# *Coding Style*

## Code formatting

1. Use **80-character** wide lines. Fold long lines on math operators, assignments, function arguments.
2. Use 4-space indentation for blocks.
3. Use spaces, not tabs, to indent text. Configure your editor to replace tabs into spaces, and to use 4-space tab stops.
4. Do not put space before the opening brace "(" in control structures and function calls; always put a space after an opening brace, and before the closing brace if the preceding character is not a brace (my style?).
5. Put spaces around operators in expressions, unless expressions extend for more than one line. In the latter case, remove spaces consistently around the highest-priority operators, so that a long expression is optically split, say, into a sum of terms or similarly.
6. Use Perl manual style for formatting Perl control structures and procedures; use the similar style (K&R with braces for even a single statement, or 1TBS) for C.
7. I always put the C function return value type on the same line as the function name; the GNU coding style, however, advocates specifying the return value type on a separate line, so that function name starts at the column 0 of the line. Both styles are good; use whatever is used in a file or a project, but use it throughout the project consistently.
8. In all cases not mentioned, the GNU coding standards as described in the GNU coding standards page (http://www.gnu.org/prep/standards/standards.html) and the Gnome coding style (http://developer.gnome.org/doc/guides/programming-guidelines/code-style.html) apply.

## Names

1. Do not use abbreviations for variable names – 'hours' is a better variable name than 'hrs'. Use only "well known" abbreviations, agreed upon in the project, and document your abbreviations. Use your own abbreviations only for names that are very common in your code, occur in complex expressions and abbreviating them would make code more readable.
2. Select descriptive variable names.
3. Select descriptive function names, explaining what does the function do.
4. Prefix function name with a package name in languages that do not have scoping such as old C.

## Error messages

1. Use our internal standard for error message formatting. Basically, it is a GNU error message format with slight changes to enhance parseability.
2. Error or warning reports should always contain:
   1. the program name which user was using to invoke the code (but usually not the subroutine name),
   2. the name of the file that was being processed when the error happened,
   3. the position within this file (line, data block name, column) where the error was detected,
   4. a short but comprehensive message describing what happened and how to correct the situation,
   5. if an error occurred in a system function call, include also the system error message.
3. Use one function to format error and warning messages in one place.

## Code commenting

1. Each function should have a one-two sentence comment describing in human terms what does the function do, and any special interface requirements not evident from the function signature (e.g.: for a C function, it should be specified whether a char* string returned should be free()-ed or if it is static)

   If a function can not be described in a simple and clear sentence, consider redesigning the function interface altogether – probably something is wrong with you function design.
2. Do not comment obvious things (such as: 'int i = 0; /* i is initialized to zero */'). In such cases, the code is more readable than the comment, and the comment will become incorrect if initialisation is changed to 'int i = 1' – the bug that no C compiler will catch.

   In other words, **comments should be orthogonal to code**.
3. Comments do not describe what is coded; they describe **what should have been coded.**
4. Bear in mind that comments are not checked by the compiler, and will probably be left alone when the code is changed. Thus, try to describe the general, persistent features of the code, and not the transient details. Never specify parameter types in the comments in a statically typed language, since they are apparent from the function signature (but this might be appropriate in a dynamically typed or untyped language). Rather, describe the "physical meaning" of the parameters, the meaning that will not be changed (probably because changing it would break too much of a the code).
5. Comments will probably stay around for a while after the code is changed, so

try to formulate them referring to context in which you write the comment, not the context which the function will appear in the future. A comment like “this function does not accept NULL pointer” will become incorrect when the function is modified to accept NULL; the comment “as of version 1.1, the function f() does not accept NULL pointer” will remain true even if function f() is changed in the version 2.0.

## Single point of truth; code refactoring

1. Never repeat the same or related information in several places. Put it into one place (variable, module) and make all other referents use that place.
2. **If you want to copy a function and modify it slightly, never do this!** Instead, refactor the code so that all places use the same function. The examples of the code refactoring are:
    1. if you need a more general function, with a wider interface, transform the specific function into a more general one (possibly by adding more parameters), under a different name, and reimplement the old function by a call to this new, more general one, passing a default or computed parameter. If you language supports default parameters, and the old function differs from the new one by fixing a value of some parameter, make this parameter a default – in this case you will not even need a new function;
    2. If you need a slightly different interface to a function, make a versatile function with complex interface to fit all cases, and then make simple wrappers that provide a more simple interface and simply call that complicated function with the appropriate parameters. The sever interfaces can coexist for as long as needed.
    3. If you need a slightly different code in the middle of a complex function, split the function in three parts, and reused the pre-code and post-code to implement two functions with that changed middle part;
3. If code refactoring changes function's behaviour into the incompatible one, **never change the function behaviour alone, always change function name as well**. If there is a good reason to leave the old function name with the new signature, first implement the new interface, make sure there are no calls to the old function, and then (in a separate revision!) rename the function into the old name. This is especially important for behaviour changes that are not caught by the compiler as incompatibilities but in fact are such.

## Functions, parameters and variables

1. **Global variables are evil.** Avoid them as much as you can. All data should be passed to functions as parameters. Use structures and objects of there are a lot of variables to pass. Even if you need to introduce extra parameter in many functions in a call chain, do this rather than introducing a global variable.

    1. Global variables make code unmaintainable (virtually);
    2. global variables make your code non-reenterable.

   Actually, the only justified use of global variables in a new code is setting a debug flag (and this flag should be local to a module, anyway).

2. Ideally, a function should behave like a, well, function – it should have no side effect and only return values. In languages that support multiple return values, use them rather than modification of a function parameter.
3. Avoid passing parameter that tells function what to do (e.g. do not use parameters like 'change( int x, in task ) /* task == 1 – change colour, task == 2 – change shape */). If you do, your program slowly becomes a bytecode interpreter of strange untyped values :). Rather, create two functions, 'change_colour( int x )' and 'change_shape( int x )'.
4. Do not force policy decisions in low level (toolbox) functions. If a low-level subroutine needs to know the current policy (e.g. if and where it is allowed to create temporary files, or whether to include debug information in error reports), pass it extra parameter or receive return extra value, so that a top-level caller can decide on the policy.

## Asserts

1. Use asserts to check function pre-conditions. In particular, use asserts **always** when a function would segfault if the asserted statement were false (e.g. assert( p != NULL ) or assert( p ) before accessing p->val in C, if the p != NULL is not obviously guaranteed by the code, say by the condition of the if() or while() statement).
2. Use assert() from the Carp::Assert package or 'die unless ...' statements in Perl as equivalents of the C assert() macro. Use them everywhere where the code would fail if the condition is not satisfied, and the condition should always be satisfied, regardless of input data or events.