CS 31 Worksheet 6

This worksheet is entirely **optional**, and meant for extra practice. Some problems will be more challenging than others and are designed to have you apply your knowledge beyond the examples presented in lecture, discussion or projects. All exams will be done on paper, so it is in your best interest to practice these problems by hand and not rely on a compiler.

Solutions are written in red. The solutions for **programming** problems are not absolute, it is okay if your code looks different; this is just one way to solve the specific problem.

Concepts: 2D Arrays, C strings

Reading Problems

1)

a) What does the following program print out?

```
#include <iostream>

using namespace std;

int main () {

      char sentence[] = "Peter loves his sandwich.";


      for (int i = 0; i < strlen(sentence); i++) { // change in part b

            sentence[strlen(sentence) - 1] = '\0';

      }

      cout << "Drum roll please: " << sentence << endl;

}
```

Drum roll please: Peter loves

One each iteration of the loop, a zero byte is stored one position closer to the beginning of the string, so strlen(sentence) will return a value one less than it did before. As i grows and strlen(sentence) shrinks, they'll meet when i reaches the middle of the array, causing the loop to end.

b) Repeat part a), but replace the for loop inside main() to the following code:

```
int n = strlen(sentence);

for (int i = 0; i < n; i++) {

      sentence[strlen(sentence) - 1] = '\0';
}
```

Avoids readjusting the changing length of the cstring in each iteration by storing the length in a variable before the loop begins. Will replace every character in the cstring with a null byte.

Programming Problems

1) Write a function with the following header:

```
void reflect(int matrix[][N], int n);
```

where :

> **`matrix`** is a 2-dimensional array of integers of size N x N. In this header, N is to be replaced by a number chosen by the programmer (you).
>
> **`n`** is the value N (passed in so that reflect knows how many rows **`matrix`** has)
>
> **`reflect`** should reflect **`matrix`** across the negative-sloping diagonal, so that the rows become the columns and vice versa. (i.e. matrix[i][j] becomes matrix[j][i] after **`reflect`**)

Example:

```
/* The second [] in a 2D array passed as a parameter requires a
number as the size, which restricts the implementation of reflect to
be able to work on only one matrix size. The following example works
only on 2D arrays of size 3x3. For reflect's declaration, the N in
the function header has been replaced by 3. */

void reflect(int matrix[][3], int n) {
      // Implementation goes here...
}

int main() {

      int foobar[3][3] = {{1, 2, 3},{4, 5, 6}, {7, 8, 9}};

      reflect(foobar, 3);

/* foobar is now expected to be:

{{1, 4, 7}, {2, 5, 8}, {3, 6, 9}} */
```

```
// to check results
   for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            cout << foobar[i][j] << " ";
        }
        cout << endl;
    }



}


// This solution is for 3x3 arrays.
void reflect(int matrix[][3], int n) {
      for (int i = 0; i < n; i++) {
            for (int j = i; j < n; j++) {
                  int temp = matrix[i][j];
                  matrix[i][j] = matrix[j][i];
                  matrix[j][i] = temp;
            }
      }
}
```

2) Write a function *charInsert* that inserts a character into a valid C-string at a given position. The function has the following header:

```
bool charInsert(char str[], int n, int idx, char c);
```

The parameter **`n`** denotes the size of the character array **`str`**, which is not necessarily equivalent to the string's length. **`idx`** refers to the index at which the insertion will be done, so if **`idx`** is 0 then the **`char c`** will be the first character in the new string. The insertion cannot be performed if **`idx`** is negative or greater than the string's length.

Additionally, the insertion cannot be performed if the result would exceed the size of the array **n**.

If the insertion is successful, the function returns true. If the insertion cannot be done, the function returns false and leaves **str** unmodified.

Examples:

```
char success[10] = "aaaaa";

bool res = charInsert(success, 10, 1, 'b'); // res should equal true

cout << success << endl; // abaaaa

                  abcd

char success[10] = "aaaaa";

bool res = charInsert(success, 10, 5, 'b'); // res should equal true

cout << success << endl; // aaaaab


char failure[6] = "aaaaa";

bool res = charInsert(failure, 6, 1, 'b'); // res should equal false

cout << failure << endl; // aaaaa    unchanged
```

```
bool charInsert(char str[], int n, int idx, char c)

{

      if ((idx < 0) || (idx > strlen(str)) || (strlen(str)+1 >= n))

            return false;

      str[strlen(str) + 1] = '\0';

      for (int i = strlen(str); i >= idx; i--)

            str[i+1] = str[i];

      str[idx] = c;

      return true;

}
```

3) Write a function wordRotateLeft that takes in a valid C-string and rotates each word left one character. A word is defined as a sequence of non-spaces separated by

spaces. Each rotated word wraps around, meaning that “CS31” would become “S31C”. The function has the following header:

```
void wordRotateLeft(char str[]);
```

Example:

```
char test[] = "I love CS31"; wordRotateLeft(test);

cout << test << endl; // "I ovel S31C"


char test2[] = "I.love.CS31"; wordRotateLeft(test2);

cout << test2 << endl; // ".love.CS31I"


void wordRotateLeft(char str[]){

	int len = strlen(str);

	int beginWord = 0;

	for(int i = 0; i < len; i++){

		if(str[i] == ' '){

			char beginChar = str[beginWord];

			for(int j = beginWord; j < i - 1; j++){ // iterate through
		the word

				str[j] = str[j + 1];	// replace leftmost character
			with the character to the right

			}

			str[i - 1] = beginChar;		// replace the last char of
		the word with the first char

			beginWord = i + 1; // set beginWord to the index of the first
		char of the next word

		}

	}

	if(beginWord < len){ // if this is the last word in your sentence

		char beginChar = str[beginWord];

			for(int j = beginWord; j < len - 1; j++){
```

```
                        str[j] = str[j + 1];
                    }
                    str[len - 1] = beginChar;
                }
            }
        }
```

4) Write a function with the following header:

```
bool rangeSearch(int sorted_nums[], int n, int target,
                 int& start, int& end);
```

`sorted_nums` is an array of integers sorted in decreasing order
`n` is the number of elements in `sorted_nums`
`target` is a number to search for within `sorted_nums`

`rangeSearch` should return *true* if `target` is found in `sorted_nums` and *false* otherwise.

If `rangeSearch` returns *true*, `start` should be set to the first index where `target` appears and `end` should be set to the last index where `target` appears, so that the integers of indices from `start` to `end` should only contain `target`.

If `rangeSearch` returns *false*, `start` and `end` should not be altered.

Example:

```
int foo[7] = {5, 4, 3, 3, 1, -2, -3};
int s = 21;
int e = 14;
rangeSearch(foo, 7, 0, s, e); // returns false, s remains 21, e == 14
rangeSearch(foo, 7, 3, s, e); // returns true, now s == 2 and e == 3
```

```
bool rangeSearch(int sorted_nums[], int n, int target, int& start, int& end){
    int s = 0;
    int e = n-1;
    while(s<n && sorted_nums[s]>target)
        s++;
    while(e>=0 && sorted_nums[e]<target)
        e--;
    if(sorted_nums[s] == target && sorted_nums[e] == target){
```

```
                start = s;
                end = e;
                return true;
        }
        return false;
}
```