C/C++ Coding Conventions

All comments should be checked for spelling and grammatical errors.

1 Commenting Conventions

Including comments for files, classes, functions, and function prototypes provides programmers with a valuable tool when maintaining or sharing ideas. Make sure to use correct spelling, complete sentences, and proper grammar.

1.1 In-Line Comments

In-line comments are comments that appear on one line, typically the line immediately before a section of code. In C++, in-line comments start with // while in C comments start and end with /* and */ respectively. Use in-line comments within a function to explain complex and unclear sections of code. Don't write comments about things that are obvious from reading the code. In-line comments should provide a clear description of the meaning of the code. Use in-line comments liberally.

C++

```
// As long as the value is not found and there are more values continue looking at
// the next element.
for( int i = 0; i < size && !found; i++ ) {
    // Check to see if the value is found
    if( list[i] == searchValue ){
        // Set the found flag and store the position where the value was found.
        found = true;
        position = i;
    }
}
```

C

```
/* As long as the value is not found and there are more values continue looking
 * at the next element. */
for( int i = 0; i < size && !found; i++ ) {
    /* Check to see if the value is found */
    if( list[i] == searchValue ){
        /* Set the found flag and store the position where the value was found. */
        found = true;
        position = i;
    }
}
```

1.2 File Headings and Class Comments.

```
/**
 * file:  someApp.cpp
 * author:  Prof. Aars
 * course: CSI 1440
 * assignment:  project 0
 * due date:  1/20/2006
 *
 * date modified:  8/23/2016
 *      - major overhaul of the format of commenting
```

```
 *      - removed stars from beginning and end of comment
 *      - added several information fields
 *
 * date modified: 1/10/2006
 *      - added assignment category to file headings
 *
 * date modified: 2/20/2005
 *      - file created
 *
 * This document will inform CSI students of comment expectations.
 */
```

1.3 Function Comments

```
/**
 * factorial
 *
 * This function computes the factorial of the given input.  The factorial
 * is defined as factorial(n) = n! = n * (n-1) * (n-2) * ... * 2 * 1
 *
 * Parameters:
 *      n:  the number on which to computer the factorial
 *
 * Output:
 *      return:  the factorial of n, or 1 if n <= 0
 *      reference parameters: none
 *      stream:  none
 */
```

2 General Conventions

2.1 Tabs

4 spaces.

- IDEs will allow the programmer to convert all tabs to spaces upon saving the file.

2.2 Line Length

- All lines of code more than 80 columns should be split across multiple lines.

2.3 Included or Imported files

Include or Import only necessary files. Never include a .cpp file.

2.4 Big Four Operations for classes

For classes with pointers as member attributes, implement the big four for the class.
These are:

- Default Constructor
- Copy Constructor
- Destructor
- Assignment Operator

3 Brace Placement

- The opening brace should appear on the same line as the preceding statement.
- The closing brace should be placed on a line of its own, in alignment with its related statement.

3.1 For-loop statement

```
for( int i = 0; i < n; i++ ) {
}
```

3.2 While statement

```
while( expression ) {
}
```

3.3 Do-while statement

```
do {
} while( expression );
```

3.4 If and else statement

- Always include the braces for all if, else if, and else.

```
if( expression1 ) {
} else if( expression2 ) {
} else {
}
```

3.5 Try, Catch blocks (Exception Handling)

```
try {
} catch( ExceptionType1 e ) {   // catch all interesting exceptions
}
```

3.6 Switch Statements

- When using switch statements, always include the default case.
- When using pass through within a case, always include comment as last line of case indicating: //pass through.

```
switch( value ) {
   case 0: {
      break;
   }
   case N: {
      //pass through
   }
   default: {
   }
}
```

4 Naming Conventions

All names should make sense within the domain of the problem being solved. This helps to minimize the "representational gap" of the program.

4.1 Preprocessor Macro and Constant Names

Use all uppercase name with underscores (_) separating names.

4.2 Type Names and Class Names

Use "UpperCamelCase" for all class names, structures, enumerations, and typedefs.

- Use nouns to define classes, structures, enumerations, and typedefs.
- Use plural nouns to define collections of things. When possible, use collection names that correspond to the plural form of what object is being collected.
- Use suffix Ifc to distinguish interface classes from other types of classes.

4.3 Function Names

Use "lowerCamelCase" for all function names. This will help distinguish quickly between user-defined types and newly defined behavior.

- Use verbs or verb-phrases as names for functions as they generally describe some action or behavior.
- When crating a function for a class which returns bool and is an Accessor function for the class, begin the name of the function with "is".
- Create 'setter' Mutator functions to enable access to member attributes of the class. Begin name of function with "set".
- Create 'getter' Accessor functions to enable reporting of member attributes within a class. Begin names of these functions with "get".

4.4 Variable and Parameter Names

Use "lowerCamelCase" for all variable and parameter names as a visual clue.

- Within class member functions, use the this pointer (this->) to visually clue readers of your code as to which variables or parameters are manipulated.
- Use nouns, compound nouns, and plural nouns to name variables.
- Use parameter right-hand side or other in functions like copy constructor, overloaded assignment operators to distinguish from this instance.

5 Class Definitions

- The declaration of the class should be created in a Widget.hpp or Widget.h file and the implementation should be found in Widget.cpp or Widget.c.
- Unless a class is a templated class, which has to be defined in a single file, like Collection.hpp or Collection.h.
- Class filenames should be the same as the class defined within.
- One class definition per file tuple (Class.hpp, Class.cpp), unless you are using inner classes.
- Test driver for class Widget should be created in file named testWidget.cpp.

- Use header guards to ensure the compiler does not attempt to declare or define the class multiple times. Include guard identifier should be all caps, separating words with underscore (_).

```
#ifndef  WIDGET_H
#define WIDGET_H
…
// class Widget declaration
…
#endif
```