COMP11 Style Guide
Tufts University
Fall 2017

Please review carefully and apply the style guidelines described below. We'll follow many of these guidelines to grade your homeworks and labs. This style guide was developed by COMP11 Teaching Assistants and instructors who are taking into consideration industry practice, academic practice, and expectations of high-level comp-sci courses such as COMP40. You'll get used to writing excellent code now and carry it forward through all your time at Tufts (and beyond)!

Table of Contents

I. Statements and Lines

C++ doesn’t care if you put multiple **statements** on a single line, but we do! Code like this is hard to read.

Bad:

```
int x = 5; cout << “Value of x is ” << x << endl;
```

Better:

```
int x = 5;
cout << “Value of x is ” << x << endl;
```

Remember that a **statement** in C++ is simply some line of code that ends in a semicolon, so it could be an assignment, a call to a function, or any number of interesting things.

II. Indentation and Spacing

Every time a new block is introduced, the code within the block should be indented by either 4 or 8 space characters.

IMPORTANT: DO NOT MIX FOUR AND EIGHT SPACE TABS

A couple of things to note:

- Notice the guideline for 4 or 8 **space characters**. We avoid literal tab characters (`‘\t’`) because the interpretation of this character when rendering output is not guaranteed to look one way or the other. Most editors have some way of specifying that tab characters should be replaced with spaces, and they further allow you to specify exactly how many characters.
- **Do not mix the two options.** If you mix these styles, you'll lose points. Period. The reasoning is consistency; nobody wants to look at code where some blocks are twice as deeply nested as others, it's really ugly and unreadable.

Using Emacs? Make a tab default to 8 spaces:

To fix your emacs configuration, run the following over ssh / on a Halligan machine:

```
/comp/11/bin/setup_emacs
```

And that's it! You should have pretty 8-space tabs from now on in Emacs.

III. Long Lines and Whitespace

All lines should be 80 characters or fewer including indents. All lines includes comments: anything you expect someone to read.

To understand this rule, try working on a computer in Halligan without maximizing the terminal window; that is, use it at its default size, something like 24 rows by 80 columns. If your lines are too long, your code will wrap in ways that are tough to read.

Different editors will interpret tabs in different ways (see *indentation and spacing*, above), so don't rely on the view of Emacs or Vim or Sublime to tell you if you're within the 80-column limit. On the terminal, the following command will tell you the length of the longest line in the file myfile.cpp. If it tells you 80 or less, you're good!

```
wc -L myfile.cpp
```

If you use emacs, you can type `M-x column-number-mode` (which can be abreviated M-x col<tab>), and emacs will tell you what column the cursor is in at the bottom of the screen.

You should strive to separate your code into logical sections using blank lines. For example, say you're writing a function that adds up all of the elements of an array (if you don't understand this code now you will soon!):

```
double sum_elems(double arr[], int length)
{
        // Local variable declarations
        double sum = 0;
        int i;

        // Loop through array, accumulate sum
        for (i = 0; i < length; ++i) {
                sum += arr[i];
        }


        // Return sum to the caller
        return sum;
}
```

See how readable this is? Using blank lines as separators, you can make it obvious how your code works, and more importantly, how you thought about and solved the problem. Good use of whitespace will save you and the people reading your code a number of headaches.

IV. Brace Placement and Spacing

There is less general agreement among C++ programmers about this than about other content in this guidebook, so we'll follow the style used in the Linux kernel and by the originators of the C programming language. If you use a different curly-brace placement style, that's OK as long as you're consistent.

For functions:

```
void myfunc(int arg1, bool arg2)
{
        // Body of myfunc
}
```

For all blocks that are not function bodies, the opening curly brace should be placed (after a space) on the same line as the start of the block.

For example:

```
      if (condition) {
            // do stuff if condition met
      }

      for (int i = 0; i < 10; ++i) {
            // do stuff on each loop iteration
      }

      while (condition) {
            // do stuff while condition is true
      }
```

Spacing: Always put a space after the following keywords `if, for, while, do` That is, write:

```
      if (condition)
```

Don't write:

```
      if(condition)
```

Along the same lines, put a space before and after binary operators such as +, -, *, /, %, =, ==, etc. Write:

```
      if (x == 3)
```

Don't write:

```
      if (x==3)
```

Finally, when either defining or calling a function, you should not put a space between the function name and opening parenthesis. Write this:

```
      void my_func();
```

Not this:

```
      void my_func ();
```

V. Naming Conventions

One of the most difficult parts of writing good, clean code is naming things well. It's hard to think of short, descriptive names for variables, functions, and classes. We want the names to describe what we're trying to represent without being too wordy. Here are some general guidelines:

- Variable names should be all lowercase, words separated by underscores. If your name looks ridiculously long, it probably is. Shorten where appropriate.
- Function names should follow the same rules as variable names, and should not include information about the return type. That's built in.
- Class names should start with capital letters and be single words, if possible. If this is impossible, do your best to get there or use shorthand. CamelCase is used if there are mulitple words.
- Reserve the use of vague names for loop counters and other bits of code that don't have much to do with the real work of your program. That is, use `i` for a loop counter instead of a variable called `loop_counter`.

We can't give much advice as to how to come up with great names. This comes with practice!

VI. Functions

As programmers, we care about modularity and abstraction. We strive to write code that can be reused to solve other problems and that hides unimportant details from the people using our code. This first point should immediately make you think of functions; we define small sub-operations that take some small, easy to solve problem and solves it in a nice little package that we can use wherever we need it with the right inputs.

There are some style guidelines that actually help us adhere to principles of abstraction and modularity more closely:

- Functions should have a single purpose.
- Functions should have at most about 5-10 local variables.
- Functions should be short; ideally less than or equal to 30 lines in length. They should ideally fit on one of those 24-line screens you get when you open a terminal on the Halligan machines.

Shorter functions are generally simpler; they have a singular purpose and don't do a million things before. Having a single purpose forces you to think harder about reusing code, and when it is appropriate to separate the work into more than one function. Having a huge number of variables in your function usually indicates you've added too much complexity, and should consider writing another function or two. Another indication that your funtion is too complex is that there are more than 3 or 4 levels of indentation.

When you're writing your functions in your .cpp files, makes sure to separate the definitions by a blank line, like this:

```
void first_function(int x)
{
        return 3 * x;
}

void second_function(double y)
{
        return y * y;
}
```

This, as described earlier, helps you keep logical blocks of code separated.

VII. Commenting

Reading code alone is not something we can do super well until we've spent months (if not years) reading it and know most of the tricks of the trade, and even then someone is likely to have used some feature or pun that you weren't aware of. This is why we need good comments; comments let you as a programmer express some of the thought process that went into your code.

In general, comments should be useful to the person reading your code. They should occur frequently enough in your code to be informative, but not so frequently as to be overwhelming. This can take some practice! Some examples of places comments belong:

- Above variable declarations (one comment for a group of variable declarations suffices).
- Above a calculation, especially if it's complex and/or unclear why it's needed
- Above function definitions
- Above a `for` loop or `while` loop (usually unnecessary inside the loop)

What a comment should do:

DO NOT EXPLAIN WHAT THE CODE DOES. EXPLAIN HOW/WHY IT DOES IT.

You should explain how you do things when the "how" is unclear, and likewise for the "why". This will save you headaches when you revisit your project in two weeks, and it will save us headaches when we're trying to figure out whether or not your process actually works.

Be cautious, however, because over-commenting can obfuscate the real point of the program. You don't want to explain the "how" or "why" when it is extremely obvious.

All that being said, here are some general style rules:

- C / C++ offer two comment styles, single line and block. Mixing the two styles is generally considered bad practice, so pick the one you like and stick with it -- with the exception that, as you've seen in labs and homeworks, we sometimes use block comments at the top of a file and single-line comments elsewhere.
- Always comment before functions in the style described below. Doing so will guarantee that you have a good understanding of the function, how it works, and how it affects the state of your program.
- Your files should always contain "signature" comments at the beginning, just like you need to do for labs and homeworks.
- Feel free to use comments to mark sections of code that need work / fixing, but remove these comments once you work on / fix the code. If someone is reading your code and sees "TODO: Make it work", they're not going to know whether you actually fixed it or not! They might even assume not, and then use your program improperly.
- Along the same lines, if we provide you with code and we have comments that say things like "TODO" or "YOUR CODE HERE", make sure to get rid of these before you submit. A good rule is to get rid of them as you implement the things described by the comments.

Here is an example function definition / comment:

```
// void print_string(string str)
// Purpose:    Print the value of str to standard output
// Parameters: The string, str, to print
// Returns:    None
// Effects:    Prints to standard output
// Notes:      None
void print_string(string str)
{
      cout << str << "\n";
}
```