1 Documentation

Take credit for your work. Every program that you write should have a leading comment block that identifies the author and the purpose of the program.

1.1 File Header Block

This header block will be the first part of each of your source files. The header block will consist of the following sections:

Author:	Your name
File name:	The name of the source code file
Date Created:	The date started on the project
Modifications:	Last time the project was changed and what was added or changed

Example:

```
/************************************************************
* Author:          Todd Breedlove
* Filename:        Lab1.cpp
* Date Created:    12/17/96
* Modifications:   1/5/01 – Corrected problem names with spaces
*                  3/29/11 – Corrected A's counting issue
************************************************************\
```

Note: Many students don't see the value of the block of asterisks. However, they help separate the documentation from the rest of the code, especially when the code is printed.

1.2 Main Documentation Header Block

In addition to the information specified above, your main source code file (the entry point to your program) should contain the following information.

Lab/Assignment: Specify the lab or assignment.

Overview: State and describe what the program does.

Input: This section will state where the input for the project is coming from, what the input represents, and any limitations placed on the input. If the input is coming from a file, the filename, type of file (i.e. text or binary), and a sample format of the file will also be given.

While writing this section consider what a colleague or grader may need in order to run your program. This section essentially replaces a "readme" file.

Output: This section states what your program produces as output and where the output is going especially if the output is written to a file.

Example:

```
/*************************************************************
*
* Lab/Assignment: Lab 1 – Grade Program
*
* Overview:
*     This program will read in a name and four grades and
*     calculate the average. The average will then be used to
*     find the number of each letter grade (i.e. 5 A's, 6 B's,
*     etc.). The program will also find the highest average and
*     the lowest average in the class.
*
* Input:
*     The input will consist of a name and four scores for a
*     varying number of people and is read from the file
*     c:\cst136\lab1.dat. The data file is a text file in the
*     format of:
*
*     Todd Breedlove 90 90 90 90
*     Bill Jones 85 65 65 65
*
* Output:
*     The output of this program will be the accumulative
*     number of each letter grade.  The program will also
*     display the name of the person with the lowest and the
*     highest average for the class and their averages. The
*     output will be to the screen and will have the form:
*
*************************************************************/
```

1.3 Function Header Blocks

Each function should be documented to show programmers that are modifying or viewing your code the purpose of this function. Also specify the state of the system necessary before calling (Precondition) this function and how the system changes (Postcondition) because of this function call. This should be placed directly above the function definition.

Example:

```
/*********************************************************************
* Purpose: This function reads the input data for each plant.
*
* Precondition:
*     LastPlantNumber - Declared size of the array plantArray.
*                       Must be a non-negative value.
*
* Postcondition:
*      Returns the number of plants read.
*      Modifies the plantArray.
*
*********************************************************************/
```

1.4 Class Header Blocks

The class header block provides a quick overview of the class and its methods. This should be placed directly above the class definition. Some manager functions like the destructor, op = and copy constructor really don't need to be described.

Think of this documentation from the viewpoint of another programmer using your class and needing to know what methods are available and any pertinent details in their usage without having to wade through your code.

Example:

```
/*************************************************************************
* Class: Array
*
* Purpose: This class creates a dynamic one-dimensional array that can be
* started at any base.
*
* Manager functions:
*       Array ( )
*               The default size is zero and the base is zero.
*       Array (int length, int start_index = 0)
*               Creates an appropriate sized array with the starting index
*               either zero or the supplied starting value.
*       Array (const Array & copy)
*       operator = (const Array & copy)
*       ~Array()
*
* Methods:
*       operator [ ] (int index)
*               ...
*************************************************************************/
```

2 Indentation

To improve the readability of programs and to aid in the debugging process, there should be indentation or tab that indicates different blocks of statements. A tab should only consist of either 3 or 4 spaces. Visual Studio defaults to 4 spaces. An example of a properly indented program is provided below.

```
int main ( )
{
    int num;

    cout << "Please enter a number: ";
    cin  >> num;

    if (num == 0)
        cout << "Zero";
    else
    {
        cout << "The number is greater than 0, or ";
        cout << "the number is less than 0";
    }
    cout << "\n\nDone";

    return 0;
}
```

The curly braces should be placed immediately beneath the line, lined up at the same indentation level. It is easy to see that every opening brace has a closing brace at the same level of indentation. This makes it easier to determine which block a brace is closing. Also, it is **MUCH** easier to verify that every closing brace has a matching opening brace. You may think you have started a block but may have inadvertently left off the opening brace at the end of a statement.

Also notice that the if statement doesn't have curly braces yet the body of the statement is indented. There is an implied level of control with this statement and therefore the body of the if statement should be indented.

3 Whitespace

Add whitespace (i.e., blank lines and or spaces) to improve readability.

3.1 Vertical Space

Vertical space, blank lines, helps break your code in logical chunks. There are no hard and fast rules, but in general:

- Use one blank line to separate logical chunks of code. Avoid using more than one blank line, but DO NOT double space lines of code.

- Place a blank line before a declaration that appears between executable statements.

- If the declarations are at the beginning of a function, separate them from the executable statements with a blank line. This highlights where declarations end and executable statements begin.

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

The most important thing is to be consistent. A consistent programming style is one of the easiest ways to improve the readability of your code.

3.2 Horizontal Space

Whereas vertical space breaks your code in more logical chunks, horizontal space helps the reader to distinguish the symbols or words within a single statement. Sometimes spaces are required by the syntax of the language, but appropriately used spaces can increase the readability of your code.

- Place a space after each comma (,) in a comma delimited list.

- Place a space on each side of a binary operator. However, unary operators are not spaced from their operands.

```
num = 0;
++num;
```

  Exceptions to this rule are the arrow (->), dot (.) and binary scope resolution (::) operators. As a standard programming convention there are no spaces between these operators and their operands.

4 Naming Conventions

Regardless of the identifier (variables, constants, functions or classes), it should always be descriptive of its purpose. This technique is crucial in making your code more readable, maintainable and reliable.

4.1 Variables

All variable names must be in lowercase. Multiple words will be camel cased – the first word lowercased with each successive word having the first character capitalized. Be consistent!

All variable declarations should have a comment explaining the purpose of the variable. Some example variable declarations and comments are provided below.

Example:

```
int fileSize (25);            // size of the input file
char selection;               // users selection of menu item
char fileName[MAX_SIZE];      // the file name of the output file
int i(0), j(0);               // index variables for loops
```

Defining variables with names like x, y, or z, gives no insight into how the variable is to be used or the types of values that can be assigned, unless taken from a well-known mathematical equation, which you should reference explicitly.

In the examples above, the variables i and j are to be used as looping control variables. In many cases these variables are acceptable because the description is sometimes is simply “controlVariable”. However, if there is a better description (row, column, pass) use it. Do not get complacent

4.2 Constants

Constants should be given names in ALL CAPITAL LETTERS with multiple words separated by underscores. This is a very common practice among professional programmers that easily aids in quickly determining the constants in a program.

Constants give names to literal values. This is preferred to using literals which are often called “magic numbers”. If a literal has a conceptual purpose, name it!

Example:

```
const int MAX_SIZE(15);             // maximum records
static const int MAX_LENGTH(63);    // maximum length of string
```

4.3 Functions

Function names must be descriptive of their purpose. Since functions are orders to do something, function names should start with strong verbs. Function names should start with uppercase letters and have uppercase letters at word boundaries. This distinguishes your

functions from standard library functions and also prevents variable name and function name conflicts.

Example:

```
Sort();
GetMenuSelection();
```

Functions should be written to do one purpose and should not be longer than one page, including appropriate documentation. Exceptions will be made to this rule if deemed necessary or appropriate. Check with the instructor.

4.4 Classes

Classes should also be mixed case as seen with functions. Functions start with a verb because they perform a task, whereas classes should be nouns because they are "things". If a class is a pure abstract base class, also known as an interface, you should preface the class name with an "I".

Example:

```
class Time
class ICreature
```

Data members in your class should always begin with "m_" to signify that it is a member of the class.

Example:

```
int m_hour;             // 0 - 23 (24-hr clock)
int m_minute;           // 0 - 59
int m_second;           // 0 - 59
```

5 Statements to avoid

- using namespace …

  This statement can lead to naming conflicts between identifiers in different namespaces. It is better to bring each individual statement into the current scope as shown below.

```
using std::cin;
using ios::right;
```

- break (except in a switch statement)

  Although there are some valid reasons to use the break statement, it is often overused. Valid reasons are listed below.

  - Switch statements require the use of breaks.
  - Embedded systems often have an infinite loop (while (1)) that keeps the program running. A break statement is used to exit this type of loop.

- return

  There are several inappropriate uses of the return statement.

  - Return statements should only return values from functions. Empty returns are not necessary and detract from the continuity of your code.
  - Never return from the body of a loop.
  - Having multiple returns in a function, such as in an if - else if - else statement, is acceptable but not preferred. It is preferred to have only one return statement per function.

- continue

  This statement is unnecessary and should not be used.

- goto

  Never, never, NEVER, use a goto statement. Period.

6 Best Practices

6.1 General

- The C++ standard states that main should return an integer. A zero is often used to signify that your program terminated normally, a negative number usually signifies that your program is exiting in an error state.

- Main should always be the first function in your source code file. All other functions, if appropriate should be placed below main.
- All functions should be declared.
- All file names, including header files, should be all lowercase to facilitate the portability to case sensitive operating systems like Linux.
- Always initialize your variables.

6.2 Classes

- No public data members.
- Use include guards (#ifndef) rather than #pragma once. Although #pragma once is supported by most development environments, it is not part of the standard yet and therefore its behavior is not guaranteed across platforms.
- The names of the .h and .cpp files should be the same as the class name except in all lowercase.
- Avoid friends. The only time a friend class is acceptable is when the classes are so intertwined that you can't have one without the other. For example, a linked list class can't exist without a node class and a node class should not exist outside of a list. Therefore, it is acceptable to have the node class friend the list class. Acceptable, but not necessary. Setters and getters can be used in lieu of friends.
- Use base member initialization whenever you can.
- Make default constructors. If you don't, among other things, you will never be able to create an array of objects from the class.
- The destructor should return your class to its default state. For example, in a list the default constructor should initialize the head data member to nullptr. The destructor should insure that head is one again set back to nullptr.