Functions

CS10001: Programming & Data Structures

Pallab Dasgupta
Professor, Dept. of Computer Sc. & 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 functions may not return anything.
  - Return data type specified as "`void`".

```c
#include  <stdio.h>

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

```c
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.**
  - **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.**

```
#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*.

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);
}
```

```
#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);
}
```

Variable Scope

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 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***

“C supports only 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);
```

```c
/* A programming example
  Randomizing die-rolling program */
#include <stdlib.h>
#include <stdio.h>

int main()
{
   int i;
   unsigned seed;

   printf( "Enter seed: " );
   scanf( "%u", &seed );
   srand( seed );

   for ( i = 1; i <= 10; i++ ) {
      printf( "%10d ", 1 + ( rand() % 6 ) );

      if ( i % 5 == 0 )
         printf( "\n" );
   }

   return 0;
}
```

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 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);
```

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

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

- **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));
}
```

Example 1 :: Factorial Execution

fact(4)

24

```
if  (4 = = 1)
return (1);
else return (4 *
fact(3));
```

6

```
if  (3 = = 1)
return (1);
else return (3 *
fact(2));
```

2

```
if  (2 = = 1)
return (1);
else return  (2 *
fact(1));
```

1

```
if  (1 = = 1)
return (1);
else return  (1 *
fact(0));
```

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

De

Example 2 :: Fibonacci number

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

$$f(0) = 0$$
$$f(1) = 1$$
$$f(n) = f(n-1) + f(n-2), \quad \text{if } n > 1$$

  - The successive Fibonacci numbers are:

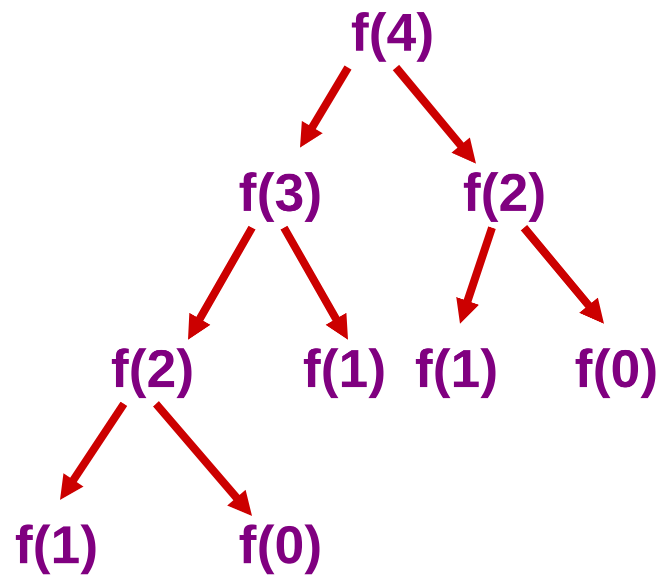
    0, 1, 1, 2, 3, 5, 8, 13, 21, …..

- Function definition:

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

Tracing Execution

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



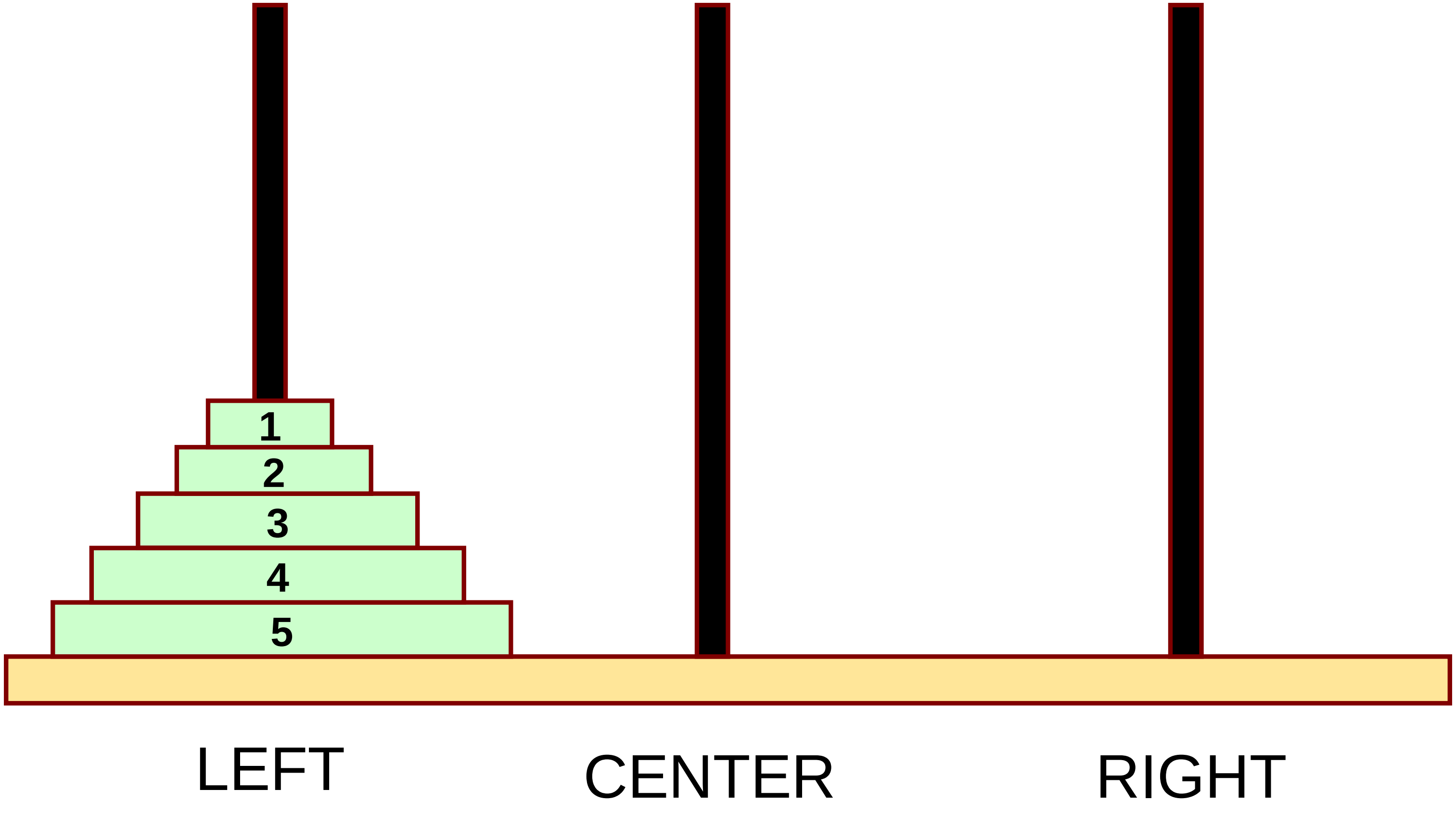
- Inefficiency:
  - Same thing is computed several times.

called 9 times

Notable Point

- Every recursive program can also be written without recursion
- Recursion is used for programming convenience, not for performance enhancement
- Sometimes, if the function being computed has a nice recurrence form, then a recursive code may be more readable

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;
}
```

Telnet 144.16.192.60

```
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]$
```

Telnet 144.16.192.60

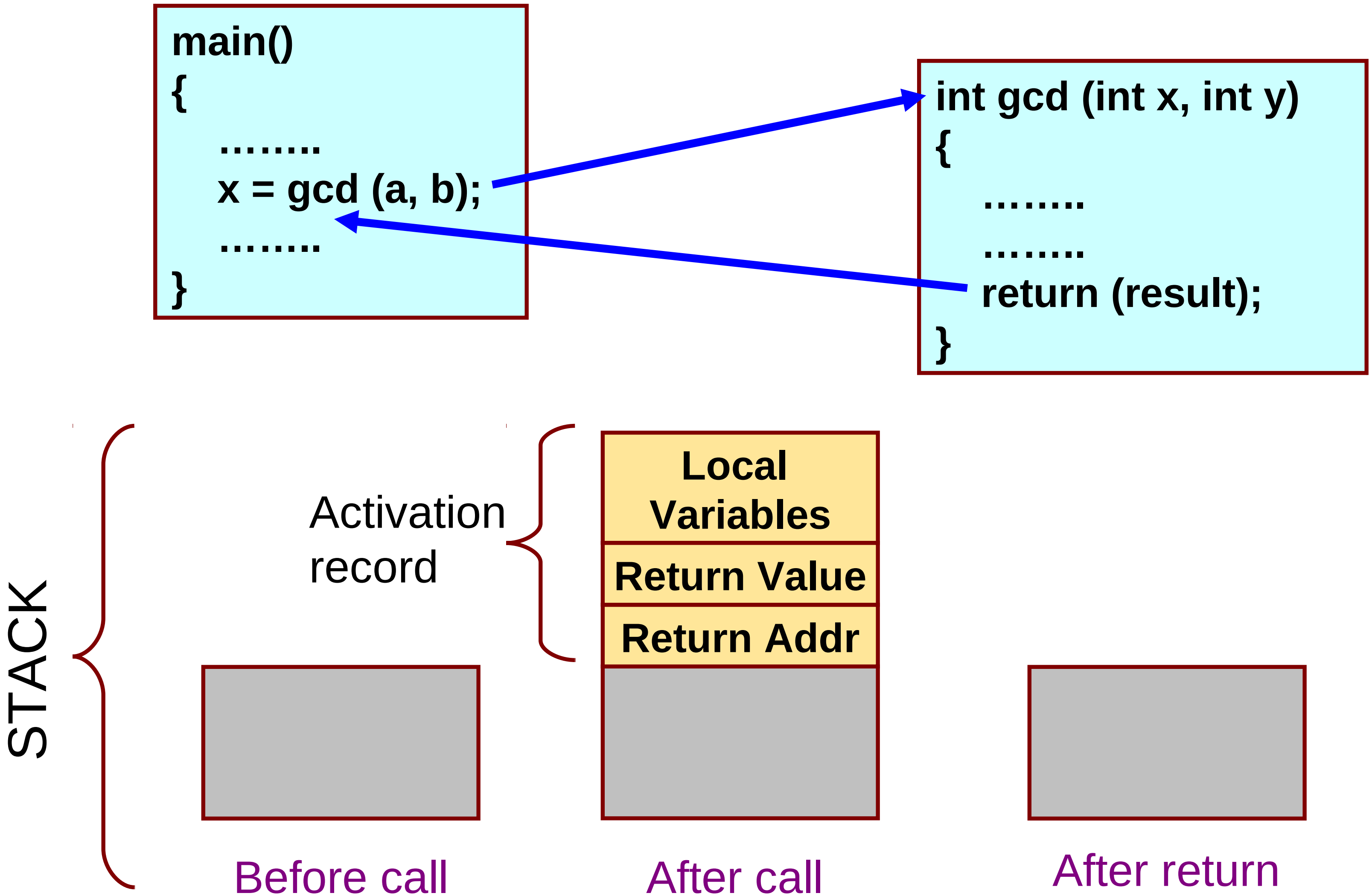
```
4
Move disk 1 from L to C
Move disk 2 from L to R
Move disk 1 from C to R
Move disk 3 from L to C
Move disk 1 from R to L
Move disk 2 from R to C
Move disk 1 from L to C
Move disk 4 from L to R
Move disk 1 from C to R
Move disk 2 from C to L
Move disk 1 from R to L
Move disk 3 from C to R
Move disk 1 from L to C
Move disk 2 from L to R
Move disk 1 from C to R
[isg@facweb temp]$
```

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.

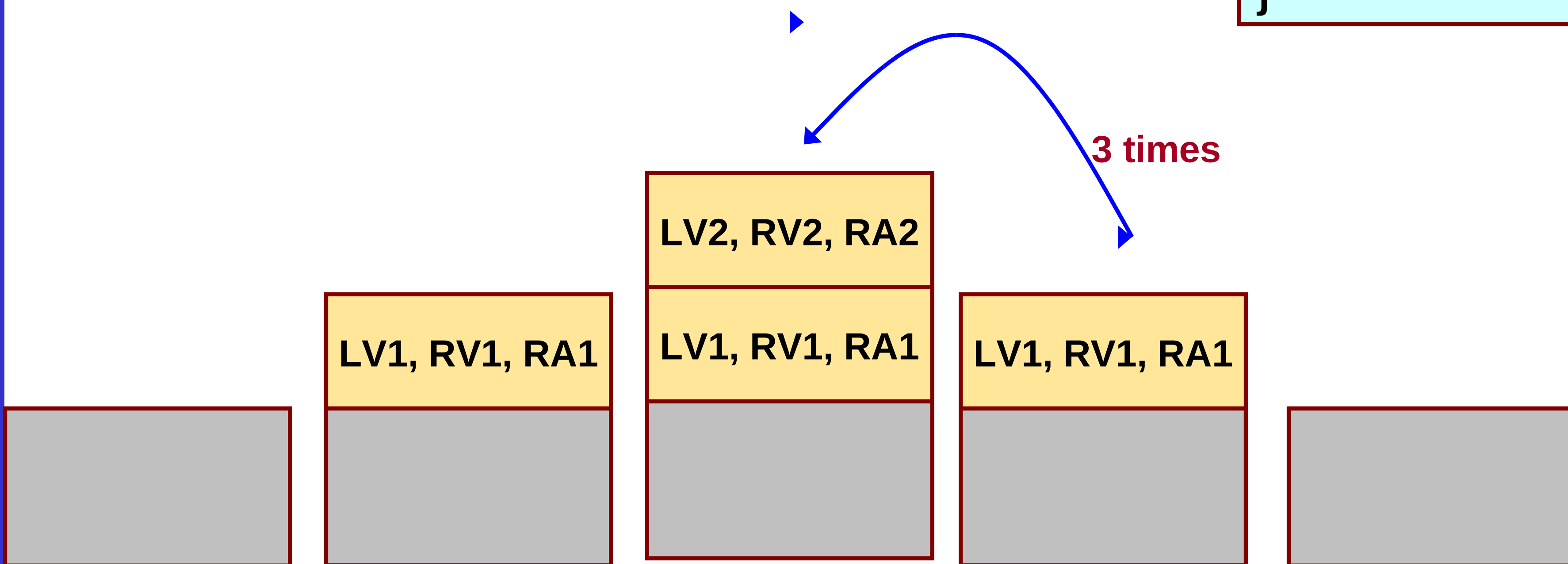

main()
{
……..
x = gcd (a, b);
……..
}
int gcd (int x, int y)
{
……..
……..
return (result);
}
STACK
Activation record
Local Variables
Return Value
Return Addr
Before call
After call
After return

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

```
int ncr (int n, int r)
{
    return (fact(n)/
        fact(r)/fact(n-r));
}
```

3 times

```
int fact (int n)
{
    .........
    return (result);
}
```



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:

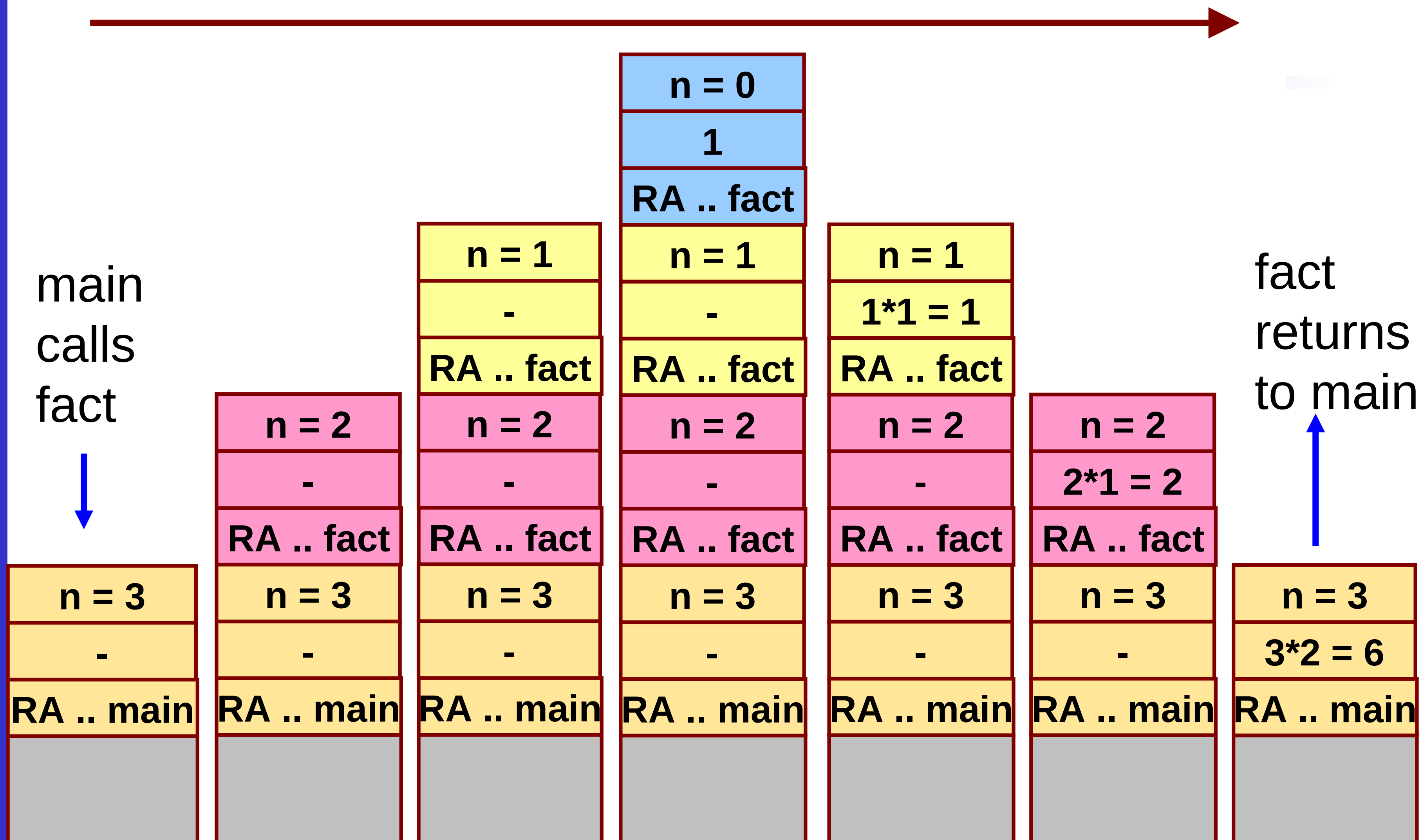
Local Variables
Return Value
Return Addr

Example:: main() calls fact(3)

```
main()
{
   int  n;
   n = 3;
   printf (“%d \n”, fact(n) );
}
```

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

TRACE OF THE STACK DURING EXECUTION



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

main

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.

```c
#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);
}
```

```c
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

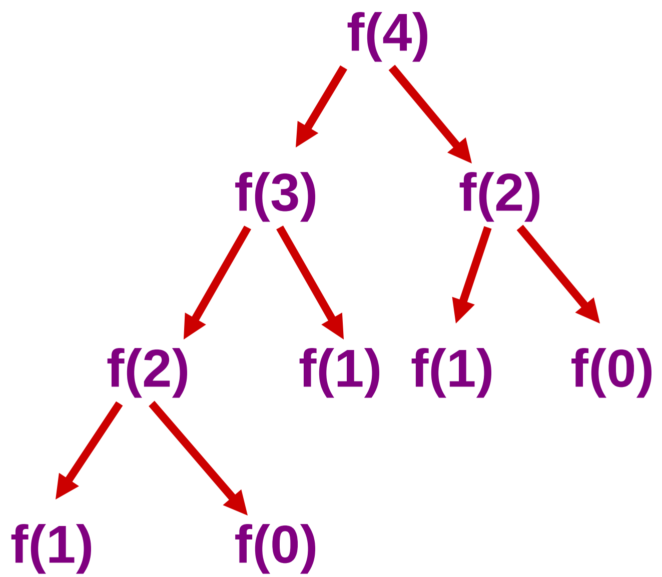
```
#include <stdio.h>

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));
}
```

- **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,….]

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
```