Lecture 10: continuation for Lecture 9

Scopes:

In programming, “**scope”** refers to the part of the code where a variable or function is accessible. It defines the visibility and lifetime of variables. Scopes are crucial for controlling how and where data is accessed, ensuring modularity and preventing naming conflicts.

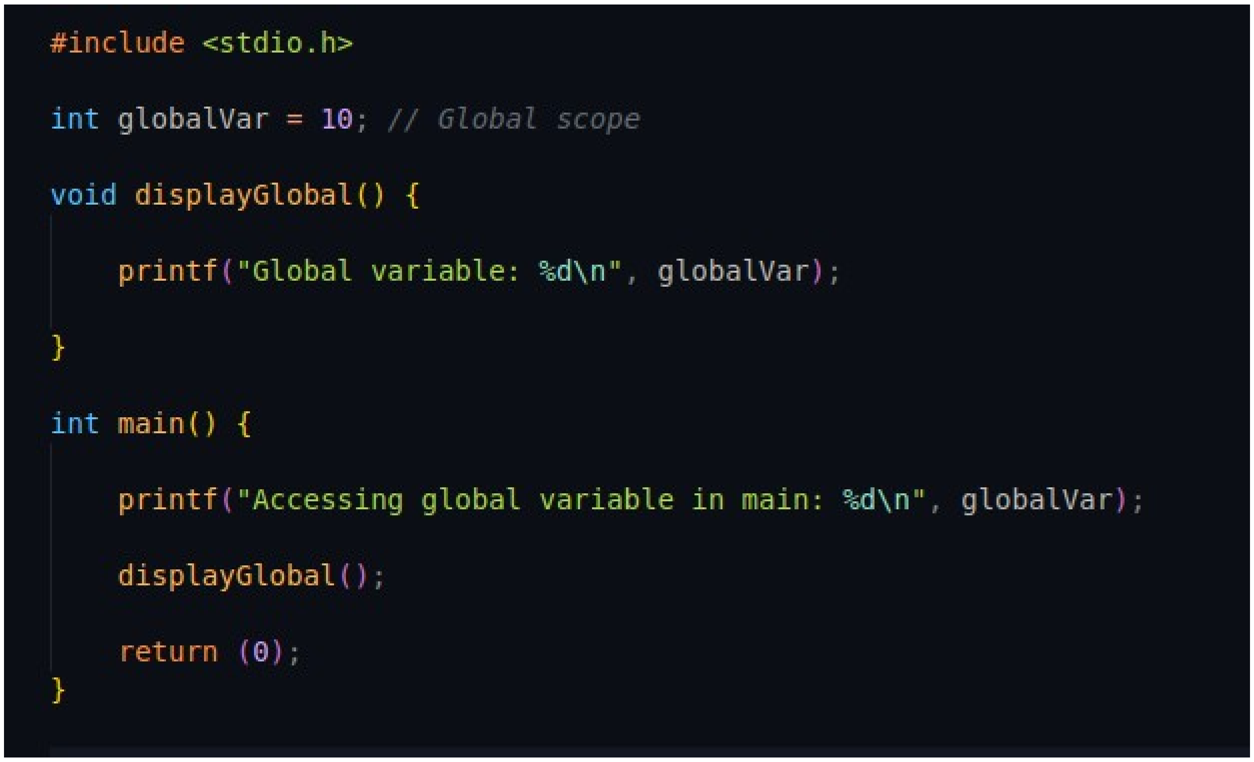
In C, scopes are primarily categorized as follows:

---

1. Global Scope

- Variables declared outside all functions are in the global scope.
- Accessible throughout the program, from any function.
- Lifetime spans the entire program's execution.

Example:



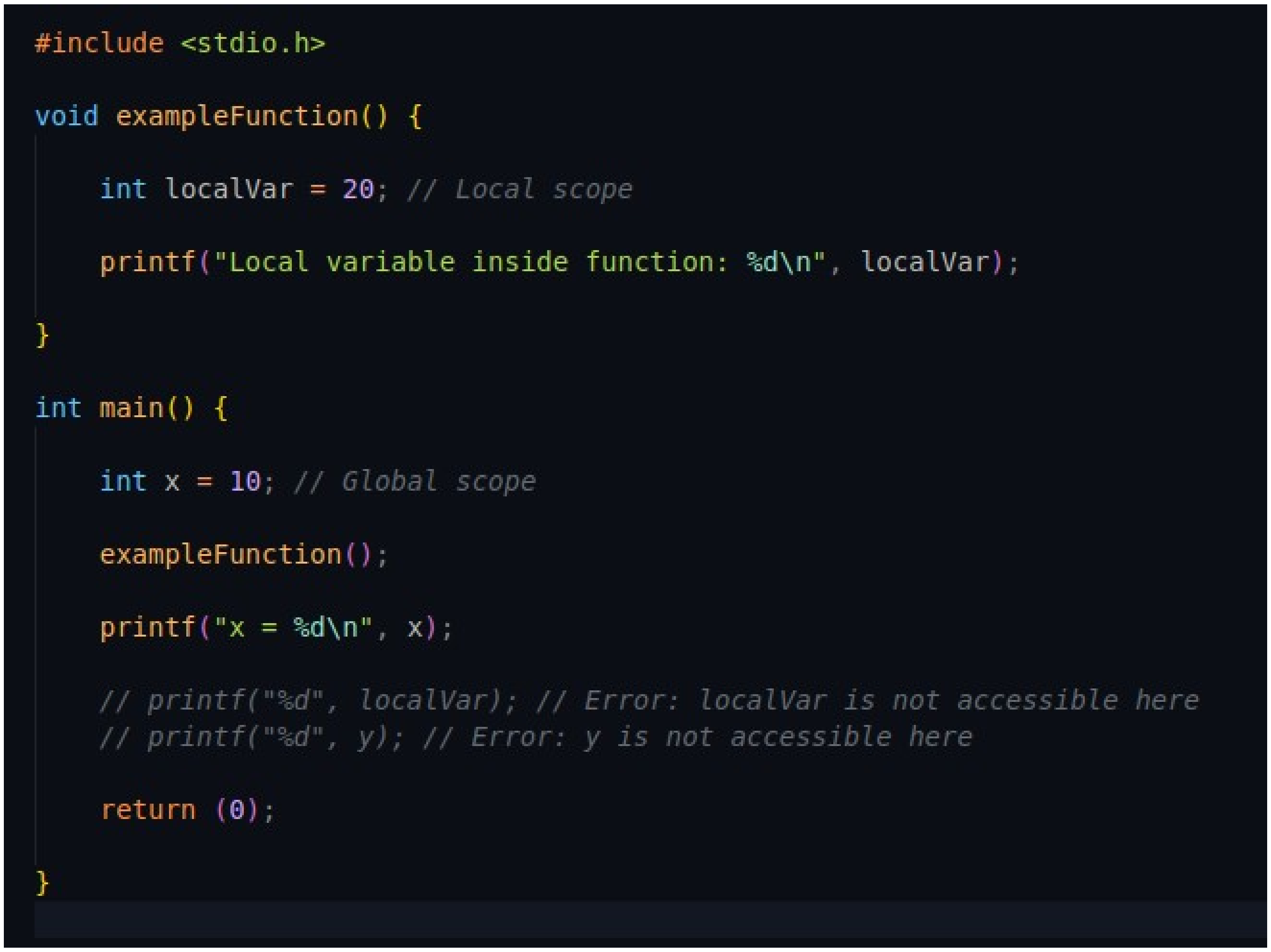
```c
#include <stdio.h>

int globalVar = 10; // Global scope

void displayGlobal() {

    printf("Global variable: %d\n", globalVar);

}

int main() {

    printf("Accessing global variable in main: %d\n", globalVar);

    displayGlobal();

    return (0);
}
```

---

2. Local Scope

- Variables declared inside a function or block are in the local scope.
- Only accessible within that specific function or block.
- Destroyed when the function or block execution ends.

Example:



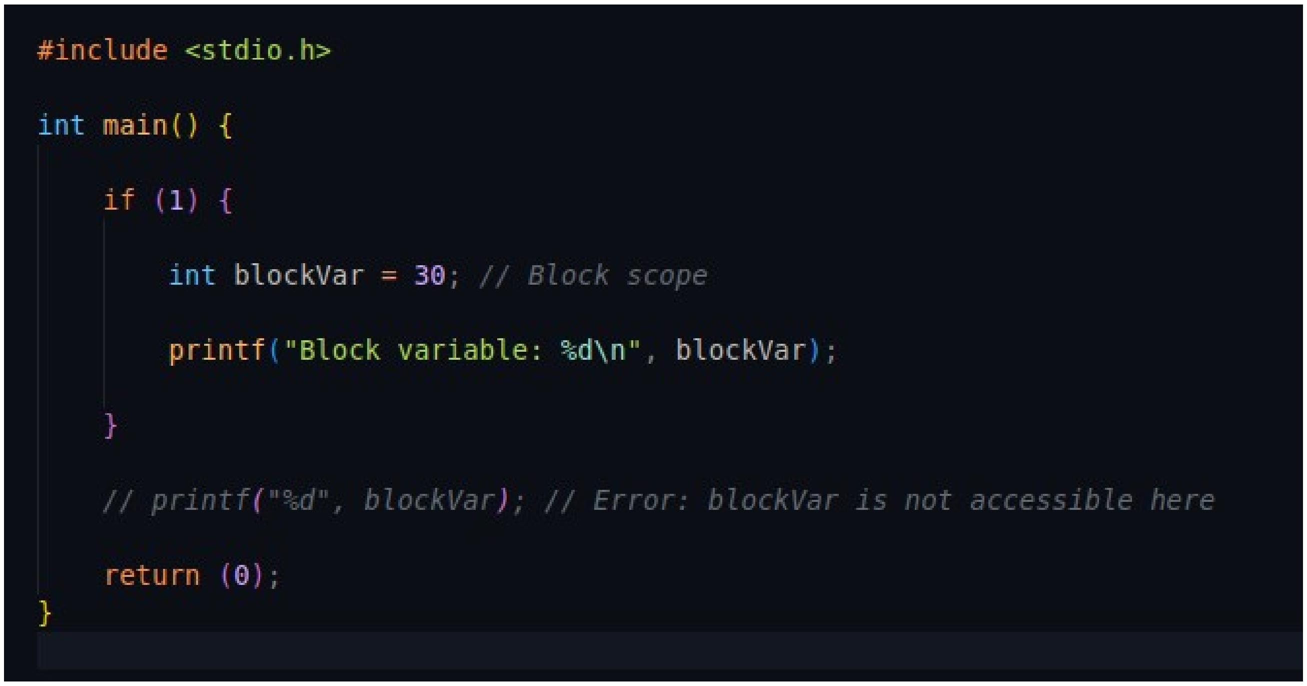
```c
#include <stdio.h>

void exampleFunction() {

    int localVar = 20; // Local scope

    printf("Local variable inside function: %d\n", localVar);

}

int main() {

    int x = 10; // Global scope

    exampleFunction();

    printf("x = %d\n", x);

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

    return (0);

}
```

---

3. Block Scope

- Variables declared inside a block (e.g., `{}`) are in block scope.
- Accessible only within that block and destroyed after the block is exited.

Example:



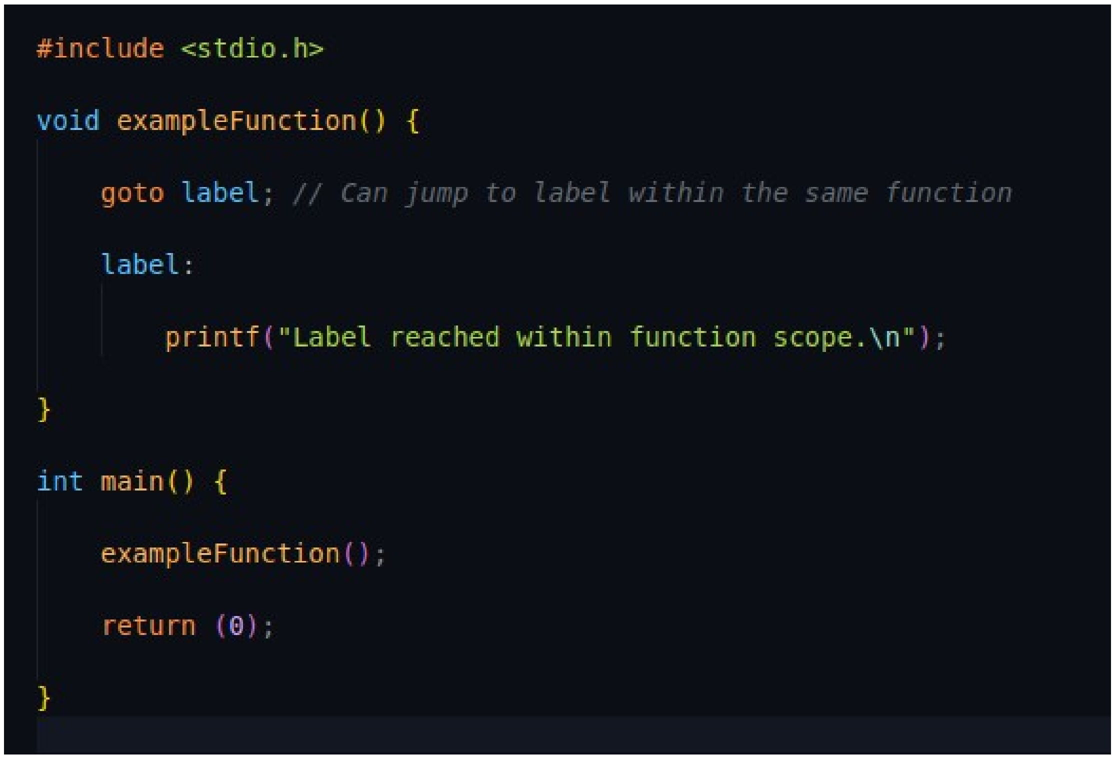
```c
#include <stdio.h>

int main() {

    if (1) {

        int blockVar = 30; // Block scope

        printf("Block variable: %d\n", blockVar);

    }

    // printf("%d", blockVar); // Error: blockVar is not accessible here

    return (0);
}
```

---

4. Function Scope

- Labels (used with `goto`) have function scope.
- They can only be used within the function where they are defined.

Example:



```c
#include <stdio.h>

void exampleFunction() {

    goto label; // Can jump to label within the same function

    label:

        printf("Label reached within function scope.\n");

}

int main() {

    exampleFunction();

    return (0);

}
```

The Stack:

When we run our code and it starts getting executed line by line, each line is mounted in a place of the memory called the “Stack”, which is a segment of a program's memory used for managing function calls, local variables, and control flow. It operates on a **Last In, First Out (LIFO)** principle, meaning the last item added is the first to be removed. But how does this work? Here is a break down of the stack functionality:

1. when we first execute our code, the main( ) function is first mounted on the stack
2. then, all the global variables get hoisted to the top of the program and get declared and mounted on the stack too.
3. After that, the execution of the rest of the code starts, and any block of code encountered whether it is a condition block or a loop block gets mounted on the stack, executed, and then removed from the stack.
4. Any code that gets executed and exited successfully, gets unmounted from the stack directly, and all its data is unmounted with it, meaning that if there is any data inside one block that is declared within the scope of the block, this data won’t be accessible in other scopes for this reason, being unmounted after its execution is done.
5. Is the execution encounters a function call, just like the main( ), it gets mounted on top of the rest of the code, executed with all its blocks and variables, and then unmounted from the stack, and the execution continues.
6. Lastly, after finishing the execution of all the lines of code, the global variables get unmounted from the stack, and after that the main( ) function, returning the exit code that will indicate the success or failure of the program.