Assignment-2

1. C program to find maximum and minimum number from given 10 numbers (using array)

```
#include <stdio.h>
int main() {
    int arr[10];
    int i, max, min;
    // Input 10 numbers
    for (i = 0; i < 10; i++) {
        printf("Enter number %d: ", i + 1);
        scanf("%d", &arr[i]);
    }
    // Assume first element as both max and min
    max = min = arr[0];
    // Check all elements
    for (i = 1; i < 10; i++) {
        if (arr[i] > max)
            max = arr[i];
        if (arr[i] < min)
            min = arr[i];
    }
    printf("Maximum number = %d\n", max);
    printf("Minimum number = %d\n", min);
    return 0;
}
```

2. What is User Defined Function?

A **user defined function** is a function that is **created by the programmer** (user) to perform a specific task.

C already has many library functions like printf(), scanf(), etc., but when we need our own task (like sum of numbers, factorial, etc.), we write **user defined functions**.

Advantages:

- Increases **reusability** of code
- Makes program **modular and readable**
- Easy to **debug and maintain**

Example:

```
int sum(int a, int b) {   // user defined function
    return a + b;
}
```

3. Function prototype, function calling and function definition

In C, using user defined functions generally has **3 main steps**:

(a) Function Prototype (Declaration)

- Tells the compiler **name**, **return type** and **parameter types** of the function.
- Written **before main()**.

```
int sum(int a, int b);   // prototype
```

(b) Function Call

- When we **use** the function in main() (or any other function).
- Control is transferred to the function body.

```
int result;

result = sum(5, 10);   // function call
```

(c) Function Definition

- Actual body of the function.
- Contains the **logic/statement block** that is executed when the function is called.

```
int sum(int a, int b) {   // function definition

   int s;

   s = a + b;

   return s;             // returns value to caller

}
```

Flow:
Prototype → main() calls function → Definition executed → value returned.

4. Storage Class Specifiers in C (with example)

Storage class decides:

- **Where** the variable is stored (memory location)
- **Scope** (where it can be used)
- **Lifetime** (how long it exists)
- **Default value**

Main storage classes in C:

1. auto

- Default for local variables inside a function.
- Stored in stack.
- Lifetime: during function execution only.
- Default value: garbage.

```
void fun() {

   auto int x = 10;  // same as: int x = 10;

}
```

2. register

- Suggests compiler to store variable in **CPU register** for fast access.
- Cannot take its address with & generally.
- Scope: local.
- Lifetime: function execution.

```
void fun() {
   register int i;
   for (i = 0; i < 1000; i++) {
      // fast loop
   }
}
```

3. static

- Preserves value **between function calls**.
- Default value: 0.
- If used inside function → local but lifetime = entire program.

```
void counter() {
   static int c = 0;  // initialized only once
   c++;
   printf("c = %d\n", c); }  /*
Calling counter() 3 times prints:
c = 1
c = 2
c = 3
*/
```

- If used outside any function → global variable with **file scope** (visible only in that file).

4. extern

- Used to declare a **global variable** which is defined in another file or at another place.
- Doesn’t allocate new memory, just refers to existing global variable.

```
int x = 10;     // global variable
void fun() {
    extern int x;   // refers to same x
    printf("%d", x);
}
```

5. How arrays are stored in contiguous memory locations in C?

In C, an array is stored in **contiguous (continuous) memory locations**.
That means:

- All elements of the array are kept **back-to-back in memory**.
- If base address of arr[0] is, say, 1000 (for int type):
  - arr[0] at 1000
  - arr[1] at 1004
  - arr[2] at 1008
    (assuming sizeof(int) = 4 bytes)

So, address of arr[i] =
base_address + i * sizeof(element_type)

6. Why is this property important for pointer arithmetic?

Because of **contiguous memory**, when we use pointers:

- If int *p = arr; then p points to arr[0].
- p + 1 will automatically point to **next element** arr[1].
- This is possible only because compiler knows that next element is exactly sizeof(int) bytes ahead.

Example:

```
int arr[3] = {10, 20, 30};
int *p = arr;    // &arr[0]
printf("%d", *p);      // 10
printf("%d", *(p+1));  // 20
printf("%d", *(p+2));  // 30
```

If elements were **not stored contiguously**, then p+1 wouldn’t correctly reach the next element and pointer arithmetic would break.

7. What happens in int arr[6] = {1, 2}; ? (with memory representation)

In C, when you partially initialize an array:

```
int arr[6] = {1, 2};
```

- arr[0] = 1
- arr[1] = 2
- All **remaining elements are automatically initialized to 0**:
  - arr[2] = 0
  - arr[3] = 0
  - arr[4] = 0
  - arr[5] = 0

Memory representation (assuming base address = 1000 and int = 4 bytes)

Index	**Value**	**Address**
arr[0]	1	1000
arr[1]	2	1004
arr[2]	0	1008
arr[3]	0	1012
arr[4]	0	1016
arr[5]	0	1020

All stored **back-to-back** (contiguously).

8. Why does the function not know the size of the array automatically?

When you pass an array to a function, **it decays to a pointer**.

Example:

```
void printArray(int a[]) {

   // here 'a' is actually treated as int *a

}
```

Inside printArray, the parameter a is not an array anymore; it is just a **pointer to the first element** (int *a).

So:

- sizeof(a) inside function gives **size of pointer** (like 4/8 bytes), not the size of full array.
- The compiler does **not** keep extra information about how many elements are in the array.

Because of this, the function **cannot automatically know the array size**, so we usually pass size separately:

```
void printArray(int a[], int n) {   // n = size
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", a[i]);
}
int main() {
    int arr[6] = {1,2};
    printArray(arr, 6);  // we pass size manually
    return 0;
}
```