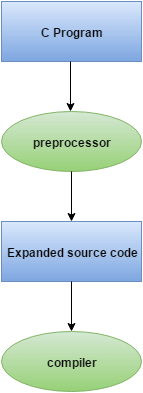
**C Preprocessor Directives**

The C preprocessor is a micro processor that is used by compiler to transform your code before compilation. It is called micro preprocessor because it allows us to add macros.

Note: Proprocessor direcives are executed before compilation.



All preprocessor directives starts with hash # symbol.

Let's see a list of preprocessor directives.

* #include
* #define
* #undef
* #ifdef
* #ifndef
* #if
* #else
* #elif
* #endif
* #error
* #pragma

**C File Inclusion (#include)**

The #include preprocessor directive is used to paste code of given file into current file. It is used include system-defined and user-defined header files. If included file is not found, compiler renders error.

By the use of #include directive, we provide information to the preprocessor where to look for the header files. There are two variants to use #include directive.

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

The **#include <filename>** tells the compiler to look for the directory where system header files are held. In UNIX, it is \usr\include directory.

The **#include "filename"** tells the compiler to look in the current directory from where program is running.

#include directive example

Let's see a simple example of #include directive. In this program, we are including stdio.h file because printf() function is defined in this file.

1. #include<stdio.h>
2. **int** main(){
3. printf("Hello C");
4. **return** 0;
5. }

Output:

Hello C

#include notes:

**Note 1:** In #include directive, comments are not recognized. So in case of #include <a//b>, a//b is treated as filename.

**Note 2:** In #include directive, backslash is considered as normal text not escape sequence. So in case of #include <a\nb>, a\nb is treated as filename.

**Note 3:** You can use only comment after filename otherwise it will give error.

**C Macros**

C macros provide a potent method for code reuse and simplification. They let programmers construct ***symbolic names*** or phrases that are changed to certain values before the compilation process begins. The use of more ***macros*** makes code easier to ***comprehend, maintain***, and makes mistakes less likely. In this article, we'll delve deeper into the concept of C macros and cover their advantages, ideal usage scenarios, and potential hazards.

A macro is a segment of code which is replaced by the value of macro. Macro is defined by #define directive. There are two types of macros:

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

**Object-like Macros**

The object-like macro is an identifier that is replaced by value. It is widely used to represent numeric constants. For example:

1. #define PI 3.14

Here, PI is the macro name which will be replaced by the value 3.14.

**Function-like Macros**

The function-like macro looks like function call. For example:

1. #define MIN(a,b) ((a)<(b)?(a):(b))

Here, MIN is the macro name.

Visit [#define](https://www.javatpoint.com/c-preprocessor-define) to see the full example of object-like and function-like macros.

**C Predefined Macros**

C defines many predefined macros that can be used in c program.

|  |  |  |
| --- | --- | --- |
| **No.** | **Macro** | **Description** |
| 1 | \_DATE\_ | represents current date in "MMM DD YYYY" format. |
| 2 | \_TIME\_ | represents current time in "HH:MM:SS" format. |
| 3 | \_FILE\_ | represents current file name. |
| 4 | \_LINE\_ | represents current line number. |
| 5 | \_STDC\_ | It is defined as 1 when compiler complies with the ANSI standard. |

**C predefined macros example**

*File: simple.c*

1. #include<stdio.h>
2. **int** main(){
4. printf("Date :%s\n", \_\_DATE\_\_ );
5. printf("Time :%s\n", \_\_TIME\_\_ );

8. **return** 0;
9. }

**Advantages of Using Macros:**

There are various advantages of Macros in C. Some main advantages of C macros are as follows:

**Code reuse:** By allowing developers to declare a piece of code just once and use it several times, ***macros*** help to promote modular programming and minimize code duplication.

**Code abbreviation: *Macros*** make it possible to ***write clear, expressive code*** that is simpler to read and comprehend the intentions of the programmer.

**Performance Optimization:** By minimizing ***function call overhead***, macros may be utilized to ***optimize code execution***. For instance, it is possible to inline brief pieces of code using function-like macros

**C #pragma**

The #pragma preprocessor directive is used to provide additional information to the compiler. The #pragma directive is used by the compiler to offer machine or operating-system feature.

Syntax:

1. #pragma token

Different compilers can provide different usage of #pragma directive.

The turbo C++ compiler supports following #pragma directives.

1. #pragma argsused
2. #pragma exit
3. #pragma hdrfile
4. #pragma hdrstop
5. #pragma inline
6. #pragma option
7. #pragma saveregs
8. #pragma startup
9. #pragma warn

Let's see a simple example to use #pragma preprocessor directive.

1. #include<stdio.h>
2. #include<conio.h>
4. **void** func() ;
6. #pragma startup func
7. #pragma exit func
9. **void** main(){
10. printf("\nI am in main");
11. getch();
12. }
14. **void** func(){
15. printf("\nI am in func");
16. getch();
17. }

Output:

I am in func

I am in main

I am in func