Standards For Programming Assignments

Shaun-inn Wu

Last Updated: January 18, 2010

General

Credit for each program will be apportioned approximately as follows:

Documentation	25%
Implementation	25%
Correctness	50%

We will discuss documentation and implementation in next two sections. Correctness depends on the assignment, but take note of our discussion on testing. In all cases *explicit instructions given in the assignment must be followed*. In the case of documentation, only a *very good* job will receive all the points possible.

It is better to do part of a problem correctly and get results than to hand in a nonworking program. We will not pore over code while grading a program to find out why the program did not work. Prior to the due day, of course, we will try to provide assistance.

Programs should be your own work. Some collaboration in the conceptual stages is acceptable, but detailed program construction should be done independently. "Detailed program construction" includes writing codes and low-level pseudo-code. In particular, copying code is completely unacceptable.

The output you turn in with your program must be EXACTLY the output produced by the execution of your program. "Fixing up" your output file with the editor is unacceptable.

Documentation

The purpose of documentation is to make the program comprehensible at all levels. Good comments are the classic tool used for this, but do not underestimate the utility and necessity for mnemonic identifiers, blanks, indentation, and proper modularization.

The program need not be buried in comments, but there should be enough to guide. It is not necessary to give line-by-line comments. One good comment can usually cover a block of statements, although occasionally a single statement such as a function call will demand a comment of its own. It is helpful to clarify the meaning of complex expressions. Each comment should explain conceptually what is being done; it should never be a transliteration of the code. (For example, the comment "Set I to I+1" is useless.)

Good variable names are not necessarily short. Try to make programs as much self-documented as possible by using meaningful names. Choosing identifiers well and modularizing a program logically will reduce the need for comments within the code. In fact, it might be argued that the ideal functions comprises a header with comments and 4-10 lines of code in which the variable and function names, in conjunction with the method description in the header, make interior comments redundant (but not undesirable).

Using the format of your program to emphasize the logical structure can increase readability greatly. Use blank lines and spaces within statements liberally to reduce visual density and emphasize logical separations. Develop a reasonable indentation style and use it consistently. This should include indenting the bodies of loops, and the *then* and *else* clauses of conditionals. Also, it is recommended to vertically align } with corresponding {.

All programs must produce well-formatted, well-labeled output no matter whether it is displayed on the screen or written to files during program execution. Here again, blank lines, spaces and indentation are indispensable tools.

All included files (classes, legally borrowed functions, etc.) must be documented about where they are from, what they are and how they are used in those program units including these files.

Each program unit (including classes and functions) should have a comment header. Comment headers should be made easily distinguishable from codes. A common way to do this is to place them into rectangular boundaries consisting of mostly some characters such as * or #. A comment header should contain the **purpose, input and output parameters, main data structures used and implementation methods that achieve the purpose of the program unit.** For the main program unit, **your name, class name, lab session number, the date of design and implementation, user's information and an overview of subparts of the program** should be included into its comment header in addition to the above.

The comment headers can be sketched as follows:

1. Program Header

```
/*****************************************************************/
/* PURPOSE:  a very brief statement about what the program does. */
/* IMPLEMENTED BY:  Your name                                    */
/* Course:  e.g., CS 111                                         */
/* INSTRUCTOR: the name of instructor                            */
/* LAB SESSION:  e.g. Session 1, 9:00 am to 11:45 am             */
/* LAB INSTRUCTOR: the name of lab instructor                    */
/* DATE:  e.g., 1/22/2010                                        */
/* INPUT:  what kind of input from the user is expected by the   */
/*    program and where the input is (e.g., being typed in when  */
/*    the program is run, in a file, etc.).                      */
/* OUTPUT:  Where the output goes must also be specified.        */
/* For more sophisticated program, include the following:        */
/* ALGORITHM:  Describe abstractly the algorithm used            */
/* DATA STRUCTURES:  Describe the main data structures           */
/* OVERVIEW OF SUBPARTS:  if the program consists of several     */
/* classes, briefly describe the role of each class.             */
/*****************************************************************/
```

2. Class/Module Header

```
/*****************************************************************/
/* PURPOSE:  What does the class do or what ADT it implements    */
/* USING:  the classes and functions used in this class          */
/* USED IN:  What classes/functions are using this class         */
/* ALGORITHM:  Describe the algorithm for each function          */
/* DATA STRUCTURES:  Describe the main data structures           */
/*****************************************************************/
```

3. Function Header

```
/*****************************************************************/
/* PURPOSE:  What exactly does the function do                   */
/* RATIONALE: why this is a separate function                    */
/* INPUT PARAMETERS:  list parameters by names                   */
/*   and what their values mean                                  */
/* OUTPUT PARAMETERS:  list parameters by names                  */
/*   and what do the values of the variable parameters mean      */
/*   after the function is executed                              */
/* RETURN: the results the function returns                      */
/* ALGORITHM:  Describe abstractly the algorithm used            */
/* DATA STRUCTURES:  Describe the main data structures           */
/*****************************************************************/
```

Implementation

The two major implementation issues will be what algorithms are used and how the code is modularized. Generally, the instructions will be rather explicit concerning these issues. However, there will always be details for you to work out. You should use efficient methods, but keep your code straightforward, simple and elegant. Awkward ways of accomplishing simple tasks will result in losing credit.

Modularization should be based on the logical (conceptual) operations being performed and on the logical structure of the data. Functions should receive and return information only through parameters and returned function values, not through references to global variables. All functions related to the same data type should be organized into classes. All classes should be divided into the header (`.h` file) and implementation (`.C` file).

Repeated codes should be abstracted into functions. Usually, a function can logically be broken into several shorter functions if it is too long. A function rarely comprises more than 20 lines of actual code, and never more than 40. This improves the structure of programs.

Testing

All programs must terminate normally. Interrupting the program or allowing it to bomb is not acceptable.

Be sure to test your program completely before you turn it in. What you turn in should demonstrate that you have tested your program adequately. This should always include any test data specified in the assignment. We will generally only give credit for those parts of the program that are tested. We will assume that if you didn't test your program to respond to a condition, then your program doesn't work correctly under that condition.

To make sure that your program is tested thoroughly, it is recommended to design the complete set of test data before the program is designed.

Sample Computer Program Evaluation Form
Assignments #1 Evaluation Comments

Student Name: _____________
Instructor: Shaun-inn Wu

Documentation and Readability

1. Program/calss header comments	Absent	Poor	So-so	MostlyOK	Good
2. Purpose description	Absent	Poor	So-so	MostlyOK	Good
3. Abstract data type description	Absent	Poor	So-so	MostlyOK	Good
4. User's information	Absent	Poor	So-so	MostlyOK	Good
5. Implementation method	Absent	Poor	So-so	MostlyOK	Good
6. Overview of subprograms	Absent	Poor	So-so	MostlyOK	Good
7. Function header comments	Absent	Poor	So-so	MostlyOK	Good
8. Purpose description	Absent	Poor	So-so	MostlyOK	Good
9. Method description	Absent	Poor	So-so	MostlyOK	Good
10. Parameters not described					
11. Variable names		Poor	So-so	MostlyOK	Good
12. Format of comments should be improved					
13. Content of comments should be improved					
14. Comprehensibility of comments should be improved.					

Implementation

15. Modularization	Poor	So-so	MostlyOK	Good
16. Data abstraction	Poor	So-so	MostlyOK	Good
17. Function abstraction	Poor	So-so	MostlyOK	Good
18. Code repeated				
19. Code body too long				
20. Function has mixed functionality				
21. Inappropriate use of global variables				
22. Choice of data structures	Poor	So-so	MostlyOK	Good
23. Overly complicated or awkward or unnecessarily obscure				
24. Inefficient				
25. Coding	Poor	So-so	MostlyOK	Good
26. Overly complicated or awkward or unnecessarily obscure				
27. Inefficient				
28. Erroneous or implementation/system dependent				

Correctness *(varying for different programs)*

29. Some operation doesn't work correctly.
30. *insert(x,p)* or *remove(x)* doesn't work properly when *p* is set to the position of the first element in the *token*.
31. *insert(x,p)* or *remove(x)* doesn't work properly when *p* is set to the position of the last element in the *token*.
32. *insert(x,p)* or *remove(x)* doesn't work properly when *p* is set to the position after the last element in the *token*.
33. *insert(x,p)* doesn't work properly when the position *p* is not set properly as in the *token*.
34. *remove(x), find(x) or findPrevious(x)* doesn't work properly if the element *x* is not in the *token*.
35. Some operations don't work properly when the *token* is empty.
36. Some operations don't work properly when the *token* is full.