C++ Style Requirements
CS 202: Programming Systems

The following describes the style requirements when programming in C++ for CS202.

1) Comments must use correct spelling and grammar.

2) Every function needs a significant header comment describing the purpose of the classes or functions that exist within the file. This is where you will want to describe the relationships between classes and why data is private versus protected, for example.

3) Use the following conventions for all variable, constant, and function names:
   - Use mnemonic names that have a clear and understandable meaning.
   - Always begin with a letter; an identifier cannot begin with a digit
   - An identifier must consist of letters, digits, or underscores only.
   - Use all lower case letters (no capitalized or upper case variables)
   - Use **meaningful words** that represent the function of the variable
   - You cannot use a C++ reserved word as an identifier
   - **Only use single letter variable names when working with loop control variables and array indices!**

3) The use of goto's and global variables is **not** allowed in this class.
   - Global constants are encouraged!

4) With loops, always use the conditional expression of the loop to determine if the loop should continue or not
   - Never use break within a loop
   - Never "return" from a function from within a loop

5) With functions always remember the following:
   - Always use the returned value coming back from a non-void function
   - Always return a value for a non-void function (through each path through the function)

6) Place all variable definitions/declarations before the executable statements of the main() program or each function.
   - This means that variable declarations should not be scattered throughout your code!
   - C++ allows otherwise...so take this as a style requirement!

7) Only place one statement per line.
   - C++ allows multiple statements to appear on the same line; however, for style please limit yourself to only one per line.
   - Always have comments PRIOR to the code in which it applies

8) A blank space is required between words in a program line.
   - Always leave a blank space after a comma.
   - Always leave a blank space before and after the following operators:
     * / + - = << >>

9) Indent each line of the program except for the curly braces that mark the beginning and end of the main program.
   - All lines between the {} are to be indented two or more spaces consistently.
   - We recommend lining curly brackets up
   - The body within a compound block surrounded by curly brackets should line up

10) Use blank lines between sections of the program.
   - For example, there should always be a blank line between the compiler directives and the rest of the program.
   - There should also be a blank line between the variable definitions/declarations and the executable statements.

10) Do not use a blank line between every line of code; this reduces program readability, instead of enhancing it!

11) When choosing between an if/else control statement and using the conditional operator (`?:`), always choose the if/else structure.
   - This is required to assist debugging and readability.

12) Indentation for if/else control statements:

```
if (conditional expression)
{
     statement;
     statement;
}
else
{
     statement;
     statement;
}
```

13) Indentation for Switch control statements:

```
switch (selector expression)
{
     case label1 : statement;
          statement;
          break;
     case label2 : statement;
          statement;
          break;
     default : statement;
          break;
}
```

- Notice: For style always use the default label.

- Place only one case label per line

- Always place a break after the last case label -- even though it is syntactically unnecessary. If you decide to add more labels later on ... you just might forget to add the break at that time.

#13) Indent loops as follows:

```
for(i = 1; i <= some_max; i++)
{
   statement;
   statement;
}

i = 0;
while (i <= some_max)
{
   statement;
   statement;
}

do
{
   statement;
   statement;
}
while (i <= some_max);
```

a) Avoid changing the loop control variable inside of the body of a *for* loop!

#14) When writing your own functions, use the following style guidelines:

a) Always use a return at the end of a function; if the function is not supposed to return a value then end the function with:

```
return;
```

b) Functions that do not have any parameters must have the term "void" side the parentheses of the function declaration:

```
data_type function_name (void);
```

c) Functions that do not return any value to the calling routine must be declared with a "void" data_type:

```
void function_name(parameters);
```

d) The rule you should live by is: NO GLOBAL VARIABLES IN FUNCTIONS. Instead, use formal parameters to input the data and output the result. And, use local variables to assist with any intermediate calculations.

#15) When writing your own .h files, uses these guidelines:

a) In .h files, put with each function prototype a clear description of what it does. In the .c files, put in the header of each function a clear description of how it does its job.

b) In implementation code for all functions which are not trivial: break the code into paragraphs. Before each paragraph put white space and a comment describing what and/or how this paragraph of code does its job.

c) In .h files, place your structure definitions, class interfaces, and function prototypes. Do <u>not</u> place function definitions (i.e., the implementation of your functions) in these files.

d) Never (unless working with templates) `include` .c or .cpp files...only .h files!

#16) When writing your own classes in C++, use the following style guidelines:

a) Place the public section first, followed by the private and protected sections.

b) Objects should be named using proper nouns, such as a_list, my_list, the_vehicle, a_peach, etc.

c) Classes, on the other hand, should be named using noun phrases, such as todo_lists, vehicles, peaches.

d) Operations (member functions) that can be performed on classes should be named with an active verb, such as retrieve, create, check, add_an_item, delete_an_item, modify, etc.

e) Always separate the implementation of a member function from the class header; place the class headers in a ".h" file and the implementation of the member functions in a corresponding ".c" file (unless you are creating inline member functions). **<u>Never</u> "include" a .c file...only .h files!**

f) The public interface (including its comments) should be self-contained and meaningful without looking at private data

g) Use the following indentation:

```
class todo_lists
{
  public:
       //constructor(s)
       //copy constructor
       //destructor

       //data members
       //member function prototypes

  private:
       //data members
       //member function prototypes

  protected:
       //data members
       //member function prototypes

};
```