C and C++ Coding Guidelines

Version 1.2

All code must follow best practices. Part (but not all) of this is adhering to the following guidelines:

Development

For code development, I recommend the following these steps strictly in this order. Make sure to complete each step fully before continuing with the next step.

1. Design your approach on paper, including class structure/relationships.
2. Implement the skeleton of your structures/relationships, including all methods, major attributes, etc. Your methods should already be documented, your attributes should already have comments, etc. Do not add any implementation to the methods. Your code should compile. (e.g., if a method returns an object, simply return null to make the code compile).
3. Fully implement testing based on the specification. For each element of the specification, write a test. Your test should be complete and compilable/executable. Of course, they will mostly fail because your classes have no implementation.
4. Inside the various unimplemented methods, add comments for the implementation you plan to do.
5. Fill in the implementation between your comments.
6. Run tests and fix broken code.

Commenting

1. Add the following to the beginning of all of your source files:

```
/*********************************************
 *
 * Author:       <Your name>
 * Assignment:   <Assignment name>
 * Class:        <CSI class name>
 *
 *********************************************/
```

2. Comment any C/C++ structs/classes.
3. Comment any methods.
4. Individually comment member variables and class constants.
5. Obvious/obfuscated comments are useless. Do not use them.
6. Properly (but reasonably) comment your code. A developer should be able to get a general idea of what's going on by just reading comments (and no code).
7. Check your comments for spelling and grammatical errors.

Coding

1. Do not use tabs.
2. Always use braces for code blocks, even for a single line of code. For example,
   ```
   if (true) {
     printf("True!");
   }
   ```
   The same rule applies for for, while, etc.
3. Include only necessary files.
4. Eliminate code replication.
5. Eliminate code replication.
6. Seriously, eliminate code replication.
7. Your code must compile on the ECS Linux machines. For C/C++, it must compile for gcc/g++. For Java, it must compile with the latest production release of Java on the first day of class.
8. Properly address all compiler warnings. Do not suppress/ignore compiler warnings unless **well** justified. Include your justification in a comment.
   At a minimum, compile with: -Wall -Wextra -Wpedantic -Wconversion
   Note well that the compiler is your friend. Look at the compiler warnings page; you may want even more warnings.
9. No spurious object creation or variable assignment.
10. For simple boolean methods, return directly from expression instead of using if.

    ```
    boolean empty() { // Yuck
      if (length == 0) {
        return true;
      } else {
        return false;
      }
    }

    boolean empty() { // Yep
      return (length == 0);
    }
    ```
    This avoids potential errors such as getting true/false returns backwards.
11. Do not use deprecated methods.
12. Always specify access (e.g., private, protected, etc). Use correct access.
13. No junk member variables. Member variables are for state related to object, not for variables used by several methods.
14. Don't write useless code
    a. Empty/autogenerated constructors - Why maintain the code?
    b. String blah = thing;
       return blah;
    c. if (done == true) // Use if (done)
15. Catch the most specific exception type.

16. Move all literal constants to variable constants except in really obvious situations.
```
if (size > 255) // Wrong
if (size > MAXSIZE)  // Great!
```
17. Do not violate class encapsulation.
18. Code should only print to console when appropriate. Inside a library is not an appropriate place to print to the console. Use logger if need to output in such cases. If you are printing to the console, print to the correct stream (stdout vs. stderr).
19. Test all system call return values and properly print complete errors (perror, fprintf, etc.). Make error messages as useful as possible. ("Parameters bad" vs "Usage: go <file> <date>").
20. Declare variables with use in the minimum scope. Do not predeclare at the function start. Predeclaration leads to overextended scope (entire function) and repeated initialization.
21. Remove all commented code.
22. Run code and runtime analysis tools to identify errors.
23. Make sure all numeric constant values in your code are justified? int[52] probably cannot. Why not 51? 53?
24. Do not use global variables unless absolute necessary. Make sure the explanation for needing global variables is clearly commented. Global constants are fine.
25. Insure the flow of the code is easily understandable.
26. Insure variable and method names meaningful.
27. Insure that you would want to be given this code for maintenance and modification.

Makefile

You should have a makefile to compile and link your code. Consider a program with `a.c`, `b.c`, and `b.h` source file where `a.c` includes `b.h`. Your makefile must

1. only recompile what is necessary. For example, if only `a.c` is changed, only `a.c` should recompile on the next make.
2. check all dependencies. For example, if `b.h` changes, `a.c` should recompile.
3. use Makefile variables to minimize repetition and maximize maintainability. For example, gcc options should not be repeated.
4. use compiler/linker options only when needed. For example, link options are not needed for compilation-only commands.
5. should include a clean target the deletes all executables and intermediaries.