CSCI 262 C++ Style Guide

Christopher Painter-Wakefield and Mark Baldwin and Alex Anderson

Last updated: 1/18/2018

Modified from:

> "C++ Student Coding Standards"
>
> Mark Baldwin
>
> Version 1.02
> 5/21/2013

mark@baldwinconsulting.org
http://baldwinconsulting.org

1. ***Introduction***

Most companies use coding standards to ensure commonality and readability of code. The below standards are simplified versions of what can be found in industry. All coding assignments need to meet these standards.

There are several reasons for this document:

- Help prevent programming errors.
- Encourage good programming practices.
- Make it easier for instructors, tutors, and graders to read your code.

These styles try to strike a balance between discipline and flexibility. There are times you may need to violate these standards. If you feel you have found such a situation, please discuss it with your instructor.

Note that these style guidelines may not agree with the style guidelines/suggestions in your textbook or supplementary materials! Also, the naming style may conflict with the style used in third-party libraries; there are ways to minimize the impact of this, but they are beyond the scope of this document (and the course); just use the third-party names of things where needed.

2. Files

2.1. There should be one primary *.cpp* file named after the project or named *main.cpp* (either is acceptable). If you are given starter code, do not change the names of the files provided.

2.2. For very small projects, it is acceptable to combine all code (including template and class definitions) in a single *.cpp* file.

2.3. For large projects, each **class** will have two files, a *.h* file that contains the class declaration, and a *.cpp* file that contains the class definitions. The files must be named after the class.

2.4. For large projects, each **class template** will have a single *.h* file that contains the declaration of the class template and all of its methods.

2.5. Every header file must be guarded against multiple inclusions. Use one of the following techniques…

```
#pragma once
```

(note, this only works with some compilers, so it might be good practice to use the alternative below)

or

```
// foo.h
#ifndef FOO_H
#define FOO_H
  .
  header body
  .
#endif   // end of foo.h
```

3. Names

3.1. All class, variable, method, and parameter names are lower case, with underscores separating parts of the name ("snake case").

Examples:

```
class auto_structure
string message_length;
void resize(int new_size)
```

3.2. Exception: constants should be uppercase, with underscores separating parts of the name.

Example:

```
const int DAYS_IN_YEAR = 365;
```

3.3. Use descriptive names

Good Example:

```
int days_since_start = 0; // days since the start of the project
```

Bad Example:

```
int d = 0;    // days
```

3.4. Exception: temporary integer variables such as those used for loop counters are traditionally single letters (i, j, k). This is preferable to longer names.

4. Globals and Constants

4.1. Global Variables

4.1.1. Do not use global variables. All variables should be local to a method or to an object.

4.1.2. Exception: constants may be global.

4.2. Constants

4.2.1. Constants should be in all caps.

4.2.2. Do not use #define but instead use *const* construct.

4.2.3. Constants should be used for magic numbers (i.e. constants that have meaning) and strings wherever possible. 0 and 1 are not magic numbers.

Good Example:

```
const int DAYS_IN_YEAR = 365;
days = years * DAYS_IN_YEAR;
```

Bad Example:

```
days = years * 365;
```

5. *Indentation and Braces*

5.1. Braces are optional only for single statements following a control structure and only if the single statement is on the same line as the control structure. Multiple control structures should not be on a single line.

***Example*:**

```
if (foo > max_foo) foo = max_foo;
```

5.2. Brace alignment

5.2.1. There are many accepted styles for aligning braces; one of the following two styles is recommended. Whatever style you adopt, you **must** be consistent throughout your program code.

5.2.2. Option 1: Opening brace is on same line as preceding code; closing brace is vertically aligned with left of preceding code:

Example:

```
if (a == b) {
   a = b + 1;
   b = 0;
} else {
   a = 0;
}
```

5.2.3. Option 2: Braces align vertically:

Example:

```
if (a == b)
{
   a = b + 1;
   b = 0;
}
else
{
   a = 0;
}
```

5.3. Code inside of braces should be indented and aligned. If your code editor auto-indents, use the default indent for the editor (however, you may need to re-indent starter code if different). Acceptable choices for indentation are a tab character **or** four spaces.

5.4. Indentation should be consistent throughout a code file. To fix bad indentation, many editors have an “auto indent” function. It may be helpful to find out how this function works within your editor.

5.5. Never place more than one line of code on the same line. Only create one-line functions when your function code is only one line.

***Bad example*:**

```
int pop() { _back--; _count--; return data_array[_back]; }
```

***Good example*:**

```
int pop() {
  _back--;
  _count--;
  return data_array[_back];
}
int peek() { return data_array[_back]; }
```

6. Comments

6.1. The purpose of comments is twofold. First, they are there to provide understanding of what exactly the code is doing, both for the programmer and any third party (including the instructor). Second, they are a powerful tool in writing effective code.

Hint: *Try writing all your primary comments before creating code. Just like an outline for a paper, the comments then become the framework by which you can then provide the details of the actual code.*

6.2. Generally, there are two types of comments.

6.2.1. Inline comments appear on the same line as the thing they describe. These are useful for presenting small bits of information about the particular line of code.

6.2.2. Block comments are one or more lines of comments that appear by themselves. These comments generally are used at the start of files, objects and methods, but they are also used to describe specific blocks of code and what is being accomplished.

6.3. All files should start with a block comment similar to this:

```
/*
   File Name: the name of the file
   Author: your name
   Course and Assignment: Course number and assignment
   Description: Describe what is contained in the file
*/
```

6.4. All class and class template declarations should start with a block comment similar to this:

```
/*********************************************************
  class name

  describe what the class does
*********************************************************/
```

6.5. Function definitions are ideally preceded with a block comment explaining the purpose of the function and its parameters, unless these are abundantly clear from the context and names of the function and its parameters.

6.6. Variable declarations may be followed by an inline comment describing the purpose of the variable, but clear naming of variables is preferred over commenting.

6.7. Comments are needed in the body of a routine if the logic or code is not obvious.

6.8. Do not comment on the obvious. Comments like below are worthless…

```
counter++;   // increment the counter
```