Coding-Style and Documentation Requirements

Follow these conventions to make your programs easy for other people to read.

1. **Identifiers** (names of variables, constants, subprograms, classes, etc.)

   - **Variables and Parameters**. A variable or parameter's name should be a *noun* or *noun phrase* that describes the value it stores. Avoid using single-letter variable names except as noted below. Variable names should be all lowercase, except for multiword names: capitalize the first letter of every word after the first.

     Good: *radius, diameter, edgeLength, windowWidth*, etc.

     Poor: *r, rad, d, e, w*, etc.

     Exceptions: Single-letter variable names are permitted as the subscript of an array – for example, *values[i]* – and as the loop-control variable in a for loop.

     Ambiguous: Abbreviated words, for example, *temp*. Is it a temporary variable or a temperature or …? Try to avoid using ambiguous identifiers.

     Distinguish instance variables from others using a convention like *myLength* or *_length*.

   - **Constants**. Use *named constants* instead of numeric literals (aka "magic numbers"). Like a variable, a constant's name should be a *noun* or *noun phrase* describing the value it stores. To distinguish them from other identifiers, constant names should be all uppercase, with multiple words separated by underscores.

     Good: *PI, GRAVITY_FORCE, DEFAULT_WINDOW_HEIGHT*, etc.

     Poor: *f, F, m, M, a, A*, etc.

   - **Functions/Methods**. function and method names should be *verbs* or *verb phrases* that describe the action the function performs. Use the same lettering-style as variables; the parentheses that follow a function's name will distinguish it from a variable.

     Good: *run, getRadius, setRadius*, *calculateArea*, etc.

     Poor: *f, g, radius, area, circumference, area*, etc.

   - **Classes**. Use a noun or noun phrase that describes the kinds of objects being modeled. To distinguish class names from other identifiers, capitalize each word in the name.

     Good: *Circle, Temperature, PriorityQueue*, *BinarySearchTree*, etc.

   Goal: Use descriptive identifiers to make your code as self-documenting as possible.

2. **White Space**. Even though C++ ignores white space, you should use white space judiciously to improve the readability of your code.

   - Use **vertical white space** / blank lines to separate chunks of code that are different logical (i.e., algorithmic) steps within your code.
   - Use **horizontal white space** / indentation to indicate chunks of code that are "inside" (i.e., controlled by) other lines of code. Indent a minimum of 3 spaces or 1 tab.

   See the next section for examples.

Coding-Style and Documentation Requirements

3. **Curly-Braces**. C-family languages use curly-braces to denote the beginning and end of scope contexts. There are two conventions for writing curly-braces: *Kernighan & Ritchie (K&R) Style* and *Aligned Style*. You may use either one, but you must be consistent.

K&R Style	**Aligned Style**

`if ( condition1 ) {`
`  statements1`
`} else if (condition2 ) {`
`  statements2`
`} else {`
`  statements3`
`}`
	`if ( condition1 )`
	`{`
	`  statements1`
	`}`
	`else if (condition2 )`
	`{`
	`  statements2`
	`}`
	`else`
	`{`
	`  statements3`
	`}`

`for ( control-expressions ) {`
`  statements`
`}`
	`for ( control-expressions )`
	`{`
	`  statements`
	`}`

`while ( condition ) {`
`  statements`
`}`
	`while ( condition )`
	`{`
	`  statements`
	`}`

`do {`
`  statements`
`} while (condition);`
	`do`
	`{`
	`  statements`
	`}`
	`while (condition);`

`switch ( expression ) {`
`  case constant1:`
`  statements1`
`  break;`
`  case constant2:`
`  statements2`
`  break;`
`  ...`
`  case default:`
`  statementsN`
`}`
	`switch ( expression )`
	`{`
	`  case constant1:`
	`  statements1`
	`  break;`
	`  case constant2:`
	`  statements2`
	`  break;`
	`  ...`
	`  case default:`
	`  statementsN`
	`}`

`int getStatus() {`
`  statements`
`}`
	`int getStatus()`
	`{`
	`  statements`
	`}`

`class List {`
`  members`
`};`
	`class List`
	`{`
	`  members`
	`};`

Coding-Style and Documentation Requirements

4. **Comments**. Comments are a key part of program documentation, as they help a human read your code. C++ supports two basic kinds of comments: *block comments* that begin with `/*` and end with `*/`, and *1-line comments* that begin with `//` and end at the end of the line. Your programs should include the following kinds of comments:

- **Header Comments**. Each file you create should begin with a block comment that answers the 5 "W" questions: *Who? What? Where? When? Why?*.

  Example:

```
/* List.h contains the declaration of class-template List,
 *  a simple, singly-linked list.
 *
 * author      Jane Doe
 * why         Project 5, CS 112
 * where       Calvin University
 * date        Spring 2020
 */
```

- **Subprogram Comments**. Each subprogram (function or method) you create should be preceded by a a block comment that describes the purpose and behavior of that subprogram using its parameters, any preconditions (what must be true in order for the method to behave correctly?), return value, and any postconditions (things that are true after the method terminates).

  Example:

```
/* print() traverses a List, outputting each Item to a given ostream.
 *
 * parameter         out, the ostream to which we are printing;
 * parameter         separator, a string containing the value
 *                    being used to separate the Items in the List.
 * precondition      out is open and << is defined for type Item.
 * return            out (modified).
 * postcondition     out contains copies of the Items in my dynamic array,
 *                    with each Item preceded by separator.
 */
ostream& List<Item>::print(ostream& out, const string& separator) const {
   Node * nodePtr = myFirstNodePtr;

   while (nodePtr != nullptr) {
      out << separator << nodePtr->myItem;
      nodePtr = nodePtr->myNextNodePtr;
   }

   return out;
}
```

  One of the reasons for writing header and subprogram comments is that there are tools like Doxygen and JavaDoc that will use your comments to produce HTML pages that you can post on the web for people who use your code. To illustrate, online documentation sites like this C++ Reference, the Java Application Programmer's Interface (API), and similar websites were all produced using these kind of tools. If you think you might be a software developer someday, documenting your code is a good habit to cultivate now.

Coding-Style and Documentation Requirements

- **Inline Comments**. Inline comments are 1-line comments used within your code to explain any lines that require explanation or are tricky. If you have used descriptive identifiers, you should not need too many of these, but don't hesitate to include them.

  Example 1:

```
bool isOdd(int value) {
   return value % 2;      // value%2 == 1 (true) for odd numbers
}
```

  Example 2: The length of a line of code should never exceed 100 characters. If an inline comment is too long to fit at the end of a line, put the information on the preceding line(s).

```
bool isOdd(int value) {
   // value%2 == 1 (true) for odd numbers; 0 (false) for even numbers
   return value % 2;
}
```

  Inline comments should always add new information and not just repeat the obvious:

  Useless comments:

```
   windowHeight = 400;    // set windowHeight to 400
   windowWidth  = 800;    // set windowWidth to 800
```

  Helpful comments:

```
   windowHeight = 400;    // reset window dimensions
   windowWidth  = 800;    //  to their default values
```

  One guideline for inline comments is that they should not explain *what* you are doing, but should explain *why* you are doing it.

5. **Parameter-Passing Modes**. A parameter's declaration should correctly indicate how information will flow through that parameter, from the caller into or out of the function, as shown below:

Kind of Type \ Information Flow	**In Only**	**Out or In+Out**
Primitive Type (e.g., `char`, `int`, `unsigned`, `float`, `double`)	Pass-by-value	Pass-by-reference
Class Type (e.g., `string`, `vector`, `list`, `stack`, `queue`, `set`, `map`)	Pass-by-const-reference	Pass-by-reference

6. **Finally**.

   Graders in each of my courses (CS 112, 214, 374, etc.) will expect you to follow these style & documentation requirements in any code you submit. The grade sheet for each project assigns points for style & documentation; if you wish to earn these points, make sure your code meets these requirements.