Preprocessors in C

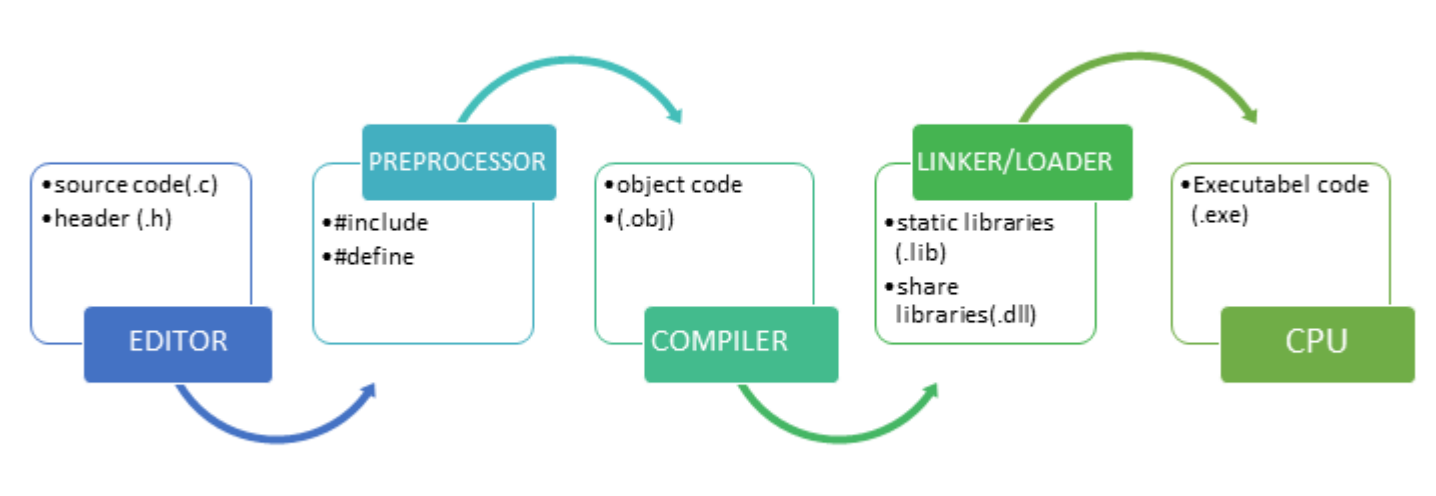
In this tutorial you will learn about preprocessors directives in C like # include, #define, #if, etc with the help of simple examples.

What are preprocessors directives in C?

The C Pre-Processor directives in C, generally denoted by the term 'CPP', functions in three different ways. it functions as a

- text substitution tool that directs the compiler to replace a certain term with a set of instructions as defined. Since it permits the user to define macros, it is also referred to as macro processors.
- insertion tool which inserts contents of other files into the source file.
- conditional tool for compilation by removing sections of codes from the file.

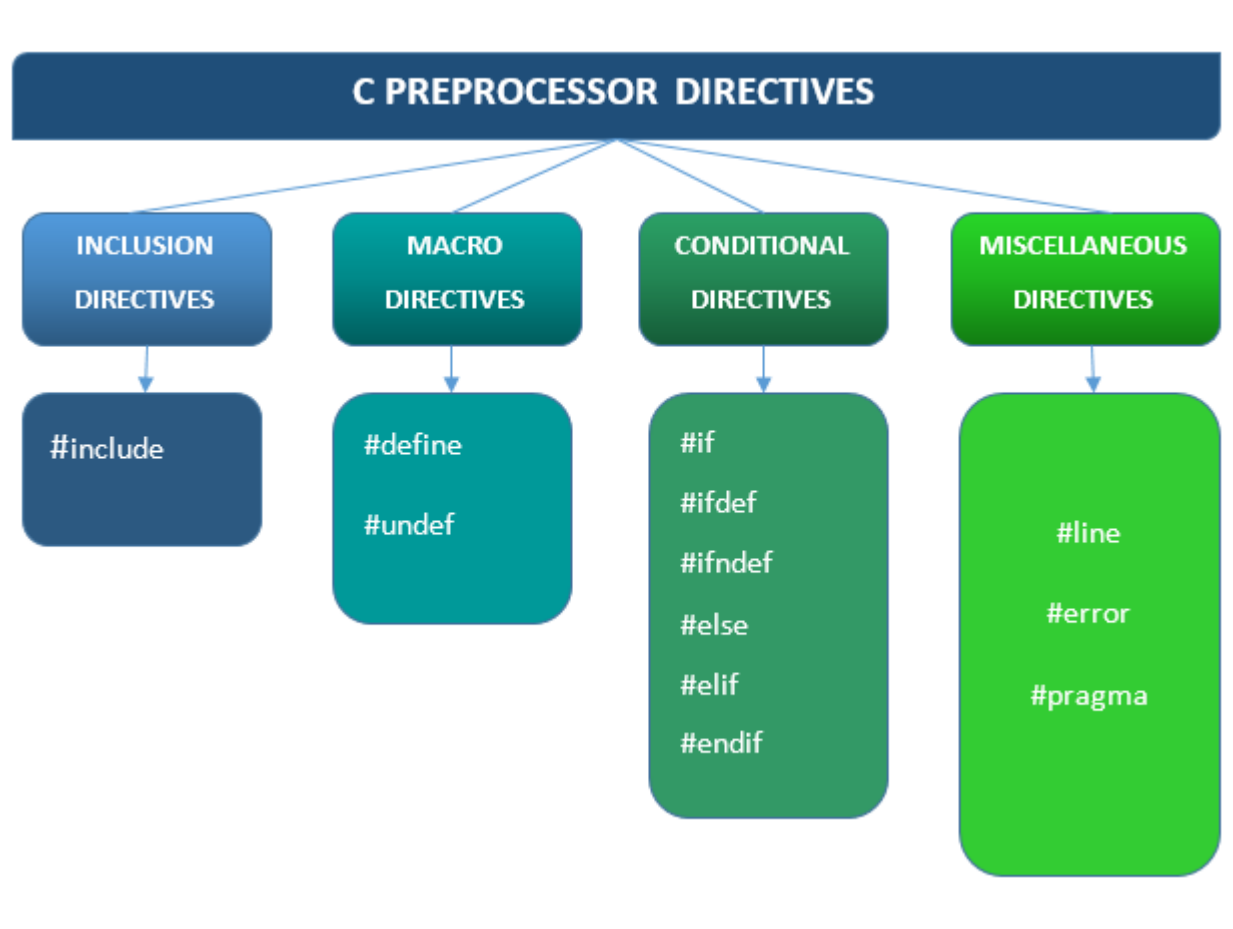
In C, we must compile a program first to run. The preprocessor is a special type of program, which processes certain parts of the program before it is processed by the compiler. This can be visualized as follows:



A hash sign (#) denotes the preprocessor inclusion. Though we can use it in any part of a program, the normal practice is defining it right at the start of it.

Preprocessor directive classification

The main preprocessor directives of the C language are classified into four groups based on the functionality.



1.Inclusion Directives

Obviously, you have noticed that there is hardly any program where we have not used the #include preprocessor. Almost every program starts like:

```
#include <stdio.h>
main()
{
}
```

It implies that all the built-in functions that are written in the header file `stdio. h` can be used inside the program. Similarly, depending on the nature of the program we can include both standard header files and user-defined header files in a program.

2.Macro Directives

2.1 #define directives

As discussed earlier in this tutorial, a macro is a code segment that can be substituted with a value. We can create macros using #define directive. The syntax is:

```
#define token value
```

Here the job of the preprocessor is to replace the identifier with the mentioned characters in each and every occasion in the program. Macros are classified into two types:

1. Object -like Macros
2. Function- like Macros

OBJECT-LIKE MACROS

Macros that usually replace an identifier with a value are categorized under object-like macros. Most commonly used to represent numeric constants. Here is the example:

```
#define LIMIT 10
```

Here, LIMIT is the macro name that replaces the value 10. observe the below example which illustrates the working of object-like macros.

```
#include<stdio.h>
#define LIMIT 10

int main()
{
    for(int i=1;i<=LIMIT;i++)
    {
        printf("%d\t",i);
    }
    return 0;
}
```

Here the compiler will read 'LIMIT' as 10 and process the program accordingly which leads to the following output:

```
1       2       3       4       5       6       7       8       9       10
```

FUNCTION LIKE MACROS OR PARAMETRISED MACROS

Parameterized form enhances the power of macros by a great deal, especially in performing mathematical calculations. It enables macro work like a function based on variables. Suppose we want to determine the cube of a number. To do this, normally we have to create a function like:

```
int cube(int a)
{
   return a*a*a;
}
```

Using parameterized macros, we can do the same operation in a single line.

```
#define cube(a) ((a) * (a) *(a))
```

Here is the example:

```
#include<stdio.h>
#define CUBE(a) ((a)*(a)*(a))

int main()
{
  printf("%d",CUBE(10));
  return 0;
}
```

Output:

```
1000
```

We must remember that before using the macros in a program, we must define them using the directive #define

2.2 #undef Directives

the undef directive in C is used to undefine a macro , clearly saying to cancel the macro which is already defined. The syntax is :

```
#undef token
```

Following program demonstrates the working of #undef directive.

```
#include<stdio.h>
#define CUBE(a) ((a)*(a)*(a))
#define LIMIT 20
#undef LIMIT
int main()
{
    printf("%d",CUBE(LIMIT));

    return 0;
}
```

In the above code snippet we have undefined the macro LIMIT and hence when we call the function the output will be obviously an error as follows

```
error: 'LIMIT' undeclared (first use in this function)|
```

3.Conditional Directives

Conditional directives in C programming are used to check whether a macro is already defined in the program or not and instructs the preprocessor either to include or discard the set of codes. We can achieve conditional compilation with the help of directives #ifdef,#ifndef, #if, #else, #elif, #endif. Now lets learn about them one by one:

3.1 #ifdef Directive:

The #ifdef preprocessor directive checks the presence of a macro defined by #define and allow a segment of code to be compiled if the condition met. Otherwise the portion of code will not be compiled. The prototype of #ifdef is as follows:

```
#ifdef MACRO
    //Set of codes
#endif
```

3.2 #ifndef Directive

This directive checks the absence of a macro and executes the set of codes placed between ifndef and endif. The syntax is as follows:

```
#ifndef MACRO
    //Set of codes
#endif
```

Now let's observe how #ifdef and ifndef can be implemented in a program.

```
#include<stdio.h>
#define CUBE(a) ((a)*(a)*(a))
#define LIMIT 20


int main()
{
    #ifdef LIMIT
    printf("Given limit is %d",LIMIT);
    #endif // LIMIT

    #ifndef CUBE
    printf("Cube %d :",CUBE(3));
    #endif // CUBE

    return 0;
}
```

Output:

```
Given limit is 20
```

3.3 #if, #else and #elif

All these directives execute codes only if they meet the specified condition or expression.

The syntax of #if as follows:

```
#if macro_expression
    //Set of codes
#endif
```

Syntax of #else:

```
#if macro_expression
    //Set of codes for if part
#else
 // set of codes for else part
#endif
```

Syntax of #elif

```
#if macro_expression
    //Set of codes for if part
#elif
 // set of codes for elif part
#else
 //set of codes for else part
#endif
```

```
#include<stdio.h>
#include<conio.h>

#define AGE

int main()
{

    #if AGE >=18
    printf("YOU ARE ELIGIBLE FOR VOTING");
    #else
    printf("YOU ARE NOT ELIGIBLE FOR VOTING");
    #endif
}
```

Preprocessior Operator

There are different types of preprocessor operators. As of now, we will discuss the most used ones. Here they are:

i. THE MACRO CONTINUATION (\) OPERATOR

Normally macros like #define work till it encounters a new line. Sometimes we need the macros to be large enough to accommodate all the characters and writing all these in a single line hampers the readability of the program. The continuation operator ('\') tells the compiler to consider the next line as a part of the previous one. Here is an example:

```
//Continuation Operator in C

#include<stdio.h>
# define CUBE(a) \
        printf("Cube of %d is %d\n",a,((a)*(a)*(a)));
int main()
{
    CUBE(6);
    return 0;
}
```

Output:

```
Cube of 6 is 216
```

ii. THE STRINGIZE (#) OPERATOR:

The STRINGIZE operator is one among the preprocessor operators which is used to manipulate strings. As the name suggests the stringize operator of a macro has the ability of converting parameters of the argument to string constants. This operator only works with macros that take arguments only.

```
#define Macro_name(arg) #arg
```

Consider the following example:

```
#include <stdio.h>
#define display(text)  #text

int main(void)
 {
    printf(display(WELCOME TO LEARN E TUTORIALS));
    return 0;
}
```

Output:

```
WELCOME TO LEARN E TUTORIALS
```

Explanation: In this code we are passing a macro to the printf() function. Since a macro gets compiled before the program, the preprocessor will expand the display(WELCOME TO LEARN E TUTORIALS) into “WELCOME TO LEARN E TUTORIALS”. Later on we have our printf() function as printf(“WELCOME TO LEARN E TUTORIALS”) and hence the output.

iii. THE TOKEN PASTING (##) OPERATOR:

The token pasting operator is a preprocessor operator used for glueing two separate tokens given as arguments inside a macro. The notation '##' acts as the joiner by combining tokens placed on its either sides. The ## operators can be used in following tokens:

1. identifiers like variable names, function names etc. 2. keywords and variable names like int, while, volatile etc. 3. data types like strings, numbers, characters, true or false. 4. mathematical operators and punctuators like (,=,); etc.

Syntx

```
#define Macro_name(arg1,arg2,...,argN) arg1##arg2##...##argN
```

Examine the following program:

```
#include <stdio.h>

#define join(x, y)  x##y

int main()
{
 printf("join(20, 30) = %d\n", join(20, 30));

 return 0;
}
```

Output:

```
join(20, 30) = 2030
```

iv. THE DEFINED() OPERATOR:

The job of this operator is to check whether the identifier has been defined by #define or not. It has two binary outputs true and false. If the identifier is there, it returns 1. Otherwise, it will flag '0'. Here is an example describing the application of #defined.

```
// defined operator
#include <stdio.h>
#if !defined (FUNCTION)
      #define FUNCTION "Hello World!!"
       #endif

int main(void)
 {
        printf("\n\t %s ", FUNCTION);
        return 0;
}
```