# 'C' OR 'C++': MAKING OWN HEADER FILES

Munani Anilkumar L.[1], Shah Pratik A.[2]
[1]Department of Computer Engineering,
BBIT, Vallabh Vidyanagar,
Gujarat, India

**Abstract: A header file is a file which contains C function declarations and macro definition. In this paper we intend to discuss header files, purpose, how to create our own header file. First we will introduce the header files in short and later we will draw focus on introduction, creation and syntax, and uses of functions that declared in specific header file.**

***Keywords:*** *– C Header files and its creation*

## I. INTRODUCTION

A header file is a file with extension **.h** which contain C function declarations and macro definitions and to be shared source files. You request the use of a header file in your program by including it, with the C preprocessing directive **#include** like you have seen inclusion of **stdio.h** header file, which comes along with your compiler. Including a header file is equal to copying the content of the header file but we do not do it because it will be very much error-prone and it is not good idea to copy the content of header file in the source files, especially if we have multiple source file comprising our program. In simple practice in C or C++ programs is that we keep all the constants, macros, system wide global variables and function prototypes in header files and include that file wherever it is required.

## II. PURPOSES OF HEADER FILES

There are mainly two purposes of header files.

- System header files declare the interfaces to parts of the operating system. You include them in your program to supply the definitions and declarations you need to invoke system calls and libraries.
- Your own header files contain declarations for interfaces between the source files of your program. Each time you have a group of related declarations and macro definitions all or most of which are needed in several different source files, it is good idea to create a header file for them.

Including a header file produces the same results as copying the header file into each source file that need it. Such copying would be time-consuming and error-prone. With a header file the related declarations appear in only one place. If they need to be changed, they can be changed in one place, and programs include the header file will automatically use the new version when next recompiled. The header file eliminates the labour of finding and changing all the copies as well as the risk that failure to find one copy will result in inconsistencies within a program.

## III. MAKE YOUR OWN HEADER FILES

A well-organized C program has good choice of module, and properly constructed header files that make easy to understand and access the functionality in module.
The following rules summarizes how to setup your header and source files for the greatest clarity and compilation convenience.

**Rule 1: Each module with its .h and .c file should correspond to a clear piece of functionality.**
Conceptually, a module is group of declarations and functions can be developed and maintained separately from other modules and perhaps even reused in entirely different projects. Don't force together into a module things that will be used or maintained separately, and don't separate things that will always be used and maintained together. The Standard Library modules math.h and string.h are good examples of clearly distinct modules.

**Rule 2: Always use "include guards" in a header file.**
The most compact form uses #ifndef. Choose a guard symbol based on the header file name, since these symbols are easy to think up and the header file names are almost always unique in a project.
Follow the convention of making the symbol all-caps. For example "Arithmatic_op.h" would start with:

```
#ifndef ARITHMATIC_OP_H
#define ARITHMATIC_OP_H
```

and end with:

```
#endif
```

*Note: Do not start the guard symbol with an underscore* leading underscore names are reserved for internal use by the C implementation – the preprocessor, compiler, and Standard Library – breaking this rule can cause unnecessary and very puzzling errors. The complete rule for leading underscores is rather complex; but if you follow this simple form you'll stay out of trouble.

**Rule 3: All of the declarations needed to use a module must appear in its header file, and this file is always used to access the module.**
Thus including the header file provides all the information

necessary for code using the module to compile and link correctly. Furthermore, if module P needs to use module L's functionality, it should always include "L.h", and never contain hard-coded declarations for structure or functions that appear in module L. Why? If module L is changed, but you forget to change the hard-coded declarations in module P, module P could easily fail with subtle run-time errors that won't be detected by either the compiler or linker. This is a violation of the "One Definition Rule" which C compilers and linkers can't detect. Always referring to a module through its header file ensures that only a single set of declarations needs to be maintained, and helps enforce the One-Definition Rule.

**Rule 4: The header file contains only declarations, and is included by the .c file for the module.**
Put only structure type declarations, function prototypes, and global variable extern declarations, in the .h file; put the function definitions and global variable definitions and initializations in the .c file. The .c file for a module must include the .h file; the compiler can detect discrepancies between the two, and thus help ensure consistency.

**Rule 5: Set up program-wide global variables with an *extern declaration* in the header file, and a *defining declaration* in the .c file.**
For global variables that will be known throughout the program, place an extern declaration in the .h file.

**i.e. extern int product_number;**

The other modules include only the .h file. The .c file for the module must include this same .h file, and near the beginning of the file, a defining declaration should appear - this declaration both defines and initializes the global variables.

**i.e. product_number=0;**

Of course, some other value besides zero could be used as the initial value, and static/global variables are initialized to zero by default, but initializing explicitly to zero is customary because it marks this declaration as the *defining declaration*, meaning that this is the unique point of definition. Note that different C compilers and linkers will allow other ways of setting up global variables, but this is the accepted C++ method for defining global variables and it works for C as well to ensure that the global variables obey the One Definition Rule.

**Rule 6: Keep a module's internal declarations out of the header file.**
Sometimes a module uses strictly internal components that are not supposed to be accessed by other modules. If you need structure declarations, global variables, or functions that are used only in the code in the .c file, put their definitions or declarations near the top of the .c file and *do not mention them in the .h file*. Furthermore, declare global and functions static in the .c file to give them internal linkage. This way, other modules do not and cannot know about these declarations, global, or functions that are internal to the module. The internal linkage resulting from the static declaration will enable the linker to help you enforce your design decision.

**Rule 7: Every header file L.h should include every other header file that L.h requires to compile correctly.**
If another structure type P is used as a member variable of a structure type L, then you must include L.h in P.h so that the compiler knows how large the L member is. Do not include header files that only the .c file code needs. E.g.<math.h> is usually needed only by the function definitions include it in .c file, not in the .h file.

**Rule 8: If an incomplete declaration of a structure type L will do, use it instead of including its header L.h.**
If a struct type L appears only as a pointer type in a structure declaration or its functions, and the code in the header file does not attempt to access any member variables of L, then you should not include L.h, but instead make an *incomplete declaration* of L (also called a "forward" declaration) before the first use of L. Here is an example in which a structure type Thing refers to L by a pointer:

```
struct L; /* incomplete ("forward") declaration */
struct Thing
{
        int i;
        struct L* m_ptr;
};
```

The compiler will be happy to accept code containing pointers to an incompletely known structure type, basically because pointers always have the same size and characteristics regardless of what they are pointing to. Typically, only the code in the .c file needs to access the members (or size) of L, so the .c file will include "L.h". This is a powerful technique for encapsulating a module and decoupling it from other modules.

**Rule 9: The content of a header file should compile correctly by itself.**
A header file should explicitly include or forward declare everything it needs. Failure to observe this rule can result in very puzzling errors when other header files or includes in other files are changed. Check your headers by compiling (by itself) a file1.c that contains nothing more than include "L.h". It should not produce any compilation errors. If it does, then something has been left out something else needs to be included or forward declared. Test all the headers in a project by starting at the bottom of the include hierarchy and work your way to the top. This will help to find and eliminate any accidental dependencies between header files.

**Rule 10: The P.c file should first include its P.h file, and then any other headers required for its code.**
Always include P.h first to avoid hiding anything it is missing that gets included by other .h files. Then, if P's implementation code uses L, explicitly include L.h in P.c, so that P.c is not dependent on L.h accidentally being included somewhere else.

There is no clear consensus on whether P.c should also include header files that P.*h has already included*. Two suggestions:

- If the L.h file is a logically unavoidable requirement for the declaration in P.h to compile, then including it in P.c is redundant, since it is guaranteed to be included by P.h. So it is OK to *not* #include L.h in P.c.
- Always including L.h in P.c is a way of making it clear to the reader that we are using L, and helps make sure that L's declarations are available even if the contents of P.h changes due to the design changes. E.g. maybe we had a struct Thing member of a struct at first, then got rid of it, but still used Things in the implementation code.

**Rule 11: Never include P.c file for any reason.**
Sometimes you need to bring in a bunch of code that really should to be shared between .c files for ease of maintenance, so you put it in a file by itself. Because the code does not consist of "normal" declarations or definitions, you know that putting it in a .h file is misleading, so you are tempted to call it a ".c" file instead, and then write include "program1.c". But this causes instant confusion for other programmers and interferes with convenience in using IDEs, because .c files are normally separately compiled, so you have to somehow tell people not to compile this one .c file out of all the others. Furthermore, if they miss this hard-to-document point, they get really confused because compiling this sort of odd file typically produces a million error messages, making people think something mysterious is fundamentally wrong with your code or how they installed it.

**Conclusion:** If it can't be treated like a normal header or source file, don't name it like one. If you think you need to do something like this, first make sure that there isn't a more normal way to share the code (such as simply creating another module). If not, then name the special include file with a different extension like ".inc" or ".inl".

## IV. SYNTAX AND EXAMPLE

### a. With using "include guards"

Now let's talk about how to make your own header files in C and C++.
Let i have to make header file "anil.h"
STEP 1: Open turbo C editor and create new file and write following syntax in it.
**Syntax:**

```
#indef<space>  HEADER_FILE_NAME_H
#define<space>  HEADER_FILE_NAME_H

// define your functions directly
ReturnType FucntionName(Aruguments)
{
        //Implementation code here
}
#endif
```

**Example:**

```
 #indef ANIL_H
 #define ANIL_H
//now let write function definition to find factorial of a given number.
 int fact(int num)
{
        int i,fact=1;
        for (i=num;i>=1;i--)
        {
                f=f*i;
        }
        return (f);
}
```

STEP 2: Save this file as ANIL.H in INCLUDE folder of TC.

STEP 3: Now create new file and use that header file which you have created.

**Example:**

```
#include<anil.h>
#include<stdio.h>
#include<conio.h>
void main( )
{
  int ans;
  clrscr( );
  ans=fact(5);
  printf("Your factorial is:%d", ans);
  getch( );
}
```

### b. Without using "include guards"

If you want to create header file without using "include guards" then you can create, but when u create your source file then you have to write include statement as following.
i.e. #include "header file" //use double quate for header file.
If you want to print something then you can also include stdio.h header file while you creating your own header file.
Let see one example of it.
Suppose i want to create a header file which supports one function to calculate restaurant bill.

```
#include<stdio.h>
void totalBill(int food_cost, int tax, int tip)
{
int result;
result = food_cost + tax + tip;
printf("Total bill is %d \n", result);
}
```

Save this file as restaurant.h and include that file in your source file.

```
#include<stdio.h>
#include"restaurant.h" //own header file
void main()
{
int food_cost, tax, tip, bill;
food_cost = 100;
tax = 15;
tip = 10;
totalBill(food_cost,tax,tip); // function created in restaurant.h
getch( );
}
```

If we execute this program totalBill function directly print bill amount on output screen.

## V. INCLUDE DIRECTIVE

Both user and system header files are included using the preprocessing directive 'include'. It has two variant.

**#include <file>**
This variant is used for system header files. It searches for a file in a standard list of system directories.
i.e. C:\TC\INCLUDE

**#include "file"**
This variant is used for header files of your own program. It searches for a file named file first in the directory containing the current file, then in the quote directories and then the same directories used for <file>.

The argument of **'#include'** whether delimited with quote marks or angle brackets, behave like a string constant in that comments are not recognized, and macros names are not expanded.

## VI. CONCLUSION

Well, the main reason would be for separating the interface from the implementation. The header declares "what" a class (or whatever is being implemented) will do, while the cpp file defines "how" it will perform those features.
This reduces dependencies so that code that uses the header doesn't necessarily need to know all the details of the implementation and any other classes/headers needed only for that. This will reduce compilation times and also the amount of recompilation needed when something in the implementation changes.

## REFERENCES:

[1] http://stackoverflow.com/questions/7109964/creating-your-own-header-file-in-c

[2] http://www.programmingspark.com/2012/12/create-your-own-header-file-in-c.html

[3] http://c-programming-language tut.blogspot.in/2011/12/how-to-create-header-file-in-c-language.html>

[4] http://www.tutorialspoint.com/cprogramming/c_header_files.html

[5] http://gcc.gnu.org/onlinedocs/cpp/Header-Files.html>