unit 2 notes

ARRAY

- A linear data structure, which is a Finite collection of similar data items stored in successive or consecutive memory locations
- Only homogenous types of data is allowed in any array. May contain all integer or all character elements, but not both together.
- char c_array [10]; short s_array [20];

Properties of Arrays

1. Non-primary/Secondary Data Type:

- Arrays are derived data types. They are built upon primary data types (like `int`, `char`, `float`, etc.) to store collections of elements of the same type.

2. Contiguous Memory Allocation:

- When you declare an array, elements are stored in consecutive memory locations. This allows for efficient access using indexing.

3. Homogeneous Elements:

- All elements within an array must be of the same data type. This ensures consistent data handling within the array.

4. Random Access:

- You can directly access any element in the array using its index, which starts from 0 (zero-based indexing). This provides flexibility in retrieving or modifying specific elements.

5. Index-Based Access:

- Each element in an array has a unique index, starting from 0. You use this index within square brackets `[]` to access or modify elements. For example, `array[2]` refers to the third element (since indexing starts from 0).

6. Compile-Time Memory Allocation:

- The memory for an array is allocated during compilation when the array size is determined. This means the size cannot be dynamically changed at runtime.

7. Fixed Size at Compile Time:

- As mentioned above, the size of an array is fixed at compile time based on the declared size. You cannot increase or decrease the size of the array once the program is running.

8. Assignment Incompatibility:

- Arrays are not directly assignable to each other. This is because arrays decay to pointers to their first element when used in expressions. You typically need to copy elements individually or use functions to work with array assignment.

9. Out-of-Bounds Access and Undefined Behavior:

- Attempting to access an element outside the valid range of the array (index less than 0 or greater than or equal to the size) leads to undefined behavior. This can cause crashes, unexpected results, or security vulnerabilities. It's essential to perform bound checking (verifying the index is within limits) before accessing elements.

classification of arrrays

category 1

• Fixed Length Array
• Size of the array is fixed at compile time
• Variable Length Array

category 2

• One Dimensional (1-D) Array
• Stores the data elements in a single row or column.
• Multi Dimensional Array
• More than one row and column is used to store the data elements

declaration and initialisations:

compile time initialisations

1. `data-type array_name[size] = { list of values };`

- This is the general syntax for initializing an array at compile time.
- `data-type` specifies the type of elements in the array (e.g., `int`, `float`, `char`).
- `array_name` is the name you give to the array.
- `size` is the number of elements the array can hold.
- `list of values` enclosed in curly braces `{}` provides the initial values for each element. The number of values must match the size of the array, or you'll get a compilation error.

Example:

```
float areas[5] = {23.4, 6.8, 5.5};
```

This creates an array `areas` of size 5 and initializes the first three elements with the given values (23.4, 6.8, and 5.5). The remaining elements (indexes 3 and 4) will be initialized to 0.0 (default for floating-point numbers).

2. `float area[5] = {23.4, 6.8, 5.5}; // Partial initialization`

- This is a variation of the first example, where you provide fewer initial values than the array size.
- In this case, the elements not explicitly initialized will be set to their default values (0 for integers, 0.0 for floats, etc.).

3. `int a[15] = {[2] = 29, [9] = 7, [14] = 48}; // C99's designated initializers`

- This syntax (designated initializers) is available in C99 and later versions.
- It allows you to initialize specific elements of the array by index.
- You list the index within square brackets `[]` followed by the assignment operator `=` and the value.
- Elements not designated will be initialized to 0 (default for integers).

Example:

```
int numbers[15] = {[2] = 10, [7] = 20, [12] = 30};
```

This creates an array `numbers` of size 15 and initializes elements at indexes 2, 7, and 12 with 10, 20, and 30, respectively. Other elements remain 0.

4. `int arr[] = {2, 3, 4}; // sizeof arr is decided`

- In this case, you don't explicitly specify the size of the array.

- The compiler infers the size based on the number of initializers provided within the curly braces `{}`.
- So, `arr` will be created with a size of 3 to accommodate the three values.

5. `int marks[4] = {67, 87, 56, 77, 59}; // undefined behavior`

- This is an example of attempting to initialize an array with more elements than the declared size.
- Since `marks` is declared with a size of 4, providing five initial values leads to undefined behavior.
- The program might compile, but how it behaves at runtime is unpredictable. It could cause crashes, unexpected results, or security vulnerabilities.

runtime initialisations

it is done using loops

problems on arrays

printing elements in reverse order

```
#include<stdio.h>

int main() {

int n;

scanf("%d", &n);

int a[n];

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

scanf("%d", &a[i]);

}

// Reversing the array
```

```
for(int i = n - 1; i >= 0; i--) {

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

}
return 0;

}
```

output:

```
5
6
3
6
7
3
3 7 6 3 6
```

input an array and display square of inputed array

```
#include<stdio.h>

int main() {

int n;

scanf("%d", &n);

int a[n];

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

scanf("%d", &a[i]);

}

for(int i =0;i<n;i++){
```

```
printf("%d ",a[i]*a[i]);

}

return 0;

}
```

output:

```
3
1
2
3
1 4 9
```

to find the largest of the given elements in the array

```
#include<stdio.h>

int main() {

int n;

scanf("%d", &n);

int a[n];

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

scanf("%d", &a[i]);

}

int largest=0;

for(int i =0;i<n;i++){
```

```
if(a[i]>largest){

largest=a[i];

}

}

printf("largest is %d",largest);

return 0;

}
```

output:

```
4
6
4
8
39
largest is 39
```

(impppppp) to delete duplicate in the given array

```
#include <stdio.h>

int main() {
    int a[200], size, i, j, k;
    printf("Enter the number of elements: ");
    scanf("%d", &size);
    printf("Enter array elements: ");
    for (i = 0; i < size; i++)
        scanf("%d", &a[i]);
    for (i = 0; i < size; i++) {
        for (j = i + 1; j < size; j++) {
            if (a[i] == a[j]) {
```

```
                for (k = j; k < size - 1; k++) {
                    a[k] = a[k + 1];
                }
                size--;
                j--;
            }
        }
    }
    printf("Array after removing duplicates: ");
    for (i = 0; i < size; i++)
        printf("%d ", a[i]);
    return 0;
}
```

output:

```
Enter the number of elements: 4
Enter array elements: 1
1
2
4
Array after removing duplicates: 1 2 4
```

to find whether the element is present in array

```
#include <stdio.h>

int main() {
    int n, d;

    // Taking input for the size of the array
    printf("Enter the size of the array: ");
    scanf("%d", &n);

    // Taking input for the element to be searched
    printf("Enter the element to be searched: ");
    scanf("%d", &d);
```

```
    // Declaring the array
    int a[n];

    // Taking input for the elements of the array
    printf("Enter the elements in the array:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &a[i]);
    }

    // Counting occurrences of the element
    int count = 0;
    for (int i = 0; i < n; i++) {
        if (a[i] == d) {
            count++;
        }
    }

    // Printing the result
    if (count > 0) {
        printf("The element %d occurs %d time(s) in the array.\n", d,
count);
    } else {
        printf("The element %d does not exist in the array.\n", d);
    }

    return 0;
}
```

pointers

•A variable which contains the address. This address is the location of another object in the memory
• Used to access and manipulate data stored in memory.
•Pointer of particular type can point to address of any value in that particular type.
•Size of pointer of any type is same/constant in that system
• Not all pointers actually contain an address
Example: NULL pointer // Value of NULL pointer is 0.

•Pointer can have three kinds of contents in it
•The address of an object, which can be dereferenced.
•A NULL pointer
•Undefined value // If p is a pointer to integer, then – int *p;
(refer notes for pointers)

functions

factorial of a number

```
#include<stdio.h>

int fact(int n);

int main() {
    int n;
    scanf("%d", &n);
    int f;
    f = fact(n);
    printf("%d", f);
    return 0;
}

int fact(int n) {
    int f = 1;
    for (int i = 1; i <= n; i++) {
        f = f * i;
    }
    return f;
}
```

fibonacci series

```
#include <stdio.h>

void fib(int n);
```

```
int main() {
    int n;
    scanf("%d", &n);
    fib(n);
    return 0;
}

void fib(int n) {
    int f0 = 0, f1 = 1, f2;
    printf("%d %d ", f0, f1);
    for (int i = 2; i <= n; i++) {
        f2 = f0 + f1;
        printf("%d ", f2);
        f0 = f1;
        f1 = f2;
    }
    return;
}
```

to find the length of the string

```
#include<stdio.h>

void length_of_string(char s[100]);

int main() {
    char s[100];
    scanf("%s", s);
    length_of_string(s);
    return 0;
}

void length_of_string(char s[100]) {
    int length = 0;
    int i;
    for (i = 0; s[i] != '\0'; i++) {
        length = length + 1;
    }
```

```
    printf("%d", length);
    return;
}
```

swap numbers

```
#include<stdio.h>

void swap(int *a, int *b); // Function prototype

int main() {
    int a, b;
    scanf("%d %d", &a, &b);
    swap(&a, &b); // Passing addresses of a and b
    printf("After swap: %d %d\n", a, b);
    return 0;
}

void swap(int *a, int *b) {
    int temp;
    temp = *a; // Dereference the pointers to access the values
    *a = *b;
    *b = temp;
}
```

write a program to input array in main function ,pass array to a function sum

```
#include<stdio.h>

void sum(int a[], int n);

int main() {
    int n;
    scanf("%d", &n); // Remove the extra space here
```

```
    int a[n];
    for(int i = 0; i < n; i++) {
        scanf("%d", &a[i]);
    }
    sum(a, n);
    return 0;
}

void sum(int a[], int n) {
    int sum = 0;
    for(int i = 0; i < n; i++) {
        sum += a[i];
    }
    printf("%d", sum);
}
```

linear search

```
#include<stdio.h>

int linear_search(int a[], int n, int e);

int main() {
    int n, a[100], e;
    scanf("%d %d", &n, &e);

    for(int i = 0; i < n; i++) {
        scanf("%d", &a[i]);
    }

    int f = linear_search(a, n, e);
    if(f == 0) {
        printf("Element not found\n");
    } else {
        printf("Element found\n");
    }

    return 0;
```

```
}

int linear_search(int a[], int n, int e) {
    int f = 0;
    for(int i = 0; i < n; i++) {
        if(a[i] == e) {
            f = 1;
            break;
        }
    }
    return f;
}
```

recursion

factorial of a number

```
#include<stdio.h>

int fact(int n);

int main() {
    int n;
    scanf("%d", &n);
    int f = fact(n);
    printf("%d", f);
    return 0;
}

int fact(int n) {
    if(n == 1) {
        return 1;
    } else {
        return n * fact(n - 1);
    }
}
```

sum of n numbers

```
#include<stdio.h>

int sum(int n);

int main() {
    int n;
    scanf("%d", &n);
    int f = sum(n);
    printf("%d", f);
    return 0;
}

int sum(int n) {
    if(n == 0) {
        return 0;
    } else {
        return sum(n - 1) + n;
    }
}
```

reverse number

```
#include<stdio.h>

int reverse(int n);

int main() {
    int n;
    scanf("%d", &n);
    int f = reverse(n);
    printf("%d", f);
    return 0;
}

int reverse(int n) {
```

```
    int remainder = n % 10;
    int result = remainder;
    n = n / 10;
    if (n == 0) {
        return result;
    } else {
        return reverse(n) * 10 + result;
    }
}
```

(imppp) binary search

```
#include <stdio.h>

int main(void) {
    int arr[] = { 2, 3, 4, 10, 40 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int x = 10, f = 0;
    int l = 0, r = n - 1;
    while (l <= r) {
        int m = l + (r - l) / 2;

        // Check if x is present at mid
        if (arr[m] == x) {
            printf("Element %d found at %d\n", x, m);
            f = 1;
            break;
        }

        // If x greater, ignore left half
        if (arr[m] < x)
            l = m + 1;

        // If x is smaller, ignore right half
        else
            r = m - 1;
    }
```

```
    if (f == 0)
        printf("Not found\n");

    return 0;
}
```

difference

```
#include <stdio.h>

// A recursive binary search function. It returns
// location of x in given array arr[l..r] is present,
// otherwise -1
int binarySearchRecursive(int arr[], int l, int r, int x)
{
    if (r >= l) {
        int mid = l + (r - l) / 2;

        // If the element is present at the middle itself
        if (arr[mid] == x)
            return mid;

        // If element is smaller than mid, then it can only be present in
left subarray
        if (arr[mid] > x)
            return binarySearchRecursive(arr, l, mid - 1, x);

        // Else the element can only be present in right subarray
        return binarySearchRecursive(arr, mid + 1, r, x);
    }

    // We reach here when element is not present in array
    return -1;
}

// A iterative binary search function. It returns
// location of x in given array arr[l..r] if present,
// otherwise -1
int binarySearchIterative(int arr[], int l, int r, int x)
```

```
{
    while (l <= r) {
        int m = l + (r - l) / 2;

        // Check if x is present at mid
        if (arr[m] == x)
            return m;

        // If x greater, ignore left half
        if (arr[m] < x)
            l = m + 1;

        // If x is smaller, ignore right half
        else
            r = m - 1;
    }

    // if we reach here, then element was not present
    return -1;
}

int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int x = 10;

    int resultRecursive = binarySearchRecursive(arr, 0, n - 1, x);
    (resultRecursive == -1) ? printf("Element is not present"
                              " in array using recursive search\n")
                  : printf("Element is present at "
                              "index %d using recursive search\n",
                              resultRecursive);

    int resultIterative = binarySearchIterative(arr, 0, n - 1, x);
    (resultIterative == -1) ? printf("Element is not present"
                              " in array using iterative search\n")
                  : printf("Element is present at "
                              "index %d using iterative search\n",
                              resultIterative);
```

```
    return 0;
}
```

(imp) bubble sort

```
#include <stdio.h>

void swap(int *xp, int *yp) {
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

// A function to implement bubble sort
void bubbleSort(int arr[], int n) {
    int i, j;
    for (i = 0; i < n-1; i++) {
        // Last i elements are already in place
        for (j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                swap(&arr[j], &arr[j+1]);
            }
        }
    }
}

/* Function to print an array */
void printArray(int arr[], int size) {
    int i;
    for (i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n"); // Add a newline after printing the array
}

// Driver program to test above functions
int main(void) {
```

```
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);
    bubbleSort(arr, n);
    printf("Sorted array: ");
    printArray(arr, n);
    return 0;
}
//Sorted array: 11 12 22 25 34 64 90
```

storage classes

enum

function pointer

```
#include<stdio.h>

int add(int a, int b);
int subtract(int a, int b);
int multiply(int a, int b);
int result(int (*f1)(int, int), int a, int b);

int main() {
    int a, b, res;
    printf("Enter two numbers: ");
    scanf("%d %d", &a, &b);
    fflush(stdin);
    printf("Press 'a' for addition\nPress 's' for subtraction\nPress 'm' for
multiplication\n");
    char ch = getchar();

    switch(ch) {
        case 'a':
            res = result(add, a, b);
            break;
        case 's':
            res = result(subtract, a, b);
            break;
```

```
        case 'm':
            res = result(multiply, a, b);
            break;
    }

    printf("Result is %d\n", res);
    return 0;
}

int add(int a, int b) {
    return a + b;
}

int subtract(int a, int b) {
    return a - b;
}

int multiply(int a, int b) {
    return a * b;
}

int result(int (*f1)(int, int), int a, int b) {
    return f1(a, b);
}
```

matrix