EXPERIMENT4: VARIABLES AND SCOPE OF VARIABLES

Q1. Declare a global variable outside all functions & use it in various functions

----

Aim

To understand how a global variable can be accessed inside multiple functions.

Theory

A **global variable** is declared outside all functions.

It can be accessed by **all functions** in the same program.

Algorithm

1. Declare a global variable.
2. Create multiple functions.
3. Access and modify the global variable inside each function.
4. Observe output.
5. End.

Pseudocode

```
declare global variable x
function A → print x
function B → modify x
main → call A and B
```

Flowchart

```
START
 ↓
Declare global variable
 ↓
Call function A → prints global variable
 ↓
Call function B → modifies global variable
 ↓
END
```

C Program

```
#include <stdio.h>

int globalVar = 10;   // Global variable

void func1() {
    printf("Inside func1: globalVar = %d\n", globalVar);
}

void func2() {
    globalVar += 5;
    printf("Inside func2: globalVar = %d\n", globalVar);
}

int main() {
    printf("Inside main: globalVar = %d\n", globalVar);
```

```
    func1();
    func2();

    printf("Back in main: globalVar = %d\n", globalVar);
    return 0;
}
```

Output

```
PS C:\Users\ASUS\Desktop\C Exp> gcc exp4.1.c
PS C:\Users\ASUS\Desktop\C Exp> ./a.exe
Initial value of count = 0
Current value of count = 1
Current value of count = 2
PS C:\Users\ASUS\Desktop\C Exp>
```

Q2. Declare a local variable inside a function & try accessing it outside

----

Aim

To compare accessibility of local vs global variables.

Theory

- A **local variable** exists only inside its function.
- It cannot be accessed from other functions or main().
- Global variables can be accessed anywhere.

Algorithm

1. Create function with local variable.
2. Try to access it in main.
3. Observe compile-time error.
4. End.

Pseudocode

```
function test:
    declare local x = 5
    print x
main:
    try print x → ERROR
```

Flowchart

```
Function block creates local variable
 ↓
Local variable destroyed after function ends
 ↓
Access outside → ERROR
```

C Program

```c
#include <stdio.h>

void test() {
    int localVar = 20;   // Local variable
```

```
    printf("Inside test(): localVar = %d\n", localVar);
}

int main() {
    test();

    // printf("%d", localVar);  // ERROR: localVar is not accessible
here

    printf("Cannot access localVar outside test() function.\n");
    return 0;
}
```

Output

```
PS C:\Users\ASUS\Desktop\C Exp> gcc exp4.2.c
PS C:\Users\ASUS\Desktop\C Exp> ./a.exe
Inside main():
Global variable = 10
Local variable in main = 5

Inside showLocal():
Global variable = 10
Local variable in showLocal = 20
PS C:\Users\ASUS\Desktop\C Exp>
```

Q3. Declare variables inside different code blocks and test accessibility

---

Aim

To understand block-level scope.

Theory

A **block** is a section enclosed in `{ }`.

Variables declared inside a block are accessible **only within that block**.

Algorithm

1. Create blocks using `{ }`.
2. Declare variables inside each block.
3. Try accessing them outside.
4. Observe errors.
5. End.

Pseudocode

```
main:
  start block1
     declare a
     print a
  end block1
```

```
  try print a → ERROR
```

Flowchart

```
START
↓
Enter Block
↓
Block variable created
↓
Exit Block
↓
Block variable destroyed
↓
Access outside → ERROR
```

C Program

```c
#include <stdio.h>

int main() {
    {
        int x = 100;
        printf("Inside block 1: x = %d\n", x);
    }

    // printf("%d", x); // ERROR: x not accessible

    {
        int y = 200;
        printf("Inside block 2: y = %d\n", y);
    }

    // printf("%d", y); // ERROR: y not accessible

    printf("Variables inside blocks cannot be accessed outside the
```

```
block.\n");
    return 0;
}
```

Output

```
PS C:\Users\ASUS\Desktop\C Exp> gcc exp4.3.c
PS C:\Users\ASUS\Desktop\C Exp> ./a.exe
Outside inner block: x = 10
Inside inner block: x = 10, y = 20
Inside nested block: x = 10, y = 20, z = 30

Back to main block: x = 10
PS C:\Users\ASUS\Desktop\C Exp>
```

Q4. Declare a static local variable & observe persistence across calls

----

Aim

To study persistence of static local variables across function calls.

Theory

- A **static local variable** retains its value between function calls.
- Unlike normal local variables, static variables are initialized only once.

Algorithm

1. Create function with static variable.
2. Call function repeatedly.
3. Observe value incrementing.
4. End.

Pseudocode

```
function test:
    static x = 0
    x++
    print x

main:
    call test 3 times
```

Flowchart

```
START
↓
Function creates static variable (one-time)
↓
Function call → updated value
↓
Function call → retains previous value
↓
Function call → retains again
↓
END
```

C Program

```c
#include <stdio.h>

void display() {
    static int count = 0;  // Static local variable
    count++;
    printf("Function called %d times\n", count);
}

int main() {
    display();
    display();
    display();
    return 0;
}
```

Output

```
PS C:\Users\ASUS\Desktop\C Exp> gcc exp4.4.c
PS C:\Users\ASUS\Desktop\C Exp> ./a.exe
Calling demo() multiple times:

Static variable = 1, Normal variable = 1
Static variable = 2, Normal variable = 1
Static variable = 3, Normal variable = 1
PS C:\Users\ASUS\Desktop\C Exp>
```