

1 Introduction

Good programming technique is not limited just to creating programs that work correctly and elegantly. More often than not, your code is given away somehow, whether you show it to a friend because you're proud of it or it's passed to another employee at work who took your job after you moved on. In any case, a clear and consistent coding style is required to facilitate the understanding your code.

2 Variables

Variables are the bread and butter of your program. They appear the most often and are the biggest contributor to the readability (or unreadability) of your program.

2.1 Naming Conventions

There are many standard conventions for naming variables. These include Hungarian, camelBack, and underbar delimiting. All of these conventions are perfectly acceptable as long as you're **consistent** about which convention you use. That is, if you start writing your program using camelBack and then switch to Hungarian notation mid-program, then you just defeated the purpose of using a convention at all and your code will be much less readable. Here are some examples of how these conventions work:

camelBack	Hungarian	underbar delimited
bool goAgain; int numIterations; double approxSolution;	bool bGoAgain; int iNumIterations; double dApproxSolution;	bool go_again; int num_iterations; double approx_solution;

All these conventions are more thoroughly documented on the web. But basically, camelBack uses a few words for its variables with first letter of the second, third, ..., n^{th} word capitalized while the first letter of the first word is not capitalized. Underbar delimiting uses all lower case while separating words using an underbar (-) character. Hungarian notation has special rules on how to name variables. They are well documented at [UMR's C++ Guide](#).

2.2 Global and const Variables

First off, non-const global variables are **highly discouraged**. They can produce very weird bugs in your program that are nearly impossible to trap and fix. They are admittedly sometimes necessary for unusual situations, but don't use them unless you absolutely have to! On the other hand, using a global variable that is a **const** is fine. For instance, if you're writing a program that shuffles a deck of cards, you may be tempted to write code such as:

```
int currCard;
for (currCard = 0; currCard < 52; currCard++)
```

But wait! What if somebody wants your shuffling program to shuffle two decks together? Then your program would only do half the job since the value 52 is hard coded in there. This creates a mess, especially if the assumption that a deck always contains 52 cards shows up in a lot of places. So, the code above can be modified to read something more like:

```
const int NUMCARDS = 104;    // two decks of cards
int currCard;
for (currCard = 0; currCard < NUMCARDS; currCard++)
```

Now you only need to change the value of the **const** variable **once**. The other thing to note is that all **const** variables should be in uppercase so they're easy to identify as a **const**.

3 Spacing

3.1 Text Editors & IDEs

UMR’s convention is to use two spaces for each indentation level. Many Integrated Development Environments (IDEs) and “smart” text editors try to handle indentation for you. Unfortunately, the default configuration of these programs won’t work. If you use these programs, be sure to consult the documentation on how to configure indentation. For `vim`, the following four lines can be added to your `~/.vimrc` file:

```
set cindent
set expandtab
set tabstop=2
set shiftwidth=2
```

Once these lines are added, `vim` will automatically handle tabs using two spaces for you ([Reference](#)).

3.2 Indentation

The general rule of indentation is to indent when either of these conditions are true:

1. You just wrote something that has an opening brace (e.g., `if`, `else`, `for`, `while`, `do...while`, etc.).
2. The line of code is greater than 75 characters long and is “bleeding” onto the next line.

Note that if applying rule 1, the opening and closing braces are not indented. Instead, the contents are indented while the braces remain at the previous level. For example:

```
if (answer < 42)
{
    guess = answer + 16;
}
```

3.3 Operator Spacing

When doing operations on your variables, don’t let them all run together. Space out your operators and variables; this makes your code much easier to read. For instance:

Avoid: `guess=answer+16;`
Better: `guess = answer + 16;`

4 Comments

Commenting is important to keep up with so other people can read your code easily. Many beginners think, “Let me just code and comment afterward.” Inevitably, they “forget” to comment and their code ends up being less readable. Don’t fall into this trap. Commenting beforehand can be an easy way to steer your program while maintaining readability.

4.1 Comment Types

There are two types of comments recognized by the compiler. C style comments and C++ style comments:

```
/* example C style comment */
// example C++ style comment
```

Either type of comment is fine. But, as with variable naming conventions, choose one style and stay with it. Consistency goes a long way in promoting code readability.

4.2 File Comments

At the beginning of each file, put your name, section, the assignment date, and a brief description of what the code in this file does. Here is the template for this as defined on [class web site](#):

```
// Name: <Your Name>
// Class and Section: CS 54, Section <n>
// Date: <Date>
// Purpose: <Statement of Purpose>
```

4.3 Pre Function Comments

At the beginning of each function, put a brief description of what the function does, what its inputs are, and what it returns. While formal pre/post conditions are not strictly required, all the information must be included in your comments. For example:

```
/* sqrt takes the square root of number and returns the answer, if it exists */
double sqrt(double number)
{
    ...
}
```

4.4 In Function Comments

Every 5 - 10 lines of code should have a one line comment briefly says what the next section of code does. You may find it helpful to write these comments **first** then use them as signposts to guide your coding. Here's a typical well-commented `main()` function:

```
int main()
{
    double area, base, height;

    /* ask the user for the parameters */
    cout << "What is the base of the triangle? ";
    cin >> base;
    cout << "What is the height of the triangle? ";
    cin >> height;

    /* find & report the area to the user */
    area = 0.5 * base * height;
    cout << "The area is " << area << endl;
    return 0;
}
```

5 Conclusion

Coding is about more than just writing a program that works or works efficiently. Coding is about writing code that is readable and accessible to other people so that they can better improve and enjoy your code just as you do. Writing clearer code will make you a greater asset to the software industry and make you more valued by employers since programming is often done on teams with several, dozens, and even hundreds of people. Writing easy-to-read code increases productivity and makes you a viable team player.

This document was written by Dylan McDonald using \LaTeX on Mac OS X. The latest version will always be on [the web](#). Any questions, comments, corrections, or clarifications can be sent to dlmyr8@umr.edu.