Lead TA(s): Namwar Rauf (namwar.rauf@lums.edu.pk)

Guided Exercise 02

Memory Management

Closed Book; Closed Notes; Time Given=120 minutes

"By proceeding I certify that I have neither received nor given unpermitted aid on this examination and that I have reported all such incidents observed by me in which unpermitted aid is given."

Zip the assignment folder and rename it to `'<rollnumer>_GE2.zip'`. Upload your zip file in the corresponding LMS submission tab. Example, `'26100181_GE2.zip'`

Task 1: Dynamic memory allocation (marks = 12).
Implement a program, in '**char* build_string()**' in 'task1.c', that builds a dynamic C-string by taking character-wise input from a user. Implement it according to the following instructions:

1. Prompt the user for a single character input. If the user enters multiple characters per prompt, only accept the first character and discard the rest.
2. Initialize a dynamic character array of size 5 (5 bytes = 5 characters).
3. The dynamic string must be null-terminated. Hence, for n-sized memory space, once n-1 characters have been entered, null-terminate the character array.
4. Once the array has reached its capacity, dynamically resize it to be twice its old size.
5. Stop once the user inputs enter without typing any character (presses 'Enter' twice), dynamically downsize the character array to free up any unused memory space.
6. Return a pointer to the C-string.

- *Important:*

- The terminating condition for input from the user is pressing 'Enter' twice. In the first lab, you must have used 'scanf(" %c", &var)' to get a single character input from the user into a variable 'var'. The space before the '%c' is used to ignore any whitespace characters (like newline or spaces) in the user input. Note that this will not work for the input terminating condition as it would keep ignoring the double 'Enter' and infinitely wait for input. Instead, use 'scanf("%c", var)" to get the input.
- Make sure to manually NULL-terminate your character arrays.
- The test cases for task 1 are all automated. However, when you run the code, it will ask you for input. I have added this for you to manually test your code and see if it is taking input correctly. Once you manually give your input, the automated test cases will run by themselves. Below is a short snippet from the starting portion of the test case output.

```
Testing Build String:

Basic Manual Test:

Input characters one by one (press enter key to finish):

Input a character: a
Input a character: b
Input a character: c
Input a character:

Final string: abc

Auto Test 1:

Input characters one by one (press enter key to finish):

Input a character: Input a character: Input a character: Input a character: Input a character: Input a cha
racter: Input a character: Input a character: Input a character: Input a character: Input a character: Inp
ut a character:

Final string: Hello World

Passed 3/3

Auto Test 2:

Input characters one by one (press enter key to finish):

Input a character: Input a character: Input a character: Input a character: Input a character: Input a cha
racter: Input a character: Input a character: Input a character: Input a character: Input a character:

Final string: abcdefghij

Passed 3/3
```

Figure 1: Task 1 output snippet

Task 2: Copying overlapping memory (marks = 15).
The standard C library provides a function, memcpy, which allows copying data (in bytes) from one memory location to another.

```
1 void *memcpy(void *dest, const void *src, size_t n);
```

The function copies n bytes from a source block of memory to a destination block of memory. However, if the source and destination memory blocks have overlapping segments, the behavior of memcpy will be undefined. Let's consider the memory space of 4 bytes as shown.

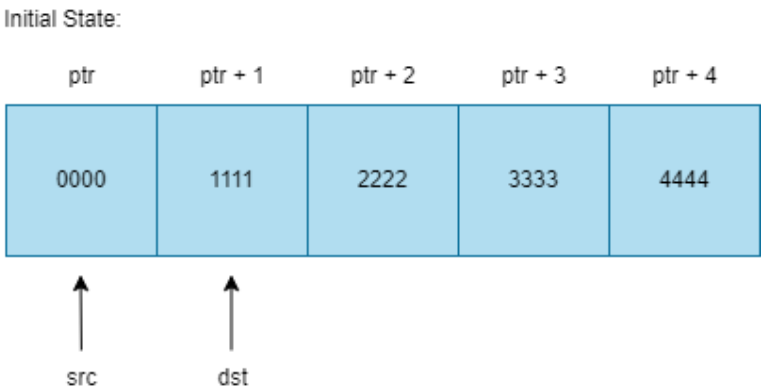

Figure 2: A memory space of size 4 bytes.

Given that 'ptr' points to the first byte of the memory space, if I perform the following call to memcpy(), I risk unexpected behavior which could be anything ranging from no damage to severe data corruption.

```
1 memcpy(ptr + 1, ptr, 3*sizeof(*ptr));
```

The above call attempts to overwrite overlapping bytes of memory. How? Let's illustrate this further. There are two ways that memcpy() could be implemented under the hood. The first is that it simply does a forward traversal of the memory blocks.

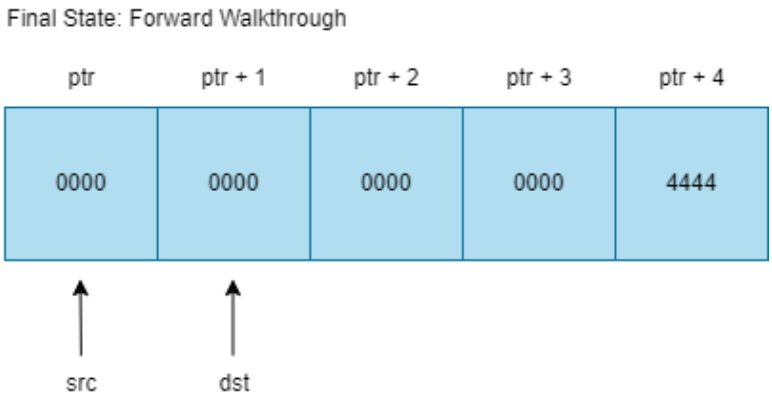

Figure 3: Forward traversal.

However, such a forward traversal results in unexpected behavior! The value '0000' from the 'src' pointer was copied to all three bytes starting from the 'dst' pointer. On the other hand, if memcpy() were to perform a backwards traversal, then the results would be different for this case. In a backwards traversal, first the destination pointer is shifted to **dst + 3*sizeof(*ptr)** and the source pointer is shifted to **src + 3*sizeof(*ptr))**, and then memory is copied by traversing in reverse until the original locations of dst and src are reached.

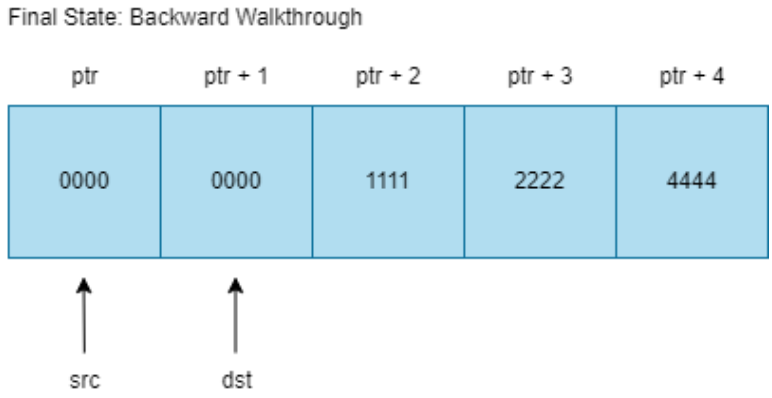

Figure 4: Backward traversal.

As we can see, the memory is copied correctly, and no data is corrupted.

Now on to the task at hand. Given two memory locations (source and destination locations), implement the function '**void safe_memcpy(void *dest, const void *src, size_t n)**' to safely copy memory from the source to the destination without using/declaring any extra memory space. **You are not allowed to copy the data into an array/buffer as an intermediate step. The copying of data must be safely performed IN-PLACE.**

Hint: I have only discussed one case of memcpy failure, you must determine the other case yourself.