

```
    statement 2 ;
```

```
    statement 3 ;
```

```
    statement 4 ;
```

```
    #endif
```

```
    statement 5 ;
```

```
    statement 6 ;
```

```
    statement 7 ;
```

```
}
```

```
    /* old code */
```

```
    /* specific options */
```

# Where would `#ifdef` be useful?

## Case-2:

Suppose an organization has two different types of computers and you are expected to write a program that works on both the machines. You can do so by isolating the lines of code that must be different for each machine by marking them off with `#ifdef`.

For example:

```
main( )  
{  
    #ifdef INTEL  
    code suitable for a Intel PC  
    #else  
    code suitable for a Motorola PC  
    #endif  
    code common to both the computers  
}
```



# Explanation

- When you compile the above program it would compile only the code suitable for a motorola PC and the common code. This is because the macro INTEL has not been defined. Note that the working of :

`#ifdef`

`#else`

`#endif`

is similar to the ordinary if - else control instruction of C.

- If you want to run your program on an INTEL PC, just add a statement at the top saying, `#define INTEL`

# May use `#ifndef` instead of `#ifdef`

- Sometimes, instead of `#ifdef` the `#ifndef` directive is used. The `#ifndef` (which means 'if not defined') works exactly opposite to `#ifdef`. The above example if written using `#ifndef`, would look like this:

```
main( )  
{  
    #ifndef INTEL  
        code suitable for a Intel PC  
    #else  
        code suitable for a Motorola PC  
    #endif  
    code common to both the computers  
}
```

# Where would #ifdef be useful?

## Case-3:

Suppose a function myfunc( ) is defined in a file 'myfile.h' which is #included in a file 'myfile1.h'.

Now in your program file if you #include both 'myfile.h' and 'myfile1.h'

The compiler flashes an error 'Multiple declaration for myfunc'.

This is because the same file 'myfile.h' gets included twice.

To avoid this we can write following code in the header file. /\* myfile.h \*/

- #ifndef \_\_myfile\_h
- #define \_\_myfile\_h
- myfunc( )
- {
- /\* some code \*/
- }
- #endif

# Explanation

- First time the file 'myfile.h' gets included the preprocessor checks whether a macro called `__myfile_h` has been defined or not. If it has not been defined then it gets defined and the rest of the code gets included.
- Next time we attempt to include the same file, the inclusion is prevented since `__myfile_h` already stands defined. Note that there is nothing special about `__myfile_h`. In its place we can use any other macro as well.