Day_4 C-Programming

(RECAP OF PREVIOUS DAY)

Pointer,pointer arithmetic and scaling, Pointer Aliasing. call by reference.

Array(one dimensional),array using pointers,manipulating array elements, Linear Search,Binary Search, Bubble Sort.

Pointers

Definition

A pointer is a variable that stores the memory address of another variable. Pointers allow for efficient memory management and enable dynamic memory allocation, passing data by reference, and creating complex data structures like linked lists and trees.

- Pointers are nothing but memory addresses.
- Every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory.
- A pointer is a variable that contains the memory address of another variable.

Declaration and Initialization

- To declare a pointer, use the * operator.
- To initialize a pointer, assign it the address of a variable using the & operator.

Syntax:

<datatype> *pointerName;

Example:

#include <stdio.h>

```
int main() {
    int x = 10;
    int *ptr = &x; // Pointer stores the address of x

    printf("Value of x: %d\n", x);
    printf("Address of x: %p\n", &x);
    printf("Pointer ptr points to address: %p\n", ptr);
    printf("Value at address stored in ptr: %d\n", *ptr); // Dereferencing

    return 0;
}
```

Output:

```
Value of x: 10
Address of x: 0x7ffee1a4c98c
Pointer ptr points to address: 0x7ffee1a4c98c
Value at address stored in ptr: 10
```

Pointer Arithmetic and Scaling

Pointer arithmetic involves operations like addition and subtraction on pointers. When performing arithmetic, the size of the data type being pointed to determines how much the pointer moves in memory.

Valid Operations:

1. Increment (`ptr++`) and decrement (`ptr--`) pointers.
2. Add/subtract integers to/from pointers (`ptr + n`, `ptr - n`).
3. Subtract one pointer from another to calculate the number of elements between them.

Example:

```
#include <stdio.h>


int main() {
    int arr[5] = {10, 20, 30, 40, 50};
    int *ptr = arr; // Points to the first element
    printf("Pointer arithmetic:\n");
    printf("Value at ptr: %d\n", *ptr);
    ptr++;
    printf("Value at ptr++: %d\n", *ptr);
    ptr += 2;
    printf("Value at ptr+2: %d\n", *ptr);
    return 0;
}
```

Output:

```
Pointer arithmetic:
Value at ptr: 10
Value at ptr++: 20
Value at ptr+2: 40
```

Scaling

- The pointer moves in increments of the size of the data type.
- For example, if `int` is 4 bytes, `ptr++` moves the pointer by 4 bytes.

Pointer Aliasing

Pointer aliasing occurs when two or more pointers point to the same memory location. Modifying the data through one pointer affects the data accessed through the other pointer(s).

Example:

```
#include <stdio.h>

int main() {

    int x = 100;

    int *ptr1 = &x;

    int *ptr2 = &x; // Alias of ptr1

    printf("Value of x using ptr1: %d\n", *ptr1);

    printf("Value of x using ptr2: %d\n", *ptr2);

    *ptr1 = 200; // Modifying x using ptr1

    printf("Value of x after modification using ptr1: %d\n", *ptr2);

    return 0;

}
```

Output:

```
Value of x using ptr1: 100

Value of x using ptr2: 100
```

Value of x after modification using ptr1: 200

Call by Reference

In the call by reference method, the function receives the address of the actual parameter. Changes made to the parameter inside the function affect the original value.

Example:

```
#include <stdio.h>

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}


int main() {
    int x = 10, y = 20;
    printf("Before swapping: x = %d, y = %d\n", x, y);
    swap(&x, &y);
    printf("After swapping: x = %d, y = %d\n", x, y);
    return 0;
}
```

Output:

Before swapping: x = 10, y = 20

After swapping: x = 20, y = 10

One-Dimensional Array

Definition

An array is a collection of elements of the same data type stored in contiguous memory locations. Arrays in C are zero-indexed, meaning the first element is at index 0.

Declaration and Initialization

Syntax:

<datatype> arrayName[size];

Example:

```
#include <stdio.h>

int main() {

    int arr[5] = {10, 20, 30, 40, 50}; // Declaration and initialization

    printf("Elements of the array:\n");

    for (int i = 0; i < 5; i++) {

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

    }

    return 0;

}
```

Output:

Elements of the array:

arr[0] = 10

arr[1] = 20

arr[2] = 30

arr[3] = 40

arr[4] = 50

Array Using Pointers

Accessing Array Elements Using Pointers

Array elements can be accessed using pointers because the array name represents the address of the first element.

Example:

```
#include <stdio.h>

int main() {

    int arr[5] = {10, 20, 30, 40, 50};

    int *ptr = arr; // Pointer to the first element
```

```
    printf("Accessing array elements using pointers:\n");
    for (int i = 0; i < 5; i++) {
        printf("Element %d: %d\n", i, *(ptr + i));
    }
    return 0;
}
```

Output:

Accessing array elements using pointers:

Element 0: 10

Element 1: 20

Element 2: 30

Element 3: 40

Element 4: 50

Manipulating Array Elements

Array elements can be modified using direct indexing or pointer arithmetic.

Example:

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

```
int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    printf("Original array:\n");
    for (int i = 0; i < 5; i++) {
        printf("arr[%d] = %d\n", i, arr[i]);
    }
    // Modify elements
    for (int i = 0; i < 5; i++) {
        arr[i] *= 2;
    }
    printf("Modified array:\n");
    for (int i = 0; i < 5; i++) {
        printf("arr[%d] = %d\n", i, arr[i]);
    }
    return 0;
}
```

Output:

Original array:

arr[0] = 1

arr[1] = 2

arr[2] = 3

arr[3] = 4

arr[4] = 5

Modified array:

arr[0] = 2

arr[1] = 4

arr[2] = 6

arr[3] = 8

arr[4] = 10

Linear Search

Linear search involves searching for an element by checking each array element sequentially.

Algorithm:

1. Start from the first element.
2. Compare each element with the target value.
3. Return the index if found, or return -1 if not found.

Example:

```
#include <stdio.h>

int linearSearch(int arr[], int size, int key) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == key) {
            return i; // Return the index
        }
    }
    return -1; // Not found
}


int main() {
    int arr[5] = {10, 20, 30, 40, 50};
    int key = 30;
    int result = linearSearch(arr, 5, key);
    if (result != -1) {
        printf("Element %d found at index %d\n", key, result);
    } else {
```

```
        printf("Element %d not found\n", key);
    }
    return 0;
}
```

Output:

Element 30 found at index 2

Binary Search

Binary search is an efficient search algorithm for sorted arrays. It repeatedly divides the search interval in half.

Algorithm:

1. Compare the middle element with the target.
2. If equal, return the index.
3. If smaller, search the right half; if larger, search the left half.
4. Repeat until the target is found or the interval is empty.

Example:

```
#include <stdio.h>

int binarySearch(int arr[], int size, int key)

{
```

```
    int low = 0, high = size - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] == key) {
            return mid;
        } else if (arr[mid] < key) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return -1; // Not found
}

int main() {
    int arr[5] = {10, 20, 30, 40, 50};
    int key = 40;
    int result = binarySearch(arr, 5, key);
```

```
    if (result != -1) {
        printf("Element %d found at index %d\n", key, result);
    } else {
        printf("Element %d not found\n", key);
    }
    return 0;
}
```

Output:

Element 40 found at index 3

Bubble Sort

Bubble Sort is a simple sorting algorithm that repeatedly swaps adjacent elements if they are in the wrong order.

Algorithm:

1. Compare adjacent elements and swap if needed.
2. Repeat for all elements, reducing the range after each pass.

Example:

```
#include <stdio.h>

void bubbleSort(int arr[], int size) {
```

```
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

int main() {
    int arr[5] = {50, 20, 40, 10, 30};
    printf("Original array:\n");
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }
```

```
    printf("\n");
    bubbleSort(arr, 5);
    printf("Sorted array:\n");
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    return 0;
}
```

Output:

Original array:

50 20 40 10 30

Sorted array:

10 20 30 40 50

Practice Problems

1. Write a program to find the maximum and minimum elements in an array using pointers.
2. Implement linear search to count the occurrences of a target value in an array.
3. Write a program to reverse an array using pointers.
4. Implement binary search on a user-input sorted array.
5. Modify the bubble sort algorithm to sort in descending order.
6. Write a program to find the sum and average of array elements using pointers.
7. Implement a program to merge two sorted arrays into one sorted array.
8. Create a program to find the second largest element in an array.
9. Implement bubble sort using a while loop instead of nested for loops.
10. Write a program to count the number of even and odd elements in an array.
11. Write a program to find the largest of three numbers using call by reference.
12. Implement pointer arithmetic to traverse an array and find the sum of its elements.
13. Demonstrate pointer aliasing by swapping two numbers without using a temporary variable.
14. Write a program to reverse a string using pointers.
15. Implement a function to find the factorial of a number using call by reference.
16. Demonstrate the difference between pointer arithmetic on int and char types.
17. Write a program to dynamically allocate memory for an array and find its average using pointers.
18. Implement matrix multiplication using pointers.
19. Write a program to compare two strings using pointers.
20. Implement a program to find the length of a string using pointer arithmetic.