

# Functions

Indranil Sen Gupta

Dept. of Computer Science & Engg.  
Indian Institute of Technology  
Kharagpur

## Introduction

- **Function**
  - A self-contained program segment that carries out some 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 function may not return anything.
  - Return data type specified as “*void*”.

```
#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 (to be discussed later).**
- **Parameters**
  - **Communicate information between functions.**
  - **They also become local variables.**

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

- 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:

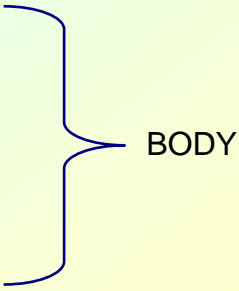
```
int gcd (int A, int B)
```

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

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

- 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);
}
```



BODY

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

```

#include <stdio.h>
/* Compute the GCD of four numbers */

main()
{
    int n1, n2, n3, n4, result;
    scanf ("%d %d %d %d", &n1, &n2, &n3, &n4);
    result = gcd ( gcd (n1, n2), gcd (n3, n4) );
    printf ("The GCD of %d, %d, %d and %d is %d \n",
           n1, n2, n3, n4, result);
}

```

## 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 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**.

## 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

```

Spring Semester 2007      Programming and Data Structure      15

## 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.

Spring Semester 2007      Programming and Data Structure      16



## Math Library Functions

<code>double acos(double x)</code>	– Compute arc cosine of x.
<code>double asin(double x)</code>	– Compute arc sine of x.
<code>double atan(double x)</code>	– Compute arc tangent of x.
<code>double atan2(double y, double x)</code>	– Compute arc tangent of y/x.
<code>double ceil(double x)</code>	– Get smallest integral value that exceeds x.
<code>double floor(double x)</code>	– Get largest integral value less than x.
<code>double cos(double x)</code>	– Compute cosine of angle in radians.
<code>double cosh(double x)</code>	– Compute the hyperbolic cosine of x.
<code>double sin(double x)</code>	– Compute sine of angle in radians.
<code>double sinh(double x)</code>	– Compute the hyperbolic sine of x.
<code>double tan(double x)</code>	– Compute tangent of angle in radians.
<code>double tanh(double x)</code>	– Compute the hyperbolic tangent of x.
<code>double exp(double x)</code>	– Compute exponential of x.
<code>double fabs (double x )</code>	– Compute absolute value of x.
<code>double log(double x)</code>	– Compute log to the base e of x.
<code>double log10 (double x )</code>	– Compute log to the base 10 of x.
<code>double pow (double x, double y)</code>	– Compute x raised to the power y.
<code>double sqrt(double x)</code>	– 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 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.**

## **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);  
}
```

## 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.

## Calling Functions: Call by Value and Call by Reference

- **Used when invoking functions.**
- **Call by value**
  - Copy of argument passed to function.
  - Changes in function do not affect original.
  - Use when function does not need to modify argument.
    - Avoids accidental changes.
- **Call by reference.**
  - Passes original argument (actually the pointer).
  - Changes in function affect original.
  - Only used with trusted functions.

**For now, we focus on call by value**

## 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)`

## Random Number Generation: Contd.

- **srand function**
  - Prototype defined in `<stdlib.h>`.
  - Takes an integer seed, and randomizes the random number generator.  
`srand (seed);`

```
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 }
```

# Program 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**

## #define: Macro definition

```
#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 macro substitution will be carried out?

`r = sqr(a) + sqr(30);` → `r = a*a + 30*30;`

`r = sqr(a+b);` → `r = a+b*a+b;`

- The macro definition should have been written as:

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

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

WRONG?

## 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)`

## Contd.

- 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

$$\begin{aligned} \text{fact}(n) &= 1, && \text{if } n = 0 \\ &= n * \text{fact}(n-1), && \text{if } n > 0 \end{aligned}$$

- **Examples:**

- **Factorial:**

$$\begin{aligned} \text{fact}(0) &= 1 \\ \text{fact}(n) &= n * \text{fact}(n-1), \text{ if } n > 0 \end{aligned}$$

- **GCD:**

$$\begin{aligned} \text{gcd}(m, m) &= m \\ \text{gcd}(m, n) &= \text{gcd}(m \% n, n), \text{ if } m > n \\ \text{gcd}(m, n) &= \text{gcd}(n, n \% m), \text{ if } m < n \end{aligned}$$

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

$$\begin{aligned} \text{fib}(0) &= 1 \\ \text{fib}(1) &= 1 \\ \text{fib}(n) &= \text{fib}(n-1) + \text{fib}(n-2), \text{ if } n > 1 \end{aligned}$$



## Example 1 :: Factorial

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

- **Mechanism of execution**

- **When a recursive program is executed, the recursive function calls are not executed immediately.**
  - **They are kept aside (on a stack) until the stopping condition is encountered.**
  - **The function calls are then executed in reverse order.**

## Example :: Calculating fact (4)

- First, the function calls will be processed:

```
fact(4) = 4 * fact(3)
fact(3) = 3 * fact(2)
fact(2) = 2 * fact(1)
fact(1) = 1 * fact(0)
```

- The actual values return in the reverse order:

```
fact(0) = 1
fact(1) = 1 * 1 = 1
fact(2) = 2 * 1 = 2
fact(3) = 3 * 2 = 6
fact(4) = 4 * 6 = 24
```

## 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:

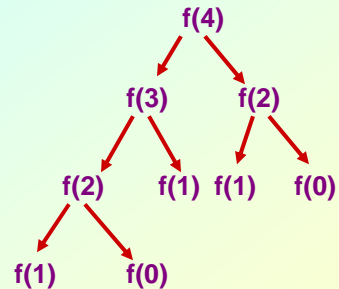
0, 1, 1, 2, 3, 5, 8, 13, 21, .....

- Function definition:

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

## Tracing Execution

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



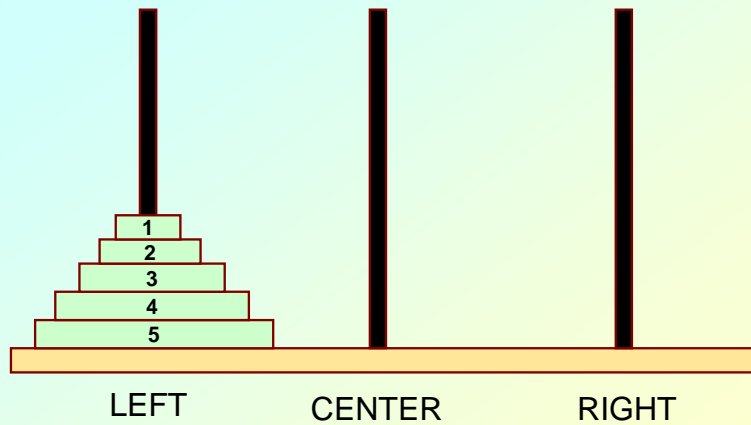
called 9 times

- Inefficiency:
  - Same thing is computed several times.

## Performance Tip

- Avoid Fibonacci-style recursive programs which result in an exponential “explosion” of calls.
- Avoid using recursion in performance situations.
- Recursive calls take time and consume additional memory.

### 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.
- 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:**

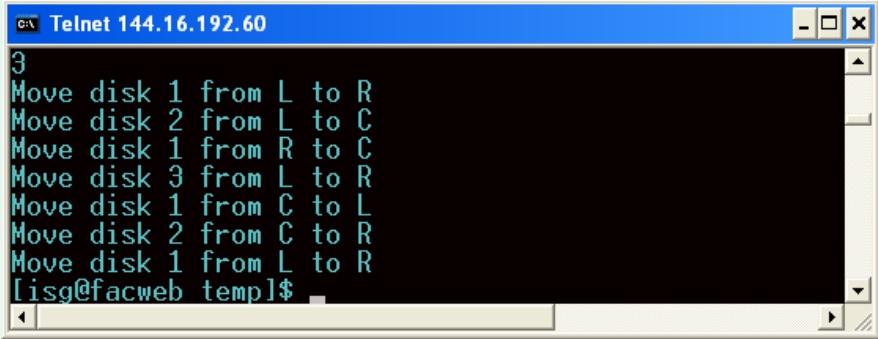
- Move the largest disk from LEFT to RIGHT.

- **Step 3:**

- Move the (n-1) disks from CENTER to RIGHT.

```
#include <stdio.h>
void transfer (int n, char from, char to, char temp);
main()
{
    int n; /* Number of disks */
    scanf ("%d", &n);
    transfer (n, 'L', 'R', 'C');
}

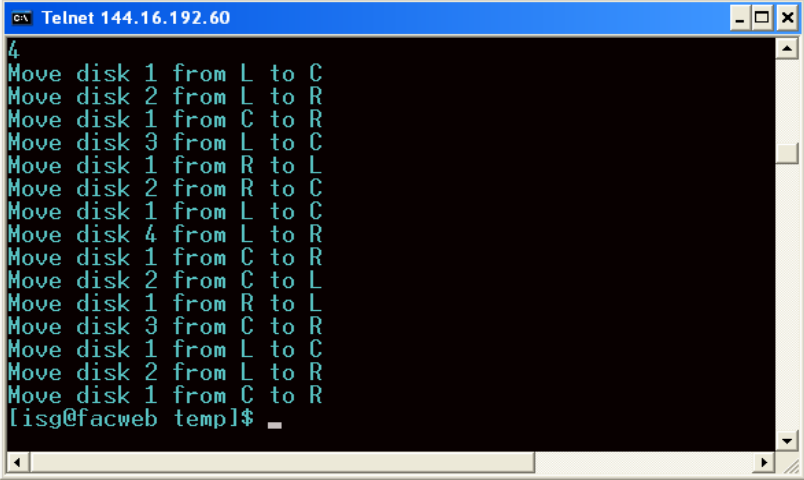
void transfer (int n, char from, char to, char temp)
{
    if (n > 0) {
        transfer (n-1, from, temp, to);
        printf ("Move disk %d from %c to %c \n", n, from, to);
        transfer (n-1, temp, to, from);
    }
    return;
}
```



A Telnet window titled "Telnet 144.16.192.60" with a blue title bar. The window contains a list of 7 moves and a prompt. The moves are: "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", and "Move disk 1 from L to R". The prompt is "[isg@facweb temp]\$" followed by a dollar sign.

```
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
[isg@facweb temp]$
```

Spring Semester 2007      Programming and Data Structure      43



A Telnet window titled "Telnet 144.16.192.60" with a blue title bar. The window contains a list of 16 moves and a prompt. The moves are: "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", and "Move disk 1 from C to R". The prompt is "[isg@facweb temp]\$" followed by a dollar sign.

```
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
[isg@facweb temp]$
```

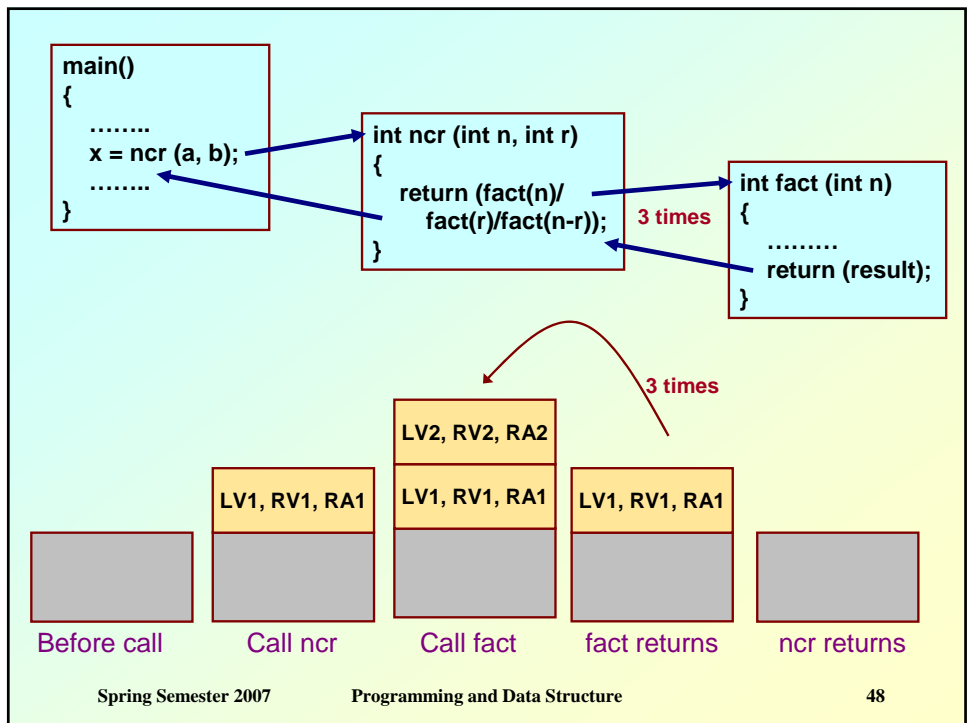
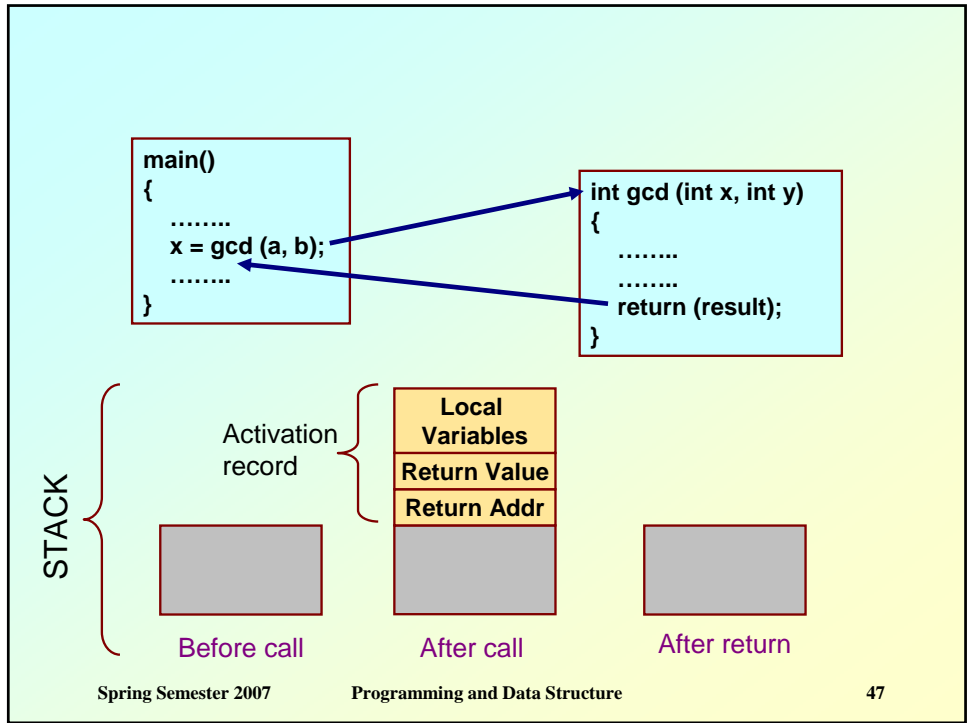
Spring Semester 2007      Programming and Data Structure      44

## 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?

- The following applies in general, with minor variations that are implementation dependent.
  - 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.





## What happens for recursive calls?

- **What we have seen ....**
  - Activation record gets pushed into the stack when a function call is made.
  - Activation record is popped off the stack when the function returns.
- **In recursion, a function calls itself.**
  - Several function calls going on, with none of the function calls returning back.
    - Activation records are pushed onto the stack continuously.
    - Large stack space required.

- Activation records keep popping off, when the termination condition of recursion is reached.
- **We shall illustrate the process by an example of computing factorial.**
  - Activation record looks like:

Local Variables
Return Value
Return Addr



## Do Yourself

- Trace the activation records for the following version of Fibonacci sequence.

```
#include <stdio.h>
int f (int n)
{
    int a, b;
    if (n < 2) return (n);
    else {
        a = f(n-1);
        b = f(n-2);
        return (a+b); }
}

main() {
    printf("Fib(4) is: %d \n", f(4));
}
```

X

Y

Local  
Variables  
(n, a, b)

Return Value

Return Addr  
(either main,  
or X, or Y)

## Storage Class of Variables

## What is Storage Class?

- It refers to the permanence of a variable, and its **scope** within a program.
- Four storage class specifications in C:
  - **Automatic:** `auto`
  - **External:** `extern`
  - **Static:** `static`
  - **Register:** `register`

## Automatic Variables

- These are always declared within a function and are local to the function in which they are declared.
  - **Scope is confined to that function.**
- **This is the default storage class specification.**
  - **All variables are considered as `auto` unless explicitly specified otherwise.**
  - **The keyword `auto` is optional.**
  - **An automatic variable does not retain its value once control is transferred out of its defining function.**

```

#include <stdio.h>

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

```

```

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

```

## Static Variables

- **Static variables are defined within individual functions and have the same scope as automatic variables.**
- **Unlike automatic variables, static variables retain their values throughout the life of the program.**
  - **If a function is exited and re-entered at a later time, the static variables defined within that function will retain their previous values.**
  - **Initial values can be included in the static variable declaration.**
    - **Will be initialized only once.**
- **An example of using static variable:**
  - **Count number of times a function is called.**

## EXAMPLE 1

```
#include <stdio.h>

int factorial (int n)
{
    static int count=0;
    count++;
    printf ("n=%d, count=%d \n", n, count);
    if (n == 0) return 1;
    else return (n * factorial(n-1));
}

main()
{
    int i=6;
    printf ("Value is: %d \n", factorial(i));
}
```

- **Program output:**

```
n=6, count=1
n=5, count=2
n=4, count=3
n=3, count=4
n=2, count=5
n=1, count=6
n=0, count=7
Value is: 720
```

## EXAMPLE 2

```
#include <stdio.h>

int fib (int n)
{
    static int count=0;
    count++;
    printf ("n=%d, count=%d \n", n, count);
    if (n < 2) return n;
    else return (fib(n-1) + fib(n-2));
}

main()
{
    int i=4;
    printf ("Value is: %d \n", fib(i));
}
```

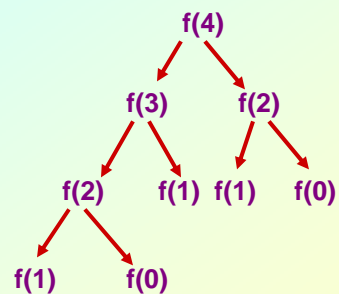
Spring Semester 2007

Programming and Data Structure

61

### • Program output:

```
n=4, count=1
n=3, count=2
n=2, count=3
n=1, count=4
n=0, count=5
n=1, count=6
n=2, count=7
n=1, count=8
n=0, count=9
Value is: 3
```



```
[0,1,1,2,3,5,8,...]
```

Spring Semester 2007

Programming and Data Structure

62

## Register Variables

- **These variables are stored in high-speed registers within the CPU.**
  - **Commonly used variables may be declared as register variables.**
  - **Results in increase in execution speed.**
  - **The allocation is done by the compiler.**

## External Variables

- **They are not confined to single functions.**
- **Their scope extends from the point of definition through the remainder of the program.**
  - **They may span more than one functions.**
  - **Also called global variables.**
- **Alternate way of declaring global variables.**
  - **Declare them outside the function, at the beginning.**



```

#include <stdio.h>

int count=0;  /** GLOBAL VARIABLE **/
int factorial (int n)
{
    count++;
    printf ("n=%d, count=%d \n", n, count);
    if (n == 0) return 1;
    else return (n * factorial(n-1));
}

main() {
    int i=6;
    printf ("Value is: %d \n", factorial(i));
    printf ("Count is: %d \n", count);
}

```

- **Program output:**

```

n=6, count=1
n=5, count=2
n=4, count=3
n=3, count=4
n=2, count=5
n=1, count=6
n=0, count=7
Value is: 720
Count is: 7

```

## Some Points on Class Test 1

- A conditional expression returns an integer value.
  - If the result of comparison is TRUE, 1 is returned.
  - If the result is FALSE, 0 is returned.

```
int a=10, b=20, c=30, d=753;
float x=5.0, y=12.5, z=7.5;
char p='A';
```

- |                                 |                |
|---------------------------------|----------------|
| a) $(a/b) * (\text{float}) c$   | ---- 0.0       |
| b) $((\text{float}) a) / b * c$ | ---- 15.0      |
| c) $(a++) + (--b) + c$          | ---- 59        |
| d) $x/a + y/5$                  | ---- 3.0       |
| e) $p + (c \% 8)$               | ---- 'G' or 71 |
| f) $(c/b) == (z/x)$             | ---- 0         |
| g) $(a\%4 == 3) ? c : b$        | ---- 20        |
| h) $(a + b / 3) != 16.6$        | ---- 1         |
| i) $(d - (d \% 10)) / 10$       | ---- 75        |
| j) $(a < b) \parallel (b < c)$  | ---- 1         |

1. A number represented in 8-bit 2's complement is 10001001. What is the value of the number in decimal?

-119

2. What is the smallest number in decimal that can be represented in 12-bit 2's complement representation?

-2048

3. What is the largest number in decimal that can be represented in 12-bit 2's complement representation?

2047

4. How many times will the following loops run?

`for (x=0; x=3; x++) { ... } ---- infinite loop`

`for (x=-2; x=0; x++) { ... } ---- zero times`

- The program for series summation.

$$S = 1/2 + 4/3 + 9/4 + 16/5 + \dots$$

$$S = 1/2 + 8/5 + 27/10 + 64/17 + \dots$$

Many students computed the next term to be added using integer arithmetic, which is wrong.

## Students Penalized due to Unfair Practices

- **5% of total marks will be deducted.**
  - Immediately deregistered if this happens again.

### Section 1:

06IM3018 Anant Kumar Jatav  
06IM1016 Bipin Kumar  
06CS1033 Anirudha Patra  
06EC3012 Piyush Gautam

### Section 2:

06AG3009 Piyush Khemka → DID NOT MEET ME

### Section 3:

06MF1015 Karan Mathur  
06MF3007 Uday Aghamarshan B.  
06EG1007 Neeraj Tulsyan  
06MI3006 Ritesh Kumar