Qno1)

Program 1 :

#include<stdio.h>

void fun(int \*p)

{

int q = 10;

p = &q;

}

int main()

{

int r = 20;

int \*p = &r;

fun(p);

printf("%d", \*p);

return 0;

}

Output : 20

Explanation and pointer box :

Before calling fun(p):

Box 1 (int r):

+----+----+

| 20 | |

+----+----+

Address: 1000

Box 2 (int \*p):

+----+----+

|1000| |

+----+----+

Address: 2000

Inside fun():

Box 3 (int q):

+----+----+

| 10 | |

+----+----+

Address: 3000

Box 4 (int \*p):

+----+----+

|3000| |

+----+----+

Address: 4000

In this program, when the fun() function is called, it doesn't change the original pointer p in the main() function. The pointer in fun() is a local variable, and any change to it doesn't affect the original pointer. Therefore, \*p in the main() function still points to r, and the output is 20.

Program 2:

#include<stdio.h>

void fun(int \*\*pptr)

{

static int q = 10;

\*pptr = &q;

}

int main()

{

int r = 20;

int \*p = &r;

fun(&p);

printf("%d", \*p);

return 0;

}

Output : 10

Explanation and pointer box:

Before calling fun(&p):

Box 1 (int r):

+----+----+

| 20 | |

+----+----+

Address: 1000

Box 2 (int \*p):

+----+----+

|1000| |

+----+----+

Address: 2000

Inside fun():

Box 3 (static int q):

+----+----+

| 10 | |

+----+----+

Address: 3000

Box 4 (int \*\*pptr):

+----+----+

|2000| |

+----+----+

Address: 4000

In this program, the fun() function receives a double pointer, and it changes the original pointer p in the main() function to point to q. Therefore, \*p in the main() function now points to q, and the output is 10.

Program 3:

#include <stdio.h>

int main()

{

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

int \*ptr = (int\*)(&a+1);

printf("%d %d", \*(a+1), \*(ptr-1));

return 0;

}

Output: ‘2 5’

Explanation and pointer box:

Box 1 (int a[5]):

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

| 1 | 2 | 3 | 4 | 5 | |

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

Address: 1000

Box 2 (int \*ptr):

+----+----+

|1000| |

+----+----+

Program 4:

#include <stdio.h>

char \*c[] = {"GeksQuiz", "MCQ", "TEST", "QUIZ"};

char \*\*cp[] = {c+3, c+2, c+1, c};

char \*\*\*cpp = cp;

int main()

{

printf("%s ", \*\*++cpp);

printf("%s ", \*--\*++cpp+3);

printf("%s ", \*cpp[-2]+3);

printf("%s ", cpp[-1][-1]+1);

return 0;

}

Output: TEST sQuiz Z CQ

Pointer box and explanation:

Before executing the printf statements:

Box 1 (char \*c[4]):

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

| "GeksQuiz"| "MCQ" |"TEST"| "QUIZ"|

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

Address: 1000

Box 2 (char \*\*cp[4]):

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

|1003 | 1001 | 1000 | 1012 |

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

Box 3 (char \*\*\*cpp):

+-----+

|2000 |

+-----+

During the printf statements:

printf("%s ", \*\*++cpp);

cpp is incremented, pointing to cp[1], which points to c[2] ("TEST"). Output: "TEST".

printf("%s ", \*--\*++cpp+3);

cpp is incremented, pointing to cp[2], which points to c[1] ("MCQ"). Then, the pointer is decremented to c[0] ("GeksQuiz") and 3 is added to the pointer, resulting in "sQuiz". Output: "sQuiz".

printf("%s ", \*cpp[-2]+3);

cpp[-2] points to cp[0], which points to c[3] ("QUIZ"). Adding 3 to the pointer results in "Z". Output: "Z".

printf("%s ", cpp[-1][-1]+1);

cpp[-1] points to cp[1], and cpp[-1][-1] points to c[2] ("TEST"). Adding 1 to the pointer results in "EST". Output: "EST".

Program 5 :

Output: GeeksQuiz

Pointer box and explanation:

Before calling fun(&str):

Box 1 (char \*str):

+-----+

|1000 |

+-----+

Address: 2000

Memory block allocated by malloc (100 \* sizeof(char)):

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

| "GeeksQuiz" |

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

Address: 1000

Inside fun():

Box 2 (char \*\*str\_ref):

+-----+

|2000 |

+-----+

Address: 3000

After calling fun(&str):

No changes in the memory layout since the function only increments the local pointer str\_ref and doesn't modify the original pointer str. Therefore, the output of the program will be "GeeksQuiz".

Program 6:

Output : gh

Pointer box and explanation:

Before calling f(argv):

Box 1 (char \*argv[6]):

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

| "ab" | "cd" | "ef" | "gh" | "ij" | "kl" |

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

Address: 1000

Inside f():

Box 2 (char \*\*p):

+-----+

|1000 |

+-----+

Box 3 (char \*t):

+-----+

|1004 |

+-----+

The function f increments the pointer p by the size of an int (which is typically 4 bytes), then assigns the result to the pointer t. It then prints the string pointed to by t, which is "ef". The output of the program is "ef".

Program 7:

Output: 2 3 5 6

Pointer box and explanation:

Before executing the printf statements:

Box 1 (int a[][3]):

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

| 1 | 2 | 3 | 4 | 5 | 6 |

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

Address: 1000

Box 2 (int (\*ptr)[3]):

+-----+

|1000 |

+-----+

During the printf statements:

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

(\*ptr)[1] is equivalent to a[0][1], which is 2.

(\*ptr)[2] is equivalent to a[0][2], which is 3.

++ptr;

ptr is incremented, pointing to a[1].

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

(\*ptr)[1] is equivalent to a[1][1], which is 5.

(\*ptr)[2] is equivalent to a[1][2], which is 6.

Program 8:

Output: 0 1 2 3 3

Pointer box and explanation:

Before executing the printf statements:

Box 1 (int \*ptr):

+-----+

|1000 |

+-----+

Memory block allocated by malloc (5 \* sizeof(int)):

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

| 0 | 1 | 2 | 3 | 4 |

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

Address: 1000

During the printf statements:

printf("%d ", \*ptr++);

Value at address pointed by ptr is printed, which is 0. ptr is incremented and now points to the second element.

printf("%d ", (\*ptr)++);

Value at address pointed by ptr is printed, which is 1. The value itself is incremented, not the pointer.

printf("%d ", \*ptr);

Value at address pointed by ptr is printed, which is now 2 (1 was incremented).

printf("%d ", \*++ptr);

ptr is incremented and now points to the fourth element. Value at the address pointed by ptr is printed, which is 3.

printf("%d ", ++\*ptr);

Value at the address pointed by ptr is incremented and printed, which is 4 (3 was incremented).

Program 9:

Output: 10 20

Pointer box and explanation:

Before calling fun(arr):

Box 1 (int arr[2]):

+----+----+

| 10 | 20 |

+----+----+

Address: 1000

Inside fun():

Box 2 (int \*arr):

+-----+

|1000 |

+-----+

During the execution of fun():

The local pointer arr is incremented to point to the second element of the array. The value of arr[0] is printed, which is 20 (the value at the incremented pointer). The original array in the main function remains unchanged.

After calling fun(arr):

Box 1 (int arr[2]):

+----+----+

| 10 | 20 |

+----+----+

Address: 1000

The printf statement in the main function prints arr[0], which is 10.

The output of the program is "20 10".

Qno.2)

Program 1:

Output : The value of a is 10

Pointer box explanation:

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

| fun |---->| fun |

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

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

| fun\_ptr |---->| fun |

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

1. The fun function is defined, which takes an integer argument a and prints its value.
2. In the main function, a function pointer called fun\_ptr is declared with the same signature as the fun function.
3. The address of the fun function is assigned to the fun\_ptr function pointer. In the pointer box representation, you can see that fun\_ptr points to the same function as fun.
4. The function fun is called indirectly through the function pointer fun\_ptr by dereferencing it and passing the argument 10. The program will output "Value of a is 10".

In summary, Program 1 demonstrates the use of function pointers to indirectly call a function. The pointer box representation helps visualize how the function pointer fun\_ptr is pointing to the fun function and how it is used to call the function with the provided argument.

Program 2:

Output: Enter Choice: 0 for add, 1 for subtract and 2 for multiply #I choose 0 for addition so output is

0

Addition is 25

Pointer box and explanation: +---+---+---+

| 0 | 1 | 2 |

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

| | |

v v v

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

|add| |subtract| |multiply|

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

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

| fun\_ptr\_arr|

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

|

v

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

| 0 | 1 | 2 |

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

1. The add, subtract, and multiply functions are defined, which take two integer arguments a and b and perform the respective operations.
2. In the main function, an array of function pointers called fun\_ptr\_arr is declared, with the same signature as the add, subtract, and multiply functions. The array is initialized with the addresses of these functions.
3. The user is asked to choose which operation to perform (0 for addition, 1 for subtraction, and 2 for multiplication). The user input is stored in the variable ch.
4. The program checks if the user's choice is within the valid range (0-2). If not, the program exits.
5. If the user's choice is valid, the corresponding function is called through the array of function pointers using (\*fun\_ptr\_arr[ch])(a, b).

In summary, Program 2 demonstrates the use of an array of function pointers to call different functions based on user input. The pointer box representation helps visualize how the array of function pointers stores the addresses of the functions and how they are used to call the functions with the provided arguments.

Program 3:

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

fun | fun1 |

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

wrapper | |

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

main | |

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

Explanation:

* void (\*fun)() declares a function pointer fun that takes no arguments and returns void.
* wrapper(fun) calls the wrapper function with the fun1 function pointer as an argument.
* wrapper(fun) calls the wrapper function with the fun2 function pointer as an argument.

Program 4:

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

arr | 10 |

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

| 5 |

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

| 15 |

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

| 12 |

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

| 90 |

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

| 80 |

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

compare | |

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

main | |

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

Explanation:

* int compare (const void \* a, const void \* b) defines the comparison function compare that takes two pointers to const void arguments and returns an integer.
* \*(int\*)a - \*(int\*)b compares the values pointed to by a and b. It first casts a and b to pointers to int using (int\*) and then dereferences them using \*. The difference between the values is returned.
* qsort (arr, n, sizeof(int), compare) sorts the arr array of n elements of size sizeof(int) using the compare function for comparison.
* for (i=0; i<n; i++) printf ("%d ", arr[i]) prints the sorted array using a for loop.

Program 5:

compare is a function that takes two pointers to constant void and returns a boolean. Inside the function, the pointers are cast to int pointers, dereferenced, and their values are compared. If the values are equal, the function returns true, otherwise it returns false.

search is a function that takes four arguments: a pointer to the array to be searched (arr), the number of elements in the array (arr\_size), the size of each element (ele\_size), a pointer to the value to be searched for (x), and a pointer to a function that compares two elements (compare). The function first casts the arr pointer to a char pointer to allow for pointer arithmetic. It then loops through the array, calling the comparison function on each element until a match is found or the end of the array is reached. If a match is found, the function returns the index of the matching element. Otherwise, it returns -1.

main declares an array arr of integers and initializes it with some values. It then calculates the size of the array and assigns it to n. It also initializes a variable x with the value 7. The function pointer compare is passed as an argument to search, along with arr, n, sizeof(int), and a pointer to x. The return value of search is printed to stdout using printf.

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

| main | | compare |

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

| | | |

| int arr[] = { ... } | | bool compare(const |

| int n = sizeof(arr) | | void \*a, const void |

| ... | | \*b) { |

| | | return (\*(int\*)a |

| | | == \*(int\*)b); |

| printf("Returned... | | } |

| search(arr, n, ... | | |

| | | |

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

| ^

| |

| |

| |

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

| | |

+----------v-----------+ +--------v------------+ +------------------v----------------+

| search | | compare wrapper | | printf |

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

|

Qno.3)

Program 1:

1. The first program declares a function returnPointer that takes a pointer to an integer as an argument and returns a pointer to an integer. In main, an integer variable i is initialized to 10. Two integer pointers ptr1 and ptr2 are then declared. ptr1 is assigned the address of i, and ptr2 is assigned the return value of returnPointer when it is called with the address of i. The program then outputs the values pointed to by ptr1 and ptr2. Since both pointers point to the same variable i, the output should be the same for both pointers. This program demonstrates the use of a function that returns a pointer to an integer.

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

| main | | returnPointer |

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

| | | |

| int i = 10; | | int \*returnPointer(int|

| int \*ptr1, \*ptr2; | | \*p) { |

| ptr1 = &i; | | return pt; |

| ptr2 = returnPointer(| | } |

| &i); | | |

| printf("\n \*ptr1 = %d| | |

| \*ptr1); | | |

| printf("\n \*ptr2 = %d| | |

| \*ptr2); | | |

| return 0; | | |

| | | |

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

| ^

| |

| |

| |

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

| | |

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

| returnPointer | | |

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

| int \*returnPointer(int| | |

| \*p) { | | |

| return pt; | | |

| } | | |

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

Program 2:

1. The second program declares a function returnFunc that returns a pointer to an integer. In main, an integer pointer ptr is declared and assigned the return value of returnFunc. The program then outputs the value pointed to by ptr three times. However, returnFunc returns a pointer to a local variable i, which goes out of scope when the function returns. This means that the value of the pointer is invalid, and the program's behavior is undefined. This program demonstrates the danger of returning a pointer to a local variable.

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

| main | | returnFunc |

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

| | | |

| int \*ptr; | | int \*returnFunc

Program 3:

1. The third program is similar to the second program, but the variable i in returnFunc is declared as static. This means that it retains its value even after the function returns. In main, an integer pointer ptr is declared and assigned the return value of returnFunc. The program then outputs the value pointed to by ptr three times. Since i is declared as static, the value of the pointer remains valid, and the program's behavior is well-defined. This program demonstrates the use of a function that returns a pointer to a static variable.

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

| main | | returnPointer |

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

| | | |

| int i = 10; | | int \*returnPointer(int|

| int \*ptr1, \*ptr2; | | \*p) { |

| ptr1 = &i; | | return pt; |

| ptr2 = returnPointer(| | } |

| &i); | | |

| printf("\n \*ptr1 = %d| | |

| \*ptr1); | | |

| printf("\n \*ptr2 = %d| | |

| \*ptr2); | | |

| return 0; | | |

| | | |

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

| ^

| |

| |

| |

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

| | |

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

| returnPointer | | |

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

| int \*returnPointer(int| | |

| \*p) { | | |

| return pt; | | |

| } | | |

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