Programming Style Tips

Software developers should write code that is:

- Reliable
- Easy to understand
- Robust (considers all contingencies)
- Easy for other developers to use
- Easy to maintain and update

Coding standards and guidelines help to achieve these goals.

Many of the following standards and guidelines are derived from the reference books *C++ Programming Guidelines* and *Effective C++.*

General

- C++ programs should be written in a straightforward manner. Good programmers strive for simple, easy-to-read, and understandable code. Avoid bizarre or overly cute usages.

- Use a named constant instead of a constant value (or so called *magic* number) to make programs clearer and easier to modify. For example, confusion may result if the constant 20 is used to represent multiple entities (e.g., the number of user inputs and some unrelated maximum limit). Also, if the number of user inputs changes, to say 30, only the constant variable need be changed.

- Each function should perform a single, well-defined task.

- Replace repetitive statements by calls to a common function whenever the function-call overhead is not too costly in terms of efficiency. This will improve readability and maintainability (easier to find bugs, make changes, etc.) of the code.

- In general, opt to pass modifiable arguments by reference, small non-modifiable arguments by value, and large non-modifiable arguments by using references to constants. This balances performance and clarity since call-by-reference exposes an argument to corruption and the copying involved in call-by-value of large objects can degrade performance. That is, for passing large objects, use a constant reference parameter to simulate the security of call-by-value and the efficiency of call-by-reference.

- To avoid errors caused by the multiple inclusion of a header file, use the name of the header file with the period replaced by an underscore in the #ifndef and #define preprocessor directives of the header file.

Documentation/Commenting

- Your program should be self-documenting; that is, a listing of your source code should provide the key information in English, such that the information will help others (and often yourself!) understand your program.

- Each file should begin with a comment section giving the name of the file, the author's name, the date the file was last modified, and a description of its purpose. In addition, it is also consider good programming practice to include any assumptions you have made (such as assumptions about reasonable input) and any error-checking that has been provided. For example,

```
/*
        filename: make_change.cp
        author: L. Stauffer
        date last modified: 8/17/98

        This program accepts a dollar amount specified by the user and displays the minimum number of
        $10, $5, and $1 bills required to make up the amount.
*/
```

- Each function prototype and function definition should be fully commented. The comments should include a summary of what the function does, preconditions (conditions assumed when the function is called) and postconditions (describing the effect of the function call). For example, consider the function prototype fahrenheit_to_celsius:

```
double fahrenheit_to_celsius( double F );
// Receives a Fahrenheit temperature F. Converts F to Celsius and returns the result.
// Precondition: F is a temperature expressed in degrees Fahrenheit.
// Postcondition: Equivalent temperature of F in degrees Celsius is returned.
```

- Insert comments in your program. A comment should appear after each declaration and after a statement to provide information which is not obvious. For example:

```
int total_cost, item_count;     //total cost of item_count purchases
double average_cost;            //average cost of an item, to be computed
```

  You may also use a comment to precede a block of code:

```
//perform linear search of costs array for a matching value

int i = 0;
while ( i < SIZE && costs[i] != search_val )
        i++;
```

Formatting

- Group related code together.

- Use consistent and reasonable indentation conventions throughout your program to improve program readability. Avoid using several spaces for an indent - instead use a tab. If there are several levels of indentation, each level should be indented the same additional amount of space.

- Indent the body of a control statement (if else, switch, for, while, do while) to emphasize structure and enhance readability.

- Place braces {} to emphasize program structure. There are two notable styles of brace placement. The first is to place each brace on a separate line (the book follows this style). The second places the opening brace at the end of the line just before the start of the braced section. The following code piece illustrates these two styles:

  Preferred style:

```
        if ( x > MAX )
        {
                cout << "x is larger\n";
                x = MAX;
        }
        else
        {
                cout << "x is smaller\n";
                x = 0;
        }
```

  Alternative style:

```
        if (x > MAX) {
                cout << "x is larger\n";
                x = MAX;
        }
        else {
                cout << "x is smaller\n";
                x = 0;
        }
```

- Nested *if* statements tend to quickly migrate to the right following a simple indent each control structure approach. If you have a situation which requires multiple nested if statements, instead of doing:

```
if ( dayOfWeek == 1 )
        // do this;
else
{
        if ( dayOfWeek == 2 )
```

```
                // do that;
        else
        {
                if ( dayOfWeek == 3 )
                        // do something else;
                etc...
```

Do this instead:

```
if ( dayOfWeek == 1 )
        // do this;
else if ( dayOfWeek == 2 )
        // do that;
else if ( dayOfWeek == 3 )
        // do something else;
```

Or, use a **switch** statement. Either will save space and be easier to read, while producing the same results as the nested ifs.

- Indent the entire body of each function one level of indentation within the braces that define the function body.
- Add white space (i.e., blank lines and or spaces) to improve readability. In general:
  - Use one blank line to separate logical chunks of code. Avoid using more than one blank line.
  - Always place a blank line before a declaration that appears between executable statements. If you prefer to place declarations at the beginning of a function, separate them from the executable statements with a blank light. This highlights where declarations end and executable statements begin.
  - Put a blank line before and after each control structure.
  - Place a space after each comma (,) to make programs more readable.
  - Place a space on either side of a binary operator. This makes the operator stand out and the program easier to read.
  - Here is an example:

```
void AddNumbers( double, double );
void MultNumbers( double, double );
void OutOpenMessage();
char InCommand();
-space here
-space here
int main()
{
        double firstDouble = 4.2;
        double secondDouble = 6.5;
        char command;
        bool valid = false;
        -space here
        -space here
        OutOpenMessage(); // outputs opening screen message
        -space here
        do
        {
                command = InCommand(); //returns user input command
                -space here
                switch( toupper( command ) )
                {
                        case 'A':
                                AddNumbers( firstDouble, secondDouble ); //outputs sum
                                valid = true;
                                break;
                                -space here
                        case 'M':
                                MultNumbers( firstDouble, secondDouble ); //outputs product
```

```
                                valid = true;
                                break;
                                -space here
                        default:
                                cout << "Input error" << endl;
                }
        } while( !valid );
        -space here
        return 0;
}
-space here
-space here
void AddNumbers( double first, double second )
{
        double sum;
        -space here
        sum = first + second;
        cout << first << " + " << second << " = " << sum << endl;
}
```

- Declare each variable on a separate line. This format allows for placing a descriptive comment next to each declaration.

Global vs. Local Variables

- Declare variables to be local to the function(s) in which they are used. Global variables and constants are counter-productive to the goal of encapsulation. Also, a global variables runs the risk of being inadvertently altered in a function in which it is not meant to be used.

- Place named constants as local as possible (this contradicts the advice given on page 143 of our text). When then should a constant be made global? There is no hard and fast rule to apply here. Judge the use of the constant - if it is used in many functions throughout the program (e.g. the tax rate in an accounting program) then it makes sense to declare the constant globally at the beginning of the file.

- Do not use global variables to get around the need to pass parameters.

Code Efficiency

- A nested if/else statement can be much faster than a series of non-nested if statements because of the possibility that an early if condition is satisfied.

Naming

- Use meaningful and descriptive identifiers for variable, function, class, constant and parameter names. This can make a program easier to understand.

    - Use all uppercase letters, with underscores, for #ifndef/#define/#endif and const identifier names. For example:

      ```
      #ifndef TIMEFILE_H
      #define TIMEFILE_H

      const int MAX_LENGTH = 10;
      ```

    - For all other identifier names, use either one of the mixed-case conventions for multiple words or all lower-case letters, with underscores.

      ```
      FinalScore
      finalScore
      final_score
      ```

    - Boolean variables (those that evaluate to true or false) should be named so that they state the affirmative action. For example:

      ```
      bool notValid; // Avoid this
      bool valid; // Use this affirmative form instead
      ```

If variables are named to assert the negative, then somewhere code will end up looking like this:

```
if(!notValid)  // Double negative is confusing
```

- Be consistent with your naming scheme.

- Avoid using the same names for the arguments in a function call and the corresponding parameters in the function definition. This reduces ambiguity.

Classes

- It's considered better style to list all the public members of a class first in one group and then list all the private members in another group. This tends to focus the attention of the user of the class on the public interface rather than on the inaccessible implementation.

- Every class should have a constructor to ensure that every object is initialized to a well-defined state.

- Declare as const all member functions that do not need to modify the current object so that you can use them on a const object if necessary.

Prepared by Lynn Stauffer and Karen Petersen, 1999