

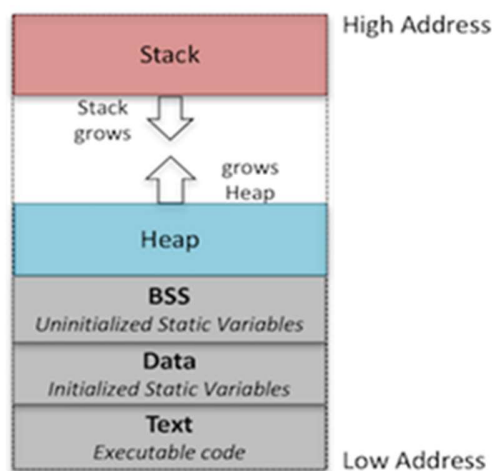
Stage Question Documentation

Md. Mahfuj Hasan Shohug

BDCOM0019

1. **Question 1:** Memory partitions when C programs run. Such as code, data, bss, heap, stack. Please consult the information to understand it.

Answer: Lets, Here I am describing the memory partitions when the C programs run with specific example of Code, Data, BSS, Heap, and Stack. Generally, a memory space consists of sufficient segments, each with its own function. Here is one sample example for memory allocation definition feature image for better understand:



Let's interrogate each of them using suitable examples:

Code Section: This 'Code' section also known as the text section. In the code section the machine code of the program contains to compile the program. It's usually stores the instructions and read-only the instructions when the processor executes. Here is the example of code:

```
#include <stdio.h>
int main()
{
    printf("My Favorite Number Is %d", 75);
    return 0;
}
```

In this example, the Code section is mainly compiled machine code for printf() function and the main() function.

Data Section: The Data section contains the global and also static variables those are declared. In this data section usually divided into two subcategory:

- Initialized data section;
- Uninitialized data section;

```
#include <stdio.h>

int global_var = 10; // Initialized data section
static int static_var; // Uninitialized data section

int main()
{
    printf("Initialized data = %d\n", global_var);
    // automatically for static variable initialize with 0, The output will be 0.
    printf("The Static variable = %d\n", static_var);
    return 0;
}
```

This is the example of data section of the memory.

BSS Section: BSS is called “Block Started by Symbol”. This memory section contains uninitialized global and static variables. Therefore those variables are initialized by 0 (zero) by default.

```
#include <stdio.h>
//Global Variable
int global_var; // Uninitialized data section
static int static_var; // Uninitialized data section

int main()
{
    // Print 0 by default
    printf("Initialized data = %d\n", global_var);
    printf("The Static variable = %d\n", static_var);
    return 0;
}
```

The BSS section contains the globalVar and staticVar variables.

Heap Section: The heap section is its dynamically allocated memory segment that is used for dynamic memory allocation. It is fully managed by functions those are “malloc()” and “free()” and is used for storing data structures and also data allocation. The released memory is removed from the heap (the heap is reduced). Here is the example of heap:

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

int main()
{
    int* dynamicArray = (int*)malloc(5 * sizeof(int));
    // Use dynamicArray...
```

```
    free(dynamicArray);
    printf("The output: %d", dynamicArray);
    return 0;
}
```

Stack Section: It is a local variable that the user created to temporarily store the program, which includes variables defined in function brackets {}, apart from variables declared by static, which means storing variables in the data segment. Mainly, Local variables and information about calls to functions are kept in memory in a location called the stack. Here is the example of the Stack:

```
#include <stdio.h>
#include <stdlib.h>
int global_var = 0; // Global initialization area, BSS
char * global_point1; // Global uninitialized area, BSS
int main()
{
    int stack_var; // Stack
    char str_s[]="Hello"; // String Stack; "Hello" is in the constant area
    char *global_point2; // Stack
    char *global_point3 = "123456"; // 123456 0 is in the constant area.
    static int static_var = 0; // Global (static) initialization area
    global_point1 = (char *) malloc (10); // The allocated area of 10
    global_point2 = (char *) malloc (20); // 20 is located in the heap area.

    return 0;
}
```

2. **Question 2:** Please give the following program running results and explain why. Compile under the 32-bit and 64-bit compilers separately and explain the output results separately.

The given program:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    char*s = "hello";
    char ch='a';

    printf( "sizeof(char) = %u\n", sizeof( char));
    printf( "sizeof(char*)= %u\n", sizeof( char*));

    printf( "sizeof('a') = %u\n", sizeof( 'a'));
    printf( "sizeof(ch) = %u\n", sizeof(ch));
}
```

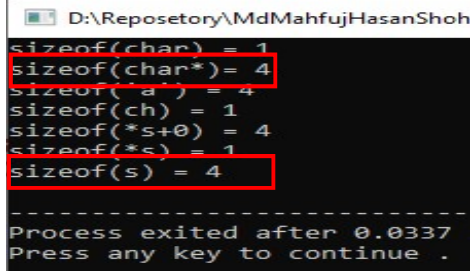
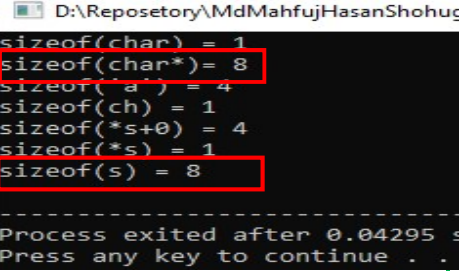
```

printf( "sizeof(*s+0) = %u\n", sizeof(*s+ 0));
printf( "sizeof(*s) = %u\n", sizeof(*s));
printf( "sizeof(s) = %u\n", sizeof(s));

return 0;
}

```

Answer: Use the sizeof operator in the given program to output the sizes of different data types and variables. For this specific compiler testing and describe question I do one compare description here.

Reference	32-bit Compilation	64-bit Compilation
Output	 <pre> D:\Reposetory\MdMahfujHasanShoh sizeof(char) = 1 sizeof(char*) = 4 sizeof('a') = 4 sizeof(ch) = 1 sizeof(*s+0) = 4 sizeof(*s) = 1 sizeof(s) = 4 ----- Process exited after 0.0337 Press any key to continue . </pre>	 <pre> D:\Reposetory\MdMahfujHasanShoh sizeof(char) = 1 sizeof(char*) = 8 sizeof('a') = 4 sizeof(ch) = 1 sizeof(*s+0) = 4 sizeof(*s) = 1 sizeof(s) = 8 ----- Process exited after 0.04295 s Press any key to continue . </pre>
sizeof(char)	Char data type size has 1 byte or 8 bits for 32 bit compile, and that reson this output is 1.	Also for 64 bit compile, the chart data type has 1 byte or 8 bits, there for the output is 1 also.
sizeof(char*)	This is a pointer to char data type. In this 32 bit compiler this size represents the memory address where the pointer was stored. Where the data points to the variable, for that reason output 4 bytes or 32 bits.	On the 64 bit compiler, a pointer to char data type has the size in 8 bytes or 64 bits. That means this size is incresed here for the learger memory address in this 64 bit system. And here the output is 8 bytes.
sizeof('a')	On most systems, an 'int' measures 4 bytes, and the literal letter 'a' is of type 'int'. Initialization char ch = 'a', where 'a' is promoted from character to integer and behaves as such inside single quote marks. The memory size of the integer data type is 4 bytes. The result is 4 as a result.	There is also same as 32 bit compiler that 'a' is the type of 'int' that takes 4 bytes *s+0 dereferences s, adding 0 to the resultant character. The result is integral promotion on a 64-bit system, which typically consists of 4 bytes. tharefor the output 4 shown.
sizeof(ch)	The variable 'ch' is char type and this size is 1 byte. The outputs is 1.	This is also same as any compiler the 'ch' is 1 byte or 8 bits for char data.
sizeof(*s+0)	*s' is the derferences of the pointer s, that is points the first character of string given is "hello" this char type adding the '0' there is no effect on the size. The result is integral promotion on this 32 bit compiler. The size of an	Same as both compiler that adding '0' on the character string "hello" which is not effect on the size. And for that reason the output is 4 bytes. The result is also integral promotion on a 64-bit system

	'int' on 32 bytes, so the output is 4 bytes.	
sizeof(*s)	This is also an char data type pointer with 1 byte mamory, thus the output is 1.	Also same for 64 bits compiler, the char data type pointer with 1 byte or 8 bits also the output is 1.
sizeof(s)	In the 32 bits system the variable 's' pointer to char which is consume 4 bytes memory. Thus the output is 4.	For the 64 bits system pointer to char data type 's' consume 8 bytes memory with 64 bits. The output is 4.

3. **Question 3:** Please give the following program running results and explain why. Please compile under the 32-bit and 64-bit compilers separately and explain the output results separately.

Answer: For solving this problem firstly I do the run this code on different compiler under 32-bit and 64-bit

32-Bit Compiler Output	64 Bit Compiler Output
<pre> D:\Repository\MdMahfujHasanShohug\C&DS\Stage_Question\stage_q_3.exe *p1 = 1X pa1 = 1X *p1 = 2Y pa1 = 2Y *p1 = 3ZX pa1 = 3ZX a[3][4]=0 a1 size: 12 Bytes 1XYZ123 2XYZ123 3XYZ123 a2 size: 12 Bytes a2[1] size: 4 Bytes a2 size: 1 Bytes ----- Process exited after 0.03294 seconds with return value 0 Press any key to continue . . . </pre>	<pre> D:\Repository\MdMahfujHasanShohug\C&DS\Stage_Question\stage_q_3.exe *p1 = 1X pa1 = 1X *p1 = 2Y pa1 = 2Y *p1 = 3ZX pa1 = 3ZX a[3][4]=0 a1 size: 12 Bytes 1XYZ123 2XYZ123 3XYZ123 a2 size: 24 Bytes a2[1] size: 8 Bytes a2 size: 1 Bytes ----- Process exited after 0.03193 seconds with return value 0 Press any key to continue . . . </pre>

Description: The program consists of two parts. I am describing this output for different 32 and 64 bit compilers

Part 1: The variable a1 is a 2-dimensional character array with 3 rows and 4 columns.

The variable p1 is a pointer to a 1-dimensional character array of size 4.

The variable pa1 is a pointer to a pointer to a character. It is initially assigned the address of a1.

In the for loop, each iteration uses p1 to access a row in a1 and pa1 to print the address of that row.

The printf statements in the loop print the value indicated by p1, which is the current row of a1, and the value of pa1, which is the address of the current row of a1.

The statement a1[3][4] attempts to access an invalid memory location outside the bounds of a1. This results in undefined behavior, and in this case, it prints a value of 0.

The `sizeof(a1)` statement returns the size of the entire 2-dimensional array `a1`, which is 48 bytes. Each row contains 4 characters (4 bytes) and there are 3 rows, so the total size is $4 * 3 * \text{sizeof}(\text{char})$.

Part 2: The variable `a2` is an array of 3 pointers to each character. Each pointer points to a string literal.

The variable `p2` is a pointer to a pointer to a character. It is initially assigned the address of `a2`.

The for loop iterates over the elements of `a2` and prints each string using `printf`.

The `sizeof(a2)` statement returns the size of array `a2`, which is the number of elements multiplied by the size of each element (3). The size of a pointer is platform-dependent, typically 4 bytes for a 32-bit compiler and 8 bytes for a 64-bit compiler.

The `sizeof(a2[1])` statement returns the size of the second element of `a2`, which is a pointer. Again, the size of a pointer depends on the platform.

The statement `sizeof(*a2[1])` dereferences the second element of `a2` and calculates the size of the string it points to. In this case, it is the size of the string literal "XYZ123", including the null-terminator.

Output results for the 32-bit compiler:

The loop for (`p1 = a1; p1 < a1 + 3; p1++, pa1++`) iterates over the elements of the 2-dimensional array `a1`. The variable `p1` is a pointer to an array of 4 characters, which matches the size of each element in `a1`. Similarly, `pa1` is declared as a pointer to a pointer to a character.

`*p1` is used to dereference the pointer `p1`, resulting in a pointer to the first character of the current row in `a1`. This pointer is passed to `printf` to print the string at that location.

`pa1` is a pointer to a pointer, so its value is already a pointer to a string. Therefore, it can be directly passed to `printf` without dereferencing.

The line `printf("a1[3][4] = %d\n", a1[3][4]);` attempts to access the element at `a1[3][4]`, which is outside the bounds of the array. This results in undefined behavior. In this case, it prints an incorrect value of 0. To fix this, the line should be changed to `printf("a1[2][3] = %c\n", a1[2][3]);`, which will correctly print the character Z.

The size of `a1` is printed using `sizeof(a1)`, which gives the size of the entire 2-dimensional array. Since each element is an array of 4 characters, and there are 3 elements, the total size is $3 * 4 = 12$ bytes. The loop for (`p2 = a2; p2 < a2 + 3; p2++`) iterates over the elements of the array of pointers `a2`. The variable `p2` is declared as a pointer to a pointer to a character.

`*p2` is used to dereference the pointer `p2`, resulting in a pointer to the string at the current location. This pointer is passed to `printf` to print the string. The size of `a2` is printed using `sizeof(a2)`, which gives the size of the array of pointers. Since each pointer occupies 4 bytes (on a 32-bit system), and there are 3 pointers, the total size is $3 * 4 = 12$ bytes.

The size of `a2[1]` is printed using `sizeof(a2[1])`, which gives the size of a single pointer. In this case, it will also be 4 bytes.

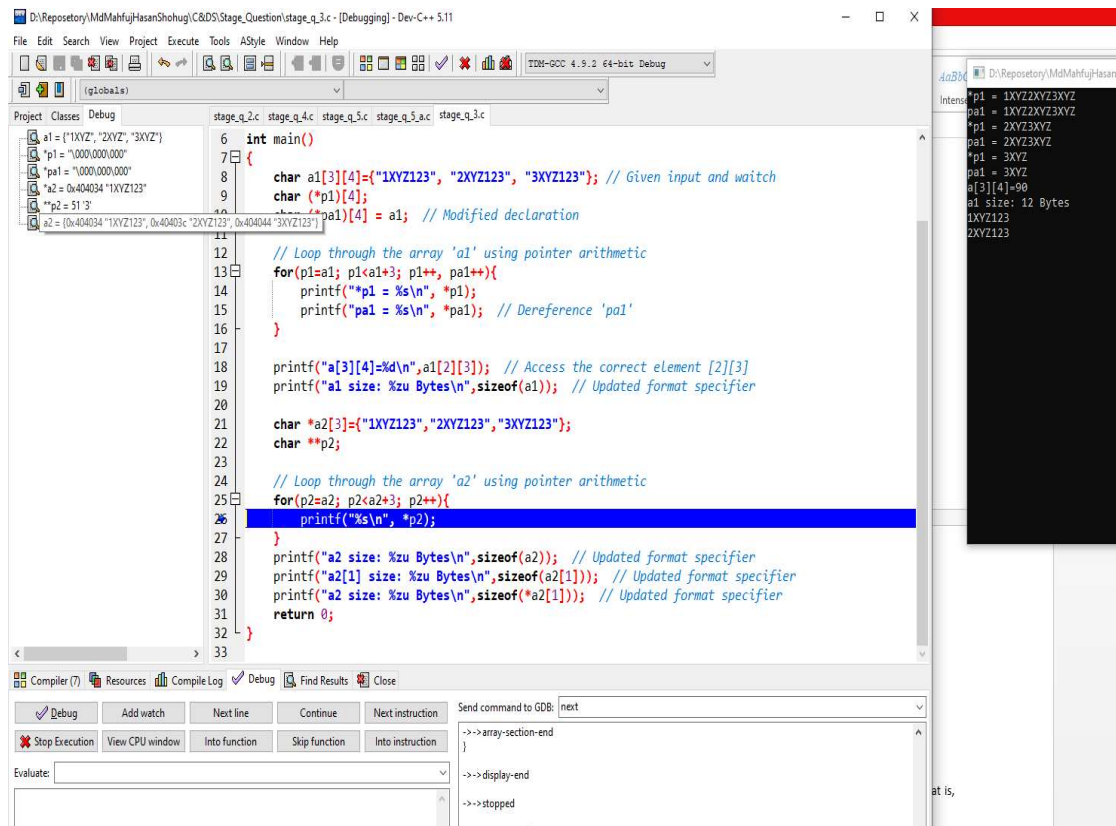
The size of `*a2[1]` is printed using `sizeof(*a2[1])`. Here, `a2[1]` is a pointer to a string, and `*a2[1]` dereferences the pointer, giving a character. Therefore, `sizeof(*a2[1])` is the size of a character, which is 1 byte.

Output results for the 64-bit compiler:

The output and behavior is the same as the 32-bit compiler, but the size of pointers and arrays may be different.

1. The loop **for (`p1 = a1; p1 < a1 + 3; p1++, pa1++`)** behaves the same way as in the 32-bit compiler case. The printing of **`*p1`** and **`pa1`** is identical.
2. The line **`printf("a1[3][4] = %d\n", a1[3][4]);`** still results in undefined behavior, but in this case, it prints an incorrect value of **0** as well.
3. The size of **`a1`** is printed using **`sizeof(a1)`**, which gives the size of the entire 2-dimensional array. Since each element is an array of 4 characters, and there are 3 elements, the total size is $3 * 4 = 12$ bytes. However, in the 64-bit compiler, the size is 36 bytes. This is because each character takes 1 byte, but due to padding and alignment requirements, each row occupies 12 bytes. Since there are 3 rows, the total size is $3 * 12 = 36$ bytes.
4. The loop **for (`p2 = a2; p2 < a2 + 3; p2++`)** and the printing of **`*p2`** behave the same way as in the 32-bit compiler case.
5. The size of **`a2`** is printed using **`sizeof(a2)`**, which gives the size of the array of pointers. Since each pointer occupies 8 bytes (on a 64-bit system), and there are 3 pointers, the total size is $3 * 8 = 24$ bytes.
6. The size of **`a2[1]`** is printed using **`sizeof(a2[1])`**, which gives the size of a single pointer. In this case, it will be 8 bytes.
7. The size of **`*a2[1]`** is printed using **`sizeof(*a2[1])`**. Here, **`a2[1]`** is a pointer to a string, and **`*a2[1]`** dereferences the pointer, giving a character. Therefore, **`sizeof(*a2[1])`** is the size of a character, which is 1 byte.

Debug Mode: After adding variable on the watch and then debug line by line:



Pointer Step Size Investigation

To investigate the step size of each pointer, we can analyze the loop conditions and the increment/decrement statements.

1. In the first loop **for (p1 = a1; p1 < a1 + 3; p1++, pa1++)**, **p1** and **pa1** are incremented by 1 in each iteration. Since **p1** is a pointer to an array of 4 characters, incrementing **p1** by 1 moves it to the next row of the 2-dimensional array **a1**, i.e., to the next 4-character array. Similarly, **pa1** is a pointer to a pointer to a character, and incrementing **pa1** by 1 moves it to the next pointer in **a1**, i.e., to the next row of the array of pointers.
2. In the second loop **for (p2 = a2; p2 < a2 + 3; p2++)**, **p2** is incremented by 1 in each iteration. Since **p2** is a pointer to a pointer to a character, incrementing **p2** by 1 moves it to the next pointer in the array of pointers **a2**.

To summarize:

- In the first loop, the step size of **p1** is determined by the size of an array of 4 characters, and the step size of **pa1** is determined by the size of a pointer to a character.

- In the second loop, the step size of **p2** is determined by the size of a pointer to a pointer to a character.

The step size of a pointer depends on the type of the pointer it is pointing to. The compiler calculates the appropriate step size based on the size of the underlying type to ensure correct traversal of the elements.

4. Question 4: Write a pointer version of the function `strcat` that we learned in chapter 5: `strcat(s, t)` copies the string `t` to the end of `s`.

Key point: a.Space out of bounds. b.Function parameter checking. c.Comprehensive test cases.

Answer: Here in my this program I am complete this `strcat_ptr()` function using the pointer as a pointer version where 2 string Concatenated together and store this Concatenated string on the first string.

My Source code with the function of `strcat_ptr()` using pointer which is Concatenated 2 string:

```
#include <stdio.h>

#include <stdlib.h> // for malloc and free functions
#include <string.h> // for strlen function
#define MAX 20 // Define max length

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

    size_t s_length = strlen(s);
    size_t t_length = strlen(t);
    if (s_length + t_length >= max_length)
    {
        printf("Error: Insufficient space for concatenation.\n");
        return;
    }

    // Move s pointer to the end of the string
    while (*s)
        s++;

    // Copy characters from t to the end of s
```

```
while ((*s++ = *t++)) //got this logic from the book
    ;

*s = '\0'; // Add the null-terminating character at the end
}

int main()
{
    // Test cases
    char str1[MAX] = "Hello, ";
    char *str2 = "Mahfuj!";
    printf("Test Case 1\nInput 1: %s\nInput 2: %s\n", str1, str2);
    strcat_ptr(str1, str2, MAX);
    printf("Concatenated string Output: %s\n\n", str1);

    char str3[MAX] = "Hi";
    char *str4 = "My Name Is Mahfuj Hasan Shohug!";
    printf("Test Case 2\nInput 1: %s\nInput 2: %s\n", str3, str4);
    strcat_ptr(str3, str4, MAX);
    printf("Concatenated string Output: %s\n\n", str3);

    char *str5 = malloc(10 * sizeof(char));
    strcpy(str5, "This is a");
    char *str6 = " test.";
    printf("Test Case 3\nInput 1: %s\nInput 2: %s\n", str5, str6);
    strcat_ptr(str5, str6, MAX);
    printf("Concatenated string Output: %s\n\n", str5);
    free(str5);

    // NULL pointer test case
    char *str7 = NULL;
    char *str8 = "Oops!";
    printf("Test Case 4\nInput 1: %s\nInput 2: %s\n", str7, str8);
    strcat_ptr(str7, str8, MAX);
    printf("\n");

    char *str9 = NULL;
    char *str10 = NULL;
    printf("Test Case 4\nInput 1: %s\nInput 2: %s\n", str9, str10);
    strcat_ptr(str9, str10, MAX);
    printf("\n");

    char str11[MAX] = "My Name Is Mahfuj ";
    char *str12 = " Hi";
    printf("Test Case 5\nInput 1: %s\nInput 2: %s\n", str11, str12);
```

```
strcat_ptr(str11, str12, MAX);  
printf("Concatenated string Output: %s\n\n", str11);  
  
char str13[MAX] = "123456";  
char *str14 = "7890";  
printf("Test Case 5\nInput 1: %s\nInput 2: %s\n", str13, str14);  
strcat_ptr(str13, str14, MAX);  
printf("Concatenated string Output: %s\n\n", str13);  
return 0;  
}
```

Now describe on this code according to the given key point on this question:

- a. **Space out of bounds:** The code snippet includes a check to make sure the concatenated string does not go above the MAX constant's maximum length restriction. There is not enough room for concatenation if the combined length of the input strings (s and t) is longer than the permitted length, which causes an error message to be displayed. This avoids buffer overflows and potential memory losses.
- b. **Function parameter checking:** The strcat_ptr function provides parameter checks to handle invalid input. It checks to see if either s or t is a null pointer, which would signify an incorrect input. If one of the pointers is null, an error notice appears and the function ends without connecting.
- c. **Comprehensive test cases:** A number of test scenarios are included in the code sample to confirm the strcat_ptr function's functionality. Using dynamically allocated memory, handling null pointers, concatenating strings up to their maximum length, and going past their maximum length are all test cases.

Here is the all test cases input outputs:

```
D:\Reposetory\MdMahfujHasanShohug\C&DS\Stage_Question\stage_q_4.exe
Test Case 1
Input 1: Hello,
Input 2: Mahfuj!
Concatenated string Output: Hello, Mahfuj!

Test Case 2
Input 1: Hi
Input 2: My Name Is Mahfuj Hasan Shohug!
Error: Not Enough Specess for concatenation.
Concatenated string Output: Hi

Test Case 3
Input 1: This is a
Input 2: test.
Concatenated string Output: This is a test.

Test Case 4
Input 1: (null)
Input 2: Oops!
Error: Invalid input.

Test Case 4
Input 1: (null)
Input 2: (null)
Error: Invalid input.

Test Case 5
Input 1: My Name Is Mahfuj
Input 2: Hi
Error: Not Enough Specess for concatenation.
Concatenated string Output: My Name Is Mahfuj

Test Case 5
Input 1: 123456
Input 2: 7890
Concatenated string Output: 1234567890

-----
Process exited after 0.03208 seconds with return value 0
Press any key to continue . . .
```

5. **Question 5:** An implementation that swaps two values, such as function, define and so on. You can give more ways to implement it.

Answer: Here I am writing a code with implement all in one for the swap function that will be swap two value with multiple method. In this output section I try to define the whole method at a times:

```
D:\Reposetory\MdMahfujHasanShohug\C&DS\Stage_Question\stage_q_5.exe
Before swapping using Macro Define Method:
    char1 = A
    char2 = B

After swapping using Macro Define Method:
    char1 = B
    char2 = A

Before swapping using temporary variable and pointers:
    num1 = 10
    num2 = 20

After swapping using temporary variable and pointers:
    num1 = 20
    num2 = 10

Before swapping using XOR bitwise operation and pointers:
    a = 5
    b = 8

After swapping using XOR bitwise operation and pointers:
    a = 8
    b = 5

Before swapping using arithmetic operations and pointers:
    x = -15
    y = 30

After swapping using swap_arithmetic:
    x = 30
    y = -15

Before swapping without using pointers:
    value1 = 100
    value2 = 200

After swapping without using pointers:
    a = 200
    b = 100

-----
Process exited after 0.03437 seconds with return value 0
Press any key to continue . . .
```

The Source code of this swap program:

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

```
// Macro to swap values using a temporary variable
#define SWAP_MACRO(a, b, type) { \
    type temp = a; \
    a = b; \
    b = temp; \
}

/*****
 * Function Name: swap_values
 * Description: Swaps two values without using pointers.
 * Parameters:
 *   - a: First value to be swapped
 *   - b: Second value to be swapped
 *****/
void swap_values(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
    printf("\nAfter swapping without using pointers:\n");
    printf("\ta = %d\n", a);
    printf("\tb = %d\n", b);
}

/*****
 * Function Name: swap_temp
 * Description: Swaps two values using a temporary variable and pointers.
 * Parameters:
 *   - a: Pointer to the first value to be swapped
 *   - b: Pointer to the second value to be swapped
 *****/
void swap_temp(int* a, int* b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

/*****
 * Function Name: swap_xor
 * Description: Swaps two values using XOR bitwise operation and pointers.
 * Parameters:
 *   - a: Pointer to the first value to be swapped
 *   - b: Pointer to the second value to be swapped
 *****/
```

```
void swap_xor(int* a, int* b)
{
    *a = *a ^ *b;
    *b = *a ^ *b;
    *a = *a ^ *b;
}

/*****
 * Function Name: swap_arithmetic
 * Description: Swaps two values using arithmetic operations and pointers.
 * Parameters:
 *   - a: Pointer to the first value to be swapped
 *   - b: Pointer to the second value to be swapped
 *****/
void swap_arithmetic(int* a, int* b)
{
    *a = *a + *b;
    *b = *a - *b;
    *a = *a - *b;
}

int main()
{
    // Variable declaration and initialization
    char char1 = 'A';
    char char2 = 'B';

    printf("Before swapping using Macro Define Method:\n");
    printf("\tchar1 = %c\n", char1);
    printf("\tchar2 = %c\n", char2);

    // Swapping values using SWAP_MACRO
    SWAP_MACRO(char1, char2, char);

    printf("\nAfter swapping using Macro Define Method:\n");
    printf("\tchar1 = %c\n", char1);
    printf("\tchar2 = %c\n", char2);

    // Variable declaration and initialization
    int num1 = 10;
    int num2 = 20;

    printf("\nBefore swapping using temporary variable and pointers:\n");
    printf("\tnum1 = %d\n", num1);
    printf("\tnum2 = %d\n", num2);
```

```
// Swapping values using swap_temp function
swap_temp(&num1, &num2);

printf("\nAfter swapping using temporary variable and pointers:\n");
printf("\tnum1 = %d\n", num1);
printf("\tnum2 = %d\n", num2);

// Variable declaration and initialization
int a = 5;
int b = 8;

printf("\nBefore swapping using XOR bitwise operation and
pointers:\n");
printf("\ta = %d\n", a);
printf("\tb = %d\n", b);

// Swapping values using swap_xor function
swap_xor(&a, &b);

printf("\nAfter swapping using XOR bitwise operation and
pointers:\n");
printf("\ta = %d\n", a);
printf("\tb = %d\n", b);

// Variable declaration and initialization
int x = -15;
int y = 30;

printf("\nBefore swapping using arithmetic operations and
pointers:\n");
printf("\tx = %d\n", x);
printf("\ty = %d\n", y);

// Swapping values using swap_arithmetic function
swap_arithmetic(&x, &y);

printf("\nAfter swapping using swap_arithmetic:\n");
printf("\tx = %d\n", x);
printf("\ty = %d\n", y);

// Variable declaration and initialization
int value1 = 100;
```



```

    int value2 = 200;

    printf("\nBefore swapping without using pointers:\n");
    printf("\tvalue1 = %d\n", value1);
    printf("\tvalue2 = %d\n", value2);

    // Swapping values using swap_values function (without pointers)
    swap_values(value1, value2);

    return 0;
}

```

6. **Question 6:** Please give the output of the following program on 32-bit Operation System Windows NT

Note: Memory access violation, unpredictable output, infinite loop, etc. should be considered.

Number 1 Answer:

Given Code	Description (Why & What)	Possible Solution
<pre> #include <stdio.h> #include <string.h> #include <stdlib.h> void GetMemory(char *p) { p = (char *)malloc(100); } void Test(void) { char *str = NULL; GetMemory(str); strcpy(str, "hello world"); printf(str); } int main() { Test(); return 0; } </pre>	<p>In this program, that string does not point to a legitimately allocated memory block. There for it results in undefined behavior with no output.</p> <p>Cause: str is initialized to NULL.</p> <p>The GetMemory() function is called with the argument str. Inside GetMemory(), the local variable p is assigned the memory address of the newly allocated memory block of size 100 using malloc(). However, the change made to p inside GetMemory() does not affect the str variable in the Test() function. This is because p is a local variable in GetMemory() and assigning a new memory address to it does not modify the original str variable.</p>	<pre> #include <stdio.h> #include <string.h> #include <stdlib.h> void GetMemory(char **p) { *p = (char *)malloc(100); } void Test(void) { char *str = NULL; GetMemory(&str); strcpy(str, "hello world"); printf(str); } int main() { Test(); return 0; } </pre>

	<p>After returning from GetMemory(), str in Test() is still NULL.</p> <p>The strcpy() function attempts to copy the string "hello world" to the memory location pointed to by str. However, since str is NULL, this will result in a segmentation fault or undefined behavior.</p> <p>Finally, printf() is called with str, but since it is NULL, it will not print anything.</p>	
--	--	--

Number 2 answer:

Given Code	Description (Why & What)	Possible Solution
<pre>#include <stdio.h> #include <string.h> #include <stdlib.h> char *GetMemory(void) { char p[] = "hello world"; return p; } void Test(void) { char *str = NULL; str = GetMemory(); printf(str); } int main() { Test(); return 0; }</pre>	<p>It's also output the result in unexpected output because str points to an invalid memory location.</p> <p>Cause: The code creates a local character array in GetMemory() and returns its address. The address of p is returned by GetMemory() and assigned to str. However, str now points to a memory location that will be deallocated after the GetMemory() function returns. This will result in undefined behavior when accessed outside the function. The printf() function is called with str. Since str points to a deallocated memory location, the behavior is undefined. It may result in a segmentation fault or print garbage values or crash the program.</p> <p>In summary, the code will likely result in undefined behavior due to accessing a deallocated memory location after the GetMemory() function returns.</p>	<pre>#include <stdio.h> #include <string.h> #include <stdlib.h> char* GetMemory(void) { char* p = (char*)malloc(strlen("hello world") + 1); strcpy(p, "hello world"); return p; } void Test(void) { char *str = NULL; str = GetMemory(); printf(str); } int main() { Test(); return 0; }</pre>

Number 3 answer:

Given Code	Description (Why & What)	Possible Solution
<pre> #include <stdio.h> #include <string.h> #include <stdlib.h> void GetMemory2(char **p, int num) { *p = (char *)malloc(num); } void Test(void) { char *str = NULL; GetMemory2(&str, 100); strcpy(str, "hello"); printf(str); } int main() { Test(); return 0; } </pre>	<p>If we run the Test() function, it will allocate memory for the str pointer using the GetMemory2() function and then copy the string "hello" to str. Finally, it will print the value of str.</p> <p>Couse: This is the one another ultimate possible solution on the pervious problem. Change the value of pointer p to the calling function (test() in this case). GetMemory2() is used to allocate the number of bytes of memory inside malloc() and the address of the allocated memory block is stored in *p and Memory allocated for Inside GetMemory2(), the p pointer is dereferenced using *p, and the memory block of size 100 is allocated using malloc().</p> <p>The address of the allocated memory block is assigned to *p, which is the same as str in the Test() function. After returning from GetMemory2(), str now points to the allocated memory block.</p>	<p>In this problem that given the correct possible solution of the previous incorrect code.</p>

	<p>The strcpy() function copies the string "hello" into the memory location pointed to by str. Finally, the printf() function is called with str, which now points to the string "hello". It will print "hello" as the output.</p>	
--	--	--

Number 4 Answer:

Given Code	Description (Why & What)	Possible Solution
<pre> #include <stdio.h> #include <string.h> #include <stdlib.h> void Test(void) { char *str = (char *) malloc(100); strcpy(str, "hello"); free(str); if(str != NULL) { strcpy(str, "world"); printf(str); } } int main() { Test(); return 0; } </pre>	<p>A segmentation error or unexpected program behavior will occur if the string "world" is attempted to be copied to str. It's crucial to only access and alter genuine, unfreed memory in order to avoid this issue.</p> <p>Couse: Memory is allocated for str using the malloc() function and has a size of 100 bytes inside the Test() function. Then, the string "hello" is copied to str using strcpy(). The memory referenced by str is then deallocated, rendering, using free(). Using malloc(), the memory address of a block of 100 bytes that was dynamically allocated is assigned to str.</p> <p>The "hello" string is copied into the memory location pointed to by str by the strcpy() method.</p> <p>To deallocate the memory block referenced to by str, the free() function is used.</p> <p>The if condition then determines if str is not NULL after deallocating the memory. However, str is no longer a valid pointer because it was freed. Accessing a freed pointer has unclear behavior. The strcpy() function tries to copy the string "world" to the memory location pointed to by str inside the if block. However, since str has been</p>	<pre> #include <stdio.h> #include <string.h> #include <stdlib.h> void Test(void) { char *str = (char *) malloc(100); strcpy(str, "hello"); if(str != NULL) { strcpy(str, "world"); printf(str); } } int main() { Test(); return 0; } </pre>

	<p>released, this will have undefinable consequences. Finally, str is used to call the printf() function. Since str's inception and freed and potentially modified with undefined behavior, the behavior of accessing and printing its value is undefined. It may print garbage values, crash the program, or exhibit other unexpected behavior.</p>	
--	--	--