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 in this order: C system headers, your project's headers; no unnecessary #include directives
c. necessary function declarations (if applicable)
d. main function (if applicable)
e. 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 in this order: C system headers, your project's headers; no unnecessary #include 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:
 * ============================================================ */
```

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).[1] 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
*/
int quote(int aNum, int bNum);
```

[1] The function documentation may also be in the implementation file. If it does, it must be an exact duplicate of the documentation in the header file.

- **@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)

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 must be: the same as the filename, in all caps, with underscore replacing spaces and DOT in the file name; as in this example:

```
/** pex01Functs.h
 */

#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`

5.4 const naming

- Each constant name should be descriptive of its use or purpose and is normally composed of two, or more, words using all uppercase using underscore as the word separator, for example: `const int MAX_STRING_LENGTH = 200;`

5.5 enum constants naming

- Each value in an enum should be descriptive of its meaning and is normally composed of two, or more, words using all uppercase using underscore as the word separator, for example: `enum classStanding { FOURTH_CLASS, THIRD_CLASS, SECOND_CLASS, FIRSTIE };`

5.6 #define naming

- Each #define constant name should be descriptive of its use or purpose and is normally composed of two, or more, words using all uppercase using underscore as the word separator, for example: `#define MAX_STRING_LENGTH 200`

Section 6: Indentation, Delimiters, Whitespace

6.1 Whitespace

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

6.2 Indentation

- Use indentation to indicate logical structure (selection, iteration, sequence) of code. Indentation is **4 spaces**.
- 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. A magic number is a numeric literal that is used in code without any explanation of its meaning. (e.g. using *9.7998* in an expression versus declaring a named constant, *const double GRAVITY = 9.7998*).

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

Section 8: Defensive Programming

8.1. **Avoid known security flaws**

- avoid library functions that don't check for buffer overflows (e.g., gets(), strcpy())
- **always** hardcode the format string when using the printf() family of functions

8.2. **Don't trust input data**

- Check the return values from functions used for reading input
- Sanitize input (e.g., ensure numeric input is: [a] a number [b] in range, and [c] invalid characters are not present)

8.3. **Fail gracefully**

- Report detected errors on stderr
- Close files and delete memory on the way out

8.4. **Be careful when handling files**

- Always check for successful file open
- Always close files

8.5. **Always free dynamically allocated memory**

Document Revision History:

- **Version 2.1 (18 Jan 2021)** – Section 2.2, 1st paragraph, added footnote
- **Version 2.1.1 (20 Jan 2021)** – added quick reference, clarification in sections 4.1, and 6.1
- **Version 2.2 (11 Feb 2021)** – added sections 5.4, 5.5, 5.6
- **Version 2.2.1 (23 Feb 2021)** – clarified indentation in section 6.2
- **Version 2.2.2 (14 Jul 2021)** – clarified section 1.1.b, added 1.1.c
- **Version 2.2.3 (31 Jul 2023)** – clarified section 7.1
- **Version 2.2.4 (15 Aug 2023)** – removed requirement for @pre @post in section 2.2
- **Version 3.0 (01 Feb 2024)** – added defensive programming Section 8

Files

1.1. **Program File** (*.c) structure: comments (2.1), #includes, [prototypes], main(), [other functions]
1.2. **Header File** (*.h) structure: comments (2.1), #define guard (4.1), #includes, #defines, typedefs, prototypes with comments (2.2)

2. Documentation

2.1. **Comment Header Block** – every file must have a comment header block
2.2. **Function Header Block** – Each function prototype must have function header block
2.3. **Documentation Statement** – must appear in the file containing the main function
2.4. **Inline comments** – explain code in the context of the problem

3. Line Length

3.1. No line exceeds 100 characters

4. Includes / Header Files

4.1. **#define guards** – constant in all CAPS using UNDERSCORE to replace spaces and dots
4.2. Include only what you use; don't depend on transient includes

5. Naming

5.1. **Variable naming** (concatenated English words in camelCase),
declaration (separate lines),
initialization (when declared)
5.2. **Function** naming (concatenated English words in camelCase)
5.3. **typedef** naming (concatenated English words in **C**amelCase)
5.4. **const** naming (concatenated English words in ALL_UPPERCASE)
5.5. **enum const** naming (concatenated English words in ALL_UPPERCASE)
5.6. **#define** naming (concatenated English words in ALL_UPPERCASE)

6. Indentation, Delimiters, Whitespace

6.1. Whitespace – spaces between operators and operands (except incr/decr).
6.2. Indentation, delimiters – **4 spaces** for indentation; consistent indentation, open curly braces on the line that starts the block; close curly braces on line alone

7. Never Use

7.1. Magic numbers
7.2. Global variables
7.3. Increment or Decrement in complex expression
7.4. Single-line if statement
7.5. Single-line loop
7.6. break in loop
7.7. continue
7.8. goto

8. Defensive Programming

8.1. Avoid known security flaws
8.2. Don't trust input data
8.3. Fail gracefully
8.4. Be careful when handling files
8.5. Always free dynamically allocated memory