C Coding Standard

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

For the C++ coding standards click here

Contents

Names

Make Names Fit

Names are the heart of programming. In the past people believed knowing someone's true name gave them magical power over that person. If you can think up the true name for something, you give yourself and the people coming after power over the code. Don't laugh!

A name is the result of a long deep thought process about the ecology it lives in. Only a programmer who understands the system as a whole can create a name that "fits" with the system. If the name is appropriate everything fits together naturally, relationships are clear, meaning is derivable, and reasoning from common human expectations works as expected.

If you find all your names could be Thing and DoIt then you should probably revisit your design.

Function Names

- Usually every function performs an action, so the name should make clear what it does: check_for_errors() instead of error_check(), dump_data_to_file() instead of data_file(). 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.

- Prefixes 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: is_hit_retry_limit.

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

- Use underbars ('_') to separate name components
- 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, although an exception can be made when declaring bitfields (to clarify that it's part of the one bitfield). Note that the use of bitfields in general is discouraged. 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 and should be "extern" if they are declared in a header file. It may be useful to use a meaningful prefix for each member name. E.g, for ``struct softc'' the prefix could be ``sc_''.

Example

```
struct foo {
        struct foo *next;       /* List of active foo */
        struct mumble amumble;  /* Comment for mumble */
        int bar;
        unsigned int baz:1,     /* Bitfield; line up entries if desired */
                     fuz:5,
                     zap:2;
        uint8_t flag;
};
struct foo *foohead;            /* Head of global foo list */
```

Variable Names on the Stack

- use all lower case letters
- use '_' as the word separator.

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.

Example

```
int handle_error (int error_number) {
   int            error= OsErr();
   Time           time_of_error;
   ErrorProcessor error_processor;
}
```

Pointer Variables

- place the *close to the variable name not pointer type

Example

```
char *name= NULL;

char *name, address;
```

Global Variables

- Global variables should be prepended with a 'g_'.
- Global variables should be avoided whenever possible.

Justification

- It's important to know the scope of a variable.

Example

```
Logger  g_log;
Logger* g_plog;
```

Global Constants

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

Justification

It's tradition for global constants to named this way. You must be careful to not conflict with other global *#define*s and enum labels.

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.

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;
```

A Line Should Not Exceed 78 Characters

- Lines should not exceed 78 characters.

Justification

- Even though with big monitors we stretch windows wide our printers can only print so wide. And we still need to print code.
- The wider the window the fewer windows we can have on a screen. More windows is better than wider windows.
- We even view and print diff output correctly on all terminals and printers.

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.
- The *default* case should always be present and trigger an error if it should not be reached, yet is reached.
- If you need to create variables put all the code in a block.

Example

```
switch (...)
{
   case 1:
      ...
   /* comments */

   case 2:
   {
      int v;
      ...
   }
   break;

   default:
}
```

Use of *goto,continue,break* and *?:*

Goto

Goto statements should be used sparingly, as in any well-structured code. The goto debates are boring so we won't go into them here. The main place where they can be usefully employed is to break out of several levels of switch, for, and while nesting, although the need to do such a thing may indicate that the inner constructs should be broken out into a separate function, with a success/failure return code.

```
   for (...) {
      while (...) {
      ...
         if (disaster) {
            goto error;

         }
      }
   }
   ...
error:
   clean up the mess
```

When a goto is necessary the accompanying label should be alone on a line and to the left of the code that follows. The goto should be commented (possibly in the block header) as to its utility and purpose.

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:

1. 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 probablity of declaring a pointer when you meant to declare just a char.

To Use Enums or Not to Use Enums

C allows constant variables, which should deprecate the use of enums as constants. Unfortunately, in most compilers constants take space. Some compilers will remove constants, but not all. Constants taking space precludes them from being used in tight

memory environments like embedded systems. Workstation users should use constants and ignore the rest of this discussion.

In general enums are preferred to *#define* as enums are understood by the debugger.

Be aware enums are not of a guaranteed size. So if you have a type that can take a known range of values and it is transported in a message you can't use an enum as the type. Use the correct integer size and use constants or *#define*. Casting between integers and enums is very error prone as you could cast a value not in the enum.

Use Header File Guards

Include files should protect against multiple inclusion through the use of macros that "guard" the files. Note that for C++ compatibility and interoperatibility reasons, do **not** use underscores '_' as the first or last character of a header guard (see below)

```
#ifndef sys_socket_h
  #define sys_socket_h  /* NOT _sys_socket_h_ */
  #endif
```

Macros

Don't Turn C into Pascal

Don't change syntax via macro substitution. It makes the program unintelligible to all but the perpetrator.

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. Archeologists 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!
*/
```

Layering

Layering is the primary technique for reducing complexity in a system. A system should be divided into layers. Layers should communicate between adjacent layers using well defined interfaces. When a layer uses a non-adjacent layer then a layering violation has occurred.

A layering violation simply means we have dependency between layers that is not controlled by a well defined interface. When one of the layers changes code could break. We don't want code to break so we want layers to work only with other adjacent layers.

Sometimes we need to jump layers for performance reasons. This is fine, but we should know we are doing it and document appropriately.

Miscellaneous

General advice

This section contains some miscellaneous do's and don'ts.

- Don't use floating-point variables where discrete values are needed. Using a float for a loop counter is a great way to shoot yourself in the foot. Always test floating-point numbers as <= or >=, never use an exact comparison (== or !=).
- Compilers have bugs. Common trouble spots include structure assignment and bit fields. You cannot generally predict which bugs a compiler has. You could write a program that avoids all constructs that are known broken on all compilers. You won't be able to write anything useful, you might still encounter bugs, and the compiler might get fixed in the meanwhile. Thus, you should write ``around'' compiler bugs only when you are forced to use a particular buggy compiler.
- Do not rely on automatic beautifiers. The main person who benefits from good program style is the programmer him/herself, and especially in the early design of handwritten algorithms or pseudo-code. Automatic beautifiers can only be applied to complete, syntactically correct programs and hence are not available when the need for attention to white space and indentation is greatest. Programmers can do a better job of making clear the complete visual layout of a function or file, with the normal attention to detail of a careful programmer (in other words, some of the visual layout is dictated by intent rather than syntax and beautifiers cannot read minds). Sloppy programmers should learn to be careful programmers instead of relying on a beautifier to make their code readable. Finally, since beautifiers are non-trivial programs that must parse the source, a sophisticated beautifier is not worth the benefits gained by such a program. Beautifiers are best for gross formatting of machine-generated code.
- Accidental omission of the second ``='' of the logical compare is a problem. The following is confusing and prone to error.

  ```
  if (abool= bbool) { ... }
  ```

  Does the programmer really mean assignment here? Often yes, but usually no. The solution is to just not do it, an inverse Nike philosophy. Instead use explicit tests and avoid assignment with an implicit test. The recommended form is to do the assignment before doing the test:

  ```
  abool= bbool;
  if (abool) { ... }
  ```

- Modern compilers will put variables in registers automatically. Use the register sparingly to indicate the variables that you think are most critical. In extreme cases, mark the 2-4 most critical values as register and mark the rest as REGISTER. The latter can be #defined to register on those machines with many registers.

Be Const Correct

C provides the *const* key word to allow passing as parameters objects that cannot change to indicate when a method doesn't modify its object. Using const in all the right places is called "const correctness." It's hard at first, but using const really tightens up your coding style. Const correctness grows on you.

Use #if Not #ifdef

Use #if MACRO not #ifdef MACRO. Someone might write code like:

```
#ifdef DEBUG
        temporary_debugger_break();
#endif
```

Someone else might compile the code with turned-of debug info like:

```
cc -c lurker.cc -DDEBUG=0
```

Alway use #if, if you have to use the preprocessor. This works fine, and does the right thing, even if DEBUG is not defined at all (!)

```
#if DEBUG
        temporary_debugger_break();
#endif
```

If you really need to test whether a symbol is defined or not, test it with the defined() construct, which allows you to add more things later to the conditional without editing text that's already in the program:

```
#if !defined(USER_NAME)
 #define USER_NAME "john smith"
#endif
```

Commenting Out Large Code Blocks

Sometimes large blocks of code need to be commented out for testing.

Using #if 0

The easiest way to do this is with an #if 0 block:

```
   void
   example()
   {
      great looking code

      #if 0
      lots of code
```

```
    #endif

    more code
}
```

You can't use **/**/** style comments because comments can't contain comments and surely a large block of your code will contain a comment, won't it?

Don't use #ifdef as someone can unknowingly trigger ifdefs from the compiler command line. #if 0is that even day later you or anyone else has know idea why this code is commented out. Is it because a feature has been dropped? Is it because it was buggy? It didn't compile? Can it be added back? It's a mystery.

Use Descriptive Macro Names Instead of #if 0

```
#if NOT_YET_IMPLEMENTED

#if OBSOLETE

#if TEMP_DISABLED
```

Add a Comment to Document Why

Add a short comment explaining why it is not implemented, obsolete or temporarily disabled.

File Extensions

In short: Use the *.h* extension for header files and *.c* for source files.

No Data Definitions in Header Files

Do not put data definitions in header files. for example:

```
/*
 * aheader.h
 */
int x = 0;
```

1. It's bad magic to have space consuming code silently inserted through the innocent use of header files.
2. It's not common practice to define variables in the header file so it will not occur to developers to look for this when there are problems.
3. Consider defining the variable once in a .c file and use an extern statement to reference it.

Mixing C and C++

In order to be backward compatible with dumb linkers C++'s link time type safety is implemented by encoding type information in link symbols, a process called *name mangling*. This creates a problem when linking to C code as C function names are not mangled. When calling a C function from C++ the function name will be mangled unless you turn it off. Name mangling is turned off with the *extern "C"* syntax. If you want to create a C function in C++ you must wrap it with the above syntax. If you want to call a C function in a C library from C++ you must wrap in the above syntax. Here are some examples:

Calling C Functions from C++

```
extern "C" int strncpy(...);
extern "C" int my_great_function();
extern "C"
{
   int strncpy(...);
   int my_great_function();
};
```

Creating a C Function in C++

```
extern "C" void
a_c_function_in_cplusplus(int a)
{
}
```

__cplusplus Preprocessor Directive

If you have code that must compile in a C and C++ environment then you must use the *__cplusplus* preprocessor directive. For example:

```
#ifdef __cplusplus

extern "C" some_function();

#else

extern some_function();

#endif
```

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.