Project 5, Program Design

1. (60 points) Joe wants a way to encode his notes he passes in class so his teacher cannot read them. Write a program that encodes a sentence by switching every alphabetical letter (lower case or upper case) with alphabetical position i with the letter with alphabetical position 25 – i. For example, letter *a* is at position 0 and after encoding it becomes letter *z* (position 25). Letter *m* is at position 12, it becomes letter *n* (25 – 12 = 13) after encoding. For example,

```
Input: at the cafeteria
Output: zg gsv xzuvgvirz
```

Your program should include the following function:

```
void convert(char *s1, char *s2);
```

The function expects `s1` to point to a string containing the input as a string and stores the output to the string pointed by `s2`.

1) Name your program `notes.c`.
2) Assume input is no longer than 1000 characters.
3) The `convert` function should use pointer arithmetic (instead of array subscripting). In other words, eliminate the loop index variables and all use of the [] operator in the function.
4) To read a line of text, use the `read_line` function (the pointer version) in the lecture notes.

2. (40 points) In this program, you will take command-line arguments of 10 numbers, which are assumed to be integers, and find the median or the average of them. The average of the array should be a floating point number (double). Before the numbers on the command line, there will be an argument that indicates whether to find the median (`-m`) or average (`-a`), if the user entered an invalid option or incorrect number of arguments, the program should display an error message. To find the median of an array, use the *selection_sort* function provided to sort the array, the median is the number that is halfway into the array: *a[n/2]*, where *n* is the number of the elements in array *a*. Example input and output:

```
./a.out -a 2 6 8 4 9 10 7 3 11 5
output: 6.5

./a.out -m 2 6 8 4 9 10 7 3 11 5
output: 7

./a.out
output: usage: ./a.out -option (a or m) followed by 10
numbers
```

```
./a.out -d 2 6 8 4 9 10 7 3 11 5
output: Invalid option
```

1) Name the program *command_line.c*
2) Use `strcmp` function to process the first command line argument.
3) Use `atoi` function in `<stdlib.h>` to convert a string to integer form.

Before you submit:

1. Compile with –Wall. Be sure it compiles on ***student cluster*** with no errors and no warnings.

gcc –Wall notes.c
gcc –Wall command_line.c

2. Be sure your Unix source file is read & write protected. Change Unix file permission on Unix:

chmod 600 notes.c
chmod 600 command_line.c

3. Test your program with the shell scripts on Unix:

chmod +x try_notes
./try_notes

chmod +x try_command_line
./try_command_line

Total points: 100 (problem 1: 60 points, problem 2: 40 points)

1. A program that does not compile will result in a zero.
2. Runtime error and compilation warning 5%
3. Commenting and style 15%
4. Functionality 80%

Programming Style Guidelines

The major purpose of programming style guidelines is to make programs easy to read and understand. Good programming style helps make it possible for a person knowledgeable in the application area to quickly read a program and understand how it works.

1. Your program should begin with a comment that briefly summarizes what it does. This comment should also include your **name**.
2. In most cases, a function should have a brief comment above its definition describing what it does. Other than that, comments should be written only *needed* in order for a reader to understand what is happening.
3. Information to include in the comment for a function: name of the function, purpose of the function, meaning of each parameter, description of return value (if any), description of side effects (if any, such as modifying external variables)
4. Variable names and function names should be sufficiently descriptive that a knowledgeable reader can easily understand what the variable means and what the function does. If this is not possible, comments should be added to make the meaning clear.
5. Use consistent indentation to emphasize block structure.
6. Full line comments inside function bodies should conform to the indentation of the code where they appear.
7. Macro definitions (#define) should be used for defining symbolic names for numeric constants. For example: `#define PI 3.141592`
8. Use names of moderate length for variables. Most names should be between 2 and 12 letters long.
9. Use underscores to make compound names easier to read: `tot_vol` or `total_volumn` is clearer than totalvolumn.