Lab 10 (Data Structure)

1. The running result of this program will be "20".

An integer variable named "r" is declared and given the value 20 in the main function. The address of "r" is then given to a pointer "p," which is then declared.

A pointer is passed as an argument to the function "fun," but in the function body, a new integer variable "q" is defined, initialized to 10, and the address of "q" is given to the pointer "p." The pointer "p" in the main function is supplied by value to the "fun" function, therefore this assignment has no impact on it.

As a result, when the main function uses the printf function and the \* operator to print the value of the variable pointed to by "p," it also publishes the value of "r," which is still 20.

+---+ +---+

r: | 20| p:-->| |

+---+ +---+

|

|

V

+---+

| |

+---+

q:10

The address of "r" is given to the pointer "p" in the main function, meaning that "p's" value is the address of the memory region holding 20. The pointer "p" is supplied by value when the function "fun" is called, which indicates that a copy of the pointer is made and utilized inside the function.

The value of the copy of the pointer inside the function changes when the function "fun" assigns the address of "q" to the pointer "p," but the original pointer "p" in the main function is unaffected. As a result, when the program uses the \* operator to output the value of the variable referred to by "p," it prints the value of "r," which is still 20.

1. The running result of this program will be "10".

The function "fun" in this program accepts a reference to another pointer as an input. A static integer variable named "q" is declared inside the function and given the initial value of 10. The address of "q" is then assigned to the pointer referred to by the argument "pptr" using the dereference operator "\*."

A pointer "p" is defined and given the location of the integer variable "r" in the main function. An integer variable "r" is declared and initialized to 20. The address-of operator "&" is then used to send "p's" address to the function "fun."

The address of the pointer "p" is supplied by reference when the function "fun" is invoked. The pointer pointed to by "pptr" is given the address "q" within the function, indicating that it now links to the memory region holding 10.

The value of "q," which is 10, will be printed when the main function uses the printf function and the \* operator to print the value of the variable pointed to by "p."

+---+ +---+

r: | 20| p:-->| |

+---+ +---+

|

|

V

+---+ +---+

pptr:| | ----> | |

+---+ +---+

|

|

V

+---+

| 10|

+---+

q:10

The address of "r" is given to the pointer "p" in the main function, meaning that "p's" value is the address of the memory region holding 20. A pointer to a pointer is used to transmit the address of "p" to the function "fun" when it is called. The pointer pointed to by "pptr" is given the address "q" within the function, indicating that it now links to the memory region holding 10.

The value of "q," which is 10, is printed when the main function uses the \* operator to display the value of the variable pointed to by "p." This is because the memory address holding 10 is now pointed to by the pointer "p," which the function "fun" assigned.

1. The running result of this program will be "2 5"

This program declares and initializes an integer array "a" of size 5 with the values 1, 2, 3, 4, and The address of the memory region after the end of array "a" is then defined and initialized using a pointer "ptr," which is created by adding the array size to the array's address. The typecast operator is used to convert the pointer "ptr" into an integer pointer.

The program reads the second and last elements of the array "a" when printing the values of \*(a+1) and \*(ptr-1), respectively. Since the size of an integer is 4 bytes and "ptr" corresponds to the memory address after the end of the array "a," the expressions \*(a+1) and \*(ptr-1) are equal to a[1] and a[4, respectively.

a ptr

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

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

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

| |

| +----------------------> (garbage value)

|

+-------------------------------> (garbage value)

In this figure, "a" denotes the array's beginning, while "ptr" denotes the memory address immediately following the array's conclusion. The matching array members are shown next to the values of \*(a+1) and \*(ptr-1).

1. This program defines a function called fun that accepts a character pointer (i.e., a char \*\*) as a parameter. The pointer to the pointer's value is increased inside of fun. (i.e., it now points to the next memory location).

A character pointer str is defined in the main function, and malloc is used to allocate memory to it.(). After that, strcpy is used to copy the string "GeeksQuiz" to the RAM that was allotted.(). A pointer to str is sent as an argument when calling the fun() method.

char \*str -> uninitialized pointer

└─ points to NULL

malloc(100\*sizeof(char)) -> allocate 100 bytes of memory for str

└─ str -> points to the beginning of the allocated memory block

strcpy(str, "GeeksQuiz") -> copy the string "GeeksQuiz" to the allocated memory

└─ str -> points to the beginning of the copied string in memory

&str -> pointer to pointer to a character

└─ points to the address of str, which is the address of the first character in the string

fun(&str) -> call the fun function with a pointer to the str pointer

└─ str\_ref -> points to the address of the str pointer in main

str\_ref++ -> increment the pointer to the pointer

└─ str\_ref -> now points to the next memory location, but this has no effect on the original str pointer in main

puts(str) -> print the string pointed to by str

└─ "GeeksQuiz" is printed to the console

free(str) -> deallocate the memory used by str

1. The output of this program will be: cd

This program takes an array of character pointers as input and calls a function f() that performs some operations on the input. In the main() function, an array argv of 6 character pointers is declared and initialized with some string literals. This array is then passed as an argument to the function f().

In the f() function, a character pointer t is declared. The pointer p is then incremented by the size of an integer, which advances it to the second element of the input array argv. The expression (p += sizeof(int))[-1] accesses the previous element of the incremented p pointer. This previous element is assigned to the t pointer.

Finally, the printf() function is used to print the string pointed to by t, which is the third element of the argv array.

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

| | | | | | | | | | | | | | | |

0 | |----->| |------>| 'a'| | 'c'|----->| 'e'|------>| 'g'|------>| 'i'|------>| 'k'|

| | | | | | | | | | | | | | | |

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

argv[0] argv[1] argv[2] argv[3] argv[4] argv[5] p t

(Initially) \*p = argv[0], \*(p+1) = argv[1], \*(p+2) = argv[2], \*(p+3) = argv[3], \*(p+4) = argv[4], \*(p+5) = argv[5]

p += sizeof(int) => p = p + 4 => \*(p-1) = argv[1], \*(p-2) = argv[0], \*(p-3) = uninitialized

t = \*(p-1) = argv[1] => t points to the string "cd"

printf("%s\n", t) => prints "cd"

1. The given program is using dynamic memory allocation to create an array of 5 integers. It then performs some operations on the elements of the array using pointer arithmetic and increment operators. Let's see the running result of this program and its explanation.

Running result: 0 1 2 3 4

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

| 0 | 1 | 2 | 3 | 4 | <- allocated memory block

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

printf("%d ", \*ptr++); // Output: 0

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

ptr ----> | 0 | 1 | 2 | 3 | 4 |

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

^

|

ptr++

printf("%d ", (\*ptr)++); // Output: 1

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

ptr ----> | 0 | 2 | 2 | 3 | 4 |

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

^

|

(\*ptr)++

printf("%d ", \*ptr); // Output: 2

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

ptr ----> | 0 | 2 | 2 | 3 | 4 |

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

^

|

printf("%d ", \*++ptr); // Output: 3

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

ptr ----> | 0 | 2 | 2 | 3 | 4 |

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

^

|

ptr++

printf("%d ", ++\*ptr); // Output: 4

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

ptr ----> | 0 | 2 | 3 | 3 | 4 |

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

^

|

(\*ptr)++

1. Initially, an integer array arr with two elements {10, 20} is declared in the main() function.

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

arr --> | 10 | 20 |

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

Then, the fun() function is called with arr passed as an argument. In the fun() function, the array arr is received as a pointer to its first element.

arr

|

v

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

arr[0] -->| 10 | 20 |

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

The pointer arr is then incremented by one. This makes the pointer point to the second element of the array.

arr

|

v

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

arr[0] | 10 | 20 |

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

| |

v v

11 21

The second element of the array is then printed, which is 20.

arr

|

v

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

arr[0] | 10 | 20 |

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

| |

v v

11 21

|

v

20

Finally, the value of the first element of the original array is printed, which is 10.

arr

|

v

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

arr[0] | 10 | 20 |

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

| |

v v

11 21

|

v

20

|

v

10

Therefore, the output of the program will be: 20 10

1. a. A function is called by the program via function pointers.

A function pointer named fun\_ptr is declared in the main() function; it is a pointer to a function that accepts an integer argument and returns void. Using the address-of operator &, the address of the fun() function is allocated to fun\_ptr.

Then, using the function call operator, the fun\_ptr is dereferenced and called with the integer argument 10. (). The fun() method is invoked with the parameter 10 and writes "Value of an is 10" to the console.

+---+ +---+

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

+---+ +---+

| ^

| |

| |

| |

| |

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

The function pointer fun\_ptr is a variable that holds the address of the fun() function. The arrow --> represents the assignment of the function address to the function pointer. When the fun\_ptr is dereferenced and called with the argument 10, it points to the fun() function and passes the argument to it. The fun() function then prints the value of the argument to the console.

b. +-------+

fun\_ptr\_arr[0] | +---|----------------> add()

+-------+

fun\_ptr\_arr[1] | +---|----------------> subtract()

+-------+

fun\_ptr\_arr[2] | +---|----------------> multiply()

+-------+

A function pointer array called "fun\_ptr\_arr" is declared by the program. It contains three function pointers, each of which points to the methods "add()," "subtract()," and "multiply()."

The application allows the user to provide an option (0, 1, or 2) for choosing the preferred function in the "main()" function. The variable "ch" stores the user's selection.

The program then calls the function pointed to by the function pointer at the index specified by ch. The program achieves this by dereferencing the function pointer with the \* operator and passing the function arguments a and b in parentheses (). The resulting function call is (\*fun\_ptr\_arr[ch])(a, b).

If the user enters a value greater than 2 for ch, the program returns 0 and terminates.

Finally, the program returns 0.

c. Two functions, "fun1()" and "fun2()," are defined by the program and each prints a string to the terminal. A function pointer is passed as an argument to the "wrapper()" function, which then calls the function it points to.

In 'main()', 'wrapper()' is called twice, once with a pointer to 'fun1()' and once with a pointer to 'fun2()'. This causes the respective functions to be called and their strings to be printed to the console.

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

| main |

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

|

|

v

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

| wrapper |

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

|

|

v

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

| fun1 |

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

| "Fun1\n" |

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

|

|

v

Console Output: "Fun1"

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

| main |

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

|

|

v

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

| wrapper |

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

|

|

v

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

| fun2 |

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

| "Fun2\n" |

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

|

|

v

Console Output: "Fun2"

d. The program utilizes the compare function as a comparison function and the qsort function to sort an array of numbers in ascending order.

Two pointers to constant void that are converted to pointers to integers are sent as inputs to the "compare" method. Following a dereference operation to acquire the integers these pointers correspond to, the function returns the difference between them.

In 'main', an array 'arr' is initialized with some integers, and its size is calculated using 'sizeof'. The 'qsort' function is called with the array 'arr', its size n, the size of an element in the array, and the compare function as arguments. After the array is sorted, the sorted elements are printed using a loop.

| arr | n | i |

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

| {10, 5, 15, | 6 | undef |

| 12, 90, 80} | | |

qsort (arr, n, sizeof(int), compare);

| arr | n | i |

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

| {5, 10, 12, | 6 | undef |

| 15, 80, 90} | | |

for (i=0; i<n; i++)

printf ("%d ", arr[i]);

Output: 5 10 12 15 80 90

e. The output of the program will be:

Returned index is 2

The 'search' function is used by the software to determine an element's index inside an array. The function requires an array, its size, the size of each element, the element to be searched for, and a comparison function. The comparison function accepts two void pointers and, after comparing the two components, produces a boolean value.

An integer array named "arr" is initialized and its size, "n," is determined in the main function. The element to be looked for is represented by the value 7 of an integer variable named "x." The array, its size, the sizes of each element, the element to "search" for, and the comparison function are sent to the search function when it is called. The index that was returned is then printed.

The "compare" function accepts two void pointers as input, converts them to integer pointers, and then compares the values. If the two integers are equal, it gives back true; otherwise, it gives back false.

The array's starting position is indicated by the pointer "ptr" used by the "search" function. The array is then iterated over while using the comparison function to compare each entry to the element being searched for. The element's index is returned if the comparison function has returned true. The return value is -1 if the element cannot be found.

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

| ptr | i | arr\_size | ele\_size |

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

| | | | |

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

| | | |

| | | +----------> Size of each element (integer)

| | |

| | +-------------------> Size of the array (number of elements)

| |

| +------------------------------> Current index in the array

|

+------------------------------------> Pointer to the beginning of the array

1. The program declares an integer variable 'i' and initializes it to 10.

The program declares two integer pointers 'ptr1' and 'ptr2'.

The program assigns the address of i to ptr1.

The program calls the function 'returnPointer' with the address of i as an argument.

The 'returnPointer' function takes a pointer to an integer as an argument and returns the same pointer.

The program assigns the returned pointer from 'returnPointer' to 'ptr2'.

The program prints the values of '\*ptr1' and '\*ptr2' which should both be 10.

Pointer Box Diagram:

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

| i | --> | 10 |

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

^

|

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

| ptr1 | --> | &i |

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

// After calling returnPointer(&i)

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

| i | --> | 10 |

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

^ ^

| |

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

| ptr1 | --> | &i |

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

|

+------+ |

| ptr2 | -----+

+------+

// After printing the values

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

| i | --> | 10 |

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

^ ^

| |

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

| ptr1 | --> | &i |

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

|

+------+ |

| ptr2 | -----+

+------+

As we can see, 'ptr1' and 'ptr2' both point to the same memory location where 'i' is stored, which is why '\*ptr1' and '\*ptr2' both have the value of 10.

1. Pointer Diagram:

+--------+

| i |

| 10 |

+--------+

|

|

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

| |

| |

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

| ptr | |returnFunc()|

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

| |

| |

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

| main() | |

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

| |

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

| | i |

| | 10 |

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

| |

| |

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

| \*ptr | |

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

There are two functions: "main()" and "returnFunc()" in the diagram above. The "returnFunc()" function declares the variable "i," and then uses the return statement to send a pointer to this variable back to the "main()" function. The variable "ptr" is then used to hold the pointer.

The address of a local variable "i" is returned by "returnFunc()," which is a flaw in this application. The memory allotted to "i" is freed once "returnFunc()" exits, and the reference to it is rendered useless. This indicates that the program's behavior is ambiguous and subject to unexpected outcomes.

1. The program defines a function named 'returnFunc' that returns a pointer to an integer.

In main, the function is called and the return value is assigned to a pointer variable named 'ptr'.

The 'printf' statements print the value pointed to by 'ptr' three times.

The variable 'i' inside 'returnFunc' is defined as static, which means it retains its value between function calls. Therefore, each time 'returnFunc' is called, it returns a pointer to the same integer variable 'i'.

The integer value 10 is stored in the memory location pointed to by 'ptr'. Therefore, all three 'printf' statements print the value 10.

Since 'i' is defined as static, its memory location is not 'deallocated' when the function 'returnFunc' returns. Therefore, the memory location pointed to by 'ptr' still contains the value 10 even after the function has returned.

Diagram:

+---+

ptr --> | |

+---+

|

|

V

+---+

| 10|

+---+