CS211 Homework – 6

Total Marks: 100

Due Date: 11/13/2019

Question 1 of 1. **Reverse Word Order**: Write a program that reverses the order of the words in a given sentence.

This program requires reversing the order of the words wherein the first and last words are swapped, followed by swapping the second word with the second to last word, followed by swapping the third word and the third to last words, and so on.

Your program will ask the user for an input string and print out the resultant string where the order of the words is reversed. Please see the hints section for more details on an example algorithm.

Assume a maximum C-string size of 1000 characters.

Make sure your code works for any input number, not just the test cases. Your code will be tested on other test cases not listed here. *Do Not Use Predefined Functions from the cstring Library.*

Please properly comment your code before submission.

For this part of the assignment, name your source file as ReverseWordOrder_WSUID.cpp. For example, if your user ID is A999B999 name your file as ReverseWordOrder_A999B999.cpp.

Sample Test Cases:

Test Case 1: **Input:** London bridge has been abducted **Output:** abducted been has bridge London	**Test Case 2:** **Input:** Hello World **Output:** World Hello
Test Case 3: **Input:** Hello World, how are you? **Output:** you? Are how World, Hello	**Test Case 4:** **Input:** I like toast **Output:** toast like I

Hint:

There are different ways of solving this problem. The following shows a relatively simple way of achieving this. You might choose to do it differently, which is fine.

One trick to solving this problem is to first reverse individual words, and then reverse the entire array.

For example:
Input string: **London bridge is falling down**
If we reverse the individual words in this string, we get
Reversed words: **nodnoL egdirb si gnillaf nwod**

Note that this is different from reversing the full array, which would have given us
Full reverse: nwod gnillaf si egdirb nodnoL
We do not want this.

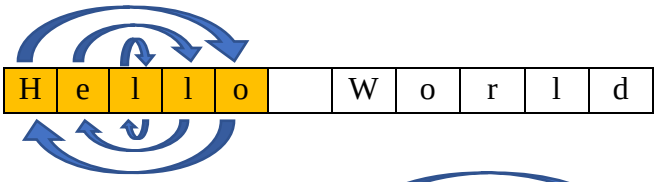
Instead we want to reverse the individual words while keeping their correct order. Which gives us: **nodnoL egdirb si gnillaf nwod**
Now if we reverse this entire array, we get the string with the word order reversed:
Reverse array: **down falling is bridge London**

For example:
Input String: char src[] = “Hello World”

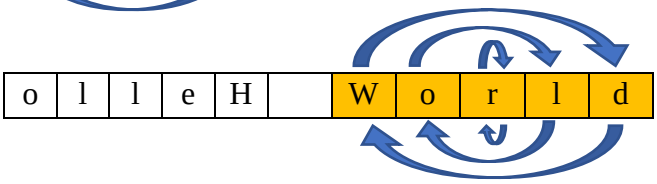
0	1	2	3	4	5	6	7	8	9	10
H	**e**	**l**	**l**	**o**		**W**	**o**	**r**	**l**	**d**

We need to reverse the individual words, which can be done by reversing each word, iterating over each word.

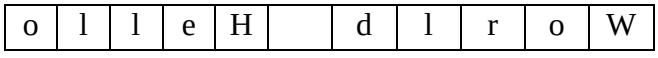
Reverse first word:



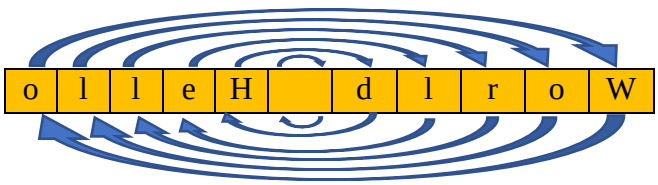
Reverse second word:



Resultant string:



Following this, we simply reverse the entire array to get the resultant string with the words in reverse order.



Result:

W	o	r	l	d		H	e	l	l	o

This algorithm requires reversing parts of a string or individual words, while keeping the rest of the string untouched. In order to achieve this, let us break this problem down into corresponding sub-problems. We need a function that can reverse individual words, and we also need a function that can locate these individual words in a given string.

Function 1 *of* 2: *reverseSubString(…)*

Function declaration:
void reverseSubString(char src[], int start_loc, int end_loc)

This function takes as arguments the input string, and the start and end location of the substring to be reversed. This function does not have a separate destination string and in essence modifies the source string 'src' by reversing the required part of the input string, indicated by the indices *start_loc* and *end_loc*.

However, before we understand how to reverse a substring, we need a method of reversing an entire string first. The following method shows the process of reversing a string by iteratively swapping elements in an array.

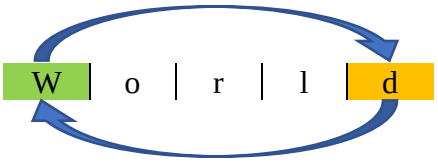
Swapping elements as a means of reversing an array.

One method for reversing an array includes swapping the respective elements of the array. This involves swapping the first element with the last element of the array, followed by swapping the second element with the second to last element of the array, the third element with the third to the last element, and so on. In order to reverse a string, we need to swap all the elements up to the **midpoint** of the array.

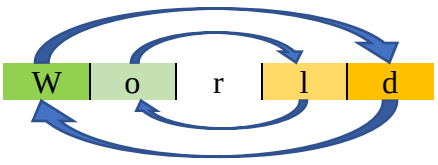
For example: String with odd number of elements
String: "World"

0	1	2	3	4
W	o	r	l	d

Swapping the first element with the last element:
String becomes: "dorlW"



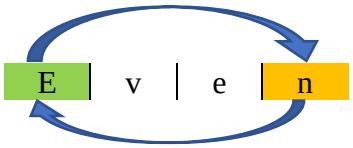
Following this, swapping the second element with the second to last element:
String becomes: "dlroW"



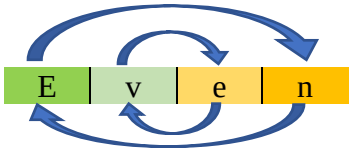
For example: String with even number of elements
String: “Even”

0	1	2	3
E	v	e	n

Swapping the first element with the last element:
String becomes: “nveE”



Following this, swapping the second element with the second to last element:
String becomes: “nevE



For every element we swap the (i^{th}) element with the ($N - i^{th} - 1$) element, where N is the length of the string.

- For strings with an **even** number of elements *(say 4 elements)*, we need to swap:
  Element i = 0, with the element at N – i – 1, (4 – 0 – 1):: 0 -> 3
  Element i = 1, with the element at N – i – 1, (4 – 1 – 1):: 1 -> 2
- For strings with an **odd** number of elements *(say 7 elements)*, we need to swap:
  Element i = 0, with the element at N – i – 1, (7 – 0 – 1):: 0 -> 6
  Element i = 1, with the element at N – i – 1, (7 – 1 – 1):: 1 -> 5
  Element i = 2, with the element at N – i – 1, (7 – 2 – 1):: 2 -> 4
  Element i = 3, with the element at N – i – 1, (7 – 3 – 1):: 3 -> 3

Using the method shown above for swapping the individual elements, allows reversing an entire string or array relatively easily. However, our function needs to be able to reverse a specific part of a given string, while keeping the remaining segment intact.

This requires a slight modification to the above approach. Since we may not start from index 0 anymore, we need to offset our original logic for swapping the elements and reversing the string. Now, for every element we swap the (start_loc + i^{th}) element with the (end_loc – i^{th}) element. Where start_loc and end_loc indicate the part of the string to be reversed. See function declaration of ***reverseSubString***.

- Reversing substrings with an **even** number of elements *(say 6 elements)*, with start_loc = 7 and end_loc = 12:
  Element i = 0, start_loc + i = 7, swap with element, end_loc – i, (12 – 0):: 7 -> 12
  Element i = 1, start_loc + i = 8, swap with element, end_loc – i, (12 – 1):: 8 -> 11
  Element i = 2, start_loc + i = 9, swap with element, end_loc – i, (12 – 2):: 9 -> 10

Note Example 1 in the following set of examples.

For example:
Input string: char src[] = “London bridge is falling”;

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
L	**o**	**n**	**d**	**o**	**n**		**b**	**r**	**i**	**d**	**g**	**e**		**i**	**s**		**f**	**a**	**l**	**l**	**i**	**n**	**g**

Example 1: ***reverseSubString(src, 7, 12)***
start _loc: 7 //Reverses the characters from index 7 up to index 12
end_loc: 12 //Essentially reversing the word “bridge”
Modifies source string to: **“London *egdirb* is falling”**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
L	**o**	**n**	**d**	**o**	**n**		**e**	**g**	**d**	**i**	**r**	**b**		**i**	**s**		**f**	**a**	**l**	**l**	**i**	**n**	**g**

Example 2: ***reverseSubString(src, 0, 5)***
start _loc: 0 //Reverses the characters from index 0 up to index 5
end_loc: 5 //Essentially reversing the word “London”
Modifies source string to: **“*nodnoL* bridge is falling”**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
n	**o**	**d**	**n**	**o**	**L**		**b**	**r**	**i**	**d**	**g**	**e**		**i**	**s**		**f**	**a**	**l**	**l**	**i**	**n**	**g**

Example 3: ***reverseSubString(src, 0, 23)***
start _loc: 0 //Reverses the characters from index 0 up to index 23
end_loc: 23 //Essentially reversing the entire string
Modifies source string to: **“*gnillaf si egdirb nodnoL*”**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
g	**n**	**i**	**l**	**l**	**a**	**f**		**s**	**i**		**e**	**g**	**d**	**i**	**r**	**b**		**n**	**o**	**d**	**n**	**o**	**L**

Function 2 *of* 2: *findNextWord(…)*

Now that we have a function which can reverse arbitrary substrings, we need to use the *reverseSubString* function to reverse the individual words in our sentence. In order to do this, we need to specify the start and end locations of individual words. Thus, we need another function which should give us the start and end location of the next word, when searching from a given location. This function can then be used to locate all the words in a given sentence.

Function declaration:
void findNextWord(char src[], int start_search_loc, int& wordStart, int& wordEnd)

This function takes as arguments the input string, and the location from which to start searching for the next word. Upon completion, this function updates the integer parameters, which are passed as references, *wordStart and wordEnd* with the start and end index of the word.

CS211 Homework – 6

Total Marks: 100

Due Date: 11/13/2019

Words in a sentence are separated by whitespaces. Thus, the location of a word can be identified, by finding the location of the next whitespace, followed by the location of the next to next whitespace.

For example:
Input string: char src[] = “London bridge is falling”;

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
L	**o**	**n**	**d**	**o**	**n**		**b**	**r**	**i**	**d**	**g**	**e**		**i**	**s**		**f**	**a**	**l**	**l**	**i**	**n**	**g**

Given any index, this function should search for the next complete word from that index onwards. If the current index *start_search_loc* is pointing at the middle of a word, we need to find the next whitespace as a marker for the start of the next word, and then find the next to next whitespace as the marker for the end of the word.

Example 1: ***findNextWord(src, 3, wordStart, wordEnd)***

start_search_loc: 3 //find the next word location. “bridge”
Returns:
wordStart: 7 //location of first character of word. ‘b’
wordEnd: 12 // location of last character of word. ‘e’

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
L	**o**	**n**	**d**	**o**	**n**		**b**	**r**	**i**	**d**	**g**	**e**		**i**	**s**		**f**	**a**	**l**	**l**	**i**	**n**	**g**

However, if the current index *start_search_loc* is pointing at a whitespace, we need to consider the next character as the start of the next word, and find the next to next whitespace as the marker for the end of the word.

Example 2: ***findNextWord(src, 13, wordStart, wordEnd)***

start_search_loc: 13 //find the next word location. “is”
Returns:
wordStart: 14 //location of first character of word. ‘i’
wordEnd: 15 // location of last character of word. ‘s’

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
L	**o**	**n**	**d**	**o**	**n**		**b**	**r**	**i**	**d**	**g**	**e**		**i**	**s**		**f**	**a**	**l**	**l**	**i**	**n**	**g**

Apart from this we also **need to cover some edge cases**. For example, when *start_search_loc* is the index 0, we need to return the indices of the first word.

Example 3: ***findNextWord(src, 0, wordStart, wordEnd)***

start_search_loc: 0 //find the first word location. “London”
Returns:
wordStart: 0 //location of first character of word. ‘L’
wordEnd: 5 // location of last character of word. ‘n’

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
L	**o**	**n**	**d**	**o**	**n**		**b**	**r**	**i**	**d**	**g**	**e**		**i**	**s**		**f**	**a**	**l**	**l**	**i**	**n**	**g**

The last word in a sentence does not end with a whitespace. Therefore, in addition to the above case, we also need to cover the case where the sentence terminates with a 'null' character.

Example 4: ***findNextWord(src, 16, wordStart, wordEnd)***

start_search_loc: 16 //find the next word location. "falling"

Returns:

wordStart: 17 //location of first character of word. 'f'

wordEnd: 23 // location of last character of word. 'g'

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
L	**o**	**n**	**d**	**o**	**n**		**b**	**r**	**i**	**d**	**g**	**e**		**i**	**s**		**f**	**a**	**l**	**l**	**i**	**n**	**g**

Putting it all together:

Finally, we can make use of these two functions to solve the given problem of reversing the word order of a given string.

Firstly, we reverse every word in the given string, by using the *findNextWord* and *reverseSubString* functions in a loop.

For example:

Input String: char src[] = "coding is life";

0	1	2	3	4	5	6	7	8	9	10	11	12	13
c	**o**	**d**	**i**	**n**	**g**		**i**	**s**		**l**	**i**	**f**	**e**

We initialize,

int start_search_loc = 0;

int wordStart = 0, wordEnd = 0;

Iteration 1:

start_search_loc = 0;

findNextWord(src, start_search_loc, wordStart, wordEnd);

wordStart becomes: 0

wordEnd becomes: 5

0	1	2	3	4	5	6	7	8	9	10	11	12	13
c	**o**	**d**	**i**	**n**	**g**		**i**	**s**		**l**	**i**	**f**	**e**

Given wordStart and wordEnd, we can then call reverseSubString to reverse this specific word.

reverseSubString(src, wordStart, wordEnd); // *reverseSubString*(src, 0, 5)

0	1	2	3	4	5	6	7	8	9	10	11	12	13
g	**n**	**i**	**d**	**o**	**c**		**i**	**s**		**l**	**i**	**f**	**e**

Then we need to find the next word. We can do this by searching the string again, starting from the index wordEnd.

Iteration 2:

start_search_loc = wordEnd; //start_search_loc = 5

findNextWord(src, start_search_loc, wordStart, wordEnd);

wordStart becomes: 7

wordEnd becomes: 8

0	1	2	3	4	5	6	7	8	9	10	11	12	13
g	**n**	**i**	**d**	**o**	**c**		**i**	**s**		**l**	**i**	**f**	**e**

reverseSubString(src, wordStart, wordEnd); // *reverseSubString*(src, 7, 8)

0	1	2	3	4	5	6	7	8	9	10	11	12	13
g	**n**	**i**	**d**	**o**	**c**		**s**	**i**		**l**	**i**	**f**	**e**

Iteration 3:

start_search_loc = wordEnd; //start_search_loc = 8

findNextWord(src, start_search_loc, wordStart, wordEnd);

wordStart becomes: 10

wordEnd becomes: 13

0	1	2	3	4	5	6	7	8	9	10	11	12	13
g	**n**	**i**	**d**	**o**	**c**		**i**	**s**		**l**	**i**	**f**	**e**

reverseSubString(src, wordStart, wordEnd); // *reverseSubString*(src, 10, 13)

0	1	2	3	4	5	6	7	8	9	10	11	12	13
g	**n**	**i**	**d**	**o**	**c**		**s**	**i**		**e**	**f**	**i**	**l**

We stop the loop when wordEnd + 1 is a ‘null’ character.

Finally, our source string has been modified where all the individual words have been reversed. Now we make one final call to *reverseSubString* in order to reverse the entire string to get the solution.

We reuse the value of wordEnd which after the above loop, which now points to the last character of our string.

Current String src:

0	1	2	3	4	5	6	7	8	9	10	11	12	13
g	**n**	**i**	**d**	**o**	**c**		**s**	**i**		**e**	**f**	**i**	**l**

reverseSubString(src, 0, wordEnd); // *reverseSubString*(src, 0, 13);

0	1	2	3	4	5	6	7	8	9	10	11	12	13
l	**i**	**f**	**e**		**i**	**s**		**c**	**o**	**d**	**i**	**n**	**g**

Result:

0	1	2	3	4	5	6	7	8	9	10	11	12	13
l	**i**	**f**	**e**		**i**	**s**		**c**	**o**	**d**	**i**	**n**	**g**

This approach shows the benefits of breaking down a problem into smaller subproblems. Here, we see how the above problem can be broken down into seemingly unrelated set of subproblems, which can then be used together to solve a grander problem.