Coding Standards

Why have coding standards:

- To make code easy to read.
- To make code easy to debug.
- To make code easy to maintain
- To make your code look professional.

Some initial thoughts

- I will be presenting of coding standards from a subset of the standards I developed for 3 other companies.
- There can be a lot of angry arguments about standards.
- The standard here is oriented toward C/C++, but is applicable to other languages.
- Not all companies have formal coding standard, but present rules during code reviews.
- Old code is not usually modified after coding standards are introduced. Why?
- There are also tools like Doxygen for commenting code.
- For more ideas, see the following link shttps://google.github.io/styleguide/cppguide.html and https://www.linkedin.com/pulse/avoid-35-habits-lead-unmaintainable-code-christian-maioli-mackeprang
  - The first is Google's standards. They are massive and discuss all aspects of coding.
  - Second has ideas on making better code by showing you the wrong way to do things.
- Reminder: the assembler must conform to the coding standards.

The coding standards

The following is a cut down version of the types of standards that are required in industry. It is a good start. I have deleted all of the rules that are not applicable to your term project

1. Documentation / Comments

1. Each source file (.cpp and .h) that you create should have header information that describes its purpose. For a class with the declarations in the .h file and the definitions within the .cpp file, you only need header information for the .h file.
2. The code, that implementation each of your functions must be prefaced with the equivalent of a UNIX man page entry. The macro facilities in most editors can be used to create all of the straightforward elements of this prefix. (Microsoft supplies such a template for .NET applications.) The following is an example of the header format. I have chosen a representative function from a library that I created many years ago to support stock trading models. You will notice that there is an assumption of some knowledge about the software package.

```
/**/
/*
spmLongShort::ProcessNewOpens() spmLongShort::ProcessNewOpens()

NAME

    spmLongShort::ProcessNewOpens - processes new opens for this model.

SYNOPSIS

    bool spmLongShort::ProcessNewOpens( spmTrObj &a_obj, double a_capital
                                        , Jar::Date a_date );
      a_obj         --> the trading object to be opened.
      a_capital     --> the amount of capital to apply.
      a_date        --> the date we are processing in the simulation.

DESCRIPTION

    This function will attempt to open the trading object a_obj with the
    specified amount of capital. Before attempting the open, it will
    apply portfolio constraints. If any of the portfolio constraints are
    not met, this object will be opened as a phantom.  The constraint
    may also reduce the amount of capital to be applied.

    The status flags and phantom flag for the object will be set
    appropriately.

RETURNS

    Returns true if the open was successful and false if it was opened
    as a phantom.  One of these two cases will always occur.

AUTHOR  Not needed for software design.

    Victor Miller
```

DATE Not needed for software design

6:27pm 9/1/2001

```
*/
/**/
void
spmLongShort::ProcessNewOpens( spmTrObj &a_obj, double a_capital, Jar::Date
a_date )
{
```

The "/**/" will allow the function description to be parsed from the code for external documentation. There are products available to parse out comments and make a reference manual.

After the closing brace of the function implementation there should be a comment appended that contains the function prototype. For the above example, it would be:

```
/*void spmLongShort::ProcessNewOpens( spmTrObj &a_obj, double a_capital,
Jar::Date a_date ); */
```

This comment helps us to know the function we are viewing even if the beginning is not visible.

Note: If you are using C#, it automatically generates a template similar to the comment header that I described. So, for C# programmers, you may used this template, as long as you supply the same information.

3. **Doxygen may be used as an alternative to this format and will be accepted in our course. Just make sure that you use labels that correspond to this document. Doxygen can be integrated with Eclipse and Visual Studio.**
4. Comments within the code should be indented the same amount as the code.
5. Each block of code that performs a well-defined task should be preceded with a blank line and a comment. The comment should explain the code, not echo it. That is, it should supply useful information about the code. Some people, when required to write comments, write some pretty useless stuff and they write a lot of it.
6. If you have a library of related functions it should be enclosed in a namespace. If you have a library of related classes, they should also be enclosed in a namespace. This defends against name pollution.
7. With the exception of variables whose scope is extremely local, variable names must reflect their use. x, i, and n are **bad** choices for variable names.
8. Every enum, structure, class, const, prototype, define, etc. that is declared in an include file should be commented.
    1. Why do you use enums?
    2. Why use consts?

3. Why use define?

9. Comments should always match the code. Do not let documentation get stale. (This is an easy thing to happen and it often does happen.) That is, if you change the code, change the comments.

10. Don't wait until the end of a project to write comments. Once you get comfortable doing it, commenting code focuses your mind. Also, it is much harder to write meaningful comments after you complete your project.

11. Don't use comments to explain overly complex code. Rewrite the code. This happens often when people want to show off something that they learned.

12. For the sake of readability, there will be one tab setting for all programmers. A tab setting of 4 is typical. You can tell your editor to replace tabs by spaces for the sake of consistency in viewing your code.

2. Defines, Constants and Variables

1. In general, all general constants, enums, and macros that are used in a program should be declared in include file(s). It is recommended that the name of these constants and macros be written in all capital letters. Ideally these should be in a namespace or a class.

2. Variable names should begin with a lower case letter. Functions and classes should begin with an upper case letter. WHY?

3. Constants, enums, structs, and classes that are specific to a class should be placed in that class. This defends against namespace pollution and gives you the flexibility changing them is they are in the private part of the class.

4. A set of related constants should be defined using an enum class. (Note difference between enum and enum class) This allows type checking and allows the debugger to indicate the name of the constant rather than merely its value.

5. Use "const" rather than "define" to specify constants. This does type checking.

6. Inline functions should be used rather than macros. Inline functions are as efficient as macros with the plus of providing type checking. They are also safer. To be effective over a variety of .cpp files, the inline functions must be implemented in a .h file.

7. Avoid global variables. They provide many more opportunities for making global mistakes. If global variables are necessary, prefix them with "g_" so that they may be easily identified. The use of classes make global variables less necessary. (I feel hard pressed to justify them.) If they are necessary, group all global variables into a namespace or class. In this way we reduce name pollution and make global variables

easier to identify. Consider using static member variables in a class for global values.

8. All variables should be initialized before being used. Don't assume that they are set to zero. (They usually are not automatically set to zero in C++.) Even if you know that they are set to zero, do it anyway for the sake of documenting your project. It won't cost performance.

9. Variables that are members of a class should be "**m_**" as they first two characters of their name. This makes them identifiable within the code.

10. Unless there is a compelling reason, no variable should be declared in the public part of a class. If the user needs to view or change the value of the variable, there should be accessor functions provided.

11. Pick appropriate names for your variables. x is not a good choice for a variable name. Yes, I know I am repeating this point. It is important. This happens a lot.

12. Avoid using similar looking variable names in the same module.

13. Variables should be declared as close as possible to the point that they are used. This makes good sense and is generally accepted in the programming community despite the fact that some textbooks claim that they should be declared at the beginning of a function. Usually these textbook were written by people who are primarily C or Pascal (ARGH) programmers.

14. Do not let one variable of the same name step in front of another. You can do this because of scope considerations but **don't** do it. C# does not let you do this.

3. Functions

1. Prototypes for functions that are global should be in include files. That is unless the function that are defined and used within a single source file. Then it should be declared as static function. With classes, you have to do this.

2. **Functions should be short. A function with more than a hundred of lines is *almost always* a sign of poor design.**

3. Function names should start with an upper case letter.

4. Closely related functions should be grouped into a single .cpp file. Namespaces should be used if the functions are not members of a class.

5. If possible, functions should be written to be reentrant. This is so that they may be used with threads and in signal handlers. The key to writing a reentrant function is to avoid the use of static variables. **Note: this will become more important in the**

near future. Why?

6. Parameters of a function should be prefaced by "a_". This makes them easier to identify in the function implementation.

7. For arguments that are passed by reference or by address use "const" in function prototypes if the the function does not change corresponding data in the calling function. Why pass by reference?

8. **There should be a consistent level of detail throughout a function. Namely, the implementation should be top-down.**

9. **A function should have only one purpose. Its definition should be straightforward. It should not have so much detail that it can not be held in your mind.**

10. If you have the same code in several places, it should live in a single function. (For the sake of maintenance, debugging and readability) **I have seen disasters occur when this was not done.**

4. Classes

1. **Class names should start with an uppercase letter.**

2. **A class should be well-define. For the most part, they should be defined by a noun. Time, date, stock, creature, and portfolio are examples of class names**.

3. Class names should indicate the purpose of the class.

4. Classes should not be tiny. It serves little purpose to have a class with one variable and one member function. This type of coding is usually done by people who have just learned classes and think that they should be used everywhere.

5. Classes should clean up after themselves. If the class allocates memory, the destructor should free it. If it opens a file, it should close it in the destructor. Most of the programming tools in C++ and other languages do not need cleanup. For example, vectors, files accessed through fstream, etc.

6. Functions that are only used within a class, should be kept private. This gives a lot more freedom to change things.

7. Inheritance should be used to expand the functionality of an existing class. Multiple inheritance is a little dangerous.

8. As mentioned earlier, all variables of a class should be private.

9. The class declaration should be in a .h file and the implementation in a .cpp file. These files should have the name of the class.

10. For the sake of efficiency, accessor functions should be implemented within the .h file so that they are inline. You don't need to do as heavy commenting on functions defined in the .h file. Why shouldn't we put everything in the .h file.

5. The code

1. Every switch statement must handle the default case. The default case will often represent an error condition.

2. Make sure that your code is readable. Don't use a more obscure language feature if a simpler one will provide the same functionality and be more readable.

3. As much as possible the flow of code should go straight down. A cascade of if statements obscures the code. The following is a demonstration:

```
// Badly written code.
if( condA == OK ) {
	if( condB == OK ) {

		do_something();
		return OK;

	} else {

		return ErrorB;
	}

} else {

	return ErrorA;
}

// Rewrite for better flow.
if( condA != OK ) {

	return ErrorA;
}
if( condB != OK ) {

	return ErrorB;
}
do_something();
return OK;
```

4. As a rule, don't test floating-point numbers for equality. Since most decimal fractions can not be represented exactly as floating point numbers, testing for equality is likely to cause bugs. It is OK with integers to test.

5. Code should be simple enough for an entry-level programmer to understand and modify. This is very important.

6. Use the assert function. For conditions that "can not possibly occur". This does not take up any space in a release version of your code. It only takes up space in the debug version. I will mention screw-up I did when I started consulting.

7. Each class should represent a well-defined project element.

8. For the sake of readability, put a space on each side of binary operators. For example, y = 2 * x + z; reads much better than y=2*x+z; This can be done automatically with some editors.

9. Be consistent in your use of braces should be used. I will talk about the style of braces in class.

10. Use braces on all but the most trivial if, for, and while statements. Namely, use braces if these statements take more than one line. Lots of bugs are caused by the lack of braces when people modify existing code.

11. If a program terminates due to an error condition, the condition must be logged and a message displayed to the screen.

12. Avoid crazy code. For example, I had one student submit a senior project with a 20 page switch statement.

13. Use arrays and vectors rather than multiple variables. For example, in simulating rolling dice, I have had students use 11 difference variables to represent the values of the dice.

14. Programs must be designed to log errors and events. If your program must terminate for any abnormal reason, the reason must be logged and a message displayed to the screen.

15. Defend your code against data entry, system errors, user, and database errors.

16. When writing classes, use .h files for declarations and .cpp for implementation. (definitions) Some people use .cxx, .hpp, and .hxx. .h and .cpp is more widespread.

17. Defend against an include file being included more than once. How do we do this?

6. Miscellaneous

1. **Clean up code. Do not keep functions, variables, classes, etc. that are not used. If you do, you will drive other programmers crazy.**

2. You should test the release version of your code in addition to the debug version. I will explain this in class.

3. **Avoid friend classes. They contradict the principle of data encapsulation which is a core concept of object oriented programming. (Yes, there are exceptions.)**

4. Call back functions might be sometimes necessary and useful. They can be overused. They can make code extremely difficult to follow.

5. As a rule, each class should have its own .h for its definition and .cpp for its implementation. (Yes, I am repeating myself.)

6. **Always check error returns for all functions that return errors. Report and respond to all error events. If a library function can throw an exception be prepared to catch it.**

7. Never have duplicate defines or constants in multiple include files. Never have multiple copies of the same function in multiple .cpp files. Mention Y2K problem.

8. Commented out code and code deleted with #ifdef 0 should not be found in production code.

9. For libraries, once an interface to has been published, it should **never** be changed. With C++, you can always add additional functionality to an interface by function overloading and/or default arguments. Not relevant to your project, but good to mention.

10. As much as possible, use standard and company libraries.

11. If the number of source files in your project is getting too large, create libraries. Not applicable to what we are doing.

12. It is suggested that programmers read such books as the following. I got rid of the older ones.
    1. Meyers, Effective C++.
    2. Meyers, More Effective C++.
    3. Meyers, Effective Modern C++
    4. Meyers, Effective STL.
    5. Pikus Design Patterns.
    6. **Stroustrup, A Tour of C++. This is very useful if you want to upgrade to the latest C++. It will tell you about all the features.**
    7. Very useful are the YouTube videos online for learning.