C Coding standards

March 18, 2018

Contents

1 Introduction ............................ 1
  1.1 Benefits ................................... 1

2 Standard .................................. 2
  2.1 Naming conventions (5) .............. 2
  2.2 Modularity (8) ......................... 2
  2.3 Robustness (8) ......................... 2
  2.4 Readability (5) ......................... 3

1 Introduction

Coding standards are a set of rules and suggestions meant to guide the programmer while writing code. Standards are designed to improve code quality, reduce risk of bugs and improve productivity. Each organization decides on its own standard, but most share a common set of rules which have been found to work well. The coding standards for the labs are described in the next section.

1.1 Benefits

- Coding standards improve code readability. They reduce the time it takes a programmer to read unfamiliar code, since they writing style becomes common to all programmers on a team.

- They reduce the the risk of bugs, since the standards often define rules specifically designed to reduce common mistakes.

- They help programmers write higher quality code by prohibiting bad practices
2 Standard

2.1 Naming conventions (5)

- Variables (global and local), functions and function parameters must have descriptive names. They should imply the meaning of the variable. You should be able to loosely answer the question "What is this for?".
- An exception to the above rule is loop indices, as their usage is usually easily inferred from the loop’s statement.
- All identifier names must use snake_case except for macros (or, generally, anything defined using #define) names, which use CAPITALIZED_SNAKE_CASE.
- Variable names should not be too long. This is usually a sign of variables with unclear meaning.

2.2 Modularity (8)

- Functions should be kept short and concise and have as few responsibilities as possible.
- Recursive functions must have a single clear responsibility.
- Code copying is always bad practice. Use functions/macros instead.
- Use local-static or global variables sparingly. You should be able to justify not taking a different approach to solve the relevant problem.

2.3 Robustness (8)

- The return value of all System/API calls must be checked.
- All code blocks (if, while, for do-while etc. . .) must include curly braces ({}{). This include single line blocks and even empty blocks.
- Avoid magic numbers. Use characters when working with ASCII values, or variables/macros otherwise.
- Macros should not modify the state of the program (alter variable values).

\(^1\)Note that macros effectively copy code during the compiler’s pre-processing stage, but this is a safe way to copy code, since there’s only a single textual instance of the code in the source file.
• Conditions should test values using Yoda conditions. This reduces the common mistake of omitting an equal (‘=’) sign, replacing "if (x == 1) . . . " with "if (x = 1) . . . ".

• When writing a multi-line macro, use the form \texttt{do \{<macro multi-line body>\} while (0)}. This turns the macro’s lines into a statement.

2.4 Readability (5)

• Variables with modifiers (pointers, unsigned etc.) should be defined on their own line.

• Functions should have few return points (preferably 1).

• break/continue should be avoided in most cases.

• Avoid compounding many boolean conditions in a single statement. Consider nesting.

• Never write two statements on a single line\footnote{Note that multi-line macros expand to a multi-line statement, but since textually they are separated into lines, this still simple to read and not an issue.}

• It’s better to use parenthesis to clarify order of execution whenever the result may differ based on that order.

• Comments should not, cannot and will not replace intelligible code.