

Correction of previous stage questions solution:

Question 2: This question primarily focuses on character storage/representation and type promotion. Here, I was looking for keywords related to type promotion rather than conversion to int.

When compiled under a 32-bit compiler:

1. `sizeof(char) = 1`: The size of a char is 1 byte on a 32-bit platform.
2. `sizeof(char*) = 4`: The size of a char* (pointer to char) is 4 bytes on a 32-bit platform.
3. `sizeof('a') = 4`: The `sizeof('a')` expression does not represent the size of the character literal itself. Instead, it evaluates to the size of an int because character literals in C are promoted to int type. On a 32-bit platform, the size of an int is 4 bytes.
4. `sizeof(ch) = 1`: The size of the ch variable of type char is 1 byte.
5. `sizeof(*s+0) = 4`: The expression `*s + 0` performs an addition operation and, as explained earlier, the `*s` expression is promoted to an int. Therefore, the result is an int with a size of 4 bytes.
6. `sizeof(*s) = 1`: The `*s` expression dereferences the pointer s, resulting in a char value. The size of a char is 1 byte.
7. `sizeof(s) = 4`: The `sizeof(s)` expression returns the size of the s pointer itself, which is 4 bytes on a 32-bit platform.

When compiled under a 64-bit compiler:

1. `sizeof(char) = 1`: The size of a char is still 1 byte on a 64-bit platform.
2. `sizeof(char*) = 8`: The size of a char* (pointer to char) is 8 bytes on a 64-bit platform.
3. `sizeof('a') = 4`: Similar to the 32-bit case, the `sizeof('a')` expression evaluates to the size of an int because character literals are promoted to int type. On a 64-bit platform, the size of an int is still 4 bytes.
4. `sizeof(ch) = 1`: The size of the ch variable of type char remains 1 byte.
5. `sizeof(*s+0) = 4`: The expression `*s + 0` behaves the same as in the 32-bit case, resulting in an int with a size of 4 bytes.
6. `sizeof(*s) = 1`: The size of the `*s` expression, which dereferences the s pointer, is still 1 byte.
7. `sizeof(s) = 8`: The size of the s pointer itself is 8 bytes on a 64-bit platform.

In summary, the size of a char is always 1 byte regardless of the platform. The size of a char* depends on the platform (4 bytes for 32-bit, 8 bytes for 64-bit). Character literals are promoted to int type when used in expressions, resulting in the size of an int (4 bytes in both cases). The size of a char variable is always 1 byte. The size of a dereferenced pointer expression, `*s`, is the size of the pointed-to type (char in this case, so 1 byte). The size of a pointer itself depends on the platform (4 bytes for 32-bit, 8 bytes for 64-bit).

Question 3:

The given code was:

```
int main(){
    char a1[3][4]={"1X","2Y","3ZX"};
    char (*p1)[4];
    char **pa1 = a1;
    for(p1=a1; p1<a1+3; p1++, pa1++){
        printf("*p1 = %s\n",*p1);
        printf("pa1 = %s\n",pa1);
    }
}
```

```

printf("a[3][4]=%d\n",a1[3][4]);
printf("a1 size: %dBytes\n",sizeof(a1));

//char *a2[3]={ "1XYZ1X","2XYZ2X","3"};
char *a2[3]={ "1XYZ123","2XYZ123","3XYZ123"};
char **p2;
for(p2=a2; p2<a2+3; p2++){
    printf("%s\n",*p2);
}
printf("a2 size: %dBytes\n",sizeof(a2));
printf("a2[1] size: %dBytes\n",sizeof(a2[1]));
printf("a2 size: %dBytes\n",sizeof(*a2[1]));
return 0;
}

```

The Questions you want the answers are:

For the third level, I want you to research:

1. Why is `*p1` used here? Can `p1` be used instead? What is the difference between the two?
2. Why are `pa1` and `*p2` used instead of `*pa1` and `p2`? What is `**pa1`? What about `**p2`?
3. Investigate the step size of each pointer in more depth.

Answer:

1. In the given code, `*p1` is used to access the value pointed to by `p1`, while `p1` itself represents a pointer to an array of characters. In other words, `*p1` dereferences the pointer and gives you the actual value stored in the memory location pointed to by `p1`. On the other hand, using just `p1` without the asterisk (`*`) would give you the memory address of the array.

To illustrate the difference, let's consider an example. Suppose we have `char a[4] = "abc";`. If we declare a pointer `char *p = a;`, then `p` points to the first character in the array `a`. By using `*p`, we can access the value 'a' stored in the memory location pointed to by `p`. However, if we use `p` without the asterisk, it represents the memory address of the first character in the array.

2. In the code, `pa1` is declared as a pointer to a pointer to char (`char **`). It is assigned the value of `a1`, which is an array of arrays of char. When `pa1` is incremented (`pa1++`), it moves to the next pointer in the array `a1`, which means it points to the next row in the `a1` array.

Similarly, `*p2` represents the value pointed to by `p2`, where `p2` is declared as a pointer to a pointer to char (`char **`). In the code, `p2` is assigned the value of `a2`, which is an array of pointers to char. When `p2` is incremented (`p2++`), it moves to the next pointer in the array `a2`, pointing to the next element in the array.

The double asterisks (`**`) indicate a pointer to a pointer. In the given code, they are used to handle arrays of strings or arrays of arrays.

3. The step size of each pointer depends on the size of the data type it points to. In the first part of the code, where `p1` and `pa1` are used, the step size is determined by the size of `char[4]` because `p1` is a pointer to an array of characters with size 4.

In the second part of the code, where `p2` is used, the step size is determined by the size of a pointer to char (`char *`). This step size depends on the system architecture, but it is typically 4 bytes on 32-bit systems and 8 bytes on 64-bit systems.

It's important to note that the step size affects how the pointer moves through the array when incremented or decremented. By incrementing a pointer, you can move it to the next element of the array it points to.

Question 4: You should immediately check if the received string exceeds the array boundary, rather than waiting for later logic: `char str[20]; gets(str)` You need to check if `strlen(str) < 20`.

```
#include <stdio.h>
#include <string.h>

void my_strcat(char *s, const char *t, size_t max_length) {
    if (s == NULL || t == NULL) {
        printf("Invalid input: NULL pointer.\n");
        return;
    }

    size_t s_length = strlen(s);
    size_t t_length = strlen(t);

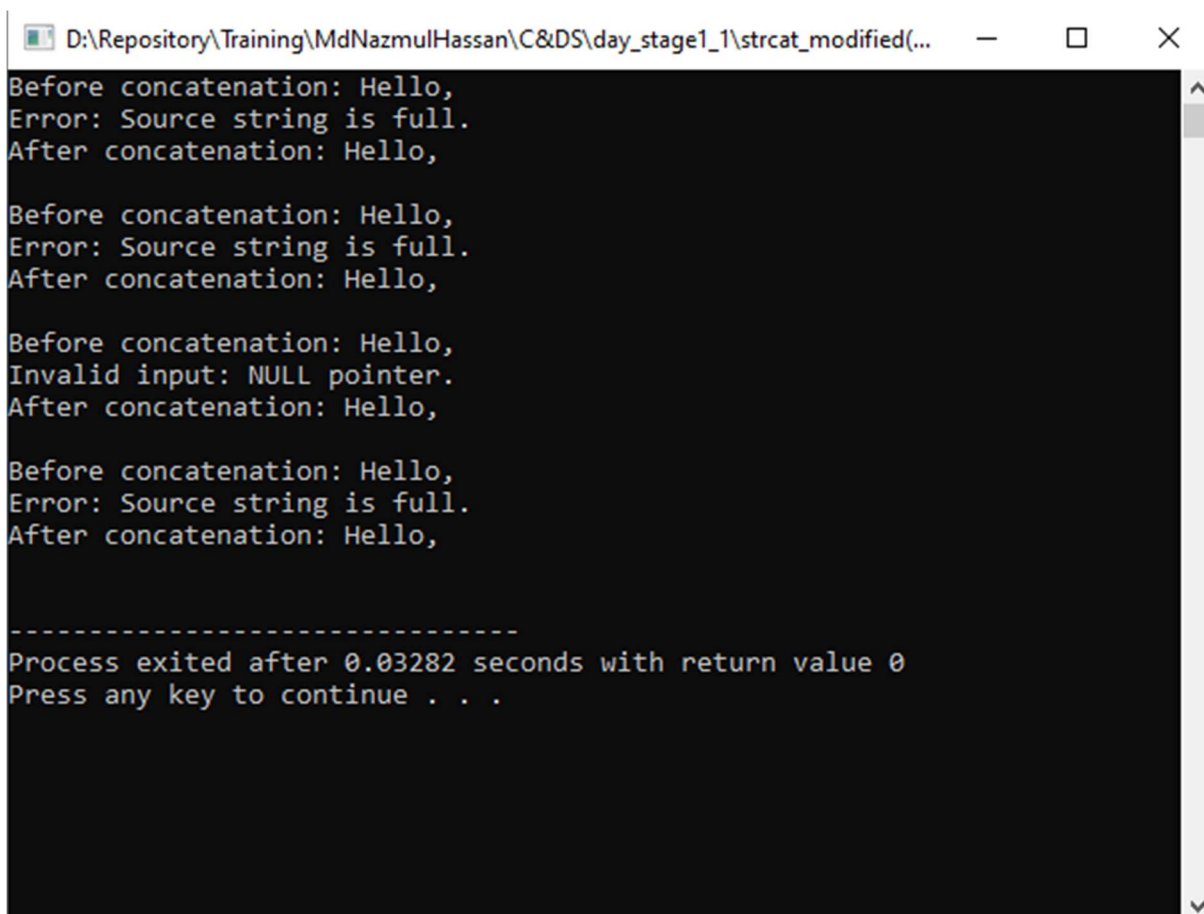
    if(s_length <= max_length) {
        printf("Error: Source string is full.\n");
        return;
    }
    else if (s_length + t_length >= max_length) {
        printf("Error: Insufficient space to concatenate strings.\n");
        return;
    }

    while (*s != '\0') {
        s++;
    }

    while ((*s = *t) != '\0') {
        s++;
        t++;
    }
}
```

This condition is added to fulfill the questions requirements.

Output:



```
D:\Repository\Training\MdNazmulHassan\C&DS\day_stage1_1\strcat_modified(...)
Before concatenation: Hello,
Error: Source string is full.
After concatenation: Hello,

Before concatenation: Hello,
Error: Source string is full.
After concatenation: Hello,

Before concatenation: Hello,
Invalid input: NULL pointer.
After concatenation: Hello,

Before concatenation: Hello,
Error: Source string is full.
After concatenation: Hello,

-----
Process exited after 0.03282 seconds with return value 0
Press any key to continue . . .
```

Additional Exercises

Question 1:

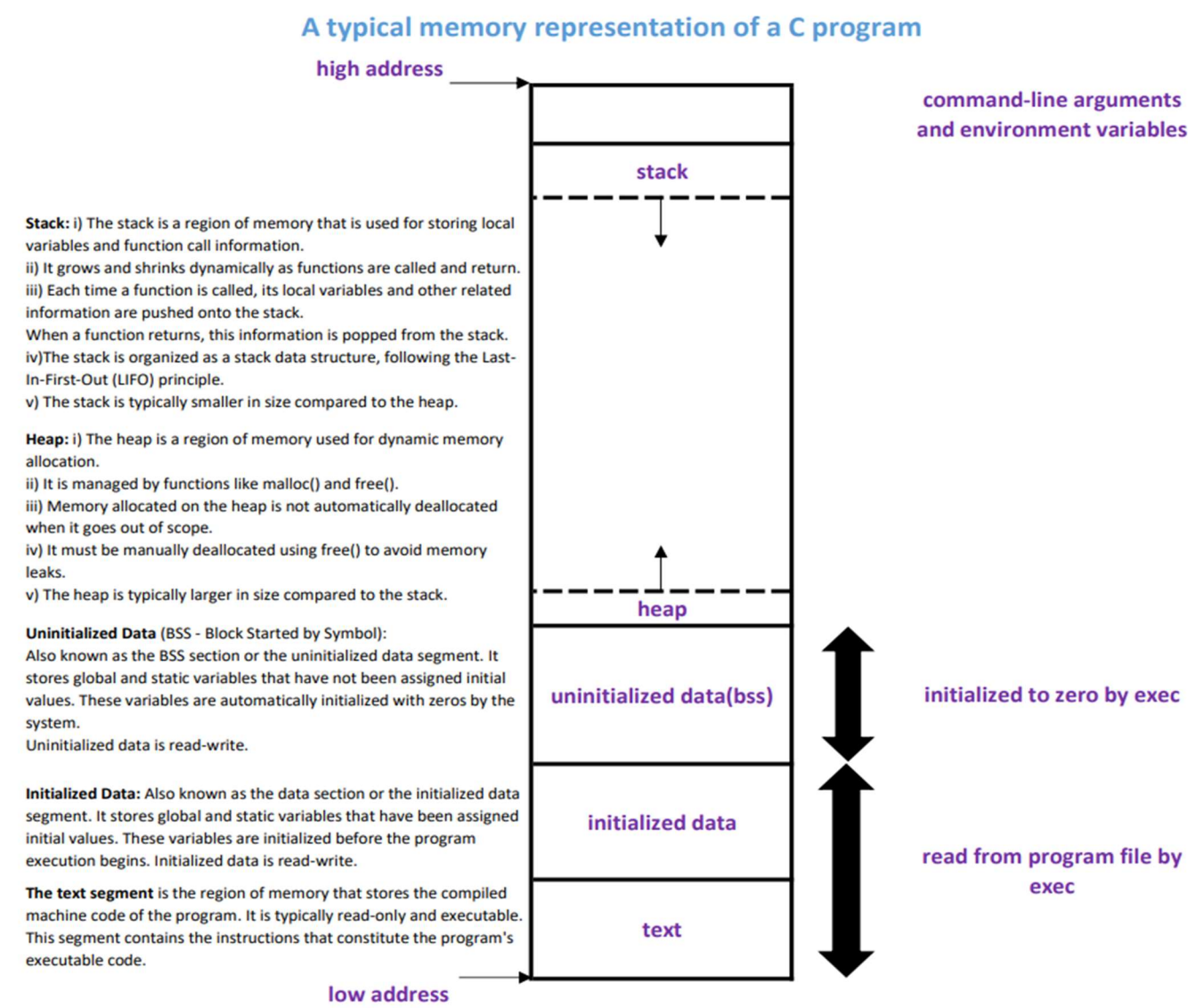


Fig. 1: A typical memory layout of a running process.

Representation of the following variable and constant positions and corresponding relationships in the memory distribution diagram,

variable or constant	length (Byte)	storage location	location	content
str1	4	data	address in text segment	"hello world"
int_a	4	data	static memory	3
int_b	4	data	constant memory	4
str2	4	stack	address in text segment	"hello world"
str3	4	stack	an address in heap	dynamic memory
str4	20	stack	constant memory	"hello world"
int_c	4	data	static memory	5
int_d	4	data	constant memory	6

```
char *str1 = "hello world";  
  
static int int_a = 3;  
  
const int int_b = 4;
```

```

void main(void)
{
    char *str2 = "hello world";

    char *str3 = malloc(100);

    char str4[20] = "hello world";

    static int int_c = 5;

    const int int_d = 6;

    free(str3);

    return;
}

```

Explanation:

str1: The pointer str1 is a global variable stored in the data segment, and it points to the address of the string literal "hello world" in the text segment.

int_a: The variable int_a is a static global variable stored in the data segment, and it has a value of 3.

int_b: The variable int_b is a constant global variable stored in the data segment, and it has a value of 4.

str2: The pointer str2 is a local variable stored on the stack, and it points to the address of the string literal "hello world" in the text segment.

str3: The pointer str3 is a local variable stored on the stack, and it points to the address of dynamically allocated memory on the heap using malloc(100).

str4: The character array str4 is a local variable stored on the stack, and it holds the string "hello world" in contiguous memory of size 20.

int_c: The variable int_c is a static local variable stored in the data segment, and it has a value of 5.

int_d: The variable int_d is a constant local variable stored in the data segment, and it has a value of 6.

Question 3:

1.

On a 32-bit platform, the addresses of array1 + 1, &array1 + 1, and &array1[0] + 1 will behave as follows:

- array1 + 1: The address of array1 + 1 will be the address of the second element of the array1 character array. Since each element of array1 is of type char (1 byte), adding 1 to the address will move the pointer by 1 byte. Therefore, the address of array1 + 1 will be address_of_array1 + 1 byte.
- &array1 + 1: The address of &array1 + 1 will be the address that is sizeof(array1) bytes past the address of the entire array1 array. The sizeof(array1) is equal to 20 * sizeof(char) (assuming sizeof(char) is 1 on your platform), which is 20 bytes. Adding 1 to the address will move the pointer by sizeof(array1) bytes. Therefore, the address of &array1 + 1 will be address_of_array1 + 20 bytes.
- &array1[0] + 1: The address of &array1[0] + 1 will be the address of the second element of the array1 character array. Since each element of array1 is of type char (1 byte), adding 1 to the address will move the pointer by 1 byte. Therefore, the address of &array1[0] + 1 will be address_of_array1 + 1 byte.

To summarize:

- array1 + 1 moves the pointer by 1 byte (address_of_array1 + 1 byte).
- &array1 + 1 moves the pointer by sizeof(array1) bytes (address_of_array1 + 20 bytes).
- &array1[0] + 1 moves the pointer by 1 byte (address_of_array1 + 1 byte).

These values behave this way because of how pointer arithmetic works in C. When you add an integer to a pointer, the result is a pointer that points to the memory location offset by the number of elements (not bytes) multiplied by the size of each element. In this case, since the type of array1 is char, each element is 1 byte, so adding 1 to the pointer moves it by 1 byte. Similarly, adding sizeof(array1) moves the pointer by sizeof(array1) bytes, which is the total size of the array.

2.

On a 32-bit platform, the addresses of array2 + 1, &array2 + 1, &array2[0] + 1, p2 + 1, *p2 + 1, p2, and *p2 will behave as follows:

1. `array2 + 1`: The address of `array2 + 1` will be the address of the second element in the `array2` array. Since each element of `array2` is an array of 5 characters, adding 1 to the address will move the pointer by `sizeof(array2[0])` bytes. Therefore, the address of `array2 + 1` will be `address_of_array2 + sizeof(array2[0])`.
2. `&array2 + 1`: The address of `&array2 + 1` will be the address that is `sizeof(array2)` bytes past the address of the entire `array2` array. The `sizeof(array2)` is equal to the total size of `array2`, which is `3 * sizeof(array2[0])` (assuming `sizeof(array2[0])` is `5 * sizeof(char)` on your platform). Adding 1 to the address will move the pointer by `sizeof(array2)` bytes. Therefore, the address of `&array2 + 1` will be `address_of_array2 + sizeof(array2)`.
3. `&array2[0] + 1`: The address of `&array2[0] + 1` will be the address of the second element in the `array2` array. Since each element of `array2` is an array of 5 characters, adding 1 to the address will move the pointer by `sizeof(array2[0])` bytes. Therefore, the address of `&array2[0] + 1` will be `address_of_array2 + sizeof(array2[0])`.
4. `p2 + 1`: The address of `p2 + 1` will be the address that is `sizeof(*p2)` bytes past the address of `p2`. Since `p2` is a pointer to an array of 5 characters, `sizeof(*p2)` is equal to `sizeof(array2[0])`, which is the size of one element in `array2`. Adding 1 to the address will move the pointer by `sizeof(*p2)` bytes. Therefore, the address of `p2 + 1` will be `address_of_p2 + sizeof(*p2)`.
5. `*p2 + 1`: The address of `*p2 + 1` will be the address of the second character in the first element of `array2`. Since `*p2` dereferences the pointer `p2`, it points to the first element in `array2`, which is an array of 5 characters. Adding 1 to the address will move the pointer by `sizeof(**p2)` bytes, which is the size of one character. Therefore, the address of `*p2 + 1` will be `address_of_first_element_of_array2 + sizeof(**p2)`.
6. `p2`: The value of `p2` is the address of the first element in `array2`. It points to an array of 5 characters. Therefore, the value of `p2` will be `address_of_first_element_of_array2`.
7. `*p2`: The value of `*p2` is the first element in `array2`, which is an array of 5 characters. Therefore, the value of `*p2` will be the same as the address of the first element of `array2`.

These values behave this way because of how pointer arithmetic works in C. When you add an integer to a pointer, the result is a pointer that points to the memory location offset by the number of elements (not bytes) multiplied by the size of each element. In the case of arrays, the size of the element is determined by the size of the array's type.

3.

On a 32-bit platform, the addresses and behaviors of `array3 + 1`, `&array3 + 1`, `&array3[0] + 1`, `p3 + 1`, `*p3`, `*p3 + 1`, `**p3`, and `**p3 + 1` are as follows:

1. `array3 + 1`: The address of `array3 + 1` will be the address of the second element in `array3`. Since each element of `array3` is a pointer to a string, adding 1 to the address will move the pointer by `sizeof(array3[0])` bytes. Therefore, the address of `array3 + 1` will be `address_of_array3 + sizeof(array3[0])`.
2. `&array3 + 1`: The address of `&array3 + 1` will be the address that is `sizeof(array3)` bytes past the address of the entire `array3`. The `sizeof(array3)` is equal to the total size of `array3`, which is `2 * sizeof(array3[0])` (assuming `sizeof(array3[0])` is the size of a pointer on your platform). Adding 1 to the address will move the pointer by `sizeof(array3)` bytes. Therefore, the address of `&array3 + 1` will be `address_of_array3 + sizeof(array3)`.
3. `&array3[0] + 1`: The address of `&array3[0] + 1` will be the address of the second element in `array3`. Since each element of `array3` is a pointer to a string, adding 1 to the address will move the pointer by `sizeof(array3[0])` bytes. Therefore, the address of `&array3[0] + 1` will be `address_of_array3 + sizeof(array3[0])`.
4. `p3 + 1`: The address of `p3 + 1` will be the address that is `sizeof(*p3)` bytes past the address of `p3`. Since `p3` is a pointer to a pointer to a string, `sizeof(*p3)` is equal to `sizeof(array3[0])`, which is the size of one element in `array3`. Adding 1 to the address will move the pointer by `sizeof(*p3)` bytes. Therefore, the address of `p3 + 1` will be `address_of_p3 + sizeof(*p3)`.
5. `*p3`: The value of `*p3` is the first element in `array3`, which is a pointer to a string. Therefore, the value of `*p3` will be the address of the first element in `array3`.
6. `*p3 + 1`: The address of `*p3 + 1` will be the address of the second character in the string pointed to by the first element of `array3`. Adding 1 to the address will move the pointer by `sizeof(**p3)` bytes, which is the size of one character. Therefore, the address of `*p3 + 1` will be `address_of_string + sizeof(**p3)`.

7. `**p3`: The value of `**p3` is the first character of the string pointed to by the first element of `array3`. Therefore, the value of `**p3` will be the first character in the string.
8. `**p3 + 1`: The address of `**p3 + 1` will be the address of the second character in the string pointed to by the first element of `array3`. Adding 1 to the address will move the pointer by `sizeof(**p3)` bytes, which is the size of one character. Therefore, the address of `**p3 + 1` will be `address_of_string + sizeof(**p3)`.

The behaviors of these values follow the rules of pointer arithmetic in C. Adding an integer to a pointer adjusts the address based on the size of the pointed-to type. The addresses depend on the sizes of the elements in `array3` and `p3`, and the increments move the pointers accordingly.