Programming Style

I often have students report how many hours they spent on an assignment. Since I have often done the assignment myself, the difference in hours spent is apparent. I asked the students why they thought I could program much faster than they could. I asked if I was that much smarter than they are. They said, “No, that isn’t it ☺. The reason you write so much faster is that you understand how to debug.” That is true, but it is also that I take the time to write elegant code. The less code I have to write, the less that can go wrong. When I use good variable names, it helps me to think clearly about the problem. When I create small service routines rather than one huge monolithic piece, I give myself smaller pieces to test and understand. When I define constants rather than use magic numbers, it makes it easier for me to test the program with small sample sizes before trying it on the complete data set. When I test for abnormal situations, it saves me hours looking for mistakes. Good design actually saves me time.

Programming style varies depending on the environment. These rules will be enforced in the grading for this class.

1. Use meaningful identifiers. Identifiers that clearly explain the value they hold greatly increases readability of the program as well as increasing the clarity with which you program. Often students greatly over-estimate the readability of their programs. Reusing a variable for two distinct values (say sum and percent) is a bad practice because the variable name cannot specify its changing meaning.
   a. Shorter identifiers should be used for a loop index.
   b. In mathematical computations, the use of identifiers similar to standard notation should be encouraged (i-n for integers, x-z for reals, a-c for coefficients or arrays).
   c. Use capitalization or underscores in multiword identifiers (monthlyPay or monthly_pay). In Java, local variables begin with a lower case letter while classes begin with a capital. I really like this convention, but since your book doesn't use it, I can't enforce it. Underscores are a pain to type and should be avoided.
   d. Use nouns to name objects such as variables, constants, and types (e.g., paid, MAX, Colors). C++ tradition says constant names should be upper case; use C++ tradition.
   e. Use verbs to name methods (e.g., getNextRec).
   f. Use forms of the verb “to be” for Boolean functions or variables (e.g., isZero, isFound). Never use flag or mySwitch as you can't tell what you are flagging. Don't use foundFlag as you can't tell what true represents and what false represents.

2. Do not use break to get out of a loop. It hinders the readability of the code. Instead of being able to read the loop condition and understand the code, one has to read every line in the loop.

3. Do not use global variables (except for possible an output file). They hinder readability and maintenance.

4. Declare all constants (except for 0, 1, and 2) as consts. Mysterious numeric constants, termed magic numbers, hinder maintenance as well as readability. This means you cannot use the literal “5” (for example) in your code. The reader asks, “Why 5?” Defining a constant with the value of 5 allows the programmer to explain why the value of 5 is used.

5. Verify that open found the specified files. If not, print an informative message and exit or use assert.
6. Verify that new was able to allocate space. If not, print an informative message and exit or use assert.
7. Using assert, verify that the subscripts fall within legal ranges.
8. Avoid deeply nested structures. Three if often considered a good nesting depth. If you need something greater, put some of the code into a separate method.
9. Indent to show nesting of structures or code. (Don't indent for other reasons.)
10. Use comments.
   a. Use comments to explain every declared variable (except indices).
   b. Use comments to explain every parameter. I know this is not fun, but it is necessary for full credit. It is not busy work, but is necessary for proper documentation.
   c. Use comments in conjunction with the beginning of every method or function, including main. Comments at the beginning of the program should list program purpose, input format, outputs produced, author, and date written.
   d. Use comments before blocks of code rather than with every line or two. Bad comments can seriously detract from good code.
   e. Comments are usually easier to maintain if they exist on a separate line rather than on the same line as code.
11. Break the program into modules that represent logical components. If a module extends beyond a page or two, you should seriously consider further subdivision.
12. If you really wish the language had some built-in function, write that function. Then you will have it!
13. Output should have appropriate spacing and be clearly labeled with a title and column headings. You are responsible for deciding what form of the output is most useful, even if the assignment does not clearly specify. The user should not have to read the programming assignment to make sense of the output. The output should be like a final report to your boss.
14. When you expect the user to provide input from the keyboard, you must prompt the user with a meaningful request such as:
    
    ```
    cout << "Enter the exam score " << endl;
    cin >> examScore;
    ```

    When you use a terminal controlled loop, the prompt may look something like:

    ```
    cout << "Enter hours worked, a negative number to quit " << endl;
    ```

15. Avoid extra, unneeded #include <header files>. Assume that they make your intermediate files more complicated than necessary.

    **Programming Hints**

1. One of the most serious problems I see with code is having large sections of code which are almost duplicates of each other. There should not be duplication as maintenance and understanding is greatly hampered. It is MUCH better to pull out common sections into routines that work for several uses. Sometimes parameters can make the same piece of code serve two purposes. Often one version can call the other version of the code. Students often feel like doing a cut and paste of the code is acceptable. It isn’t. Take the time to design elegant code. Near duplicate portions of code is a sign of sloppy programming.
2. Another serious problem is abuse of object oriented principles. Objects should protect their data and provide all of the needed functionality. For example, creating a “toString” method for each object which converts all of the data to a string form is a routine which can be used even by non-member operator << routines. A “doit” routine which manipulates the object should be part of the object code, even if a tester routine could do it. All methods which manipulate the object should be part of the object. That way, the organization is easier to understand and reuse of the object is facilitated.

3. Consider using for loops for a wide variety of circumstances. Because most looping needs have an initialization, a control condition, and an update condition, some programmers use for loops for almost every looping need.

   For example, a linked list could be traversed using a for loop such as

   ```
   for (Node * n = head; n != NULL; n = n->next) cout << n->value;
   ```

4. Use the appropriate type for a variable. When a value is appropriately an int, make it an int, not a double. Doubles typically take either 2 or 4 times as much space in memory than an int, and arithmetic using doubles takes 1 or 2 or even 3 orders of magnitude more time than integer arithmetic. So for both space and time, ints should be ints.

5. For simple parameters, use pass-by-value whenever possible. Pass by reference puts your original data at risk, and is often used when you must get more than one value out of a function. However, for class parameters, a reference is preferred. Otherwise, you are calling the copy constructor everytime you pass a parameter, and you are calling the destructor on exit. It you do a shallow copy and a deep destruct, this is deadly as well as being expensive.

6. Echo all input values to the output file. This is very helpful in debugging.

   ```
   cin >> hours >> rate;
   fout << "hours worked is " << hours << " and rate is " << rate << endl;
   ```

7. Beware of integer division when that is not what you want to happen. If both numerator and denominator are integers, the result is truncated to an integer before assignment into the receiving variable.