**Preprocessors**

**Writer:** Sayak Haldar

**1.What is the sequence for preprocessor to look for the file within <> ?**

1. The predefined location then the current directory
2. The current directory then the predefined location
3. The predefined location only
4. The current directory location

**Answer is a) The predefined location then the current directory**

# 2.Which directory the compiler first looks for the file when using #include

1. Current directory where program is saved
2. C:COMPILERINCLUDE
3. S:SOURCEHEADERS
4. Both (b) and (c) simultaneously

**Answer is b) C:COMPILERINCLUDE**

**Explanation:** The order of searching by compilers when using #include C:COMPILERINCLUDE->S:SOURCEHEADERS-> Current directory where program is saved

# What would happen if you create a file stdio.h and use #include “stdio.h” ?

1. The predefined library file will be selected
2. The user-defined library file will be selected
3. Both the files will be included
4. The compiler won’t accept the program

## Answer is b) The user-defined library file will be selected

# How is search done in #include<somelibrary.h> and #include ”somelibrary.h” normally or conventionally?

1. When former is used, current directory is searched and when latter is used, standard directory is searched
2. When former is used, predefined directory is searched and when latter is used, current directory is searched and then predefined directories are searched
3. When former is used, search is done in implementation defined manner and latter is used to search current directory
4. For both, search for somelibrary is done in implementation-defined manner

**Answer) is b)** When former (#include<somelibrary.h>) is used, predefined directory is searched and when latter is used, current directory is searched and then predefined directories are searched.

# Can function definition be present in header files?

1. Yes
2. No
3. Depends on the compiler
4. Depends on the standard

**Answer is a) Yes**

Explanation: Though implementing functions within header files is a bad practice it can be done in c

## More Explanation:

**Why do we consider defining functions within header files as a bad practice in c?**

In C, if you define a function in a header file, then that function will appear in each module that is compiled that includes that header file, and a public symbol will be exported for the function. So if function additup is defined in header.h, and foo.c and bar.c both include header.h, then foo.o and bar.o will both include copies of additup.

When you go to link those two object files together, the linker will see that the symbol additup is defined more than once, and won't allow it.

## More Explanation:

**Difference between header files and libraries:**

Generally, a header file notifies the *compiler* of certain things (mostly their existence or declarations) so that the compiler can correctly build a single translation unit (such as a single C file).

A library file is the actual *executable code* that does the work as specified in that header file. This is linked in by the *linker* to provide the actual functionality (the \_definitions rather than just the declarations).

Header Files are the files that are included at the top of any program. If we use any function inside a program, then the header file containing declaration or definition of that function ,has to be included.Like printf() is defined in stdio.h.So, we must include it (by #include in order to use printf().

Library Files are the files which the compiler uses in order to define the functions which have been used in the program and had been declared inside the header file.Like, printf() has its complete definition ,like how it will work etc. in an I/O library! So, the compiler uses that library to get the machine code for printf.

Some of the differences between c header and library files (not user defined, but the real ones provided by c compiler):

* + Header files generally contain the declarations of the function while libraries contain the definitions of the function.
  + Header files are TEXT files while library files are BINARY. This means, we can read and modify the header file but not the library.
  + Header file is in C language while the library is in machine language!
  + Header file has to be included by the programmer while the compiler automatically relates the library file(s) with the program!

## More Explanation:

**Do's and Dont's to follow for creating a user-defined header files:**

**DO create one .h header file for each “module” of the system.** A module may comprise one or more compilation units (e.g., .c or .asm source code files). But it should implement just one aspect of the system. Examples of well-chosen modules are: a device driver for an A/D converter; a communication protocol, such as FTP; and an alarm manager that is solely responsible for logging error conditions and alerting the user of the active errors.

**DO include in the header file all of the function prototypes for the public interface of the module it describes.** For example a header file adc.h might contain function prototypes for adc\_init(), adc\_select\_input(), and adc\_read().

**DON’T include in the header file any other function or macro that may lie inside the module source code.** It is desirable to hide these internal “helper” functions inside the implementation. If it’s not called from any other module, hide it! (If your module spans several compilation units that need to share a helper function, then create a separate header file just for this purpose.) Module A should only call Module B through the public interface defined in moduleb.h.

## DON’T include any executable lines of code in a header file, including variable declarations.

But note it is necessary to make an exception for the bodies of some inline functions. (inline is a c++ feature)

**DON’T expose any variable in a header file, as is too often done by way of the ‘extern’ keyword.** Proper encapsulation of a module requires data hiding: any and all internal state data in private variables inside the .c source code files. Whenever possible these variables should also be

declared with keyword ‘static’ to enlist the linker’s help in hiding them.

**DON’T expose the internal format of any module-specific data structure passed to or returned from one or more of the module’s interface functions.** That is to say there should be no “struct { … } foo;” code in any header file. If you do have a type you need to pass in and out of your module, so client modules can create instances of it, you can simply “typedef struct foo moduleb\_type” in the header file. Client modules should never know, and this way cannot know, the internal format of the struct.

When you will learn c++, you will see this:

inline is mandatory if a function (no matter how complex or "linear") is defined in a header file, to allow multiple sources to include it without getting a **"multiple definition" error by the linker.**

# Comment on the output of this C code?

* 1. #include <stdio.h>
  2. #include "test.h"
  3. #include "test.h"
  4. int main()

5. {

1. //some code 7. }
2. true
3. Compile time error
4. false
5. Depends on the compiler

**Answer is d) It depends upon the compiler.**

Explanation: In some compiler It will give compilation error while in some other compiler it will not cause anything.

# What is the output of this C code?

* 1. #include <stdio.h>
  2. #define foo(m, n) m ## n
  3. void myfunc();
  4. int main()

5. {

1. myfunc();
2. return 0;
3. }

9. void myfunc()

10. {

11. printf("%d\n", foo(2, 3));

12. }

1. 23
2. 2 3
3. Compile time error
4. Undefined behaviour

## Answer is a) 23

**8. If the file name is enclosed in double quotation marks**

1. The preprocessor treats it as a user-defined file
2. The preprocessor treats it as a system-defined file
3. Both a & b
4. None of the mentioned

Answer) a) The preprocessor treats as a user defined file

# 9. If the file name is enclosed in angle brackets

1. The preprocessor treats it as a user-defined file
2. The preprocessor treats it as a system-defined file
3. Both a & b
4. None of the mentioned

Answer) b) The preprocessor treats as a system defined file.

# 10.What is the output of this C code?

* 1. #include <stdio.h>
  2. int main()

3. {

1. printf("hello");
2. return 0;  
    6. }
3. hello
4. Nothing
5. compile time error
6. Depends on compiler

**Answer is a) hello**

# 11.The below two lines are equivalent to

#define C\_IO\_HEADER #include C\_IO\_HEADER

1. #include
2. #include”printf”
3. #include”C\_IO\_HEADER”
4. #include

**Answer is a) #include Explanation:**

In #include C\_IO\_HEADER, C\_IO\_HEADER will be replaced by nothing

# What is the output of this C code?

* 1. #include <stdio.h>
  2. #include "printf"
  3. int main()

4. {

1. printf("hello");
2. return 0;
3. }
4. hello
5. Error
6. Depends on compiler
7. Varies

**Answer is b)** Compilation error if no file named “printf” is present in the current directory

a)hello if a file named “printf” is present in the current directory

# 13.Property which allows to produce different executable for different platforms in C is called?

1. File inclusion
2. Selective inclusion
3. Conditional compilation
4. Recursive macros

**Answer) c) Conditional Compilation**

Explanation: A useful facility provided by the he preprocessor is conditional compilation; i.e. the selection of lines of source code to be compiled and those to be ignored. Conditional compilation can be used for many purposes including its use with debug statements.

Example of conditional compilation derivatives in c: ifdef,ifndef, if, elif, else

# 14.#include

1. Preprocessor directive
2. Inclusion directive
3. File inclusion directive
4. None of the mentioned

**Answer is a) Preprocessor directive**

# 15.C preprocessors can have compiler specific features.

1. true
2. false
3. Depends on the standard
4. Depends on the platform

Explanation: pragma: this preprocessor directive is a compiler specific feature. More Explanation:

#pragma compiler specific extension

The pragma directive is used to access compiler-specific preprocessor extensions. A common use of #pragma is the #pragma once directive, which asks the compiler to include a header file only a single time, no matter how many times it has been imported:

#pragma once

// header file code

In this example, using #pragma once is equivalent to an include guard that prevents the file from being processed multiple times.

#ifndef \_FILE\_NAME\_H\_ #define \_FILE\_NAME\_H\_

/\* code \*/

#endif // #ifndef \_FILE\_NAME\_H\_

#pragma once is available on many major compilers, including Clang, GCC, the Intel C++ compiler and MSVC.

The #pragma directive can also be used for other compiler-specific purposes. #pragma is commonly used to suppress warnings. For example, in MSVC

#pragma warning (disable : 4018 )

Can be used to disable warning 4018, warning of signed/unsigned mismatch. While you should be reluctant to suppress warnings sometimes it is necessary.

For more uses of the #pragma directive, consult your compiler's documentation.

# 16.What is the output of this C code?

1. #include <stdio.h>

2. #define foo(m, n) m \* n = 10

3. int main()

4. {

5. printf("in main\n");

6. return 0;

7.}

1. In main
2. Compilation error as lvalue is required for the expression m\*n=10
3. Preprocessor error as lvalue is required for the expression m\*n=10
4. None of the mentioned

**Answer is a) In main.**

Explanation: Preprocessor just replaces whatever is given compiler then checks for error at the replaced part of the code. Here it is not replaced anywhere.

# 17.C preprocessor is conceptually the first step during compilation

1. true
2. false
3. Depends on the compiler
4. Depends on the standard

Answer) a) true.

# Preprocessor feature that supply line numbers and filenames to compiler is called?

1. Selective inclusion
2. macro substitution
3. Concatenation
4. Line control

Answer) d) Line control.

By using \_\_LINE\_\_ and \_\_FILE\_\_ we can do that.

# #include

1. Library, Library
2. Library, user-created header
3. User-created header, library
4. They can include all types of file

**Answer) d) They can include all type of file.**

# A preprocessor is a program

1. That processes its input data to produce output that is used as input to another program
2. That is nothing but a loader
3. That links various source files
4. All of the mentioned

**Answer is a)** : A preprocessor is a preprocessor is a program that processes its input data to produce output that is used as input to another program.

# Which of the following are C preprocessors?

1. #ifdef
2. #define
3. #endif
4. All of the mentioned

**Answer) d) All of the mentioned.**

1. **#include<stdio.h> statement must be written**
2. Before main()
3. Before any scanf/printf
4. After main()
5. It can be written anywhere

**Answer is b) It could be mentioned before any printf or scanf**

Explanation: We generally write it before main to improve the readability of the program

# 23.#pragma exit is primarily used for?

1. Checking memory leaks after exiting the program
2. Informing Operating System that program has terminated
3. Running a function at exiting the program
4. No such preprocessor exist

**Answer is c) Running a function at exiting the program**

# 24.What is the output of this C code?

1. #include <stdio.h>
2. int main()

3. {

1. int one = 1, two = 2;
2. #ifdef next
3. one = 2;
4. two = 1;
5. #endif
6. printf("%d, %d", one, two);
7. return 0;

11. } a) 1, 1

b) 1, 2

c) 2, 1

d) 2, 2

if next is defined, values will be changed. Now, it is not defined. Hence, line 6 and 7 will not be compiled and executed. Hence, b) 1,2

# 25. The C-preprocessors are specified with symbol.

1. #
2. $
3. ” ”
4. None of the mentioned

Answer) is a) #

# The #include directive

1. Tells the preprocessor to grab the text of a file and place it directly into the current file
2. are statements are typically placed at the top of a program
3. both a & b
4. None of a & b

**Answer) is c) both a & b.**

# 27.The preprocessor provides the ability for .

1. The inclusion of header files
2. The inclusion of macro expansions
3. Conditional compilation and line control.
4. All of the mentioned

**Answer) is d) All of the mentioned.**

# 28.#include is used with file name in angular brackets

1. The file is searched for in the standard compiler include paths
2. The search path is expanded to include the current source directory
3. Both a & b
4. None of the mentioned

**Answer is a)**

The file is searched for in the standard compiler include paths

Explanation: #include might check the current directory but #include<filename.h> will never check the existence of the filename.h in the current directory.

1. **What is the output of this C code?**
2. #include <stdio.h>
3. #define foo(m, n) m ## n
4. int main()
5. {
6. printf("%s\n", foo(k, l));
7. return 0;
8. }
9. k l
10. kl
11. Compile time error
12. Undefined behaviour

**Answer) c) Compile time error.**

# 30.What is the output of this C code?

1. #include <stdio.h>
2. #define foo(m, n) " m ## n "
3. int main()
4. {
5. printf("%s\n", foo(k, l));
6. }
7. k l
8. kl
9. Compile time error
10. m ## n

**Answer is d) m##n**

Since, foo(k,l) will simply replaced by “m##n” after preprocesssing in the c code. So, m##n will be printed as a string

1. **What is the output of this C code?**
2. #include <stdio.h>
3. #define foo(x, y) #x #y
4. int main()

4. {

1. printf("%s\n", foo(k, l));
2. return 0;
3. }
4. kl
5. k l
6. xy
7. Compile time error

**Answer) a) kl**

# What is the output of this C code?

# #include <stdio.h>

# #define foo(x, y) x / y + x

# int main()

1. {
2. int i = -6, j = 3;
3. printf("%d\n",foo(i + j, 3));
4. return 0;
5. }
6. Divided by zero exception
7. Compile time error
8. -8
9. -4

**Answer is c) -8**

**Explanation:** foo(i + j, 3) at line 6, will be replaced by x+y/3+x+y in the program with x=-6 and y=3

Now, / operator has highest precedence among the present operators in the expression So, -6+3/3-6+3 will be evaluated as the following:

=-6+1-6+3

Now, we get an expression consists of only – and + operator. Both operator have same precedence. So, associativity will effect the evaluation of the expression. Now, associativity of + and – is from left to right. So,

=-5-6+3

=-11+3

=-8

# What is the output of this C code?

1. #include <stdio.h>
2. void f();
3. int main()

4. {

5. #define foo(x, y) x / y + x 6. f();

7. }

8. void f()

9. {

10. printf("%d\n", foo(-3, 3));

11. }

1. -8
2. -4
3. Compile time error
4. Undefined behaviour

**Answer) b) -4**

# What is the output of this C code?

1. #include <stdio.h>
2. void f();
3. int main()
4. {
5. #define max 10
6. f();
7. return 0;
8. }
9. void f()
10. {
11. printf("%d\n", max \* 10);

12. } a) 100

1. Compile time error since #define cannot be inside functions
2. Compile time error since max is not visible in f()
3. Undefined behaviour

**Answer is a) 100**

# What is the output of this C code?

1. #include <stdio.h>
2. #define foo(x, y) x / y + x
3. int main()
4. {

5. int i = -6, j = 3;

6. printf("%d ", foo(i + j, 3)); 7. printf("%d\n", foo(-3, 3));

8. return 0; 9. }

a) -8 -4

b) -4 divided by zero exception

c) -4 -4

d) Divided by zero exception

Answer) a)

# What is the output of this C code?

1. #include <stdio.h>
2. int foo(int, int);
3. #define foo(x, y) x / y + x
4. int main()
5. {
6. int i = -6, j = 3;
7. printf("%d ",foo(i + j, 3));
8. #undef foo
9. printf("%d\n",foo(i + j, 3));

10. }

11. int foo(int x, int y)

12. {

13. return x / y + x;

14. }

a) -8 -4

b)Compile time error

c) -8 -8

d) Undefined behaviour  
  
Answer) a) This is an ideal example of difference between macro substitution and function call. Also, an ideal difference between macro substitution and inline function.

# 36. What is the advantage of #define over const?

1. Data type is flexible
2. Can have a pointer
3. Reduction in the size of the program
4. Both (a) and (c)

**Answer is a) Data type is flexible**

# 37.What is the output of this C code?

1. #include <stdio.h>
2. int main()

3. {

1. #define max 37;
2. printf("%d", max);
3. return 0;
4. }
5. 37
6. Compile time error
7. Varies
8. Depends on compiler

**Answer is b) Compile time error due to ';' at the end of line 4**

# What is the output of this C code?

1. #include <stdio.h>
2. void main()

3. {

1. #define max 37
2. printf("%d", max);
3. }
4. 37
5. Run time error
6. Varies
7. Depends on compiler

Answer) a) 37.

# What is the output of this C code?

1. #include <stdio.h>
2. void main()

3. {

1. #define const int
2. const max = 32;
3. printf("%d", max);
4. }
5. Run time error
6. 32
7. int
8. const

**Answer is b) 32**

**Explanation:** Though const is a keyword in both c and c++, it will work fine

1. **What is the output of this C code?**
2. #include <stdio.h>
3. void main()

3. {

4. #define max 45

5. max = 32;

6. printf("%d", max); 7. }

1. 32
2. 45
3. Compile time error
4. Varies

**Answer) c) Compile time error.   
  
Explanation:** Compile time error since max is used as an identifier at line 5, but max is undeclared Compile time error due to absence of lvalue at the left side of assignment operator at line 5.

# What is the output of this C code?

1. #include <stdio.h>
2. # define max
3. void m()

4. {

5. printf("hi"); 6. }

7. void main()

1. {
2. max;
3. m();
4. }
5. Run time error
6. hi hi
7. Nothing
8. Hi

Answer is d) hi

# 41.What is the output of this C code?

1. #include <stdio.h>
2. #define A 1 + 2
3. #define B 3 + 4
4. int main()

5. {

1. int var = A \* B;
2. printf("%d\n", var);
3. }
4. 9
5. 11
6. 12
7. 21

**Answer is b) 11**

Explanation: A\*B at line at line 6 will be replaced by 1+2\*3+4=11  
  
Note: this is another example, in which a function call and macro substitution will behave differently.

# 42.Which of the following Macro substitution are accepted in C?

1. #define A #define A VAR 20
2. #define A define #A VAR 20
3. #define #A #define #A VAR 20
4. None of the mentioned

**Answer) d) None of the mentioned.**

# 43.Comment on the following code?

1. #include <stdio.h>
2. #define var 20);
3. int main()

4. {

1. printf("%d\n", var
2. return 0; 7. }
3. No errors, it will show the output 20
4. Compile time error, the printf braces aren’t closed
5. Compile time error, there are no open braces in #define
6. Both (b) and (c).

**The answer is a) No error, it will show the output 20.**

Explanation: Since printf("%d\n", var will simply become printf("%d\n",20); after preprocessing

# Which of the following properties of #define not true?

1. You can use a pointer to #define
2. #define can be made externally available
3. They obey scope rules
4. All of the mentioned

Answer) d) All of the mentioned. Neither of them are true.

# 45.What is the output of this C code?

1. #include <stdio.h>

2. #define SYSTEM 20

3. int main()

4. {

1. int a = 20;
2. #if SYSTEM == a
3. printf("HELLO ");
4. #endif
5. #if SYSTEM == 20
6. printf("WORLD\n");
7. #endif

12. }

a) HELLO

b) WORLD

c) HELLO WORLD

d) No Output

46)The answer is b) World

Explanation: a needs to be a macro to execute line 7

# 46. Comment on the following code?

1. #include <stdio.h>
2. #define Cprog
3. int main()

4. {

1. int a = 2;
2. #ifdef Cprog

7. a = 1;

8. printf("%d", Cprog); 9. }

1. No output on execution
2. Output as 1
3. Output as 2
4. Compile time error

**Answer is d) Compile time error**

**Explanation:** Compiler would say the following things: a)Unterminated #ifdef

b)expected expression before ‘)’ token

printf("%d", Cprog); Since Cprog would expand to nothing thus causing the error because %d would expect one integer type argument at printf.

Note: #ifdef must be terminated by #endif

# The “else if” in conditional inclusion is written by?

1. #else if
2. #elseif
3. #elsif
4. #elif

**Answer is d) elif**

# What is the output of this C code?

1. #include <stdio.h>
2. #define COLD
3. int main()
4. {
5. #ifdef COLD
6. printf("COLD\t");
7. #undef COLD
8. #endif
9. #ifdef COLD
10. printf("HOT\t");
11. #endif
12. }
13. HOT
14. COLD
15. COLD HOT
16. No Output

**Answer is b) COLD.**

# Which of the following sequences are unaccepted in C language?

1. #if #else

#endif

1. #if #elif

#endif

1. #if #if

#endif

1. #if

#undef  
#endif

**Answer) c)**

# In a conditional inclusion, if the condition that comes after the if holds.

1. Then the code up to the following #else or #elif or #endif is compiled
2. Then the code up to the following #endif is compiled even if #else or #elif is present
3. Both a & b
4. None of the mentioned

**Answer is a)**

# Conditional inclusion can be used for

1. Preventing multiple declarations of a variable
2. Check for existence of a variable and doing something if it exists
3. Preventing multiple declarations of same function
4. All of the mentioned

Answer)d) All of the mentioned.

# The #elif directive cannot appear after the preprocessor #else directive.

1. true
2. false
3. None of the mentioned
4. Varies

**Answer) a)**

# For each #if, #ifdef, and #ifndef directive.

1. There are zero or more #elif directives
2. Zero or one #else directive
3. One matching #endif directive
4. All of the mentioned

Answer) d) all of the mentioned.

1. **The #else directive is used for**
2. Conditionally include source text if the previous #if, #ifdef, #ifndef, or #elif test fails.
3. Conditionally include source text if a macro name is not defined
4. Conditionally include source text if a macro name is defined
5. Ending conditional text

Answer is a) Conditionally include source text if the previous #if, #ifdef, #ifndef, or #elif test fails.

# What is the output of this C code?

1. #include <stdio.h>
2. #define MIN 0
3. #if MIN
4. #define MAX 10
5. #endif
   1. int main()

7. {

1. printf("%d %d\n", MAX, MIN);
2. return 0;

10. } a) 10 0

1. Compile time error
2. Undefined behaviour
3. None of the mentioned

# What is the output of this C code?

1. #include <stdio.h>
2. #define MIN 0
3. #ifdef MIN
4. #define MAX 10
5. #endif
6. int main()
7. {
8. printf("%d %d\n", MAX, MIN);
9. return 0;

10. } a) 10 0

1. Compile time error
2. Undefined behaviour
3. None of the mentioned

**Answer is b) Compile time error**

**Explanation:** #if min Now, min is 0 so, the next statement which defines MAX macro would never be executed. Thus, MAX would be undeclared during its use in main.

**Note:** This is the basic difference between #ifdef and #if

If in line 3, #if MIN is replaced by #ifdef MIN then the program will not cause compilation error. It will print 10 0

# What is the output of this C code?

1. #include <stdio.h>
2. #define MIN 0
3. #if defined(MIN) + defined(MAX)
4. #define MAX 10
5. #endif
6. int main()

7. {

1. printf("%d %d\n", MAX, MIN);
2. return 0;
3. }

a) 10 0

1. Compile time error
2. Undefined behaviour
3. Somegarbagevalue 0

Answer is a) 10 0

# What is the output of this C code?

# #include <stdio.h>

# #define MIN 0

# #if defined(MIN) - (!defined(MAX))

# #define MAX 10

# #endif

# int main()

1. {
2. printf("%d %d\n", MAX, MIN);
3. return 0;
4. }

a) 10 0

1. Compile time error
2. Undefined behaviour
3. Somegarbagevalue 0

**Answer)**

#if defined(MIN) + defined(MAX) at line 3

Now, defined will check if a macro is defined or not and returns 1 or 0 accordingly/respectively Now, MIN macro is already defined and MAX macro is not defined

So, defined(MIN)+defined(MAX)=1+0 #if 1

So, line 4 will execute/will get a chance to be executed

Hence, 10 0.

Answer) a) 10 0

# What is the output of code given below?

# #include <stdio.h>

# #define MIN 0);

# #ifdef MIN

# #define MAX 10

# #endif

# int main()

* 1. {

1. printf("%d %d\n", MAX, MIN
2. return 0;

10. } a) 10 0

1. Compile time error due to illegal syntax for printf
2. Undefined behaviour
3. Compile time error due to illegal MIN value

**Answer) a) 10 0.**

**Extras:**

**Some necessary application of preprocessor:**

#define FATAL 1

#define ERROR 2

#define INFO 3

#define LOGLEVEL INFO

#define check\_log\_level(level,level\_to\_be\_checked) (level>=level\_to\_be\_checked?1:0)

//this is log level

#if check\_log\_level(LOGLEVEL,DEBUG)

#define debug(LOG,...); fprintf(httpcallback\_log,"%s-%s [INFO] %s %d "LOG,\_\_DATE\_\_,\_\_TIME\_\_,\_\_FILE\_\_,\_\_LINE\_\_,##\_\_VA\_ARGS\_\_);fprintf(stdout,"%s-%s [INFO] %s %d "LOG,\_\_DATE\_\_,\_\_TIME\_\_,\_\_FILE\_\_,\_\_LINE\_\_,##\_\_VA\_ARGS\_\_);

#else

#define debug(LOG,...);

#endif

#if check\_log\_level(LOGLEVEL,INFO)

#define info(LOG,...); fprintf(httpcallback\_log,"%s-%s [INFO] %s %d "LOG,\_\_DATE\_\_,\_\_TIME\_\_,\_\_FILE\_\_,\_\_LINE\_\_,##\_\_VA\_ARGS\_\_);fprintf(stdout,"%s-%s [INFO] %s %d "LOG,\_\_DATE\_\_,\_\_TIME\_\_,\_\_FILE\_\_,\_\_LINE\_\_,##\_\_VA\_ARGS\_\_);

#else

#define info(LOG,...);

#endif

#if check\_log\_level(LOGLEVEL,ERROR)

#define error(LOG,...); fprintf(httpcallback\_log,"%s-%s [ERROR] %s %d "LOG,\_\_DATE\_\_,\_\_TIME\_\_,\_\_FILE\_\_,\_\_LINE\_\_,##\_\_VA\_ARGS\_\_);fprintf(stderr,"%s-%s [ERROR] %s %d "LOG,\_\_DATE\_\_,\_\_TIME\_\_,\_\_FILE\_\_,\_\_LINE\_\_,##\_\_VA\_ARGS\_\_);

#else

#define error(LOG,...);

#endif

#if check\_log\_level(LOGLEVEL,FATAL)

#define fatal(LOG,...); fprintf(httpcallback\_log,"%s-%s [FATAL] %s %d "LOG,\_\_DATE\_\_,\_\_TIME\_\_,\_\_FILE\_\_,\_\_LINE\_\_,##\_\_VA\_ARGS\_\_);fprintf(stderr,"%s-%s [FATAL] %s %d "LOG,\_\_DATE\_\_,\_\_TIME\_\_,\_\_FILE\_\_,\_\_LINE\_\_,##\_\_VA\_ARGS\_\_);

**\_\_DATE\_\_ :** predefined macro This macro expands to a string constant that describes the date on which the preprocessor is being run. The string constant contains eleven characters and looks like "Feb 12 1996". If the day of the month is less than 10, it is padded with a space on the left. (MM DD YYYY)

**\_\_TIME\_\_: hh:mm:ss**

**\_\_FILE\_\_: (filename with filepath)**

**\_\_LINE\_\_: (line number)**

**##\_\_VA\_ARGS\_\_**

## will make the last comma ignored in case there is no argument

Without it, there will be a compilation error for the cases when there is no variable arguments.

**Once Only Header:**

If a header file happens to be included twice, the compiler will process its contents twice. This is very likely to cause an error, e.g. when the compiler sees the same structure definition twice. Even if it does not, it will certainly waste time.

The standard way to prevent this is to enclose the entire real contents of the file in a conditional, like this:

/\* File foo. \*/

#ifndef FILE\_FOO\_SEEN

#define FILE\_FOO\_SEEN

the entire file

#endif /\* !FILE\_FOO\_SEEN \*/

This construct is commonly known as a wrapper #ifndef. When the header is included again, the conditional will be false, because FILE\_FOO\_SEEN is defined. The preprocessor will skip over the entire contents of the file, and the compiler will not see it twice.

GNU CPP optimizes even further. It remembers when a header file has a wrapper `#ifndef'. If a subsequent `#include' specifies that header, and the macro in the `#ifndef' is still defined, it does not bother to rescan the file at all.

You can put comments outside the wrapper. They will not interfere with this optimization.

The macro FILE\_FOO\_SEEN is called the controlling macro or guard macro. In a user header file, the macro name should not begin with `\_'. In a system header file, it should begin with `\_\_' to avoid conflicts with user programs. In any kind of header file, the macro name should contain the name of the file and some additional text, to avoid conflicts with other header files.

**Computed Includes:**

Sometimes it is necessary to select one of several different header files to be included into your program. They might specify configuration parameters to be used on different sorts of operating systems, for instance. You could do this with a series of conditionals,

#if SYSTEM\_1

# include "system\_1.h"

#elif SYSTEM\_2

# include "system\_2.h"

#elif SYSTEM\_3

...

#endif

That rapidly becomes tedious. (too long or dull)  
  
Instead, the preprocessor offers the ability to use a macro for the header name. This is called a computed include. Instead of writing a header name as the direct argument of `#include', you simply put a macro name there instead:

#define SYSTEM\_H "system\_1.h"

...

#include SYSTEM\_H

SYSTEM\_H will be expanded, and the preprocessor will look for `system\_1.h' as if the `#include' had been written that way originally. SYSTEM\_H could be defined by your Makefile with a `-D' option.

gcc -DSYSTEM\_H='"system\_1.h"' ...

Like this.

**(-D is for using compile time macro)**

**Wrapper Headers**

Sometimes it is necessary to adjust the contents of a system-provided header file without editing it directly. GCC's fixincludes operation does this, for example. One way to do that would be to create a new header file with the same name and insert it in the search path before the original header. That works fine as long as you're willing to replace the old header entirely. But what if you want to refer to the old header from the new one?

You cannot simply include the old header with `#include'. That will start from the beginning, and find your new header again. If your header is not protected from multiple inclusion (see section 2.4 Once-Only Headers), it will recur infinitely and cause a fatal error.

You could include the old header with an absolute pathname:

#include "/usr/include/old-header.h"

This works, but is not clean; should the system headers ever move, you would have to edit the new headers to match.

There is no way to solve this problem within the C standard, but you can use the GNU extension `#include\_next'. It means, "Include the next file with this name." This directive works like `#include' except in searching for the specified file: it starts searching the list of header file directories after the directory in which the current file was found.

Suppose you specify `-I /usr/local/include', and the list of directories to search also includes `/usr/include'; and suppose both directories contain `signal.h'. Ordinary #include <signal.h> finds the file under `/usr/local/include'. If that file contains #include\_next <signal.h>, it starts searching after that directory, and finds the file in `/usr/include'.

`#include\_next' does not distinguish between <file> and "file" inclusion, nor does it check that the file you specify has the same name as the current file. It simply looks for the file named, starting with the directory in the search path after the one where the current file was found.

The use of `#include\_next' can lead to great confusion. We recommend it be used only when there is no other alternative. In particular, it should not be used in the headers belonging to a specific program; it should be used only to make global corrections along the lines of fixincludes.

**In simple language,**

t is used if you want to replace a default header with one of your own making, for example, let's say you want to replace "stdlib.h". You would create a file called stdlib.h in your project, and that would be included instead of the default header.

#include\_next is used if you want to add some stuff to stdlib.h rather than replace it entirely. You create a new file called stdlib.h containing:

#include\_next "stdlib.h"

int mystdlibfunc();

And the compiler will not include your stdlib.h again recursively, as would be the case with plain a #include, but rather continue in other directories for a file named "stdlib.h".