C++ Programming Style Guide - CS 161

Goal: Our goal is to produce well-written code that can be easily understood and will facilitate life-cycle maintenance. These rules lay out principles that C++ programmers have found useful for producing code that contains fewer bugs, is easier to debug and maintain, and that others can understand well. While these rules do not represent coding practices that are *required* by the C++ compiler, it is required that you follow these rules during this course.

Layout and Comments:

1) Each file will begin with a file header block. This block should contain, as a minimum, an overview of the code in the file, the author, date created, date and summary of modifications.

```
// ****************************************************************
// Rational.cpp
//
// Summary:  This program determines whether or not a value input by the user
//       is a rational number. A rational number is one that can be represented as
//       a fractional number, with an integer numerator and an integer denominator
//       that is not zero.
// Author: Pam Wiese
// Created: Jan 8, 2017
// Summary of Modifications:
//       Jan 9, 2017 – PW – corrected bug in input validation
//       Jan 19, 2017 – PW – added function Fraction to display value as a fraction
//
// ****************************************************************
```

2) Statements within code files (.cpp) should be ordered as follows:
   a) File header comments
   b) Any required #include statements (only libraries for C++ should be used)
   c) Namespaces to be accessed (only std should be included)
   d) Declaration of function prototypes
   e) main( ) function
   f) Other required functions

3) An appropriate amount of comments should be used throughout your code. Although it is most common for programmers to provide too few comments, over-commenting can also negatively impact the readability of the code. It is important to comment on any sections of code whose function is not obvious (to another person). For algorithmic-type code which follows a sequence of steps, it may be appropriate to summarize the algorithm at the beginning of the section (perhaps in the function header) and then highlight each of the major steps throughout the code. Any C++ style comments may be used. However, the same style should be used throughout your program. Comments should be indented at the same level as the code they describe.

4) Functions should always have a return_type. For functions not returning a value, use void type. All functions should complete with an explicit return statement.

Naming Conventions:

5) Use descriptive names for variables and functions in your code, such as xPosition, distanceToGo, or findLargest(). The only exception to this may be loop iteration variables, if no descriptive name makes sense. In this case, use the lower case letters beginning with i (i, j, k, etc).

6) For variable and function/method names, the first word should be lowercase, and subsequent words should be capitalized ("camel case"). For example, upperLimit, averageValue, makeConnection(), addBody(). Whenever possible, function/method names should be verbs: draw(), getX(), setPosition().

7) No variables and functions should have the same names in any parts of the program.

8) For constants, capitalize all letters in the name, and separate words in the name using an underscore, e.g., PI, MAX_ARRAY_SIZE. Any numeric constants needed in your code should be replaced by a named constant.

9) For Boolean variables and functions, the name should reflect the Boolean type, e.g., isEmpty (), isLastElement, hasChanged. Names should always be positive; i.e., use hasChanged rather than hasNotChanged.

Statements:

10) Only one statement is allowed per line, and each line of code will be no more than 70 characters in length (to prevent line-wrap). If a statement requires more than one line, subsequent lines will be indented to make it obvious that the statement extends over multiple lines, as shown below.

```
int myFunction (int variableA, int variableB, int variableC, int variableD,
                int variableE, int variableF );

cout << "Please enter values to represent the X and Y coordinates of a "
     << " single point on a matrix." << endl;
```

11) Each variable declaration is to be on a separate line.

```
int id, grades;              // bad
int height, weight = 5;      // bad

int id,
    grades;                  // good
int height;                  // good
int weight = 5;              // good
```

12) Variable declarations should be separated from function calls.

```
int value = getValue();          // bad

int value;                       // good, though variable declarations should be in their own
value = getValue();              //       section, if possible
```

13) Braces can be done in one of two styles: the opening brace can be put on the end of the line defining the block, or it can be on a separate line by itself, as shown below. Code inside the braces will always be indented. Either style is allowed in this course. However, your code must be consistent. Do not mix braces styles.

```
for (i = 0; i <= maxSize; i++)   {
    …
    …
}
```

```
for (i = 0; i <= maxSize; i++)
{
    …
}
```

14) Make wide use of horizontal white space. Always include spaces after commas, and between operands and operators in expressions. Do not use tabs to create white space.

```
x=y*2;          // bad
x = y * 2;      // good
```

15) Use vertical white space to separate your code into "paragraphs" of logically connected statements. White space helps clarify the connectedness of blocks of code and makes placement of comments logical and easy to manage. Do not double-space all lines of code.

16) Use parentheses liberally for clarity in mathematical expressions.

```
totalCost = quantity * price - discountPercent * price;        // bad

totalCost = (quantity * price) - (discountPercent * price);    // good
```

17) Avoid implicit tests for zero, except for Boolean variables

```
OK:         if (isEmpty) …
            if (!isEven() ) ...
NOT OK:     if (counter) ...
            if (!counter) ...
```

18) Avoid "slick code." For example, for statements should typically only include initialization, test, and increment of iteration variable, not additional statements which belong in the loop body.
Avoid: for (i = 0, sum=0; i <= maxCount; i++, sum+=i*4) {…}

19) Code should include only concepts covered in class. The methodology used to teach programming skills in this course may differ from code seen in discussion boards. Students are to follow and use *only* the programming style and concepts presented in the course.

20) For if statements, the nominal or more frequent case should be placed in the then block, and the exceptional or less frequent case in the else block.

21) The goto statement shall not be used.

22) while (true) shall not be used.

23) Lines of code to be executed in the block of a conditional or repetition statement should not appear on the same line as the condition.

```
if (value > 1000) { cout << "big"; value = 0; }  // bad

if (value > 1000) {                              // good
    cout << "big";
    value = 0;
}

if (value > 1000)                                // good
{
    cout << "big";
    value = 0;
}
```

24) Selection and repetition structures do not need to have braces, if the code is only one line. Whether or not the braces are used, the lines of code in selection and repetition structures must be indented.

```
if (value == 0)                                  // bad
cout << "You have entered a zero.";

if (value == 0) cout << "You have entered a zero.";   // bad


if (value == 0)                                  // good
        cout << "You have entered a zero.";

if (value == 0)                                  // good
{
        cout << "You have entered a zero.";
}

if (value == 0) {                                // good
        cout << "you have entered a zero.";
}
```