Tips to Writing Clean Code

By Frank McCown

Functions:

- Function names should usually consist of a verb and a noun which describe the function's purpose. Capitalize the first letter of each word in the function. *Example:* **`Print()`** vs. **`PrintStudentList()`**

- A function should have a single purpose. *Example:* Figuring an average, printing a student list, and getting input in one function vs. 3 functions which perform each of those actions.

- Functions which return a single value should generally do this through a return statement, not through the parameter list.

- A function that won't fit on a single printed page is typically too complex/long and should be broken into smaller pieces.

- Every function should have a header containing an accurate description of what the function does, what all parameters are used for, and all possible return values. Someone reading your code should not have to search through the function's body to understand what the function is really doing. *Example:*

```
/* This function takes two parallel arrays: an array of IDs and an array of
 * scores. The third parameter is the ID to be searched for. The score
 * corresponding to the search ID is returned.  If the ID cannot be found in the
 * IDs array, -1 is returned.
 */
```

Code:

- Use plenty of white space to clarify code.

- Indent properly to show structure, and be consistent with your indentation throughout the entire program.

- Put statements ending with semicolons and { } on separate lines (except for loops).

```
for (i=o;i<5;i++)
{ cout<<i*j; j=j-3; }
```

vs.

```
for (i = 0; i < 5; i++)
{
  cout << i * j;
  j = j - 3;
}
```

Error Checking:

- Give user useful feedback in error messages, and don't scare user away with harsh or ambiguous messages. *Example:* **`"Illegal value!"`** vs. **`"Please enter a number between 1 and 10."`**

- Ensure code is free of potential run-time errors like division by 0, strcpy/strcat with a NULL string, accessing arrays out of bounds, etc.

- "Dummy-proof" your programs by making them as robust as possible. *All* input which is received from an external source should be validated. *Example:* Making sure a number entered was between 1-10 or ensuring an input-file exists before trying to read from it.

Variables / Constants:

- Carefully chosen variable names and constants will often be self-documenting.
  Example: `c = a - b;` vs. `score = total_points - points_missed;`

- Declare all variables at the top of the function.

- Variables should generally not be initialized to a computed value on the same line on which it's declared although initialing to a constant is acceptable. *Example:* `int c = sqrt(b);` vs. `int c = 0;`

- Comment the purpose of each variable declaration.

- Constants should generally be all capitalized, and variables should generally be all lower case (camel-casing is also acceptable).
  Example: `const int NAME_LEN = 4;` and `int num_of_students = 20; // or numOfStudents`

- Use constants for all "magic numbers" or whenever a particular value is used more than once.

Documentation:

- Write comments that express more of the *why* than the *how*. Document the code's intent.
  Example: `// Divide total grades by total students`
  vs. `// Compute student grade average`

- Always place your name, date, and program description at the top of the program.

- Make your comments say something about the code that the code can't say about itself.
  Example: `level++;  // Add 1 to level`
  vs. `level++;  // Advance to next level before processing remaining lives`

- Typically a comment should refer to a logical grouping of lines rather than a single line of code.

- Avoid all but the most common abbreviations in comments so they're easy to read.

- Clearly separate comments from code.

- Use a commenting style that is not overly tedious, time-consuming, or a maintenance monster.

- Outline your code with comments before you write it rather than doing all of the documentation at the end. You'll save yourself time from having to figure out tricky places or forgetting details, assumptions, and subtleties of design.