Coding Standards

This document specifies coding standards. In the past, I have written such documents for three different companies. Not every company will have coding standards, but the lack of them can lead to a lot of problems. Sometimes the coding standards are informal. You may find out your company's standards during code reviews.

When coding standards are created, all new code is require to adhere to the standards. (New standards do not usually require that old code be modified to meet them. WHY?) Many companies have periodic code reviews to make sure that people are following the coding standard. (Among other things) This can occur even if there are no formal coding standards.

The reason that coding standards are created is to support the development of code that is easy to read, debug, maintain, and replace. This standard was developed for code written for C++ and C. (There is still a LOT of C code around. According the the 2020 survey by the IEEE) The standard is easily adaptable to Python, C# and Java. There are also tools like Doxygen that support commenting code. You may use it. Just stay consistent with this document.

Note: developing coding standards can cause a lot of fights. An an entry level programmer, you don't get much choice. If your company has coding standards, you live by them or quit. Some people get weird about standards. I once had an experienced programmer freak out over whether or not to capitalize the first letter in comments about the arguments of a function.

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

You must abide by this coding standard for your Software Design term project and for your Senior Project. Particularly relevant are the sections with multiple "*"s in front of the section name. Some of the sections, are not applicable to your efforts as a student. The most import items are in blue.

Note: I left in a lot of stuff that is not applicable to your projects just so you can give it a look.

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. Within the code, the implementation of a function must be prefaced with the equivalent of a UNIX man page entry. (Not required for inline functions. A simple comment will serve.) 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. So does Doxygen.) 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 applications area for which the software has been written. This keeps the comment more compact.

```
/**/
/*
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

    Victor Miller

DATE

    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. No need to do this at Ramapo.

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. Note: type: "///" in front of the function and a template will be generated for the type data we discuss. You should add to this.

Doxygen may be used as an alternative to this format and will be accepted in my courses. Just make sure that you use labels that correspond to this document. Doxygen can be integrated with Eclipse and Visual Studio. If you use it, it will be your job to learn how to use it and supply the same information as I have indicated.

3. Comments within the code should be indented the same amount as the code.
4. 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.
5. 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.
6. 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.
7. Every enum, structure, class, constant, prototype, define, etc. that is declared in an include file should be commented.

8. If an enum, structure or constant is only used in a class, it should be declared in the class.
9. Use an enum class rather than a pure enum.
10. 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.
11. 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.
12. Don't use comments to explain overly complex code. Rewrite the code.
13. Comments should be meaningful and tell us something. The following is an example of a ridiculous use of commenting:

```
// Assign taxes the value of withholding.
taxes = withholding;
```

14. For the sake of readability, there will be one tab setting for all programmers. Otherwise, code that is written to using one tab setting may look very ragged when displayed using another. We will use the most typical tab setting which is 4. Your editor should be configured so that tabs are replaced by blanks. (Let's find out where this is specified for Visual Studio - look under <Tools>-<Options><Text Editor><C/C++><Tabs>) The replacement of tabs with blanks is essential in that some utility programs that assume 8 character for tabs. This can make your nicely written code look like garbage.

2. *****Source Code Control and testing.** If you are reading this for the Senior Projects course, you should be using a source code control system. Namely GIT in that it is free and it is integrated with Visual Studio and Eclipse. **Please keep your code private**. We will describe these systems in class so that the following comments make sense.
    1. Never check code into the main source control system that has not been tested. This is true no matter how trivial the change. It is job threatening not to do this.
    2. It is the responsibility of the programmer to make sure that all code produced meets the specifications. It is not the job of QA to find your bugs.
    3. The debugger is a test tool. It is not just used to find bugs. Always use it to walk through the code to verify that it works in the manner that you designed it.
    4. When possible, test code with a product like bounds checker or purifier.
    5. Always enter meaningful comments when checking code into a source control system.
    6. Don't keep code checked out if you are not modifying it. Others may need to work on it.
    7. For a large project, you should write code to test modules of your program and this code should also be checked into source control.
    8. Make sure that make files (UNIX) or solutions/project file (VS) are checked in with the code.

9. Check in all additional documentation. SharePoint and Confluence are used by many companies. Jira is often used for project tracking.
10. Check in test programs.
11. 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.
12. Defend your code against data entry, user, and database errors. That is check for all errors.
13. When writing classes, use .h files for declarations and .cpp for implementation. (definitions) Some people use ,cc, .cxx, .hpp, and .hxx. .h and .cpp is more widespread.
14. Defend against an include file being included more than once. How do we do this?

3. Portability. (Not applicable to Ramapo courses, but can be quite an issue in the working world.) This is an issue when there is any possibility that the code will have to be run on a number of different operating systems. This is an issue that one should address from the beginning. It is much harder to make existing code portable than it is to write it to be so.
    1. There should be a library of functions and classes (a port library) to support system dependent system calls. This library will have "ifdef"s to allow this code to be compiled for any system that that you wish to support. For example, there a class should be written to implement access control to shared resources. It will be implemented using critical sections on WIN32 and using mutexes on UNIX. The class will hide all details. The program will merely have to call the member functions lock and unlock to control access.
    2. In the port library will be an include file that will include all those standard include files that might be needed by a program. There will of course be different sets of include files based on the operating system used. This include file will be used by all source files. For most modern compilers, a single set of include files may actually improve the performance of the compiler. This is also useful for building precompiled headers.
    3. All C++ files should be named with the .cpp extension. This is the most universal extension. Similarly, all include files that you write should had .h as an extension. Note: there are other extentions. For example .cc and .cxx are also used for C++ source code files.
    4. Since we are writing code for multiple platforms, there should be no assumptions made on byte ordering or alignment of the various data types. This is not that important as it once was.
    5. When the size of the data type matters, don't use types like int and long. Instead use types that are defined by typedefs. These can be user-defined types or types defined in such include files as stdint.h.
    6. There should be few if any #ifdefs within the code that is not part of the port library. Scattering #ifdefs throughout all the code is not an acceptable way of achieving portability.

4. **********Defines, Constants and Variables.
    1. 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 start with a capital letter. 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. (Yes, I did mention this earlier)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) Enums 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 instead of pure 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.
    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. Do not use register variables. They impede the optimization done by the compiler. They can actually make your program run slower. **This should be no problem any more in that they are out of the C++ standard. They are deprecated in the compilers.**
    12. 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 an important point.)
    13. Avoid using similar looking variable names in the same module.
    14. **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.

15. 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. (A lot of big mistakes happen due to this.) C# does not let you do this.

5. **********Functions
   1. Prototypes for functions that are global should be in include files. That is unless the function that is 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 .h and .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.**
   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.
   8. **As much as possible, 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. You should not have the same code in several places. It should live in a single function. (For the sake of maintenance, debugging and readability)

6. **********Classes
   1. A c**lass name should start with an uppercase letter.**
   2. **A class should be well-define. For the most part, the names should be 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 the class opens a file, it should close it in the destructor. (Note: this not needed for C++ fstream files and other will written classes.) There are now tools in C++ and other languages where most cleanup is not necessary.

6. Functions that are only used within a class, should be kept private.
7. Inheritance should be used to expand the functionality of an existing class.
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. Same with other short functions.

7. ********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 is 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 simple example:

```
// 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.
5. Code should be simple enough for an entry-level programmer to understand and modify. This is very important.
6. Use the assert function test for conditions that "cannot 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. 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. 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. (Amazingly it worked, but it would still be considered very poorly written code.)
13. Use arrays and vectors rather than multiple variables. For example, in simulating rolling dice, I have had students use 11 different variables to represent the values of the dice.
14. Do not copy and paste code. It means any error in that code will be in multiple places. So, if there is a need to duplicate code, put the code in a function.
15. Don't change large amounts of existing code to be consistent with your coding style. This will get a lot of people angry. I have been know to be less than happy with this

8. *******Miscellaneous
1. **Clean up code. Do not keep functions, variables, classes, etc. that are not used in your project. 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 may 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. Each include file should defend itself against being included multiple times. This is done in the manner that the standard include files defend themselves. "#pragma once" is one way to do this.
12. If the number of source files in your project is getting too large, create libraries.
13. It is suggested that programmers read such books as the following. Some of these are getting a little old.
    1. Meyers, Effective C++.
    2. Meyers, More Effective C++.
    3. Meyers, Effective Modern C++
    4. Meyers, Effective STL.
    5. Sutter, Exceptional C++
    6. Porter, The Best C++ Tips Ever.
    7. Coplien Advanced C++.
    8. Booch Design Patterns.
    9. **Stroustrup, A Tour of C++. This is very useful if you want to upgrade to the latest C++. It will tell you about almost all the features of C++ is a very concise manner.**