Preprocessors in C

By Abdallah Ghazy

- 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 replacement** tool and it instructs the compiler to do the required pre-processing before the actual compilation.
- preprocessing is the first step in the compilation of a C code.
- It occurs before the tokenization step.
- One of the important functions of a preprocessor is to include the header files that contain the library functions used in the program.
- The preprocessor in C also defines the constants and expands the macros.
- The preprocessor statements in C are called directives.
- A preprocessor section of the program always appears at the top of the C code.
- Each preprocessor statement starts with the hash **(#)** symbol.

Preprocessor Directives in C

Directive	Description
# define	Substitutes a preprocessor macro.
#include	Inserts a particular header from another file.
#undef	Undefines a preprocessor macro.
#ifdef	Returns true if this macro is defined.
#ifndef	Returns true if this macro is not defined.
#if	Tests if a compile time condition is true.
#else	The alternative for #if.
#elif	#else and #if in one statement.
#endif	Ends preprocessor conditional.
#error	Prints error message on stderr.
#pragma	Issues special commands to the compiler, using a standardized method.

Preprocessors Examples

- ➢ *Analyze the following examples to understand various directives.*

- ✓ This **#define** directive tells the compiler to replace the instances of MAX_ARRAY_LENGTH with 20. Use **#define** for constants to increase readability.

.C

```c
#include <stdio.h>
#define MAX_ARRAY_LENGTH 20

int main() {

    printf("MAX_ARRAY_LENGTH = %d\n", MAX_ARRAY_LENGTH);

    return 0;
}
```

After preprocessing

.i

```c
# 4 ".\\macros.c"
int main() {

 printf("MAX_ARRAY_LENGTH = %d\n", 20);

    return 0;
}
```

- ✓ Now, take a look at the following #define and **#undef** directives. They tell the compiler to undefine existing FILE_SIZE and define it as 42.

.C

```c
#include <stdio.h>

int main() {
    #define FILE_SIZE 42
    printf("FILE_SIZE = %d\n",FILE_SIZE );

    #undef  FILE_SIZE
    #define FILE_SIZE 33
    printf("FILE_SIZE = %d\n",FILE_SIZE );

    return 0;
}
```

After preprocessing

.i

```c
# 3 ".\\macros.c"
int main() {

 printf("FILE_SIZE = %d\n",42 );



 printf("FILE_SIZE = %d\n",33 );

    return 0;
}
```

- ✓ The following directive tells the compiler to define FILE_SIZE only if FILE_SIZE isn't already defined.

.C

```c
#include <stdio.h>

int main() {
    #define FILE_SIZE 42
    printf("FILE_SIZE = %d\n",FILE_SIZE );

    #ifndef FILE_SIZE
    #define FILE_SIZE 33
    #endif

    printf("FILE_SIZE = %d\n",FILE_SIZE );

    return 0;
}
```

After preprocessing

.i

```c
# 3 ".\\macros.c"
int main() {

 printf("FILE_SIZE = %d\n",42 );





 printf("FILE_SIZE = %d\n",42 );

    return 0;
}
```

✓ The following directives tell the compiler to get "stdio.h" from the System Libraries and add the text to the current source file.

.c

```c
#include <stdio.h>

int main() {

    return 0;
}
```

.i

```c
# 1 ".\\macros.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 ".\\macros.c"
# 1 "c:\\mingw\\include\\stdio.h" 1 3
# 38 "c:\\mingw\\include\\stdio.h" 3

# 39 "c:\\mingw\\include\\stdio.h" 3
# 56 "c:\\mingw\\include\\stdio.h" 3
# 1 "c:\\mingw\\include\\_mingw.h" 1 3
# 55 "c:\\mingw\\include\\_mingw.h" 3

# 56 "c:\\mingw\\include\\_mingw.h" 3
# 66 "c:\\mingw\\include\\_mingw.h" 3
# 1 "c:\\mingw\\include\\msvcrtver.h" 1 3
# 35 "c:\\mingw\\include\\msvcrtver.h" 3

# 36 "c:\\mingw\\include\\msvcrtver.h" 3
# 67 "c:\\mingw\\include\\_mingw.h" 2 3






# 1 "c:\\mingw\\include\\w32api.h" 1 3
# 35 "c:\\mingw\\include\\w32api.h" 3

# 36 "c:\\mingw\\include\\w32api.h" 3
# 59 "c:\\mingw\\include\\w32api.h" 3
# 1 "c:\\mingw\\include\\sdkddkver.h" 1 3
# 35 "c:\\mingw\\include\\sdkddkver.h" 3

# 36 "c:\\mingw\\include\\sdkddkver.h" 3
# 60 "c:\\mingw\\include\\w32api.h" 2 3
# 74 "c:\\mingw\\include\\_mingw.h" 2 3
# 57 "c:\\mingw\\include\\stdio.h" 2 3
# 69 "c:\\mingw\\include\\stdio.h" 3
# 1 "c:\\mingw\\lib\\gcc\\mingw32\\6.3.0\\include\\stddef.h" 1 3 4
# 216 "c:\\mingw\\lib\\gcc\\mingw32\\6.3.0\\include\\stddef.h" 3 4

# 216 "c:\\mingw\\lib\\gcc\\mingw32\\6.3.0\\include\\stddef.h" 3 4
typedef unsigned int size_t;
# 328 "c:\\mingw\\lib\\gcc\\mingw32\\6.3.0\\include\\stddef.h" 3 4
typedef short unsigned int wchar_t;
# 357 "c:\\mingw\\lib\\gcc\\mingw32\\6.3.0\\include\\stddef.h" 3 4
typedef short unsigned int wint_t;
# 70 "c:\\mingw\\include\\stdio.h" 2 3
```

Line 450, Column 1

DIFFERENCE BETWEEN COMPILERS VS INTERPRETERS IN C LANGUAGE?

Compilers	**Interpreters**
Compiler reads the entire source code of the program and converts it into binary code. This process is called compilation. Binary code is also referred as machine code, executable, and object code.	Interpreter reads the program source code one line at a time and executing that line. This process is called interpretation.
Program speed is fast.	Program speed is slow.
One time execution. Example: C, C++	Interpretation occurs at every line of the program. Example: BASIC

Preprocessor Operators in C

Preprocessor operators are special symbol(s) that are used in the context of the **#define** directive. These operators are also called preprocessor-specific operators.

Operator	Action
Continuation operator (/)	Used to continue a macro that is too long for a single line.
Stringizing operator (#)	Causes the corresponding actual argument to be enclosed in double quotation marks
Token-pasting operator (##)	Allows tokens used as actual arguments to be concatenated to form other tokens
defined operator	Simplifies the writing of compound expressions in certain macro directives

Continuation Operator (/)

This operator is used where the macro is quite complex, and spreads over multiple lines. In case of a complex logic inside macro expansion, you'll need to break the line and write code that spreads over more than one lines. In such cases macro continuation operator is very helpful.

Example 1: Preprocessor Continuation Operator (/)

```
#include <stdio.h>

#define  message() { \
   printf("TutorialsPoint Library contains\n"); \
   printf("High quality Programming Tutorials"); \
}

int main() {

   message();
   return 0;
}
```

```
PS C:\MinGW\bin> .\gcc.exe .\macros.c -o macros.exe
PS C:\MinGW\bin> .\macros.exe
TutorialsPoint Library contains
High quality Programming Tutorials
PS C:\MinGW\bin>
```

```
# 8 ".\\macros.c"
int main() {

   { printf("TutorialsPoint Library contains\n"); printf("High quality Programming Tutorials"); };
   return 0;
}
```

Example 2: Preprocessor Continuation Operator (/)

```c
#include <stdio.h>

#define SHAPE(x) switch(x) { \
    case 1: printf("1. CIRCLE\n"); break; \
    case 2: printf("2. RECTANGLE\n"); break; \
    case 3: printf("3. SQUARE\n"); break; \
    default: printf("default. LINE\n"); \
}

int main() {

    SHAPE(1);

    SHAPE(2);

    SHAPE(3);

    SHAPE(0);
    return 0;
}
```

```
PS C:\MinGW\bin> .\gcc.exe -E .\macros.c -o macros.i
PS C:\MinGW\bin> .\gcc.exe .\macros.c -o macros.exe
PS C:\MinGW\bin> .\macros.exe
1. CIRCLE
2. RECTANGLE
3. SQUARE
default. LINE
PS C:\MinGW\bin>
```

```c
# 10 ".\\macros.c"
int main() {

    switch(1) { case 1: printf("1. CIRCLE\n"); break; case 2: printf("2. RECTANGLE\n"); break; case 3: printf("3.
    SQUARE\n"); break; default: printf("default. LINE\n"); };

    switch(2) { case 1: printf("1. CIRCLE\n"); break; case 2: printf("2. RECTANGLE\n"); break; case 3: printf("3.
    SQUARE\n"); break; default: printf("default. LINE\n"); };

    switch(3) { case 1: printf("1. CIRCLE\n"); break; case 2: printf("2. RECTANGLE\n"); break; case 3: printf("3.
    SQUARE\n"); break; default: printf("default. LINE\n"); };

    switch(0) { case 1: printf("1. CIRCLE\n"); break; case 2: printf("2. RECTANGLE\n"); break; case 3: printf("3.
    SQUARE\n"); break; default: printf("default. LINE\n"); };
    return 0;
}
```

Stringizing Operator (#)

Sometimes you may want to convert a macro argument into a string constant. The number-sign or "stringizing" operator (#) converts macro parameters to string literals without expanding the parameter definition. This operator may be used only in a macro having a specified argument or parameter list.

Example 1: Stringizing Operator

.C

```c
#include <stdio.h>
#define stringize(x) printf(#x "\n")

int main() {

    stringize(Welcome To TutorialsPoint);
    stringize("The Largest Tutorials Library");
    stringize("Having video and Text Tutorials on Programming Languages");
}
```

After preprocessing .i

```c
# 4 ".\\macros.c"
int main() {

   printf("Welcome To TutorialsPoint" "\n");
   printf("\"The Largest Tutorials Library\"" "\n");
   printf("\"Having video and Text Tutorials on Programming Languages\"" "\n");
}
```

```
PS C:\MinGW\bin> .\gcc.exe -E .\macros.c -o macros.i
PS C:\MinGW\bin> .\gcc.exe .\macros.c -o macros.exe
PS C:\MinGW\bin> .\macros.exe
Welcome To TutorialsPoint
"The Largest Tutorials Library"
"Having video and Text Tutorials on Programming Languages"
PS C:\MinGW\bin>
```

Token Pasting Operator (##)

The double-number-sign or token-pasting operator (**##**), which is sometimes called the merging or combining operator. It is often useful to merge two tokens into one while expanding macros.

When a macro is expanded, the two tokens on either side of each "**##**" operator are combined into a single token, which then replaces the "**##**" and the two original tokens in the macro expansion.

Example 1: Token Pasting Operator (##)

.c

```c
#include <stdio.h>

#define PASTE(arg1,arg2) arg1##arg2

int main() {

    int value_1 = 1000;
    printf("value_1 = %d\n", PASTE(value_,1));
}
```

```
PS C:\MinGW\bin> .\gcc.exe -E .\macros.c -o macros.i
PS C:\MinGW\bin> .\gcc.exe .\macros.c -o macros.exe
PS C:\MinGW\bin> .\macros.exe
value_1 = 1000
PS C:\MinGW\bin>
```

After preprocessing

.i

```c
# 5 ".\\macros.c"
int main() {

    int value_1 = 1000;
    printf("value_1 = %d\n", value_1);
}
```

Example 2: Token Pasting Operator (##)

.c

```c
#include <stdio.h>
#define TOKENS(X, Y) X##Y

int main() {

    printf("value1: %d\n",TOKENS(12,20));
    printf("value2: %d\n",TOKENS(12,20)+10);
    return 0;
}
```

```
PS C:\MinGW\bin> .\gcc.exe .\macros.c -o macros.exe
PS C:\MinGW\bin> .\macros.exe
value1: 1220
value2: 1230
```

After preprocessing

.i

```c
# 4 ".\\macros.c"
int main() {

    printf("value1: %d\n",1220);
    printf("value2: %d\n",1220 +10);
    return 0;
}
```

The defined Operator

The defined preprocessor operator can only be used as a part of **#if** and **#elif** directives.

```
#if defined(macro1)
// code
#elif defined(macro2)
// code
#endif
```

Example 1: defined Operator

After preprocessing

.c

```c
#include <stdio.h>

#define DEBUG 1

int main() {

    #if defined(DEBUG)
    printf("DEBUG mode is on.\n");
    #else
        printf("DEBUG mode is off.\n");
    #endif

    return 0;
}
```

.i

```c
# 5 ".\\macros.c"
int main() {


    printf("DEBUG mode is on.\n");




    return 0;
}
```

```
PS C:\MinGW\bin> .\gcc.exe -E .\macros.c -o macros.i
PS C:\MinGW\bin> .\gcc.exe .\macros.c -o macros.exe
PS C:\MinGW\bin> .\macros.exe
DEBUG mode is on.
PS C:\MinGW\bin>
```

Example 2: defined Operator

.C

```
#include <stdio.h>

#define square(x) ((x) * (x))

int main(){

    #if defined square
    printf("The square of the given number is: %d", square(5));
    #endif

    return 0;
}
```

After preprocessing .i

```
# 5 ".\\macros.c"
int main(){


    printf("The square of the given number is: %d", ((5) * (5)));


    return 0;
}
```

```
PS C:\MinGW\bin> .\gcc.exe -E .\macros.c -o macros.i
PS C:\MinGW\bin> .\gcc.exe .\macros.c -o macros.exe
PS C:\MinGW\bin> .\macros.exe
The square of the given number is: 25
PS C:\MinGW\bin>
```

Macros in C

Macros in C are the names given to specific constant values or code statements which are replaced with their value/code before the compilation processor. **C Macros** are defined using the **#define** preprocessor directive.

Macros are useful for code reusability, defining constant values, defining inline functions, and conditional compilations.

The following are the different types of C macros that we are going to cover in this tutorial –

1) Object-like Macros
2) Function-like Macros
3) Chained Macros
4) Variadic Macros
5) Predefined Macros

1) Object-like Macros

A macro that defines a constant is an object-like macro.

```
#define name value
```

Example of Object-like Macros

.C

```c
#include <stdio.h>

// Defining macros
#define PI 3.14
#define MAX 10

int main() {
    // Printing the values
    printf("Value of PI = %d\n", PI);
    printf("Value of MAX = %d\n", MAX);

    return 0;
}
```

After preprocessing

.i

```c
# 7 ".\\macros.c"
int main() {

    printf("Value of PI = %d\n", 3.14);
    printf("Value of MAX = %d\n", 10);

    return 0;
}
```

```
Value of PI = 1374389535
Value of MAX = 10
PS C:\MinGW\bin>
```

2) Function-like Macro

To define a function-like macro, you use the same **"#define"** directive, but you put a pair of parentheses immediately after the macro name, with one or more arguments. Such a macro is expanded only when its name appears with a pair of parentheses after it.

Syntax

A function-like macro is defined as follows –

```
#define macro_name([parameter_list]) replacement_text
```

Example of Function-like Macros

.C

```c
#include <stdio.h>

#define square(x) ((x) * (x))

int main(){

    int x = 5;
    printf("x: %d \tSquare of x: %d", x, square(x));
    return 0;
}
```

```
PS C:\MinGW\bin> .\gcc.exe -E .\mac
PS C:\MinGW\bin> .\gcc.exe .\macros
PS C:\MinGW\bin> .\macros.exe
x: 5    Square of x: 25
PS C:\MinGW\bin>
```

After preprocessing

.i

```c
# 5 ".\\macros.c"
int main(){

    int x = 5;
    printf("x: %d \tSquare of x: %d", x, ((x) * (x)));
    return 0;
}
```

Rules for Defining Function-like Macros

- A macro can be defined without arguments
- A macro can be defined with a fixed number of arguments
- A macro can be defined with a variable number of arguments

Function-like Macros without Arguments

Example 1

.c

```c
#include <stdio.h>

#define MESSAGE() printf("Hello World");

int main() {

    int x = 5;
    MESSAGE();
    return 0;
}
```

After preprocessing

.i

```c
# 5 ".\\macros.c"
int main() {

    int x = 5;
    printf("Hello World");;
    return 0;
}
```

```
PS C:\MinGW\bin> .\gcc.exe -E .\macros.c -o
PS C:\MinGW\bin> .\gcc.exe .\macros.c -o mac
PS C:\MinGW\bin> .\macros.exe
Hello World
```

Some standard libraries also provide macro definitions in it. For example, the getchar() macro when expanded, implements the getc() function as follows –

```c
#define getchar() getc(stdin)
```

Similarly, the putchar() macro encapsulates the putc() function –

```c
#define putchar(x) putc(x, stdout)
```

Example 2

.c

```c
#include <stdio.h>

#define LOOP(x) {\
    for (int i = 1; i <= (x); i++) {\
        printf("Iteration no: %d\n", i);\
    }\
}

int main() {
    int x = 3;
    LOOP(x)
    return 0;
}
```

```
PS C:\MinGW\bin> .\macros.exe
Iteration no: 1
Iteration no: 2
Iteration no: 3
PS C:\MinGW\bin>
```

After preprocessing .i

```c
# 9 ".\\macros.c"
int main() {
    int x = 3;
    { for (int i = 1; i <= (x); i++) { printf("Iteration no: %d\n", i); }}
    return 0;
}
```

3) Chained Macros

When we have a macro nested inside another macro, they are called Chained Macros.

.C

```c
#include <stdio.h>

#define PI 3.142
#define CIRCUMFERENCE(x) (2*PI*x)

int main(){

    int x = 5;
    printf("Circumference of a circle with radius %d is %f", x, CIRCUMFERENCE(x));
    return 0;
}
```

After preprocessing .i

```c
# 6 ".\\macros.c"
int main(){

    int x = 5;
    printf("Circumference of a circle with radius %d is %f", x, (2*3.142*x));
    return 0;
}
```

```
PS C:\MinGW\bin> .\gcc.exe -E .\macros.c -o macros.i
PS C:\MinGW\bin> .\gcc.exe .\macros.c -o macros.exe
PS C:\MinGW\bin> .\macros.exe
Circumference of a circle with radius 5 is 31.420000
PS C:\MinGW\bin>
```

4) Variadic Macros

You can also define a macro with variable number of arguments or variadic macros.

A macro with variable-length argument is a feature that enables a macro to accept any number of arguments. You can pass positional as well as keyword arguments to a macro.

When a variadic macro is defined, the ellipsis (…) is given as an argument to capture variable number of arguments.

To use variadic macros, the ellipsis may be specified as the final formal argument in a macro definition.

This sequence of tokens replaces the identifier __VA_ARGS__ in the macro body wherever it appears.

The __VA_ARGS__ is replaced by all of the arguments that match the ellipsis, including commas between them. Note that the variadic macros can be used only in the C99 compatible C compilers and above.

Example of Variadic Macros

.C

```c
#include <stdio.h>

#define MAX_ARGS 5      // Define maximum number of allowed arguments
#define ARGS(x, ...) do {\
   printf("%d\n", ##__VA_ARGS__); \
} while (0)

int main() {

   ARGS(1, 2, 3);        // 3 arguments
   ARGS(1, 2, 3, 4, 5, 6);  // 6 arguments
   return 0;
}
```

```
PS C:\MinGW\bin> .\macros.exe
2
2
PS C:\MinGW\bin>
```

After preprocessing

.i

```c
# 8 ".\\macros.c"
int main() {

   do { printf("%d\n", 2, 3); } while (0);
   do { printf("%d\n", 2, 3, 4, 5, 6); } while (0);
   return 0;
}
```

5) 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.

Macro	Description
__DATE__	The current date as a character literal in "MMM DD YYYY" format.
__TIME__	The current time as a character literal in "HH:MM:SS" format.
__FILE__	This contains the current filename as a string literal.
__LINE__	This contains the current line number as a decimal constant.
__STDC__	Defined as 1 when the compiler complies with the ANSI standard.

.C

```c
#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__ );
}
```

After preprocessing

.i

```c
# 3 ".\\macros.c"
int main() {

    printf("File :%s\n", ".\\macros.c" );
    printf("Date :%s\n", "Jul 16 2024" );
    printf("Time :%s\n", "14:02:52" );
    printf("Line :%d\n", 8 );
    printf("ANSI :%d\n", 1 );
}
```

```
PS C:\MinGW\bin> .\macros.exe
File :.\macros.c
Date :Jul 16 2024
Time :14:02:53
Line :8
ANSI :1
PS C:\MinGW\bin>
```

Quiz:

Create a function factory and use it with a macro.

- Use one macro to create two functions: one that multiplies the input argument by 4 and another that multiplies the input argument by 2.
- The name of the first function will be fun_quadruple(int x).
- The name of the second function will be fun_double(int x).

```c
#include <stdio.h>

#define CREATE_MULTIPLY_FUNCTIONS(NAME, FACTOR) \
    int fun_##NAME(int x) { return x * FACTOR; }

CREATE_MULTIPLY_FUNCTIONS(quadruple, 4)
CREATE_MULTIPLY_FUNCTIONS(double, 2)

int main() {
    int x = 5;
    printf("fun_quadruple(%d) = %d\n", x, fun_quadruple(x));
    printf("fun_double(%d) = %d\n", x, fun_double(x));
    return 0;
}
```

.C

After preprocessing

```c
# 6 ".\\macros.c"
int fun_quadruple(int x) { return x * 4; }
int fun_double(int x) { return x * 2; }

int main() {
    int x = 5;
    printf("fun_quadruple(%d) = %d\n", x, fun_quadruple(x));
    printf("fun_double(%d) = %d\n", x, fun_double(x));
    return 0;
}
```

.i

```
PS C:\Users\Abdallah Ghazy> CD C:\MinGW\bin
PS C:\MinGW\bin> .\gcc.exe -E .\macros.c -o macros.i
PS C:\MinGW\bin> .\gcc.exe .\macros.c -o macros.exe
PS C:\MinGW\bin> .\macros.exe
fun_quadruple(5) = 20
fun_double(5) = 10
PS C:\MinGW\bin>
```

conditional compilation in C

n C programming language, several directives control the selective compilation of portions of the program code.

- #if
- #else
- #elif
- #endif

The general form of **#if** is as follows −

```
#if constant_expression
  statement sequence
#endif
```

else works much like the C keyword else.

#elif means "else if" and establishes an if else-if compilation chain.

Amongst other things, #if provides an alternative method of "commenting out" code.

For example,

```
#if 0
  printf("#d", total);
#endif
```

Here, the compiler will ignore printf("#d", total);

✓ The following directive tells the compiler to define FILE_SIZE only if FILE_SIZE isn't already defined.

After preprocessing

.c

```c
#include <stdio.h>

int main() {
    #define FILE_SIZE 42
    printf("FILE_SIZE = %d\n",FILE_SIZE );

    #ifndef FILE_SIZE
    #define FILE_SIZE 33
    #endif

    printf("FILE_SIZE = %d\n",FILE_SIZE );

    return 0;
}
```

.i

```c
# 3 ".\\macros.c"
int main() {

 printf("FILE_SIZE = %d\n",42 );




 printf("FILE_SIZE = %d\n",42 );

   return 0;
}
```

Macros Vs Functions

No	Macro	Function
1	Macro is **Preprocessed**	Function is **Compiled**
2	**No Type Checking**	**Type Checking** is Done
3	**Code** Length **Increases**	**Code** Length remains **Same**
–		
5	Speed of Execution is **Faster**	Speed of Execution is **Slower**
6	Before Compilation macro name is replaced by macro value	During function call , Transfer of Control takes place
7	Useful where small code appears many time	Useful where large code appears many time
8	Generally Macros do not extend beyond one line	Function can be of any number of lines

Header Files in C

The **#include preprocessor** directive is used to make the definitions of functions, constants and macros etc. from one file, usually called as a header file, available for use in another C code. A header file has "**.h**" extension from which you can include the forward declarations of one or more predefined functions, constants, macros etc. The provision of header files in C facilitates a modular design of the program.

System Header Files

The C compiler software is bundled with many pre-compiled header files. These are called **system header files**. A well-known example is "stdio.h" – a header file included in almost every C program.

Each of the system header files contains a number of utility functions. These functions are often called **library functions**. For example, printf() and scanf() functions, needed for performing IO operations, are the library functions available in the "stdio.h" header file.

Syntax to Include Header Files in C

A header file is loaded with the **#include directive**. Its usage follows either of the following two methods –

```
#include <filename.h>
```

The name of the header file put inside angular brackets if it is available in system/default directory.

```
#include "filename.h"
```

The name of the header file put inside double quotation marks for user defined or non-standard header files available in same directory as source file

The **#include** directive works by directing the C preprocessor to scan the specified file as input before continuing with the rest of the current source file. The output from the preprocessor contains the output already generated, followed by the output resulting from the included file, followed by the output that comes from the text after the **#include** directive.

#pragma Directive in C

What is #pragma Directive in C?

The preprocessor directive **#pragma** is used to provide additional information to the compiler in C/C++ language. This is used by the compiler to provide some special features.

Note that **pragmas are compiler dependent**. Not all the pragma directives are supported by all the compilers.

Syntax

Here is the **syntax** of using a #pragma directive in C/C++ language −

```
#pragma token_name
```

Types of Pragma Directives in C

Directive	Description
#pragma startup	Before the execution of main(), the function specified in pragma is needed to run.
#pragma exit	Before the end of program, the function specified in pragma is needed to run.
#pragma warn	Used to hide the warning messages.
#pragma GCC dependency	Checks the dates of current and other file. If other file is recent, it shows a warning message.
#pragma GCC system_header	It treats the code of current file as if it came from system header.
#pragma GCC poison	Used to block an identifier from the program.
#pragma once	Compiler loads the header file only once.

#pragma startup and exit

These pragma directives are executed before and after the main() function. Not all the compilers support these directives.

Example

```c
#include <stdio.h>

int display();

#pragma startup display
int main(){

    printf("\nI am in main function");
    return 0;
}

int display() {
    printf("\nI am in display function");
    return 0;
}
```

.c

After preprocessing

```c
# 3 ".\\macros.c"
int display();

#pragma startup display
#pragma exit display

int main(){

    printf("\nI am in main function");
    return 0;
}

int display() {
    printf("\nI am in display function");
    return 0;
}
```

.i

إذا كان المترجم يدعم توجيه `#pragma startup`، فإن خروج البرنامج سيكون كما يلي:

نسخ الكود   css

```css
I am in display function
I am in main function
```

#pragma GCC dependency

This pragma allows you to check the relative dates of the current file and another file. If the other file is more recent than the current file, a warning is issued.

```c
#include <stdio.h>

#pragma GCC dependency "depends.c"

int main(){

    printf("Hello, World!");
    return 0;
}
```

Output

The above source code is depending on depends.c. If its compilation timestamp is more recent, then the following **warning** is issued –

```
warning: current file is older than depends.c
```

#pragma GCC poison

The GCC compiler removes an identifier completely from the program. If we want to block an identifier, then we can use the **#pragma GCC poison** directive. Its syntax is as follows −

```
#pragma GCC poison identifier
```

Example

```
#include <stdio.h>

#pragma GCC poison printf

int main(){

   int a = 10;
   if (a == 10) {
      printf("Hello World");
   }
   else
      printf("TutorialsPoint");

   return 0;
}
```

```
error: attempt to use poisoned "printf"
```

#pragma GCC dependency

This pragma allows you to check the relative dates of the current file and another file. If the other file is more recent than the current file, a warning is issued.

```
#include <stdio.h>

#pragma GCC dependency "depends.c"

int main(){

   printf("Hello, World!");
   return 0;
}
```

Output

The above source code is depending on depends.c. If its compilation timestamp is more recent, then the following **warning** is issued −

```
warning: current file is older than depends.c
```

#pragma once

The **#pragma once** directive causes the header file to be included only once, even if the programmer includes it multiple times.

`example.h`:

c نسخ الكود

```c
#pragma once

void exampleFunction();
```

`example.c`:

c نسخ الكود

```c
#include "example.h"

void exampleFunction() {
    printf("Example function called.\n");
}
```

`main.c`:

c نسخ الكود

```c
#include <stdio.h>
#include "example.h"
#include "example.h"  // This inclusion will be ignored due to #pragma once

int main() {
    exampleFunction();
    return 0;
}
```

Type qualifier in C

There are two important type qualifiers in C

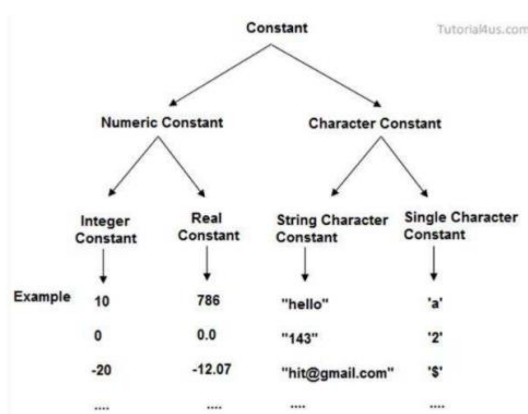
- const
- volatile

Applying these qualifiers to variable declaration is called **qualifying the declaration**. The qualifier allows the programmer to add some **features to a variable.**

C – Constants

- The const type qualifier in C is used to enforce read only features on variables
- **const** doesn't mean that the value **NEVER** changes, its only programming **safety feature** to ensure that the programmer shouldn't try to modify the value.
- All **const variables** are **stored in memory in the same way as standard variables**.
- They are placed in **RAM**. The only difference, in this sense, is the **read-only feature**.
- All **global const variables** are stored in **ROM or FLASH memory**. This also further depends on linker script rules and the hardware on which code runs.
    - The flash memory of the micro-controller is indeed **write-protected**, that means the operation won't have any effect.
    - In the **PC**, the program crashes because we're trying to write in the write-protected section.

```
1.const float pie =3.147;
2.const int radius =4;
3.const char c = 'A';
4.const char name[] = "C Course";
```



Instead of repeatedly using hard-coded values in a program, it is advised to define a constant and use it. ***Constants*** *in a C program are usually employed to refer to a value which may be error-prone if it is to be used repetitively in the program, at the same time its value is not likely to change.*

You an declare a constant in C program with either of the following two ways –

- Using the const Keyword
- Using the #define Directive

Defining a Constant Using the const Keyword

The syntax of declaring a constant is as follows –

```
const type NAME = val;
```

For example,

```
const float PI = 3.14159265359;
```

We can now use PI in any expression, as we would use any variable.

```
const float PI = 3.14159265359;
PI=22/7;
```

Here, you will get the following **error message** –

```
error: assignment of read-only variable 'PI'
```

```
const float PI;
PI = 3.14159265359;
```

Here, you will get this error message –

```
error: assignment of read-only variable 'PI'
```

Changing the Value of a Constant

The technique uses the concept of pointers in C. A Pointer is a variable that stores the address of another variable. Since it is a variable, its value can be changed. Moreover, this change reflects in the original variable.

Example: Change the Value of a Constant

```c
#include <stdio.h>
int main(){
   const int x = 10;
   printf("Initial Value of Constant: %d\n", x);

   // y is a pointer to constant x
   int* y = &x;

   // assign new value
   *y = 100;
   printf("Value of Constant after change: %d", x);
   return 0;
}
```

```
Initial Value of Constant: 10
Value of Constant after change: 100
```

EX Point out the error in the program

```
#include<stdio.h>
#define SI(p, n, r) float si; si=p*n*r/100;
int main()
{
    float p=2500, r=3.5;
    int n=3;
    SI(p, n, r);
    SI(1500, 2, 2.5);
    return 0;
}
```

(A) 26250.00 7500.00

(B) Nothing will print

(C) Error: Multiple declaration of `si` ✅

(D) Garbage values

EX

Output?

```
# include <stdio.h>
# define scanf  "%s Geeks Quiz "
int main()
{
   printf(scanf, scanf);
   return 0;
}
```

1) Compiler Error

2) %s Geeks Quiz

3) Geeks Quiz

4) %s Geeks Quiz Geeks Quiz

EX What will the SWAP macro in the following program be expanded to on preprocessing? will the code compile?

```
#include<stdio.h>
#define SWAP(a, b, c)(c t; t=a, a=b, b=t)
int main()
{
    int x=10, y=20;
    SWAP(x, y, int);
    printf("%d %d\n", x, y);
    return 0;
}
```

(A) It compiles

(B) Compiles with an warning

(C) Not compile ✅

(D) Compiles and print nothing

EX Point out the error in the program

```
#include<stdio.h>

int main()
{
    int i;
    #if A
        printf("Enter any number:");
        scanf("%d", &i);
    #elif B
        printf("The number is odd");
    return 0;
}
```

(A) Error: unexpected end of file because there is no matching `#endif` ✅

(B) The number is odd

(C) Garbage values

(D) None of above

```
#include<stdio.h>

int main()
{
    int i;
    int A =1 ;
    #if A
        printf("Enter any number:");
        scanf("%d", &i);
    #endif
        printf("The number is odd");
    return 0;
}
```

```
# 2 ".\\macros.c" 2


# 3 ".\\macros.c"
int main()
{
    int i;
    int A =1 ;




        printf("The number is odd");
    return 0;
}
```

```c
#include<stdio.h>

int main()
{
    int i;
    #define A  1
    #if A
        printf("Enter any number:");
        scanf("%d", &i);
    #endif
        printf("The number is odd");
    return 0;
}
```

```c
# 3 ".\\macros.c"
int main()
{
    int i;


        printf("Enter any number:");
        scanf("%d", &i);

        printf("The number is odd");
    return 0;
}
```

EX. What will be the output of the program?

```c
#include<stdio.h>
#define FUN(arg) do\
                 {\
                    if(arg)\
                        printf("IndiaBIX...", "\n");\
                  }while(--i)

int main()
{
    int i=2;
    FUN(i<3);
    return 0;
}
```

(A) IndiaBIX...
IndiaBIX...
IndiaBIX

(B) IndiaBIX... IndiaBIX... ✔

(C) Error: cannot use control instructions in macro

(D) No output

EX. What will be the output of the program?

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

int main()
{
    int a, b=3;
    a = CUBE(b++);
    printf("%d, %d\n", a, b);
    return 0;
}
```

(A) 9, 4

(B) 27, 4

(C) 27, 6 ✔

(D) Error

EX. What will be the output of the program?

```c
#include<stdio.h>
#define SQUARE(x) x*x

int main()
{
    float s=10, u=30, t=2, a;
    a = 2*(s-u*t)/SQUARE(t);
    printf("Result = %f", a);
    return 0;
}
```

(A) Result = -100.000000 ✔

(B) Result = -25.000000

(C) Result = 0.000000

(D) Result = 100.000000

EX. What will be the output of the program?

```c
#include<stdio.h>
#define MIN(x, y) (x<y)? x : y;
int main()
{
    int x=3, y=4, z;
    z = MIN(x+y/2, y-1);
    if(z > 0)
        printf("%d\n", z);
    return 0;
}
```

(A) 3 ✔

(B) 4

(C) 0

(D) No output

EX What will be the output of the program?

```
#include<stdio.h>
#define MAX(a, b, c) (a>b ? a>c ? a : c: b>c ? b : c)

int main()
{
    int x;
    x = MAX(3+2, 2+7, 3+7);
    printf("%d\n", x);
    return 0;
}
```

Ⓐ 5

Ⓑ 9

Ⓒ 10 ✅

Ⓓ 3+7

EX What will be the output of the program?

```
#include<stdio.h>
#define MAN(x, y) ((x)>(y)) ? (x):(y);

int main()
{
    int i=10, j=5, k=0;
    k = MAN(++i, j++);
    printf("%d, %d, %d\n", i, j, k);
    return 0;
}
```

Ⓐ 12, 6, 12 ✅

Ⓑ 11, 5, 11

Ⓒ 11, 5, Garbage

Ⓓ 12, 6, Garbage

EX

```
#include <stdio.h>
#define PRINT(i, limit) do \
                        { \
                            if (i++ < limit) \
                            { \
                                printf("GeeksQuiz\n"); \
                                continue; \
                            } \
                        }while(1)

int main()
{
    PRINT(0, 3);
    return 0;
}
```

How many times **GeeksQuiz** is printed in the above program?

- ○ 1
- ○ 2
- ○ 3
- ◉ compile time-error

```
# 2 ".\\macros.c" 2
# 10 ".\\macros.c"

# 10 ".\\macros.c"
int main() {
    do { while (0 < 3) { printf("GeeksQuiz\n"); 0 ++; } } while (0);
    return 0;
}
```

EX

What is the output of following program?

```
#include <stdio.h>
#define macro(n, a, i, m) m##a##i##n
#define MAIN macro(n, a, i, m)

int MAIN()
{
    printf("GeeksQuiz");
    return 0;
}
```

1) **Compiler Error**

2) **GeeksQuiz**

3) **MAIN**

4) **main**

- ○ 1
- ◉ 2
- ○ 3
- ○ 4

EX

```
#include <stdio.h>
#define ISEQUAL(X, Y) X == Y
int main()
{
    #if ISEQUAL(X, 0)
        printf("Geeks");
    #else
        printf("Quiz");
    #endif
    return 0;
}
```

Output of the above program?

1) **Geeks**

2) **Quiz**

3) **Any of Geeks or Quiz**

4) **Compile time error**

EX

```
#include <stdio.h>
#define X 3
#if !X
    printf("Geeks");
#else
    printf("Quiz");

#endif
int main()
{
        return 0;
}
```

1) **Geeks**

2) **Quiz**

3) **Compiler Error**

The conditional macro #if ISEQUAL(X, 0) is expanded to #if X == 0. After the pre-processing is over, all the undefined macros are initialized with default value 0. Since macro X has not been defined, it is initialized with 0. So, Geeks is printed.

" من ضيع الأصول حرم الوصول ومن ترك الدليل ضل السبيل"