Tyler Tran for CSM 125 Spring 2020

General Basic Coding “Style Guide”

This is a brief style guide for how clean code should look. The examples provided span multiple languages including C++, Python, and Java. Even if you do not understand the code written, please do try to learn something from the points I make about structuring code to be “readable”.

Overall, the most important thing with your own coding style is that it looks clean and that you’re consistent with how you do things.

**Why Does Code “Style” Matter**

Coding style refers to how well written your code is in terms of its readability, not its functionality. This concept is important for several reasons. In the shorter term, many universities structure code style and/or readability into their rubrics to enforce the syntax and structure that their professors prefer (for consistency among students for grading and such).

More importantly, having good coding style is important for both your own learning and when writing code with others.

Here’s two snippets from Google’s own C++ Style guide for their engineers.

**“**Be consistent with existing code

Using one style consistently through our codebase lets us focus on other (more important) issues. Consistency also allows for automation: tools that format your code or adjust your #includes only work properly when your code is consistent with the expectations of the tooling. In many cases, rules that are attributed to "Be Consistent" boil down to "Just pick one and stop worrying about it"; the potential value of allowing flexibility on these points is outweighed by the cost of having people argue over them.**”**

**“**Optimize for the reader, not the writer

Our codebase (and most individual components submitted to it) is expected to continue for quite some time. As a result, more time will be spent reading most of our code than writing it. We explicitly choose to optimize for the experience of our average software engineer reading, maintaining, and debugging code in our codebase rather than ease when writing said code. "Leave a trace for the reader" is a particularly common sub-point of this principle: When something surprising or unusual is happening in a snippet of code (for example, transfer of pointer ownership), leaving textual hints for the reader at the point of use is valuable (std::unique\_ptr demonstrates the ownership transfer unambiguously at the call site). **”**

Essentially, the take away is this:

1. Consistency in coding style leads to less arguments about semantics, and style, or other time wasted that could (and should) be spent worrying about more important aspects of your code (especially when working with a group)
2. Code comments and actions should be helpful to you as a programmer and to anyone reading your code. When working with a team, if someone on your team cannot read or understand your code, it becomes problematic. It is equally problematic if you cannot read or understand your own old code when you go back to review it. Stylizing (through comments and such) your code in a way that is helpful to the reader also makes it easier for them to merge their code with yours (when working on a project together) and for others to help you debug your code.

Overall, properly styled code is more readable code. It is easier for you to work with, easier for others to work with, and easier to use as a reference point for your own learning at a later date. If your code is poorly styled, it can serve as an unnecessary headache when you open and review it when you study.

As a final introductory note, the actual guide itself begins on the next page. I’ve written it so that it will be useful to you throughout your classes at Moorpark, and as a foundation for your coding style beyond this college.

**Commenting**

Comment well. Make sure that your remarks quickly explain what the block of code is and gives a brief overview of its functionality without overuse of programmer semantics. Use multi-line comments. Single line comments are best used only if the comment is actually a single line or two.

Below is an example of what this might look like.

/\* Handles the creation of the background for our main menu.

It sets the size of the menu and then attempts to load the background image. \*/

void MainMenuState::initBackground()

{

this->background.setSize(

sf::Vector2f // vector to float

(

static\_cast<float>(this->window->getSize().x),

static\_cast<float>(this->window->getSize().y)

)

);

if (!this->backgroundTexture.loadFromFile("Resources/Images/Backgrounds/bg1.png"))

{

throw "ERROR::MAIN\_MENU\_STATE::FAILED\_TO\_LOAD\_BACKGROUND\_TEXTURE";

}

this->background.setTexture(&this->backgroundTexture);

}

Comments are generally good as a reminder of the intended purpose of a section of code. It should not deal with the specifics of how a code is run. If you use an *if statement* in your code, for example, you do not need to mention that if statement in your comment. However, if a particular part of your code is confusing, feel free to add a single line comment next to it for clarity.

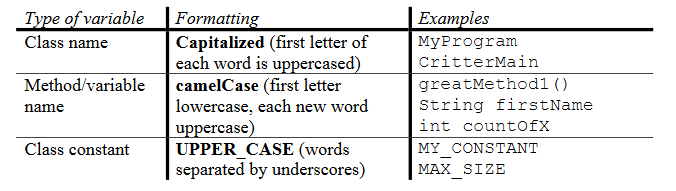
Take a look at the Python example below. For the purposes of my own learning and clarification, I noted what the statement did so that I could clearly understand it if I opened the file at a later date. (The comment is on two lines because of Microsoft Word)

# POST /store data: {name:}

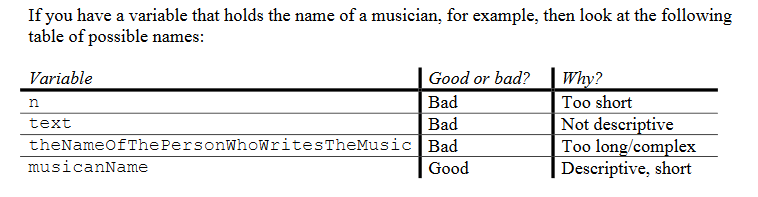
@app.route('/store', methods=['POST']) # makes it only accessible via POST requests

**Naming Conventions**

In general, I follow this table as a good reference on how to name things. The table below is borrowed from the University of Washington, where the professors prefer camelCase over using\_underscores.

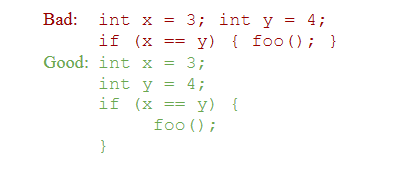


It’s important to name your variables and methods/functions well. This leads to less confusion while you’re writing your code, which means less mistakes and less time wasted trying to decipher your own writing. Try to be both descriptive and concise. In basic loops, and if the line of code would otherwise be too long, short names are preferred. At all other times, generic names like *x,* *myFunction*, or *stuff*, should be avoided when better names would be suited. Also avoid relying on numbers to sequence and differentiate variable names, such as *playerVelocity1, playerVelocity2, playerVelocity3.* Single letter names are usually poor choices but are acceptable in niche situations like the *i counter* in a for loop. When abbreviating long words for naming, be sure that its still obvious what the full word would be. For example, *dX* is worse than *dirX* which could also maybe be replaced by *directionX* . Below is a table with more examples for clarification.



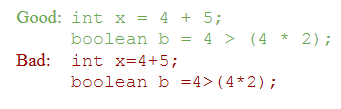
**Formatting**

* Keep lines of code under 100 or so characters, and preferably under 80 characters. This makes it both easier to read and to keep track of.
* Put comments for specific lines of code above the line it refers to or next to it if space allows. This ensures that the line of code plus the comment is not too long.
* Leave a blank line between functions or methods. This improves readability.
* Put only one statement on a line (unless declaring variables of the same type and you still have proper amounts of space to do so, and it makes complete sense)



Note: in C++ it is perfectly fine to declare multiple integers on one line.

* Leave spaces after EVERY OPERATOR. Please, please, please.



**Indenting Versus Curly Braces**

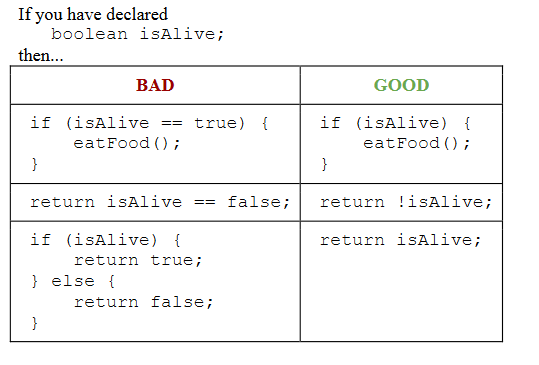
As you code, you’ll find that curly braces normally used for things like if statements can be simply replaced by proper indentation of code. If you prefer indentation over use of curly braces in this scenario, that is fine. However, please be consistent about your usage or non-usage of curly braces.

My personal preference is to use curly braces everywhere in languages where I am forced to (or simply learned to) use braces for functions and classes (C++, Java, JavaScript) and to forgo them completely otherwise (Python).

At the end of the day, do what is most comfortable for you.

**Boolean Zen**

Don’t rely on == or != if you can do without them. I prefer naming my Boolean statements almost like they are questions. This allows you to treat them like true or false statements. Remember that you can just use the ! to negate any Boolean. Here’s an example.



**Redundancy**

* Use methods/functions when they make sense. Don’t make them for things that are 1 or 2 lines of code. The creation of the function would already take up that many lines anyways.
* If you have a block of code within a function that is over 50 lines, consider if you can split it into multiple functions (without breaking the overall structure of your program)