Short-term Hands-on Supplementary Course on C Programming

SESSION 8: Recursive Functions and Variable Scope

KARTHIK D
NIVEDHITHA D

Time: 6:30 - 8:00 PM
Date: June 18th, 2022
Location: Online

Agenda

1. Administrative Instructions
2. What are functions?
3. Recursive Functions
    a. Understanding recursion using the Fibonacci Series
    b. Iteration vs. Recursion: DEMO
    c. WHY and When to use Recursion?
4. Variable Scopes
5. Storage Classes
6. Modular Programming
7. Tutorial: Implementing Recursive Functions
8. Next Session

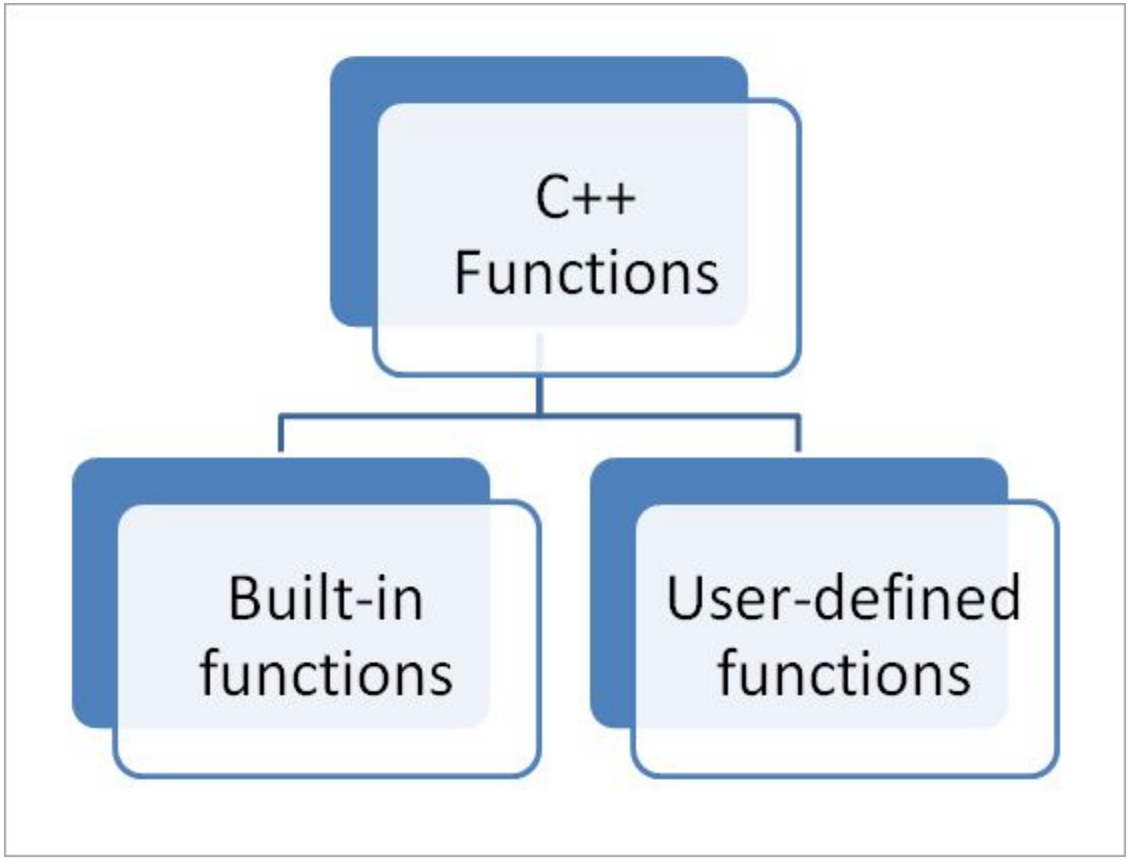
Administrative Instructions

- Please fill out the feedback form - will be shared in the chat
- Join us on Microsoft Teams,

  Team Code: **rzlaicv**

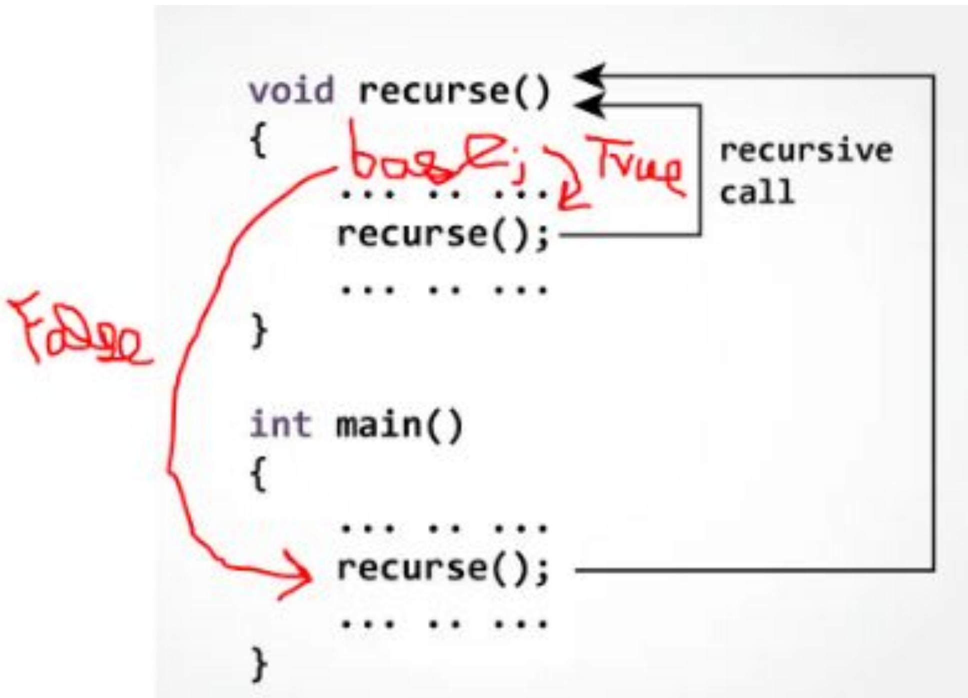
GITHUB REPOSITORY! ⭐🌟

What are Functions?



At times, a certain portion of code has to be used many times. Instead of re-writing the codes many times, it is better to put them into a "**subroutine**", and "call" this "subroutine" many time - for ease of **maintenance** and **understanding**. This subroutine is called a function (in C/C++).

Recursive Functions

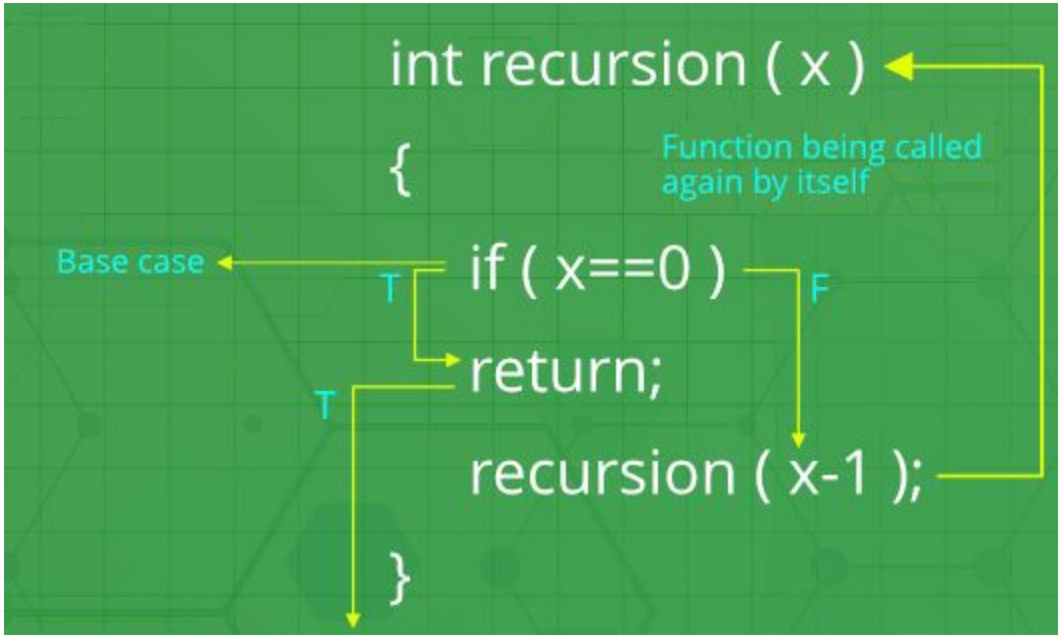


Structure of a Recursive Problem

Every recursive algorithm involves at least **two** cases:

- base case: The simple case; an occurrence that can be answered directly; the case that recursive calls reduce to.

- recursive case: a more complex occurrence of the problem that cannot be directly answered, but can be described in terms of smaller occurrences of the same problem.



Three Musts of Recursion

1. Your code must have a case for all valid inputs

2. You must have a base case that makes no recursive calls

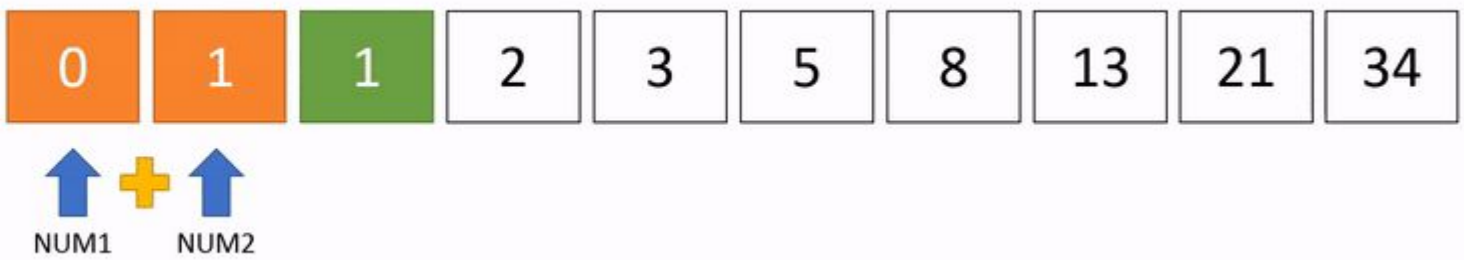
3. When you make a recursive call it should be to a simpler instance and make forward progress towards the base case.

There is a "recursive leap of faith"

The Fibonacci Series

Fibonacci Series

A series of numbers in which each number (*Fibonacci number*) is the sum of the two preceding numbers.

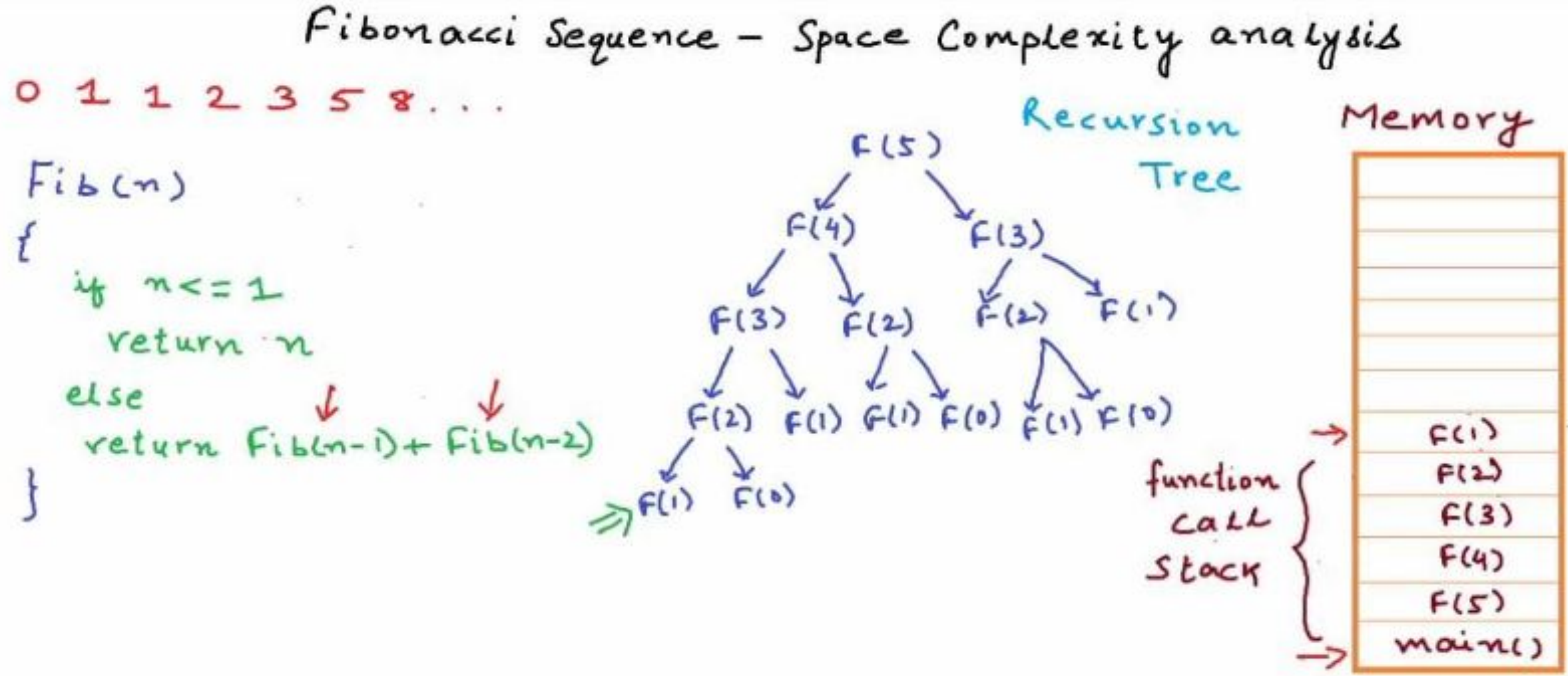


Recurrence Relation

In general, the Fibonacci numbers can be defined by the rule

$$\mathrm{Fib}(n) = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2) & \text{otherwise.} \end{cases}$$

Call Stack for Fib Series



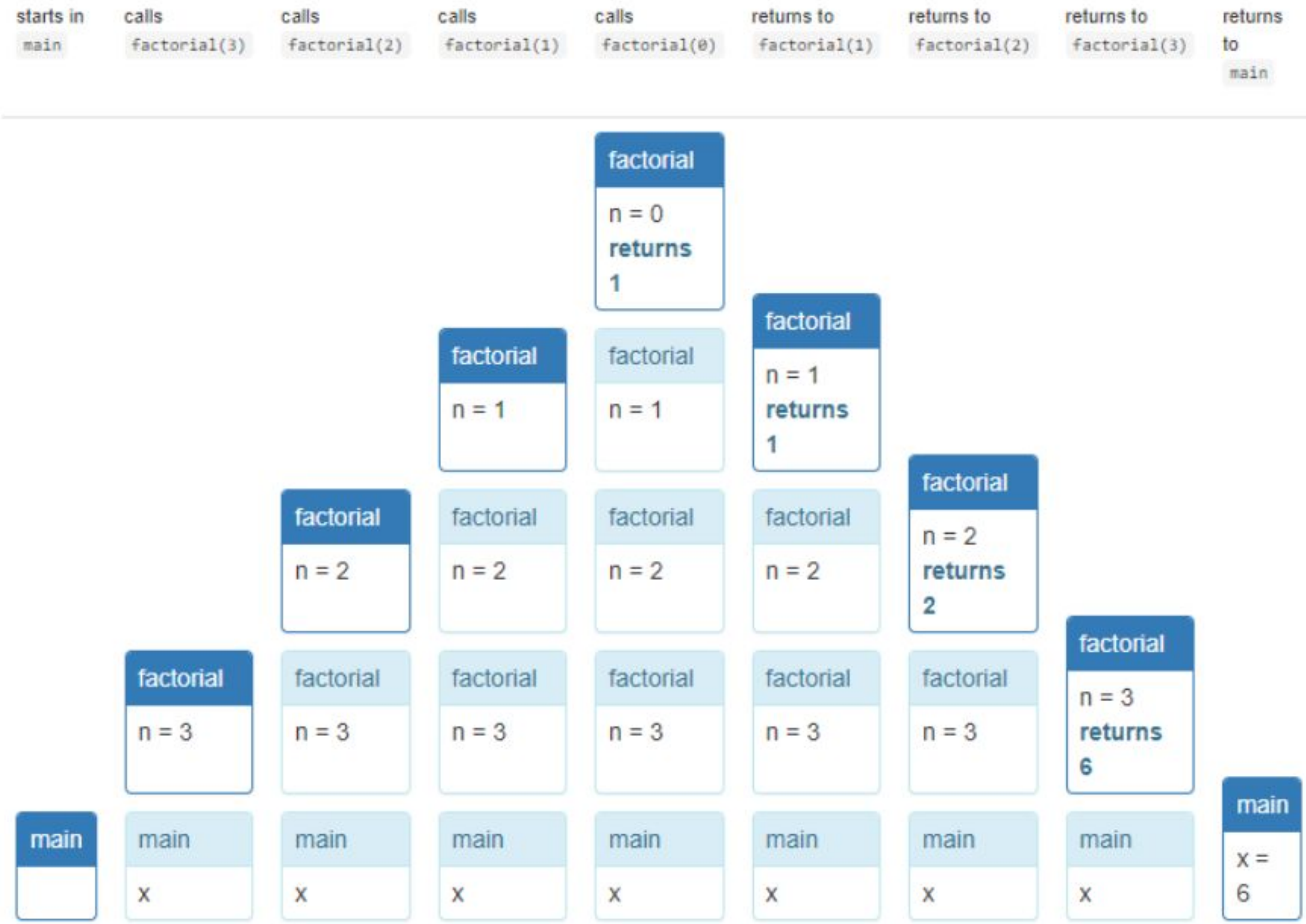
Iteration vs. Recursion

Product	Recurrence relation
$n! = n \times (n-1) \times \cdots \times 2 \times 1 = \prod_{k=1}^{n} k$ (where the empty product equals multiplicative identity *1*)	$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! \times n & \text{if } n > 0 \end{cases}$

$$\begin{aligned} 1! &= 1 \\ 2! &= 2 \times 1 = 2 \\ 3! &= 3 \times 2 \times 1 = 6 \\ 4! &= 4 \times 3 \times 2 \times 1 = 24 \\ 5! &= 5 \times 4 \times 3 \times 2 \times 1 = 120 \end{aligned}$$

Factorial Demo

Factorial Call Stack



Why use Recursion?

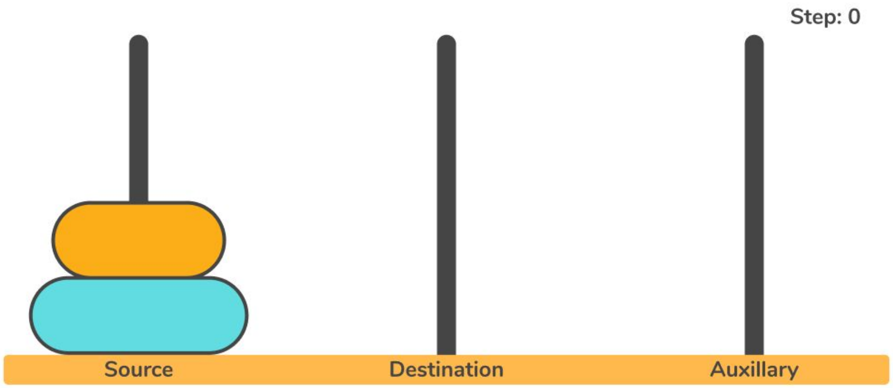
1. **Great style**: for programming solutions to naturally recursive problems or recursive data.
2. **Powerful tool** to elegantly solve certain problems whose iterative solutions are messy or impossible to construct, e.g. Tower of Hanoi.
3. **Master of flow of control** as it is:
   - **Safe from bugs**. Recursive code is simpler and often uses immutable variables and immutable objects.
   - **Easy to understand**. Recursive implementations for naturally recursive problems and recursive data are often shorter and easier to understand than iterative solutions.
   - **Ready for change**. Recursive code is also naturally re-entrant, which makes it safer from bugs and ready to use in more situations.

When to use Recursion?

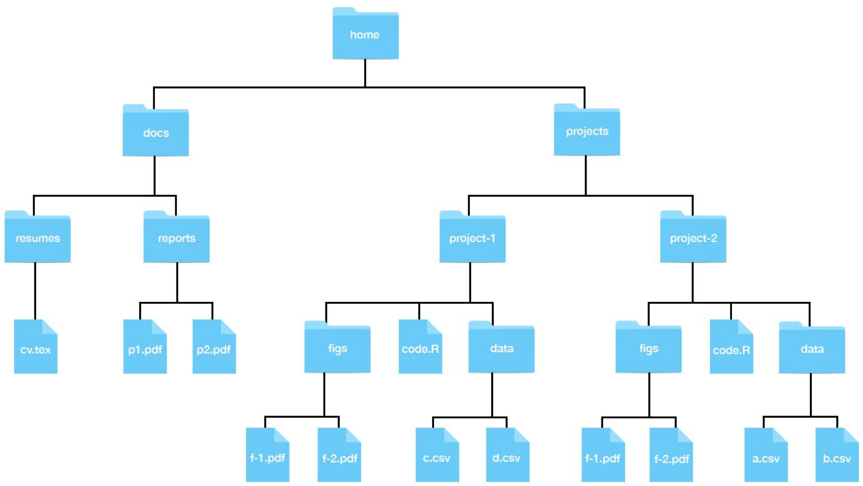
Recursive Problems

Recurrence relation

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! \times n & \text{if } n > 0 \end{cases}$$



Recursive Data



Variable Scope & Lifetime

- The **scope** of a variable is the part of the program for which the declaration is in effect.
- C uses **lexical**/**static** scoping — variable are, by default, known only within their block

- The **lifetime** of a variable is the time period in which it allocated memory is guaranteed to be valid.
- They can be synonymously called "**allocation method**" or "**storage duration**"

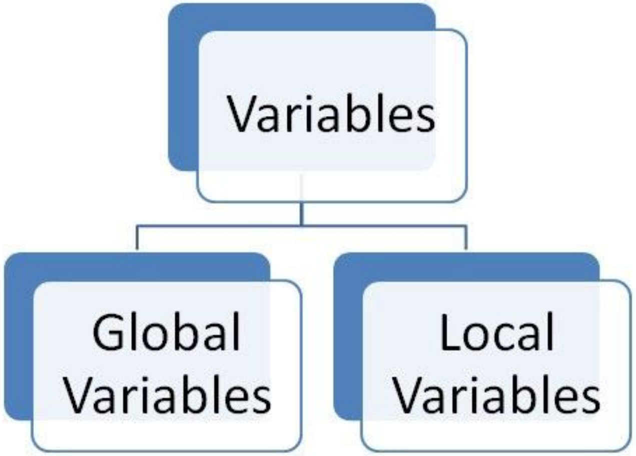
Variable Scope & Lifetime

Scope can be (very broadly):

- Local
- Global

Lifetime can be:

- Static
- Automatic
- Dynamic (heap)



“Return” Multiple Values from Functions

Use Global Variables

```
#include<stdio.h>

int prev;
int next;

void get_prev_and_next(int n){
    prev = n-1;
    next = n+1;
}
```

User Reference Args

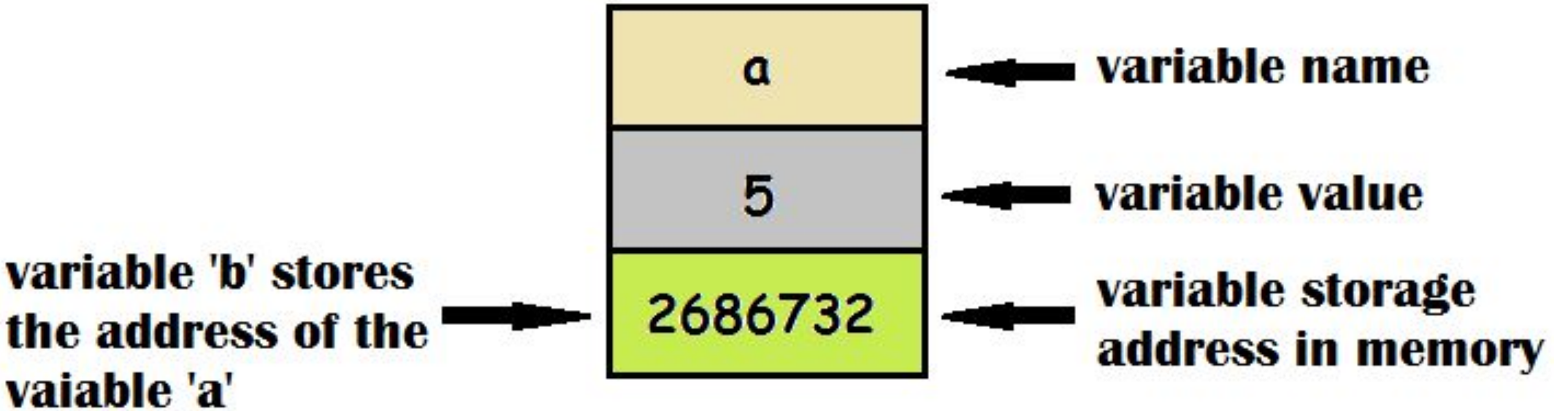
```
#include<stdio.h>

void get_prev_and_next(int n, int *prev, int *next){
    *prev = n - 1;
    *next = n + 1;
}
```

Storage Classes in C

Storage Specifier	Storage	Initial value	Scope	Life
auto	stack	Garbage	Within block	End of block
extern	Data segment	Zero	global Multiple files	Till end of program
static	Data segment	Zero	Within block	Till end of program
register	CPU Register	Garbage	Within block	End of block

Variable Storage in C

```
int a = 5;
int *b;
b = &a;
```



TUTORIAL

Recursion with Pointers

```c
/*
In each recursion step, I am heading towards the end of the string.
Finally (base case), I reach the end of the string (\0)
The recursive calls start returning, and I start storing those positions into `rev_string` (back to front)

For instance: (string: hey) rev_string-> yeh
- ("hey\0", 0, "", 0)
    - ("hey\0", 1, "", 0)
        - ("hey\0", 2, "", 0)
            - ("hey\0", 3, "", 0)
              HIT BASE CASE - START RETRACING
            - ("hey\0", 3, "", 0)
        - ("hey\0", 2, "y", 1)
    - ("hey\0", 1, "ye", 2)
- ("hey\0", 1, "yeh", 3)
*/

void reverse_string_rec(const char string[], int posn, char rev_string[], int *size){
    if(string[posn]=='\0'){
        return;
    }
    // h -> e -> y : y -> e -> h
    rev_string[*size] = string[posn];
    *size = *size + 1;
    reverse_string_rec(string, posn+1, rev_string, size);
    /* no profceses */
    return;
}
```

TUTORIAL

Tail-Call-Recursion - Optimizing Space

Tail Recursive vs Not Tail Recursive

```
(factorial 6)
(fact-iter   1 1 6)
(fact-iter   1 2 6)
(fact-iter   2 3 6)
(fact-iter   6 4 6)
(fact-iter  24 5 6)
(fact-iter 120 6 6)
(fact-iter 720 7 6)
720
```

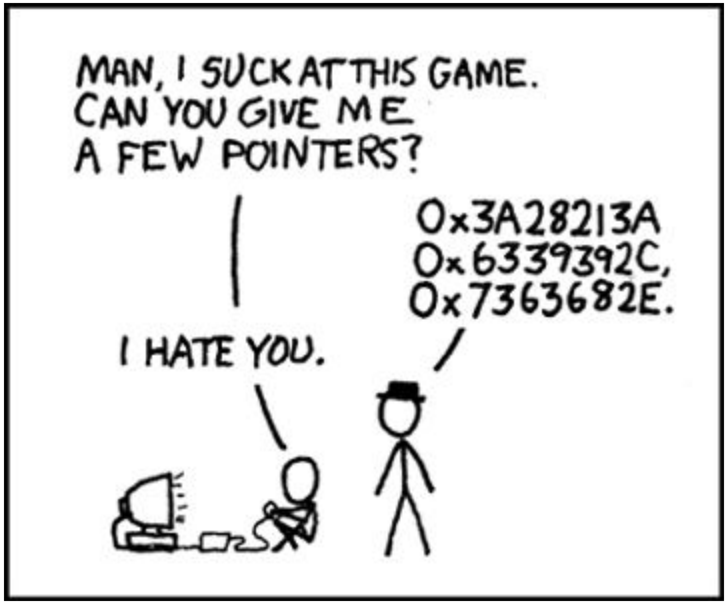
Only uses the registers

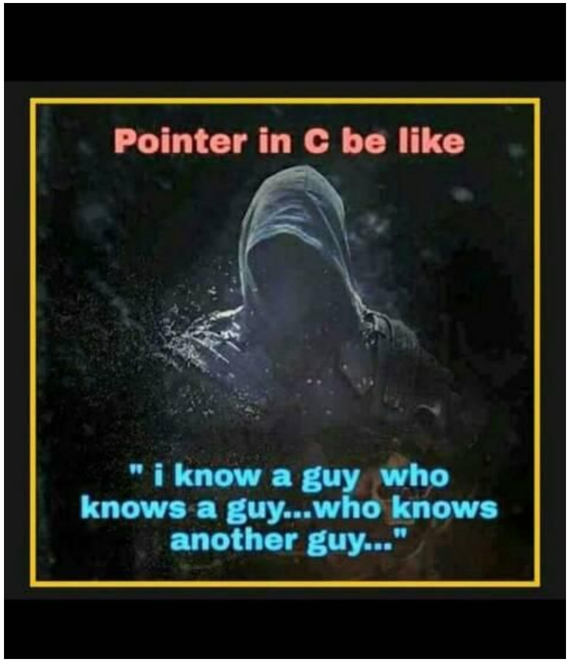
```
(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

Uses the stack

Next Session

POINTERS!!!





Any Questions