***C* Coding Standard**

Adapted from <http://www.possibility.com/Cpp/CppCodingStandard.html>and NetBSD's style guidelines

For the C++ coding standards click here

**Function Names**

*Usually every function performs an action, so the name should make clear what it does:* ***CheckForErrors()*** *instead of* ***ErrorCheck(****),* ***DumpDataToFile()*** *instead of* ***DataFile().*** *This will also make functions and data objects more* distinguishable.

**Structs** are often nouns. By making function names verbs and following other naming conventions programs can be read more naturally.

**Suffixes are sometimes useful:**

**max** - to mean the maximum value something can have.

**cnt** - the current count of a running count variable.

**key** - key value.

For example: retry\_max to mean the maximum number of retries, retry\_cnt to mean the current retry count.

**Pefixes** are sometimes useful:

**is** - to ask a question about something. Whenever someone sees Is they will know it's a question.

**get** - get a value.

**set** - set a value**.**

For example:isHitRetryLimit.

**Include Units in Names**

If a variable represents time, weight, or some other unit then include the unit in the name so developers can more easily spot problems. For example:

uint32 timeout\_msecs;

uint32 my\_weight\_lbs;

**Structure Names**

When declaring variables in structures, declare them organized by use in a manner to attempt to minimize memory wastage because of compiler alignment issues, then by size, and then by alphabetical order. E.g, don't use ``int a; char \*b; int c; char \*d''; use ``int a; int b; char \*c; char \*d''. Each variable gets its own type and line. Major structures should be declared at the top of the file in which they are used, or in separate header files, if they are used in multiple source files. Use of the structures should be by separate declarations.

**Example**

struct foo {

int Abc;

int Bde;

char Abcde;

char Bcdedg;

};

struct foo \*foohead; /\* Head of global foo list \*/

**Justification**

With this approach the scope of the variable is clear in the code.

Now all variables look different and are identifiable in the code.

**Pointer Variables**

Pplace the \* close to the variable name not pointer type

**Example**

char \*Name= NULL;

char \*Name, Address;

**Global Variables**

Global variables should be prepended with “global”.

Global variables should be avoided whenever possible.

**Example**

Int globalLogger; // Global vairiable

Int Logger; // Local variable

**Global Constants**

Global constants should be all caps with '\_' separators.

**Example**

const int A\_GLOBAL\_CONSTANT= 5;

**#define and Macro Names**

Put #defines and macros in all upper using '\_' separators. Macros are capitalized, parenthesized, and should avoid side-effects. Spacing before and after the macro name may be any whitespace, though use of TABs should be consistent through a file. If they are an inline expansion of a function, the function is defined all in lowercase, the macro has the same name all in uppercase. If the macro is an expression, wrap the expression in parenthesis. If the macro is more than a single statement, use ``do { ... } while (0)'', so that a trailing semicolon works. Right-justify the backslashes; it makes it easier to read.

**Justification**

This makes it very clear that the value is not alterable and in the case of macros, makes it clear that you are using a construct that requires care.

Some subtle errors can occur when macro names and enum labels use the same name.

**Example**

#define MAX(a,b) blah

#define IS\_ERR(err) blah

#define MACRO(v, w, x, y) \ do

\ { v = (x) + (y);

\ w = (y) + 2;

\ )

\ while (0);

**Enum Names**

Labels All Upper Case with '\_' Word Separators

This is the standard rule for enum labels. No comma on the last element. Example

Enum PinStateType

(

PIN\_OFF,

PIN\_ON

};

**Make a Label for an Error State**

It's often useful to be able to say an enum is not in any of its *valid* states. Make a label for an uninitialized or error state. Make it the first label if possible.

Example

enum

{

STATE\_ERR,

STATE\_OPEN,

STATE\_RUNNING,

STATE\_DYING

};

**Formatting**

**Brace Placement**

Of the three major brace placement strategies one is recommended:

if (condition)

{

while (condition)

{

...

...

}

}

**When Braces are Needed**

All if, while and do statements must either have braces or be on a single line.

**Always Uses Braces Form**

All if, while and do statements require braces even if there is only a single statement within the braces. For example:

if (1 == somevalue)

{

somevalue = 2;

}

**Justification**

It ensures that when someone adds a line of code later there are already braces and they don't forget. It provides a more consistent look. This doesn't affect execution speed. It's easy to do.

**One Line Form**

if (1 == somevalue) somevalue = 2;

**Justification**

It provides safety when adding new lines while maintainng a compact readable form.

**Add Comments to Closing Braces**

Adding a comment to closing braces can help when you are reading code because you don't have to find the begin brace to know what is going on.

while(1)

{

if (valid)

{

} /\* if valid \*/

else

{

} /\* not valid \*/

} /\* end forever \*/

**Consider Screen Size Limits**

Some people like blocks to fit within a common screen size so scrolling is not necessary when reading code.

**A Line Should Not Exceed 78 Characters**

**Parens *()* with Key Words and Functions Policy**

 Do not put parens next to keywords. Put a space between.

 Do put parens next to function names.

 Do not use parens in return statements when it's not necessary.

**Justification**

Keywords are not functions. By putting parens next to keywords keywords and function names are made to look alike.

**Example**

if (condition)

{

}

while (condition)

{

}

strcpy(s, s1);

return 1;

***If Then Else* Formatting**

**Layout**

It's up to the programmer. Different bracing styles will yield slightly different looks. One common approach is:

if (condition)

{

}

else if (condition)

{

}

else

{

}

If you have *else if* statements then it is usually a good idea to always have an else block for finding unhandled cases. Maybe put a log message in the else even if there is no corrective action taken.

**Condition Format**

Always put the constant on the left hand side of an equality/inequality comparison. For example:

if ( 6 == errorNum ) ...

One reason is that if you leave out one of the = signs, the compiler will find the error for you. A second reason is that it puts the value you are looking for right up front where you can find it instead of buried at the end of your expression. It takes a little time to get used to this format, but then it really gets useful.

***switch* Formatting**

Falling through a case statement into the next case statement shall be permitted as long as a comment is included.

If you need to create variables put all the code in a block.

Example

switch (...)

{

case 1: /\* comments \*/

...

case 2:

{

int v;

...

}

break;

default:

}

**Continue and Break**

Continue and break are really disguised gotos so they are covered here.

Continue and break like goto should be used sparingly as they are magic in code. With a simple spell the reader is beamed to god knows where for some usually undocumented reason.

The two main problems with continue are:

It may bypass the test condition

It may bypass the increment/decrement expression

Consider the following example where both problems occur:

while (TRUE)

{

...

/\* A lot of code \*/

...

if (/\* some condition \*/)

{

continue;

}

...

/\* A lot of code \*/

...

if ( i++ > STOP\_VALUE) break;

}

Note: "A lot of code" is necessary in order that the problem cannot be caught easily by the programmer.

From the above example, a further rule may be given: Mixing continue with break in the same loop is a sure way to disaster.

?:

The trouble is people usually try and stuff too much code in between the *?* and *:*. Here are a couple of clarity rules to follow:

Put the condition in parens so as to set it off from other code,If possible, the actions for the test should be simple functions.

Put the action for the then and else statement on a separate line unless it can be clearly put on one line.

**Example**

(condition) ? funct1() : func2();

or

(condition)

? long statement

: another long statement;

**One Statement Per Line**

There should be only one statement per line unless the statements are very closely related. The reasons are:

The code is easier to read. Use some white space too. Nothing better than to read code that is one line after another with no white space or comments.

**One Variable Per Line**

Related to this is always define one variable per line:

Not:

char \*\*a, \*x;

Do:

char \*\*a = 0; /\* add doc \*/

char \*x = 0; /\* add doc \*/

The reasons are:

1. Documentation can be added for the variable on the line.

2. It's clear that the variables are initialized.

3. Declarations are clear which reduces the probability of declaring a pointer when you meant to declare just a char.

**Replace Macros with Inline Functions**

In C macros are not needed for code efficiency. Use inlines. However, macros for small functions are ok.

**Example**

#define MAX(x,y) (((x) > (y) ? (x) : (y))// Get the maximum

The macro above can be replaced for integers with the following inline function with no loss of efficiency:

inline int

max(int x, int y)

{

return (x > y ? x : y);

**}**

**Be Careful of Side Effects**

Macros should be used with caution because of the potential for error when invoked with an expression that has side effects.

**Example**

MAX(f(x),z++);

**Always Wrap the Expression in Parenthesis**

When putting expressions in macros always wrap the expression in parenthesis to avoid potential communitive operation abiguity.

**Example**

#define ADD(x,y) x + y

**must be written as**

#define ADD(x,y) ((x) + (y))

**Make Macro Names Unique**

Like global variables macros can conflict with macros from other packages.

1. Prepend macro names with package names.

2. Avoid simple and common names like MAX and MIN.

**Initialize all Variables**

You shall always initialize variables. Always. Every time. gcc with the flag -W may catch operations on uninitialized variables, but it may also not.

**Justification**

More problems than you can believe are eventually traced back to a pointer or variable left uninitialized.

**Short Functions**

Functions should limit themselves to a single page of code.

**Justification**

The idea is that the each method represents a technique for achieving a single objective.

Most arguments of inefficiency turn out to be false in the long run.

True function calls are slower than not, but there needs to a thought out decision (see premature optimization).

**Document Null Statements**

Always document a null body for a for or while statement so that it is clear that the null body is intentional and not missing code.

while (\*dest++ = \*src++)

{

}

**Do Not Default If Test to Non-Zero**

Do not default the test for non-zero, i.e.

if (FAIL != f())

is better than

if (f())

Even though FAIL may have the value 0 which C considers to be false. An explicit test will help you out later when somebody decides that a failure return should be -1 instead of 0. Explicit comparison should be used even if the comparison value will never change; e.g., if (!(bufsize % sizeof(int))) should be written instead as if ((bufsize % sizeof(int)) == 0) to reflect the numeric (not boolean) nature of the test. A frequent trouble spot is using strcmp to test for string equality, where the result should *never ever* be defaulted. The preferred approach is to define a macro *STREQ*.

#define STREQ(a, b) (strcmp((a), (b)) == 0)

Or better yet use an inline method:

inline bool

string\_equal(char\* a, char\* b)

{

strcmp(a, b) == 0) ? return true : return false;

/\* Or more compactly: \*/

return (strcmp(a, b) == 0);

}

Note, this is just an example, you should really use the standard library string type for doing the comparison.

The non-zero test is often defaulted for predicates and other functions or expressions which meet the following restrictions:

 Returns 0 for false, nothing else.

 Is named so that the meaning of (say) a true return is absolutely obvious. Call a predicate is\_valid(), not check\_valid().

**Usually Avoid Embedded Assignments**

There is a time and a place for embedded assignment statements. In some constructs there is no better way to accomplish the results without making the code bulkier and less readable.

while (EOF != (c = getchar()))

{

process the character

}

The ++ and -- operators count as assignment statements. So, for many purposes, do functions with side effects. Using embedded assignment statements to improve run-time performance is also possible. However, one should consider the tradeoff between increased speed and decreased maintainability that results when embedded assignments are used in artificial places. For example,

a = b + c;

d = a + r;

should not be replaced by

d = (a = b + c) + r;

even though the latter may save one cycle. In the long run the time difference between the two will decrease as the optimizer gains maturity, while the difference in ease of maintenance will increase as the human memory of what's going on in the latter piece of code begins to fade.

**Documentation**

**Comments Should Tell a Story**

Consider your comments a story describing the system. Expect your comments to be extracted by a robot and formed into a man page. Class comments are one part of the story, method signature comments are another part of the story, method arguments another part, and method implementation yet another part. All these parts should weave together and inform someone else at another point of time just exactly what you did and why.

**Document Decisions**

Comments should document decisions. At every point where you had a choice of what to do place a comment describing which choice you made and why. Archaeologists will find this the most useful information.

**Use Headers**

Use a document extraction system like Doxygen.

These headers are structured in such a way as they can be parsed and extracted. They are not useless like normal headers. So take time to fill them out. If you do it right once no more documentation may be necessary.

**Comment Layout**

Each part of the project has a specific comment layout. Doxygen has the recommended format for the comment layouts.

**Make Gotchas Explicit**

Explicitly comment variables changed out of the normal control flow or other code likely to break during maintenance. Embedded keywords are used to point out issues and potential problems. Consider a robot will parse your comments looking for keywords, stripping them out, and making a report so people can make a special effort where needed.

**Gotcha Keywords**

@author:

specifies the author of the module

@version:

specifies the version of the module

@param:

specifies a parameter into a function

@return:

specifies what a function returns

@deprecated:

says that a function is not to be used anymore

@see:

creates a link in the documentation to the file/function/variable to consult to get a better understanding on what the current block of code does.

@todo:

what remains to be done

@bug:

report a bug found in the piece of code

**Gotcha Formatting**

Make the gotcha keyword the first symbol in the comment.

Comments may consist of multiple lines, but the first line should be a self-containing, meaningful summary.

The writer's name and the date of the remark should be part of the comment. This information is in the source repository, but it can take a quite a while to find out when

and by whom it was added. Often gotchas stick around longer than they should. Embedding date information allows other programmer to make this decision.

Embedding who information lets us know who to ask.

**Commenting function declarations**

Functions headers should be in the file where they are declared. This means that most likely the functions will have a header in the .h file. However, functions like main() with no explicit prototype declaration in the .h file, should have a header in the .c file.

**Include Statement Documentation**

Include statements should be documented, telling the user why a particular file was included.

/\*

\* Kernel include files come first.

\*/

/\* Non-local includes in brackets. \*/

/\*

\* If it's a network program, put the network include files next.

\* Group the includes files by subdirectory.

\*/

/\*

\* Then there's a blank line, followed by the /usr include files.

\* The /usr include files should be sorted!

\*/

**No Magic Numbers**

A magic number is a bare naked number used in source code. It's magic because no-one has a clue what it means including the author inside 3 months. For example:

if (22 == foo) { start\_thermo\_nuclear\_war(); }

else if (19 == foo) { refund\_lotso\_money(); }

else if (16 == foo) { infinite\_loop(); }

else { cry\_cause\_im\_lost(); }

In the above example what do 22 and 19 mean? If there was a number change or the numbers were just plain wrong how would you know? Instead of magic numbers use a real name that means something. You can use *#define* or constants or enums as names. Which one is a design

choice. For example:

#define PRESIDENT\_WENT\_CRAZY (22)

const int WE\_GOOFED= 19;

enum {

THEY\_DIDNT\_PAY= 16

};

if (PRESIDENT\_WENT\_CRAZY == foo) { start\_thermo\_nuclear\_war(); }

else if (WE\_GOOFED == foo) { refund\_lotso\_money(); }

else if (THEY\_DIDNT\_PAY == foo) { infinite\_loop(); }

else { happy\_days\_i\_know\_why\_im\_here(); }

Now isn't that better? The const and enum options are preferable because when debugging the debugger has enough information to display both the value and the label. The #define option just shows up as a number in the debugger which is very inconvenient. The const option has the downside of allocating memory. Only you know if this matters for your application.

**Error Return Check Policy**

Check every system call for an error return, unless you know you wish to ignore errors.

For example, *printf* returns an error code but rarely would you check for its return code. In which case you can cast the return to (void) if you really care.

Include the system error text for every system error message.

Check every call to malloc or realloc unless you know your versions of these calls do the right thing. You might want to have your own wrapper for these calls, including new, so

you can do the right thing always and developers don't have to make memory checks everywhere.