Question no 1

Program 1

* The running result for this code is 20

here is the pointer box with an explanation:

[ &q ] [ q=10 ] => [ \*p ] [ r=20 ]

[ p ] => [ &q ]

[ q ]

* In this case, the fun() function takes a pointer to an integer p as an argument. Inside fun(), a new integer q is declared and assigned a value of 10. The pointer variable p is then assigned the address of q. However, this only modifies the local variable p inside the fun() function, and does not affect the pointer variable p in main().

When the fun() function returns, p in main() still points to r, and the value of \*p is 20, which is the value of r. Therefore, the output of the printf() statement is 20.

Program 2

* The running result for this code is 10.

here is the pointer box with an explanation

[ &q ] [ q=10 ] => [ &q ] [ q=10 ]

[ \*\*pptr ] => [ \*p ]

[ p ] => [ &r ]

[ &r ] [ r=20 ]

* n this code, fun() takes a pointer to a pointer pptr as an argument. Inside fun(), a static integer q is declared and assigned a value of 10. The dereferenced pointer \*pptr is then assigned the address of q, which changes the value of p in main() to point to the memory location of q.

After returning from the fun() function, p in main() points to the memory location of q, and the value of \*p is 10, which is the value of q. Therefore, the output of the printf() statement is 10.

Program 3

* the output for the above code is 2, 2
* here is the pointer box with its explanation
* =
* [ &a ] [ a[0]=1 ] [ a[1]=2 ] [ a[2]=3 ] [ a[3]=4 ] [ a[4]=5 ] [ &a+1 ] [Garbage ]

[ ptr ] => [ (int\*)(a+1) = 2 ]

* Explanation: The first bracket represents the memory location of array a, and the next five brackets represent the memory location of its elements with their respective values.

The seventh bracket represents the memory location of &a+1, which is a pointer to the memory location just after the end of the array. However, since this pointer points to an invalid memory location, the value pointed by ptr is garbage.

The second arrow shows the value of ptr, which is cast to an integer pointer pointing to the second element of the array. Therefore, \*(a+1) and \*(ptr-1) both point to the same memory location, which holds the value 2. Hence, the output of the printf statement is 2 2.

Program 4

* The output of the above code is TEST QUIZ EST IK
* Here is the explanation of the code by using the pointer box

[ c ] ["GeksQuiz"] ["MCQ"] ["TEST"] ["QUIZ"]

[ cp ] => [ c+3 ] [ c+2 ] [ c+1 ] [ c ]

|

V

[ cpp ]

This pointer box represents four related variables: c, cp, cpp, and the string literals "GeksQuiz", "MCQ", "TEST", and "QUIZ".

c is an array of character pointers initialized to point to the string literals "GeksQuiz", "MCQ", "TEST", and "QUIZ".

cp is an array of pointers to character pointers, initialized to point to the memory addresses of c elements in reverse order. Therefore, cp[0] points to c, cp[1] points to c+1, cp[2] points to c+2, and cp[3] points to c+3.

cpp is a pointer to a pointer to a character pointer. It is initialized to point to cp. Dereferencing cpp gives us the address of cp. Dereferencing \*cpp gives us the address of c, which is the first element of cp. Incrementing \*cpp moves it to point to the next element of cp. Therefore, \*\*++cpp points to "TEST", which is the value of c+1.

\*++cpp increments cpp to point to the next element of cp (c+2). Dereferencing \*++cpp gives us the value of cp[2], which is c+1. Adding 3 to \*--\*++cpp+3 gives us the address of the fourth character of the string "GeksQuiz", which is 's'.

cpp[-2] is equal to cp[1], which is c+2. Dereferencing cpp[-2] gives us the value of cp[1], which is c+2. Adding 3 to \*cpp[-2]+3 gives us the address of the fourth character of the string "TEST", which is 'T'.

cpp[-1][-1] gives us c+3-1, which is equal to c+2. Dereferencing cpp[-1][-1] gives us the value of \*(c+2), which is "TEST". Adding 1 to cpp[-1][-1]+1 gives us the address of the second character of the string "TEST", which is 'E'.

program 5

* The running result for this code is "GeeksQuiz".

[ str ] ["GeeksQuiz"]

[ str\_ref ] => [ str+1 ]

|

V

[ &str ]

* The program begins by declaring a pointer variable str which points to the string "GeeksQuiz" stored in memory. The function fun() is declared to take a pointer to a pointer to a char, str\_ref is initialized to the address of str. fun() is called with str\_ref as its argument, which increments the value of str\_ref to point to the next memory location (i.e., str+1), but the value of str remains unchanged.

The string "GeeksQuiz" is printed using the puts() function, which takes str as its argument. Since the value of str has not been changed, it still points to the beginning of the string "GeeksQuiz". Finally, the memory allocated to str using malloc() is freed using the free() function to prevent memory leaks. Overall, the program demonstrates the use of pointer to pointer concept, and dynamic memory allocation and deallocation.

Program 6

* The running result of the code is k1

= 5, b = 10

Add() subtract() multiply()

* Here is the pointer box of the above code with

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

argv ---> ---->"ab"----> "cd"---->| "ef"|---->| "gh"|---->| "ij"|---->| "kl"|

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

p --->

t --->

In the main function, an array of char pointers argv is defined and initialized with 6 string literals. These string literals are stored in read-only memory and the addresses of the first characters of these strings are assigned to the pointers in the argv array.

The f function takes a char pointer pointer p as its argument. In C, arrays can decay into pointers when passed to functions, so when f(argv) is called, argv decays into a pointer to its first element, which is a char pointer pointer.

In the f function, the p pointer is incremented by the size of an int (sizeof(int)), which advances the pointer to the 3rd element of the argv array, which is the string literal "ef". The expression (p += sizeof(int))[-1] accesses the element that is one position before the current position of p, which is "ef". This string is assigned to the char pointer t.

Finally, the printf function is called to print the string pointed to by t, which is "ef". The program returns 0 to indicate successful execution.

Program 7

* The out put of the above code is 2, 3,5,6.
* Explain by pointer box:

Variable Memory Address Value

a 0x7fff5fbff5c0 {1,2,3,4,5,6}

ptr 0x7fff5fbff5c0 0x7fff5fbff5c0

0x7fff5fbff5c4 0x7fff5fbff5c8

0x7fff5fbff5c8 0x7fff5fbff5cc

This program defines a 2D integer array a and a pointer to an integer array ptr. The a array is initialized with 6 elements and 3 columns. The ptr pointer is initialized with the address of the first element of the a array.

In the first printf statement, the second and third elements of the first row of the a array are printed by dereferencing the ptr pointer using the array subscript notation. Then the ptr pointer is incremented by one, which means it now points to the second row of the a array.

In the second printf statement, the second and third elements of the second row of the a array are printed by dereferencing the ptr pointer using the array subscript notation.

Program 8

* The running result of the above code is 0,1,1,2,3.

Here is the explanation by pointer box,

Variable Memory Address Value

Ptr 0x7fb60b401c20 0x7fb60b401c30

0x7fb60b401c24 0x7fb60b401c34

0x7fb60b401c28 0x7fb60b401c38

0x7fb60b401c2c 0x7fb60b401c3c

0x7fb60b401c30 0x7fb60b401c40

This program dynamically allocates an integer array ptr of size 5 using the malloc function. Then it initializes the first five elements of the ptr array using a loop.

In the first printf statement, the value of the first element of the ptr array is printed by dereferencing the ptr pointer and then incrementing it by one.

In the second printf statement, the value of the second element of the ptr array is printed by dereferencing the ptr pointer and then incrementing the value it points to by one.

In the third printf statement, the value of the second element of the ptr array is printed by dereferencing the ptr pointer.

In the fourth printf statement, the value of the third element of the ptr array is printed by incrementing the ptr pointer by one and then dereferencing it.

In the fifth printf statement, the value of the third element of the ptr array is printed by dereferencing the ptr pointer and then incrementing the value it points to by one.

Program 9

* The running result of the above code is 20 and 10 .
* Here is the explanation by pointer box

Variable Memory Address Value

arr 0x7ffeefbff4a0 {10,20}

This program defines an integer array arr of size 2 and a function fun that takes an integer array as a parameter. The function increments the parameter array pointer by 1, which means it now points to the second element of the arr array. Then it prints the value of the first element of the parameter array, which is the second element of the arr array.

In the main function, the arr array is initialized with two values. The fun function is called with the arr array as a parameter. Then the value of the first element of the arr array is printed.

Question no 2

Program 1

In this program, a function pointer is defined named fun\_ptr that can point to a function that takes an integer as an argument and returns nothing. A function named fun is also defined which takes an integer as an argument and prints its value.

Pointer box:

[ fun ] 🡺 [ fun\_ptr] 🡺 [ &fun ]

[pointer to fun]

In the main function, fun\_ptr is assigned the address of the fun function using the address-of operator &. Then, the fun function is called using the function pointer fun\_ptr by dereferencing it with the \* operator and passing an argument 10. Finally, the program returns 0.

Program no 2

Pointer Box:

|  |  |  |
| --- | --- | --- |
| 15 | 10 | ch |
| a | b |  |
| add() | Subtract() | multiply() |
| &add, &multiply | &subtract | &multiply |
| fun\_ptr\_arr[0] | fun\_ptr\_arr[1] | fun\_ptr\_arr[2] |

In this pointer box, a and b are integer variables that hold the values 15 and 10, respectively. The three functions add(), subtract() and multiply() are represented by boxes, and the arrows pointing from a and b to the boxes indicate that these values are passed as arguments to the functions. The function pointers &add, &subtract and &multiply are stored in the array fun\_ptr\_arr, and each element of this array points to one of the functions. The user input ch is also shown in the pointer box and is used to select the appropriate function pointer from the array fun\_ptr\_arr. The selected function is then called with the arguments a and b to perform the desired operation.

Program no 3.

main()

Wrapper()

fun1()

fun2()

Here, wrapper() is a function that takes a function pointer as an argument and calls the function it points to. In main(), we call wrapper() twice, passing in the function pointers for fun1() and fun2() respectively.

When wrapper() is called with fun1() as an argument, it calls fun1() and prints "Fun1". Similarly, when wrapper() is called with fun2() as an argument, it calls fun2() and prints "Fun2".

Overall, this program demonstrates how function pointers can be used as arguments to other functions, allowing us to pass functions as arguments to other functions and call them dynamically.

Program 4

│ Program │

│ Variable │ Value │

│ arr[] │ [10, 5, ││ │ 15, 12, ││ │ 90, 80] │

│ n 6 |

│ i │uninitial│

│ compare() │

│ a void \*

│ b void \* │

│ main() │

│ arr[] int \*

│ n 6

│ i │uninitial│

The program begins by defining a function called compare() which takes in two const void pointers as arguments and returns an integer value.

In the pointer box, compare() is represented with its name and signature, along with its two arguments of type void\*.

Inside compare(), the values of the two pointer arguments are cast to int\* using typecasting.

The difference between the two int values pointed to by these pointers is calculated and returned as the result of the compare() function.

Next, in the main() function, an integer array arr is defined and initialized with six values.

In the pointer box, main() is represented with its name and signature, along with the arr[] variable declared as an int\* pointer.

The size of the arr array is calculated using the sizeof() operator, and then divided by the size of a single element in the array to get the number of elements.

The qsort() function is called with the arr array, its size, the size of each element, and the compare() function as arguments. This function sorts the elements of the array in ascending order using the compare() function.

Finally, the sorted elements of the array are printed using a for loop, and the program returns 0.

In the pointer box, the n variable is declared to store the number of elements in the arr[] array, and the i variable is declared to use as an index in the for loop.

The qsort() function and its arguments are also represented in the pointer box, along with the sorted values of the arr array.

Program no 5.

Pointer to int array | -------->Pointer to search key

int\* int\*

Size of array (int) | --------> Pointer to compare

-

int bool (\*)(const void\*,

const void\*)

| Size of element (int) -------->| Result of comparison |

| int bool |

Pointer to void -------->

void\*

The program is designed to search for an element in an array of integers using a function pointer. Here's an explanation of each component of the pointer box:

Pointer to int array: This is a pointer to the beginning of the integer array that needs to be searched.

Size of array (int): This is an integer that represents the size of the integer array in bytes.

Size of element (int): This is an integer that represents the size of each element in the array in bytes.

Pointer to search key: This is a pointer to the integer that needs to be searched in the array.

Pointer to compare: This is a function pointer that points to a function used to compare two elements of the array. The function returns true if the two elements are equal, and false otherwise. The function takes two arguments of type const void\*.

Result of comparison: This is a boolean value returned by the compare function that indicates whether the search key was found in the array or not.

Pointer to void: This is a pointer to void that points to the beginning of the array. This is used to increment the pointer to point to the next element in the array during the search.

Overall, this pointer box shows how the different components of the program are connected to each other and how the data flows between them during the search for an element in the array.

Question no 3

Program 1

returnPointer() main()

int\* return int i = 10

int\* ptr1

int\* ptr2

pt

v

ptr

int\* ptr2

int\* ptr1

int i = 10

Int\* pt = &i

In this program, main() declares an integer variable i with the value of 10. It also declares two integer pointer variables ptr1 and ptr2. ptr1 is assigned to the address of i, while ptr2 is assigned to the value returned by the returnPointer() function when called with the address of i as its argument. Finally, the program prints the value of \*ptr1 and \*ptr2.

The returnPointer() function simply takes a pointer to an integer as its argument, and returns the pointer itself. In other words, it returns the address of the integer passed as an argument.

The pointer box illustrates that returnPointer() takes a pointer to an integer as its input and returns the same pointer as its output. main() calls returnPointer() with the address of i, which is assigned to pt within returnPointer(). The function then returns pt, which is assigned to ptr2 in main(). Finally, \*ptr2 is printed, which should output the value of i (which is 10 in this case).

Program 2

Function

int \*returnFunc()

Pointer to int

int \*ptr

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

Integer

i

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

The program defines a function returnFunc() which returns a pointer to an integer.

In the main() function, a pointer ptr is defined and initialized with the value returned by returnFunc().

The program then tries to print the value pointed by ptr multiple times.

However, there is an error in the returnFunc() function where it returns the address of a local variable i.

The local variable i is destroyed once the function returns, making the pointer returned by the function invalid.

When the main() function tries to access the value pointed by ptr, it may result in undefined behavior, suchs printing a garbage value or crashing the program.

Program 3

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

Function returning

int pointer

int\* returnFunc()

int i = 10

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

V

Pointer to integer

int \*ptr

printf statement

" \*ptr = %d"

\*ptr

* The program defines a function named returnFunc() which returns a pointer to an integer. In the main() function, the return value of returnFunc() is stored in an integer pointer named ptr. Then, the value pointed to by ptr is printed thrice using a printf() statement.
* Inside the returnFunc() function, a static integer i is initialized to 10. The address of this variable i is returned by the function as an integer pointer.

Since i is a static variable, it exists throughout the lifetime of the program. Therefore, the pointer ptr pointing to i will still be valid even after the returnFunc() function has finished executing.

* The output of the program will be the value 10 printed three times, as the value of i is static and doesn't change between function calls.