CS - 114 : Computer Workshop

Prof. Chamakuri Nagaiah
Mahindra-École Centrale, Hyderabad
nagaiah.chamakuri@mechyd.ac.in

Introduction : Functions

- Function : A function is a module or block of program code which deals with a specific, well-defined task.
- Some properties : Every C program consists of one or more functions.
  - One of these functions must be called “main”.
  - Execution of the program always begins by carrying out the instructions in “main“.
- 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.
  - Information is passed to the function via special identifiers called arguments or parameters
  - The value is returned by the ”return“ statement.
- Some functions may not return anything : Return data type specified as ”void“.

Example

```
#include  <stdio.h>

int  factorial (int m)
{
    int i, temp=1;
    for (i=1; i<=m; i++)
    temp = temp * i;
    return (temp);
}
```

```
main()
{
    int  n;
    for  (n=1; n<=10; n++)
          printf ("%d! = %d \n",
               n, factorial (n) );
}
```

Output:

1! = 1

2! = 2

3! = 6 …….. upto 10!

Why Functions? : Functions

- Allows one to develop a program in a modular fashion.
  - Divide-and-conquer approach.
- All variables declared inside functions are local variables.
  - Known only in function defined.
  - There are exceptions (will be discussed later).
- Parameters
  - Communicate information between functions.
  - They also become local variables.

Why Functions? : Benefits

- Divide and conquer.
  - Manageable program development.
  - Construct a program from small pieces or components.
- Software reusability.
  - Use existing functions as building blocks for new programs.
  - Abstraction: hide internal details (library functions).

Defining a Function

- A function definition has two parts:
  - The first line.
  - The body of the function.

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

Defining a Function

- The first line contains the return-value-type, the function name, and optionally a set of comma-separated arguments enclosed in parentheses.
  - Each argument has an associated type declaration.
  - The arguments are called formal arguments or formal parameters.
- Example:

  ```
  int gcd (int A, int B)
  ```

- The argument data types can also be declared on the next line:

  ```
  int gcd (A, B)
   int A, B;
  ```

Defining a Function

- The body of the function is actually a compound statement that defines the action to be taken by the function .

```
int gcd (int A, int B)
   {
     int temp;
     while ((B % A) != 0) {
          temp = B % A;
          B = A;
          A = temp;
     }
     return (A);
   }
```

Defining a Function

- 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.
  - The notion of positional parameters is important
- Point to note : The identifiers used as formal arguments are ”local“.
  - Not recognized outside the function.
  - Names of formal and actual arguments may differ.

Function Not Returning Any Value

- Example: A function which prints if a number is divisible by 7 or not.

```
void div7 (int n)
 {
  if ((n % 7) == 0)
     printf ("%d is divisible by 7", n);
  else
     printf ("%d is not divisible by 7", n);
  return;     Optional
 }
```

Returning control

- If nothing returned
  - return;
  - or, until reaches right brace
- If something returned
  - return expression ;

Some Key Points

- 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.

Example:: main calls ncr, ncr calls fact

```
#include  <stdio.h>

int ncr (int n, int r);
int fact (int n);

main()
{
    int i, m, n, sum=0;
    scanf ("%d %d", &m, &n);

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

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

```
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);
}
```

Variable Scope

```
#include  <stdio.h>
int A;
void main()
{  A = 1;
   myProc();
   printf ( "A = %d\n", A);
}

void myProc()
{    int A = 2;
     while( A==2 )
   {
      int A = 3;
      printf ( "A = %d\n", A);
      break;
   }
   printf ( "A = %d\n", A);
}
```

Output:

A = 3

A = 2

A = 1

Math Library Functions

- Math library functions
  - perform common mathematical calculations

    #include <math.h>
- Format for calling functions

  FunctionName (argument);
  - If multiple arguments, use comma-separated list

    printf ("%f", sqrt(900.0));
  - Calls function sqrt, which returns the square root of its argument.
  - All math functions return data type double.

  – Arguments may be constants, variables, or expressions .

Math Library Functions

```
double acos(double x) - Compute arc cosine of x.
double asin(double x) - Compute arc sine of x.
double atan(double x) - Compute arc tangent of x.
double atan2(double y, double x)
                - Compute arc tangent of y/x.
double cos(double x)
          - Compute cosine of angle in radians.
double cosh(double x) - Compute the hyperbolic cosine of x.
double sin(double x)  - Compute sine of angle in radians.
double sinh(double x) - Compute the hyperbolic sine of x.
double tan(double x)
          - Compute tangent of angle in radians.
double tanh(double x)
          - Compute the hyperbolic tangent of x.
```

Math Library Functions

```
double ceil(double x)  -
            Get smallest integral value that exceeds x.
double floor(double x) -
            Get largest integral value less than x.
double exp(double x)   - Compute exponential of x.
double fabs (double x )- Compute absolute value of x.
double log(double x)   - Compute log to the base e of x.
double log10 (double x )- Compute log to the base 10 of x.
double pow (double x, double y) -
             Compute x raised to the power y.
double sqrt(double x) - Compute the square root of x.
```

Function Prototypes

- Usually, a function is defined before it is called.
  - main() is the last function in the program.
  - Easy for the compiler to identify function definitions in a single scan through the file.
- However, many programmers prefer a top-down approach, where the functions follow main().
  - Must be some way to tell the compiler.
  - Function prototypes are used for this purpose.
    - Only needed if function definition comes after use.

Function Prototypes

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

  ```
  int gcd (int A, int B);
  void div7 (int number);
  ```

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

Header Files

- Header files
  - Contain function prototypes for library functions.
  - <stdlib.h> , <math.h> , etc
  - Load with: #include <filename>
  - Example:

    #include <math.h>
- Custom header files
  - Create file(s) with function definitions.
  - Save as filename.h (say).
  - Load in other files with #include "filename.h"
  - Reuse functions.

Parameter passing: by Value and by Reference

- Used when invoking functions.
- Call by value
  - Passes the value of the argument to the function.
  - Execution of the function does not affect the original.
  - Used when function does not need to modify argument.
    - Avoids accidental changes.
- Call by reference
  - Passes the reference to the original argument.
  - Execution of the function may affect the original.
  - Not directly supported in C - can be effected by using pointers

Example: Random Number Generation

- rand function
  - Prototype defined in <stdlib.h>
  - Returns ”random“ number between 0 and RAND_MAX

    i = rand();
  - Pseudorandom
  - Preset sequence of "random" numbers

    Same sequence for every function call
- Scaling
  - To get a random number between 1 and n

    1 + (rand() % n)
  - To simulate the roll of a dice:

    1 + (rand() % 6)
- srand function (Mersenne Twister)
  - Prototype defined in <stdlib.h>.
  - Takes an integer seed, and randomizes the random number generator.

    srand (seed);

Example

```
1  /* A programming example
2    Randomizing die-rolling program */
3  #include <stdlib.h>
4  #include <stdio.h>
5
6  int main()
7  {
8     int i;
9     unsigned seed;
10
11    printf( "Enter seed: " );
12    scanf( "%u", &seed );
13    srand( seed );
14
15    for ( i = 1; i <= 10; i++ ) {
16       printf( "%10d ", 1 + ( rand() % 6 ) );
17
18       if ( i % 5 == 0 )
19          printf( "\n" );
20    }
21
22    return 0;
23 }
```

Output

```
Enter seed: 67
          6          1          4          6          2
          1          6          1          6          4
```

```
Enter seed: 867
          2          4          6          1          6
          1          1          3          6          2
```

```
Enter seed: 67
          6          1          4          6          2
          1          6          1          6          4
```

#define: Macro definition

- Preprocessor directive in the following form :
  #define string1 string2
  - Replaces ”string1“ by ”string2“ wherever it occurs before compilation. For example,
    #define PI 3.1415926

```
#include <stdio.h>
#define PI 3.1415926
main()
{
  float r=4.0,area;
  area=PI*r*r;
}
```

```
#include <stdio.h>

main()
{
  float r=4.0,area;
  area=3.1415926*r*r;
}
```

#define with arguments

#define statement may be used with arguments.

- **Example: #define sqr(x) x*x**
- **How will macro substitution be carried out?**

  r = sqr(a) + sqr(30); ➔ r = a*a + 30*30;

  r = sqr(a+b); ➔ r = a+b*a+b;

WRONG?

- **The macro definition should have been written as:**

  #define sqr(x) (x)*(x)

  r = (a+b)*(a+b);

C inline functions

- C allows you to define special functions called inline functions.
- An inline function is a relatively small function that the compiler will optimize it to ensure that the inline function will execute as fast as possible.
- The compiler will copy the code of an inline function to the calling function when it reaches an inline function.
- To define an inline function, you use the inline keyword as follows:

```
inline int max(int a,int b)
 {
   return a > b ? a : b;
 }
```

Recursion

- A process by which 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.

  $fact(n) = n * fact(n-1)$
- 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

    $fact(n) = 1$, if $n = 0$

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

Recursion : Syntax

```
 void recursion() {
  StoppingCondition;
   recursion(); /* function calls itself */
}

int main() {
   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) = 1

  fib (1) = 1

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

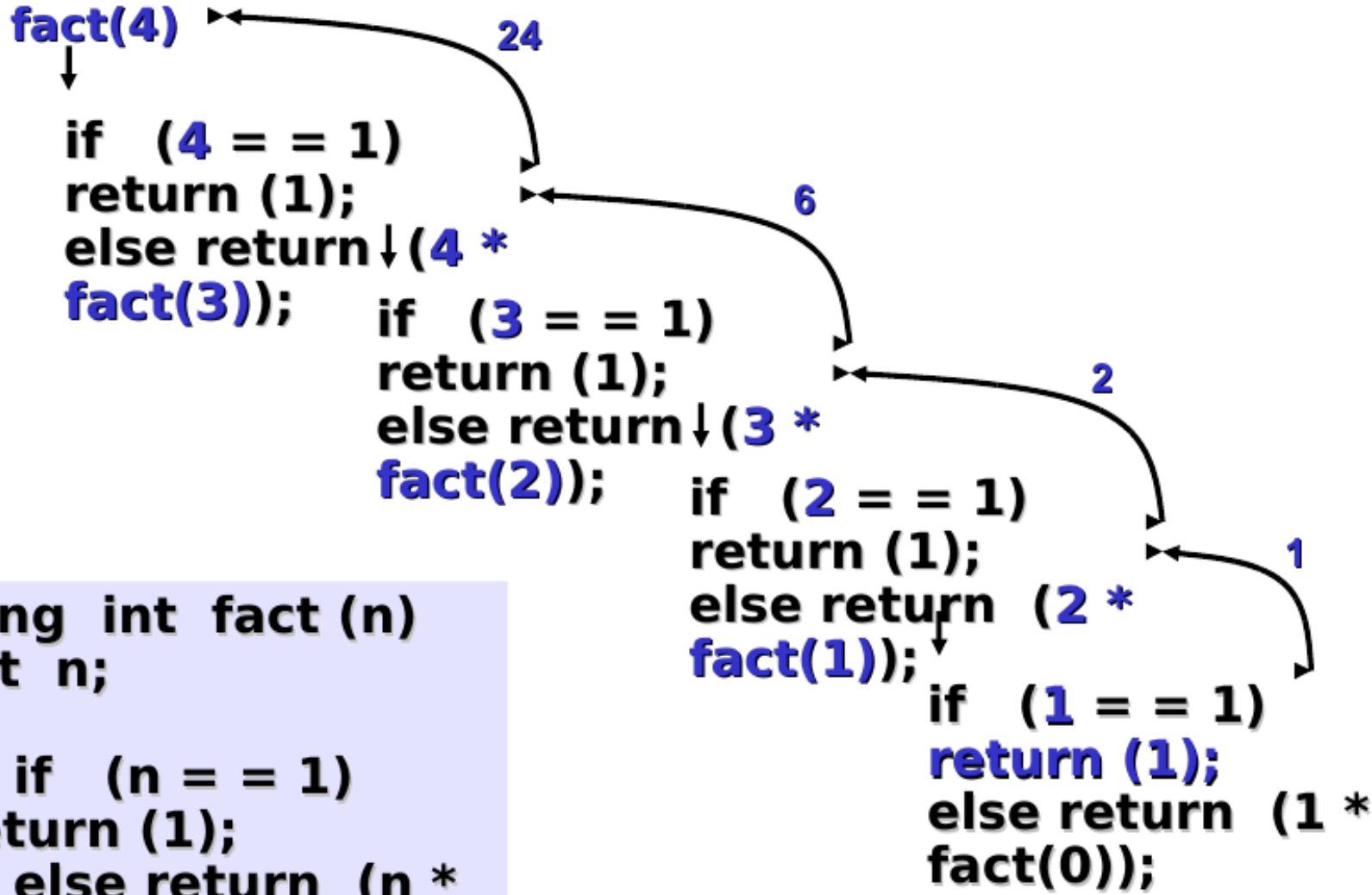
Example 1 :: Factorial

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

int  main() {
  int i = 15;
  printf("Factorial of %d is %d\n", i, fact(i));
  return 0;
}
```

- Factorial of 15 is 2004310016.

Example 1 ::Factorial Execution




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

Example 2 :: Fibonacci number

- **Fibonacci number f(n) can be defined as:**

  f(0) = 0

  f(1) = 1

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

  - **The successive Fibonacci numbers are:**

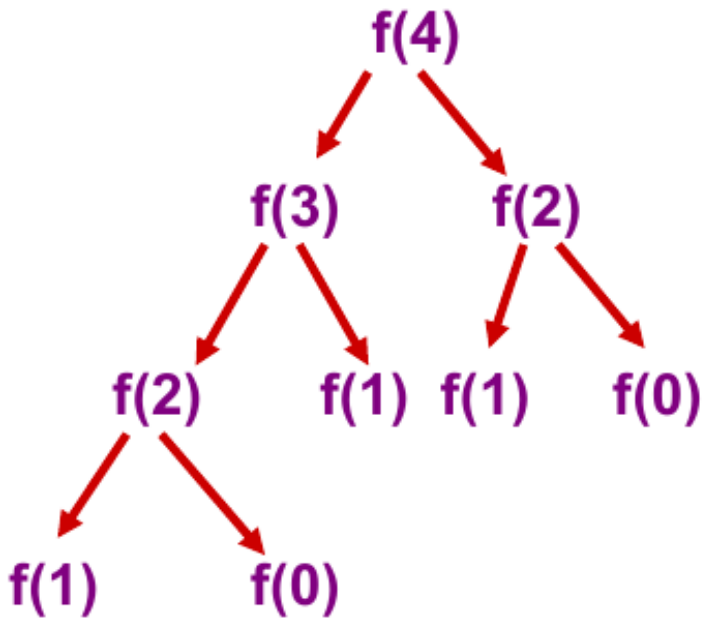
    0, 1, 1, 2, 3, 5, 8, 13, 21, ...

- **Function definition:**

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

Example 2 :: Tracing Execution

- **How many times is the function called when evaluating f(4) ?**



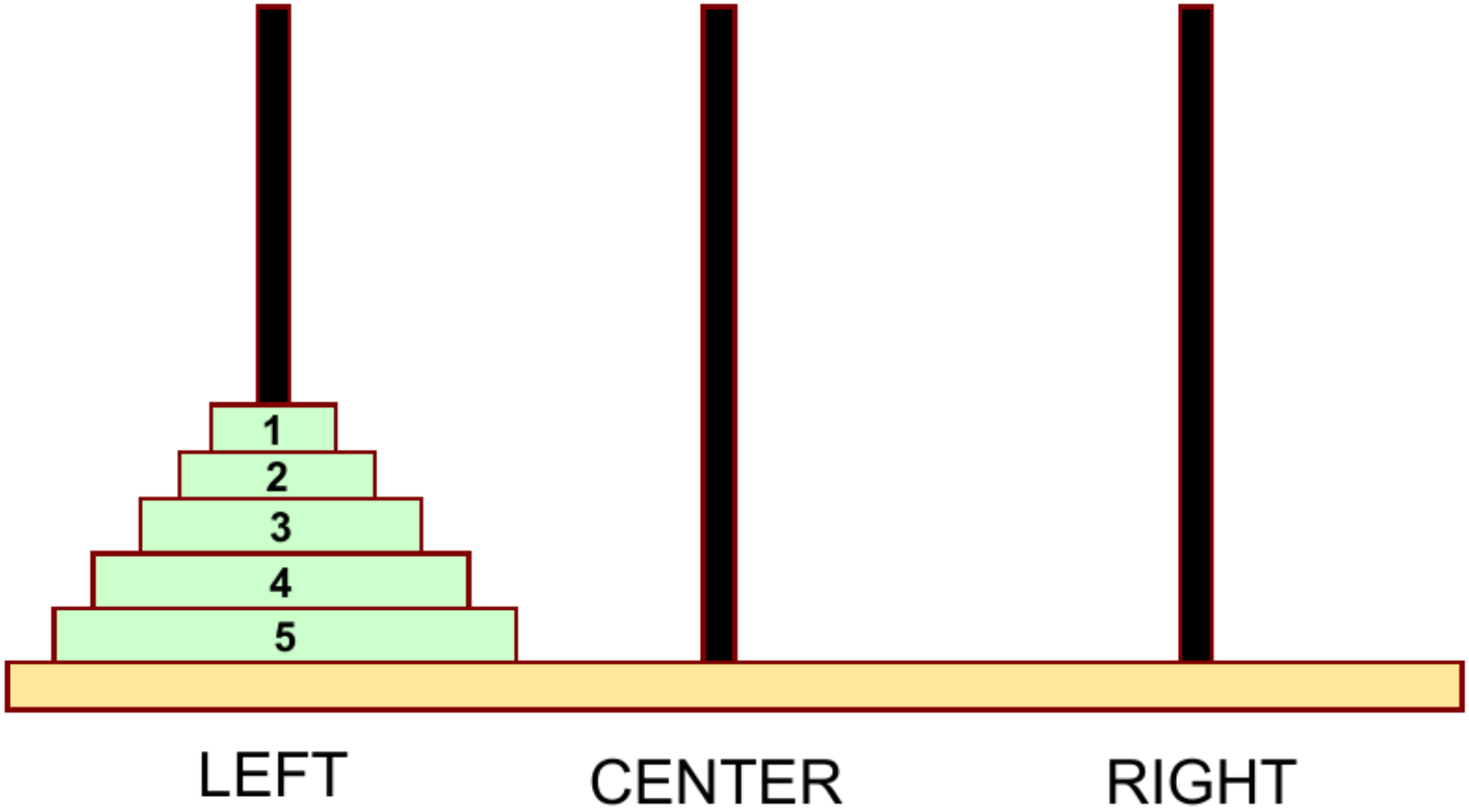
- **Inefficiency:**
  - **Same thing is computed several times.**

called 9 times

Recursive functions : Notable Points

- Every recursive program can also be written without recursion
- Recursion makes program elegant and cleaner. All algorithms can be defined recursively which makes it easier to visualize and prove.
- Sometimes, if the function being computed has a nice recurrence form, then a recursive code may be more readable
- If the speed of the program is vital then, you should avoid using recursion. Recursions use more memory and are generally slow. Instead, you can use loop.

Example 3 :: Towers of Hanoi Problem



Example 3 :: 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.
- CENTER pole is used for temporary storage of disks.
- Recursive statement of the general problem of n disks.
  - Step 1:
    - Move the top (n-1) disks from LEFT to CENTER.
  - Step 2:
    - Move the largest disk from LEFT to RIGHT.
  - Step 3:
    - Move the (n-1) disks from CENTER to RIGHT.

Recursion vs. Iteration

- Repetition
  - Iteration: explicit loop
  - Recursion: repeated function calls
- Termination
  - Iteration: loop condition fails
  - Recursion: base case recognized
- Both can have infinite loops
- Balance
  - Choice between performance (iteration) and good software engineering (recursion).

How are function calls implemented?