Date: July 3, 2023

Additional Exercises

Md. Mahfuj Hasan Shohug

BDCOM0019

1. **Question 1:** represent the following variable and constant positions and corresponding relationships in the memory distribution diagram, taking str3 as an example.

variable or constant	length(Byte)	storage location	content
str3	4	stack	a address in heap

```
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;

}
```

Answer: Here I am representing the following variable and constant positions and corresponding relationships in the memory distribution diagram according to the given example:

variable or constant	length(Byte)	storage location	content
str1	**4**	**stack**	**Address of string literal "hello world"**
int_a	**4**	**Data Segment**	**3**
int_b	**4**	**Data Segment**	**4**
str2	**4**	**Stack**	**Address of string literal "hello world"**
str3	**4**	**Stack**	**Address of dynamically allocated memory (heap)**
str4	**20**	**Stack**	**Array of characters containing "hello world"**
int_c	**4**	**Data Segment**	**5**
int_d	**4**	**Data Segment**	**6**

Storage locations are categorized in memory distribution as "data" for global variables and constants and "stack" for local variables. Each variable or constant's associated values or addresses are shown in the content column of the table. Global variables and constants like int_a, int_b, int_c, and int_d are stored in the "data" portion of the figure above. The contents column displays their corresponding values or contents. Local variables, on the other hand, like str4, str3, str2, str1, and length, are kept in the "stack" segment. The content column lists their URL or contact information.

The heap is used to store the dynamic memory that malloc() allocated for str3 in the sample code. The parameter supplied to malloc() determines the heap size, which in our example is 100 bytes. However, since str3 is a pointer and operates in a 32-bit environment, the memory reference it is assigned to can be stored in just 4 bytes.

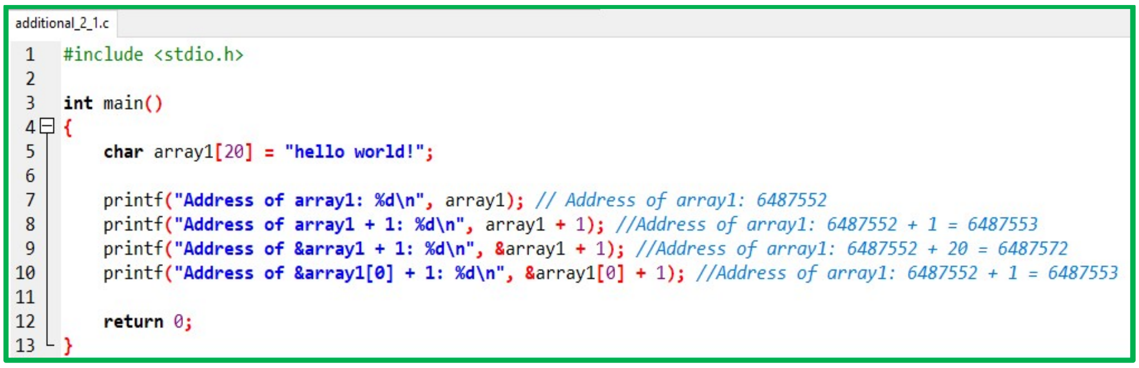
In summary, the table explains memory allocation by dividing variables and constants into "data" and "stack" pieces. The contents column lists the corresponding values, addresses, or memory locations for each of these sections. In a 32-bit system, dynamic memory allotted for str3 using malloc() is stored on the heap and needs 4 bytes.

2. **Question 3:** Based on 32-bit platform,Notice that, what I care about is why rather than what the address is!

Date: July 3, 2023

1. char array1[20] = "hello world!"; array1 + 1, &array1 + 1, &array1[0] + 1， What are their addresses?Why do these values behave this way?

Answer: For visualization those address value and then describe why those values behave this way. Code:



additional_2_1.c

```c
#include <stdio.h>

int main()
{
    char array1[20] = "hello world!";

    printf("Address of array1: %d\n", array1); // Address of array1: 6487552
    printf("Address of array1 + 1: %d\n", array1 + 1); //Address of array1: 6487552 + 1 = 6487553
    printf("Address of &array1 + 1: %d\n", &array1 + 1); //Address of array1: 6487552 + 20 = 6487572
    printf("Address of &array1[0] + 1: %d\n", &array1[0] + 1); //Address of array1: 6487552 + 1 = 6487553

    return 0;
}
```

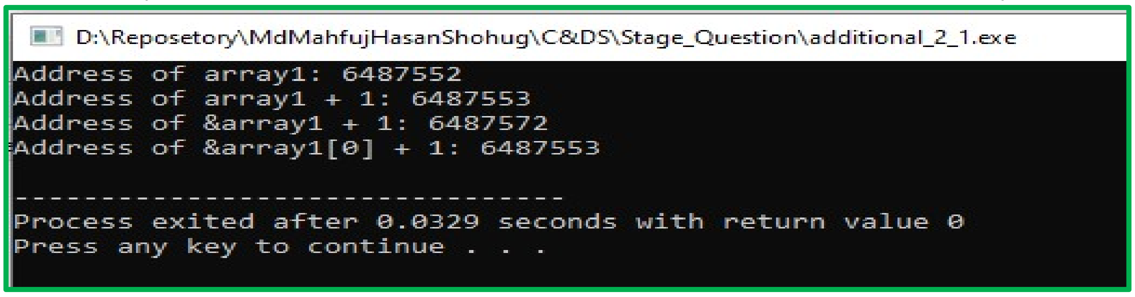
array1 + 1: The address of array1 + 1 will point to the second element of the array1 character array, which is the character 'e'. It increments the address by the size of the data type, which is 1 byte for char. The behavior here is due to pointer arithmetic, where adding 1 to a pointer moves it to the next element in the array.

&array1 + 1: The address of &array1 + 1 will point to the memory location after the entire array1 array. It increments the address by the size of the whole array, which is 20 * sizeof(char) = 20 bytes. This behavior occurs because &array1 gives the address of the whole array, and adding 1 to it moves to the next block of memory after the array.

&array1[0] + 1: The address of &array1[0] + 1 will point to the second element of the array1 character array, similar to array1 + 1. It increments the address by the size of the data type, which is 1 byte for char. This behavior is equivalent to taking the address of the first element of the array and adding 1 to it.

In summary, array1 + 1 and &array1[0] + 1 both point to the second element of the array1 character array, while &array1 + 1 points to the memory location after the entire array1 array. The behavior of these expressions is consistent with pointer arithmetic and the size of the data type being accessed.

The output shows the decimal addresses of these expressions.



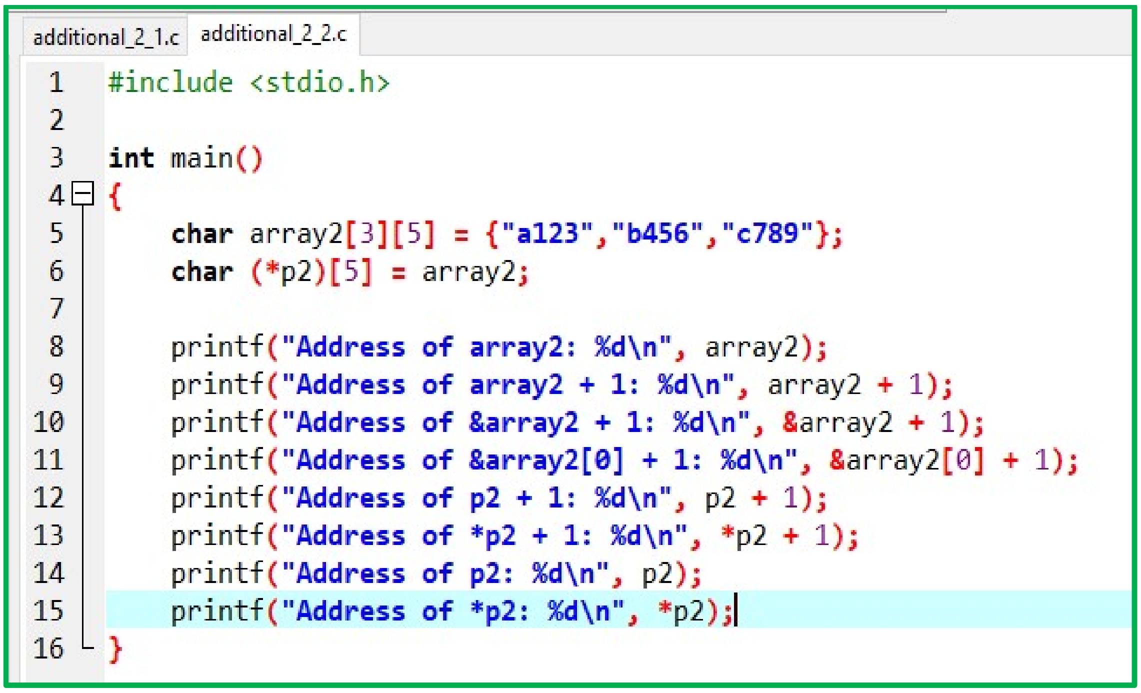
D:\Reposetory\MdMahfujHasanShohug\C&DS\Stage_Question\additional_2_1.exe

```
Address of array1: 6487552
Address of array1 + 1: 6487553
Address of &array1 + 1: 6487572
Address of &array1[0] + 1: 6487553

--------------------------------
Process exited after 0.0329 seconds with return value 0
Press any key to continue . . .
```

2.char array2[3][5]={"a123","b456", "c789"}; char (*p2)[5] = array2; array2 + 1, &array2 + 1, &array2[0] + 1, p2 + 1, *p2 + 1,p2, *p2, What are their addresses?Why do these values behave this way?

Answer: Let's analyze the addresses and behaviors of the expressions those are mentioned on this question:

Code file:



```
#include <stdio.h>

int main()
{
    char array2[3][5] = {"a123","b456","c789"};
    char (*p2)[5] = array2;

    printf("Address of array2: %d\n", array2);
    printf("Address of array2 + 1: %d\n", array2 + 1);
    printf("Address of &array2 + 1: %d\n", &array2 + 1);
    printf("Address of &array2[0] + 1: %d\n", &array2[0] + 1);
    printf("Address of p2 + 1: %d\n", p2 + 1);
    printf("Address of *p2 + 1: %d\n", *p2 + 1);
    printf("Address of p2: %d\n", p2);
    printf("Address of *p2: %d\n", *p2);
}
```

In this example, we have a two-dimensional character array array2 with 3 rows and 5 columns. We also declare a pointer to an array of 5 characters p2 and initialize it with the address of array2.

array2 + 1: This expression points to the memory location of the second row of array2. It increments the address by the size of a row, which is **5 * sizeof(char) = 5 bytes.**

&array2 + 1: This expression points to the memory location after the entire array2 two-dimensional array. It increments the address by the size of the whole array, which is **3 * 5 * sizeof(char) = 15 bytes.**

&array2[0] + 1: This expression points to the second element of the first row of array2, similar to array2 + 1. It increments the address by the size of a row, which is **5 * sizeof(char) = 5 bytes.**

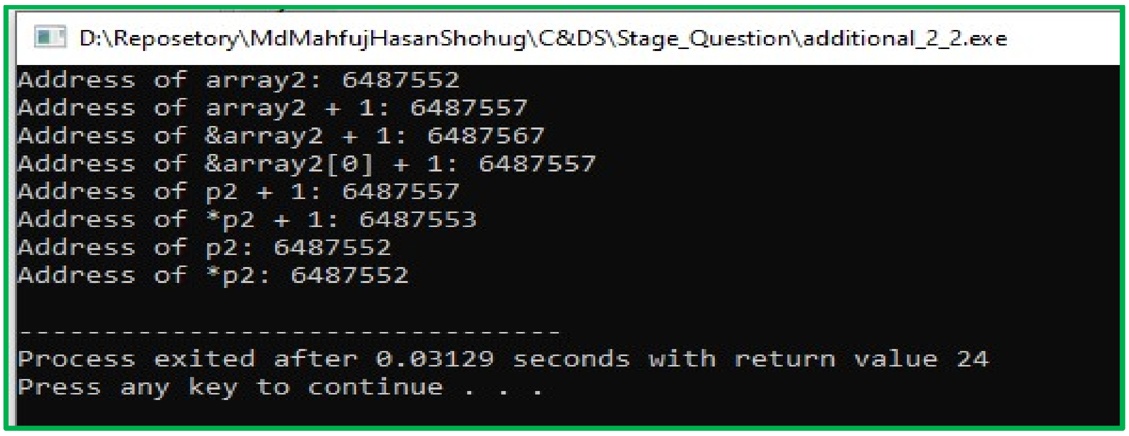
p2 + 1: This expression points to the memory location of the second row of array2. Since p2 is a pointer to an array of 5 characters, incrementing it moves to the next row, which is **5 * sizeof(char) = 5 bytes.**

*p2 + 1: This expression points to the second element of the first row of array2, similar to &array2[0] + 1. It increments the address by the size of a character, which is 1 byte.

p2: This is the address of the p2 pointer itself, which points to the first row of array2.

*p2: This is the address of the first row of array2, similar to array2.

The output shows the decimal addresses of these expressions. The behaviors of these values are consistent with pointer arithmetic and the size of the data type or the array being accessed.

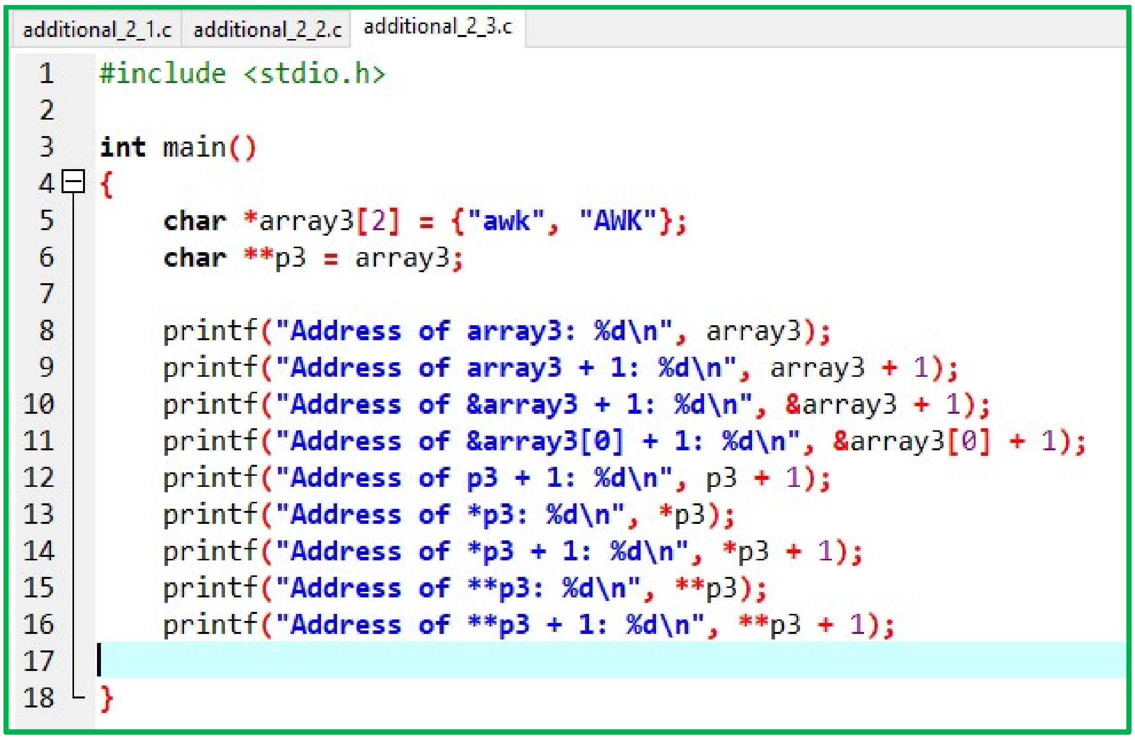


D:\Reposetory\MdMahfujHasanShohug\C&DS\Stage_Question\additional_2_2.exe

```
Address of array2: 6487552
Address of array2 + 1: 6487557
Address of &array2 + 1: 6487567
Address of &array2[0] + 1: 6487557
Address of p2 + 1: 6487557
Address of *p2 + 1: 6487553
Address of p2: 6487552
Address of *p2: 6487552

--------------------------------
Process exited after 0.03129 seconds with return value 24
Press any key to continue . . .
```

3. char *array3[2]={"awk","AWK"}; char **p3 = array3; array3 + 1, &array3 + 1, &array3[0] + 1,p3 + 1, *p3, *p3 + 1, **p3, **p3 + 1,   What are their addresses?Why do these values behave this way?

Answer: Code file:

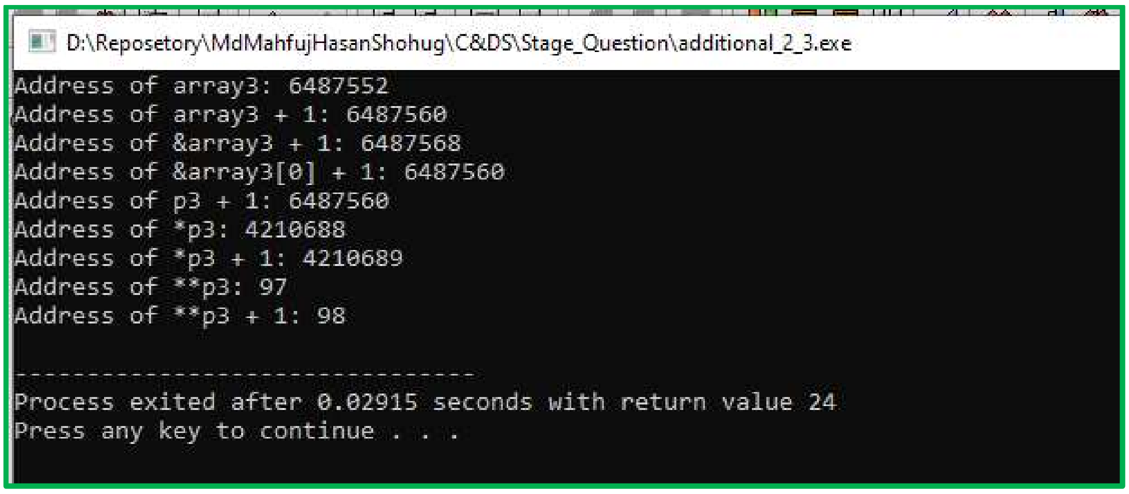


additional_2_1.c additional_2_2.c additional_2_3.c

```c
#include <stdio.h>

int main()
{
    char *array3[2] = {"awk", "AWK"};
    char **p3 = array3;

    printf("Address of array3: %d\n", array3);
    printf("Address of array3 + 1: %d\n", array3 + 1);
    printf("Address of &array3 + 1: %d\n", &array3 + 1);
    printf("Address of &array3[0] + 1: %d\n", &array3[0] + 1);
    printf("Address of p3 + 1: %d\n", p3 + 1);
    printf("Address of *p3: %d\n", *p3);
    printf("Address of *p3 + 1: %d\n", *p3 + 1);
    printf("Address of **p3: %d\n", **p3);
    printf("Address of **p3 + 1: %d\n", **p3 + 1);

}
```

In this example, we have an array of two pointers to characters array3, initialized with the strings "awk" and "AWK". We also declare a pointer to a pointer to a character p3 and initialize it with the address of array3.

array3 + 1: This expression points to the memory location of the second element in array3, which is the second pointer in the array. It increments the address by the size of a pointer, which is **sizeof(char*) bytes.**
&array3 + 1: This expression points to the memory location after the entire array3 array. It increments the address by the size of the whole array, which is 2 * sizeof(char*) bytes.
&array3[0] + 1: This expression points to the memory location after the first pointer in array3, similar to array3 + 1. It increments the address by the size of a pointer, which is **sizeof(char*) bytes.**
p3 + 1: This expression points to the memory location of the second element in array3, which is the second pointer in the array. Since p3 is a pointer to a pointer, incrementing it moves to the next element in the array, which is sizeof(char*) bytes.
*p3: This is the address of the first character in the first string, similar to array3[0]. It dereferences p3, giving the value of the first pointer in array3.
*p3 + 1: This expression points to the second character in the first string, similar to &array3[0] + 1. It increments the address by the size of a character, which is 1 byte.
**p3: This is the address of the first character in the first string, similar to *array3[0]. It dereferences *p3, giving the value of the first character in the first string.
p3 + 1: This expression points to the second character in the first string, similar to *p3 + 1. It increments the address by the size of a character, which is **1 byte.
The output shows the decimal addresses of these expressions. The behaviors of these values are consistent with pointer arithmetic and the size of the data type being accessed.



D:\Reposetory\MdMahfujHasanShohug\C&DS\Stage_Question\additional_2_3.exe

```
Address of array3: 6487552
Address of array3 + 1: 6487560
Address of &array3 + 1: 6487568
Address of &array3[0] + 1: 6487560
Address of p3 + 1: 6487560
Address of *p3: 4210688
Address of *p3 + 1: 4210689
Address of **p3: 97
Address of **p3 + 1: 98

--------------------------------
Process exited after 0.02915 seconds with return value 24
Press any key to continue . . .
```

4. Make a list of all the pointer ++ values and behaviors you have master now, and understand why. Take question 2 above for example array2 + 1， step size 5（Number of columns）&array2 + 1， step size 15 = 3 * 5， p2++， step size 5（Number of columns ） *p2 + 1， step size 1

Answer: Surly, after learning the pointer lets I want to summarize the pointer increment values and behaviors based on the example in question 2 (array2[3][5]={"a123","b456", "c789"}):

array2 + 1: This increments the address by the size of a row, which is equal to the number of columns (5) multiplied by the size of each element (typically 1 byte for char). In this case, the **step size is 5.**

&array2 + 1: This increments the address by the size of the entire two-dimensional array. The size is calculated by multiplying the number of rows (3) by the number of columns (5) and the size of each element. In this case, the step size is **15 = 3 * 5**.

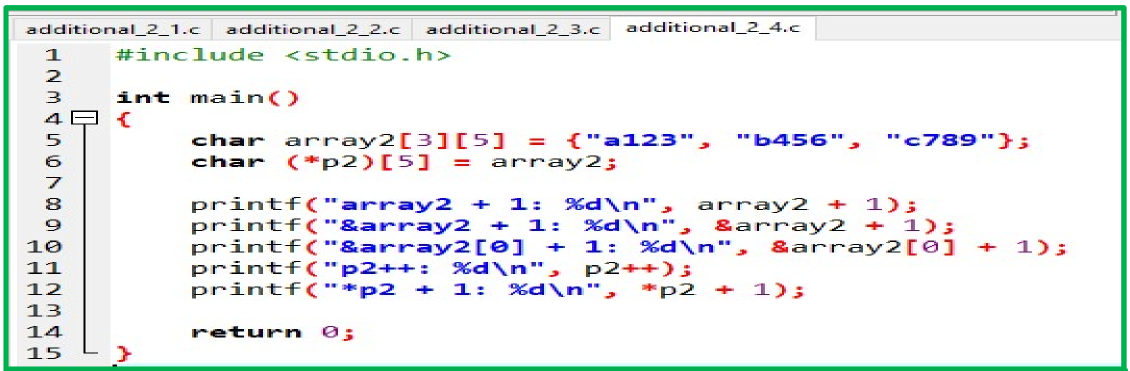
&array2[0] + 1: This increments the address by the size of a row, similar to array2 + 1. The **step size is 5**.

p2++: This increments the address stored in p2 by the size of an array of 5 characters (one row). In this case, **the step size is 5.**

*p2 + 1: This increments the address stored in *p2 by the size of each element, which is typically 1 byte for char. **The step size is 1.**

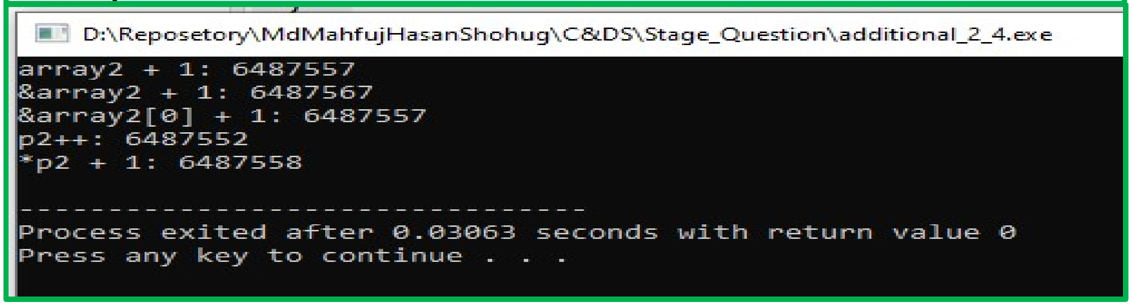
These pointer increment values and behaviors are based on the size of the data type or the array being accessed. Incrementing a pointer moves it to the next element or row in memory based on the size of that element or row.

It's important to note that the step sizes mentioned here are specific to the example provided (array2[3][5]). If you have a different array size or data type, the step sizes would change accordingly. The source code and also the output in decimal is here now:



additional_2_1.c | additional_2_2.c | additional_2_3.c | additional_2_4.c

```c
#include <stdio.h>

int main()
{
    char array2[3][5] = {"a123", "b456", "c789"};
    char (*p2)[5] = array2;

    printf("array2 + 1: %d\n", array2 + 1);
    printf("&array2 + 1: %d\n", &array2 + 1);
    printf("&array2[0] + 1: %d\n", &array2[0] + 1);
    printf("p2++: %d\n", p2++);
    printf("*p2 + 1: %d\n", *p2 + 1);

    return 0;
}
```



D:\Reposetory\MdMahfujHasanShohug\C&DS\Stage_Question\additional_2_4.exe

```
array2 + 1: 6487557
&array2 + 1: 6487567
&array2[0] + 1: 6487557
p2++: 6487552
*p2 + 1: 6487558

--------------------------------
Process exited after 0.03063 seconds with return value 0
Press any key to continue . . .
```

Also form the question number 1:

Arrat1[20] = “Hello World”

Array1 + 1 = step size = 1 * sizeof(char) = 1.

Same as array[0] + 1.

But for array for 20 size step size is = 20 * sizeof(char) = 20.

All the pointer increment values and behaviors the step size is calculated.