Course: Advanced Programming In C **Semester:**1st term 2026/2027	**Lecturers:** Dr Dina Abdelhafiz

Tutorial 2: Introduction to Variable scope in C Programming

Introduction to Variable Scope in C Programming

What is Variable Scope?

Variable scope refers to the visibility or accessibility of a variable within a program. In C programming, the scope determines where a variable can be used. There are several types of variable scope: **Block Scope**, **Function Scope, and Global Scope**

Summary of Variable Scope

- **Block Scope**: You can only use these variables inside the block (like inside curly braces `{}`) where they are created.
- **Function Scope**: You can only use these variables inside the function where they are created.
- **Global Scope**: You can use these variables in any function throughout the entire program.

Understanding variable scope is important for using memory wisely and writing clear code. Always pick the right scope for your variables to keep your code clear and avoid problems.

1. Block Scope

Variables declared within a block (enclosed by `{}`) are only accessible within that block.

Example 1 on block scope:

```c
#include <stdio.h>

void blockScopeExample() {
    int x = 10;

    if (x > 5) {
        int y = 20;
        printf("Inside if block: x = %d, y = %d\n", x, y);
    }

    // y is not accessible here
    printf("%d", y); // Uncommenting this line will cause a compilation error

    printf("Inside function: x = %d\n", x);
}

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

```
prog.c: In function ‘blockScopeExample’:
prog.c:12:18: error: ‘y’ undeclared (first use in this function)
   12 |     printf("%d", y); // Uncommenting this line will cause a compilation error
      |                  ^
prog.c:12:18: note: each undeclared identifier is reported only once for each function it appears in
```

Example 2 on block scope

```c
#include <stdio.h>

void anotherBlockExample() {
    for (int i = 0; i < 3; i++) {
        int j = i * 2;
        printf("Inside loop: i = %d, j = %d\n", i, j);
    }

    // j is not accessible here
     printf("%d", j); // Uncommenting this line will cause a compilation error
}

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

```
prog.c: In function ‘anotherBlockExample’:
prog.c:10:19: error: ‘j’ undeclared (first use in this function)
   10 |      printf("%d", j); // Uncommenting this line will cause a compilation error
      |                   ^
prog.c:10:19: note: each undeclared identifier is reported only once for each function it appears in
```

2. Function Scope

Variables declared within a function are accessible only within that function.

Example 1:

```c
#include <stdio.h>

void functionScopeExample() {
    int num = 100;
    printf("Inside function: num = %d\n", num);
}

int main() {
    functionScopeExample();
    printf("%d", num); // Uncommenting this line will cause a compilation error
    return 0;
}
```

```
prog.c: In function ‘main’:
prog.c:10:18: error: ‘num’ undeclared (first use in this function)
   10 |     printf("%d", num); // Uncommenting this line will cause a compilation error
      |                   ^~~
prog.c:10:18: note: each undeclared identifier is reported only once for each function it appears in
```

Example 2:

```c
void calculateSum() {
    int a = 5, b = 10;
    int sum = a + b;
    printf("Sum inside function: %d\n", sum);
}

int main() {
    calculateSum();
    printf("%d", sum); // Uncommenting this line will cause a compilation error
    return 0;
}
```

```
prog.c: In function ‘main’:
prog.c:11:18: error: ‘sum’ undeclared (first use in this function)
   11 |     printf("%d", sum); // Uncommenting this line will cause a compilation error
      |                   ^~~
prog.c:11:18: note: each undeclared identifier is reported only once for each function it appears in
```

3. Global Scope

Global variables can be accessed from any function in the program. However, overusing global variables can lead to code that is hard to maintain.

Example 1:

```c
#include <stdio.h>

float balance = 500.0; // Global variable

void deposit(float amount) {
    balance += amount;
}

void withdraw(float amount) {
    balance -= amount;
}

int main() {
    printf("Initial Balance: %.2f\n", balance);
    deposit(200.0);
    printf("Balance after deposit: %.2f\n", balance);
    withdraw(100.0);
    printf("Balance after withdrawal: %.2f\n", balance);
    return 0;
}
```

```
Initial Balance: 500.00
Balance after deposit: 700.00
Balance after withdrawal: 600.00
```

Global Scope Shadowing

- Shadowing happens when a local variable has the same name as a global variable.
- The local variable is used instead of the global one when they share the same name.
- To reach the global variable, you can use different naming strategies to avoid confusion.

Example 1: Basic Shadowing

In this example, the local variable `value` in `displayValue()` shadows the global `value`. Inside the function, the local `value` is used, while the global `value` remains unchanged.

```c
#include <stdio.h>

int value = 10; // Global variable

void displayValue() {
    int value = 20; // Local variable shadows the global variable
    printf("Inside function: value = %d\n", value);
}

int main() {
    printf("In main: value = %d\n", value);
    displayValue();
    printf("After function call: value = %d\n", value);
    return 0;
}
```

```
In main: value = 10
Inside function: value = 20
After function call: value = 10
```

2: Here, the local `count` is incremented, but it does not affect the global `count`. The global variable remains unchanged.

```c
int count = 5; // Global variable

void modifyCount() {
    int count = 10; // Local variable shadows the global variable
    count++; // Modifies the local variable
    printf("Inside modifyCount: count = %d\n", count);
}

int main() {
    printf("Before modifyCount: count = %d\n", count);
    modifyCount();
    printf("After modifyCount: count = %d\n", count);
    return 0;
}
```

```
Before modifyCount: count = 5
Inside modifyCount: count = 11
After modifyCount: count = 5
```

Happy Coding!