Enigmatic Engineers Coding Style

By M. D. Patel, B.S.

Includes parts adapted from the Linux Kernel Coding Style by Linus Torvalds, M.S.

### Indentation

* Tabs are 8 spaces(unlike the Linux style where spaces are not used), and thus indentations are also 8 characters.
* The preferred way to ease multiple indentation levels in a switch statement is to align the switch and its subordinate case labels in the same column instead of double-indenting the case labels.
  + Ex.

switch (suffix) {

case 'G':

case 'g':

mem <<= 30;

break;

case 'M':

case 'm':

mem <<= 20;

break;

case 'K':

case 'k':

mem <<= 10;

/\* fall through \*/

default:

break;

}

* Don’t put multiple statements on a single line.
  + Ex.

if (condition) do\_this;

do\_something\_everytime;

* Don’t put multiple assignments on a single line.
* Don’t leave whitespace at the end of lines.

### Breaking long lines and strings

* The preferred limit on the length of lines is 80 columns.
* Statements longer than 80 columns will be broken into sensible chunks, unless exceeding 80 columns significantly increases readability and does not hide information.

### Placing Braces and Spaces

* For all statement blocks other than those defining functions, structures, or classes put opening braces on the same line as the previous statement and closing braces on their own line.
  + Ex.

if (x is true) {

we do y

}

* The closing brace is empty on a line of its own, except in the cases where it is followed by a continuation of the same statement, ie a while in a do-statement or an else in an if-statement
  + Ex.

do {

body of do-loop

} while (condition);

and

if (x == y) {

..

} else if (x > y) {

...

} else {

....

}

* For function bodies both braces should be placed on their own lines.
  + Ex.

int function(int x)

{

body of function

}

* Do not unnecessarily use braces where a single statement will do.
  + Ex.

if (condition)

action();

and

if (condition)

do\_this();

else

do\_that();

* If only one branch of a conditional statement is a single statement use braces in both branches.
  + Ex.

if (condition) {

do\_this();

do\_that();

} else {

otherwise();

}

### Spaces:

* Use a space after (most) keywords.
  + The exceptions are sizeof, typeof, alignof, and \_\_attribute\_\_, which look somewhat like functions (and are usually used with parentheses). That said, these are low-level C language keywords which should not be used in C++ especially in userspace code where the C++ language provides safer abstractions that allow programmers to avoid their direct use.
  + Do not add spaces around (inside) parenthesized expressions.
* When declaring pointer data or a function that returns a pointer type, the preferred use of \* is adjacent to the data name or function name and not adjacent to the type name.
  + Ex.

char \*linux\_banner;

unsigned long long memparse(char \*ptr, char \*\*retptr);

char \*match\_strdup(substring\_t \*s);

* Use one space around (on each side of) most binary and ternary operators, such as any of these: = + - < > \* / % | & ^ <= >= == != ? :
* Don’t use spaces after unary operators such as: & \* + - ~ ! sizeof typeof alignof \_\_attribute\_\_ defined
* No space before the postfix increment & decrement unary operators and no space after the prefix increment and decrement operators.
* No spaces around the . and -> structure and class member operators.
* Do not leave trailing whitespace at the ends of lines.

### Naming

* All variable and function names including those that are class or structure members are have snake case names. If you are unfamiliar with snake case the following examples show what it looks like.
  + Ex.

size\_t count\_of\_users;

char last\_read;

bool is\_locked(const mutex &m); //function

* Names should be meaningful and make it clear to readers what a given value represents or what a given function does. Do not obfuscate variable or function names in an attempt to shorten them. This isn’t C in the 70s, characters in a variable name do not come at a premium!
* All type names are to use pascal case in which all words start with a capital letter and there are no spaces or underscores.
  + Ex.

struct Point

{

Body

};

class ChessBoard

{

Body

};

using Move = unsigned char; // An example using a type alias.

enum GameMarker

{

Body

};

union SumType

{

Body

};

* Types should have meaningful names that make it apparent to a reader what they and their instances represent.
* Each discrete section of a project is to be placed into its own namespace to avoid naming collisions. Namespaces should have an all lowercase name that is less than 10 characters long.
* Use of the using keyword with entire namespaces defeats the purpose of using namespacing and is disallowed. Instead opt to import individual items from a given namespace with the using keyword.
  + Ex.

Namespace board {

class GameBoard…

}

//in another location

using namespace board; //VERY BAD!!! NEVER DO THIS!

using board::GameBoard;//Importing individual item. Much safer.

* Type aliasing should be used ONLY when it makes your code more readable. The most common example of this would be making built-in types more meaningful such as by aliasing unsigned long as Inches where it represents a measurement in inches or std::pair<long, long> as Point where it represents a point in cartesian coordinates.

Ex.

using inches = unsigned long int; //aliasing a primitive

/\*aliasing a standard library type\*/

using point = std::point<long, long>;

* C style typedefs should NEVER be used in C++ because they do not understand or work with C++ namespacing.

### Includes

* Only #include header files (.h, .hpp), never source files (.cc, .cpp).
* All header files should be wrapped in preprocessor header guards to prevent include duplication:
  + Ex.

#ifndef filename\_H

#define filename\_H

Entire file body

#endif

### Functions:

* These rules apply equally to free functions and class and structure member functions (sometimes called methods).
* Functions should be short and do just one thing.
* There should be a single space after each comma in the parameter list of a function declaration and for each of them in the argument list of a function call.
* Function definitions should fit on one or two screenfuls of text (the ISO/ANSI screen size is 80 columns x 24 rows).
* The maximum length of a function is inversely proportional to the complexity and indentation level of that function. So, if you have a conceptually simple function that is just one long (but simple) case-statement, where you have to do lots of small things for a lot of different cases, it’s OK to have a longer function.
* However, if you have a complex function, and you suspect that a less-than-gifted first-year high-school student might not even understand what the function is all about, you should adhere to the maximum limits all the more closely. Use helper functions with descriptive names.
* The number of local variables in a function should not exceed 7, or you are doing something wrong. Re-think the function and split it into smaller pieces.
* In source files, separate functions with one blank line. In header files you may separate individual groups of related functions with a single blank line but ideally try to place each such group into its own header/source file pair. The implementations of all of the functions defined in a header file should all be placed in a corresponding source file with the same name.
* Any function parameter that is larger than 64 bits (the size of a register in a 64-bit architecture) should be passed by reference. If the original value does not need to be changed then write the function so that it takes that parameter by const reference.
* The previous rule also applies to the this pointer implicitly taken by class and structure member functions. These functions should be marked const as in the following example if they do not need to modify the calling object. A single space should be placed between the closing parenthesis of the function and the keyword const.
  + Ex.

class Point

{

…

long norm() const;

};

* Functions should never take raw pointers as parameters unlike in C.
* Avoid recursion unless the recursive implementation of a function is very obviously cleaner than its iterative counterpart.
* Do not use varargs.
  + In an imperative language like C++ recursion tends to severely reduce the readability of a function and it also makes it harder for the compiler to optimize the code.

### Variables and Constants

* Any variables whose value should not change after initialization should be declared as const. If you are unsure about this it is safer to declare a variable as const and then change it later.
* All variables should be declared on their own line with their type clearly written.
* Any variable that can be initialized upon creation should be and those that cannot, should be initialized as soon as possible.
* All variables should be declared in the narrowest scope in which they can.
* Pointer variables should be declared with the \* adjacent to the variable name and not the type.
* Global variables should be avoided to the extent possible.
* Static variables should be used for values that need to exist for the entire lifetime of the program, but they should be avoided to the extent possible.
  + When they are used, they should be initialized where they are declared.
* Do not use numeric literals (so called magic numbers) or string literals in function bodies. Instead opt to use named constants (variable declared with const).
  + These allow you to change those values more easily and give meaningful names to the values making your code more readable.

### Structures

* Structures should be used as plain old data (POD) types only. They should not have private or protected members, member functions, or be involved in any inheritance hierarchies at all even though the C++ language allows them to be used in all of the same ways as classes.
* Structures should not contain classes other than those in the standard library.
  + Allowing this would be inconsistent with the previous rule by allowing inheritance to be substituted with composition thus breaking the semantics of structures being POD types.
* Both braces in a structure definition should be on their own lines except for the ending semicolon immediately after the closing brace.

### Classes and Objects

* Static member functions should be used for only one purpose: to access or modify non-public static members of their class.
  + C++ isn’t Java where everything must go inside classes and putting static functions inside classes hurts readability. If you have a function that needs to take more than one instance of a class or none at all but is related to the class then make it a free function in the same namespace as the class.
* Inversely if you have a function that takes one instance of some class, make it a member function.
* Avoid superfluous getters and setters. If a member variable of a class is meant to be manipulated and viewed by users of that class then simply declare it as public. However if that value being assigned to such a variable needs to be validated or it should not always be visible then make the member variable protected or private and create the appropriate getter or setter.
* Prefer composition to inheritance where possible.
* Never use multiple inheritance. It creates more problems than it solves. Any problem that can be modelled using multiple inheritance can be modelled more cleanly using composition.
* Never create an inheritance hierarchy that consists of more than 3 classes.
* Adhere to the principle of least access: declare every class member with the most restrictive access modifier that can be used for it.
* Both braces in a class definition should be on their own lines except for the ending semicolon immediately after the closing brace.
* Classes should be declared in a header file and all their member functions should be defined in a source file with the same name.
* Each class should have at most one of each of a public, protected, and private section. In each of these sections each variable and functions should be declared on its own line. Each access modifier should start at the same column as the class keyword and opening brace.
  + Ex.

class Book

{

public:

std::string title;

std::string author;

Book();

virtual ~Book();

protected:

std::string isbn;

float get\_ncopies(const Library &lib) const;

};

* Virtual functions and dynamic polymorphism should be avoided to the extent possible but in certain situations they may be used if the team has agreed on their use beforehand.
* When initializing use the function-like notation instead of using the brace-based notation allowed as of C++11.
  + Ex.

Point a(5, 10); //good

Point b{5, 10}; //Don’t do this. Harder to read.

* When calling a constructor outside of an initialization, call it like a function. Once again do not use the brace-based notation and definitely do not use a raw initializer list.
  + Ex.

Point(5, 10); //good

Point{5, 10}; //bad

//in a function that returns point

return {5, 10}; //worst

* Standard library classes and class templates should be used where possible (along with type aliases where they make sense) instead of writing new classes that implement the same behavior.

### Unions

* Use the C++ standard library type std::variant instead of unions.
  + Using raw unions to represent sum types generally makes type safety difficult for the uninitiated. It is better to avoid these problems altogether.

### Templates

* Minimize the use of templates unless you are certain that genericity is needed.

### Comments

* All code should be commented.
* Function declarations should be preceded by block comments written in complete sentences in the following format:

/\*Purpose: Write what the functions does.

Preconditions: Write the preconditions of the function.

Postconditions: Write the postconditions of the function

making it clear what changes when it is called.

Notes: This part is optional for any other info like what exceptions the function throws.

\*/

* Variables can have an inline comment explaining what they are for if it is not already evident by their names.
  + Ex.

Book checked\_out; //The book currently checked out by the user

* All type declarations (class, struct, enum, union) and type aliases (using keyword) should be preceded by a block comment that explains the purpose of the type or alias.
* Comments should be written in complete sentences with proper grammar, spelling, and punctuation. Make your high school English teacher proud.

### Notes

* The author of this document has intentionally violated other parts of this style guide in the code examples for certain concepts. This was done both for the sake of brevity and to ensure that this document could be completed in a reasonable timeframe. Future revisions of this document may modify these examples to be fully compliant with the entirety of the coding style.
* Additionally, the astute reader will note that some of the code examples are in fact not actual code but rather C++-like pseudocode. This is for the same reasons as above.