# Functions

# 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) );
}
```

# Functions: Why?

- **Functions**
  - **Modularize a program**
  - **All variables declared inside functions are local variables**
    - **Known only in function defined**
  - **Parameters**
    - **Communicate information between functions**
    - **They also become local variables.**
- **Benefits**
  - **Divide and conquer**
    - **Manageable program development**
  - **Software reusability**
    - **Use existing functions as building blocks for new programs**
    - **Abstraction - hide internal details (library functions)**
  - **Avoids code repetition**

# 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:**

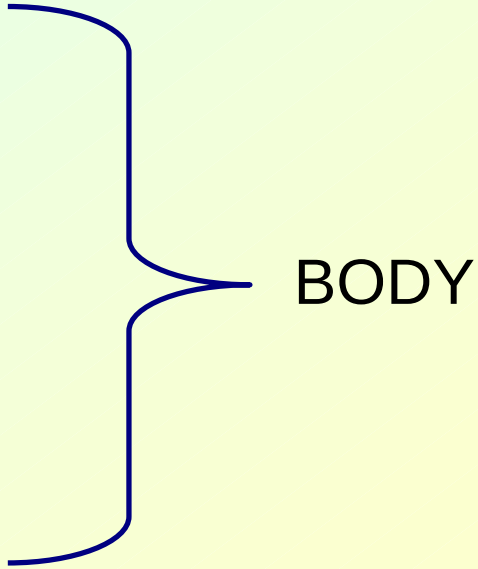
```
int  gcd  (int  A,  int  B)
```

- **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);
}
```



- 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 only prints if a number if divisible by 7 or not.**

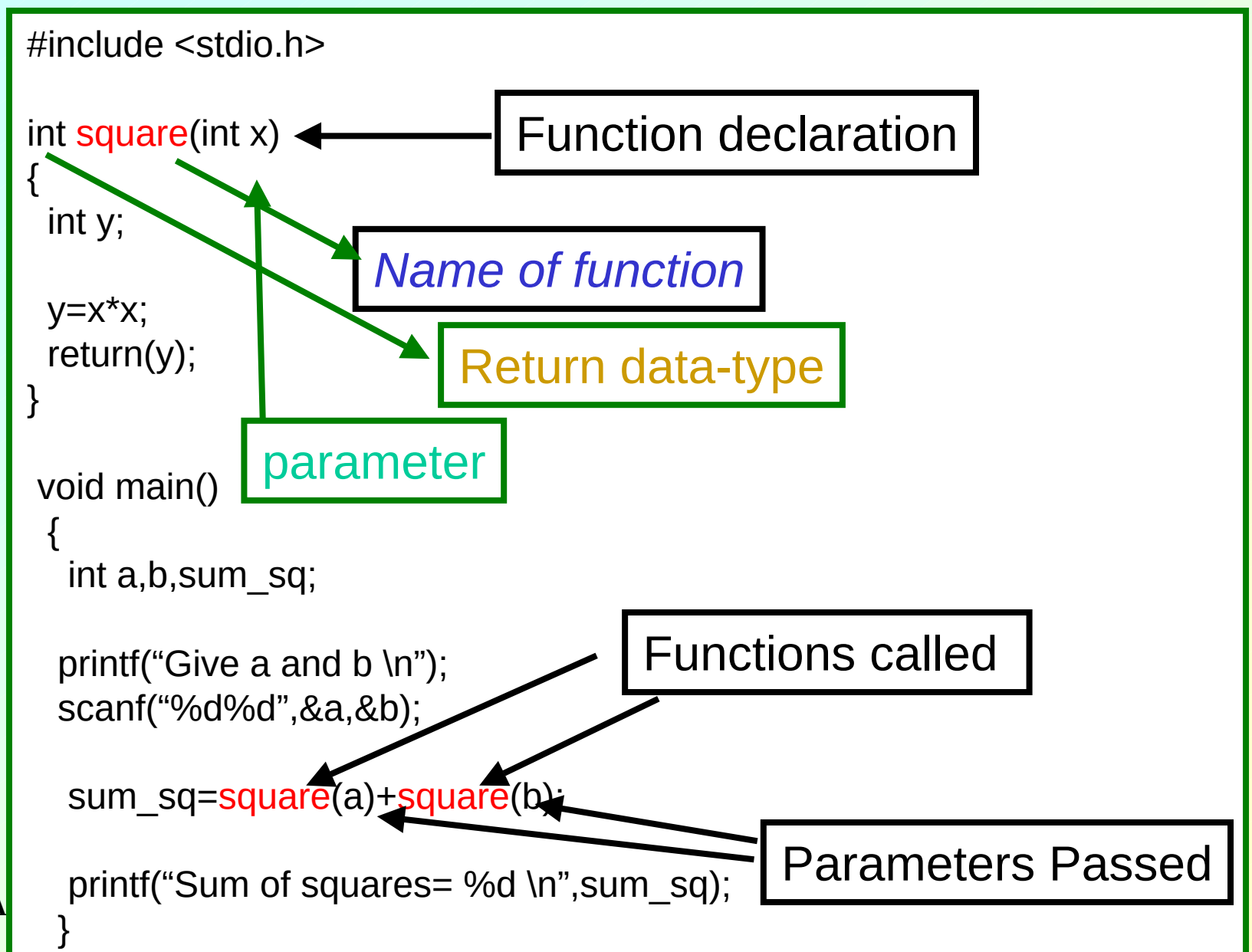
```
void  div7 (int n)
{
   if  ((n % 7) == 0)
        printf ("%d is divisible by 7", n);
   else
        printf ("%d is not divisible by 7", n);
   return;
}
```

OPTIONAL

- **Returning control**
  - **If nothing returned**
    - `return;`
    - **or, until reaches right brace**
  - **If something returned**
    - `return` *`expression`*`;`

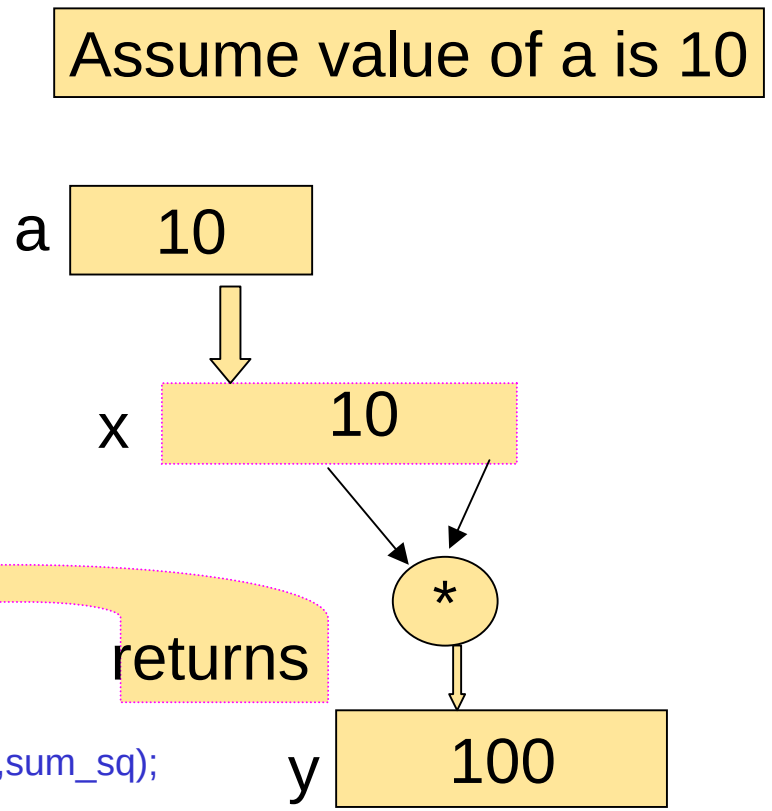
# Function: An Example

```
#include <stdio.h>

int square(int x)
{
  int y;

  y=x*x;
  return(y);
}

 void 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);
   }
```



# Invoking a function call : An Example

```
#include <stdio.h>

int square(int x)
{
  int y;

  y=x*x;
  return(y);
}

 void 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);
  }
```



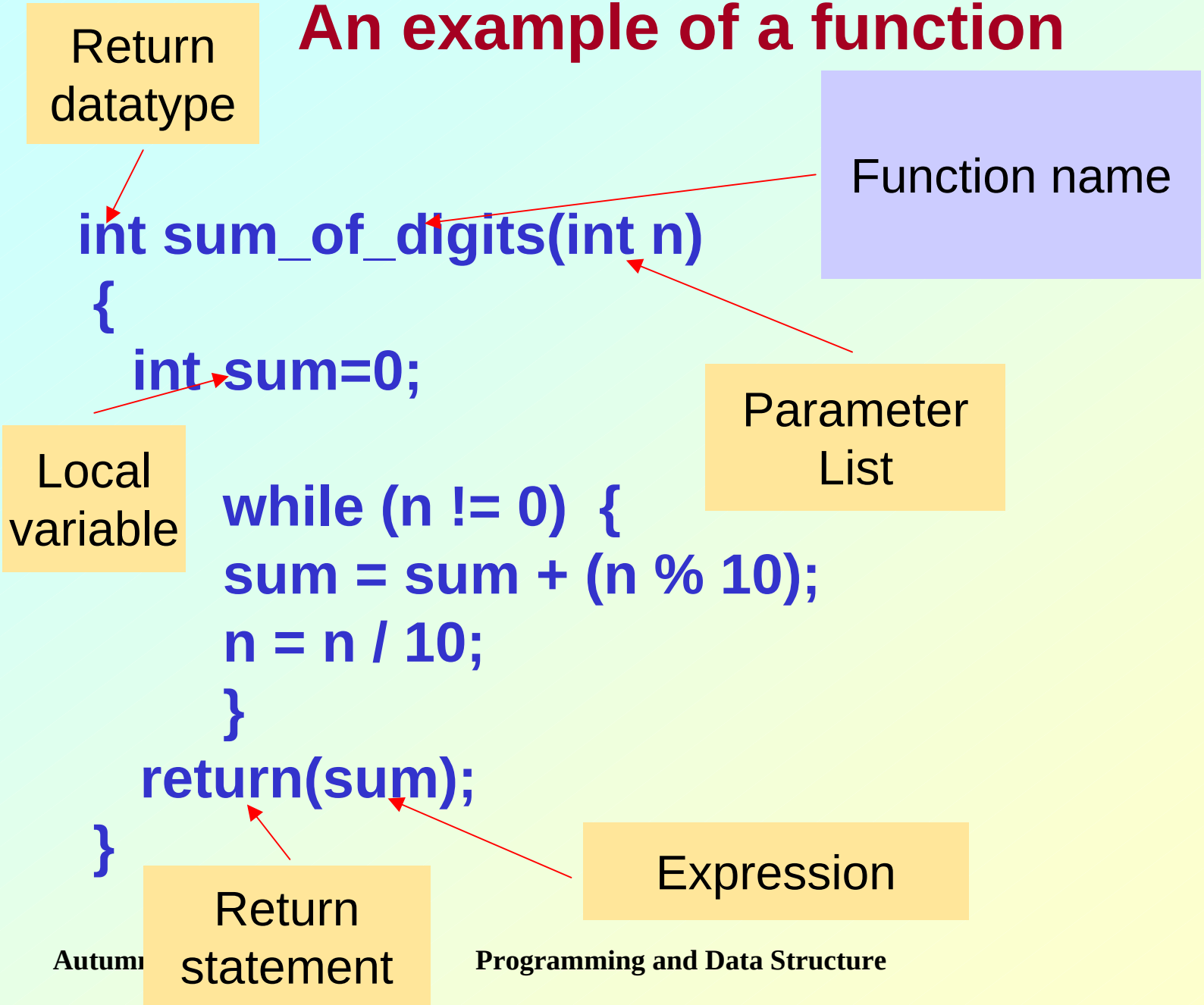
# Function Definitions

- **Function definition format (continued)**

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

  - **Declarations and statements: function body (block)**
    - **Variables can be declared inside blocks (can be nested)**
    - **Function can not be defined inside another function**
  - **Returning control**
    - **If nothing returned**
      - `return;`
      - **or, until reaches right brace**
    - **If something returned**
      - `return` ***expression*** `;`

# An example of a function



# Variable Scope

```
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);
}
. . .
```

Printout:

---

**A = 3**

**A = 2**

**A = 1**

# Function: Summary

```
#include  <stdio.h>

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

*Returned data-type*

parameter

Function name

Local vars

Return statement

Self contained programme

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

main()
is a function

Calling a function

# 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.

# Math Library Functions

- **Math library functions**
  - **perform common mathematical calculations**
  - `#include <math.h>`
  - `cc <prog.c> -lm`

- **Format for calling functions**

  `FunctionName (argument);`
    - **If multiple arguments, use comma-separated list**
  - `printf( "%.2f", 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 ceil(double x) -- Get smallest integral value that exceeds x.**

  **double floor(double x) -- Get largest integral value less than 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.**
- **double exp(double x -- Compute exponential of x**
  **double fabs (double x ) -- Compute absolute value of x.**
  **double log(double x) -- Compute log(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.**

# More about scanf and printf

# Entering input data :: scanf function

- **General syntax:**

  **scanf (control string, arg1, arg2, …, argn);**

  – **“control string refers to a string typically containing data types of the arguments to be read in;**

  – **the arguments arg1, arg2, … represent pointers to data items in memory.**

  **Example: scanf (%d %f %c”, &a, &average, &type);**

- **The control string consists of individual groups of characters, with one character group for each input data item.**

  – **‘%’ sign, followed by a conversion character.**

- Commonly used conversion characters:

| | |
|---|---|
| c | single character |
| d | decimal integer |
| f | floating-point number |
| s | string terminated by null character |
| X | hexadecimal integer |

- We can also specify the maximum field-width of a data item, by specifying a number indicating the field width before the conversion character.

  Example: `scanf ("%3d %5d", &a, &b);`

# Writing output data :: printf function

- **General syntax:**

  **printf (control string, arg1, arg2, …, argn);**

  – **“control string refers to a string containing formatting information and data types of the arguments to be output;**

  – **the arguments arg1, arg2, … represent the individual output data items.**

- **The conversion characters are the same as in scanf.**

- **Examples:**

```
printf  (“The average of %d and %d is %f”, a, b, avg);
printf  (“Hello \nGood \nMorning \n”);
printf (“%3d %3d %5d”, a, b, a*b+2);
printf (“%7.2f  %5.1f”, x, y);
```

- **Many more options are available:**
  - **Read from the book.**
  - **Practice them in the lab.**
- **String I/O:**
  - **Will be covered later in the class.**

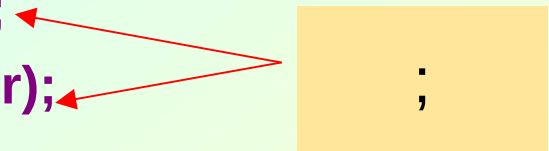
# Function Prototypes

- 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 Prototype (Contd.)

- 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.

# Function Prototype: Examples

```
#include  <stdio.h>

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

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);
}
```

Prototype declaration

```
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

# Header Files

- **Header files**
  - **contain function prototypes for library functions**
  - `<stdlib.h>`, `<math.h>`, **etc**
  - **Load with**

```
#include <filename>
```

  - `#include <math.h>`

- **Custom header files**
  - **Create file with functions**
  - **Save as** `filename.h`
  - **Load in other files with** `#include "filename.h"`
  - **Reuse functions**

```c
 /* Finding the maximum of three integers */
#include <stdio.h>

int maximum( int, int, int );   /* function prototype */

int main()
{
   int a, b, c;

   printf( "Enter three integers: " );
   scanf( "%d%d%d", &a, &b, &c );
   printf( "Maximum is: %d\n", maximum( a, b, c ) );

   return 0;
}

/* Function maximum definition */
int maximum( int x, int y, int z )
{
   int max = x;

   if ( y > max )
      max = y;

   if ( z > max )
      max = z;

   return max;
}
```

Prototype Declaration

Function Calling

Function Definition

# 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 effect original**
  - **Use when function does not need to modify argument**
    - **Avoids accidental changes**
- **Call by reference**
  - **Passes original argument**
  - **Changes in function effect original**
  - **Only used with trusted functions**
- **For now, we focus on call by value**

# An Example: Random Number Generation

- `rand` **function**
  - **Prototype defined in** `<stdlib.h>`
  - **Returns "random" number between** `0` **and** `RAND_MAX` **(at least 32767)**

    ```
    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 )
    ```
    - `rand % n` **returns a number between** `0` **and** `n-1`
    - **Add** `1` **to make random number between** `1` **and** `n`

      ```
      1 + ( rand() % 6)   // number between 1 and 6
      ```

# Random Number Generation: Contd.

- `srand` **function**
  - `Prototype defined in <stdlib.h>`
  - **Takes an integer seed - jumps to location in "random" sequence**

```
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 }
```

## Algorithm

1. Initialize seed
2. Input value for seed
   - 2.1 Use srand to change random sequence
   - 2.2 Define Loop
3. Generate and output random numbers

# 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
```

# #include: Revisited

- Preprocessor statement in the following form

  #include “filename”

- Filename could be specified with complete path.

  #include “/usr/home/rajan/myfile.h”

- The content of the corresponding file will be included in the present file before compilation and the compiler will compile thereafter considering the content as it is.

# #include: Contd.

```
#include <stdio.h>
int x;

main()
{
  printf("Give value of x \n)";
 scanf("%d",&x);
  printf("Square of x=%d \n",x*x);
}
```

prog.c

```
#include <stdio.h>

int x;
```

myfile.h

/usr/include/filename.h

#include <filename.h>

It includes the file "filename.h" from a specific directory known as include directory.

# #define: Macro definition

```
#include <stdio.h>
#define PI 3.14
main()
{
  float r=4.0,area;
  area=PI*r*r;
}
```

```
#include <stdio.h>
main()
{
  float r=4.0,area;
  area=3.14*r*r;
}
```

# #define with argument

- **#define statement may be used with argument e.g.**

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

```
#include <stdio.h>


main()
{
  int y=5;
  printf("value=%d \n", ((y)*(y))+3);
}
```

sqr(x) written as macro definition?

Which one is faster to execute?

sqr(x) written as an ordinary function?

# #define with arguments: A Caution

- **#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;**

    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

fact(n) = 1, if n = 0

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

- **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 = = 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

# Another Example :: 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:

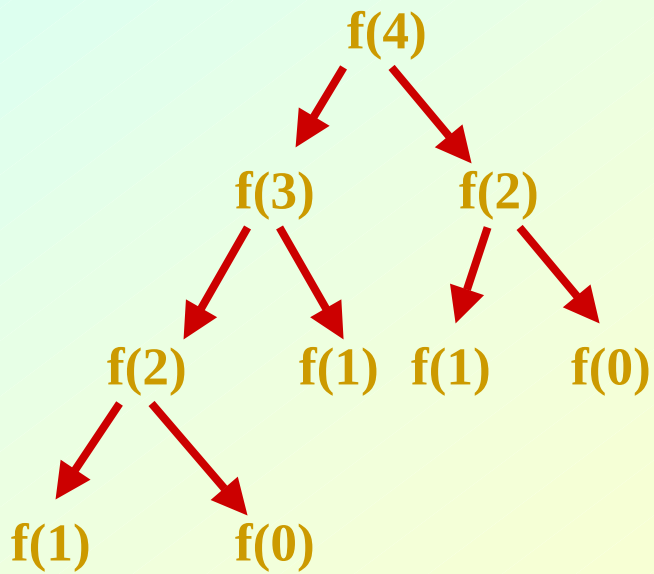
    0, 1, 1, 2, 3, 5, 8, 13, 21, .....

- 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) ?




- Inefficiency:
  - Same thing is computed several times.

**9 times**

# Example Codes: fibonacci()

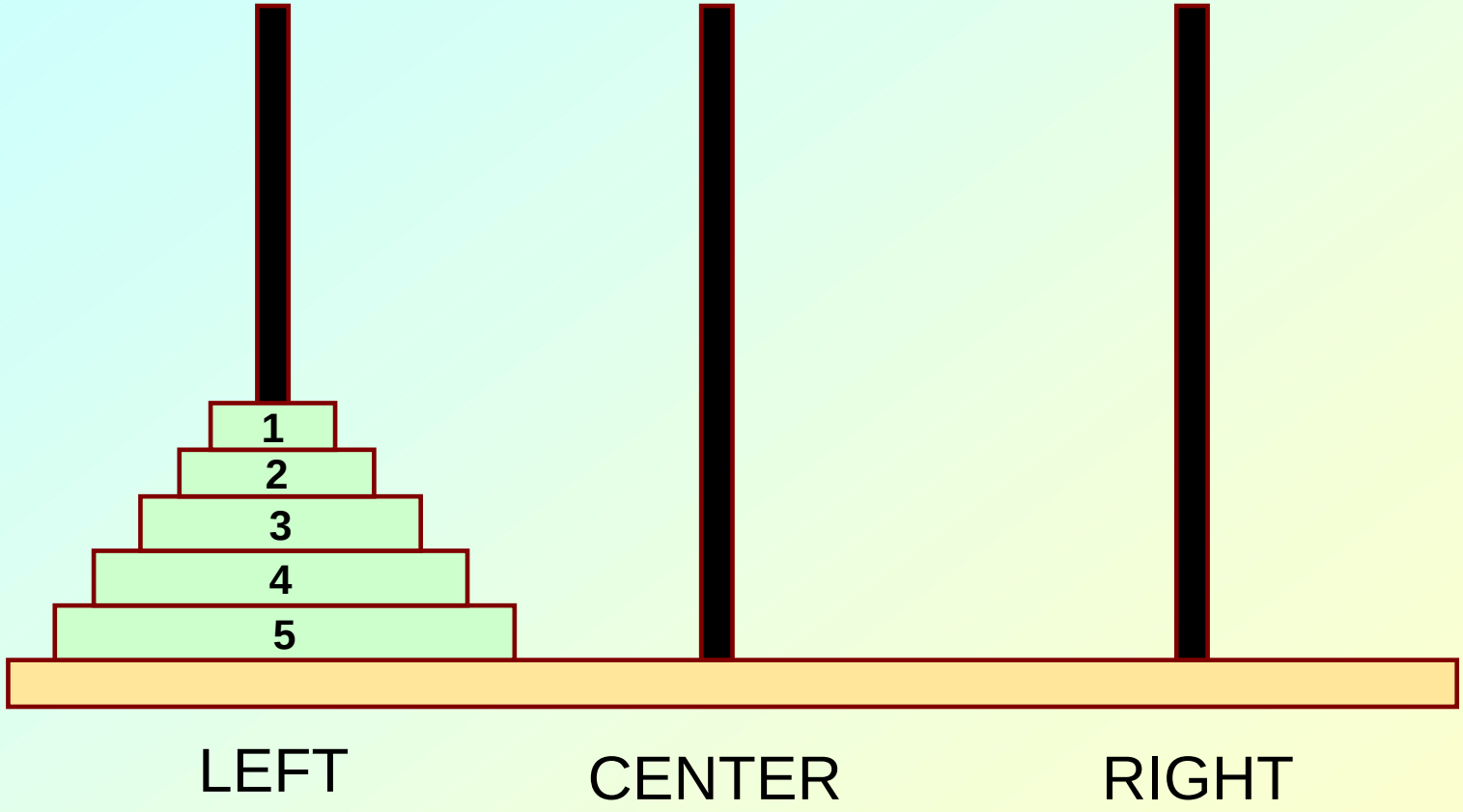
- Code for the `fibonacci` function

```
long fibonacci( long n )
{
  if (n == 0 || n == 1)  // base case
     return n;
  else
     return fibonacci( n - 1) +
        fibonacci( n – 2 );
}
```

# Performance Tip

- **Avoid Fibonacci-style recursive programs which result in an exponential “explosion” of calls.**

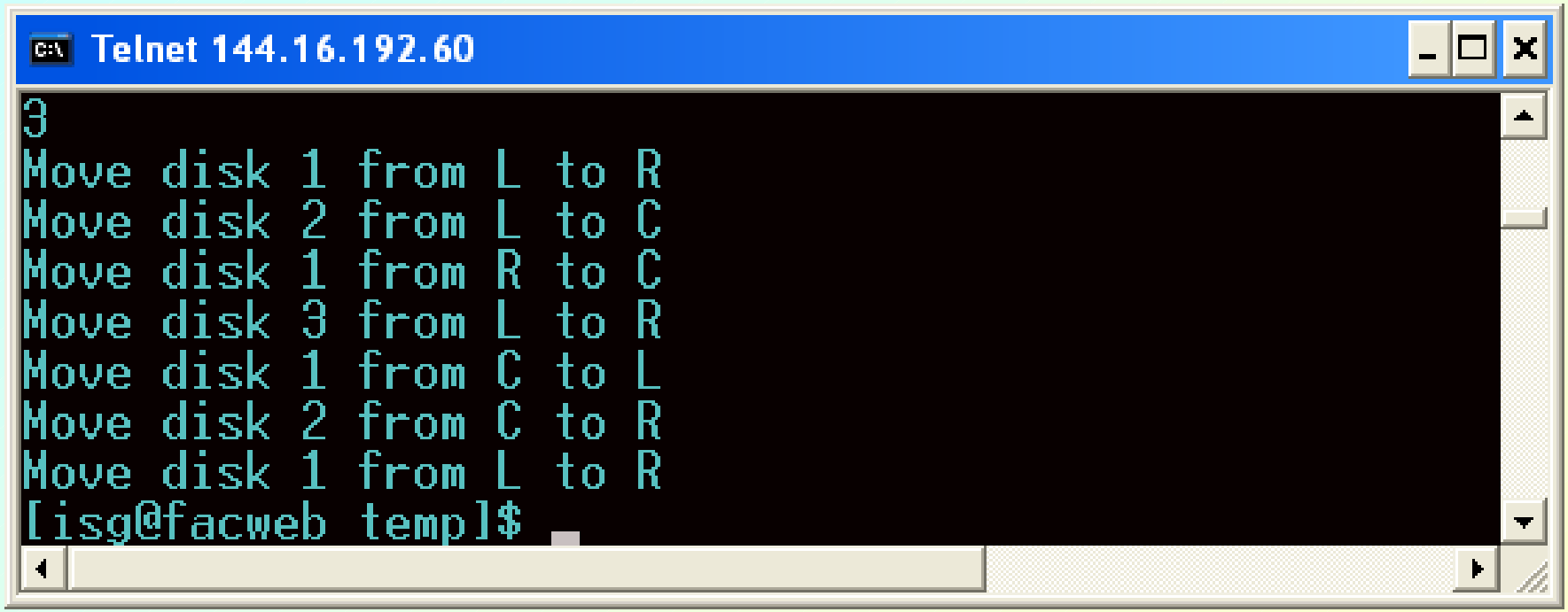
# Example: Towers of Hanoi Problem

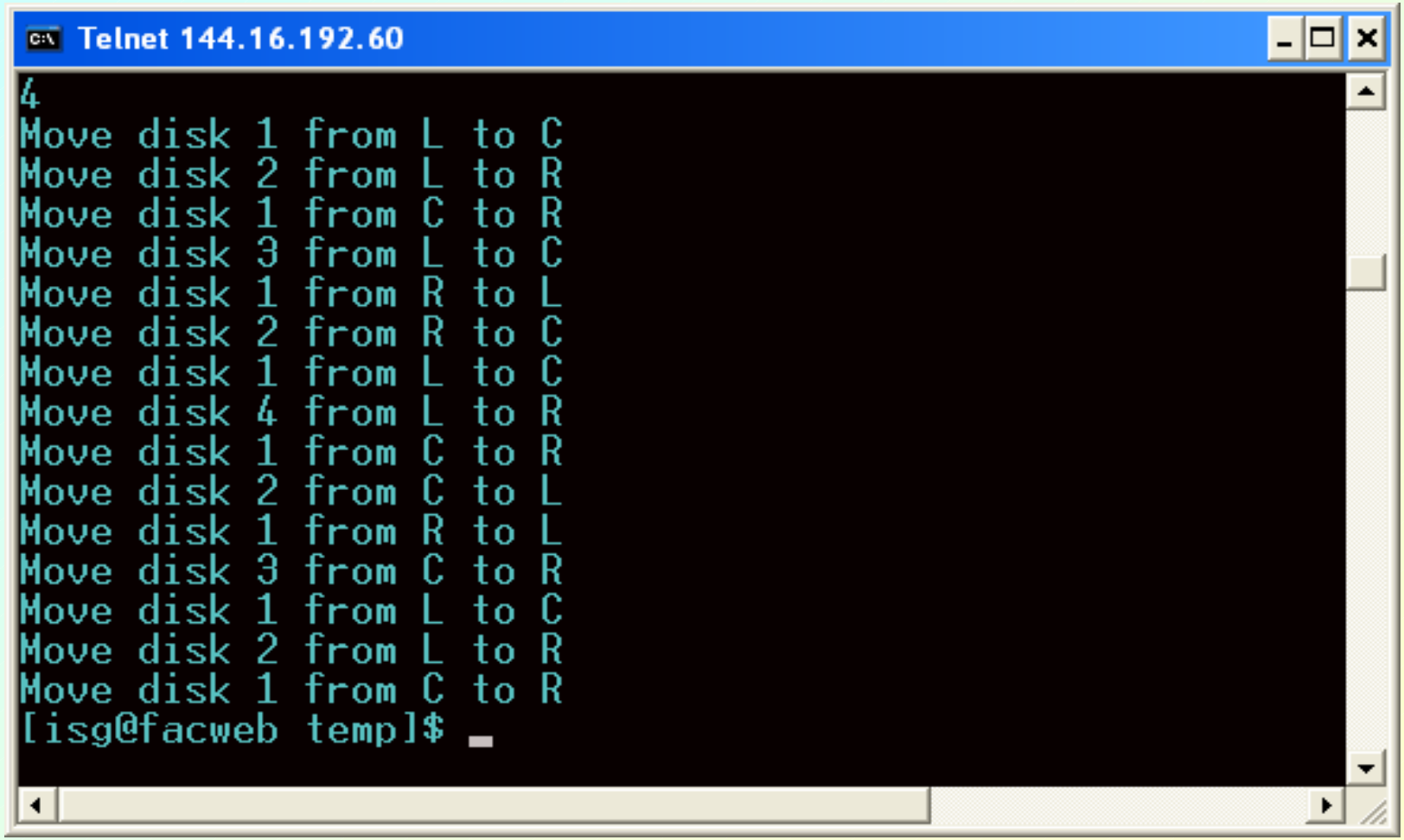


- **The problem statement:**
  - **Initially all the disks are stacked on the LEFT pole.**
  - **Required to transfer all the disks to the RIGHT pole.**
    - **Only one disk can be moved at a time.**
    - **A larger disk cannot be placed on a smaller disk.**

- **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

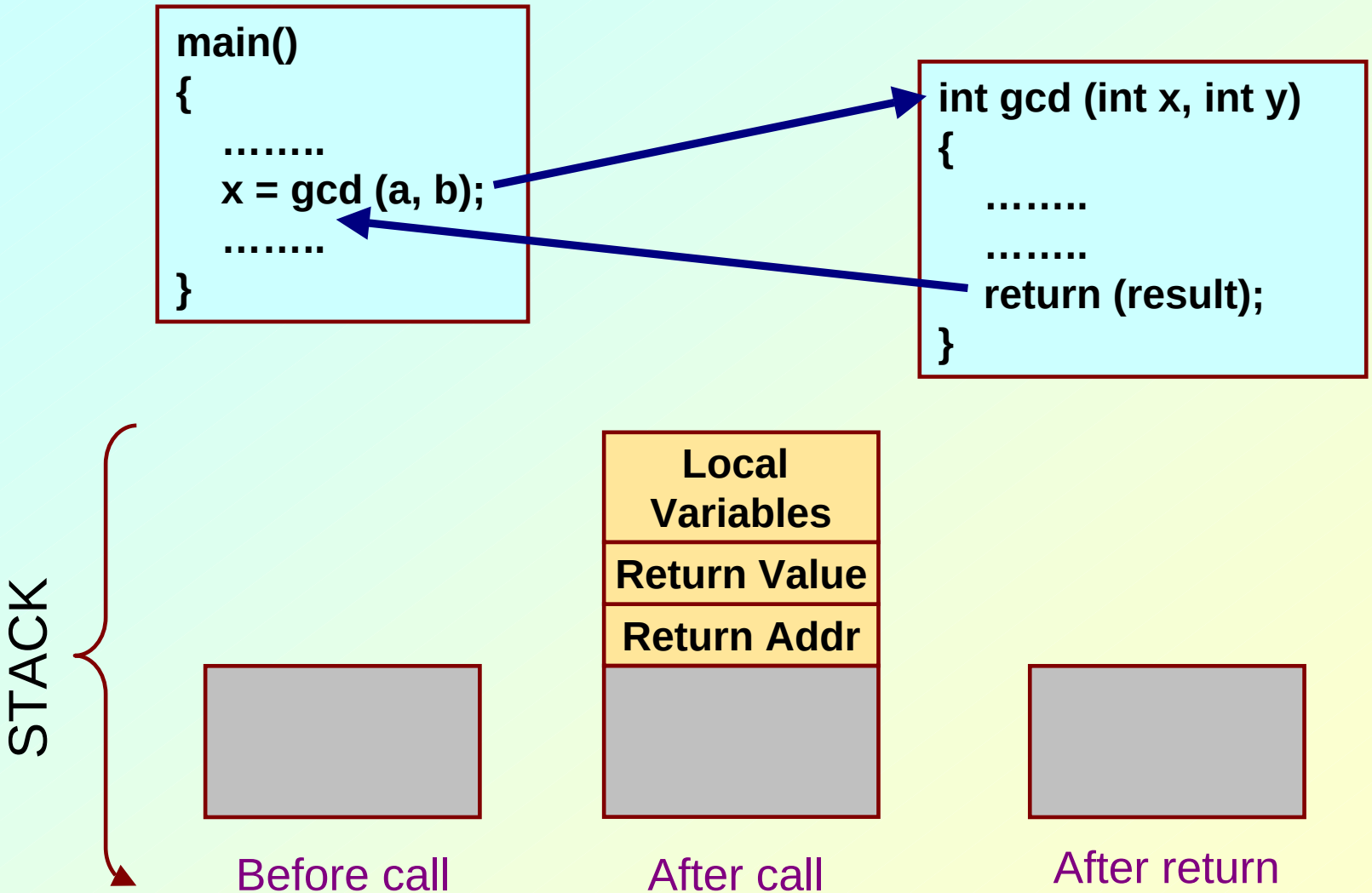
- Repetition
  - Iteration:  explicit loop
  - Recursion:  repeated function calls
- Termination
  - Iteration: loop condition fails
  - Recursion: base case recognized
- Both can have infinite loops
- Balance
  - Choice between performance (iteration) and good software engineering (recursion)

# Performance Tip

- **Avoid using recursion in performance situations. Recursive calls take time and consume additional memory.**

# 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.


main()
{
……..
x = gcd (a, b);
……..
}
int gcd (int x, int y)
{
……..
……..
return (result);
}
STACK
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);
}
```