Coding Standards

This document describes some of the local coding standards for use in this class. If you don't agree with the standard or don't understand what it says or why I think it's important, then please ask.

The standards are arranged by severity levels, where violation of "F" rules will result in the greatest loss of points and "C" rules the least.

1 Level-F errors: Constructs to Avoid

If you believe that you MUST disobey one of these rules, ask for permission.

1. Do not use `goto`.
2. Do not use global variables. Global constants are OK.
3. All class data members (and some functions) should be private.
4. Use C++ iostreams, not C stdio, for input and output.
5. Use only the 2017 standard C++ language – no proprietary functions or include files such as are found in Visual C++.
6. Your code must compile on the Zoo without *errors*.
7. Structure your source code appropriately into header files (`.hpp`) and implementation files (`.cpp`).

2 Level-D errors: Functionality

These are very basic rules.

1. Your code must compile on the Zoo without *warnings* when using the required `-O1 -g -Wall -std=c++17` compiler flags.
2. Insofar as possible, test every line of code in your program.
3. Check for I/O errors routinely, especially for files that do not open properly.
4. File system **path names** are not portable and should not be used in your program. To tell the compiler to look in non-standard places, use the `-I` and `-L` compiler flags. These flags should be put in your `Makefile` when needed. If for any reason you must write a path name in your program, put it in a `#define` or a `const` declaration at the top of `main` so that it can be found and changed easily.

3 Level-D errors: Indentation and Whitespace

Any time you take a job, you must learn the local coding standards and follow them. These are the standards for this class.

1. Use this style for indentation and brackets. Note that the opening `{` is at the end of one line, not at the start of the next. Note also the convention of leaving spaces around function arguments (as in `inRange( ageIn )`) but not around the Boolean expression that appears within the parentheses for `if` and `while` statements.

```
while (k < numItems) {
    cin >> ageIn;
    if (inRange( ageIn )) {
        age[k] = ageIn;
        k++;
    }
    else {
        cout << "An invalid age was entered, try again.");
    }
}
```

2. Set your indentation amount to 4 columns. Never let comments interrupt the flow of the code indentation.

3. Confine all code and comments to 80 columns. Break up long lines of code into parts at logical break points. **Do not let them wrap around** from the right side of the page to the left edge of the page on your listings.

4. Make your work look professional: clear, neat, and well-organized. Do not turn in work that is full of commented-out code.

5. Use whitespace to make the code easy to read. Use blank lines to break the code into paragraphs of related actions. Do not put a blank line between every line of your code. Generally put a blank space after a semicolon or around a binary operator, e.g., `x = y * -z;`, but it's okay to omit the spaces when it improves readability, e.g., `x*x + y*y`.

4 Level-C errors: Semantics and Organization

1. Simplicity is good; complexity is bad. If there are two ways to do something, the more straightforward or simpler way is preferred.
2. Clean up after yourself. If you allocate dynamic memory with `new`, free it with `delete`. Explicitly close all files that you have opened.
3. Learn to use the names of the various **zero-constants** appropriately. `nullptr` is a pointer to nowhere, `'\0'` is the null character, `""` is the empty string, `false` is a `bool` value, and 0 and 0.0 are numbers. Use the constant that is correct for the current context.
4. If a function is simple and fits entirely on **one line**, write it that way. Example:
   `bool squareSum( double x, double y ) { return x*x + y*y; }`

5. Otherwise, each function definition should **start with a whole-line comment** that forms a visual divider. If the function is nontrivial, a comment describing its purpose is often helpful. If there are preconditions or postconditions for the function, state them here.
6. Keep all functions short. With rare exceptions, a function should be no longer than one screen. Ideally, the function should be shorter than a dozen lines. Break up long or complex chunks of logic by defining more functions.

5 Level-C errors: Style Standards.

1. Please do not use `i`, `I`, `l`, or `O` as a variable name because `i` and `l` look like 1, and `O` looks like 0. Use `j`, `k`, `m`, or `n` instead.
2. Long, jointed names and short, meaningless names make code hard to read. Try for moderate-length names, 3 to 8 characters. Local variables and private class members should have short names because they are always used in a restricted context. Longer descriptive names are appropriate for non-local names such as global constants and class names.
3. Do not use the same name for two purposes (a class and a variable, for example). Do not use two names that differ only by an '`s`' on the end of one. Do not use two names that differ only by the case (for example `Object` and `object`).
4. `#include "tools.hpp"` in each module. Call `banner()` at the top of `main()`, call `bye()` at the end, and call `fatal()` to handle fatal errors until we introduce exceptions.