CS 215 - Fundamentals of Programming II
C++ Programming Style Guideline

Most of a programmer's efforts are aimed at the development of correct and efficient programs. But the readability of programs is also important. There are many "standards" in use. Since following them will not guarantee good code, these "standards" are actually guidelines of style. Each one has its proponents and detractors. This document gives some style requirements as well as giving the documentation requirements for this course. The essential point is that a program is a medium of communication between humans; a clear, consistent style will make it that much easier for you to communicate.

Program Documentation

While this course does not require a separate analysis and design document, much of what would have been in such a document is required to appear in the comments of your program files.

Makefiles

Makefiles should begin with a comment section of the following form and with the following information filled in:

```
# File: <name of file>
# Class: CS 215                    Instructor: Dr. Deborah Hwang
# Assignment:                      Date assigned:
# Programmer:                      Date completed:
```

Code File Headings

Every code file should begin with a comment section of the following form and with the information filled in as appropriate:

```
// File: <name of file, e.g., main.cpp>
// <Description of what is in this file.>
//
// ------------------------------------------------------------
// Class: CS 215                    Instructor: Dr. Deborah Hwang
// Assignment:                      Date assigned:
// Programmer:                      Date completed:
```

The description should be short, but meaningful, at a minimum the following should given:

Type of code file	**Description**
Main program file	What the program does
Library header file	What kind of functions
Class definition header file	What the class represents
Class implementation file	What the class represents

Class Files

Class definitions and implementations should be divided into multiple files so that we can reuse them easily. By convention, the class definition and related free function prototypes are stored in a *header* file which has the extension "`.h`". The implementations of the class member, friend, and overloaded operator functions along with any helper functions are stored in a *source* file with the extension "`.cpp`". The main program usually is stored in a separate source file that includes the header files for each class used.

Every class header file should use compiler directives `#ifndef`, `#define`, and `#endif` to ensure a header file is only included once. The symbol should be the name of the header file in all capital letters with an underscore (_) replacing the dot and a trailing underscore. For example, for the header file `counter.h`, the directives would be:

```
#ifndef COUNTER_H_
#define COUNTER_H_

class Counter
{
   ...
};  // end Counter

#endif  // COUNTER_H_
```

Preprocessor Section

In the preprocessor section, include statements for header files should have comments indicating the types, constants, variables, or functions used from the header if they are not commonly used. For example,

```
#include <cmath>       // sin, cos
#include "rational.h"  // Rational class
```

Class Definitions

The qualifiers **public** and **private** should be indented and the member and friend function prototypes and data member declarations indented in from the qualifiers. Free function prototypes should appear after the class definition. For example,

```
class Counter
{
   public:
      // Constructors
      Counter ();             // Creates counter with value of 0

      // Accessors
      int GetValue () const;  // Returns the current value

      // Mutators
      void SetValue (int newValue);
      // Member overloaded operators
      Counter & operator++(); // Increments counter by 1

      // Friend overloaded operators
      friend bool operator< (const Counter & leftOperand,
                             const Counter & rightOperand);
         // Returns true if left operand value < right operand value
      ...
   private:
      int value;           // Current value of the counter
};  // end Counter

// Free overloaded I/O operators
istream& operator>> (istream& in, Counter& theCounter);
ostream& operator<< (ostream& out, const Counter& theCounter);
```

Member and friend function prototypes should appear in the same order as presented in the class specification. If there are many member functions, comments indicating grouping are helpful. E.g., constructors, accessors, mutators, I/O, etc.

Short comments describing what member functions do and what member data represent should be included as shown above.

Member functions that do not change the state of the object **must** be declared as `const` as shown above for the **`GetValue`** member function.

Class member, friend, and overloaded function implementations should appear in the same order as presented in the class definition.

Function Headings

Include a short description for each function (not including `main`) in a comment block before each function **prototype**. Place each function parameter on a separate line and annotate with its analysis (i.e. movement and a short description). For example:

```
/* Function: Sample
   Returns: <description of what is computed> OR
   <description of what function does if no returned value>
*/
type1 Sample (type2 arg1,  // Received: description of obj1
              type3 &arg2, // Passed back: description of obj2
              ...);
```

“Received” may be abbreviated “Rec'd” and “Passed back” may be abbreviated “P'back”.

Alternatively, an analysis table can be used. For example:

```
/* Function: Sample
   < A short description of what function does >
   Objects               Type     Movement        Name
   ---------------------------------------------------
   Description of obj1   type2    received        arg1
   Description of obj2   type3    passed back     arg2
   Description of result type1    returned        result
*/
type1 Sample (type2 arg1, type3 &arg2, ...);
```

Function comment headers should be included with member function prototypes as well. If there are many function prototypes, comments indicating grouping are helpful. (E.g., accessors, mutators, I/O, etc.)

Identifiers

Identifiers should be chosen to be self-documenting. Abbreviations, except where standard, should be avoided. In addition, comment variables when usage is restricted. (E.g., an integer used to represent a calendar month, so its valid values are 1-12.)

Two styles of identifiers are acceptable for this course. However, **only one style must be used** consistently within **all files** comprising a project unless otherwise required. Using both styles in one project will incur grading penalties. The styles are as follows:

- The style used in this document and often by the instructor is a style that was in vogue in the 80's and 90's and often is called "camel case". Each word in a function identifier, class identifier, or structure type identifier should start with an uppercase letter. E.g., `FindMinIndex` or `PointType`. Each word except the first one in an variable identifier should start with an uppercase letter. E.g., `firstName`. No underscores should be used in function, class, structure, or variable identifiers. Constant identifiers should be written in all uppercase with words separated by an underscore (_). E.g., `MAX_STRING_SIZE`.

- The style used by most current professional programmers is the original style advocated by the inventors of C. All identifiers except constant identifiers are in all lowercase with underscore between each word. E.g., `find_min_index`, `point_t`, or `first_name`. Constant identifiers should be written in all uppercase with words separated by an underscore (_). E.g., `MAX_STRING_SIZE`.

Commenting

Local variables in functions should be commented when usage is not apparent. That is, when the identifier does not completely describe the purpose of the variable.

Comment the ending curly brace with `// end <description>` whenever the body is **more than 10 lines long.** This applies to function bodies, selection bodies, and loop bodies. For example:

```
// Function: Try
// Attempts to do some things...
void Try (...)
{
   ...
   while (!done)
   {
      // more than 10 lines of code
      ...
   }  // end while not done
   ...
} // end Try
```

A sufficient number of design steps should be given in comments so that the reader can determine WHAT is being done or WHY it is being done without reading the actual code in detail (which describes HOW something is being done). For very short or very simple functions, the function description will suffice. The code that implements a step should immediately follow the comment for the step. In addition, comment complex code to improve clarity. Note that a comment should not be the meaning of the code written in English. For example,

```
   // Adjust i to point to the end of the previous word:
   i = i - 1
```

is better than

```
   // Subtract 1 from i:
   i = i - 1
```

Comments should be in good English. Grammar and spelling should be correct.

Abbreviations in comments should be used sparingly, and then only those that would be found in a standard dictionary.

Constants and Variables

Constants

Constants must be declared using the `const` keyword. The use of `#define` to declare constants is **not** allowed.

Constant values in your algorithm should be replaced by constant identifiers in your program. Exceptions should be made only when the constant conveys its own meaning, such as 0 as an initial value for a sum or to start a count, or is part of a constant mathematical formula, such as 2 in $2\pi r$.

All constants should be defined after any include statements and before any function prototypes and the main program function, even if they are used only in one function.

Variable use

Each (non-class attribute) variable identifier that occurs in a function should be **local** to that function - that is, declared in the function's header or in the function's body.

1. If the variable may have its value changed in the body of the function and that new value will be needed back in the calling program (i.e., it is passed back), then the variable should be declared as a reference formal parameter. (Sometimes data is both received and passed back.)

2. If the variable gets its initial value from the calling program but does not send a different value back (i.e., it is only received), then the variable should be declared as a value formal parameter if it is a primitive type, except for C++ arrays, which are automatically passed as reference parameters. Due to concerns of the inefficiency of copying , large data like structs or class objects should be declared as reference formal parameters. Such reference formal parameters should be declared as `const`. For example,

   ```
   int Function1 (const vector<int> & v);  // REC'D: list of scores
   ```

3. If the variable does not get its initial value from the calling program and does not pass its value back (via a parameter), then the variable should be declared as a local variable of the function. This generally includes the returned object, if any. Local variables should be declared just before first use. If at all possible, they should be initialized when declared.

NEVER use a global variable in this course unless explicitly told otherwise. While there are valid reasons for having global variables, they generally should be avoided and **will not** be tolerated in this course.

Program Formatting

Any preprocessor statements (`#include`, `#define`, etc.) should be at the beginning of a file (after the program file heading comment). Any global constants should follow, then any type definitions (`typedef`, `struct`, etc.). Finally, function prototypes are listed. No other statements should appear outside a function body unless explicitly allowed.

Function headers should start at the left edge.

There should be a blank line between each function definition. A blank line should be used to separate logical sections within a program or function. In general, blank lines should be used wherever their use will improve readability.

Indenting should be used to convey structure. Indentation should be at least 3 spaces, and generally no more that 6 spaces unless otherwise noted. Indenting should be consistent. (See notes at the end of this handout regarding emacs auto-indenting and printouts of vi files.)

For declarations, each identifier should be declared on a separate line. The type identifier should be indented, and the variables of that type should line up under each other. Use commas to separate variable identifiers of the same type.

For example, in the main function, we might declare

```
const int MAX_SIZE = 100;
const float PI     = 3.14159;

int main (int argc, char *argv[])
{
   int i, j,                  // Loop indexes
       score,                 // Input: a test score
       tests[MAX_SIZE];       // Array of test scores
   double average;            // Output: average score
   PointType p1;              // Point to test
   ...
}  // end main
```

Comments should explain the purpose of each variable where appropriate, and should line up with the code being described.

A statement should not be longer than will fit on a printed page (about 80 characters wide). If a non-I/O, non-function call statement must be continued on more than one line, the second line should be indented to at least 6 spaces and successive continuation lines should be aligned with the second line of the statement. For example, we might write

```
   while ((('a' <= line [i]) && (line[i] <= 'z'))
         || (('A' <= line[i]) && (line[i] <= 'Z')))
      i++;
```

An I/O statement should be broken up so that the << or >> operators line up. For example, we might write

```
   cout << setw(15) << name << setw(30) << address
        << setw(15) << phone << endl;
```

A long function call statement should be broken up so that the function arguments lined up. For example, we might write

```
   Sample (argument1, argument2, argument3, argument4,
           argument5, argument6);
```

For the statement that follows `if`, `else`, `while`, `for`, `do`, and `switch`: the statement should start on the next line and be indented. For example,

```
   ...
   // first and last indexes have crossed, so everything matched
   if (first >= last)
      found = true;
```

Each line of the body of a compound statement should be indented within the curly braces surrounding it. Two styles of curly brace alignment are acceptable for this course. However, **only one style must be used** consistently within **all files** comprising a project with one exception given below. Using both styles in one project will incur grading penalties. The styles are as follows:

- The style used by most current professional programmers is the original style advocated by the inventors of C. The opening curly brace appears at the end of the line introducing the compound statement, and the closing curly brace is lined up under the first line. For example,

```
/* test for the target item */
if (a[middle] == item) {
   item     = a[middle];
   found    = true;
   position = middle;
}
```

- Another style (often used by the instructor and textbooks) is a style that was in vogue in the 80's and 90's that places the opening curly brace under the line introducing the compound statement. For example,

```
/* test for the target item */
if (a[middle] == item)
{
   item     = a[middle];
   found    = true;
   position = middle;
}
```

The only exception to consistent use of one of these styles is that function definitions may use the latter style regardless of the style used for executable statement bodies as is done in the textbook. However, **all** function definitions must use the same style, otherwise grading penalties will be incurred.

Column alignment should be observed for each set of reserved words `if` and `else`. This include multi-branch constructs, for example:

```
if (x > 90)
   grade = 'A';
else if (x > 80)
   grade = 'B';
else if (x > 70)
   grade = 'C';
else if (x > 60)
   grade = 'C';
else
   grade = 'F';
```

Comments that describe one or more statements should be immediately above and aligned with the statement or collection of statements which they describe. For example,

```
/* Consider items starting at index i */
j = i;
/* Swap elements until finding the right place or end of list */
while ((j > 1) && (a[j - 1] > a[j])) {
   /* a[1..j-1] is unsorted and a[j..i] sorted: */
   swap_ints (a[j], a[j - 1]);
   j = j - 1
}
```

At least one space should be used in the following locations within C++ text (this does not apply within comments and character strings):

1. before and after `=`, `//`, any relational, logical, arithmetic, or assignment operator
2. before `(` when not preceded by another `(`
3. after a comma in argument lists, and after semicolon in for-loop headers

A function (including the main program function) should fit on one screen (about 25-30 lines) if possible and must fit on a listing page (about 50 lines). Ideally, an analysis and design will not produce code that is more than a page long, but if the code does not fit initially, introduce one or more new functions to split up the work in logical places.

A note about emacs

The default indenting scheme of the emacs text editor is consistent with putting the left curly brace at the end of the first line. It is not really compatible with the instructor's column alignment style. If you use emacs and want to use the column alignment style, you should download the instructor's `.emacs` configuration file from the course webpage into your home directory. This file is named `emacs` and must be renamed to `.emacs` (with a '.' at the beginning).

Once you have downloaded this file you must exit out of emacs and start it up again to see the effect.

A note about vim

Vim does not have smart indenting using the TAB key like emacs. However, you can configure vim to display tabs as 3 spaces instead of the default 8 spaces. Tim DeBaillie provides a configuration file (`.vimrc`) that may be downloaded into your home directory. A link is available on the course webpage. This file must be named `.vimrc` (note the '.', some browsers will not save a file starting with a dot).