Féidearthachtaí as Cuimse
Infinite Possibilities

Semester 2
Week 4 - Tutorial

OLLSCOIL TEICNEOLAÍOCHTA BHAILE ÁTHA CLIATH
DUBLIN
TECHNOLOGICAL UNIVERSITY DUBLIN

Overview

- Types of storage classes
- Lifetime, scope, and visibility of variables
- Auto
- Extern
- Static
- Registers

Variable Scope

- Variable scope refers to the visibility and lifetime of a variable in a program. It determines where a variable can be accessed and how long it exists in memory.
- **Local** variables exist only inside a function.
- **Global** variables are available throughout the program.
- **Static** variables retain their values between function calls.
- Function parameters are temporary and only used inside the function.
- File-scope static variables are limited to the current file only.

Variable Lifetime

- **Variable lifetime** refers to how long a variable **exists in memory** before it is destroyed. It is determined by the **storage class** used when declaring the variable.

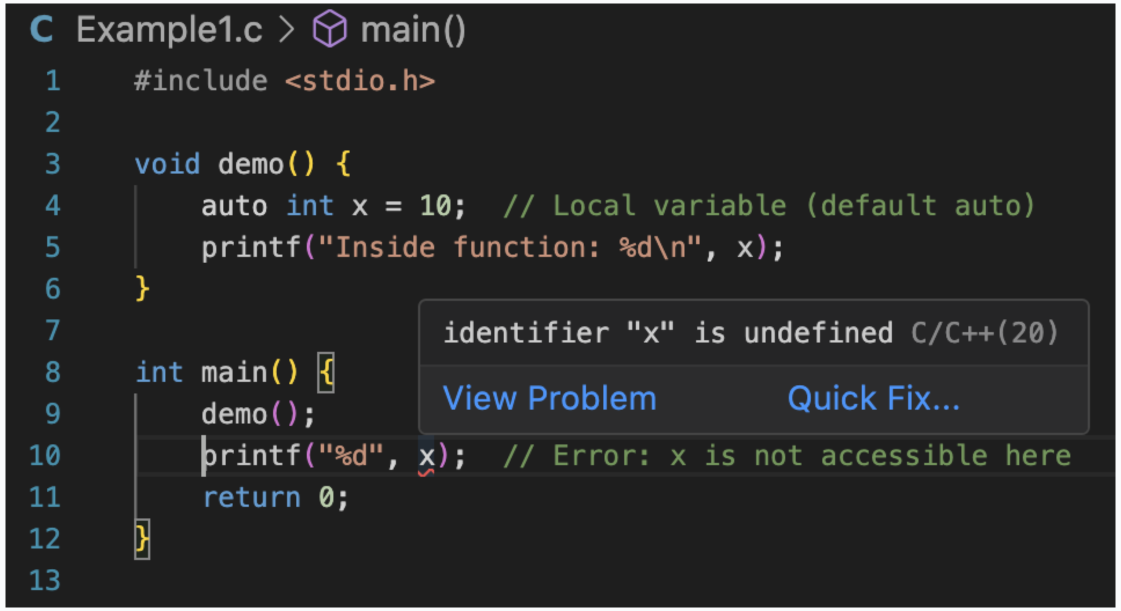
Stack

- The **stack** is a region of memory used to **store function calls, local variables, and control information**. It follows the **LIFO (Last In, First Out)** principle, meaning the most recently added item is the first to be removed.

Automatic Storage (auto)

- **auto** is the default storage class for local variables.
  - **Scope:** Local to the block/function where it's defined.
  - **Lifetime:** Exists only during function execution.
  - **Memory Location:** Stored in stack.

auto example

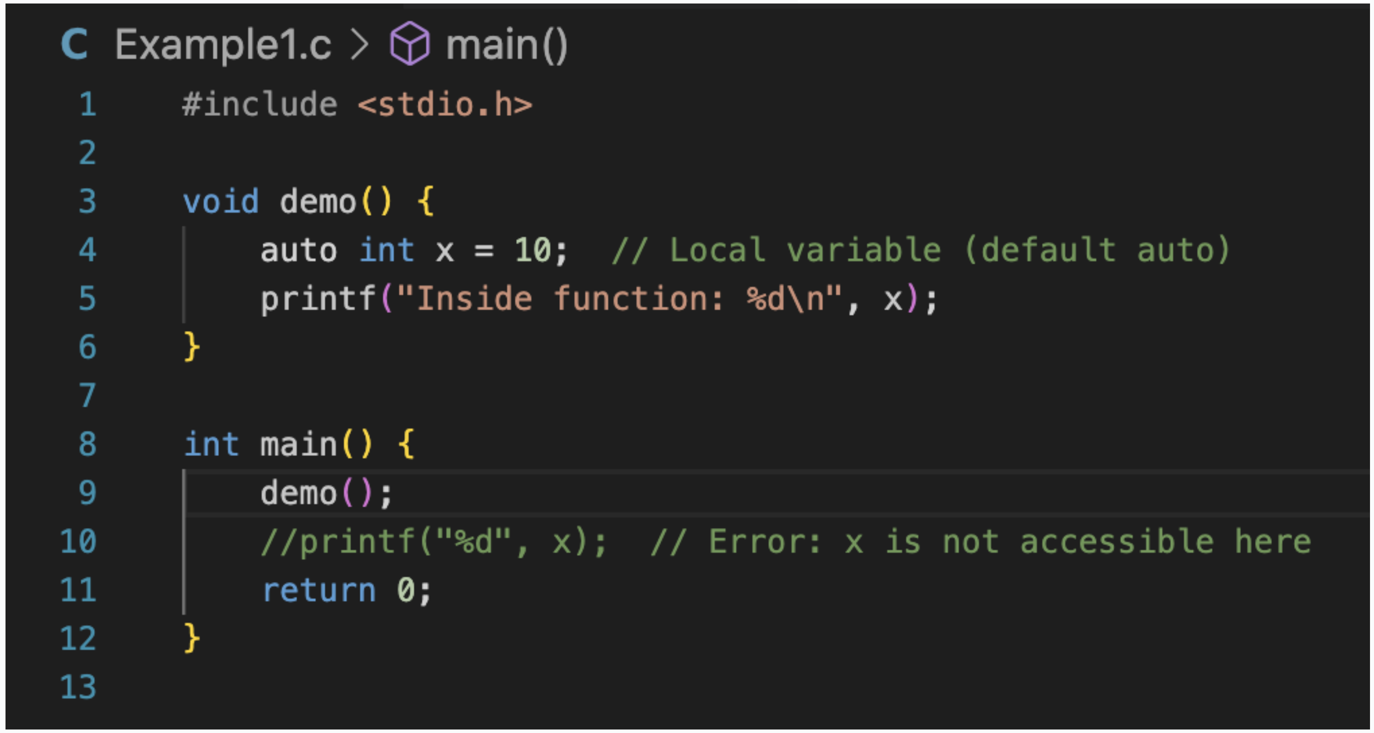


Example1.c > main()

```c
#include <stdio.h>

void demo() {
    auto int x = 10;  // Local variable (default auto)
    printf("Inside function: %d\n", x);
}

int main() {
    demo();
    printf("%d", x);  // Error: x is not accessible here
    return 0;
}
```

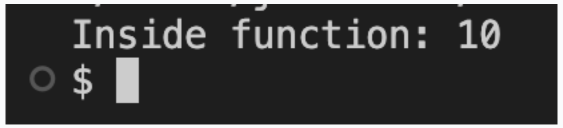
identifier "x" is undefined C/C++(20)

The variable x only exists in the demo() function. It is not accessible in main().



Example1.c > main()

```c
#include <stdio.h>

void demo() {
    auto int x = 10;  // Local variable (default auto)
    printf("Inside function: %d\n", x);
}

int main() {
    demo();
    //printf("%d", x);  // Error: x is not accessible here
    return 0;
}
```

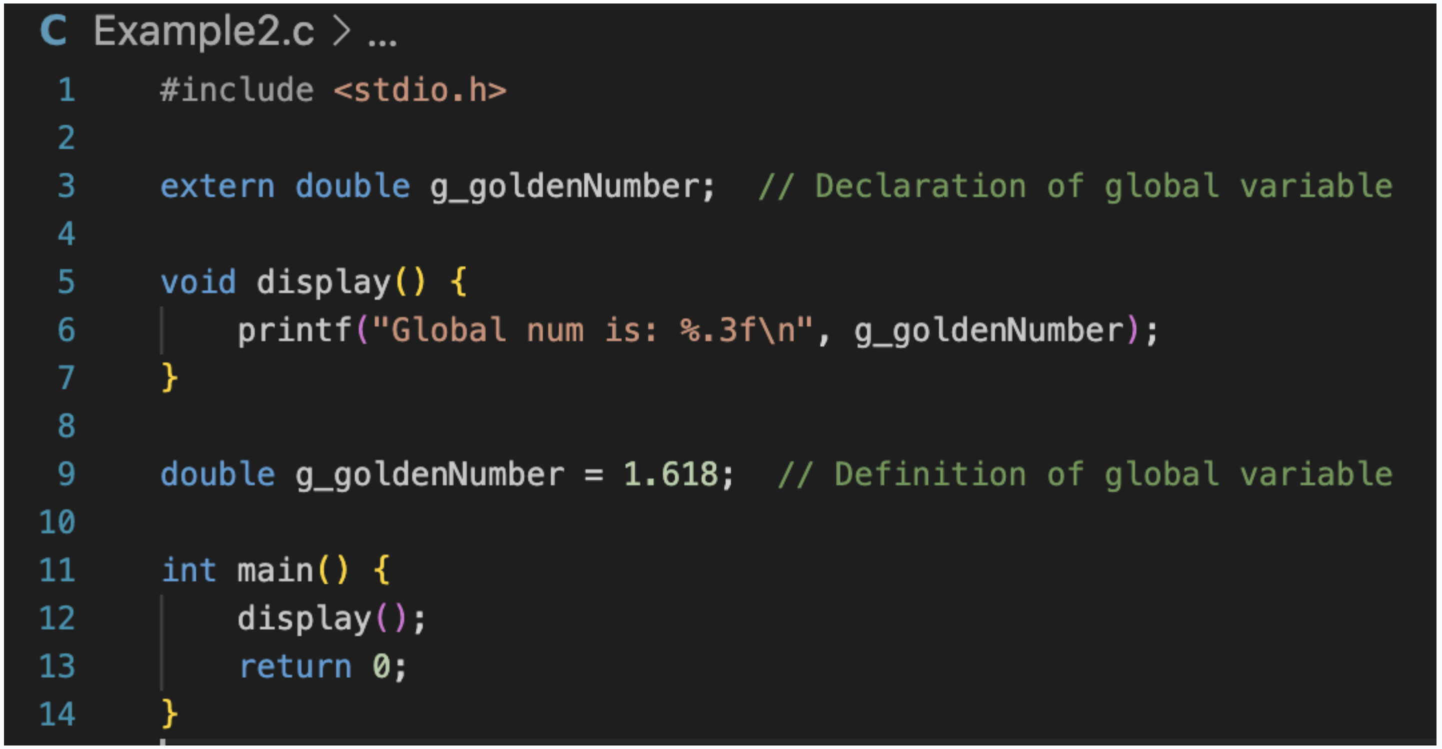


```
Inside function: 10
$ 
```

External Storage (extern)

- **extern** is used for global variables, which can be shared across multiple files.
  - **Scope:** Available throughout the program.
  - **Lifetime:** Exists throughout program execution.
  - **Memory Location:** Stored in global memory (Data Segment).

Extern example



C Example2.c > ...

```c
#include <stdio.h>

extern double g_goldenNumber;  // Declaration of global variable

void display() {
    printf("Global num is: %.3f\n", g_goldenNumber);
}

double g_goldenNumber = 1.618;  // Definition of global variable

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

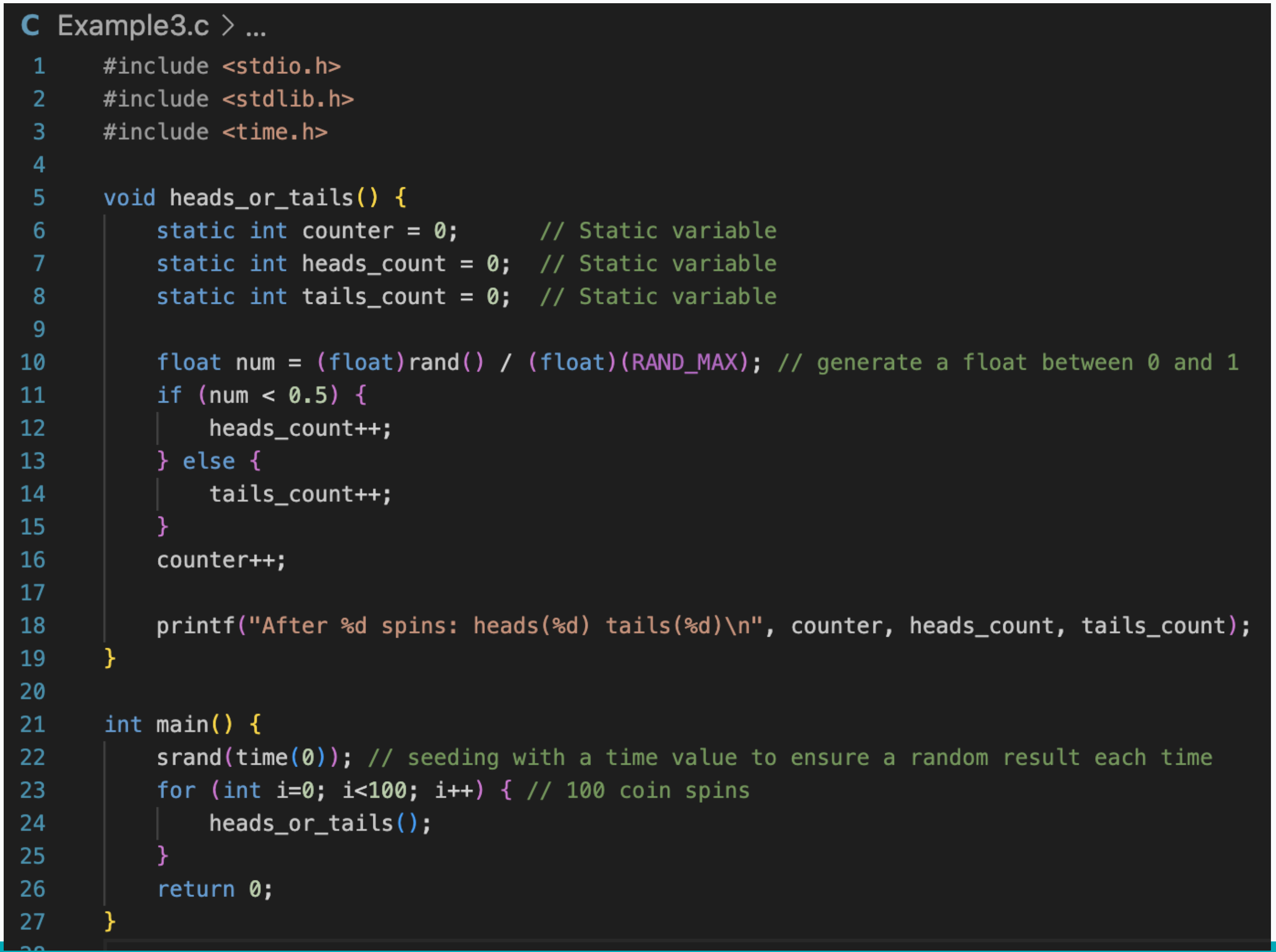
Note: Careful consideration should be given to the use of global variables - avoid when possible

- In this program we define a global variable to store our golden number (1.618)
- In this program we are using a programming convention where we declare all global variables starting with g_ . This will make it easier to identify global variables in the code

Static Storage (static)

- Static retains its value between function calls.
  - **Scope:** Local to the function or file where declared.
  - **Lifetime:** Persists throughout program execution.
  - **Memory Location:** Stored in global memory (Data Segment).

Static example



C Example3.c > ...

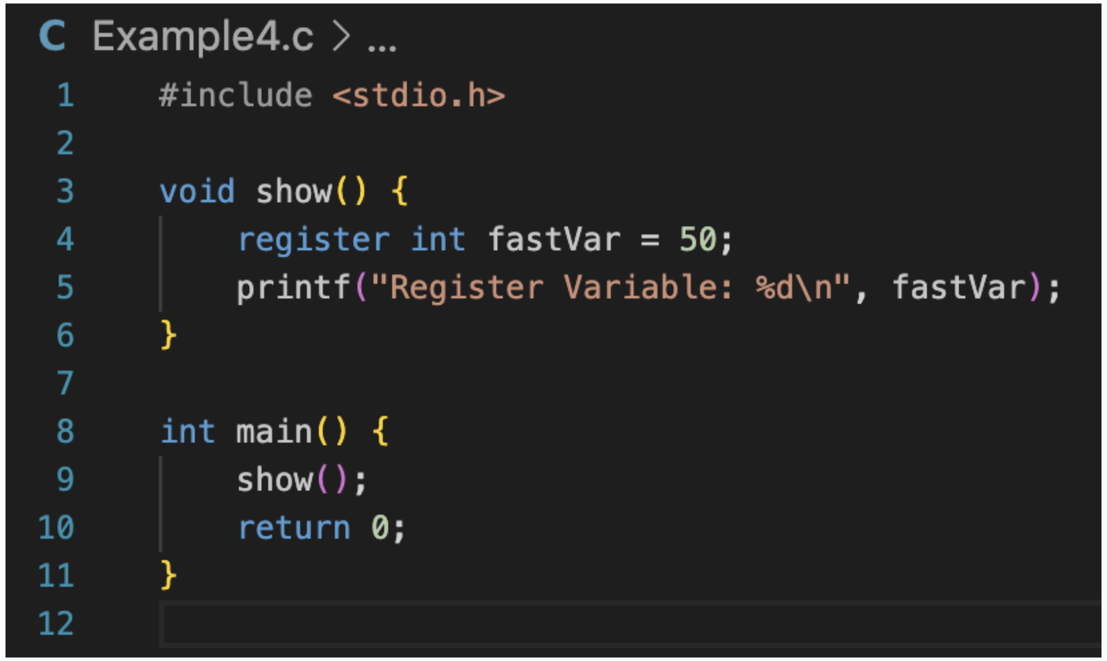
```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void heads_or_tails() {
    static int counter = 0;      // Static variable
    static int heads_count = 0;  // Static variable
    static int tails_count = 0;  // Static variable

    float num = (float)rand() / (float)(RAND_MAX); // generate a float between 0 and 1
    if (num < 0.5) {
        heads_count++;
    } else {
        tails_count++;
    }
    counter++;

    printf("After %d spins: heads(%d) tails(%d)\n", counter, heads_count, tails_count);
}

int main() {
    srand(time(0)); // seeding with a time value to ensure a random result each time
    for (int i=0; i<100; i++) { // 100 coin spins
        heads_or_tails();
    }
    return 0;
}
```

- Using static variables to store the counter and the heads / tails results.
- The static variables are only initialized once.
- When the function completes the static variables are not destroyed with the function, these values are available when the function is called again.

Register Storage (register)

- Register Storage suggests storing the variable in a CPU register (for fast access).
  - **Scope:** Local to the function/block.
  - **Lifetime:** Exists only while function is active.
  - **Memory Location:** CPU registers (if available); otherwise, it behaves like auto.

Register Storage example


Example4.c > ...

```c
#include <stdio.h>

void show() {
    register int fastVar = 50;
    printf("Register Variable: %d\n", fastVar);
}

int main() {
    show();
    return 0;
}

```

- **Note:** The register keyword is a hint to the compiler. Modern compilers optimize variables automatically, so register is rarely needed.

Summary

Storage Class	Scope	Lifetime	Memory Location	Example Use
auto (default)	Local	Until function ends	Stack	Local variables
extern	Global	Entire program	Data Segment	Shared global variables
static	Local to function/file	Entire program	Data Segment	Retain values between function calls
register	Local	Until function ends	CPU Register (if available)	Fast-access variables

Usage

- Use **auto** (default) for **local variables** unless you need persistence.
- Use **extern** for **global variables** that need to be accessed in multiple files.
- Use **static** if a variable needs to **retain its value** across function calls.
- Use **register** for frequently accessed variables (though modern compilers optimize this automatically).
- Wherever possible variables are to be kept at the narrowest scope.

Mandatory Question – in class solution

- Passing 1-D Array. Write a program that asks the user to enter 5 numbers from standard input. Pass the array to a function where the function checks each number in the array if it is even or odd. Your function should display each number and state whether it is even or odd. Finally, your function should calculate the total number of even numbers only and return this number to your main() and display it.

Questions