CS106B Style Guide

Written by Chris Piech and Keith Schwarz

Those of you who took CS106A are probably aware of the emphasis this course will place on style and design. From our perspective, it's just as important that you write clean, elegant, readable code as it is that you write efficient and powerful code. Code that's difficult to read or write greatly increases the chance that you'll encounter bugs and makes those bugs much harder to fix and track down when they occur. On the other hand, well-written code can be comparatively easy to fix and debug, and the time you invest in developing a good programming style will pay for itself several times over.

This handout is a refresher on good programming style. It's not an exhaustive list of the elements of good style, and over the course of the quarter you and your section leader will work together to help you hone your style savvy. Think of this list as a friendly reminder and recap of the major style lessons that you've learned either in CS106A or on your own time.

Program Using Stepwise Refinement

Stepwise refinement, also known as decomposition, is the process of solving a large problem by breaking the problem down into manageable, logical chunks. Stepwise refinement is the most fundamental cornerstones of good style—programs that are formulated in a simple, elegant manner are easy to write, easy to debug, easy to read and easy to maintain. Stepwise refinement is not aesthetic icing on the cake. It is not a process you apply to your code after you've finished writing it. Rather, it should be the approach you should take to solving the problem as you go.

Large programming problems can be overwhelming if you try and think of all the minute challenges that you will face simultaneously. When presented with a problem, try to solve the high level challenges first and allow yourself to call substantial helper methods or functions. Assuming that the helper methods are written correctly can you solve the high level problem? If so you have reduced the size and scope of your program to filling in helper methods—and when writing the helper methods feel free to continue to utilize slightly lower level functions. It allows you to write programs little piece at a time and the short easy to grasp methods that are the result of stepwise refinement are stylistically ideal.

Make Code Self-Documenting

Well-commented code is certainly easier to read than poorly-commented code, but comments alone are not necessarily a mark of good style. Consider, for example, this code snippet:

```
/* Function: Spork(int x, string& y);
 * Usage: Spork(137, myString);
 * ------------------------------------------------------------------
 * Truncates the string either after the first x characters or after
 * the first instance of a space character, whichever comes first.
 *
void Spork(int x, string& y) {
    int pizkwat = MumboJumbo(y);
    if (pizkwat < x)
        y = y.substr(0, pizkwat);
}
```

From the comments in this code it's somewhat easy to figure out what's going on, but that doesn't mean that the code is easy to read. The variable and function names are completely arbitrary and don't communicate at all how the code works. Without the comments, this code would be entirely inscrutable. Although the comments help, it still doesn't mean that the code is necessarily easy to work with. In a debugger window, would you be able to differentiate between the **`pizkwat`** and **`y`** variables? Or remember what the call to **`MumboJumbo`** does?

A much cleaner way of writing this code would be something like this:

```
/* Function: breakWordAtSpace(int maxChars, string& toChop);
 * Usage: breakWordAtSpace(137, myString);
 * ------------------------------------------------------------------
 * Truncates the string either after the first maxChars characters or
 * after the first instance of a space character, whichever comes
 * first.
 *
void breakWordAtSpace(int maxChars, string& toChop) {
    int firstSpace = FindFirstSpaceIn(toChop);
    if (firstSpace < maxChars)
        toChop = toChop.substr(0, firstSpace);
    else
        toChop = toChop.substr(0, maxChars);
}
```

This code is significantly easier to read and maintain, as even without the comments it should be possible to determine more or less what's going on.

What sorts of practices can help make code self-documenting? As the above example should hopefully illustrate, one of the major style points to keep in mind is developing a good, consistent naming conventions for functions and variables. Function names should contain a verb (since functions do things) and should be descriptive. They should be more than one word long. Variable names should be descriptive of the value(s) they contain and they should be more than one letter.

When developing a good naming convention, it is also a good idea to have separate conventions for functions, variables, constants, and types. Often, constants will be in ALL_CAPS with underscores to make them visually distinct from other variables.

Functions usually use camelCaseNames. Structs and classes tend to have ProperCapitalization. Again, you are free to choose whatever naming convention you want, but *make sure to be consistent!*

Use Indentation and Braces Correctly and Consistently

Incorrect indentation or bracket use is one of the fastest ways to make your code unreadable. Code blocks give visual cues about program flow and variable scope. The most important rule is that code within a nested code block—whether the code block is a conditional statement, loop, function, etc—should be indented one level deeper than code in the parent code block.

There is some disagreement as to exactly what spacing and indentation scheme to use—for example programmers disagree as to whether the code block starting brace `{` should be placed on its own line. It doesn't matter if you put your starting braces on their own line or not, but it does matter that within a project your code is consistently and hierarchically indented.

For example, here are two perfectly fine ways of writing the same function. Both examples have three nested code blocks; the function, the for loop and the if statement:

```
void myFunction(int max) {
  for(int i = 0; i < max; i++) {
     if(isPrime(i)) {
        cout << i << endl;
     }
   }
}
```

```
void myFunction(int max)
{
  for(int i = 0; i < max; i++)
  {
    if(isPrime(i))
    {
      cout << i << endl;
    }
  }
}
```

C++ allows for you to write single-line code blocks without surrounding the code block with braces. This is generally acceptable to utilize if you are writing a condition code block and both the condition test and the body of the condition fit on one line. If your code does not fit the above description remember the Orthodontist Rule: You always need braces!

Function, Inline, & Program Commenting

The great Greek philosopher Aristotle is famous for his discussion of the "golden mean." According to Aristotle, every virtue has two vices, one of excess and one of deficiency. For example, it's perfectly fine and necessary to eat meals, but eating too much food or too little food can cause health problems. Eating just the right amount hits this golden mean and is a perfect harmonious balance.

Were Aristotle to become a computer programmer, I guarantee that he would have said

the same thing about commenting. Having either too much commenting or too little commenting can significantly detract from a program.

The best comments focus on what the code aims to do, what the assumptions are, and how edge cases are handled, rather than the mechanics of what the code does. Consequently, if you find yourself looking over the code wondering what it does, you can look at the comments to recover the intuition behind what's being accomplished.

For example, here's a particularly good function comment:

```
/* Function: string LongerString(string one, string two);
 * Usage: cout << LongerString("Hi!", "Bye!") << endl; // Prints "Bye!"
 *        cout << LongerString("A", "B") << endl;      // Prints "A"
 * ------------------------------------------------------------------
 * Given two strings "one" and "two", returns the longer of the two
 * strings.  If the two strings have equal length, the first string is
 * returned.  This function assumes that both strings have length >= 1.
 */
string LongerString(string one, string two);
```

This approach to commenting has four advantages. (1) First, it makes the code significantly easier to debug. If your program contains a bug that manifests at a particular point in the code, then the comments can make it easier to identify the root cause of the bug—did your code not do what you were trying to do? Did it break an assumption? (2) Second, it makes the code easier to edit in the future. If at some point you realize that the program you've written needs to do a bit more than what it was initially designed for, leaving helpful comments in makes it easier to make changes to the code. (3) Third, it allows someone who's never looked at the implementation of the function to use it correctly. (4) And finally, adding these types of comments force you as a programmer to stop and think about the design as you go. If you have the discipline to add comments to your code as you're writing it, then you'll be forced to describe to yourself exactly what it is that you're doing. Often, this can eliminate logic errors before they even manifest themselves.

Avoid Magic Numbers

If your program depends on constant values, instead of using raw numbers in your code you should use a **`const`** variable that articulates the meaning of the number. The general capitalization convention for constants is to make all characters upper case with words separated by an underscore. This helps differentiate them from variables and functions. Be aware, however, that the book only capitalizes the first letter of constants and the first letter of any new words, leaving all other letters lowercase.

For example if you were running an experiment 10 times, instead of using the raw number 10 in your code, it would be better style to declare a **`const`** variable:

```
const int NUM_ITERATIONS = 10;
```

Using the above **const** expression would allow you to refer to NUM_ITERATIONS in the loop which controlled your experiment. Constants make your code more readable and allow you to make safe, easy changes to your program—image a scenario where you wanted to modify the number of times you ran your experiment.

An alternative convention (that you should avoid) is to use a **#define** expression to avoid magic numbers. Pound define expressions are effectively precompiler search and replace operations. They can be very powerful but if there is a name collision with pound define expressions—a scenario which is hard to avoid in large projects—the resulting bugs can be very difficult to solve.

3. Redundant Boolean Expressions

Redundant boolean expressions should be avoided. For example, imagine you wanted to return whether or not a **bool** isWord has the value **true**. One way to accomplish this would be to write the following code:

```
if(isWord == true) {
    return true;
} else {
    return false;
}
```

However, since isWord already evaluates to either **true** or **false** you should use the more direct statement, **return** isWord;

4. Global Variables and Minimal Scope

You may have heard the C++ axiom never use global variables. Global variables in C++ are not the same as instance variables in Java and the habit of using globals can cause name collisions that are very hard to debug (the problems with globals are similar to the problems that can arise from unsafe pound defining).

Another problem that derives from the use of globals is that the wide scoping of the variable means that it takes a lot of work to identify what is causing a bug related to that variable. If a variable is accessible throughout an entire programs there are countless ways that its value could become corrupted—and this problem is substantially magnified when working on large programs.

This thought process—that globals are bad because their universal scope makes it hard to solve bugs—leads to a more general rule: All variables should have as small a scope as possible. If you only need a variable within a given code block, declaring it outside provides you with new ways to introduce unintended errors and makes it harder for a reader to understand how you intend to use that variable.