#Solution-1:

The output of the program will be 20.

Explanation by pointer box:

At the beginning of the main function, an integer variable r is declared and initialized with the value 20. Then, a pointer p is declared and initialized to point to the memory location of r.

When the fun function is called with the argument p, a new integer variable q is declared and initialized with the value 10. Then, the pointer p is assigned to point to the memory location of q, but this change only affects the local copy of p inside the fun function, and not the original p pointer in the main function.

After the fun function returns, the printf statement in the main function dereferences the pointer p and prints the value at its memory location, which is still the original value of r, i.e., 20.

So, the pointer box for this program would look like this:

main:

+------+------+ fun:

| r | 20 | +------+

| p---|----->|---+ | q |

+------+------+ | | 10 |

| +------+

|

+--->p (local copy)

The memory locations shown in the pointer box are for illustrative purposes only and may differ on different systems.

#Solution-2:

The output of the program will be 10.

Explanation by pointer box:

At the beginning of the main function, an integer variable r is declared and initialized with the value 20. Then, a pointer p is declared and initialized to point to the memory location of r.

When the fun function is called with the address of p, a pointer to a pointer pptr is declared and initialized to point to the address of p. A new static integer variable q is declared and initialized with the value 10. Then, the pointer to pointer pptr is dereferenced to point to the memory location of q, which means that the p pointer in the main function now points to the memory location of q.

After the fun function returns, the printf statement in the main function dereferences the p pointer and prints the value at its memory location, which is the value of q, i.e., 10.

So, the pointer box for this program would look like this:

main:

+------+------+ fun:

| r | 20 | +------+

| p---|----->|------>| q |

+------+------+ | 10 |

+------+

^

|

+----|----+

| pptr |

+--------+

#Solution-3:

The output of the program will be 2 5.

Explanation:

+------+------+------+------+------+

a: | 1 | 2 | 3 | 4 | 5 |

+------+------+------+------+------+

^

|

&a

+------+------+------+------+------+

ptr:| &a[5] |

+------+------+------+------+------+

^

|

ptr

In the diagram, the array a is stored as a contiguous block of memory that holds 5 integers. The ptr variable is a pointer to an integer that is initialized to the address of one past the end of the array, i.e., the memory location immediately after the last element in the array.

When the expression \*(ptr-1) is evaluated, it points to the last element in the array a, which has the value 5.

When the expression \*(a+1) is evaluated, it points to the second element in the array a, which has the value 2.

#Solution-4:

The running result is - TEST sQuiz Z CQ

+---+ +---+---+---+---+

c: | | ---> | G | e | k | s | "GeksQuiz"

+---+ +---+---+---+---+

| M | C | Q | | "MCQ"

+---+---+---+---+

| T | E | S | T | "TEST"

+---+---+---+---+

| Q | U | I | Z | "QUIZ"

+------+------+------+------+

cp: | c+3 | c+2 | c+1 | c |

+------+------+------+------+

+------+ +------+------+------+------+

cpp: +->| cp[0]|\ | c+3 | c+2 | c+1 | c |

| +------+ +-------+ +------+------+------+------+

| | cp[1]|\ | | G | M | T | Q |

| +------+ +--+ | +------+------+------+------+

| | cp[2]| | | | e | C | E | U |

| +------+ | | +------+------+------+------+

| | cp[3]| +----+ | k | Q | S | I |

| +------+ | +------+------+------+------+

+------------------+

At the start of the program, c contains four string literals, each with a pointer to its first character. The cp array contains four pointers to pointers, pointing to the elements of c in reverse order. The cpp pointer is initialized to point to the first element of cp.

As the program executes, cpp is incremented and decremented to point to different elements of cp, and the \* and [] operators are used to dereference pointers and access characters of strings.

The visualization shows the memory locations of the variables and the values they point to. The arrows indicate pointers, and the boxes represent arrays and string literals.

#Solution-5:

The output is- GeeksQuiz

+------+ +-----+

str: +->| | | |

| | | |

| | | |

| | | |

| | | |

| | | |

| | | |

| | | |

| | | |

| | | |

+------+ +-----+

+------+

str\_ref: | |

+------+

At the start of the program, str is a pointer to the first element of a dynamically allocated block of memory containing 100 characters, with the string "GeeksQuiz" stored in it. str\_ref is a pointer to a pointer to a character, initially pointing to the same memory address as str.

When fun is called with &str, the address of str is passed by reference, so str\_ref is now pointing to the same memory address as str. When str\_ref is incremented in fun, it now points to the next memory address after str, but this has no effect on the original value of str in main.

After fun returns, puts is called with str as an argument, which prints the string "GeeksQuiz" to the console.

Finally, the memory allocated for str is freed using free.

#Solution-6:

The output is : gh

The main function declares an array of six string literals called argv. The f function is then called with argv as an argument.

Inside f, the pointer p is set to the address of argv[0], which is a pointer to the first string literal in the array. Then, p is incremented by the size of an integer, which in this case is 4 bytes (assuming a 32-bit architecture).

After this, the expression (p += sizeof(int))[-1] is evaluated. This is equivalent to \*(p - 1) which is equivalent to argv[3]. This value is the pointer to the string literal "gh".

Finally, this pointer value is passed to printf as the argument to the %s format specifier, causing the string "gh" to be printed to the console.

| Address | Value |

|-----------|-----------|

| 0x1000 | "ab" |

| 0x1004 | "cd" |

| 0x1008 | "ef" |

| 0x100C | "gh" |

| 0x1010 | "ij" |

| 0x1014 | "kl" |

p = &argv[0] (p points to "ab")

p += sizeof(int) (p now points to "gh")

t = (p -= sizeof(int))[-1] (t points to "gh")

#Solution-7:

The output is : 2 3 5 6

This program declares a two-dimensional array a with 2 rows and 3 columns, initializes it with some values, and declares a pointer to an array of 3 integers ptr that points to the first row of a. It then prints the second and third elements of the first row of a, increments ptr to point to the second row of a, and prints the second and third elements of the second row of a.

+-------+ +------+------+------+

| a | ----------> | 1 | 2 | 3 |

+-------+ +------+------+------+

| | | | 4 | 5 | 6 |

| v | +------+------+------+

+-------+ int a[][3] = {1, 2, 3, 4, 5, 6};

+-------+

| ptr | ----------> +-------+

+-------+ | a |

+-------+

| | |

| v |

+-------+

int (\*ptr)[3] = a;

+-------------+

| printf(...) | (\*ptr)[1] is 2 and (\*ptr)[2] is 3

+-------------+ printf("%d %d ", (\*ptr)[1], (\*ptr)[2]);

+-------+

| ptr++ | ptr now points to the second row

+-------+

+-------------+

| printf(...) | (\*ptr)[1] is 5 and (\*ptr)[2] is 6

+-------------+ printf("%d %d\n", (\*ptr)[1], (\*ptr)[2]);

#Solution-8:

The output is :

0 1 2 2 3

+----+----+----+----+----+

ptr | | | | | |

+----+----+----+----+----+

| | | | |

v v v v v

+----+----+----+----+----+

| 0 | 1 | 2 | 3 | 4 |

+----+----+----+----+----+

After the memory is allocated and initialized with values, the pointer ptr points to the start of the allocated memory block.

printf("%d ", \*ptr++); - Dereferences the current value of ptr (which is 0) and prints it, then increments ptr to point to the next integer in the block.

printf("%d ", (\*ptr)++); - Dereferences the current value of ptr (which is now 1) and prints it, then increments the value at that memory location (now 2).

printf("%d ", \*ptr); - Dereferences the current value of ptr (which is still 1, since it was only incremented in the previous step) and prints it.

printf("%d ", \*++ptr); - Increments ptr to point to the next integer in the block (which has the value 3), dereferences it and prints it.

printf("%d ", ++\*ptr); - Dereferences the current value of ptr (which is 2 after the second step) and increments it (now 3), then prints the new value.

#Solution-9:

The output : 20 10

The integer array 'arr' in this program is declared with two items, and its initial values are 10 and 20. Then, we execute the method "fun()" and supply an address to the array's first element. In the 'fun()' function, we utilize pointer arithmetic to add 1 to the pointer 'arr', making it point to the second element of 'arr'. The second element of 'arr', which has the value 20 is then printed. We print the value of the first element of 'arr', which is still 10, once more in the'main()' function before returning 0 to end the program.

+----+----+

| 10 | 20 |

+----+----+

arr arr

|

v

+----+

| 20 |

+----+

10

Question-2:

#Solution-1:

The output is : Value of a is 10.

The program defines a function fun that takes an integer parameter and prints its value. Then, it defines a function pointer variable fun\_ptr that points to fun. The program calls the function fun using the function pointer fun\_ptr and passes an integer value of 10 as its argument.

Here is the visualization of the memory representation of the program:

+-----------------+

| Code of fun() |

+-----------------+

| printf() code |

+-----------------+

| a |

+-----------------+

+-----------------+

| Code of main() |

+-----------------+

| fun\_ptr |

+-----------------+

| 10 |

+-----------------+

Initially, the code of fun() is stored in memory, followed by the code for printf() and the memory location for the parameter a. In the main() function, the code for main() is stored in memory, followed by the memory location for the function pointer fun\_ptr. When the program executes, it assigns the memory location of fun() to fun\_ptr and then calls fun() using the function pointer. The integer value 10 is stored in memory and passed as an argument to fun(). The value of a is printed as 10.

#Solution-2

The output is : Enter Choice: 0 for add, 1 for subtract and 2 for multiply 1

+---------------+

| |

| fun\_ptr\_arr |

| |

+---------------+

|

v

+----+ +----+ +----+

| | | | | |

|add |--|sub |--|mul |

| | | | | |

+----+ +----+ +----+

| | |

v v v

+----+ +----+ +----+

| a1 | | a2 | | a3 |

+----+ +----+ +----+

| b1 | | b2 | | b3 |

+----+ +----+ +----+

Each of the function pointers fun\_ptr\_arr[0], fun\_ptr\_arr[1], and fun\_ptr\_arr[2] points to the corresponding functions add(), subtract(), and multiply(). When a choice is made by the user, the corresponding function pointer is dereferenced and invoked with the values of a and b passed as arguments. In the above visualization, a1, a2, and a3 represent the values of a passed to the respective functions, and b1, b2, and b3 represent the values of b passed to the respective functions.

#Solution-3:

The output is : Fun1, Fun2

+------+ +-------+ +-------+

| main | ---> |wrapper| ---> | fun1 |

+------+ +-------+ +-------+

^

|

| +-------+

+-----> | fun2 |

+-------+

In this visualization, there are three boxes representing functions: main, wrapper, and either fun1 or fun2 depending on which function is being called. The arrows indicate the flow of control: main calls wrapper, which in turn calls either fun1 or fun2.

#Solution-4:

The output is : 5 10 12 15 80 90

+----+----+----+----+-----+-----+

arr | 10 | 5 | 15 | 12 | 90 | 80 |

+----+----+----+----+-----+-----+

+----+----+----+----+-----+-----+

arr | 5 | 10 | 12 | 15 | 80 | 90 |

+----+----+----+----+-----+-----+

The compare function is used as a parameter for qsort, which takes a pointer to the first element of the array, the number of elements in the array, the size of each element, and the comparison function. The compare function is implemented to compare two integers and return a negative number if the first integer is less than the second, a positive number if the first integer is greater than the second, and zero if the two integers are equal.

After the qsort function is called, the sorted array is printed using a loop to iterate over each element.

#Solution-5:

The output is: Return index is 2.

arr pointer | | --> +-----+

| arr | | 2 |

+-----+ +-----+

| 5 |

+-----+

| 7 |

+-----+

| 90 |

+-----+

| 70 |

+-----+

x value -> +-----+

| | --> | 7 |

+-----+

This is a visualization of an integer array arr and an integer variable x using a memory diagram. The array arr has five elements {2, 5, 7, 90, 70} and the variable x has a value of 7.

The pointer arr points to the first element of the array which is 2. Each element of the array occupies a memory space of the size of an integer (usually 4 bytes) and is stored in contiguous memory locations.

The pointer x points to another memory location where the value 7 is stored. This value can be compared with the elements of the array using the compare function in the search function.

#Question-3:

#Solution-1:

The output is : \*ptr1=10

\*ptr2 =10

+------+

| i=10|

+------+

^

|

+-----+

| ptr1|

+-----+

|

v

+------+

|&i=?? |

+------+

^

|

+------+

| ptr2 |

+------+

|

v

+------+

|&i=?? |

+------+

This program defines a function returnPointer that takes an integer pointer as an argument and returns the same pointer. In the main function, an integer i is declared and initialized to 10. Two integer pointers ptr1 and ptr2 are declared and ptr1 is assigned the address of i. ptr2 is assigned the value returned by returnPointer function when it is called with the address of i as an argument. Finally, the values pointed to by ptr1 and ptr2 are printed. When returnPointer is called with the address of i as an argument, it simply returns the same address. Therefore, ptr2 also points to i. In the main function, ptr1 and ptr2 are pointing to the same integer variable i. When the values pointed to by ptr1 and ptr2 are printed, they both have the value 10.

#Solution-2:

An integer pointer is returned by the program's returnFunc function, which is named after it. The & operator is used to get the location of the local variable i, which is created with a value of 10. The address is then returned as a pointer.

The value returned by returnFunc is assigned to the pointer ptr, which is declared in the main function. In the returnFunc function's local variable i, this pointer points to the memory location of that variable.

The value pointed to by ptr is then printed three times by the program using the%d format directive. Since the program is attempting to access a memory address that has already been deallocated by the time the returnFunc function returns, the program's behavior is, however, undefinable.

Visualization:-

+------+

return -> | | -> +----+

| i=10| | 10 |

+------+ +----+

Here, the returnFunc() function returns the address of a local variable i. As soon as the function returns, the local variable i goes out of scope and is destroyed. Therefore, the pointer returned by returnFunc() becomes invalid and dereferencing it results in undefined behavior.

#Solution-3:

The output is;

\*ptr = 10

\*ptr = 10

\*ptr = 10

The code declares a function named returnFunc that returns a pointer to an integer. The main function calls returnFunc and assigns the returned pointer to ptr. Then, it prints the value pointed by ptr three times using printf function.

The difference between this code and the previous one is that i is declared as static int i=10; inside the returnFunc function. This makes the variable i persist throughout the lifetime of the program, even after the function has returned.

+-------------+

| main |

|-------------|

| ptr |

+-------------+

|

|

v

+-------------+

| returnFunc |

|-------------|

| i = 10 |

+-------------+

^

|

|

+----------------------+

| Static Memory |

|----------------------|

| |

| i = 10 (persisted) |

| |

+----------------------+

After the function returns, i still exists in memory because it is stored in a static memory location. The pointer ptr points to this static memory location, and thus the value of \*ptr is still valid even after the function returnFunc has returned.