GENERAL INSTRUCTIONS FOR SUBMITTING PROGRAMMING ASSIGNMENTS & STATEMENT OF PROGRAM GRADING POLICIES

FORM OF THE ASSIGNMENT:

Typically for each programming assignment, I will require you to turn in separate "levels" of your program: for example, a top-level may be due at the end of the first week, a second-level due at the end of the second week, and a final level due at the end of the third week.

Each time you turn in a level of a program, it must be complete down to the assigned depth. It must include all the appropriate documentation and data structures for the assigned depth. Portions of the program at levels not yet completed must be represented with stubs.

TO BE SUBMITTED ELECTRONICALLY:

For each version of the program that you turn in, you are required to turn in two items:

==> a compilable and correctly-working copy of the program source code.
==> a record of the set of tests that you performed on your program.

The testing record can be conveniently made by using the "script" command, and filtering the script file through "col -b". (For more information on "script", type "man script" while logged into a Sun). While recording each test run using "script", first cat the input file(s) (if any) for the run, then run the program, and then cat the output file(s) (if any).

Turn in each of the two items listed above as a separate e-mail message. Furnish a suitable subject line with your messages. Address mail to lamie@altair.csustan.edu.

I will, at my discretion, compile and run (on my own test input) the programs you submit electronically. This is a part of my grading procedure.

POINT DISTRIBUTION:

A program must compile correctly, run according to specifications, and give correct results.

For full credit, each version of the program must meet the criteria below as well. The criteria with their relative weights are:

<table>
<thead>
<tr>
<th>Weight</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>Minimally acceptable program as above.</td>
</tr>
<tr>
<td>20</td>
<td>Good design, including good algorithms.</td>
</tr>
<tr>
<td>20</td>
<td>Adequate testing, especially the testing of data boundaries and special cases.</td>
</tr>
<tr>
<td>20</td>
<td>Good form, including documentation, and readability.</td>
</tr>
</tbody>
</table>
DOCUMENTATION:

The proper way to create a function is to first write a header comment, then perhaps a stub, and finally the code.

Each function (this includes "main") must have a header comment containing a short description of what the function is supposed to accomplish. (In the case of a function stub, the header comment tells what the function will do when it is finished.)

In order to competently tell what the function does, these "headers" must say what the inputs and outputs are, what preconditions or assumptions, if any, have to be true in order for the function to be assured of executing correctly, and what postconditions will be true when execution of the function ends.

The header of a function must contain the names of the important variables, and data structures it contains. It must explain what the function accomplishes with these objects.

You must include enough detail in your headers to get the main idea across, but no more detail than is necessary.

The beginning of the header for the function "main" must contain your name, your username (login name), the course name, and the date the writing of the program was completed. Each of these things must be clearly labeled. The header of any function besides "main" must give the names of all its callers (the subprograms that call it) and the names of all its callees (the subprograms it calls),

All names of constants, types, variables, functions, etc. must be "self-documenting" and "self-describing." For example, call a variable "total" if it is used to store a total. Call a function "PrintList" if its job is to print a list. If you are able to name a particular constant, type, or variable so well that it becomes abundantly clear from its name alone what it is used for, then it doesn't need any further documentation. Otherwise, you must explain its use with "in-line" documentation next to its declaration. I will take points off if I don't think you have documented named objects adequately.

The decision whether to make other in-line comments is left up to you. However, in-line comments are only needed occasionally in programs that have good headers, good naming conventions, and good organization. "Good form", which is one of the criteria upon which your program is graded, demands that you avoid over-commenting, as well as under-commenting. For example, the program line:

```
Sum = 0; // We initialize the value of the sum to zero
```

illustrates blatant over-commenting. The comment inserted to the right of the statement does not add anything to our understanding of what is going on. The statement

```
Sum = 0;
```
tells us just as much, partly because the variable "sum" is named so as to make it "abundantly clear" what is being done.

GOOD FORM:

You must use a readable, logical, and coherent set of style and formatting rules. For an example, see pp 36-42 in Chapter One of "Data Abstraction and Problem Solving with C++: Walls and Mirrors" by Carrano and Prichard (3rd edition).

STRUCTURED PROGRAMMING AND MODULARITY:

I expect you to stick to the "structured approach" in programming. This means using top-down design, modular code, information hiding, and "orderly flow of control".