**Team\_Alpha BC Coding standards:**

**File Names and Extensions**

There are multiple file extensions used to designate C++ source code files and header files. For this course, the file extensions .cpp for source code files and .h for header files shall be used. No other extensions are permitted.

For every project, a file containing the main() function is required. This file shall be named after the project and have a .cpp file extension. For example, the main() function for Project 3 should be in a file named proj3.cpp.

Auxiliary files (e.g proj3aux.cpp) and header files (e.g. proj3aux.h) are permitted, but must be named appropriately.

The executable for a project shall be named after the project. For example, the executable for Project 3 must be named proj3. This is very important because the graders will look for that file to run your program. The graders will not run a.out or any other executable. The executable name is controlled by your Makefile — get it right.

For most projects, you will be creating your own classes. For these projects, the following standards also apply:

* All C++ classes shall be defined in separate header files, with each header file named after the class it contains and having a .h extension. For example, a class that represents a Car object would be named Car and would be defined in the header file Car.h.
* All C++ classes shall be implemented in separate source code files, with each file named after the class it implements and having a .cpp extension. For example, the implementation of the Car class would be contained in the file Car.cpp.
* Member functions shall only be implemented in an implementation file. That is, no inline methods are permitted.

**File Organization**

* Every header file shall be *guarded* to prevent multiple inclusions using the #ifndef, #define, and #endif preprocessor directives. For example, the header file Foo.h would be guarded as follows:
* #ifndef FOO\_H
* #define FOO\_H
* // Code for Foo.h goes here
* #endif
* Source files (header or implementation) shall include all header files they require. For example, if Foo.h refers to iostream and string objects, it must include the iostream and string header files — it may not rely on other source files to include the headers on its behalf.
* Source files (header or implementation) shall *not* include any header files they do *not* require. For example, if Foo.h does not refer to any string objects, it should not include the string header file.

**Class Definition**

The following standards must be followed when a class is defined within it's header (.h) file:

* All class names shall begin with an uppercase letter. Multi-word class names must follow the variable and function naming convention described below.
* A class definition shall have at most one private, protected, and public section. The public section must be first, followed by the protected section, and lastly the private section.
* Every data member of a class shall be private.
* Global variables shall not be used; global symbolic constants (const) may be used.
* Class methods shall follow the function naming conventions given below.
* Class methods must have complete [function header comments](https://www.csee.umbc.edu/courses/undergraduate/202/spring15/projects/coding-standards.shtml#FunctionHeaderComments) in the header file. Fewer comments are required in the .cpp file.
* Class methods must be const whenever possible.
* Class data members begin with m\_ and follow the variable naming conventions below. Example:
* int m\_length;
* int m\_nrStudents;

**Variable, Constant, and Function Declarations**

These standards detail how variables and constants should be declared, proper function prototype declarations, and naming conventions.

* Variables names shall be meaningful and descriptive. For example, if your program needs a variable to represent the radius of a circle, call it radius, *not* r or rad.
* Variables names shall begin with lowercase letters.
* Single-letter variables shall not be used, except as simple for-loop indices, temporary pointer variables, and in special cases such as x- and y-coordinates.
* The use of obvious, common, meaningful abbreviations is permitted. For example, number can be abbreviated as num as in numStudents, or as nr as in nrStudents.
* If commented, variables shall be declared one per line. Comments must appear on the same line without wrapping (use two lines if necessary).
* Separate "words" within variable names with mixed upper and lowercase. For example, grandTotal.
* Function names shall begin with an uppercase letter, and "words" within function names shall be separatd with mixed upper and lowercase. For Example, ProcessError().
* Function names shouold be "active," beginning with a verb and including a noun, whenever possible. For example, GetName().
* Function prototypes shall include parameter names as well as types. If default values for parameters are provided, they shall be specified in the prototype.
* Parameters shall be passed by reference whenever appropriate. In addition, parameters passed by reference shall be const whenever appropriate.
* Constants shall be used for "magic" numbers and strings whenever possible. For example, even well known mathematical constants should be given a name: const double PI = 3.14159;. Some constants found in formulas, for example Celsius = (5/9)\*(Fahrenheit-32), have no meaningful names and are not considered "magic".

**Documentation**

The following sections detail the required program documentation. Failure to adhere to these standards will result in point deductions from your project submission.

**Use of Whitespace**

The prudent use of whitespace (blank lines as well as spaces) goes a long way to making your program readable.

* Use blank lines to separate unrelated sections of code and thereby group logically related lines of code together. For example, the following code fragment is difficult to read and points would be deducted:
* // count total number of students
* for (int i = 0; i < size; i++)
* totalStudents += class[i];
* cout << "total students = " << total students << endl;
* while (moreStudents == true) {
* // read student last name
* ReadStudentName();
* // print full name
* PrintStudentName();
* // now verify student info
* goodInfo = VerifyStudentInfo(student);
* if (goodInfo == true)
* cout << "good info" << endl;
* }
* cout << " this is the end" << endl;

The following formatting would be acceptable:

// count total number of students

for (int i = 0; i < size; i++)

totalStudents += class[i];

cout << "total students = " << total students << endl;

while (moreStudents == true)

{

// read student last name

ReadStudentName();

// print full name

PrintStudentName();

// now verify student info

goodInfo = VerifyStudentInfo(student);

if (goodInfo == true)

cout << "good info" << endl;

}

cout << " this is the end" << endl;

* Do *not* use tabs (unless your text editor changes tabs to 3 or 4 spaces). Tabs may be interpreted differently by different editors.
* Use 2 to 4 spaces for each level of indentation, and be consistent. Too much indenting makes code unreadable and causes lines to wrap. Using Emacs along with an appropriate .emacs configuration file will automatically indent your code appropriately.

// This is an example of good formatting

if ( x > 90 )

{

statement 1;

statement 2;

}

else

{

statement 3;

}

// This is an example of too much indenting

if ( x > 90 )

{

statement 1;

statement 2;

}

else

{

statement 3;

}

// This is an example of inconsistent indenting

if ( x > 90 )

{

statement 1;

statement 2;

}

else

{

statement 3;

}

* Use spaces around all operators. For example, write

x = y + 5;

not

x=y+5;

**Use of Braces**

* Use a consistent style for braces.
* Braces are not required for single statement if/else/while/for structures, assuming suitable indentation is used to show the structure. For example, the following formats are permitted:

if (grade > 90)

cout << "You got an A" << endl;

else

cout << "No A for you" << endl;

for (i = 0; i < 30; i++)

array[i] = -3;

**Comments**

Comments are the programmer's main source of documentation. From "The Practice of Programming" by Brian Kernighan and Rob Pike, page 23:

Comments are meant to help the reader of a program. They do not help by saying things that the code already plainly says, or by contradicting the code, or by distracting from the code with elaborate typographical displays. The best comments aid the understanding of a program by briefly pointing out salient details or by providing a larger-scale view of the proceedings.

**Rule of Thumb:** Every five lines of code needs *at least* one comment. Not every line of code needs a comment. Constant declarations *must* have one comment.

**File Header Comments**

Every .cpp and .h file shall contain an opening comment describing the contents of the file and other pertinent information. This "file header comment" *must* include the following information.

1. The file name
2. The project number
3. Author's name
4. The date the file was created
5. Author's section number
6. Author's UMBC e-mail address
7. A description of what the code in the file does

For example:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* File: Proj1.cpp

\*\* Project: CMSC 202 Project 1, Fall 2005

\*\* Author: Bob Smith

\*\* Date: 9/22/05

\*\* Section: 0304

\*\* E-mail: bsmith32@gl.umbc.edu

\*\*

\*\* This file contains the main driver program for Project 1.

\*\* This program reads the file specified as the first command line

\*\* argument, counts the number of words, spaces, and characters, and

\*\* displays the results in the format specified in the project description.

\*\*

\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**Function Header Comments**

Function header comments are the primary form of documentation for the user of our functions and classes. It is important that this documentation be both complete and accurate as it forms a "contract" between the user and and the implementer.

Each function and class method shall have a header comment that includes the following information:

1. The function's name
2. The function's *pre-conditions* (if there are no pre-conditions, say so)
3. The function's *post-conditions*

The full function header comment must appear in the appropriate .h file.

A *pre-condition* is a condition that is required or assumed to be true when a function is called. The function is not guaranteed to perform correctly unless the pre-condition is satisfied. It is not just a restatement of the parameter names and types. All functions must test for pre-conditions to the extent possible.

A *post-condition* is a condition that will be true when the function is completed, assuming the pre-conditions for the function have been satisfied and the function completes.

For example, in Circle.h we would expect to find prototypes and full function header comments as follows:

//-------------------------------------------------------

// Name: CircleArea

// PreCondition: the radius is greater than zero

// PostCondition: Returns the calculated area of the circle

//---------------------------------------------------------

double CircleArea (double radius ):

whereas in the .cpp file we expect to find the function implementation and comments meant for the programmer:

// CircleArea

// Given the radius, returns the area

double CircleArea ( double radius )

{

const double PI = 3.14159;

// handle unmet precondition

if (radius < 0.0)

return 0.0;

else

return PI \* radius \* radius;

}

Pre- and post-conditions will be discussed in more detail in class.

**In-Line Comments**

* In-line comments are used to clarify what your code does, not how it does it.
* An in-line comment must appear above the code to which it applies and is indented to the same level as the code. The following example is acceptable:

// check every element of the array

for (i = 0; i < ARRAYSIZE; i++)

{

// add one to odd values

if (array[i] % 2 == 1)

{

array[i]++;

}

}

This example is not acceptable:

for (i = 0; i < ARRAYSIZE; i++) // check every element of the array

{

if (array[i] % 2 == 1) // add one to odd values

{

array[i]++;

}

}

* Be sure your comments don't contradict the code.
* Don't comment bad code — rewrite it.
* Don't comment the obvious. For example, the following comments are superfluous:

// return true

return true;

// increment counter

++counter;

* Constants and variables should be commented on the same line on which they occur. Such comments should not wrap; use two lines if necessary.

const int NUM\_STUDENTS = 35; // number of students in the class

const int NUM\_SECTIONS = 4; // number of sections of CMSC 104

const float CUTOFF = 88.5; // cutoff for an "A" for the 2001

// spring semester

bool isCurrPresent = false; // is the current student present?

int currStudentInd = 0; // index of current student