C Style Guide

Ethan Ruffing

March 3, 2015

Contents

Chapter 1

Introduction

When programming, it is essential for all programmers on a project to follow a common style. Doing so ensures consistency and productivity, allowing for a final product which is better functioning and more appealing to the client. A consistent style and detailed documentation will allow colleagues and future programmers to easily understand and use new source code.

This style guide is intended to provide a unifying basis for programming in the C programming language, and to provide a standard for documenting the programs created.

Chapter 2

File & Program Organization

File Format

Source code should be written so that no line exceeds eighty characters in length.

A header block should be placed at the top of each file giving the author, the date of creation, and, if applicable, the version number. This header block should also include a copyright notice and license header if applicable.

File names should be clear and concise so that someone can tell the general purpose of the file without having to open and read it or its header block.

File Structure

Function *declarations* should always be located within a header (`*.h`) file, while function *definitions* should be located in a source (`*.c`) file. If choosing to place documentation for the function in only one of these locations, it is best to place it in the header file, as the header file is intended to contain a high-level overview of each function.

Program Structure

Every effort should be made to avoid the use of global variables. Instead, preprocessor macros should be used for constants, and variables should be passed and returned between functions.

Chapter 3

Commenting & Documentation

Comments

Comments should be included where necessary to give a better understanding of code. When writing longer comments, follow Strunk & White's *The Elements of Style* for English grammar guidelines.

When writing dates and times, use the ISO 8601 standard. That is, for dates, YYYY-MM-DD; for times, hh:mm:ss; and, for date-times, YYYY-MM-DDThh:mm:ss.

Comments should be included that give detailed descriptions of all functions, classes, and instance variables.

Program Documentation

It is essential to provide a brief overview of your program. Such an overview should be placed in a file named `README.md`, and formatted using Markdown (a markup language designed to create nicely-formatted, but still easily parsed, plain-text files). This file should contain an overview of the program's purpose, its authors, and relavent copyright information. It should also include directions on where to find more extensive documentation, for both users and developers. However, detailed documentation on the program's functionality should *not* be included in this file.

In addition, it is prudent to maintain a `CHANGELOG` (or, if preferred, `CHANGELOG.md`) file, in which a brief overview of the changes made between each release of the program are listed. Note that the changes listed here should be only the major differences between versions; all minor changes and modifications should be recorded using some form of version control, such as Git or SVN.

Function Documentation

DocBlock-style comments should be provided for *every* function. DocBlock comments should be formatted using the `/** ... */` system, as illustrated in the example further down.

The most commonly used DocBlock interpreter for C documentation is Doxgen. It supports export into a multitude of formats and supports a large number of tags. For a reference of what tags are supported, see the Special Commands section of the Doxygen Manual.

Author and date (`@since`) should *always* be given for a file DocBlock. If using version numbering for the project, the version tag should also *always* appear in the class DocBlock.

At a minimum, functions should *always* have the date (since) tag. If multiple authors have contributed to the same class, they should also include the author tag. (If not given, the author is to be assumed to be the same as the class author.)

Functions should also have `@param` and `@return` tags where applicable. Any method other than a simple getter/setter (one which does not operate on the variable, but merely sets or returns its value) must have a description of its functionality. When applicable to the language, parameter tags should also specify the parameter direction (`[in]` or `[out]`). The description portion of these tags should be located on the next line (after the tag and variable name) and indented one level past the tag itself.

Global variables should have a DocBlock comment to explain their purpose, and macros should be described in a block somewhere near the beginning of the file.

Description text should be written in sentences and be punctuated with a period. Variable and `@param`/`@return` value descriptions, however, should be kept brief and not written as full sentences, unless absolutely necessary. (The first letter of these descriptions, though, should still be capitalized.)

A blank line should be placed after each paragraph of explanation; between the explanation and author, date, and version tags; and between those tags and the param and return tags.

Example

```
/**
 * This is an explanation of a function.
 *
 * @author Ethan Ruffing <ruffinge@gmail.com>
 * @since 2014-08-06
 *
 * @param[in] varName
 *     A brief description of the parameter varName
 * @param[out] outName
 *     A brief description of the output outName
 * @return
 *     A brief description of the function's return value
 */
```

Chapter 4

Basic Formatting

C source files should be written according the the Kernighan and Ritchie (K & R) style. Furthermore, source files should be written according to the C11 (ISO/IEC 9899:2011) standard. Extensive documentation on both of these can be found online.

Naming

In general, names should be concise, but long enough to understand immediately.

Variables

All variables should be named in headlessCamelCase.

Constants

Constants (including enum entries) should be named in UPPER_CASE.

Whitespace

Whitespace helps tremendously to improve readability. The following are basic guidelines, but can be expanded on to further improve readability.

Indentation

There seems to be an ongoing war regarding indentation in programming. For the purposes of this style guide, we will require the use of hard tabs (an actual `\t` character). IDEs and text editors should be set to use a tab width of four, and alignment of items such as comments should be based on this assumption.

Exception

There is one exception to this rule (which was alluded to previously).

When formatting block comments, all text within the block should be aligned using spaces in order to guarantee readable documentation everywhere. (NOTE, however, that the comment block itself should be indented using tabs.)

Blank Lines

One blank line should always be used in the following locations:

- Between methods
- Between a local variable in a method and its first statement
- Before a block or single-line comment
- Between logically separate sections of code within a method

Blank Spaces

Blank spaces should be used in the following locations:

- Between a keyword and a parenthesis. Note that a blanks space should *not* be used between a method and its opening parenthesis. Also, there should not be any blank space following an opening parenthesis or preceding a closing parenthesis. Example:

  ```
  while (true) {
      printf("test");
      ...
  }
  ```

- Before an opening brace (see above example)
- Between binary and ternary operators and their operands
- *Not* between unary operators and their operands
- After each semicolon in a `for` loop (for example, `for (int i = 0; i < n; i++) {}`)

Braces

- All loops, conditionals, and other such structures where braces are functionally "optional" *must* use braces, even if their contents is a single line.
- Opening braces should always be on their own line, aligned on the column of the item they are for.
- Close braces should always be on their one line, aligned horizontally with their open statement.

Chapter 5

Error Handling

There are multiple existing systems of error handling. For example, one established way to pass errors in C is to return “error code” values as integers. In some places, errors are passed as simple booleans (i.e., `false` means an error occurred).

It is prudent to devise and follow and error handling and checking system to use in all programs.