Variables:

- Required: Starts with lower case letter
- Required: Use descriptive names (exception for “standard loop variables” used in that context (‘i', ‘j’, ‘k’))
- Required: Name is not a verb (i.e. doesn’t “sound like” a function)
- Preferred: Use a case where non-first words start with upper case letters (eg. ‘numQuizzes’, ‘numberOfValues’, etc.)
- Required: Consistent naming scheme (use of case, underscores, etc.)

Constants:

- Required: Name is in all upper case letters
- Required: Underscores separate individual words of identifier (eg. ‘NUMBER_OF_EXAMS’)

Functions:

- Required: Starts with lower case letter
- Required: Use descriptive names
- Required: Name is a form of verb (i.e. “sounds like” functionality (i.e. ‘computeAverage’ as opposed to ‘average’)
- Preferred: Use a case where non-first words start with upper case letters
- Required: Consistent naming scheme (use of case, underscores, etc.)

Curly brace placement:

- Required: Consistent placement (i.e. always on next line, indented same amount as prev line, etc.)
- Preferred: Use curly braces to enclose code in a compound statement after an “if” or a loop, even if only one statement is currently needed

Whitespace use:

- Required: No line extends past 80 characters long, for any reason
- Required: No use of tabs in source code
- Required: Indentation with consistent number of spaces through entire source code
- Required: Every “block” or “compound statement” or “chunk of code within curly braces” is indented

Comments / Documentation:

- Required: “Header block” on every source code file – must include programmer’s name, approximate date the code was written and a brief statement about the purpose the code serves.
- Required: Inline documentation (i.e. comments) – especially to describe areas of your code that are not especially “self documenting” or “very obvious” to a reasonably knowledgeable reader of your source code

Program organization, Design:

- Required: Avoid use of “while (true)” type loops that use “break” to stop the loop – use proper loop conditions instead
- Required: Avoid use of “break” and “continue” to alter flow of control in loops
- Required: Use only topics covered in class – do not use libraries, functions, concepts, keywords, datatypes, etc., that have not been discussed in class as of the time the project specifications are posted
- Required: Each global function has its prototype specified and documented in its own “.h” header file (only exception is overloaded functions having the same name, for which all functions with the same name are prototypes and documented in a single “.h” file)
- Required: Each global function has its implementation in its own “.cpp” file (only exception is overloaded functions having the same name, for which all functions with the same name are implemented in a single “.cpp” file)
- Required: Each class is defined in its own “.h” header file. Class methods will be prototyped and documented in the “.h” file (only exception is for exceptionally simple (i.e. 1 to 4 lines or so) functions which can be implemented directly in the “.h” file
- Required: All non-exceptionally simple methods of a class are implemented in a single “.cpp” file for that class. Each class’ methods are implemented in a “.cpp” file for the class.

Arrays:

- Required: Statically allocated array sizes must be specified using literal values or named constants

Classes and Objects:

- Required: Class names start with upper case letters, and use “camel case” after that - eg. QuizClass, ModeOfTransportationClass, etc.
- Required: Objects are just variables of a class type, so they are named the same as variables (I.e. start with lower case letter, “camel case” after that)
- Required: All but the simplest member functions are implemented outside the class definition (I.e. NOT within the curly braces for the class)
- Required: Functions in your class definition (i.e. prototypes or function headers) are documented with comments describing the function
- Required: ALL data members (attributes) are kept “private”
- Required: Function that ought to be member functions are implemented as member function as opposed to global functions (and vice versa)
- Required: Data members of a class are attributes of the class - do not include member variables for convenience if they’re not descriptive of objects of the class - eg. Don’t have a member variable called “i” because you often need a loop variable in many of your member functions - if “i” doesn’t describes objects of the class in same way, it should not be a member
- Required: Avoid storing calculable values as attributes - eg. Don’t have both “val” and “valSquared” as attributes; instead, store val as an attribute and implement a function to compute and return val squared when needed
- Required: Avoid duplicated or redundant data attributes. This is almost the same as “avoid storing calculable values as attributes”, but, for example, if something can be specified multiple ways, store the data in only one way - eh. Say a RectangleClass has “upperLeftXLoc” and

"upperLeftYLoc" attributes - store the rectangle's extent as EITHER "lengthVal" and "widthVal" OR "lowerRightXLoc" and "loweRightYLoc", but not both

- Required: Avoid having data attributes that don't apply to all objects of the class. Don't have an attribute that describes some objects of the class, but is not applicable to others. Eg. Don't have an "altitudeAboveGround" attribute in a "ModeOfTransportationClass" that would be applicable for airplane and helicopter objects, but not car, bike, or motorcyle objects.
- Required: Classes should be "responsible for" their own attributes. If you have a "MichiganCourseClass" and that class has an attribute that is an array of "StudentClass" objects enrolled in the course, the MichiganCourseClass should manage the attributes *directly associated* with MichiganCourseClass, and the StudentClass should manage the attributes *directly associated* with StudentClass. Typically the MichiganCourseClass should not directly deal with things like the student's name or ID number, but rather, it should "relinquish responsibility" to the StudentClass for those purposes. For example, when reading in a course's attributes and its students from a data file, the MichiganCourseClass should read the attributes for the course (course number, course name, room location, etc.), but it should call a function in the StudentClass to read in attributes of a student (student name, student ID, degree program, etc.)

Other things to look out for:

- "Magic numbers" - essentially any literal values in your code not bound to a variable or constant (exception for "standard values" that are obvious and clear in the code)
- "Duplicated code" – same code to perform same functionality in multiple places – even minimal amounts of duplicated code is dangerous and must be avoided when at all possible
- Overly complicated code – you should avoid writing complicated or difficult to understand code when unnecessary

Disclaimer:

- While I tried to capture most of the significant style and design type issues we will be looking for when grading your programs, this is not necessarily a complete list. You may receive deductions for other style- or design-related issues that are not listed here when they are noticed in your submissions.