First Year First Semester Course

M.Tech. (CS) [Batch 2021-23]

Lecture #05

Introduction to Programming

User-defined Functions in C

Course Instructor:

Dr. Monidipa Das

DST-INSPIRE Faculty

Machine Intelligence Unit (MIU), Centre for Artificial Intelligence and Machine Learning (CAIML)

Indian Statistical Institute (ISI) Kolkata, India

Introduction

- **Function**
  - A self-contained program segment that carries out some specific, well-defined task.
  - Every C program consists of one or more functions.
  - A function will carry out its intended action whenever it is *called* or *invoked*.
  - In general, a function will process information that is passed to it from the calling portion of the program, and returns a single value.
  - Some function may not return anything.

- **Why functions?**
  - Modularize a program
  - Manageable program development
  - Software reusability
  - Avoids code repetition

Function Example


```c
#include <stdio.h>
int factorial (int);
int main()
{
    int n,fact;
    for (n=1; n<=10; n++) {
          fact=factorial (n);
          printf ("%d! = %d \n",n,fact);
    }
    return 0;
}
int factorial (int m)
{
    int i, temp=1;
    for (i=1; i<=m; i++)
          temp = temp * i;
    return (temp);
}
```

Function declaration

```c
#include <stdio.h>
int factorial (int m)
{
    int i, temp=1;
    for (i=1; i<=m; i++)
          temp = temp * i;
    return (temp);
}
int main()
{
    int n;
    for (n=1; n<=10; n++)
          printf ("%d! = %d \n",n,factorial(n));
    return 0;
}
```

Function definition

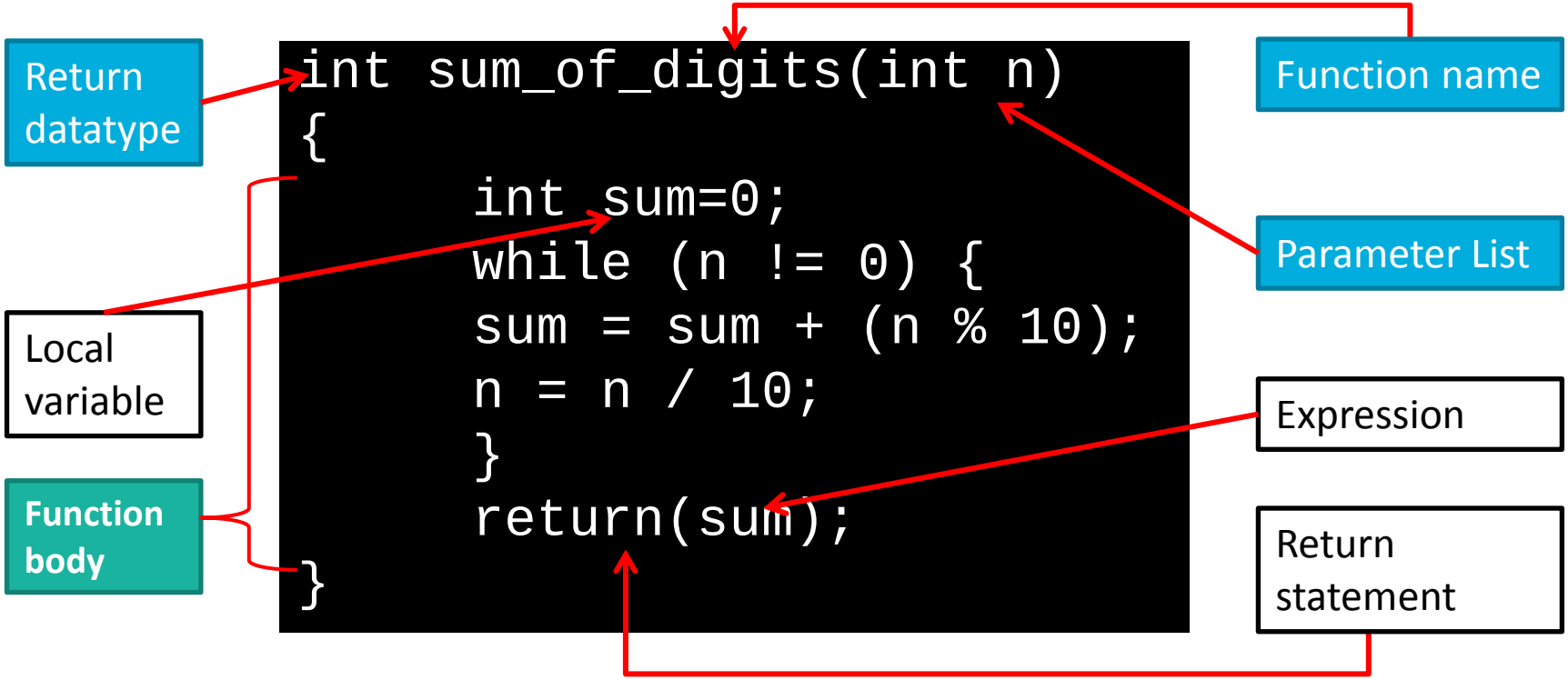
Function call

Defining a Function

- A function definition has ***two*** parts:
  - The first line (function header)
  - The body of the function.

```
<return-value-type> <function-name> ( <parameter-list> )
{
        declarations and statements
}
```

Example: Components of a function definition



Function Declaration (or Prototype)

- Function prototypes are usually written at the beginning of a program, ahead of any functions (including main()).

- Examples:

```
int  gcd (int A, int B);
int  gcd (int A, int B);
int gcd (int, int);  /* allowed in function prototype */
void div7 (int); /* allowed in function prototype */
```

- Note the semicolon at the end of the line.
- The argument names can be different (optional too); but it is a good practice to use the same names as in the function definition.

Function Prototype: Examples

Function prototype/declaration

```
#include <stdio.h>
int ncr (int n, int r);
int fact (int n);
int main()
{
        int i, m, n, sum=0;
        printf("Input m and n \n");
        scanf ("%d %d", &m, &n);

        for (i=1; i<=m; i+=2)
                sum = sum + ncr (n, i);

        printf ("Result: %d \n", sum);
        return 0;
}
```

Function prototype is optional if it is defined before call.

```
int ncr (int n, int r)
{
   return (fact(n) / fact(r) / fact(n-r));
}
int fact (int n)
{
        int i, temp=1;
        for (i=1; i<=n; i++)
                temp *= i;
        return (temp);
}
```

Function definition

Function: Call

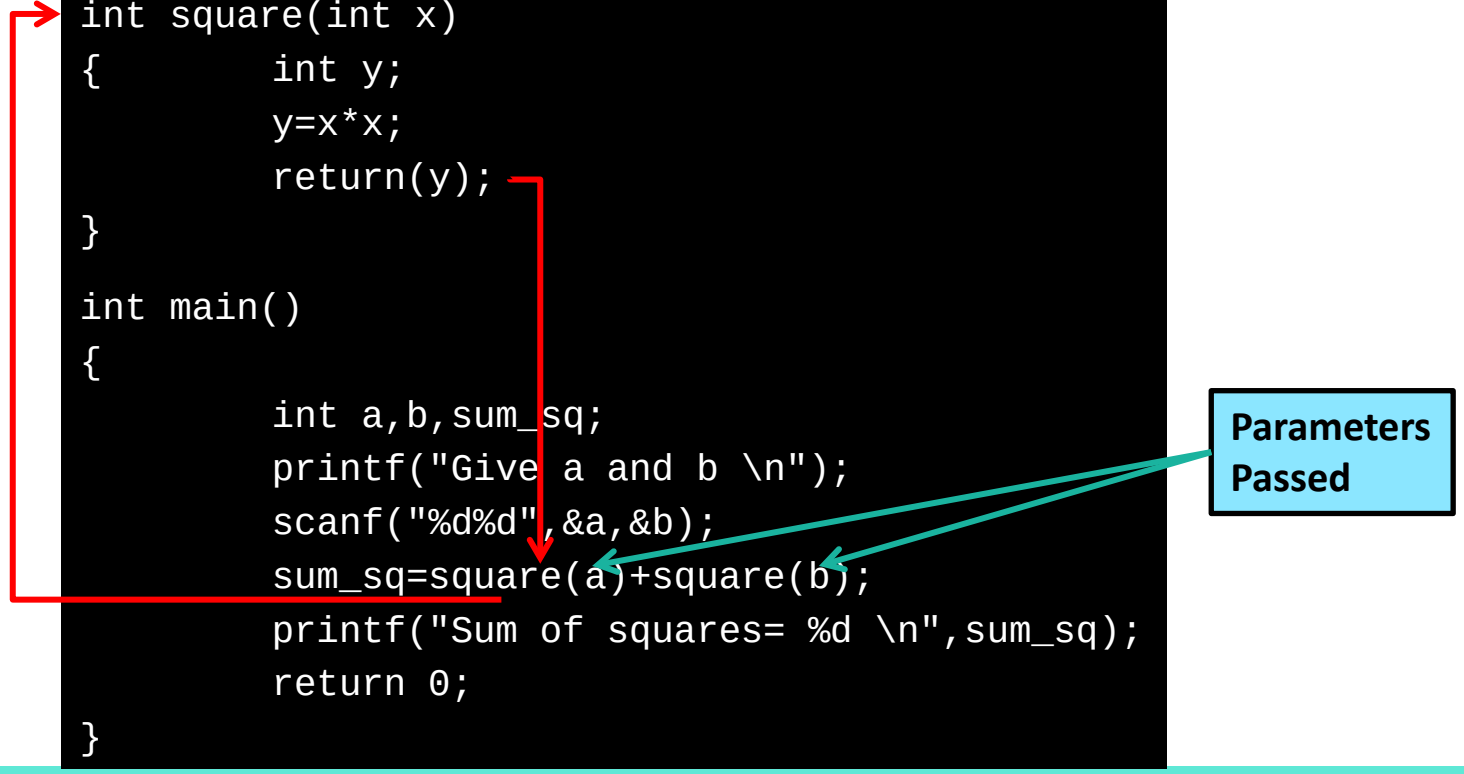
- When a function is called from some other function, the corresponding arguments in the function call are called ***actual arguments*** or ***actual parameters***.
  - The ***formal*** and ***actual*** arguments must match in their data types.

- **Note:**
  - The identifiers used as formal arguments are “local”.
    - Not recognized outside the function.
    - Names of formal and actual arguments may differ.

Function Call: Example


```
#include <stdio.h>
int square(int x)
{       int y;
        y=x*x;
        return(y);
}

int main()
{
        int a,b,sum_sq;
        printf("Give a and b \n");
        scanf("%d%d",&a,&b);
        sum_sq=square(a)+square(b);
        printf("Sum of squares= %d \n",sum_sq);
        return 0;
}
```



Function call : Example [contd.]


```
#include <stdio.h>
int square(int x)
{
        int y;
        y=x*x;
        return(y);
}
int main()
{
        int a,b,sum_sq;
        printf("Give a and b \n");
        scanf("%d%d",&a,&b);
        sum_sq=square(a)+square(b);
        printf("Sum of squares= %d \n",sum_sq);
        return 0;
}
```

Functions: Some Facts

- A function cannot be defined within another function.
  - All function definitions must be disjoint.
- Nested function calls are allowed.
  - A calls B, B calls C, C calls D, etc.
  - The function called last will be the first to return.
- A function can also call itself, either directly or in a cycle.
  - A calls B, B calls C, C calls back A.
  - Called ***recursive call*** or ***recursion***.

Functions: Some Facts

- A function can be declared within a function.

- The function declaration, call, and definition must match with each other (in terms of parameter type, parameter count, and parameter order).

```
int gcd(int a, int b); // function declaration
gcd(a,b); //function call, a and b is int
int gcd(int a, int b) // function definition
{
 …..
 }
```

- The parameter names do not need to be the same in the function definition, function call, and prototype declaration

Categories of Functions

- **Category 1:** Functions ***with no argument*** and ***no return value***
- **Category 2:** Functions ***with argument(s)*** and ***no return value***
- **Category 3:** Functions ***with argument(s)*** and ***one return value***
- **Category 4:** Functions ***with no argument*** but ***a return value***
- **Category 5:** Functions that ***returns multiple values***

Function with **no argument** and **no return value**


```
#include<stdio.h>
void printline (void);

int main()
{
        printline();
        printf("\nExample of function with no argument and no return value.\n");
        printline();
        return 0;
}

void printline(void)
{
        int i;
        for(i=1;i<=60;i++)
                printf("_");
        return;
}
```

Functions **with argument(s)** and **no return value**


```
#include<stdio.h>
void checkprime (int);
int main()
{
        int n;
        printf("Enter a natural number: ");
        scanf("%d",&n);
        checkprime(n);
        return 0;
}
```

```
void checkprime(int x){
    int i;
    for(i=2;i<x;i++){
        if(x%i==0){
            printf("\n%d is NOT PRIME!",x);
            break;
        }
    }
    if(i==x)
        printf("\n%d is PRIME!",x);
    return;
}
```

Functions **with argument(s)** and **one return value**


```
#include <stdio.h>
int factorial (int);
int main()
{
    int n,fact;
    for (n=1; n<=10; n++) {
           fact=factorial (n);
           printf ("%d! = %d \n",n,fact);
    }
    return 0;
}
int factorial (int m)
{
    int i, temp=1;
    for (i=1; i<=m; i++)
           temp = temp * i;
    return (temp);
}
```

Functions **with no argument** but **a return value**


```
#include<stdio.h>
int readnumber (void);
int main()
{
        int n= readnumber();
        printf("\nThe number is: %d",n);
        return 0;
}
int readnumber(void)
{
        int num;
        scanf("%d",&num);
        return(num);
}
```

Functions that **returns multiple values**


```
#include<stdio.h>
void mathoperation(int, int, int *s, int *d);
int main()
{
        int x=30, y=12, sum, diff;
        mathoperation(x,y,&sum,&diff);
        printf("\nsum=%d, diff=%d",sum,diff);
        return 0;
}
void mathoperation(int a, int b, int* s, int* d)
{
        *s=a+b;
        *d=a-b;
}
```

Pass by pointers

Return type of any function may be void…. Still it can return value(s) when called by pointers.

Parameter Passing mechanism

- Passing parameters in this way is called
  - Pass-by-pointers or Call-by-pointers.
- Normally parameters are passed in C using
  - Pass-by-value or Call-by-value.
- What does it mean?
  - If a function changes the values with an access to the memory address of a variable, then these changes will be made to the original array that is passed to the function.
  - This does not apply when an individual element is passed on as argument.

Pass by Value and Pass by Pointers

- **Pass by value**
  - Copy of argument passed to function
  - Changes in function do not effect original
  - Use when function does not need to modify argument
  - Avoids accidental changes

- **Pass by pointers**
  - Passes original argument
  - Changes in function effect original
  - Only used with trusted functions

Recursion

- A special case where a function calls itself repeatedly.
  - Either directly
    - X calls X.
  - Or cyclically in a chain.
    - X calls Y, and Y calls X.
- Used for repetitive computations in which each action is stated in terms of a previous result.
  - fact(n) = n * fact (n-1)

Recursion

- For a problem to be written in recursive form, two conditions are to be satisfied:
  - It should be possible to express the problem in recursive form.
  - The problem statement must include a stopping condition

$$fact(n) = 1, \quad \text{if } n = 0$$
$$= n * fact(n-1), \quad \text{if } n > 0$$

Recursion

- Examples:
  - **Factorial:**

    fact(0) = 1

    fact(n) = n * fact(n-1), if n > 0

  - **GCD:**

    gcd (m, m) = m

    gcd (m, n) = gcd (m-n, n), if m > n

    gcd (m, n) = gcd (n, n-m), if m < n

  - **Fibonacci series (1,1,2,3,5,8,13,21,….)**

    fib (0) = 0

    fib (1) = 1

    fib (n) = fib (n-1) + fib (n-2), if n > 1

Example-1: Recursively Computing Factorial


```
#include <stdio.h>

int fact(int n)
{
        if (n == 0)
                return 1;
        else
                return (n * fact(n-1));
}

int main()
{
        int i=6;
        printf (“Factorial of 6 is: %d \n”, fact(i));
        return 0;
}
```

Example 2: Fibonacci series


```
#include <stdio.h>
int fib(int n)
{
        if (n < 2)
                return n;
        else
                return (fib(n-1) + fib(n-2));
}
int main()
{
        int i=4;
        printf ("%d \n", fib(i));
        return 0;
}
```

Execution of Fibonacci number

- Fibonacci number fib(n) can be defined as:

  fib(0) = 0

  fib(1) = 1

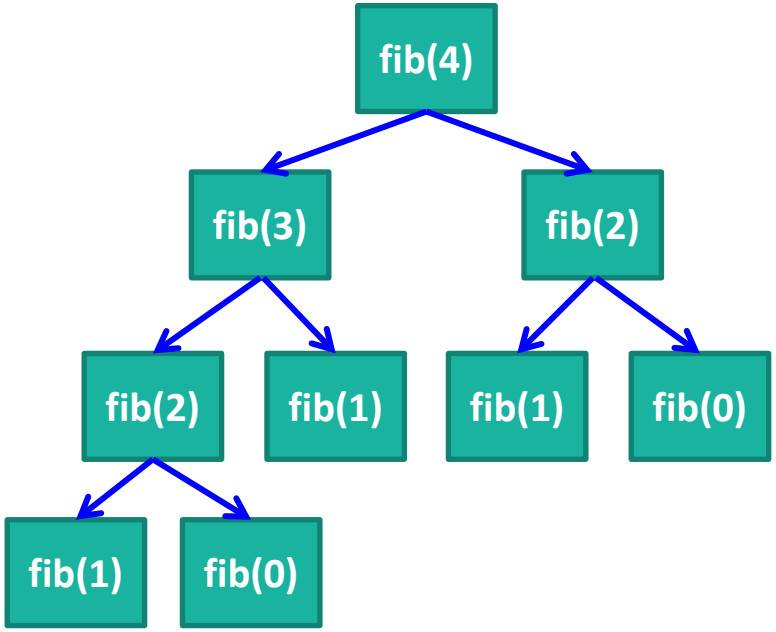
  fib(n) = fib(n-1) + fib(n-2), if n > 1

  – The successive Fibonacci numbers are:

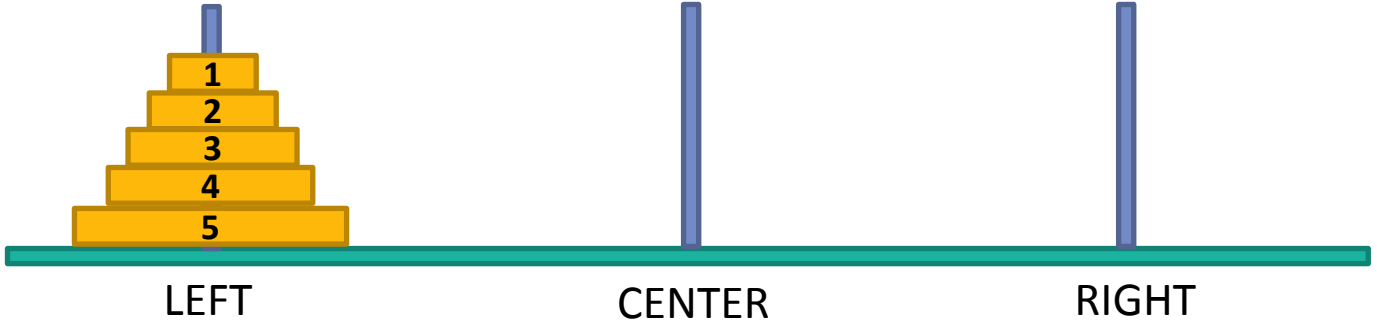
    0, 1, 1, 2, 3, 5, 8, 13, 21, …..

```
int fib(int n)
{
        if (n < 2)
                return (n);
        else
        return (fib(n-1) + fib(n-2));
}
```

Inefficient! The same thing is computed several times.



Example 3: Towers of Hanoi Problem



- **The problem statement:**
  - Initially all the disks are stacked on the LEFT pole.
  - Required to transfer all the disks to the RIGHT pole.
    - **Only one disk can be moved at a time.**
    - **A larger disk cannot be placed on a smaller disk.**

Recursion is implicit

- General problem of n disks.
  - **Step 1:**
    - Move the top (n-1) disks from LEFT to CENTER.
  - **Step 2:**
    - Move the largest (n-th) disk from LEFT to RIGHT.
  - **Step 3:**
    - Move the (n-1) disks from CENTER to RIGHT.

Recursive C code: Towers of Hanoi


```
#include <stdio.h>
void transfer (int n, char from, char to, char temp);
int main()
{
        int n; /* Number of disks */
        scanf ("%d", &n);
        transfer (n, 'L', 'R', 'C');
        return 0;
}
void transfer (int n, char from, char to, char temp)
{
        if (n > 0) {
        transfer (n-1, from, temp,to);
        printf (" Move disk %d from %c to %c \n", n, from, to);
        transfer (n-1, temp, to, from);
        }
        return;
}
```

Towers of Hanoi: Example Output


```
3
Move disk 1 from L to R
Move disk 2 from L to C
Move disk 1 from R to C
Move disk 3 from L to R
Move disk 1 from C to L
Move disk 2 from C to R
Move disk 1 from L to R
```

```
4
Move disk 1 from L to C
Move disk 2 from L to R
Move disk 1 from C to R
Move disk 3 from L to C
Move disk 1 from R to L
Move disk 2 from R to C
Move disk 1 from L to C
Move disk 4 from L to R
Move disk 1 from C to R
Move disk 2 from C to L
Move disk 1 from R to L
Move disk 3 from C to R
Move disk 1 from L to C
Move disk 2 from L to R
Move disk 1 from C to R
```

```
5
Move disk 1 from L to R
Move disk 2 from L to C
Move disk 1 from R to C
Move disk 3 from L to R
Move disk 1 from C to L
Move disk 2 from C to R
Move disk 1 from L to R
Move disk 4 from L to C
Move disk 1 from R to C
Move disk 2 from R to L
Move disk 1 from C to L
Move disk 3 from R to C
Move disk 1 from L to R
Move disk 2 from L to C
Move disk 1 from R to C
Move disk 5 from L to R
Move disk 1 from C to L
Move disk 2 from C to R
Move disk 1 from L to R
Move disk 3 from C to L
Move disk 1 from R to C
Move disk 2 from R to L
Move disk 1 from C to L
Move disk 4 from C to R
Move disk 1 from L to R
Move disk 2 from L to C
Move disk 1 from R to C
Move disk 3 from L to R
Move disk 1 from C to L
Move disk 2 from C to R
Move disk 1 from L to R
```

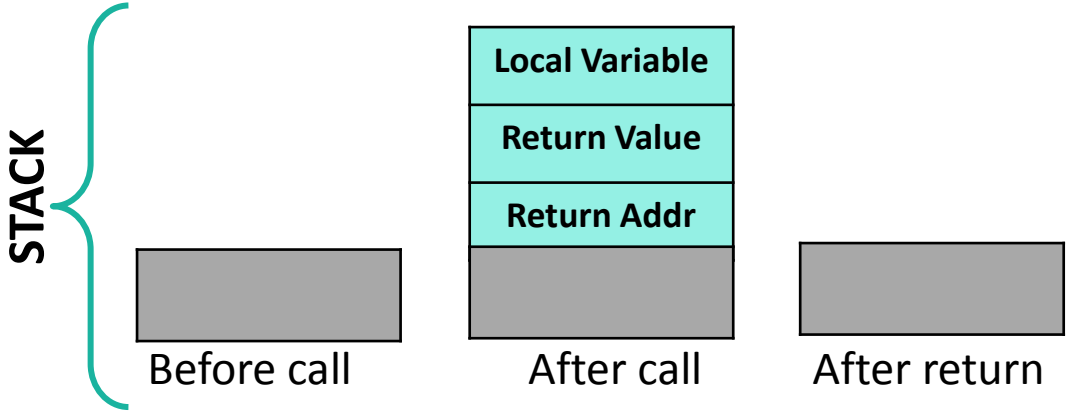
How are function calls implemented?

- In general, during program execution
  - The system maintains a *stack* in memory.
    - *Stack* is a *last-in first-out* structure.
    - Two operations on stack, *push* and *pop*.
  - Whenever there is a function call, the *activation record* gets *pushed* into the stack.
    - Activation record consists of the *return address* in the calling program, the *return value* from the function, and the *local variables* inside the function.
    - At the end of function call, the corresponding *activation record* gets *popped* out of the stack.

At the system


```
main()
{
        ……..
        x = gcd (a, b);
        ……..
}
```

```
int gcd (int x, int y)
{
        ……..
        ……..
        return (result);
}
```

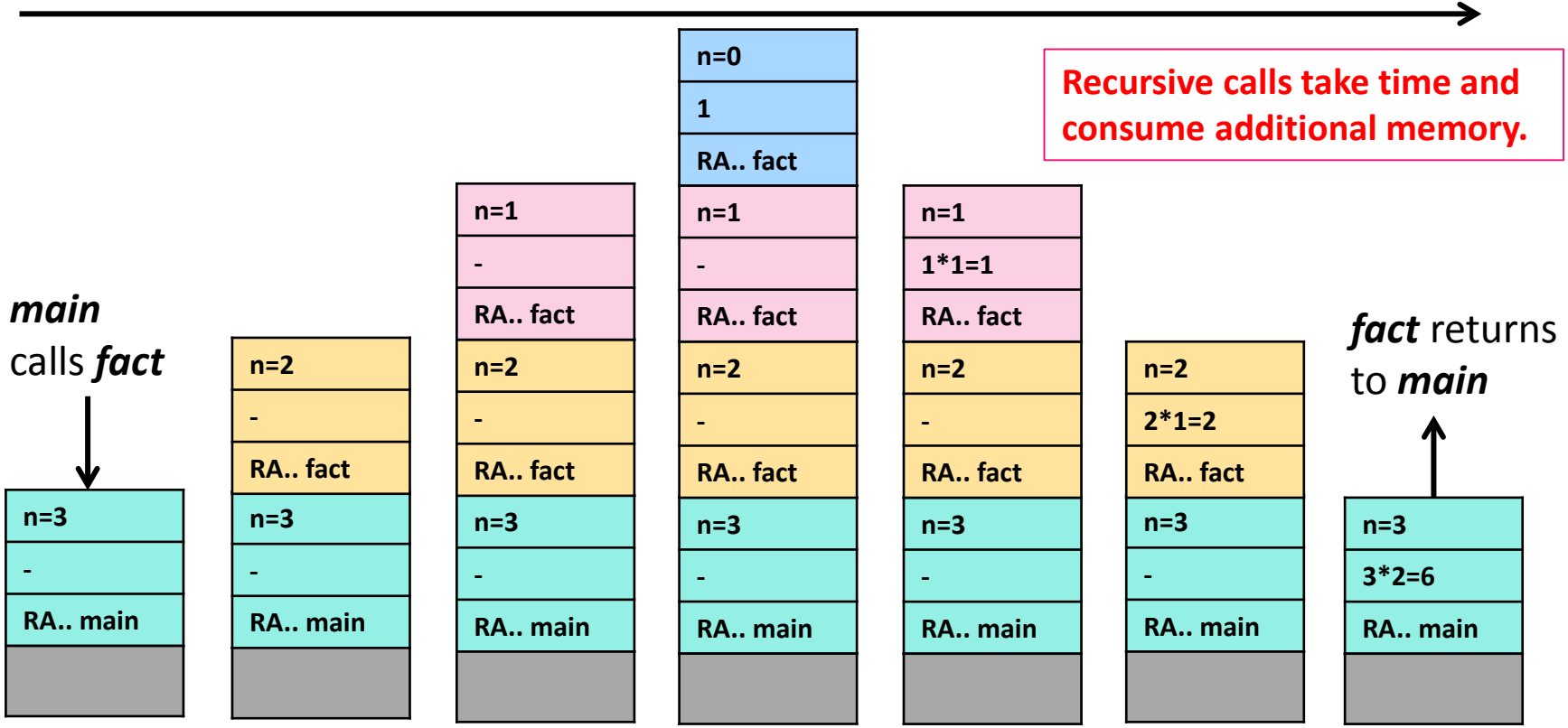


Example: main() calls fact(3)


```
void main()
{
        int n;
        n = 4;
        printf (“%d \n”, fact(n) );
}
```

```
int fact (int n)
{
        if (n = = 0)
                return (1);
        else
                return (n * fact(n-1));
}
```

Trace of the Stack During Execution



Recursive calls take time and consume additional memory.

Questions?