C

Coding Standard & Style Guide

for

CPTS 121

By
Jack R. Hagemeister and Andrew O'Fallon

School of Electrical Engineering and Computer Science
Washington State University

TOC

1. Introduction

Welcome to programming and the C language. As an introduction to programming and C family of programming languages there are several educational goals for CPTS121. These include:

Master the syntax of a standard programming language
Develop good programming skills and practices that will make this task easier.
Accumulate design skills that assist you in problem solving and creating software solutions.
Build up your understanding and ability with programming; start filling your programmer's toolbox.

2. Purpose

This coding standard and guideline have been written up for several reasons which should help writing high quality code that is easy to understand and develop. We will apply code reviews to validate code quality, so it is important that all students use the same style of coding. Style in this sense means using common constructs, writing proper documentation and code comments, and organizing code to a common layout.

Although complying with coding guidelines may seem to appear as unwanted overhead or limit creativity, this approach has already proven its value for many years in industry and here.

Additional goals include:
* Preventing common mistakes and pitfalls.
* Preventing language constructs that are less comprehensive.
* Promoting good design
* Improving readability and extensibility of the code.
* Facilitate the ease of learning to program
* Help you develop superior coding skills and habits.

This standard is a summary of known good habits and industrially accepted practices. It is a simplified version for Academic use.

You Must Remember:
Commenting and code layout is used as a mechanism (maybe the only reliable one) for the code writer (author) to effectively communicate purpose, functionality, logic, intent, decision processes, and motivation for any piece of code to the code reader. Anytime there is ambiguity, incompleteness, or obfuscation in the code there is the opportunity for mistakes and misunderstanding to occur. We hope to teach you how to minimize these types of problems and in that effort also help you learn to be a great software developer.

3. Naming Conventions of programmer defined identifiers

The naming and style conventions used can greatly enhance or reduce the ease with which a programmer can read and understand a piece of source code. It is amazing how even the simplest function or code block can be made completely obtuse by poor and thoughtless selection of names. Developing good naming habits early will help you write better code and solve problems faster and easier.

3.1. General

The selection of identifiers is crucial to understanding. Meaningful names should be selected for all identifiers. They should communicate the purpose and use in context with the problem being solved. In general, this means that you should use complete words in the construction of identifiers. For example, square is preferable to sq. Single letter identifiers should be avoided. The only

exception would be loop control variables that have NO OTHER PURPOSE. Below are further examples of good and bad identifier names.

Valid Identifiers in C:
Valid identifiers in C are a sequence of one or more letters, digits or underscore characters (_). Neither spaces nor punctuation marks or symbols can be part of an identifier. Only letters, digits and single underscore characters are valid. In addition, variable identifiers always have to begin with a letter. They can also begin with an underline character '_', but in some cases these may be reserved for compiler specific keywords or external identifiers, as well as identifiers containing two successive underscore characters anywhere. In no case they can begin with a digit.
Another rule that you have to consider when inventing your own identifiers is that they cannot match any keyword of the C language or your compiler's specific ones, which are reserved keywords.

3.1.1. Programmer defined type names

	Names for user defined types will be in mixed case alpha-numeric. They will start with an uppercase letter. Each word in the identifier will be uppercase.
Example	`UserName, Student, Line, SavingsAccount`
Justification	This is the dominant method for naming types and entities in all of the C like languages. It uniquely labels the name as a user defined type.

3.1.2. Variable names

	Variable names are mixed case alpha-numeric and will start with a lowercase letter. Each subsequent word in the identifier will be uppercase. In C the variables
Example	`userName, studentList, baseLine, savingsAccountBalance`
Justification	Common practice in the C development community. Makes variables easy to distinguish from types, and effectively resolves potential naming collision as in the declaration `Line line;`

3.1.3. Named Constants

	Named constants and enumeration values are uppercase alpha-numeric with the underscore separating words.
Example	`MAX ITERATIONS, COLOR RED, PI`
Justification	This is the dominant method for naming constants in all C like languages. It uniquely labels the name as a defined constant.

3.1.4. Function Names

	Names representing functions must be verbs and written in mixed case starting with lower case.
Example	`getName(), computeTotalWidth()`
Justification	This is the dominant method for naming types and entities in all of the C like languages. It uniquely labels the name as a user defined variable.

3.1.5. Abbreviations and acronyms

	Abbreviations and acronyms must not be uppercase when used as name [

Example	`exportHtmlSource(); // NOT: exportHTMLSource();`
	`openDvdPlayer(); // NOT: openDVDPlayer();`

Justification	Using all uppercase for the base name will give conflicts with the naming

	conventions given above. A variable of this type whould have to be named dVD, hTML etc. which obviously is not very readable. Another problem is illustrated in the examples above; When the name is connected to another, the readbility is seriously reduced; the word following the abbreviation does not stand out as it should.

3.1.6. English Language

	All names should be written in English.
Example	`fileName;  // NOT: filNavn`
Justification	English is the preferred language for international development.

3.2. Specific Recommendations

3.2.1. Name compute

	The term compute can be used in functuins where something is computed.
Example	`computeAverage();` `computeInverse()`
Justification	Give the reader the immediate clue that this is a potential time consuming operation, and if used repeatedly, he might consider caching the result. Consistent use of the term enhances readability.

3.2.2. Name find

	The term find can be used in funcitons where something is looked up.
Example	`findNearestVertex();` `findMinimumElement();`
Justification	Give the reader the immediate clue that this is a simple look up method with a minimum of computations involved. Consistent use of the term enhances readability.

3.2.3. Name initialize

	The term initialize can be used where an object or a concept is established.
Example	`initializeFontSet();`
Justification	The American initialize should be preferred over the English initialise. Abbreviation **init** should be avoided.

3.2.4. Name is

	The prefix is should be used for flag (Boolean) variables and predicate functions.
Example	`isSet, isVisible, isFinished, isFound, isOpen` `isEven(), isOdd(), isPrime()`
Justification	Common practice in the C development community and partially enforced in Java. Using the “is” prefix solves a common problem of choosing bad boolean names like status or flag. isStatus or isFlag simply doesn't fit, and the programmer is forced to choose more meaningful names. There are a few alternatives to the “is” prefix that fits better in some situations. These are the has, can and should prefixes: `int hasLicense();`

	`int canEvaluate();` `int shouldSort();`

3.2.5. Names that are written as plural

	Plural form should be used on names representing a collection of objects.
Example	`int values[]; // array names`
Justification	Enhances readability since the name gives the user an immediate clue of the type of the variable and the operations that can be performed on its elements.

3.2.6. Iteration

	Iteratoration variables should be called i, j, k, m, n

Example	`for (int i = 0; i < numberOfTables); ++i)`
	`{`
	`:`
	`}`

Justification	The notation is taken from mathematics where it is an established convention for indicating iterators. Variables named j, k etc. should be used for nested loops only.

3.2.7. Use of abbreviation

	Abbreviations in names should be avoided.
Example	`computeAverage(); // NOT: compAvg();`

Justification	There are two types of words to consider. First, the common words listed in a language dictionary. These must **never** be abbreviated. Never write:
	`cmd` instead of `command`
	`cp` instead of `copy`
	`pt` instead of `point`
	`comp` instead of `compute`
	`init` instead of `initialize`
	etc.
	
	Then there are domain specific phrases that are more naturally known through their abbreviations/acronym. These phrases should be kept abbreviated. Never write:
	
	`HypertextMarkupLanguage` instead of `html`
	`CentralProcessingUnit` instead of `cpu`
	etc.

3.2.8. Function names

	Functions returning something should be named after what they return and procedures (void functions) after what they do.
Example	
Justification	Increase readability. Makes it clear what the unit should do and especially all the things it is not supposed to do. This again makes it easier to keep the code clean of side effects.

3.2.9. Enumeration constants

	Enumeration constants can be prefixed by a common type name.

Example	`enum Color`
	`{`
	`  COLOR_RED,`
	`  COLOR_GREEN,`
	`  COLOR_BLUE`
	`};`

Justification	The notation is taken from mathematics where it is an established convention for indicating iterators. Variables named j, k etc. should be used for nested loops only.

3.2.10. Negated boolean variable names

	Negated boolean variable names must be avoided.

Example	`int isError; // NOT: isNoError`
	`int isFound; // NOT: isNotFound`

Justification	The problem arises when such a name is used in conjunction with the logical negation operator as this is a double negative. It is not immediately apparent what `!isNotFound` means.

4. Statements

4.1. Types

4.1.1. Type Conversion

	Type conversions must always be done explicitly. Never rely on implicit type conversion.

Example	`int x;`
	`float y = 4.3;`
	`x = (int) y;  // explicit type conversion.`

Justification	By this, the programmer indicates that he is aware of the different types involved and that the mix and conversion is intentional.

4.2. Variables

4.2.1. Initialization

	Variables should be initialized where they are declared.
Example	`float y = 4.3;`
Justification	This ensures that variables are valid at any time. Sometimes it is impossible to initialize a variable to a valid value where it is declared: In these cases it should be initialized to a non-valid value.

4.2.2. Dual Meaning

	Variables must never have dual meaning.
Example	

Justification	Enhance readability by ensuring all concepts are represented uniquely. Reduce chance of error by side effects.

4.2.3. Global Variables

	Use of global variables should be minimized.
Example	
Justification	In C, there is no reason global variables need to be used at all. The same is true for global functions or file scope (static) variables.

4.2.4. Pointer Declaration

	C pointers have their reference symbol next to the type rather than to the name.
Example	`float* x; // NOT: float *x; n`
Justification	The pointer-ness of a variable is a property of the type rather than the name.

4.2.5. Implicit test for 0

	Implicit test for 0 should not be used other than for boolean variables and pointers.
Example	**`if (nLines != 0)  // NOT: if (nLines)`** **`if (value != 0.0) // NOT: if (value)`**
Justification	By using explicit test the statement give immediate clue of the type being tested. It is common also to suggest that pointers shouldn't test implicit for 0 either, i.e. if (line == 0) instead of if (line).

4.3. Loops

4.3.1. Control

	Only loop control statements must be included in the for() construction.
Example	**`sum = 0;`** **`for (i = 0; i < 100; i++)`** **`{`** **`  sum += value[i];`** **`}`** **`// NOT: for (i = 0, sum = 0; i < 100; i++)`** **`// sum += value[i];`**
Justification	This ensures that variables are valid at any time. Sometimes it is impossible to initialize a variable to a valid value where it is declared: In these cases it should be initialized to a non-valid value.

4.3.2. Loop control Variable

	. Loop variables should be initialized immediately before the loop.
Example	**`isDone = FALSE;`** **`while (!isDone)`** **`{`** **`  :`** **`}`**
Justification	

4.3.3. break and continue in loops

	These statements should not be used.

Example	
Justification	

4.4. Conditionals

4.4.1. Complex Expressions

	Complex conditional expressions must be avoided.

Example	`if ( year % 400 == 0 )`
	`{`
	`  isLeap = TRUE;`
	`}`
	`else if ( year % 100 == 0 )`
	`{`
	`  isLeap = FALSE;`
	`}`
	`else if ( year % 4 == 0 )`
	`{`
	`  isLeap = TRUE`
	`}`
	`else`
	`{`
	`  isLeap = FALSE;`
	`}`
	
	`/* Not:`
	`if( year%400 ==0 || ( year%100 != 0 && year%4 == 0 ) )`
	`  {`
	`  isLeap = TRUE;`
	`  }`
	`  else`
	`  {`
	`  isLeap = FALSE;`
	`  }`
	`*/`

Justification	Simple code is always better.

4.4.2. Nominal case

	The nominal case should be put in the if-part and the exception in the else-part of an if statement.

Example	`isOk = readFile (fileName);`
	`if (isOk)`
	`{`
	`  :`
	`}`
	`else`
	`{`
	`  :`
	`}`

Justification	Makes sure that the exceptions don't obscure the normal path of execution. This is important for both the readability and performance.

4.4.3. Complex Expressions

	Complex conditional expressions must be avoided.
Example	
Justification	

Header files are named with extension .h and source files with extension .c This is expected by the compiler.

5. Coding style, white space, code formatting

5.1. Layout

5.1.1. Indentation

	Indentation should be 3 spaces, tabs should be replaced by spaces.

Example	`for (i = 0; i < nElements; i++)`
	`{`
	`  a[i] = 0;`
	`}`

Justification	Improves blocking and is easily accomplished by sets in the IDE.

5.1.2. if - else

	The if-else class of statements should have the following form:

Example	`if (condition)`
	`{`
	`  statements;`
	`}`
	
	`if (condition)`
	`{`
	`  statements;`
	`}`
	`else`
	`{`
	`  statements;`
	`}`
	
	`if (condition)`
	`{`
	`  statements;`
	`}`
	`else if (condition)`
	`{`
	`  statements;`
	`}`
	`else`
	`{`
	`  statements;`
	`}`

Justification	

5.1.3. for

	A for statement should have the following form:
Example	`for ( initialization; condition; update )`

	`{`
	`  statements;`
	`}`

Justification	

5.1.4. while

	A while statement should have the following form:

Example	`while ( condition )`
	`{`
	`  statements;`
	`}`

Justification	

5.1.5. do while

	A do-while statement should have the following form:

Example	`do`
	`{`
	`  statements;`
	`} while (condition);`

Justification	

5.1.6. switch

	A switch statement should have the following form:

Example	`switch (condition)`
	`{`
	`  case ABC :`
	`  statements;`
	`  // Fallthrough`
	
	`  case DEF :`
	`  statements;`
	`  break;`
	
	`  case XYZ :`
	`  statements;`
	`  break;`
	
	`  default :`
	`  statements;`
	`  break;`
	`}`

Justification	

5.1.7. function definitions

	A switch statement should have the following form:

Example	`int myFunction( float )`
	`{`
	`  :`
	`}`

5.2. White space

5.2.1. In lines

	- Conventional operators should be surrounded by a space character. - C reserved words should be followed by a white space. - Commas should be followed by a white space. - Colons should be surrounded by white space. - Semicolons in for statements should be followed by a space character.

Example	`a = (b + c) * d; // NOT: a=(b+c)*d`
	
	`while (true)  // NOT: while(true)`
	`{`
	`  ...`
	`}`
	
	`doSomething(a, b, c, d);  // NOT: doSomething(a,b,c,d);`
	
	`case 100 :  // NOT: case 100:`
	
	`for (i = 0; i < 10; i++)  // NOT: for(i=0;i<10;i++)`
	`{`
	`  ...`

5.2.2. Logical code sections.

Logical units within a block should be separated by one blank line.
Functions should be separated by three blank lines.
Variables and initialization should be line left aligned.
Use alignment when ever it enhances readability (makes the code pretty).

6. Commenting, writing useful comments

While the importance of properly documenting work cannot be over emphasized, it is important to properly comment code in order to get the expected benefits. First of all, realize that a comment is no different than any other programming statement. Once it is created, it has to be maintained and since a comment is not a functional part of the code, it is the last thing to be changed, if changed at all. Comments are not inherently good and if incorrect can be detrimental.

When the time comes to document your code (before you write it) try to answer the following questions:

What does the code do?
In general, nothing documents what code does better than the code itself. Therefore, be careful when adding comments that describe what the code does. These types of comments can be helpful when the code is difficult to read or understand. But if the code is difficult to read or understand you should also think about the possibility that it should be rewritten in a clearer more readable way.
Also, these types of comments are subject to becoming obsolete quickly as the code changes with time.

Why was the code written?
This is probably the most useful type of comment. Every piece of code has a purpose and every piece of code should have a comment that explains that purpose. Sometimes the purpose is due to a business need, in other times the purpose is purely technical.
Typically, the purpose of a routine is constant no matter how many different ways are used to implement the purpose. This makes these types of comments much more stable.

If a routine has multiple purposes it is a good candidate for splitting into separate routines.

Who wrote the code?
There is no better way to figure out what code is doing by talking to the dolt, i.e. the original programmer that wrote it. So, if you put your initials on a block of code you change or modify it may be helpful. Of course, if it is the gnarliest code you ever wrote, put your manager's initials on it!

When was the code written?
Add a date stamp to your comments/modifications. Perhaps also insert some reference to a specification, change order, change request number in a bug tracking system. The reference will allow someone to locate further info about possible discussions or specs about your code.

How does the code work?
This is similar but slightly different and more useful than the "What" type of comment. In this type you are trying explain in a general sense how the code meets the purpose of the routine. These types of comment only need to change when a different approach to implementation is tried.

Which other parts of the application calls this function?
It can be very difficult to track application flow when going through source code that does not document from where and how it is called.

How to invoke this routine; what do parameters stand for?
I like to see a line or two at the top of a function identifying how and why this routine gets called. If there are parameters, it tells what they mean, whether or not they are optional, and what their defaults might be. *And what it returns!*

Who made the change? Why was a change made?
I like information that indicates who made a change to the code, when it was made, and more important...why it was changed.

There is a lot to be said for self-documenting code. The idea being that you are naming funcitons by what they do, variables by what they hold. Also, writing top to bottom without a-lot of entry and exit points. If you find that you cannot read through code and know what it is doing, it needs to be re-written.

A critical element to self-documenting code is concise, single concept functions which, as mentioned above, make use of intelligently named variables. A function which needs to combine concepts should have each concept broken out into a separate function.

6.1.1. All comments should be in English.

6.1.2. File and function header comments follow the convention in Appendix A.

7. Files, file organization, and unit testing

7.1. Source Files

7.1.1. File naming

Header files are named with extension .h and source files with extension .c This is expected by the compiler.

7.1.2. Declaration and Definition

All declarations, functions, types, constants, will be located in a .h file.
All definitions will be in a matching .c file.
The header files should declare an interface; the source file should implement it. When looking for an implementation, the programmer should always know that it is found in the source file.

7.1.3. Header files must contain include guard code.

```
#ifndef NAME_H
#define NAME_H
  :
#endif
```

The construction is to avoid compilation errors. The name convention resembles the location of the file inside the source tree and prevents naming conflicts.

7.1.4. Include statements should be sorted and grouped.

```
#include <stidio.h>
#include <time.h>

#include "com/company/ui/PropertiesDialog.h"
#include "com/company/ui/MainWindow.h"
```

In addition to show the reader the individual include files, it also give an immediate clue about the modules that are involved. Include file paths must never be absolute. Compiler directives should instead be used to indicate root directories for includes.

7.2. File organization

Files should be organized in .h .c pairs as outlined above. Each file should group a logically related part of the project, such as I/O functionality, utility functions, drawing functions, etc. Each file must have a File Header Comment block.

7.3. Unit Testing

The unit testing drive code can be in one .h .c pair. Each function of the project should have a unit test that can exercise it in isolation of the working project. The testing operation should be automated such that any component or collection of components can be tested and verified.