Programming is a team activity. Adhering to programming style standards aids the programmer in development and maintenance, as well as enabling team development.

Section 1: Files

1.1. Global Program File (*.c) Structure - elements of source code files must appear in this order:

a. comment header block (see 2.1)
b. `#include` pre-processor directives
c. main function (if applicable)
d. other function definitions

1.2 Global Header File (*.h) Structure - elements of header files must appear in this order:

a. comment header block (see 2.1)
b. #define guard (see 4.1)
c. `#include` pre-processor directives
d. `#define` pre-processor directives
e. type definitions
f. function prototypes, including function headers (see 2.2)

Section 2: Documentation / Comments

2.1 Comment Header Block

- Every file (both .c and .h) must have a "comment header block" like this template:

```
/** pex01Main.c
 * ============================================================
 * Name: name, date
 * Section: your section
 * Project: assignment information
 * Purpose: high level description of purpose of the program
 *          could be multiple lines
 * ============================================================ */
```

2.2 Function Header Block

- Each function prototype within a header file must have descriptive comments directly preceding the function prototype (no blank line between comments and prototype). The function header comments describe the use of the function, not how it works.

```
/**
 * @brief quotient of two ints
 * @param aNum the dividend
 * @param bNum the divisor
 * @return quotient of a divided by b
 * @pre bNum is not zero
 * @post aNum and bNum are unchanged
 */
int quote(int aNum, int bNum);
```

- **@brief** – provide a description of what the function does; it's intended purpose (not how it does it, but what it is for)
- @**param** – list each parameter and describe the use

- **@return** – describe the return value (if any)
- **@pre** – describe the preconditions: things that must be true before calling the function (e.g., limits on ranges of input values, other functions that should be called first, etc.)
- **@post** – describe the postconditions: what will change as a result of the function call, what will be true as a result of calling the method

2.3 **Documentation Statement** – the documentation statement for the project must appear in the file containing the main function.

2.4 **Inline Comments**

- Do not comment every line of code
- Comments explain a line of code, or code block, in the context of the problem domain (does not simply rephrase the code in English).
- Comment should appear before the code being explained
- Comments are subject to the line length restriction (see 3.1)

Section 3: Line Length

3.1 **No line should exceed 100 characters** (indentation included).

Section 4: Including / Header Files

4.1 All header files should have **`#define` guards** to prevent multiple inclusion. The format of the symbol name should be `FILE_H`, as in this example:

```
/** pex01Functs.c
 */

#ifndef PEX01FUNCTS_H
#define PEX01FUNCTS_H

...

#endif // PEX01FUNCTS_H
```

4.2 **Include only what you use**. If a source or header file refers to a symbol defined elsewhere, the file should directly include a header file that provides a declaration or definition of that symbol. It should not include header files for any other reason. Do not rely on transitive inclusions.

Section 5: Naming

5.1 **Variable naming, declaration, initialization**

- Each variable name should be descriptive of its use or purpose and is normally composed of two, or more, words using "camelCase" starting with a lowercase letter, for example: `interestRate`
- Using single letter variable names, such as `j`, `k`, or `n` for loop counters is OK.
- Each variable declaration must be on a separate line.
- Each variable must be initialized when declared.
- If the name of an identifier is not explicitly clear, it must be clarified using a comment.

Examples of clear identifiers:

```
int widthInPixels; float milesFromCenter;
```

Examples that need a comment for clarification:

```
int width; // in pixels
float distance; // from the center in miles
```

5.2 Function naming

- Each function name should be descriptive of its use or purpose and is normally composed of two, or more, words using "camelCase" starting with a lowercase letter, for example: `computeInterestRate`

5.3 Typedef naming

- Each typedef name should be descriptive of its use or purpose and is normally composed of two, or more, words using "camelCase" starting with an uppercase letter, for example: `ComplexFloat`

Section 6: Indentation, Delimiters, Whitespace

6.1 Whitespace

- Include one space between operators and operands to improve the readability of the code.
- Use blank lines to separate 'logical paragraphs' of code.

6.2 Indentation

- Use indentation to indicate logical structure (selection, iteration, sequence) of code.
- Be consistent with curly-braces. Place the beginning brace on the line that starts the statement block, and the last brace in the starting column of the initial statement. For example:

```
void func(int numBunnies) {
    for (int j=0; j < n; j++) {
        printf("%d\n", j);
    }

    if(numBunnies > 10) {
        printf("that's a lot of bunnies\n");
    } else {
        printf("we might be able to contain them\n");
    }
}
```

Section 7: Never Use

7.1 Magic Numbers – always use named constants instead

7.2 global variables – There are meaningful uses of global variables; but they can be the source of bugs and hard to maintain code. When it is reasonably possible to avoid using a global variable, then avoid using a global variable.

7.3 `increment or decrement in a complex expression` – increment (++) and decrement (--) should only be used in a larger/complex expression when the meaning is obvious (which is almost never).

7.4 Single line if-statement.

7.5 Single line loop.

7.6 `break` in loops – `break` in a switch-case is necessary; any other use of `break` leads to unreadable code

7.7 `continue` – in a loop leads to unreadable code

7.8 `goto` – should never be used