PROGRAMMING/CODING STYLE

A program may produce correct outputs for each input,
but that has nothing to do with the program being well-designed/coded.

1. Don't change function-names (parameters and their ordering), local/global variables, and datatypes given in a program specification. Also,

   - Keep #includes, #defines, typedefs, and variables in that order at the beginning of program-file; there should be no unused #includes, etc.
   - Do not introduce unnecessary variables/functions/typedefs.
   - Do not use unnecessary assignments (and initializations), if-statements, and unnecessary iterations of loops.
   - If an operation need not be in a loop-body, then move it out of the loop. Similarly, for then/else-parts of if-statements.
   - Avoid unnecessary memory allocation (calloc/malloc) and deallocation (free) operations.

2. Choose variable/function names that are short, informative, and suggestive of their meanings/roles; use easily understood, consistent/standard abbreviations. (You should be able to guess the meaning of a name if you saw it in someone else's code or your own code a year later.) Don't make up for bad names by adding comments. Don't use lower-case ('`l`') for a variable-name.

   - Begin a function name with a capital letter; similarly, for user-defined structures/typedefs. `PrintMatrix` is shorter and just as readable as `Print_matrix`. Typedef all structures; do not typedef pointers.

```
typedef struct ListNode {
        int nodeNum;  //0 <= nodeNum < numNodes
        struct Node *next;
} ListNode;
```

   - Use //-form for comments instead of /*-*/ form (which creates problems with nested-comments); keep comments short and informative (easier to update as the code changes). Avoid comments that do not correspond to the code.

3 Avoid unnecessary parameters in functions/macros; order them meaningfully. Keep the parameters reset by a function to a minimum and put them at the end of parameter-list; use return-values whenever possible. Don't use spaces before/after '(' and ')' in parameter-lists; see below.

```
                PrintMatrix(double **matrix, int numRows, int numCols)
is better than  PrintMatrix(double **matrix, int m, int n)
is better than  PrintMatrix(double **matrix, int i, int j)
is better than  PrintMatrix(double **matrix, int numCols, int numRows).
```

4. Use 'else' with an if-then whenever possible, save the cases such as "error in opening file ···". (For this course, testing for failures in opening files and memory allocations is not a significant issue.) The form "`1 == found`" in an if-condition (instead of "`found == 1`") helps catch a common typing-error ("=" instead of "==") at compile time. Keep the if-conditions in the most direct form; avoid negation, if possible. For example, if the variable `found` takes only the values 1 and 2, then "`if (1 == found) ...`" is better than each of the following:

```
if (2 != found) ...
if (!(2 == found) ...
if (2 > found) ...
```

   Also, "`x = y = 0.1;`" is better than "`x = 0.1; y = 0.1;`" and "`x = 0.1; y = x;`"(in the same line or two consecutive lines). Likewise, if `firstCall` takes only the values 0 and 1, then "`firstCall = 1`" is better than "`firstCall++`" after its initialization to 0 (the former is more direct, more efficient, and less dependent on other parts of the code).

5. Use "break" for an early termination of a loop. Use a do-while or while-do loop only if the loop control-variable is modified in an irregular way or it is shorter than a for-loop or in special cases (e.g, termination-test boundary value is determined only in loop-body).
6. Keep related variables in a declaration together and meaningfully ordered; use boolean, char, int, double, FILE, and user-defined types in that order otherwise.

   - Use global variables only if justified (e.g., only one function writes to them; avoids passing them often as parameters to functions).
   - Use static variables in a function instead of global variables whenever possible.
   - Keep variable usage consistent (e.g., `i` for rows and `j` for columns in matrix operations) as much as possible.

7. Keep proper and consistent indentation of code throughout the program code; avoid 'tab'.
8. Avoid unnecessary {}. Can use local variable declarations in a {}-block; keep them to a minimum.
9. Sometimes it helps to avoid a line containing only '{' or '}' to allow more code on the same page; give a space before and after '{' and '}' in that case. For a short if-statement or for-statement, you may use an one-liner as in
   "`if (i > j) i = j;`" or "`for (i=total=0; i<numItems; i++) total += items[i];`"
10. Use a space on two sides of the assignment '=' (or '+='); use a space after ',' and ';' (but not before them) and after "if", "for", "while", etc.
11. Use #define for constants; use dynamic memory allocation (calloc/malloc) whenever possible.
12. Use auxiliary print-functions for testing intermediate results during program-development. (Leave some of these print-calls as comments to indicate the things that you had tested in detail and to get back to them in future when the program is modified or new errors are detected.) Do not print anything without indicating what it is (e.g., the variable name) that you are printing. The main() should mostly call other functions; the other functions should be in a form that can be used directly elsewhere, if needed, without any change.
13. If you use static variables in a function, then make sure the function works properly if called more than once.
14. Keep the program logic clean and simple as much as possible (depending on the algorithm).

Show your creativity and programming-knowledge without violating the above rules.
Don't make your program punish the computer.