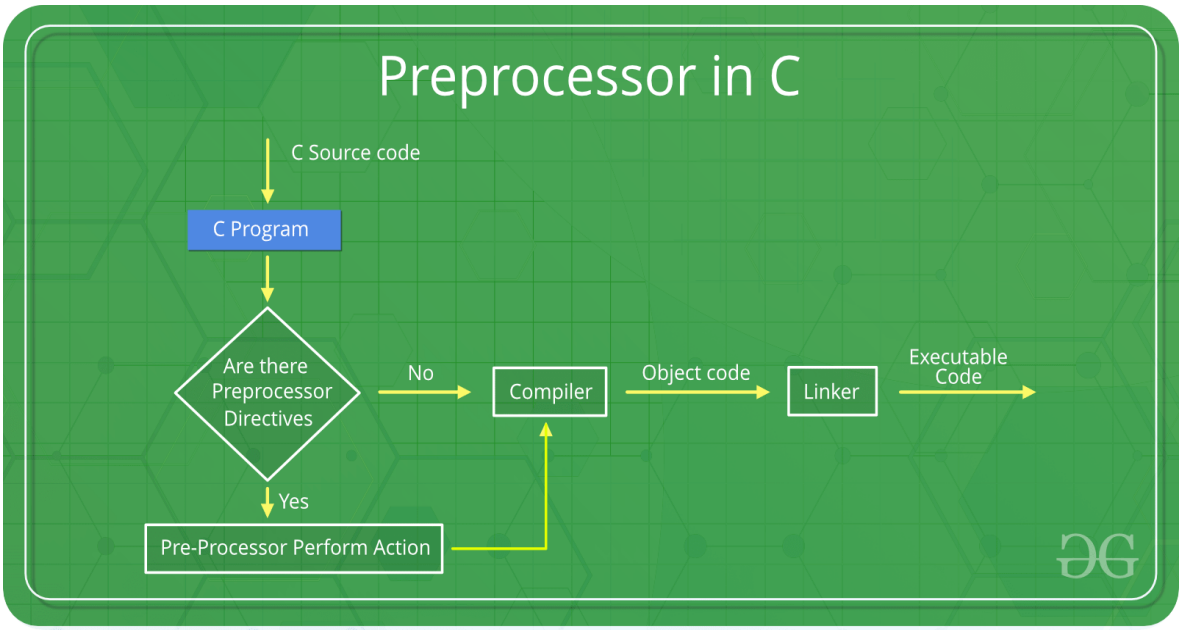
Preprocessor



- Preprocessors are programs that process our source code before compilation.
- The C Preprocessor is not a part of the compiler, but is a separate step in the compilation process. In simple terms, a C Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation.
- The advantage of preprocessor is:
  - Readability of the program is increased.
  - Program modification becomes easy.
  - Makes the program portable and efficient.
- There are 4 main types of preprocessor directives:
  - Macros
    - Simple Macro : #define LIMIT 5
    - Macro with Arguments : #define AREA(l, b) (l * b)
  - File Inclusion
    - `#include <file_name>`
    - `#include "file_name"`
  - Conditional Compilation
    - #if
    - #else
    - #elif
    - #endif
    - #ifdef
    - #ifndef

- Other directives
  - #undef
  - #line
  - #error
  - #pragma
  - Null Directives

Sr.No.	Directive & Description
1	**#define** Substitutes a preprocessor macro.
2	**#include** Inserts a particular header from another file.
3	**#undef** Undefines a preprocessor macro.
4	**#ifdef** Returns true if this macro is defined.
5	**#ifndef** Returns true if this macro is not defined.
6	**#if** Tests if a compile time condition is true.
7	**#else** The alternative for #if.
8	**#elif** #else and #if in one statement.

9	**#endif** Ends preprocessor conditional.
10	**#error** Prints error message on stderr.
11	**#pragma** Issues special commands to the compiler, using a standardized method.

- **Macros:**
  - Macros are a piece of code in a program which is given some name. Whenever this name is encountered by the compiler the compiler replaces the name with the actual piece of code. The *'#define'* directive is used to define a macro.

  - **Simple Macro:**
    - Macro is any valid C identifier, and it is generally taken in capital letters to distinguish it from other variables.
    - The macro expansion can be any text.
    - A space is necessary between the macro name and macro expansion.
    - The C preprocessor replaces all the occurrences of macro name with the macro expansion.
    - Syntax:

      ```
      #define macro_name macro_expansion
      ```
    - Example:

      ```
      #define TRUE 1
      #define FALSE 0
      #define PI 3.14159265
      ```

  - **Macro with Argument:**
    - We can also pass arguments to macros.
    - Macros defined with arguments works similarly as functions.
    - Syntax:

      ```
      #define macro_name(arg 1, arg2, .....) macro_expansion
      ```

- Example:

```
#define SUM(x, y) ( (x) + (y) )
#define PROD(x, y) ( (x) * (y) )
```

Now suppose we have these two statements in our programs-

```
s = SUM(5, 6);
p = PROD(m, n);
```

After passing through the preprocessor these statements would be expanded as-

```
s = ( (5) + (6) );
p = ( (m) * (n) );
```

- **Nesting with Macros:**
  - One macro name can also be used for defining another macro name i.e. the macro expansion can also contain the name of another macro.
  - Example:

```
#define PI 3. 14
#define PISQUARE PI*PI
```

• File Inclusion:

- This type of preprocessor directive tells the compiler to include a file in the source code program.
- There are two types of files which can be included by the user in the program:
  - **Header File or Standard Files:**
    - ❖ If the filename is within angle brackets, then the file is searched in the standard include directory only. The specification of standard include directory is implementation defined.
    - ❖ Generally angled brackets are used to include standard header files.
    - ❖ These files contains definition of pre-defined functions like printf(), scanf() etc. These files must be included for working with these functions.
    - ❖ Different function is declared in different header files. For example standard I/O functions are in 'iostream' file whereas functions which perform string operations are in 'string' file.
    - ❖ Syntax:

```
#include < file_name >
```

- **User Defined Files:**
  - ❖ If the filename is in double quotes, first it is searched in the current directory (where the source file is present), if not found there then it is searched in the standard include directory.
  - ❖ Generally double quotes are used to include header files related to a particular program.
  - ❖ When a program becomes very large, it is good practice to divide it into smaller files and include whenever needed. These types of files are user defined files.
  - ❖ Syntax:

    #include "*filename*"

- ## Conditional Compilation:
  - Conditional Compilation directives are type ofdirectives which helps to compile a specific portion of the program or to skip compilation of some specific part of the program based on some conditions.

  - **#if and #endif:**
    - An expression which is followed by the #if is evaluated, if result is non-zero then the statements between #if and #endif are compiled, otherwise they are skipped.
    - Syntax:

      #if constant-expression

      …….

      statements

      …….

      #endif

  - **#else:**
    - #else is used with the #if preprocessor directive. It is analogous to if..else control structure.
    - Syntax:

      #if constant-expression

      …….

      statements

      …….

      #else

      …….

statements
…….
#endif

- If the constant expression evaluates to non-zero, then the statements between #if and #else are compiled otherwise the statements between #else and #endif are compiled.

- **#elif:**
  - Nesting of #if and #else is also possible and this can be done using #elif.
  - This is analogous to the else...if ladder that we had studied in control statements.
  - Every #elif has one constant expression, first the expression is evaluated if the value is true then that part of code is compiled and all other #elif expressions are skipped, otherwise the next #elif expression is evaluated.
  - Syntax:

```
#if constant-expression1
        …….
        statements
        …….
#elif constant-expression2
        …….
        statements
        …….
#elif constant-expression3
        …….
        statements
        …….
#else
        …….
        statements
        …….
#endif
```

- **#ifdef and #ifndef:**
  - The directives #ifdef and #ifndef provide an alternative short form for combining #if with defined operator.
  - #if defined(macro_name) is equivalent to #ifdef macro_name
  - #if !defined(macro_name) is equivalent to #ifndef macro_name

- Syntax for #ifdef:

```
#ifdef macro name
        …….
        statements
        …….
#endif
```

Explanation: In the macro_name has been defined with the #define directive, then the statements between #ifdef and #endif will be compiled. If the macro_name has not been defined or was undefined using #undef then these statements are not compiled.

- Syntax for #ifndef:

```
#ifndef macro_name
        …….
        statements
        …….
#endif
```

Explanation: In the macro_name has not been defined using #define or was undefined using #undef, then the statements between #ifndef and #endif are compiled. If the macro_name has been defined, then these statements are not compiled.

- **Other Directives:**
  - **#undef Directive:**
    - The definition of a macro will exist from the #define directive till the end of the program. If we want to undefine this macro we can use the #undef directive.
    - Syntax:

```
#undef macro_name
```

    - Example:

```
#undef LIMIT
```

Explanation: Using this statement will undefine the existing macro LIMIT. After this statement every "#ifdef LIMIT" statement will evaluate to false.

- **#line**
  - The directive is used for debugging purposes.
  - Syntax:

    #line dec_const string_const

    Explanation: Here dec_const is any decimal constant and string_const is any string constant. This directive assigns dec_const and string_const to the macros _LINE_ and _FILE_ respectively. If the string_const is not specified then the macro _FILE_ remains unchanged.

  - Example:
    - ✓ *Input:*

```
int main()
{
        printf("C in depth\n");
        printf ("%d %s\n",_LINE_,_FILE_) ;
        #line 25 "myprog.c"
        printf("%d %s\n", __LINE__,__FILE__);
}
```

    - ✓ *Output:*

```
C in depth
6 C:\prog.c
25 myprog.c
```

- **#error**
  - This preprocessor directive is used for debugging purpose.
  - #error directive stops compilation and displays a fatal error message attached with it.
  - Syntax:

    #error message

  - Example:
    - ✓ *Input:*

```
int main()
{
        printf("C in depth\n");
        #error check here
        printf("Preprocessor\n");
}
```

    - ✓ *Output:*

```
error: #error check here
```

- **#pragma Directive:**
  - This directive is a special purpose directive and is used to turn on or off some features.
  - This type of directives are compiler-specific, i.e., they vary from compiler to compiler.
  - Example:

    #pragma pack(1)

    Explanation: This will stop the structure padding. It will force compiler to use 1 byte packaging.

- **Null Directive:**
  - The null directive is a line with a pound sign but contains nothing else.
  - These directives are simply deleted by the preprocessor.
  - Example:

    #
    #include “local_include.h”
    #

    Explanation: Because preprocessor deletes null directives, lines containing them remained as blank lines.

- **Stringize Operator (#):**
  - The stringize or number-sign operator ('#'), when used within a macro definition, converts a macro parameter into a string constant.
  - This operator may be used only in a macro having a specified argument or parameter list.
  - **Example:**
    - ✓ *Input*

```
#define PRINT(var, token) printf(#var " = %" #token "\n", var);
int main()
{
    int x     = 9;
    float y = 2.5;
    char z ='$';

    PRINT(x, d);
    PRINT(y, f);
    PRINT(z, c);
}
```

- *Output:*

```
x = 9
y = 2.500000
z = $
```

- **Token Pasting Operator (##):**
  - The token-pasting operator (##) within a macro definition combines two arguments.
  - It permits two separate tokens in the macro definition to be joined into a single token.
  - **Example:**
    - ✓ *Input:*

```
#define PASTE(x, y) x##y
#define MARK(sub) mark_##sub
int main()
{
        int a1 = 5, a2 = 6;
        int mark_phy = 47, mark_chem = 73, mark_math = 81;

        printf("a1 = %d\n", PASTE(a, 1));
        printf("a2 = %d\n", PASTE(a, 2));

        printf("mark_math = %d\n", MARK(math));
        printf("mark_chem = %d\n", MARK(chem));
        printf("mark_phy  = %d\n", MARK(phy));
}
```

    - ✓ *Output:*

```
a1 = 5
a2 = 6
mark_math  = 81
mark_chem  = 73
mark_phy   = 47
```

- **Macro Continuation Operator (\):**
  - A macro is normally confined to a single line.
  - The macro continuation operator (\) is used to continue a macro that is too long for a single line.

- **Example:**
  - ✓ *Input:*

```
#define MESSAGE    \
        printf("Preprocessor in C");

int main()
{
        MESSAGE
}
```

  - ✓ *Output:*

```
Processor In C
```

- **Predefined Macros:**
  - ANSI C defines a number of macros. Although each one is available for use in programming, the predefined macros should not be directly modified.

Sr.No.	Macro & Description
1	**__DATE__** The current date as a character literal in "MMM DD YYYY" format.
2	**__TIME__** The current time as a character literal in "HH:MM:SS" format.
3	**__FILE__** This contains the current filename as a string literal.
4	**__LINE__** This contains the current line number as a decimal constant.
5	**__STDC__** Defined as 1 when the compiler complies with the ANSI standard.

✓ *Input:*

```
#include <stdio.h>

int main()
{
  printf("File :%s\n",__FILE__);
  printf("Date :%s\n",__DATE__);
  printf("Time :%s\n",__TIME__);
  printf("Line :%d\n",__LINE__);
  printf("ANSI :%d\n",__STDC__);
}
```

✓ *Output:*

```
File :test.c
Date :Jun 2 2012
Time :03:36:24
Line :8
ANSI :1
```

- **Macro vs Function:**

MACRO	FUNCTION
Macro is Preprocessed.	Function is Compiled.
No Type Checking is done in Macro.	Type Checking is Done in Function.
Using Macro increases the code length.	Using Function keeps the code length unaffected.
Use of macro can lead to side effect at later stages.	Functions do not lead to any side effect in any case.
Speed of Execution using Macro is Faster.	Speed of Execution using Function is Slower.
Before Compilation, macro name is replaced by macro value.	During function call, transfer of control takes place.
Macros are useful when small code is repeated many times.	Functions are useful when large code is to be written.
Macro does not check any Compile-Time Errors.	Function checks Compile-Time Errors.
