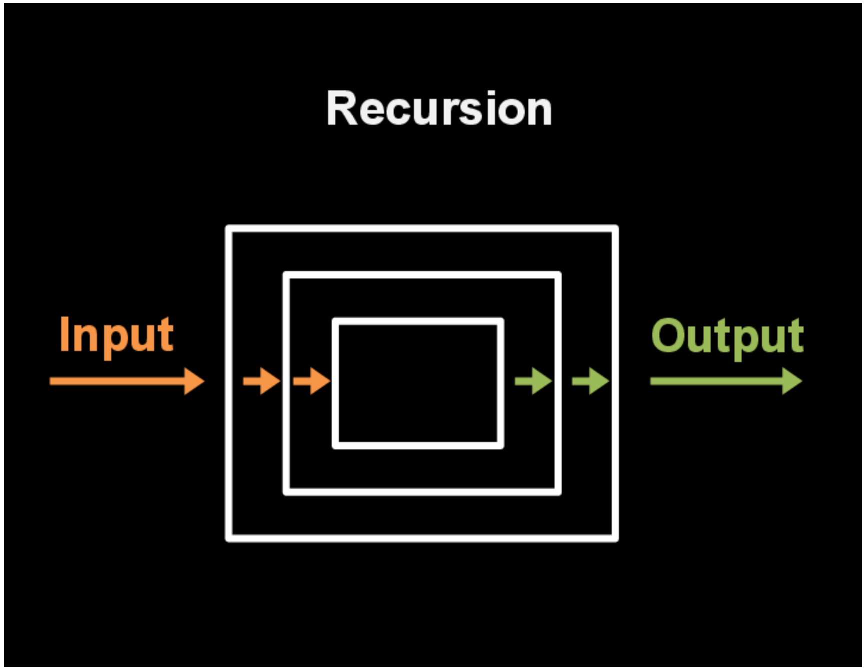


Week 5 - Lecture 3

Variable Scope and Recursion

Edited by: Heshan Du

Autumn 2024

Overview

- **Header file**
- Recursive function
- Scope of variables

Header File

Each standard library has a corresponding header containing the function prototypes for all the functions in that library and definitions of various data types and constants needed by those functions.

Header	Explanation
<assert.h>	Contains information for adding diagnostics that aid program debugging.
<ctype.h>	Contains function prototypes for functions that test characters for certain properties, and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa.
<errno.h>	Defines macros that are useful for reporting error conditions.
<float.h>	Contains the floating-point size limits of the system.
<limits.h>	Contains the integral size limits of the system.
<locale.h>	Contains function prototypes and other information that enables a program to be modified for the current locale on which it's running. The notion of locale enables the computer system to handle different conventions for expressing data such as dates, times, currency amounts and large numbers throughout the world.
<math.h>	Contains function prototypes for math library functions.
<setjmp.h>	Contains function prototypes for functions that allow bypassing of the usual function call and return sequence.

Header File (2)

- Standard libraries e.g., #include <stdio.h>
- Your own libraries e.g., #include “add.h”

```c
#include<stdio.h>

int add(int i, int y);

int main(void)
{
    printf("%d\n", add(1, 2));

    return 0;
}

int add(int i, int y)
{
    return (i + y);
}
```

```c
#include<stdio.h>
#include "add.h"

int main(void)
{
    printf("%d\n", add(1, 2));

    return 0;
}
```

Example: Int Calculator

- Increase reusability of your code.
- Encapsulate (hide) unnecessary information from the user.

```c
#include<stdio.h>

int addInt(int a, int b);
int subtractInt(int a, int b);
int multiplyInt(int a, int b);
int divideInt(int a, int b);

int main(void)
{
    printf("1 + 2 = %2d\n", addInt(1, 2));
    printf("1 - 2 = %2d\n", subtractInt(1, 2));
    printf("1 * 2 = %2d\n", multiplyInt(1, 2));
    printf("1 / 2 = %2d\n", divideInt(1, 2));

    return 0;
}

int addInt(int a, int b)
{
    return (a + b);
}

int subtractInt(int a, int b)
{
    return (a - b);
}

int multiplyInt(int a, int b)
{
    return (a * b);
}

int divideInt(int a, int b)
{
    return (a / b);
}
```

```c
#include <stdio.h>
#include "calculator.h"

int main(void)
{
    printf("1 + 2 = %2d\n", addInt(1, 2));
    printf("1 - 2 = %2d\n", subtractInt(1, 2));
    printf("1 * 2 = %2d\n", multiplyInt(1, 2));
    printf("1 / 2 = %2d\n", divideInt(1, 2));

    return 0;
}
```

University of Nottingham
UK | CHINA | MALAYSIA

Example: Int Calculator (2)

- #include “calculator.h” NOT <calculator.h>
- #ifndef protects you from including the same .h files multiple times.

```
#ifndef CALCULATOR_H
#define CALCULATOR_H

int addInt(int a, int b);
int subtractInt(int a, int b);
int multiplyInt(int a, int b);
int divideInt(int a, int b);

#endif
```

File Name: math_func.h

```
int add (int x, int y)
{
   return x+y;
}

int subtract (int x, int y)
{
   return x-y;
}
```

File Name: my_math.h

```
#ifndef MY_MATH_H
#define MY_MATH_H

int add (int, int);
int subtract (int, int);

#include "math_func.h"
#endif
```

File Name: my_prog.c

```
#include <stdio.h>
#include "my_math.h"

int main (void)
{
   printf("%d\n", add(2, 1));
   printf("%d\n", subtract(2, 1));
   return 0;
}
```

#ifndef … #endif

```
#include <stdio.h>

#define YEARS_OLD 12
#ifndef YEARS_OLD
#define YEARS_OLD 10
#endif

int main()
{
  printf("this is older than %d ", YEARS_OLD);

  return 0;
}
```

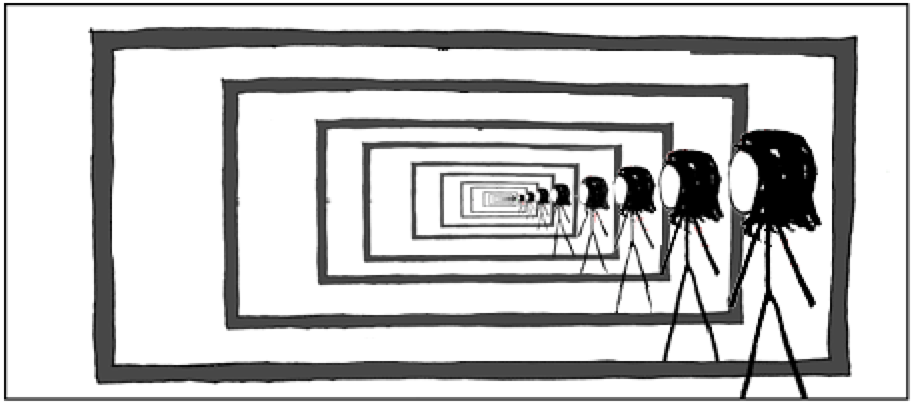
- **The #ifndef directive of the C Programming Language helps in allowing the conditional compilation.**
- **The #ifndef preprocessor only checks If the specific macro is not at all defined with the help of the #define directive.**

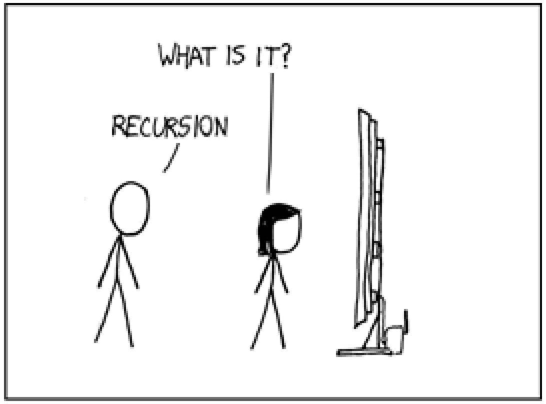
Overview

- Header file
- **Recursive function**
- Scope of variables

Recursion

- A recursive function is a function which calls itself.
- Base case – stopping condition for recursive function, to avoid infinite loop.





Source: http://vaidehijoshi.github.io/blog/2014/12/14/to-understand-recursion-you-must-first-understand-recursion/

Example: Factorials

- A factorial of a number *n* is the product of all integers between *n* and 1.
- e.g., factorial of 5 is 5 * 4 * 3 * 2 * 1
- Factorial base case:
  - if $n \leq 1$

    return 1;
- Factorial how to:
  - return $(n * fact(n - 1))$;

Source: http://vaidehijoshi.github.io/blog/2014/12/14/to-understand-recursion-you-must-first-understand-recursion/

```
int fact(int n)
 {
        if (n < = 1) // base case
                return 1;
        else
                return n*fact(n-1);
 }
```

2 = 2 * 1

fact(2) = 2 * fact(1)

6 = 3 * 2

fact(3) = 3 * fact(2)

24 = 4 * 6

fact(4) = 4 * fact(3)

120 = 5 * 24

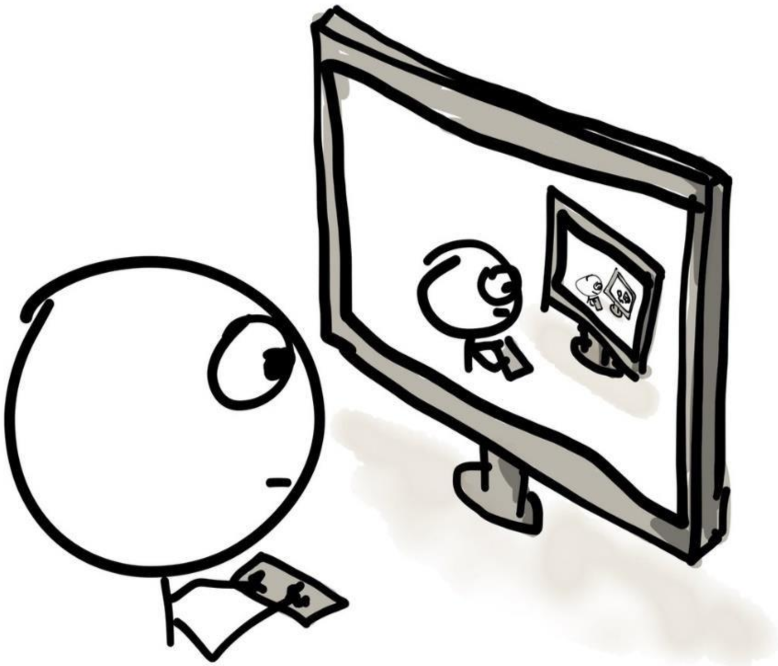
fact(5) = 5 * fact(4)

Example: Factorials (2)

- Starting from the largest, i.e., *x*.
- Each loop iteration reduces x by 1.

```c
/* Iterative implementation of factorial */
int factorial_iter(int x)
{
    if(x < 1)
    {
        printf("Error, x < 1\n");
        exit(1);
    }

    int total = 1;
    int i = 0;
    for(i = x; i > 1; i--)
    {
        total = total * i;
    }
    return total;
}
```



Source: https://derickbailey.com/2015/01/21/hiding-recursion-with-nested-functions-in-javascript/

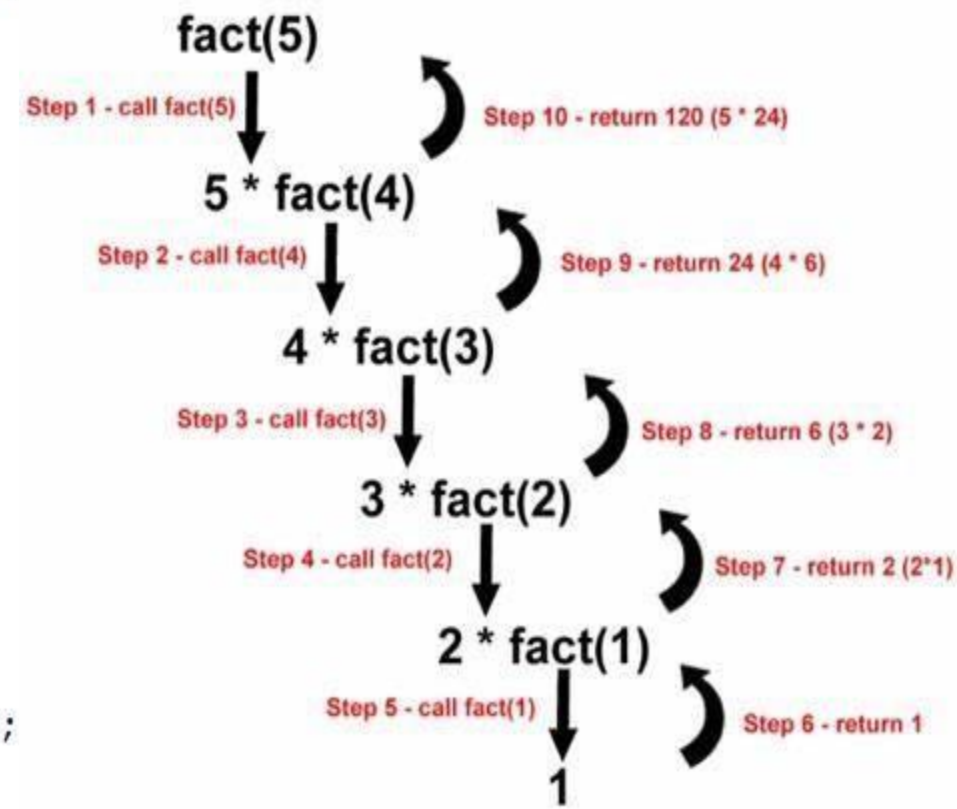
University of Nottingham
UK | CHINA | MALAYSIA

Example: Factorials (3)

- Starting from the largest, i.e., *x*.
- If x is not one, repeatedly call itself with x – 1.

```c
/* Recursive implementation of factorial */
int factorial_rec(int x)
{
    if(x < 1)
    {
        printf("Error, x < 1\n");
        exit(1);
    }

    if(x == 1)
    {
        return 1;
    }
    else
    {
        return factorial_rec(x - 1) * x;
    }
}
```



Recursion vs. iteration

- Recursion is a very important tool in developing algorithms. However, careless use of recursion in programming has many negatives.

- It repeatedly invokes the mechanism---and consequently the overhead---of function calls.

- This can be expensive in both processor time and memory space.

Recursion vs. iteration (2)

- Each recursive call causes another copy of the function (actually only the function’s variables) to be created; this can consume considerable memory.

- Iteration normally occurs within a function, so the overhead of repeated function calls and extra memory assignment is omitted.

- So why choose recursion?
    - It makes the code easier to underhand and debug.

Example: Searching an Array

- Given an array of integers.
- Return the array index of the first element of the array which is equal to x

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

// Iterative implementation of search.
int search_iter(const int *ns, int len, int target)
{
    int i = 0;
    for(i = 0; i < len; i++)
    {
        if(ns[i] == target)
        {
            return i;
        }
    }
    return -1;
}
```

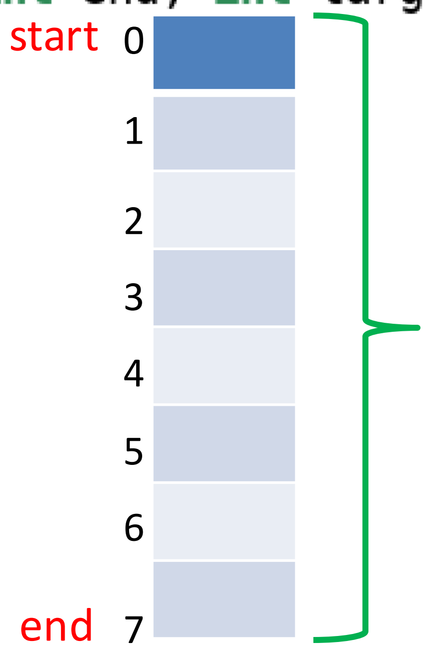
Iterative

Example: Search an Array (2)

- Start from array[0].
- Increase the index by 1 each time.

```
int search_rec_a(const int *ns, int start, int end, int target)
{
    if (start > end)
    {
        return -1;
    }
    if (ns[start] == target)
    {
        return start;
    }
    else
    {
        return search_rec_a(ns, start+1, end, target);
    }
}
```

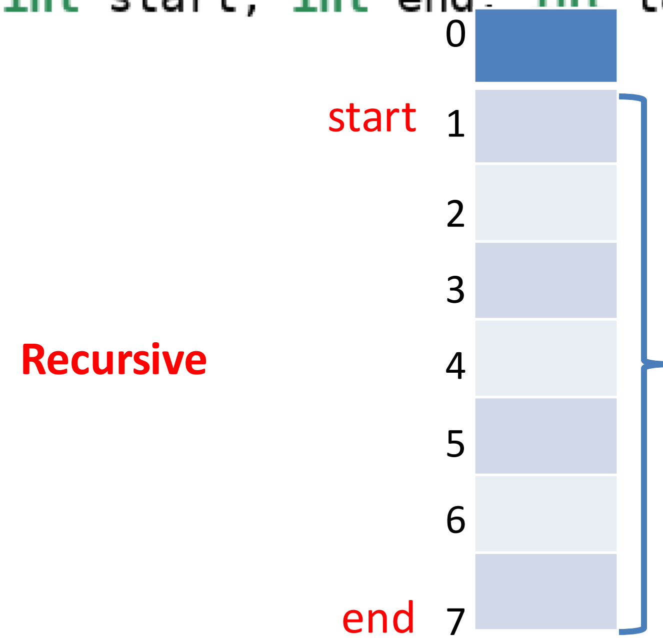
Recursive



Example: Search an Array (2)

- Start from array[0].
- Increase the index by 1 each time.

```c
int search_rec_a(const int *ns, int start, int end, int target)
{
    if (start > end)
    {
        return -1;
    }
    if (ns[start] == target)
    {
        return start;
    }
    else
    {
        return search_rec_a(ns, start+1, end, target);
    }
}
```

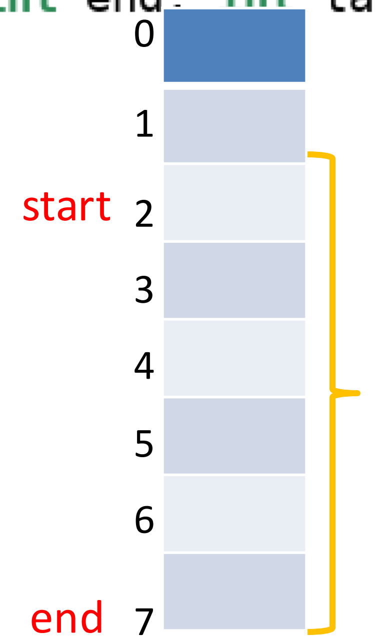


Example: Search an Array (2)

- Start from array[0].
- Increase the index by 1 each time.

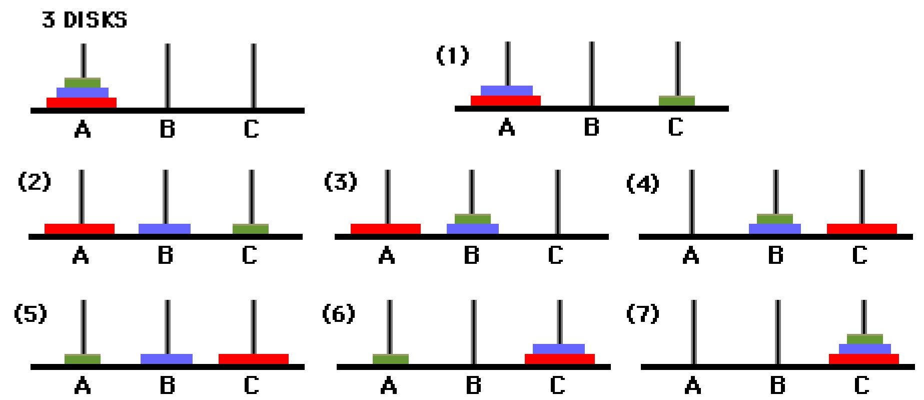
```
int search_rec_a(const int *ns, int start, int end, int target)
{
    if (start > end)
    {
        return -1;
    }
    if (ns[start] == target)
    {
        return start;
    }
    else
    {
        return search_rec_a(ns, start+1, end, target);
    }
}
```

Recursive

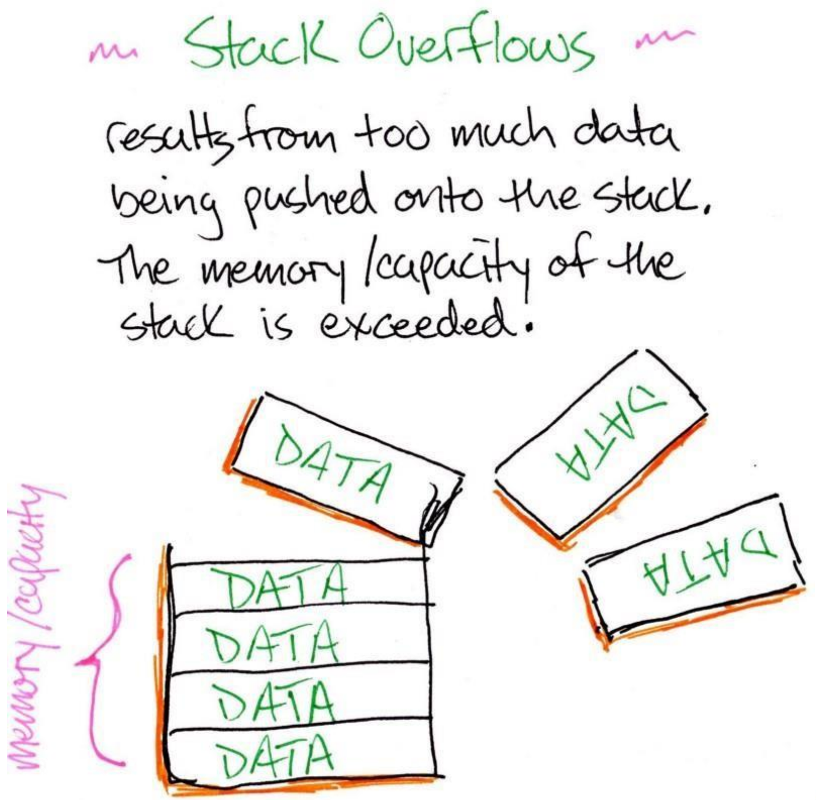


Be Careful … Stack Overflow

- Computers have limited memory.
- Each function call uses a part of your computer's memory.



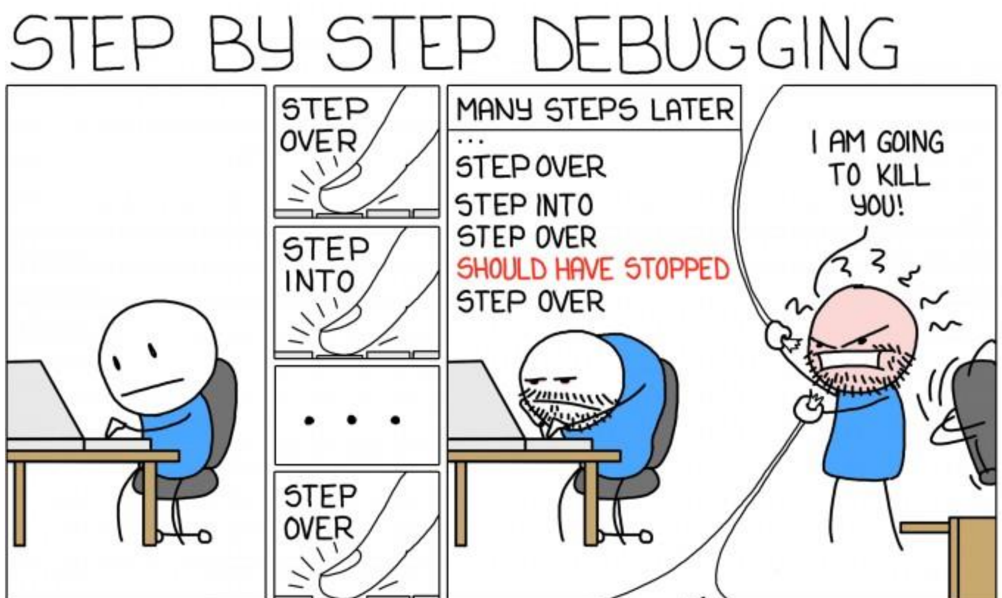
Source: https://www.codewithc.com/c-program-for-tower-of-hanoi-recursion/



Source: https://www.freecodecamp.org/news/recursion-is-not-hard-858a48830d83/

Learn to Debug!!

- Print something … anything to see when your program breaks!!
- Print the values of the parameters to see if your program does what it suppose to do!



Source: http://devhumor.com/media/monkeyuser-step-by-step-debugging

University of Nottingham
UK | CHINA | MALAYSIA

Overview

- Header file
- Recursive function
- **Scope of variables**

Local vs. Global Variables

- Variable's **scope** is the part of the program in which the variable can be accessed.

```
void test(int a, int b);

int global = 10;

int main(){
    int c;
    test(10, 20);
    return 0;
}

void test(int a, int b){
    int c;
    /* Function body. */
}
```

Local

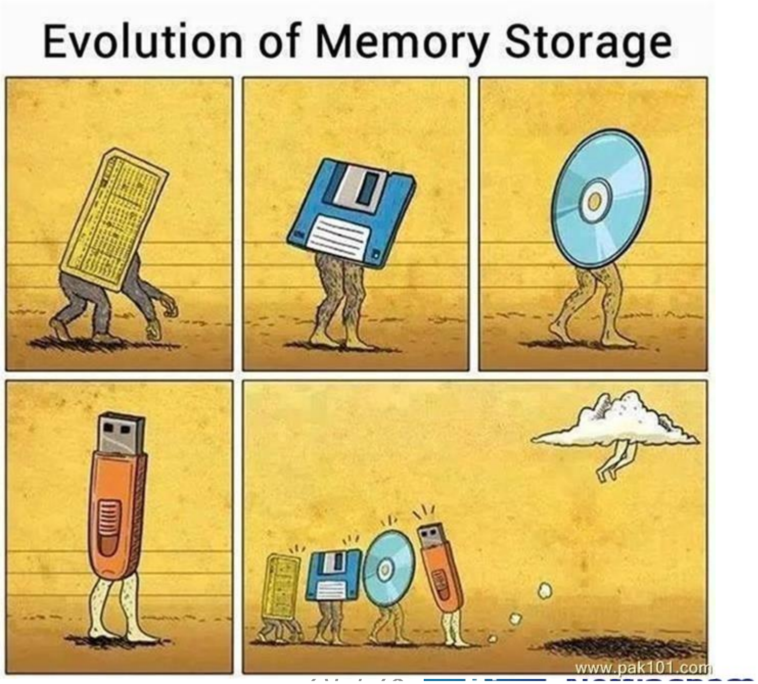
Global Variables

- SHOULD be avoided, unless application performance is critical.

- Global variables allow unintended side effects e.g., a function does not need to access a variable but access and modify it accidentally or maliciously.

Storage Classes in C

- Storage class specifiers in C are auto, register, extern and static.
- They specify the period during which the identifier exists in memory, e.g., briefly, repeatedly created and destroyed, or entire execution of the program.



Source: https://itzone.com.vn/en/article/funny-image-evolution-of-memory-storage-610/

Automatic Storage Duration

- Use keyword ***auto***.
- Local variables have automatic storage duration by default so the keyword itself is rarely used.

```c
int main(void)
{
    auto int myInt =  1;

    return 0;
}
```

- This conserves memory because automatic variables are created and destroyed when needed e.g., when a function is entered and exited.

static Variables

Global variable – global scope, whole program life time

Static variable – local scope, whole program life time

- The memory that is allocated to store the local variables is allocated and *initialised only once*, before the program begins execution.

- Static variables retain their values when the function is exited.

```
void test();
int main(){
    test();
    test();
    return 0;
}
void test(){
    static int i = 100;
    int j = 0;
    i++;
    j++;
    printf("%d %d\n", i, j);
}
```

displays
101 1
102 1

Scope of Arrays as Arguments

- Note that by default, arrays are **passed by reference**.
- e.g., void test (int arr[]);

  test(arr);

Use the name of the array as a pointer

To prevent a function from changing the values of the array elements, use the word **const** in its declaration.

void test(const int arr[]);

```
void test(int arr[]){
	printf("Size = %d bytes\n", sizeof(arr));
}
```

Return the size of the pointer, not arr

University of Nottingham

UK | CHINA | MALAYSIA

Scope of Arrays as Arguments (2)

```c
#include <stdio.h>

void testArray(int *a, int size);

int main(void)
{
    int array[3] = {1, 2, 3};

    printf("Output from main()\n");
    int i = 0;
    for(i = 0; i < 3; i++)
    {
        printf("%d %d\n", i, array[i]);
    }
    printf("\n\n");

    testArray(array, 3);

    printf("Output from main()\n");
    for(i = 0; i < 3; i++)
    {
        printf("%d %d\n", i, array[i]);
    }

    return 0;
}

void testArray(int *a, int size)
{
    printf("Output from testArray()\n");
    int i = 0;
    for(i = 0; i < size; i++)
    {
        printf("%d %d\n", i, a[i]);
    }
    printf("\n\n");

    a[0] = 333;

    printf("Output from testArray() after changing the value of a[0]\n");
    for(i = 0; i < size; i++)
    {
        printf("%d %d\n", i, a[i]);
    }
    printf("\n\n");
}
```

```
C:\Users\z2017233\Downloads>recursion
Output from main()
0 1
1 2
2 3


Output from testArray()
0 1
1 2
2 3


Output from testArray() after changing the value of a[0]
0 333
1 2
2 3


Output from main()
0 333
1 2
2 3

C:\Users\z2017233\Downloads>
```

Passing 2D array: dimensions are available globally

```
#include <stdio.h>
const int M = 3;
const int N = 3;

void print(int arr[M][N])
{
    int i, j;
    for (i = 0; i < M; i++)
      for (j = 0; j < N; j++)
        printf("%d ", arr[i][j]);
}

int main()
{
    int arr[][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    print(arr);
    return 0;
}
```

University of Nottingham
UK | CHINA | MALAYSIA

Passing 2D array: a dimension is available globally

```
#include <stdio.h>
const int N = 3;

void print(int arr[][N], int m)
{
   int i, j;
   for (i = 0; i < m; i++)
     for (j = 0; j < N; j++)
       printf("%d ", arr[i][j]);
}

int main()
{
   int arr[][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
   print(arr, 3);
   return 0;
}
```

Passing 2D array: a pointer

```
#include <stdio.h>
void print(int *arr, int m, int n)
{
   int i, j;
   for (i = 0; i < m; i++)
     for (j = 0; j < n; j++)
       printf("%d ", *((arr+i*n) + j));
}

int main()
{
   int arr[][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
   int m = 3, n = 3;

   print((int *)arr, m, n);
   return 0;
}
```

Summary

- Header file
- Recursive function
- Scope of variables