Chetu Inc.

Chetu
Delivering World-Class IT Solutions

C CODING GUIDELINES

A Generic Recommendation to all

From ETC

Disclaimer of Content

Contents

1.0 NAMES

1.1 Function Names

- 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.

- 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.

  Example: is_hit_retry_limit.

1.2 Include Units in Names

If a variable represents time, weight, or some other unit then include the unit in the name.

Example:
```
uint32 timeout_msecs;
uint32 my_weight_lbs;
```

1.3 Structure Names

- Use underbars ('_') to separate name components
- When declaring variables in structures, declare them organized.
  E.g,
  don't use ``int a; char *b; int c; char *d";
  use ``int a; int b; char *c; char *d".
  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.

1.4 Variable Names on the Stack

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

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;
    }
```

1.5 Pointer Variables

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

Example

```
char *name= NULL;
char *name, address;
```

1.6 Global Variables

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

Example

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

1.7 Global Constants

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

Example

```
    const int A_GLOBAL_CONSTANT= 5;
```

1.8 #define and Macro Names

- Put #defines and macros in all upper using '_' separators. 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.

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)
```

1.9 Enum Names

- Labels All Upper Case with '_' Word Separators

Example

```
      enum PinStateType {
            PIN_OFF,
            PIN_ON
      };
```

2.0 Formatting

2.1 Brace Placement

Of the three major brace placement strategies one is recommended:

```
if (condition) {                    while (condition) {
      ...                                 ...
}                                   }
```

2.2 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.

Example

```
if (1 == somevalue) {
      somevalue = 2;
}
```

One Line Form

```
if (1 == somevalue) somevalue = 2;
```

2.3 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 */
```

2.4 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.

Example

```
      if (condition) {
      }

      while (condition) {
      }

      strcpy(s, s1);

      return 1;
```

2.5 A Line Should Not Exceed 78 Characters

- Lines should not exceed 78 characters.

2.6 If Then Else Formatting

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.

Condition Format

Always put the constant on the left hand side of an equality/inequality comparison. For example:
if (6 == errorNum) ...

2.7 *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:
}
```

2.8 Avoid use of goto,continue,break and ?

2.9 One Statement Per Line

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

2.10 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 */
```

2.11 Enums

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.

2.12 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
```

3.0 MACROS

3.1 Don't Turn C into Pascal

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

3.2 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) {
      if( x > y )
            return x;
      return y;
}
```

3.3 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))
```

3.4 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.

3.5 Initialize all Variables

You shall always initialize variables. Always. Every time. Many problems are eventually traced back to a pointer or variable left uninitialized.

3.6 Short Functions

- Functions should limit themselves to a single page of code.

3.7 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++)
{
        ;
}
```

3.8 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())
```

4.0 Documentation

4.1 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

4.2 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.

4.3 Make Gotchas Explicit

Explicitly comment variables changed out of the normal control flow or other code likely to break during maintenance.

4.4 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.

4.5 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. */
```

4.6 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.

5.0 MISCELLANEOUS

5.1 General advice

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

- Don't use floating-point variables where discrete values are needed. Always test floating-point numbers as <= or >=, never use an exact comparison (== or !=).
- Do not rely on automatic beautifiers.
- Use good **modular design**. Think carefully about the functions and data structures that you are creating before you start writing code.
- Use good **error detection and handling**. Always check return values from functions, and handle errors.
- Every time you **dynamically allocate memory** (malloc), you should have code to free this memory at some other point in your program.
- As a general guide, **no function should be longer than a page long**. Of course there are exceptions, but these should truly be the exception.
- **Use descriptive names** for variables, functions, and constants You don't want to make function and variable names too long, but they should be descriptive.
- **Pick a capitalization style** for function names, local variable names, global variable names, and stick with it. For example "square_the_biggest" or "squareTheBiggest" or "SquareTheBiggest".
- **Define Constants and use them** in your program rather than using numerical values. Constants make your code more readable, and easier to change.
  For example:
  ```
  Do this:                        Not this:
  --------                        --------
  #define MAX 50
  int buf[MAX];                   int buf[50];
  if( i< MAX) { ... }             if( i < 50) { ... }
  ```
- **Avoid using global variables**; instead, pass variables by reference to functions that change their value.
- **The main function should not contain low-level details**. It should be a high-level overview of your solution (remember top-down design).
- **Use good indentation**. Bodies of functions, loops, if-else stmts, etc. should be indented, and statements within the same body-level should be indented the same amount:
  ```
  int blah(int x, int y) {
        stmt1;
        while (...) {
              stmt2;
  ```

```
                if(...) {
                      stmt3;
                } else {
                      Stmt4;
                } stmt5;
          }
          Stmt6;
    }
```

- Your source code should not contain lines that are longer than 80 characters long. If you have a line that is longer than 80 character, break it up into multiple lines. Here is an example of how to break up a long boolean expression into three lines:

```
      if ( ((blah[i] < 0 ) && (grr[j] > 234))
            || ((blah[j] == 3456) && (grr[i] <= 4444))
            || ((blah[i] > 10) && (grr[j] == 3333))
         )
      {
            stmt1;
            stmt2;
      } else { ...
```

  Here is an example of breaking up a long comment into multiple lines:

```
      x = foo(x);  /* compute the value of the next prime number */
                   /* that is larger than x (foo is a really bad */
                   /* choice for this function's name) */
```

  Here are two ways to break up a long string constant:

```
printf("here is a really long string with an int value %d", x);
printf(" that I want to print to stdout\n");
OR
printf("here is a really long string with an int value %d"
      " that I want to print to stdout\n", x);
```

The easiest way to make sure you are not adding lines longer than 80 characters wide is to always work inside a window that is exactly 80 characters wide.

5.2 Position of main()

Declare your main() function as the first function after the preamble.

5.3 Structures and Unions

Always use typedefs method of declaring structures and unions. They must contain a structure name, object name and pointer name even if these are not used. You should always use the pointer declaration within your code wherever possible.

```
            typedef struct AjSFubar
```

5.4 Variable declarations

Declare all variables at the top of each function. Declare one variable per line i.e.
this is good

```
{
  ajint a;
  ajint b;
  ajlong c;
  ...
```

this is bad.

```
{
  ajint a,b;
  ajlong c;
  ...
```

Always initialize Object pointer variables to NULL. Initialize other data types as appropriate in the code. Align initializations for easy reading (e.g.)

```
AjPStr seqsubstring = NULL;
AjPDouble xarray = NULL;
```

Do not initialise declarations with functions e.g. never use

```
AjPStr seqsubstring = ajStrNew();
```

5.5 Precedence of operators

Avoid confusion introduced by using operator precedence e.g. this is bad:
a = b * c + 1;

whereas this is good.

a = (b * c) + 1;

5.6 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".

5.7 Use #if Not #ifdef

Use #if MACRO not #ifdef MACRO. Someone might write code like:
Alway use #if, if you have to use the preprocessor. This works fine, and does the right thing.

```
#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
```

5.8 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
      }
```

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.

5.9 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 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.

Source File

Use the following organisation for source files:

- includes of system headers
- includes of local headers
- type and constant definitions
- global variables (There should not be any)
- prototypes functions

Header File

In header files, use the following organisation:

- type and constant definitions
- external object declarations
- external function declarations

It is not necessary to make the declarations 'extern'.Never use nested includes! Avoid exporting names outside individual C source files; i.e., declare as static every function that you possibly can. Always use full ANSI C prototypes e.g.

5.10 Mixing C and C++

If you want to create a C function in C++ you must wrap it with the *extern "C"* syntax. If you want to call a C function in a C library from C++ you must wrap in the *extern "C"* syntax.
Example:

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
```

5.11 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. For example:

```
if (22 == foo) { start_thermo_nuclear_war(); }
```

Instead of magic numbers use a real name that means something.

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(); }
```

5.12 Error Return Check Policy

- Check every system call for an error return, unless you know you wish to ignore errors.
- Include the system error text for every system error message.
- Check every call to malloc or realloc.