PRE PROCESSOR DIRECTIVES IN –C LANGUAGE.

2401 – Course Notes.

The C preprocessor is a macro processor that is used automatically by the C compiler to transform programmer defined programs before actual compilation takes place. It is called a macro processor because it allows the user to define macros, which are short abbreviations for longer constructs.

This functionality allows for some very useful and practical tools to design our programs.

- To include header files(Header files are files with pre defined function definitions, user defined data types, declarations that can be included.)
- To include macro expansions. We can take random fragments of C code and abbreviate them into smaller definitions namely macros, and once they are defined and included via header files or directly onto the program itself it(the user defined definition) can be used everywhere in the program. In other words it works like a blank tile in the game scrabble, where if one player possesses a blank tile and uses it to state a particular letter for a given word he chose to play then that blank piece is used as that letter throughout the game. It is pretty similar to that rule.
- Special pre processing directives can be used to, include or exclude parts of the program according to various conditions.

To sum up, preprocessing directives occurs before program compilation. So, it can be also be referred to as pre compiled fragments of code. Some possible actions are the inclusions of other files in the file being compiled, definitions of symbolic constants and macros and conditional compilation of program code and conditional execution of preprocessor directives.

All preprocessor directives has begin with the # operator.

include **preprocessor directive:**

Only defined at the top of the program definition and only white space and comments may appear before preprocessor directive line.This directive includes a copy of the specified file or library. And is written like so-

# include <filename>

Or #include “file name”

If included inside <> then the compiler looks for the file in an implementation defined manner (or system directory paths).

If included inside a quote " " statement then it searches for the file in the same directory as the program being written.

So if we include a file named file1 while writing a program name code.c which is being stored in a folder called test. Then inclusion like so -- #include "file1"

Will tell the compiler to look for the file inside the test folder, It looks at neighboring folders or subfolders too but it starts its search in the test folder.(This sub search option is compiler dependent).

#define preprocessor directives (Symbolic constants)

Used to create symbolic constants and macros, meaning useful for renaming long named data types and certain constants or values which is being used throughout the program definition.

Symbolic constant definition-

Written as –

#define identifier replacement-text

When this line appears in a file, all subsequent occurrences of **identifier that do not appear in string literals will be replaced by the replacement text automatically before program compilation takes place.**

For example:

#define PI 3.14159

Replaces all subsequent occurrences of the symbolic constant PI with numeric constant 3.14159. Symbolic constants enable the programmer to create a name for a constant and use that name throughout program execution. If this needs to be modified ,it has to be done in the #define directive statement. After recompilation all definitions get modified accordingly.

#define Preprocessor Directive : Macros

- Symbolic constants and macros can be discarded by using the #undef preprocessor directive.
- Directive #undef "undefines" the symbolic constant or macro name.
- The scope of a symbolic constant or macro is from its definitions until it is undefined with #undef or until the end of the file.

Example Code –

```
#include<stdio.h>   --→ Include external source code file
#include "file1"   -→ Include a file which is in the same folder as the parent file
#define BUFFER_SIZE  250

Int maint(){
        Char array[BUFFER_SIZE] = [0];
        Int I = 9;

        Printf("%d",i);
        Printf("%s",array);

        Return 0;
}//end main body
```

Conditional Compilation:

Conditional compilation enables the coder to control the execution of preprocessor directives and the compilation of program code.
->Conditional preprocessor directives evaluate constant integer expressions.The ones that cannot be evaluated in preprocessor directives are sizeof expressions and enumeration constants.
Every #if construct ends with #endif

Example –

```
#if !defined(MY_CONSTANT)
        #define MY_CONSTANT 0
#endif
```

These directives determine if *MY_CONSTANT is defined* or not.The expression defined(*MY_CONSTANT*) evaluates to 1 if *MY_CONSTANT* is not defined and is defined else it evaluates to 0.

Predefined Symbolic Constants:

Standard C provides some predefined symbolic constants, several of which are shown below-

DIRECTIVES	EXPLANATION
LINE	The line number of the current source code line(An integer constant.)
FILE	The presumed name of the source file(a string).
DATE	The date source file was compiled (The string of the form ‘MM dd YYYY’ such as Jan 20 2013).
TIME	The time source file was compiled (A string literal of the form ‘hh:mm:ss’).
STDC	Value 1 if the compiler supports Standard C.

#error and #pragma not covered.
Assert macro defined in <assert.h> not covered.

Sources:
1.Internet Google Search - http://docs.freebsd.org/info/cpp/cpp.pdf
2.C How to Program –Deitel and Deitel (7th Ed),Chapter 13.
3.System programming (with Unix and C) –Adam Hoover