C++ Coding Standard and Program Style

All the code you write for this class must adhere to the following coding standard. It is not a recommendation, it is a requirement. Clarity, simplicity, and organization are the goals. The detailed coding standard is discussed and a program style template is also provided in this documentation.

Structure

Overall program structure is as follows:

- program header block (program documentation)
- #include statements
- const declarations
- type declarations and typedef statements
- function prototypes
- class interfaces followed by their implementations
- function implementations, ordered alphabetically or in calling order
- main() function goes first or last, not somewhere in the middle

Documentation

The program must have a header block containing:

- a line of dashes or asterisks at the top and bottom of the header block
- the name of the program
- the name of the author
- the class for which it was written: course number, section, and title
- the name of the course instructor
- the due date and the date(s) it was actually written
- a brief but comprehensive description of the program (i.e., program requirements)
- a brief description of program usage, including all external resources needed by the program (e.g.,
- data files, output files)
- any special compilation or linking instructions
- program errors (anything that does not work as specified)

Every method and function (including main) must have a header block containing:

- a line of dashes or asterisks at the top and bottom of the header block
- the name of the function
- a brief description of the purpose of the function
- a description of parameters

```
Parameters:    ifstream &fin          - file to read the values from
               int n                  - the number of values in the file
```

Blocks of code need short descriptions to identify their purpose. For example:

```
// fill the array with values 1,2,3,…,n
for ( i = 0; i < n; i++ )
        list[i] = i + 1;
```

Inline comments should be used to clarify anything that is not immediately obvious in the code:

```
roll = rand() % 12 + 1;          // compute random roll of dice
```

White space should be used liberally:

- Put at least one blank line between logically connected blocks of code, such as loops.
- Put at least 2 blank lines after the end of a function or a program section.
- Put space around parentheses, and after comma-separated lists:
  ```
  Compute_sum(  list,  n );
  ```
- Binary operators need one space on both sides:
  ```
  x = 3 + y;              Not: x=3+y;
  ```
- Logical expressions need one space on both sides:
  ```
  if ( x < 10 && y > 13 )  Not: if(x<10&&y>13)
  ```

Indentation

- Indent 4 spaces (or use Tab) for each nested block level.
- Indent continuation lines.
- The { is aligned below the first character in the preceding line, and the } is aligned with the matching {.
- The { and } are alone on a line except for a possible appended comment:

```
for ( i = 0;  i < num;  i++ )
{
        if ( i % 2 == 0 )                       // if even i value
                list[i] = i * i;                // store i squared
        else                                    // else if odd i value
                list[i] = ( i + 1 ) * ( i + 1 );    // store i+1 squared
        cout << list[i] << endl;
}
```

- Switch statement blocks are aligned as follows:

```
switch ( day )
{
        case 1 : // handle Monday
                cout << "Today is Monday" << endl;
                break;
        case 2 : // handle Tuesday
                cout << " Today is Tuesday " << endl;
                break;
}
```

- Position inline comments to enhance code readability; if possible, neatly lined up at the same column to the right of the code.
- No line should exceed 80 columns in length.
- These indentation rules also apply to struct/class members:

```
struct rec
{
        int id;
        char name[30];
        float salary;
};

class abc
{
    public:
        abc();
        ~abc();
        int Get_x();
        int Get_y();
    private:
        int x;
        int y;
};
```

Variables and other identifiers

- Named objects in your program (variables, functions, etc.) must be given meaningful names. For example, sum is a good name, s is not. On the other hand, x is a fine name if it refers to the x coordinate or is the name of the variable used in a well-known formula. Loop index variables named i, j, k are fine (but avoid lower case L, it's impossible to distinguish from the integer one).
- Begin all variable names with a lower-case letter, and begin all function names with an upper-case letter. Use the underscore character to enhance readability of identifiers (sum_of_squares). Avoid use of all upper case (except for symbolic constants and global variables; see below).
- Symbolic constants should use all upper-case letters:

```
#define MAX 1000
const float PI = 3.14159;
```

- Global variables should be used very sparingly, and should always be commented. If you cannot justify the need for a global variable to your instructor, points will be deducted. To highlight global variable usage in your code, use all caps surrounded by underscores for global variable names:

```
int _HASH_TABLE_SIZE_ = 1023;
```

- Variables are not vanity plates, so avoid cryptic abbreviations. What is cute to you may well be annoying or incomprehensible to someone else.

- Variables must be declared and initialized at the top of each function. List variables alphabetically, or alphabetically within each data type. With the exception of loop indices or a series of very closely related variables, place only one variable declaration per line. Include a descriptive comment describing the variable to aid in readability and debugging. Initialization of variables at declaration time is required.
  For example,
  ```
      int max_num, min_num, average, median;
  ```
  is not an allowable declaration. The correct declaration is :
  ```
      int average = 0;        // contains the average of all random numbers
      int max_num = 0;        // contains the largest random number
      int median = 0;         // contains the median of the random numbers
      int min_num = 0;        // contains the smallest random number
  ```

Expressions

- Learn your operator precedence rules and try to avoid unnecessary parentheses. However, a complicated expression may be "over-parenthesized" to make its meaning more clear. For example, the inner sets of parentheses following the "if" are unnecessary but permissible:
  ```
      if ( ( x < y ) && ( y < z ) )
          cout << x << "is the largest value" << endl;
  ```
- Use temporary variables to break long, complicated expressions into shorter, simpler subexpressions for greater readability, easier editing, and debugging.
- Only one C++ statement is allowed per line.
- Goto statements are not allowed. Continue and break statements should be used sparingly. In general, the only exit from a loop should be a result of the test at the top (or bottom) of the loop.

C++ Classes

Make a habit of separating the class interface from the implementation. The class interface contains declarations for data members and members function prototypes, not executable code. It should be placed in a header file (.h extension), and #included in all files that use the class. The class implementation contains the member function definitions. It should be placed in a C++ source code file (.cpp or .cc extension), compiled separately, and the resulting object module should be linked in with your program. Class implementation files should NOT be #included like header files. (The only valid exception to this rule involves template classes. Many C++ compilers/linkers cannot correctly manage separate compilation of template classes, so #including the entire template class may be your only option.)

Runtime requirements

· Every request for input must be preceded by a descriptive prompt.
· All output must be labeled and neatly formatted.

A Program Style Template is provided for you to follow:

Program Style Template

```
/*============================================================================
Program:
Author:
Class:
Instructor:
Date:
Description:
Input:
Output:
Compilation instructions:
Usage:
Modifications:
Date Comment:
---- -------------------------------------------------
============================================================================*/
#include <header file1>
#include <header file2>
...
/*====================== global symbolic constants ===================*/
/*====================== global type definitions =====================*/
/*======================= function prototypes ========================*/

return_type function_name (param_type,  param_type,  ...);
...
[template for each function definition]

/*============================================================================
Function:
Description:
Parameters:
============================================================================*/

return_type function_name (type param1,  type param2,  ...)
{
        // variable declarations
        data_type1
            var1, // what this variable is for ...
            var2, // what this variable is for ...
        ...
        data_type2
        ...
        // major comment block
            [block]
        // major comment block
            [block]
        return expression;
}
```