**C Coding Guidelines**

.

**Revision History**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

[1. File Organization 5](#_Toc211747926)

[1.1 File Layout 5](#_Toc211747927)

[1.2 Header File Layout 7](#_Toc211747928)

[1.3 Source File Layout 8](#_Toc211747929)

[1.4 Declarations 9](#_Toc211747930)

[1.4.1 Global Variables 9](#_Toc211747931)

[1.4.2 Variable Declaration 9](#_Toc211747932)

[1.4.3 Enumerations 10](#_Toc211747933)

[1.4.4 Structures 11](#_Toc211747934)

[1.4.5 Unions 13](#_Toc211747935)

[1.5 Function Declarations 14](#_Toc211747936)

[1.6 Initialization 15](#_Toc211747937)

[2. Naming Conventions 16](#_Toc211747938)

[2.1 File Names 16](#_Toc211747939)

[2.2 Identifier Names 17](#_Toc211747940)

[2.2.1 General 17](#_Toc211747941)

[2.2.2 Identifier Character Set Rules 18](#_Toc211747942)

[2.2.3 Decorators 19](#_Toc211747943)

[2.2.4 Types 19](#_Toc211747944)

[2.3 Function Names 19](#_Toc211747945)

[2.4 Variable Types and Prefixes 19](#_Toc211747946)

[2.5 Pointer and Array Variables 20](#_Toc211747947)

[2.6 Variable Names 21](#_Toc211747948)

[3. Documentation 22](#_Toc211747949)

[3.1 Comment Types and Usage 22](#_Toc211747950)

[3.1.1 General comments 22](#_Toc211747951)

[3.2 Function Level Documentation 24](#_Toc211747952)

[3.3 Special Comments 27](#_Toc211747953)

[3.4 Commenting out code 30](#_Toc211747954)

[4. Formatting Conventions 30](#_Toc211747955)

[4.1 Expression Formatting 30](#_Toc211747956)

[4.1.1 Spacing 30](#_Toc211747957)

[4.1.2 Parenthesis 31](#_Toc211747958)

[4.1.3 Line Breaking 32](#_Toc211747959)

[4.1.4 Pointer Variables 33](#_Toc211747960)

[4.2 Statement Formatting 33](#_Toc211747961)

[4.2.1 Labels and goto 33](#_Toc211747962)

[4.2.2 Function 34](#_Toc211747963)

[4.2.3 Line Breaking 35](#_Toc211747964)

[4.2.4 For 35](#_Toc211747965)

[4.2.5 If : 36](#_Toc211747966)

[4.2.6 While 37](#_Toc211747967)

[4.2.7 Switch 38](#_Toc211747968)

[4.2.8 Return/ Sizeof statements 39](#_Toc211747969)

[5. Preprocessor 40](#_Toc211747970)

[5.1 General 40](#_Toc211747971)

[5.2 # define 41](#_Toc211747972)

[5.3 #if / #elif / #else / #endif 42](#_Toc211747973)

[6. Programming/ Coding Standards 44](#_Toc211747974)

[6.1 Pointers 44](#_Toc211747975)

[6.2 Functions & Methods 44](#_Toc211747976)

[6.2.1 Metrics 44](#_Toc211747977)

[6.2.2 Design Guidelines 44](#_Toc211747978)

[6.3 Statements & Expressions 45](#_Toc211747979)

[6.3.1 If 45](#_Toc211747980)

[6.3.2 For 46](#_Toc211747981)

[6.3.3 Switch 47](#_Toc211747982)

[6.3.4 While: 48](#_Toc211747983)

[6.3.5 Others: 49](#_Toc211747984)

[6.4 Security 50](#_Toc211747985)

[7. Quick Reference 50](#_Toc211747986)

# 1. File Organization

## 1.1 File Layout

**1.1.1 [MANDATORY] Files should have a file header. Use the following format for file header section content:**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\

\*\*

\*\* Filename

\*\*

\*\* Description

\*\*

\*\*

\*\* Copyright (c) <date> NSCL

\*\* All Rights Reserved

\*\*

\*\* Author:

\*\* Authors

\*\*

\*\* General Comments

\*\*

\*\*

\*\* $Header: $ ???

\*\*

\*\*

\*\* $Log: $ ???

\*\*

\\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**File Name** – required in the File Header.

The exact name of the file.

**Description** – required in the File Header.

A description of the file's purpose and content. If this file is part of a multi-file module, this section should briefly describe the module and this file's relationship in the module.

**Copyright** – required in the File Header.

Contains the following: "Copyright (c)", the copyright date(s), "…." in a new line. The copyright date is the date on which any code in the file was written. If you are starting a brand new file, you should just put the current year here (e.g.: "2005"). If you are modifying a file that has an older date, you should update it to the current year. For example, if you are editing a file that says "1991-1995", you should update it to read "1991-2005"; if the file says "1997", you should update it to "1997-2005".

**NOTE:** If the file is created or modified as part of a client project, make the copyright assignment as required by the client.

**Author** – optional in the File Header.

If the file was initially written by one person or extensively overhauled by someone else, their names should go here. The purpose of this is to provide the programmers who are working on this file, a person to contact, if they runs into problems or needs to make a questionable change. Your name should go here if you feel that you have a certain amount of ownership or expertise in this file and want it to be known to other programmers.

**NOTE:** If the file is created or modified as part of a client project, do not enter your name, without the client's consent.

**General Comments** – required in the File Header.

Comments about the file or special information about the file for example algorithms, required external libraries, and so on.

**Header** – required in the File Header.

This is the string "$Header: $" (no space between "$" and "Header:"). This is the keyword that Version Control System like CVS will update with project name, version, time, and programmer who most recently checked in this file. If not using CVS, put the equivalent keyword for the Version Control System that you are using here.

**Log** – required in the File Header.

The keyword used will differ depending on the version manager in use. Assuming that the source code control system in use is CVS, the string used is "$Log: $" (no space between "$" and "Log:"). This is the keyword that CVS will update with history information for the file whenever it is checked in.

**1.1.2 [OPTIONAL] Do not include code or C-style comments in the check-in message in the File Header.**

**1.1.3 [OPTIONAL] Use spaces for indentation instead of tabs.**

**1.1.4 [OPTIONAL] If tabs are used, tab width of 8 spaces are encouraged.**

With a tab width of 8 spaces and line length of 79 characters, it is difficult to have more than three or four levels of indentations on a line. Consider this as a good thing and also an opportunity to re-factor the code, so that the number of indentation levels can be reduced (for example, by moving parts of the code to a separate inline function). It results in overall reduction of code complexity.

## 1.2 Header File Layout

**1.2.1 [MANDATORY] Header files should be stand alone.**

If a header file,for example, header 1 uses definitions in another header file, for example, header 2, then header1 must #include “Header 2” only. If header2 in turn contains references to typedefs, constants, and so on declared in another header file, for example, header3, it is the responsibility of header 2 to #include “Header 3”.

Use forward declarations instead of #include directives and include the files that contain the declarations needed. This is preferred instead of including the files that contain files and have the declaration.

**1.2.2 [MANDATORY] Header files should contain only declarations, not definitions.**

Header files should not contain any constructs which will generate code or reserve memory (that is,. declarations). Do not declare global variable definitions (those not preceded by “extern”) and function implementations in header files. Inline functions are allowed as they do not actually generate code in the header.

**Example:**

/\*

\* header.h

\*/

#define MY\_STRING "Hello World" //OK

void WriteMyString(); //OK

char \*GetResidueName() {return residuename;} //OK

**1.2.3 [MANDATORY] Header files should be able to be compiled at Warning Level 4 without any warnings.**

**1.2.4 [MANDATORY] Every header file must have a mechanism to prevent multiple inclusion. Preprocessor directives must be the first line of the header file to prevent multiple inclusions. The #define for the filename should appear only in the header.**

This should be achieved by using a #ifndef/#define pair following the comments block at the start and a #endif at the end of the file. Please refer to 5.4.1 for details on **#ifndef/#define pair formatting.**

**Example:**

#ifndef EXAMPLE\_H

#define EXAMPLE\_H

...

....

...

#endif

**1.2.5 [OPTIONAL] Header file contents can be organized into the following sections:**

1. Required Includes

This section should contain any #includes for other header files which declare entities (constants, types, macros, and so on.) used by the include file itself.

1. Constants

This section contains any parameter less preprocessor (#define) constants, const and enum values.

1. Forward Declarations

This section contains any forward declarations.

1. Types

This section contains typedefs.

1. Extern Data

This section contains declarations of any data items which are external to this module.

1. Function prototypes
2. Inline prototypes
3. Macro Functions

This section contains macros that take parameters.

**1.2.6 [OPTIONAL] Place public (inter-module) interfaces and private (intra-module) interfaces in separate files.**

A header file can contain internal interface information to access inside the sub-system. This is private information and it should be in a separate header file from the public or external interface information, to hide as much information as possible from the external users.

## 1.3 Source File Layout

**1.3.1 [MANDATORY] Source file contents must be organized into the following sections:**

1. Required Includes

This section should contain any #includes for other header files which declare entities (constants, types, macros, and so on.) used by this source file.

1. Constants

This section contains any parameter-less preprocessor (#define) constants, const and enum values.

1. Types

This section contains typedefs.

1. Global Data

Any data which the file declares and initializes globally is declared and initialized here.

1. Static Data

Any data which is global to this file only is declared and initialized here.

1. Static function prototypes
2. Function prototypes
3. Functions

**1.3.2 [MANDATORY] Do not declare external data in a source file.**

To avoid making the same change in multiple locations, declare global data in the source file and place the extern reference in the public or private header, whichever is appropriate.

## 1.4 Declarations

### 1.4.1 Global Variables

**1.4.1.1[OPTIONAL] Do not use global variables unless absolutely necessary.**

**1.4.1.2 [OPTIONAL] To declare global variables always start in the first column.**

### 1.4.2 Variable Declaration

**1.4.2.1 [MANDATORY] Initialize variables while defining them and these definitions should be on separate lines.**

Declaring multiple variables on the same line is not recommended. The code might be difficult to read and understand.

**Example:**

|  |  |
| --- | --- |
| **Not Recommended** | **Recommended** |
| int value ; | int value = -1; |
| /\* Intialization on sameline\*/  int value = -1, count = 0; // NOT OK | int value = -1; // OK  int count = 0; // OK |

Some common mistakes are also avoided. Remember, that when you declare a pointer, a unary pointer is bound only to the variable that immediately follows.

**Example:**

|  |  |
| --- | --- |
| **Not Recommended** | **Recommended** |
| int i, \*ip, ia[100], (\*ifp)(); | // pointer to the old object  LoadModule \*oldLm = 0;  // pointer to the new object  LoadModule \*newLm = 0; |

**1.4.2.2 [MANDATORY] Do not declare variables with the same name in nested scopes.**

It becomes very confusing (especially in the debugger) if there are multiple variables named same, at any point in the execution.

**1.4.2.3 [MANDATORY] Use the keyword "volatile" for variables that are not modified directly by the program but by some external control logic (for example, a memory mapped register) to prevent inadvertent optimization by the compiler.**

**1.4.2.4 [OPTIONAL] Variable declaration blocks are separated from other code by one blank line.**

### 1.4.3 Enumerations

**1.4.3.1 [OPTIONAL] Format enumerations as follows:**

enum {

kConstant [= Value],

kConstant [= Value],

...

};

* The keyword enum is followed by a space.
* The opening brace follows the enum keyword with a space to the left.
* The constants in the enumeration are indented one tab from the enum keyword. There should be only one constant per line. Each constant may have an optional value assigned to it. Each constant but the last one is followed by a comma.
* The closing brace is on a separate line positioned in the same column as the enum keyword.
* A semicolon (";") immediately follows the closing brace with no space to the left.

**1.4.3.2 [OPTIONAL] Enumerated types should never be used in structures**

Unfortunately, the sizes of enumerated types are compiler dependent. Even within the same compiler, different settings can force different enumeration sizes. Therefore, enumerated types must not be used in structures that are written to disk, sent across a network, or used in function calls that may be exported through a library interface.

You cannot declare an enumerated type in a structure, and then assign an enumeration constant to it. The reason for this should be evident: one compiler might decide that the enumeration type needs only 1 byte to hold all values, and treat the type as an **uint8**, but another compiler might always treat all enumerated types as **int32**s. This affects the bytes that are allocated within the structure and therefore, makes the structure incompatible between the two compilers. But if you make the same field an **int16**, both compilers will generate the same structure, and implicitly coerce any enumeration constant to **int16**, regardless of what size it originally was.

### 1.4.4 Structures

**1.4.4.1 [MANDATORY] Format structures as follows:**

struct [Structure nameStruct] {

[/\* Offset \*/] Field list

} [Variable or Type list];

* The keyword struct is followed by a space.
* An optional structures name. This should end with "\_struct.
* The opening brace is at the end of the line containing the struct keyword, with a space to the left.
* Optionally, the offset of the field (in bytes) from the first element of the structure, is placed inside a comment. This comment starts on the same column as the "struct" keyword.
* The fields in the structure are indented one tab from the struct keyword (or one tab from the typedef keyword if there is one). There should be only one field per line, except in trivial cases.
* The closing brace is on a separate line, positioned in the same column as the struct keyword. If this structure is being typedefd, then the brace should line up under the typedef keyword instead.
* Any variables or types are placed after the closing brace, separated from it by one space.
* If there are no variables or types, a semicolon (";") immediately follows the closing brace, with no space to the left. If there are variables or types, then the semicolon immediately follows the last variable or type, with no space to the left.

**1.4.4.2 [MANDATORY] Declare structure with a name so that it can be used in a forward declaration.**

**Example:**

|  |  |
| --- | --- |
| **Not Recommended** | **Recommended** |
| struct  {  int hour;  int minute;  int second;  } current; | struct time  {  int hour;  int minute;  int second;  } current; |

**1.4.4.3 [MANDATORY] Fields within a structure must be 8-byte aligned.**

By default, compilers align individual structure elements on appropriate memory boundaries. This introduces additional padding bytes in places where they may not be expected (for example: when the structure is written to disk or sent across the network). To avoid this, and to avoid bloating of structure sizes with unwanted padding, we need to explicitly align structure fields on appropriate memory boundaries.

|  |  |
| --- | --- |
| **Not recommended** | **Recommended** |
| /\* NOT recommended as this structure occupies 12 bytes because of additional padding \*/  typedef struct  {  char c1;  short s1;  char c2;  long l1;  }mystruct\_t; | /\* OK as this occupies only 8 bytes with minimal padding \*/  typedef struct  {  long l1;  short s1;  char c1;  char c2;  }mystruct\_t; |

**1.4.4.4 [OPTIONAL] It is strongly discouraged to declare a named structure inside another structure and then use the name of the nested structure in another location.**

**Example:**

|  |  |
| --- | --- |
| **Not recommended** | **Recommended** |
| struct EmpInfo  {  int ID;  struct info {  char name[20];  char address[50];  } person[50];  }Employee; | struct info  {  char name[20];  char address[50];  } PERSONAL;  struct EmpInfo  {  int ID;  struct info person[50];  }Employee; |

**1.4.4.5 [OPTIONAL] Do not declare structure with more than 20 fields.**

This improves the readability and the maintainability of the code.

### 1.4.5 Unions

**1.4.5.1 [MANDATORY] Format unions as follows:**

union [Union nameUnion] {

[/\* Size \*/] Field list

} [Variable or Type list];

* The keyword union is followed by a space.
* An optional union name. This should end with "union".
* The opening brace is at the end of the line containing the union keyword, with a space to the left.
* Optionally, the size of the field is (in bytes), placed inside a comment. This comment starts on the same column as the "struct" keyword.
* The fields in the union are indented one tab from the union keyword (or one tab from the typedef keyword if there is one). There should be only one field per line, except in trivial cases.  
  The closing brace is on a separate line, positioned in the same column as the union keyword. If this union is being typedefd, then the brace should line up under the typedef keyword instead.
* Any variables or types are placed after the closing brace are separated from it by one space.
* If there are no variables or types, a semicolon (";") immediately follows the closing brace with no space to the left. If there are variables or types, then the semicolon immediately follows the last variable or type, with no space to the left.

**1.4.5.2 [OPTIONAL] It is recommended to define the union with a name so that it can be used in a forward declaration.**

**Example:**

|  |  |
| --- | --- |
| **Not recommended** | **Recommended** |
| union {  char birthday[10];  int age;  } people; | union speople {  char birthday[10];  int age;  } people; |

**1.4.5.3 [OPTIONAL] Do not declare unions with more than 20 fields.**

This improves the readability and the maintainability of the code.

**1.4.6 [MANDATORY] Use typedefs for data types. Do not use typedefs for pointers to data types.**

Using typedef struct str\* str\_p is dangerous since it reduces code readability and the user may not realize that it is a structure.

|  |  |
| --- | --- |
| **Not recommended** | **Recommended** |
| struct current\_time  {  int hour;  int minute;  int second;  } ;  typedef struct current\_time \*str\_p; | struct current\_time  {  int hour;  int minute;  int second;  } ;  typedef struct current\_time str\_t;  str\_t \*str\_p; |

## 1.5 Function Declarations

**1.5.1 [MANDATORY] Function declarations/prototypes must list all the arguments along with the variable names. If a function does not have an argument, then void must be specified.**

[Storage class] Return type Function name(Arguments);

**Example:**

|  |  |
| --- | --- |
| **Not recommended** | **Recommended** |
| extern int getsum( int , int ); | extern int getsum( int a, int b); |
| void Func1();  void Func1(int); | void Func1(void); //OK  void Func1(int count); //OK |

**1.5.2 [MANDATORY] Declare a prototype of all static functions before the first function in the source file.**

This avoids moving functions around in the file since static functions must be defined before their invocation.

**1.5.3 [MANDATORY] The return type is required even if the function does not return a value.**

If the function does not return a value, its return type is "void".

**Example:**

|  |  |
| --- | --- |
| **Not recommended** | **Recommended** |
| func1(int count); | void func1(int count); |

**1.5.5 [MANDATORY] Functions must not take enumerated types as parameters.**

**1.5.6 [OPTIONAL] For function declarations, write the leading parenthesis and the first argument on the same line as the function name.**

This improves the readability and the maintainability of the code.

|  |  |
| --- | --- |
| **Not recommended** | **Recommended** |
| void foo(  int i,  int j); | void foo( int i,  int j); |

**1.5.7 [OPTIONAL] Function declarations/prototype:** **The storage class designation is not required for declaration of non-static functions.**

Storage class keyword is either "static" or "extern". If the storage class is not mentioned, it is assumed by default to be “extern”. Hence non-static functions, that is, extern do not need the storage class to be explicitly mentioned.

**1.5.8 [OPTIONAL] The close parenthesis for the argument list is followed immediately by a semicolon (;) without a space.**

**Example:**

|  |  |
| --- | --- |
| **Not recommended** | **Recommended** |
| void Func1(int count) ; | void Func1(int count; |

## 1.6 Initialization

**1.6.1 Initialization of Variables and Constants**

**1.6.1.1 [MANDATORY] Initialize all pointer variables to NULL.**

This helps to avoid dereferencing of un-initialized pointer variables.

**Example:**

|  |  |
| --- | --- |
| **Not Recommended** | **Recommended** |
| void foo()  {  /\* Not intialized \*/  int\* ptr;  } | void foo()  {  /\* Intialized to NULL\*/  int\* ptr = NULL;  } |

**1.6.1.2 [OPTIONAL] Do not declare the array size when the array is initialized.**

Allow the magnitude of an array to be set automatically during definition. This way, changes to the number of elements in the array do not require corresponding changes to the explicit array size, making the code easier to maintain.

**Example:**

|  |  |
| --- | --- |
| **Not Recommended** | **Recommended** |
| #define SIZE 4  int tab1[SIZE] = {1,2,3}; | int tab2[] = {1,2,3}; |

**1.6.1.3 [OPTIONAL] Do not declare the size of the array in the function prototype/definition when it is used as a parameter in a function call.**

Different invocation of the function pass array arguments with different magnitudes.

Explicitly specifying the magnitude of array in a function argument definition will make the code difficult to maintain. Therefore, do not declare size of the array.

**Example:**

|  |  |
| --- | --- |
| **Not Recommended** | **Recommended** |
| void foo2(int x[30])  {  } | void foo2(int x[]) |

# 2. Naming Conventions

## 2.1 File Names

**2.1.1 [MANDATORY] Filenames should only contain roman alphabetic, numbers, and underscores in the format: [a-z] [0-9 a-z A-Z \_\.]\* .**

**2.1.2 [MANDATORY] Use underscores to separate words in filenames.**

**2.1.3 [MANDATORY] The length of a filename should not exceed 31 characters.**

**2.1.4 [OPTIONAL] As far as possible do not use numbers in filenames. Use numbers only when they say something meaningful about the functionality contained in the file.**

## 2.2 Identifier Names

### 2.2.1 General

**2.2.1.1 [MANDATORY] Identifiers should use underscores to delineate words within the identifier.**

This applies only to the identifiers that are not rendered in mixed-case.

**2.2.1.2 [MANDATORY] All preprocessor identifiers that do not take parameters should be all uppercase.**

For parameterized macros

Parameterized preprocessor macros fall into two categories: **real macros** and **function emulators**. Real macros are any macros that have side effects or will never be (or cannot be) implemented as a function. Function emulators do not have side effects and may conceivably be changed to a function someday.

A macro always has side effects if it:

* Changes the value of any parameter.
* Uses any parameter more than once.

Real macros should be in all uppercase to clearly identify these as macros that have side effects.

Function emulators may be in either all uppercase or mixed case with initial letter of each word capitalized and an initial uppercase.

**2.2.1.3 [OPTIONAL] Identifier names should be rendered in mixed-case or all lowercase. If using mixed-case, the first character should be lowercase**

**2.2.1.4 [OPTIONAL] Names must be composed of American English words.**

We need to do this for consistency across all our applications. An engineer should not have to remember if the color library is a "color\_lib" or a "colour\_lib." This also means that hard to understand or pronounce words should not be used in the variable names. Also, English words should not be cruelly tortured, mangled, or spindled, otherwise mutilated before being used in the code; specifically, all vowels should be left in identifier names.

**2.2.1.5 [OPTIONAL] Negative variables are discouraged as they can introduce confusion easily.**

I.E. if (!notLoggedOn && !(noBytesLeft && transferDone)) {...}.

Variable names may be descriptive of their contents ("selectedItem") or their functions ("memoryWasAllocated") are appropriate.

Function names many times can be formed as a simple Verb-Noun (or Noun-Verb) pair ("DeleteNode()", "ServerLogoff()"). Functions that test states may contain linking verbs ("IsTextBox()", "DocHasMargins()").

**2.2.1.6 [OPTIONAL] Short variables should be used with caution.**

The use of very short variables, such as "i" as a loop control variable, "c" for a single character, and "cp" for a pointer to a character, are something like a C tradition. Although we do not discourage the use of these types of variables, you should consider the following before using them:

#### Advantages:

* Simple, short names are quick to type, and easy to read in a very limited scope. For example, the loop "for (cp = str, i = 0; \*cp != '\0'; i++, cp++) {}" is quickly recognized as finding the length of a C string, and it is short and concise.
* Since "i" and "cp" have been in use for so long and are used in so many C instructional materials, they can be considered (in some light) as standard, well defined variables. If an engineer sees "i" in use in a function, they generally assume that it is a loop control variable and may be used as such.

#### Disadvantages:

* Using variables like "i" for anything other than a loop control variable confuses other engineers until they figure out that it is not a loop control variable.
* "i" and "cp" work well in small functions or loops but their use in larger functions is likely to introduce bugs.For example: Often after a loop, the termination condition may be checked by testing "if (i < maxLoopValue) {...}" to see if the loop terminated early. In large functions it is possible that these tests get separated from the loop itself. After this happens, if someone sees "i" and uses it for a new loop between the original loop and the test, the value of "i" is corrupted and a bug is introduced which is very tricky to find.
* One character constants are pain to search, since you have to turn on whole words only.
* Short variable names are not maintainable as long ones and can cause conflicts.

### 2.2.2 Identifier Character Set Rules

**2.2.2.1 [MANDATORY] Identifiers should only contain letters from the roman alphabet, underscores, and numbers [a-z A-Z] [a-zA-Z0-9\_]\*.**

**2.2.2.2 [MANDATORY] Limit the use of numbers in an identifier to a case where the number indicates something significant about the contents or purpose of the identifier. It can also be used in a case where multiple items otherwise have nothing to distinguish them from each other.**

**2.2.2.3 [OPTIONAL] Identifiers must not start with an underscore ("\_") or a double underscore ("\_\_").**

Compilers generally generate internal identifier names with leading underscores, so this should not be done in our code.

### 2.2.3 Decorators

**2.2.3.1 [OPTIONAL] Use of Scope Decorators is encouraged for certain category of identifiers.**

The use of scope decorators should follow the given table:

|  |  |  |
| --- | --- | --- |
| **Identifier Category** | **Decorator** | **Example** |
| Enumeration elements | K | kErrCode\_BadID |
| Application-or-File-scope variables | G | gNodeID |
| Function – static variables | S | sCount |

Scope decorators help to identify important semantic consequences of the scope of the variable/constant that they decorate. Scope decorators should not be used with any other identifier categories like local variables or structure members.

### 2.2.4 Types

**2.2.4.1 [OPTIONAL] Render non-simple type names, anything other than a redefinition of an intrinsic type (that is. int16 et al.) either in mixedcase or all lowercase.**

If using mixedcase, the first character should be uppercase. If using lowercase, words should be delineated using underscores.

## 2.3 Function Names

**2.3.1 [OPTIONAL] Render function names in mixed case or all lowercase.**

If using mixedcase, the first character should be uppercase. If using lowercase, delineate words using underscores.

## 2.4 Variable Types and Prefixes

|  |  |  |
| --- | --- | --- |
| **Variable Type** | **Prefix** | **Example** |
| Pointer Variables | p | int\* pnNumber = NULL; |
| Void Pointer declarations | v | void\* g\_pvmy; |
| Constant Parameters | k | void fun(const unsigned char kucparam) |
| Global Variables | the | int theWindows; |
| Static Variables | s | static int s\_nCountOfInstances; |
| Structure Names | S | struct SEmployeee {  // data declarations  }; |
| Enum Names | e | enum eColors { COLOR\_RED, COLOR\_GREEN, COLOR\_BLUE } |
| Array Names | rg | int rgiTab[10]; |

## 2.5 Pointer and Array Variables

|  |  |  |
| --- | --- | --- |
| **Pointer/Array Type** | **Prefix** | **Example** |
| Int Pointer and array variables | i | int\* piMyVariable; |
| Short int Pointer and array variables | si | short\* siMyVariable; |
| Long int Pointer and array variables | li | long int\* pliMyVariable; |
| Word Pointer and array variables | w | unsigned short\* wMyVariable; |
| Dword Pointerand array variables | dw | unsigned long\* m\_pdwMyVariable; |
| Bool Pointer and array variables | B | bool \*g\_pbMyVariable; |
| Byte Pointer and array variables | by/y | unsigned char \*m\_pbyMyVar2; |
| Char Pointer and array variables | c | Char \*pcMyParam; |
| Floating Point Pointer and array variables | f | float \*g\_pfMyVar; |
| Double precision floating point Pointers and array variables | d | double \*g\_pdMyVariable; |
| Long double precision floating point Pointers and array variables | ld | long double \*pldMyParam; |
| Null-terminated string pointers and array variables | sz | char (\*g\_pszMyVar)[1]; |
| String pointers and array variables | str | string \*pstrMyParam; |

## 2.6 Variable Names

Variable names use prefixes (based on a reduced form of ‘Hungarian’ simplified notation) to identify the storage and the type of the variable. Use long names for clarity

The general format of variable names is:

StoragePrefix + “\_” + TypePrefix + VariableName

For example m\_bLogEnabled a Boolean class member

* The following prefixes identify the storage occupied by a variable:
* **s**  Static Variable
* **None !** Automatic (local) variable

The following prefixes identify the type of the variable:

|  |  |  |
| --- | --- | --- |
| **Variable** | **Prefix** | **Example** |
| Boolean types | b | bEnabled |
| Numerical integer types | n | nFlags |
| Handle Types | h | hWnd |
| Pointers, smart pointers and arrays | p | pValue |
| string values/ array of char | str | szCity |
| signedchar variables | c | cMyVariable |
| unsignedchar / byte | by/y | byMyVariable |
| float variables | f | fMyVariable |
| double precision floating point variables | d | dMyVariable |
| Short int | si | siMyVariable |
| Signed long int | li | liMyVariable |
| Long double precision floating point variables | ld | ldMyVariable |
| Unsigned long int/dword | dw | dwMyVariable |
| Unsigned int/ word | w | wMyVariable |

# 3. Documentation

## 3.1 Comment Types and Usage

### 3.1.1 General comments

**3.1.1.1 [MANDATORY] Comments are not a substitute for clearly written code.**

Comments can give insight or overviews that are not obvious.  The comments should describe what is happening, how it is being done, what do the parameters mean, which global variables are used and which are modified, and any restrictions or bugs. However, avoid comments that are clear from the code, as such information can be collected from the code and is unnecessary. Comments that disagree with the code are of negative value. Short comments should be ‘what comments’, such as "compute mean value", rather than ‘how comments’, such as "sum of values divided by n". C is not an assembler; putting a comment at the top of a 3-10 line section telling what it does overall is often more useful than a comment on each line describing micro logic.

Remember that many people will be reading the code over the lifetime of the product. A useful test for a comment is to consider "if I was on the information highway and want to, run over by an Information Truck, would this comment be useful for someone to continue to maintain my code, even when they cannot ask me any questions?"

**3.1.1.2 [OPTIONAL] Comments must be indented at the same level as the code block that they describe.**

Comments should assist in the understanding of the code. Comments which are not at the same level as the code which they describe inhibit the easy reading of the code and interfere with the visual flow.

**3.1.1.3 [OPTIONAL] Comments must be provided for each non-trivial variable declaration or any variable which is not self-commenting.**

Trivial variables are simple variables in short functions. Such variables should be self-commenting and should be used in a straight-forward manner within a localized scope. Examples of such variables are *colorIndex*, *soundToPlay*, *error*, or *storyWordCount*. Such variables need not be commented.

Variables whose function is not clear from their name, or can be easily confused with other variables must be commented. Generic or obscure variable names like *boxid*, *index*, *thing1*, *thing2*, and *tryToCatchUp* are discouraged and if used must be commented unless their use is trivial, as to be obvious to the most casual observer (that is, a COBOL programmer).

Comments that merely restate the variable name are not useful and should be left out or replaced.

|  |
| --- |
| **Not Recommended** |
| int count; /\* Variable used for counting \*/ |
| Char \*StudentName; /\* Variable used to store student name \*/ |
| int32 SocketDescriptorID /\*Unique ID of socket descriptor \*/ |

**3.1.1.4 [OPTIONAL] Provide comments should for each field or member of a structure.**

**3.1.1.5 [OPTIONAL] Do not use C++ style comments in C code.**

Even though C++ style comment ("//") seems useful for single-line comments, mixing the two styles is not recommended.

**3.1.1.6 [OPTIONAL] Offset comments for a single line of code by at least one tab from the end of the code line.**

This helps a reader to follow the logic of the code by picking out comments, without searching the code for the comments. This does not say that if only one line of code is commented, then the comment must be at the end line of the code. The single line comment can go on the previous line if desired. The intent is to specify the style of comments that you want to put with one line of code.

|  |
| --- |
| **Recommended** |
| for (i = 1; i <= MAXARRAYSIZE; ++i) {     // array is base 1    array[i] = FIX(array[i]);              // convert to fixed } |
| for (i = 1; i <= MAXARRAYSIZE; ++i) {     /\* array is base 1 \*/    array[i] = FIX(array[i]);              /\* convert to fixed \*/ } |
| for (i = 1; i <= MAXARRAYSIZE; ++i) {     /\* array is base 1 \*/    /\* convert to fixed \*/    array[i] = FIX(array[i]); } |

**3.1.1.7 [OPTIONAL] Explicitly identify source code, introduced solely for suppressing compiler-warnings / lint- messages by a suitable comment.**

## 3.2 Function Level Documentation

**3.2.1 [OPTIONAL] Provide every function/method in a file with a function comment having the following form:**

/\*\*

\*

\* Function Name

\*

\* Description

\*

\* @param Parameter Direction Description

\* @param Parameter Direction Description

\*

\* @return Description

\*

\* @exception Description

\*

\*/

The following are the description of the given comment block.

* The comment block is initiated by the /\*\* (forward slash and two stars) symbol.
* The first line of comment beginning with a \*, gives the function name.
* The second line is used to give a description of the function.
* The third line of comment uses the @param keyword that creates a parameter section in the documentation for this function and describes the specified parameter with the text that follows.
* The @return keyword creates a section in the documentation detailing return values.
* The comment block is ended by the \*/ (star forward slash) symbol.

**3.2.1.1 Function Header Section Tags**

**Function Name** - required

Exact name of the function. Optionally include the fully qualified name.

/\*\*  
 \*  
 \* Shape::Draw  
 \*/

**Description** - required

Description of what the function does. This should not be a description of *how* the function does something, but a description of *what* or *why* the function does.

/\* GetServerName  
\*  
\* Retrieves the name from the server the user is currently logged\* on to. This is the name the user should see in any User

\* interface.

\*/

**List of parameters** - required if any arguments

Describes the argument and value of each argument into or out of the function. Each entry under this section needs to contain:

* A tab.
* @param
* A tab
* The name of the argument.
* Input arrow ("-->" – minus minus greater-than) if the argument's value is significant when the function is called.
* Output arrow ("<--" less-than minus minus) if the argument's value is significant when the function exits. Note that if the value is unchanged from when the function is called, it is not considered significant and need not be specifically listed as an output value.
* A description of the argument and its value. If a function has different values on input and output, use the "<->" (less-than minus greater-than) and combine the input/output descriptions.
* If the argument contains flags, or an enumeration constant, you must include a possible input value as an example. It can be very difficult to figure out what set of flags or constants are valid, especially, if the flag is not used in the function, but is just passed into another function. With an example, another programmer can figure out what valid parameters are by using the browser on the example and finding where the macros, constants, or enumerations are defined. Once the programmer has found the declarations for valid parameters, they should be able to determine from the comments there what will happen if they pass each value in. Also, note the next bullet. If the entire set of values is not listed, you should include "etc." to indicate that other values are also legal.
* If the argument takes a small set of values and the behavior of the function is not obvious,you should include a list of values and the behavior they produce.

**Complete list if convenient:**

 \* DrawLine  
 \*  
 \* @param   lineType --> Type of line to draw (kLineType\_Solid,  
 \*          kLineType\_Dotted, kLineType\_Dashed, kLineType\_None).

**At least one example with "etc.":**

 \* DrawLine  
 \*  
 \* @param   lineType --> Type of line to draw (kLineType\_None, etc.).

\* DrawLine  
 \*  
 \* @param  useBlack --> TRUE - should draw the character in black  
 \*                      FALSE - draw character in the default

\* color as found in gCharDefColor

**@return** - required if does not return void

Specifies the meaning of the return value. If a handle or pointer is returned, document whether the caller is expected to dispose of it or not.

**Globals** - required if globals used

Specifies any global variables used or changed by the function/method. This section should have the same format as the Arguments section.

**Side Effects** - required if there are side effects

Specifies any side effects of the function/method. This may include whether memory was moved, something was drawn on the screen, system state ws changed (that is, window closed), and so on.

**Notes** - optional

Useful information about the function/method. This may include where to go for more documentation, documentation of non-obvious uses of the function, comments on future or desired behavior that is not implemented yet, ideas for optimization, and so on.

**Algorithm** - required if algorithm is not obvious

Specifies the algorithm used by the function/method for whatever it is that it does. For example,, a sorting function may indicate what sorting method is used.

**Warning** - optional

Provides a warning about the use/behavior of the function/method. This may include warning to lock memory, non-obvious but dangerous ramifications of the side effects, and so on.

**Others - optional**

You may include other sections if you find them necessary. However, you should try to use one of the list, if possible, just for consistency.

## 3.3 Special Comments

Special comments are used to call attention to areas in the code.These might be used to indicate that action is required or to indicate that a hack is about to happen.Some development environments (like .Net Studio) have a special utility which will parse through the source code and create "to do" lists based on these special tagged comments. Follow the rules given to find the right special comment tag to use.

**3.3.1 [MANDATORY] Format the Time sensitive special comments with three dollar signs ("$$$").**

Time sensitive special comments use "$$$" instead of "TODO" and include some sort of deadline following the date, in the form:

/\* $$$ Name Date Deadline - Comment \*/

/\* $$$ Name Date Deadline - Comment

\*\* (continued on multiple lines)

\*/

Following are the description of the comment block.:

* A normal C ("/\* \*/") comment which contains as the first non-whitespace characters three dollar signs ("$$$").
* The name of the person entering the comment.
* The date when the comment was entered in DD-MMM-(YY)YY format.
* The deadline by which action needs to take place.
  + The text of the comment.

These issues are those that ***must*** be addressed by some definite point in time (as opposed to should be addressed sometime, which is what "TODO" means). The deadline is a required component of a "$$$". Historically, the "$$$" comment was introduced because of the overuse of "|||".  The symbol "$$$" was selected since not addressing these issues may result in bad software, and therefore, we may lose money.

The deadline may be a fairly free-form. You may use this for personal notes, so that the deadline may be "Integration"; and searching on "Integration" will find everything that you intend to address, before integrating your code.

**3.3.2 [OPTIONAL] Code needs some kind of follow-up action: TODO**

/\* TODO Name Date - Comment \*/

/\* TODO Name Date - Comment

\*\* (continued on multiple lines)

\*/ \*

The following are thedescription of the comment block:

* A normal C ("/\* \*/") comment which contains as the first non-whitespace characters the word "TODO".
* The name of the person entering the comment.
* The date when the comment was entered in DD-MMM-(YY)YY format.
* An optional deadline.
* The text of the comment.

**3.3.3 [OPTIONAL] For multi-line comments do not include a “TODO” on any line after the first one.**

To find and research special action comments, programmers may need to search for "TODO" in the code. A search of this nature should result in exactly one hit per TODO comment ;so that the searcher has a reliable count of the number of issues to address. This also makes it easier to run through the list of TODO comments to see that how old they are without getting distracted by extraneous information.

**3.3.4 [OPTIONAL] Use the HACK special comment if you are placing a “hack” in a file.**

"Hack" is an inelegant, temporary, and probably obtuse solution to a particular problem. (You recognize a hack when you code one!). Such hacks should be prefaced with a HACK special comment in the form:

/\* HACK Name Date - Comment \*/

/\* HACK Name Date - Comment

\*\* (continued on multiple lines)

\*/

Following are the description of the comment block:

* A normal C ("/\* \*/") comment which contains as the first non-whitespace characters the word "HACK".
* The name of the person entering the comment.
* The date when the comment was entered in DD-MMM-(YY)YY format.
* The text of the comment.

**3.3.5 [OPTIONAL] Use the BUGFIX special comment to identify a bug fix.**

/\* BUGFIX BUG\_ID: ID Name Date - Comment \*/

/\* BUGFIX BUG\_ID: ID Name Date - Comment

\*\* (continued on multiple lines)

\*/

Following are the description of the comment block:

* A normal C ("/\* \*/") comment which contains as the first non-whitespace characters the word "BUGFIX".
* The string "BUG\_ID:" and a space, followed by the bug identifier.
* The name of the person entering the comment.
* The date when the comment was entered in DD-MMM-(YY)YY format.

(MMM – month in three letter format, that is, JAN, FEB, MAR, and so on)

* The text of the comment.

**3.3.6 [OPTIONAL] Use the Bug fix comments sparingly. The fix details should be part of the check-in logs.**

A single bug fix may involve modifying, adding,or deleting several lines of code across multiple files. Adding a bug fix comment at all the places is unnecessary and inappropriate. Check-in logs and the bug databases are the places for describing bug fixes.

## 3.4 Commenting out code

Commenting out code is something that every engineer does when debugging, and while debugging a particular module, it is very clear to that engineer what was commented out and why. With commented code it is difficult to determine whether the code has been commented out for debugging the purposes and left commented out accidentally or if the code is unnecessary and can it be permanently deleted. .

**3.4.1 [MANDATORY] Old, unused code should be deleted. Check in code that contains commented out blocks of code only when absolutely necessary.**

# 4. Formatting Conventions

## 4.1 Expression Formatting

### 4.1.1 Spacing

Spacing should be used to enhance readability and consistency. All the spacing rules here are recommendations. These are the suggestions which, if followed, would lead to good readable and maintainable code. To make an expression readable, these rules should be followed in entirety unless the engineer feels that specific expressions can be made more readable and easier to understand with different spacing.

**4.1.1.1 [OPTIONAL] Always put space after ‘,’**

**4.1.1.2 [OPTIONAL] Always put space between binary arithmetic, logical and bitwise operators and operands.**

**4.1.1.3 Use the following table for spacing recommendation**

|  |  |  |
| --- | --- | --- |
| **[MANDATORY]**  **Unary Operators – No space between operator and operand** | | |
| **Operator Type** | **Examples** | **Spacing** |
| Unary Arithmetic | + - ++ -- | No space between operator and operand |
| Unary Logical | ! | No space between operator and operand |
| Unary Bitwise | ~ | No Space between operator and operand |
| Unary Dereference | \* & | No Space between operator and operand |
| Reference Declaration | \* & | No Space between operator and operand |
| **[MANDATORY]**  **Other Operators – No space between operator and operand** | | |
| **Operator Type** | **Examples** | **Spacing** |
| Binary Dereferences | **.** -> | No spaces on either sides |
| Scope Resolution | **::** | No spaces on either sides |
| Parenthesis | ( ) | No spacing between operator and enclosed expression |
| Array References | [ ] | No spacing between operator and enclosed expression.  No space to left of [ |
| List Element Separator | , | No space on left. Space required on right |
| Statement Separator | , | No space on left. Space required on right |
| Statement Terminator | ; | No space on left. Newline required on (unless in for) |
| Typecast | (Type) | No space between parenthesis and type. No space between cast and expression. |
| **[OPTIONAL]**  **Other Operators – Space between operator and operand** | | |
| **Operator Type** | **Examples** | **Spacing** |
| Binary Arithmetic | + - % / \* | Spaces required on both sides |
| Binary Logical | && || | Spaces required on both sides |
| Binary Bitwise | ^ | & >> << | Spaces required on both sides |
| Bit-field Declaration | **:** | Spaces required on both sides |
| Assignment | **= += /= -= \*=** | Spaces required on both sides |
| Comparison | **== != < > <= >=** | Spaces required on both sides |
| Ternary Operator | ? | Spaces required on both sides. |

### 4.1.2 Parenthesis

**4.1.2.1 [OPTIONAL] The use of parenthesis around a basic expression is discouraged.**

**4.1.2.2 [OPTIONAL] Always place left parenthesis directly after the function name.**

This improves the readability and the maintainability of the code.

**Example:**

|  |  |
| --- | --- |
| **Not recommended** | **Recommended** |
| int foo ()  {  } | void foo()  {  } |

**4.1.2.3 [OPTIONAL] Always use parenthesis with the “return” and “sizeof” statements.**

This improves the readability of the code.

|  |  |
| --- | --- |
| **Not recommended** | **Recommended** |
| void foo()  {  return true;  } | void foo()  {  return(true);  } |

**4.1.2.4 [OPTIONAL] Always use a single ASCII space between conditional keyword and its opening parenthesis.**

This promotes the readability by providing a clear indication that a statement is a conditional statement or a function, and by providing a clear break between the conditional keyword and its associated expression.

**Example**

|  |  |
| --- | --- |
| **Not recommended** | **Recommended** |
| if( x==y )  {  } | if( x==y )  {  } |

### 4.1.3 Line Breaking

Lines should be restricted to 79 printable characters in length.

**4.1.3.1 [MANDATORY] Line breaking must be consistent on a file level**

**4.1.3.2 [OPTIONAL] Break lines in excess of 79 characters according to following guidelines**

* Continuation lines should be indented two tabs from the current block level. This helps to show that the line is a part of the previous line, but is not confused with a following code block that is indented one tab. So, if an expression is broken 3 times onto 4 separate lines, the last 3lines should all be indented 2 tabs from the first line.
* Continuation of lines should start with an operator. This makes it clearer that the line is a continuation of the previous line. (Lines broken in a function call and which would start with a comma are exempted.)
* Conceptual units should be kept together. Break the line at a position which keeps the sub-expressions together, not just at the token closest to the right edge of the 79 character length.
* If a function declaration is broken in the argument list, the line should be broken after a comma and the new line indented two tabs. Note that it may make more sense to break a line before the function name than in the argument list. In fact, breaking before the function name is preferable to breaking after the function name.

### 4.1.4 Pointer Variables

**4.1.4.1 [MANDATORY] Place the ‘\*’ close to the variable and not to the data type.**

**Example :**

|  |  |
| --- | --- |
| **Not recommended** | **Recommended** |
| int\* piMyVariable; | int \*piMyVariable; |
| char\* EmployeeName; | char \*EmployeeName |

## 4.2 Statement Formatting

### 4.2.1 Labels and goto

**4.2.1.1 [OPTIONAL] Use labels only as targets for goto statements.**

**4.2.1.2 [OPTIONAL] Avoid the “goto” statement as far as possible.**

The goto statement may be used only to jump to a cleanup label at the end of the function, when an error condition has been encountered.

**4.2.1.3 [OPTIONAL] Labels should be aligned at column 1 (that is,. left aligned).**

### 4.2.2 Function

**4.2.2.1 [OPTIONAL] Format functions as follows:**

Function comment

[Storage class] Return type Function name(Arguments)

{

Statement list

} [/\* Function name \*/]

Following are the description of the blocks:

* First line is a comment that describes the function. This comment format should be similar to the format described in Section 3.2 “Method Level Documentation”.
* The function definition must be the same as the prototype.
* The opening brace is a new line by itself aligned with the start of the function.
* The statements in the function body are indented one tab. This does not apply to the [Labels](http://blr-ec-mbx02.wipro.com/kalyani.angal/Inbox/RE:%20C%20Rules%20in%20VelociQ-WCC%20spreadsheet-4.EML/C%20Coding%20Standard.htm/C58EA28C-18C0-4a97-9AF2-036E93DDAFB3/StyleGuide.htm#9.12), the [Preprocessor directives](http://blr-ec-mbx02.wipro.com/kalyani.angal/Inbox/RE:%20C%20Rules%20in%20VelociQ-WCC%20spreadsheet-4.EML/C%20Coding%20Standard.htm/C58EA28C-18C0-4a97-9AF2-036E93DDAFB3/StyleGuide.htm#7.1.2), or the statements included in other statements.
* The closing brace is on a separate line aligned with the start of the function.
* For long functions, you may optionally put the name of the function in a comment, separated by one tab or space from the close brace. This may help in navigating the file and assist in locating context quickly, when looking at the end of a function.

**4.2.2.2 [OPTIONAL] Format “return” statements as follows:**

return (return-expr);

or

return;

* There is a space after the return keyword unless it is a return from a void function.
* The return expression is always enclosed in parenthesis.

**4.2.2.3 [OPTIONAL] Minimize the number of return statements in a function.**

Single entry, single exit should be the goal. Sometimes it might be easier to have multiple return statements, but whenever there is a way to rewrite it with a single return and without unnecessarily increasing complexity to read, functions should only have one return.

### 4.2.3 Line Breaking

**4.2.3.1 [OPTIONAL] Break the lines broken within the Condition clause of the conditional statement wherever possible, at the highest level logical conjunctive ("&&"or "||") or comparison operator ("==", ">", etc.).**

This makes it easier to visually identify and group higher level constructs. Continued lines are always indented two tabs.

if ((( gcount == 100 ) && (( lockEnabled ) || ( overrideFlag ))) ||

( redirCount == 5 ))

{

/

../

}

### 4.2.4 For

**4.2.4.1 [MANDATORY] Format "for" statements as follows:**

for (Initialization; Condition; Update) {

Statement list

}

* There is a space after *for*.
* Condition is a Boolean expression, not an integer expression.
* There are spaces after the semicolons (";").
* The opening brace is on the same line as the end of the Update clause and set off by one space.
* The closing brace is at the same level of indentation as the *for* keyword and on a line alone.

**4.2.4.2 [MANDATORY] Line-breaking in a "for" statement.**

Lines which need to be broken within the Initialization; Condition; Update clause should be broken after all semicolons and indented two tabs.

for (manualOverrideIndex = 0;

manualOverrideIndex < 10 ;

manualOverrideIndex ++) {}

**4.2.4.3 [OPTIONAL] Format the special case of a "for" statement with an empty body as follows (although its use is discouraged):**

for (Initialization; Condition; Update) {};

* The open/close brace pair is on the same line as the end of the Update clause and set off by one space.

### 4.2.5 If :

**4.2.5.1 [MANDATORY] Format "if" statements as follows:**

if (Condition) {

Statement list

}

* There is a space after the *if* keyword.
* Condition is a **Boolean** expression, not an integer expression.
* Always use braces for if statements
* The opening brace is on the same line as the end of the Condition clause and set off by one space.
* The closing brace is at the same level of indentation as the *if* keyword and on a line alone.

**4.2.5.2 [MANDATORY] Format "if else" statements as follows:**

if (Condition) {

Statement list

} else {

Statement list

}

* There is a space after *if* and *else*.
* Condition is a Boolean expression, not an integer expression.
* The opening brace is on the same line as the end of the Condition clause for the if and the same line as the else and set off by one space.
* The closing brace for each clause is at the same level of indentation as the associated main keyword that is, “if”.
* “else if” / “else” keyword is on the same line as the associated if closing brace.
* The last closing brace is at the same level of indention as the associated keyword and on a line alone.

**4.2.5.3 [MANDATORY] Do not use the single line representation of an if statement:  if (Condition) Statement.**

**4.2.5.4 [MANDATORY] Format cascading "if else if" statements as follows:**

if (Condition) {

Statement list

} else if (Condition) {

Statement list

} else if (Condition) {

Statement list

} else {

Statement list

}

* There is a space after all *if*s and *else*s.
* The *else if* keyword pairs are on the same line.
* The opening braces of each *else if* clause are on the same line as the end of the associated Condition clause, and set off by one space.
* The closing brace for each clause is at the same level of indentation as the associated main keyword, that is,. “if”.
* “else if” / “else” keyword is on the same line as the associated if closing brace.
* The last closing brace is at the same level of indention as the associated keyword and on a line alone.

**4.2.5.5 Conditional expression with the ternary operator ("?:")**

**4.2.5.5.1 [MANDATORY] Format the conditional expression n one of the 2 ways listed:**

var = (Condition) ? (value0) : (value1);

var = ((Condition)

? value0

: value1);

**4.2.5.6 [OPTIONAL] If all of the Condition clauses are all constants, consider using a switch statement.**

### 4.2.6 While

**4.2.6.1 [OPTIONAL] Format "while" statements as follows:**

while (Condition) {

Statement list

}

* There is a space after the *while* keyword.
* Condition is a Boolean expression, not an integer expression.
* The opening brace is on the same line as the end of the Condition clause and separated by one space.
* The closing brace is at the same level of indentation as the while keyword and on a line alone.

**4.2.6.2 [OPTIONAL] Format "do-while" statements as follows:**

do {

Statement list

} while (Condition);

* There is a space after the *do* keyword.
* Condition is a Boolean expression, not an integer expression.
* The opening brace is on the same line as the *do* keyword.
* The closing brace is at the same level of indentation as the *do* keyword and followed on the same line by the *while* keyword and Condition clause.
* The *while* keyword is separated from the block-enclosing brace by one space and from the Condition clause by one space.

### 4.2.7 Switch

**4.2.7.1 [OPTIONAL] Format "switch" statements as follows:**

switch (Expression) {

case Constant:

Statement list

break;

case Constant:

case Constant:

Statement list

break;

case Constant:

Statement list

// fall-through

case Constant:

{

Statement list

}

break;

default:

Statement list

break;

* There is a space after the *switch* keyword.
* The opening brace is on the same line as the end of the Expression clause and separated by one space.
* The closing brace is at the same level of indentation as the *switch* keyword and on a line alone.
* Each *case* keyword is at the same level of indentation as the *switch* keyword.
* There is one space between the *case* keyword and the Constant.
* There is no space between the Constant and its terminating colon.
* Statements associated with a *case* clause are always on a new line below and indented one tab.
* The *break* statement is indented one tab from its associated case clause(s).
* Fall-through cases always contain a comment acknowledging a fall-through. If there is no code in the *case*, then this comment is not necessary.
* If it is necessary to declare variables local to the scope of a case, braces are indented one tab, and code within the braces are indented another tab, but the break keyword is lined up with, and outside the braces.

**4.2.7.2 [MANDATORY] Always include a "default" case in a "switch" statement. The “default” case in the switch statement should be the last case.**

This makes the developer think about this case -- even if all that happens in the default case is an ASSERT statement.

**4.2.7.3 [MANDATORY] Do not break lines in switch Expression clause.**

If the Expression is so complex as to require a line break, consider assigning into a variable and switching on that.

### 4.2.8 Return/ Sizeof statements

**4.2.8.1 [OPTIONAL] There should be no white spaces between the “return” or “sizeof” statements and its opening parenthesis and surrounding “return” or “sizeof” statements argument or expression.**

|  |  |
| --- | --- |
| **Not recommended** | **Recommended** |
| void foo1() {  sizeofint ( int );  } | void foo1() {  sizeofint(int);  } |

# 5. Preprocessor

## 5.1 General

**5.1.1 [OPTIONAL] The preprocessor directive is separated from any arguments by a space.**

This means that there should be one space between #define and MINVALUE.

**Example:**

#define MINVALUE 5

**5.1.2 [OPTIONAL] Within the code, align ‘#’ of all preprocessor directives in the left most column.**

The Preprocessor directives should not flow with the rest of the code. Since preprocessor directives may change the behavior of the code drastically and unexpectedly, it is better that they draw the attention of the reader, rather then blend with the code.

**5.1.3 [OPTIONAL] Within a block of preprocessor directives, indent the "#" logically.**

If there is a block of preprocessor directives, it can get difficult to follow the logic of the #includes or #defines. In these cases, the directives may be indented logically, according to the structure of the #if blocks. There should never be white space between the "#" and the directive.

#ifdef DEBUG  
 #define Debug\_log printf

#include <hdr1.h>  
#else  
 #define Debug\_log

#include <hdr2.h>  
#endif /\* DEBUG \*/

**5.1.4 [OPTIONAL] Do not add relative path names in #include statements.**

There is no guarantee that the path would be the same on all machines on which the code is compiled. Hence, do not add relative path names in #include statements if path names for the include directory should be mentioned as a command line parameter to the compiler.

|  |  |
| --- | --- |
| **Not recommended** | **Recommended** |
| #include “..\..\data.h” | #include “data.h” |

**5.1.5 [OPTIONAL] Annotate the "endif" directive with the condition check of its matching "if\*" directive.**

This usually causes a harmless warning during preprocessing, but the readability is enhanced. Especially, when many such directives are used in the same file.

**Example:**

#ifdef DEBUG  
#define Debug\_log printf   
#else  
#define Debug\_log  
#endif /\* DEBUG \*/

**5.1.6 [OPTIONAL] Avoid too many "ifdef DEBUG"s in the source code.**

Too may “ifdef DEBUG”s can reduce the readability. It is recommended to isolate all debugging related functionality into a separate module and use the interfaces provided by it by conditionally compiling the source into the main program (for example,, debug\_printf()).

## 5.2 # define

**5.2.1 [OPTIOINAL] The use of “#define” preprocessor is highly discouraged. Use const, enum, or inline functions where appropriate. Only use “#define” preprocessor directive if there is no other alternative.**

The size of a constant declared with #define is platform dependent (32-bit or 64-bit), but is not obvious from the declaration. Using const will make the dependency explicit.

**5.2.2 [MANDATORY] “#define” preprocessor directives must be formatted as follows: #define Symbol Value**

* The “#” is aligned to the left.
* There is no space between the “#” and “define”.
* The “#define” keyword is separated from the symbol to be defined by a space
* The definition for the symbol is separated from the symbol by one or more tabs. These definitions should be lined up within a block of related definitions.

**5.2.3 [MANDATORY] Use inline functions in place of macros. However, inline functions should never be used for functions with more than few lines.**

Inline functions provide a better way of declaring local variables and returning values to the caller. Achieving the same with macros would require some trickery and may even involve using specific pre-processor extensions. Inline functions also avoid some pitfalls with macros, involving parenthesizing parameters and conditionals.

**5.2.4 [MANDATORY] Macros and macro parameters must be fully parenthesized.**

Unexpected precedence problems may arise while using the expression in the Macros. Therefore, every reference to the parameter should be fully parenthesized.

|  |  |
| --- | --- |
| **Not Recommended** | **Recommended** |
| #define max(a,b) a < b ? b : a | #define max(a,b) ((a) < (b) ? (b) : (a)) |

**5.2.5 [OPTIONAL] Do not use macros in include statements.**

* Including the header files with “#define” predefined makes a code hard to read and understand.

|  |
| --- |
| **Not Recommended** |
| #define A(string) #string  #define HEADER\_FILE1 A(stdio.h)    #include HEADER\_FILE1    #define HEADER\_FILE2() A(string.h)    #include HEADER\_FILE2() |

**5.2.6 [OPTIONAL] Limit the length of macro to 10 lines.**

**5.2.7 [OPTIONAL] Limit the number of parameters in a macro to 5.**

## 5.3 #if / #elif / #else / #endif

**5.3.1 [MANDATORY] Format #if et al. statements as follows:**

#if Condition1  
...  
#elif Condition2 /\* if Condition1 \*/  
...  
#else /\* if Condition1 elif Condition2 \*/  
...  
#endif /\* if Condition1 elif Condition2 else \*/

* All "#" are aligned to the left, unless it is in a block of preprocessor directives as mentioned above.
* There is no space between the '#" and the "if", "ifdef", "ifndef", "elif", "else", or "endif".
* The preprocessor keyword is separated from any expression clause by one space.

**5.3.2 Commenting #if et al. statements**

**5.3.2.1 [OPTIONAL] Comments are separated from their #if element by one tab or space**

**5.3.2.2 [OPTIONAL] Do not include “#”characters in the comments for #if et al. statements**

Some editors (notably the Visual C editor) sometimes have problems parsing multiple "#"s in one line, so, do not use them.

**5.3.3 [MANDATORY] If a function is enclosed in #if...#endif block, then include the related function header comments also, inside the same #if...#endif block**

For better readability, functions that are entirely within a #if…#endif clause need to have the function comment also within the same scope, that is, the #if should be put before the function comment header.

**5.3.4 [OPTIONAL] Only one function may be enclosed in each #if**

If a group of functions are enclosed in #if, it can be difficult to determine that a function in the middle of the group is indeed inside the #if.

**5.3.5 [OPTIONAL] Minimize the use of #if...#else...#endif blocks inside functions. Instead hide the complexity in macros declared outside the function.**

The complexity of conditional compilation can be kept out of functions for the sake of readability and simplicity. The block that needs to be compiled conditionally can be declared as a macro or inline function and this can be called unconditional from the function.

**5.4 #ifndef**

**5.4.1 [OPTIONAL] Format #ifndef/#define pair as follows : "File name in ALL CAPS"\_H\_ (the macro name is the filename in ALL CAPS, followed by an underscore and the file extension in all uppercase ("H"), followed by an underscore).**

#ifndef EXAMPLE\_H\_

...

...

#endif

# 6. Programming/ Coding Standards

## 6.1 Pointers

**6.1.1 [OPTIONAL] Do not cast a long pointer to an integer pointer as this might result in loss of data.**

**6.1.2 [OPTIONAL] Do not cast an integer pointer to a long pointer as this might result in loss of data.**

## 6.2 Functions & Methods

### 6.2.1 Metrics

**6.2.1.1 [OPTIONAL] Limit the size of the function block to 50 lines.**

It is easy to glance up the declaration without shuffling through paper or screens.

**6.2.1.2 [OPTIONAL] Limit the number of parameters in a function to 5 or less.**

This will reduce the amount of coupling between functions. If more parameters are needed, a structure should be used to hold related data and a pointer should be passed.

**6.2.1.3 [OPTIONAL] Limit the number of blocks in a function to 10.**

This will make errors more obvious and function testing will be easier. Blocks in the function refers to blocks related to if/if-else/ for/ do-while/ switch statements.

**6.2.1.4 [OPTIONAL] Limit the number of function calls within a function to 10.**

**6.2.1.5 [OPTIONAL] Do not use excessive block nesting depth (more than 5 levels).**

This will make code easier to understand and improve maintainability. Give a single function a single cohesive responsibility in order to avoid excessively long functions and nested code blocks

### 6.2.2 Design Guidelines

**6.2.2.1 [MANDATORY] Use void when a function has no argument or when it returns no values.**

**.**

In many compiler implementations, functions which do not declare return type are automatically assigned return type int. This conflicts with semantics of function implementation. Explicitly specifying parameter and return types as void would clearly convey the function intent.

**Example:**

|  |  |
| --- | --- |
| **Not Recommended** | **Recommended** |
| #include <stdio.h>  func1() { //NOT OK  ....  }  void func2(){ //NOT OK  ...  }  func3(void) { //NOT OK  ...  } | #include <stdio.h>  void func1(void) { //OK  ..  }  void func2(void) { //OK  ..  }  void func3(void) { //OK  ..  } |

**6.2.2.2 [OPTIONAL] Use a typedef when declaring function pointers.**

This simplifies the code and improve the readability.

**Example:**

|  |  |
| --- | --- |
| **Not Recommended** | **Recommended** |
| void (\*p)(); | typedef void (\*p)(); |

## 6.3 Statements & Expressions

### 6.3.1 If

**6.3.1.1 [MANDATORY] Do not check floats for equality; instead, check for less than or greater than.**

Checking floats for equality makes the code more susceptible to errors. The == operator checks for exact equality of the two variables. However, many values can not be accurately represented in binary floating point. This results in unexpected results for the comparison. Hence, check for less than or greater than.

**Example:**

|  |  |
| --- | --- |
| **Not Recommended** | **Recommended** |
| void func( float a, int b)  {  If(a==b){ }  } | void func( float a, int b)  {  If(a>=b){ }  } |

**6.3.1.2 [OPTIONAL] Avoid using negation in if else conditions.**

**Example:**

|  |  |
| --- | --- |
| **Not Recommended** | **Recommended** |
| if ( ! isalpha(argv[0]) ) {  cout << "Not a letter";  } else {  cout << "A letter";  } | if ( isalpha(argv[0]) ) {  cout << "A letter";  } else {  cout << "Not a letter";  } |

### 6.3.2 For

**6.3.2.1 [OPTIONAL] Do not use a for loop without a condition.**

**Example:**

|  |  |
| --- | --- |
| **Not Recommended** | **Recommended** |
| for ( int i = 0; ; i++ ) {  if ( i < argc ) {  break;  } } | for ( int i = 0; i < argc; i++ ) {    } |

**6.3.2.2 [OPTIONAL] Using Pre-increment and Pre-decrement is more efficient when returned result is unused.**

**Example:**

|  |  |
| --- | --- |
| **Not Recommended** | **Recommended** |
| int main() {  for(int j =0; j<10; j++){  }  printf("%i", array[j]);  return 0;  } | int main() {  for(int j =0; j<10; ++j){  }  printf("%i", array[j]);  return 0;  } |

**6.3.2.3 [OPTIONAL] Do not assign to loop control variables in the body of a for-loop.**

Move the assignment-to-control variable from inside of the for loop to the increment clause of the loop.

**Example:**

|  |  |
| --- | --- |
| **Not Recommended** | **Recommended** |
| void main( int argc, char\* argv[]) {  int i;  for ( i = 0; i < argc; i++ )  {  i = i + 1;  }  } | void main( int argc, char\* argv[]) {  int i;  for ( i = 0; i < argc; i = i + 2)  {  }  } |

**6.3.2.4 [OPTIONAL] Do not use a for statement without initialization and an increment counter.**

**Example:**

|  |  |
| --- | --- |
| **Not Recommended** | **Recommended** |
| void main( int argc, char\* argv[]) {  int i = 0;  for ( ; i < argc ; ) {  i++;  }  } | void main( int argc, char\* argv[]) {  int i = 0;  while ( i < argc ) {  i++;  }  } |

**6.3.2.5 [OPTIONAL] Do not use breaks in for loops.**

This makes the code easier to understand.

### 6.3.3 Switch

**6.3.3.1 [OPTIONAL] Do not use a switch statement to represent a value that is effectively Boolean.**

This may lead to errors in code.

**Example:**

|  |
| --- |
| **Not recommended** |
| switch(data1 > data2 )  {  case 0: break;  default:;  } |

**6.3.3.2 [OPTIONAL] Use if-else instead of switch statements with few branches**

**Example:**

|  |  |
| --- | --- |
| **Not Recommended** | **Recommended** |
| void main( int argc, char\* argv[]) {   switch ( argc ) {   case 0:  printf(“No Parameters”);  break ;   default :  printf("Parameters: ");  }  } | void main( int argc, char\* argv[]) {   if ( argc == 0 ) {   printf(“No Parameters”);  }  else {  printf(“Paramaters”);  }  } |

**6.3.3.3 [OPTIONAL] Avoid use of switch statements with many cases.**

### 6.3.4 While:

**6.3.4.1 [OPTIONAL] Avoid making assignments in conditional statements.**

**Example:**

|  |  |
| --- | --- |
| **Not Recommended** | **Recommended** |
| void main( int argc, char\* argv[]) {  while( a = getNum() ) {  printf("A is still 5");  }  } | void main( int argc, char\* argv[]) {  int a = 5;  int b = 0;  b = getNum();  while( a == b ) {  printf("A is still 5");  }  } |

### 6.3.5 Others:

**6.3.5.1 [MANDATORY] Do not convert constants to non-constants.**

Converting constants to non-constants will undermine the data integrity by allowing modification of values which are assumed to be constant. This will also cause confusion and reduce readability since constant variables cannot be relied on to remain constant.

**Example:**

|  |  |
| --- | --- |
| **Not Recommended** | **Recommended** |
| void f(const int\* x)  {  int\* y;  y = (int\*)x;  } | void f(const int\* x)  {  const int\*z;  z = (const int\*)x;  } |

**6.3.5.2 [OPTIONAL] Limit the cyclomatic complexity to less than 30.**

Cyclomatic complexity is calculated according to the formula:

CC = Number of decisions + 1.

By decision we mean every occurrence of:

-if

-for

-while

-do...while

-switch

-case

-catch

-conditional expression (a ? b : c)

-logical operator (&& and ||)

More the cyclomatic complexity more error-prone is the code.

**6.3.5.3 [OPTIONAL] Do not use magic numbers.**

Use of literal constants in code makes it difficult to understand and maintain the code.

Use symbolic values instead.

**Example:**

|  |  |
| --- | --- |
| **Not Recommended** | **Recommended** |
| void bol(int);  int fun( int a[], int b ) {  int e;  int f = 0;  int i = 5;  a[i] = 5;  e = bol(8);  return (e);  } | const int MIN = 8;  void bol(int);  int fun( int a[], int b ) {  int e;  int f = 0;  int i = MIN;  a[i] = MIN;  e = bol(MIN);  return (e);  } |

**6.3.5.5 [OPTIONAL] Avoid use of continue statements.**

Avoid using continue statements as they make the code difficult to read.

## 6.4 Security

**6.4.1 [MANDATORY] Do not use gets(), use fgets() instead.**

Gets() will continue to store characters past the end of the buffer and can be used to break computer security. Hence, it is extremely dangerous to use. Use fgets() instead.

**Example:**

|  |  |
| --- | --- |
| **Not Recommended** | **Recommended** |
| #include <stdio.h>    void main()  {  char line[100];    printf( "Input a string: " );  gets( line );  printf( "The line entered was: %s",  line );  } | #include <stdio.h>    void main()  {  char line[100];    printf( "Input a string: " );  fgets( line, 100, stdin );  printf( "The line entered was:%s",  line );  } |

**6.4.3 [OPTIONAL] Do not use unsafe string functions as they can cause buffer overflow.**

Unsafe string functions are strcpy, strcat, sprintf, vsprintf, and gets. These are dangerous functions as they do not check the bounds. Instead, use strncpy, strncat, snprintf, and fgets respectively.

# 7. Quick Reference

* Files should have a file header
* Header files should stand alone.
* A header file should not contain more #include directives than the need to be self contained.
* Header files should contain only declarations, not definitions.
* Header files should be able to be compiled at Warning Level 4 without any warnings.
* The #define for the filename should appear only in the header file itself.
* All header files should include protection from double inclusion.
* Place public (inter-module) interfaces and private (intra-module) interfaces in separate files.
* Do not declare external data in a source file.
* Global variables should not be used unless absolutely necessary.
* Global variable declarations should always start in the first column.
* Variables must be initialized while defining them and these definitions should be on separate lines.
* Variables declaration blocks are separated from other code by one blank line.
* Variables should not be declared with the same name in nested scopes.
* Declare each variable in a separate declaration statement.
* Never use enumerated types should.
* It is strongly discouraged to declare a named structure inside another structure and then use the name of the nested structure in another location.
* Declare a structure with a name as that it can be used in a forward declaration.
* Fields within a structure must be 8-byte aligned.
* Do not declare structs or unions with more than 20 fields.
* It is recommended to define the union with a name so that it can be used in a forward declaration.
* Use typedefs for datatypes. Do not use typedefs for pointers to datatypes.
* For function declarations, write the leading parenthesis and the first argument on the same line as the function name.
* The storage class designation is not required for declaration of non-static functions.
* Declare a prototype for all static functions and the do the declaration before the first function in the source file.
* The return type is required even if the function does not return a value.
* A function declaration must include the name of the argument in the prototype. If a function doesn’t have an argument, then void must be specified.
* The close parenthesis for the argument list is followed immediately by a semicolon (;) without a space.
* Functions may not take enumerated types as parameters.
* Declare each variable in a separate declaration statement.
* Do not declare the array size when the array is initialized.
* Do not declare the size of the array in the function prototype/definition when it is used as a parameter in a function call.
* Initialize all pointer variables.
* Filenames should only contain roman alphabetics, numbers and underscores in the format: [a-z][0-9 a-z A-Z \_\.]\* .
* Do not use spaces or non-platform portable characters such as bullets and greek characters.
* The length of a filename should not exceed 31 characters.
* As far as possible, do not use numbers in filenames. Numbers should only be used when they say something meaningful about the function contained in the file.
* Use underscores to separate the words in filenames.
* Identifiers should use underscores to delineate words within the identifier.
* Render identifier names mixed-case or all lowercase. If the mixed-case is being used,then the first character should be lower case. If the lowercase is being used words should be delineated using underscores.
* Names must be composed of American English words.
* Negative variables are discouraged as they can introduce confusions .
* Use variables which are not complete words with caution.
* All preprocessor identifiers that do not take parameters should be all uppercase.
* Identifiers should only contain letters from the roman alphabet, underscores, and numbers [a-zA-Z][a-zA-Z0-9\_]\*.
* The use of numbers in an identifier should be limited to a case where the number indicates something significant about the contents or purpose of the identifier or where multiple items otherwise have nothing to distinguish them from each other.
* Do not start identifiers with an underscore ("\_") or a double underscore ("\_\_").
* Use of Scope Decorators is encouraged for certain category of identifiers.
* Render non-simple type names, anything other than a redefinition of an intrinsic type (that is,. int16 et al.) either in mixed-case or all lower-case.
* Render function names in mixed-case or all lower-case. If using mixed-case, the first character should be uppercase. If using lowercase, words should be delineated using underscores.
* Prepend pointer variables by a ‘p’ or a ‘lp’ in most cases.
* Prepend global variables, if any, with a ‘the’
* Static variables may be prepended with ‘s’
* Prepend the structure tags with ‘S’.
* Prepend array variables and parameters with a ‘rg’
* Comments are not a substitute for clearly written code
* Comments must be indented at the same level as the code block that they describe.
* Comments must be provided for each non-trivial variable declaration, or any variable which is not self-commenting.
* Comments should be provided for each field or member of a structure.
* Do not use C++ style comments in C code.
* Offset comments for a single line of code by at least one tab from the end of the code line.
* For multi-line comments do not include a “TODO” on any line after the first one.
* Format time sensitive special comments with a three dollar signs ("$$$").
* Use the HACK special comment if you are placing a “hack” in a file.
* Use the BUGFIX special comment to identify a bug fix.
* Use the Bug fix comments sparingly. Make the fix details to be a part of the check-in logs.
* Delete old, unused code.  Check in code that contains commented out blocks of code only when absolutely necessary.
* Always put a space between binary arithmetic, logical and bitwise operators and operands.
* The use of parenthesis around a basic expression is discouraged.
* Always place left parenthesis directly after the function name.
* Always use parenthesis with the the “return” and “sizeof” statements.
* Always place a maximum of one ASCII space character following the opening parenthesis in conditional statements.
* Always use a single ASCII space between conditional keyword and its opening parenthesis.
* Line breaking must be consistent on a file level.
* Resrict lines to 79 printable characters in length.
* Unary arithmetic operator is never separated from its operand
* Unary logical, deference, and bit operator is never separated from its operand
* Binary, bi,t and logical arithmetic operator is always separated from its operand.
* Binary dereference operator is never separated from its operand
* Bitfield Declaration Operator is never separated from the scope or the identifier
* Scope Resolution Operator :: is always separated from its operands by a space.
* Close non trivial sub-expressions in parentheses
* Conditional ternary operator ?: always separated from its operands by a space.
* Only use labels as targets for goto statements.
* Avoid the “goto” statement as far as possible.
* Labels are aligned at column 1 (i.e. left aligned).
* Minimize the number of return statements in a function.
* Break lines broken within the *Condition* clause of the conditional statement wherever possible, at the highest-level logical conjunctive ("&&"or "||") or comparison operator ("==", ">", etc.).
* Always use curly braces (“{}”) for if-else statement.
* If all of the *Condition* clauses are all constants, consider using a switch statement.
* Always include a "default" case in a "switch" statement.
* The “default” case in the switch statement should be the last case.
* Do not break lines in switch *Expression* clause.
* There should be no white spaces between the “return” or “sizeof” statements and its opening parenthesis and surrounding “return” or “sizeof” statements argument or expression.
* The preprocessor directive is separated from any arguments by a space
* Within the code, align the ‘#’ of all preprocessor directives in the left most column.
* Within a block of preprocessor directives, indent the "#" logically.
* Do not add relative path names in #include statements.
* The use of “#define” preprocessor is highly discouraged. Use const, enum, or inline functions where appropriate.
* Replace Macros with inline functions.
* Do not use macros in include statements.
* Macros and macro parameters must be fully parenthesized.
* Limit the length of macro to 10 lines.
* Limit the number of parameters in a macro to 5.
* Comments are separated from their #if element by one tab or space
* Comments include all the related #if conditions without the "#"characters
* If an entire function is being enclosed in an #if...#endif block, include function header comment inside the #if...#endif block
* Only one function may be enclosed in each #if
* Minimize the use of #if...#else...#endif blocks inside functions. Instead, hide the complexity in macros declared, outside the function.
* Do not cast a pointer to an int pointer as this might result in loss of data.
* Do not cast an int pointer to a long pointer as this might result in loss of data.
* Limit the size of the function block to 50 lines.
* Limit the number of parameters in a function to 5 or less.
* Limit the number of blocks in a function to 10.
* Limit the number of function calls within a function to 10.
* Do not use excessive block nesting depth (more than 5 levels).
* Use void when a function is passed or returns no values.
* Use a typedef when declaring function pointers.
* Avoid using negation in if else conditions.
* Do not check floats for equality; instead, check for less than or greater than.
* Do not use a for loop without a condition.
* Using Pre-increment and Pre-decrement is more efficient when returned result unused.
* Do not assign to loop control variables in the body of for-loop.
* Do not use a for statement without initialization and an increment counter.
* Do not use breaks in for loops.
* Do not use a switch statement to represent a value that is effectively Boolean.
* Use if-else instead of switch statements with few branches
* Avoid the use of switch statements with many cases.
* Avoid making assignments in conditional statements.
* Do not convert the constants to non-constants.
* Limit the cyclomatic complexity to less than 30.
* Do not use magic numbers.
* Do not use continue statements.
* Do not use gets(), use fgets() instead.
* Do not use unsafe string functions as they can cause buffer overflow.