Functions

Palash Dey
Department of Computer Science & Engg.
Indian Institute of Technology
Kharagpur

Slides credit: Prof. Indranil Sen Gupta

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 return 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);
}
```

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

Output:

```
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

```
#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=11; n<=20; n++)
        printf (″%d! = %d \n″,
                n, factorial (n));
}
```

Output:

```
11! = 39916800
12! = 479001600
13! = 1932053504
14! = 1278945280
15! = 2004310016
16! = 2004189184
17! = -288522240
18! = -898433024
19! = 109641728
20! = -2102132736
```

```
#include  <stdio.h>

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

```
int main()
{
   int  n;
   for  (n=11; n<=20; n++)
        printf (″%d! = %ld \n″,
                n, factorial (n));
}
```

Output:

```
11! = 39916800
12! = 479001600
13! = 6227020800
14! = 87178291200
15! = 1307674368000
16! = 20922789888000
17! = 355687428096000
18! = 6402373705728000
19! = 121645100408832000
20! = 2432902008176640000
```

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 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*`;`

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 A
  - 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 ceil(double x) – Get smallest integer 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 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.

An example

```c
#include <stdio.h>
#include <math.h>

int main()
{
   double value, result;
   float  a, b;

   value = 2345.6;   a = 23.5;
   result = sqrt(value);
   b = pow(23.5,4);

   printf ("\nresult = %f, b = %f", result, b);
}
```

Must be compiled as:

```
gcc examp.c -lm
```

Link math library

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:: function prototypes

```
#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 the reference to the original argument.
  - Execution of the function may affect the original.
  - Not directly supported in C – can be effected 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);
```

```
#include <stdio.h>
#include <stdlib.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;
}
```

A programming example.
Randomizing die rolling program.

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 discr b*b-4*a*c
```

#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;
```

WRONG?

  - The macro definition should have been written as:

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

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

Recursion: Function calling itself

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 (0, n) = n
gcd (m, 0) = m
gcd (m, n) = m, if m = n
gcd (m, n) = gcd (m%n, n), if m > n
gcd (m, n) = gcd (m, n%m), if m < n
```

  - Fibonacci series (0, 1, 1, 2, 3, 5, 8, 13, ….)

```
fib (0) = 0
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));
}
```

Example 2 :: GCD

```
int  gcd (m, n)
int  m, n;
{
    if (m == 0) return n;
    if (n == 0) return m;
    if (m == n) return (m);
    if (m > n)
       return gcd (m%n, n);
    else
       return gcd (m, n%m);
}
```

- 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 3 :: 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:

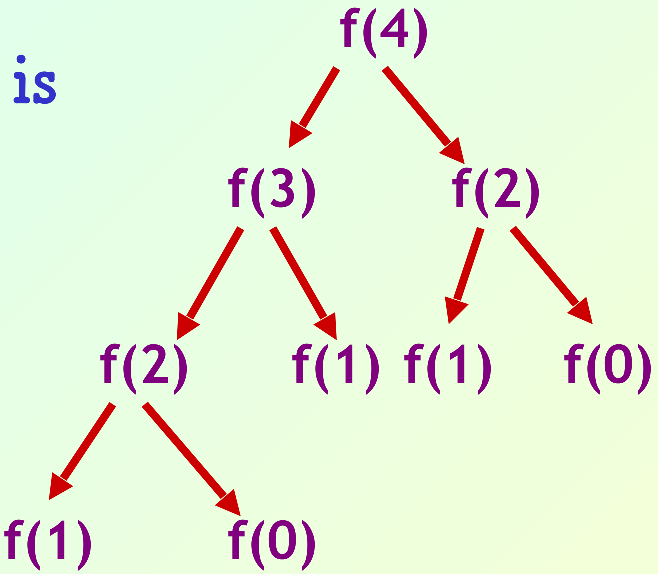
    0, 1, 1, 2, 3, 5, 8, 13, 21, …..

- 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

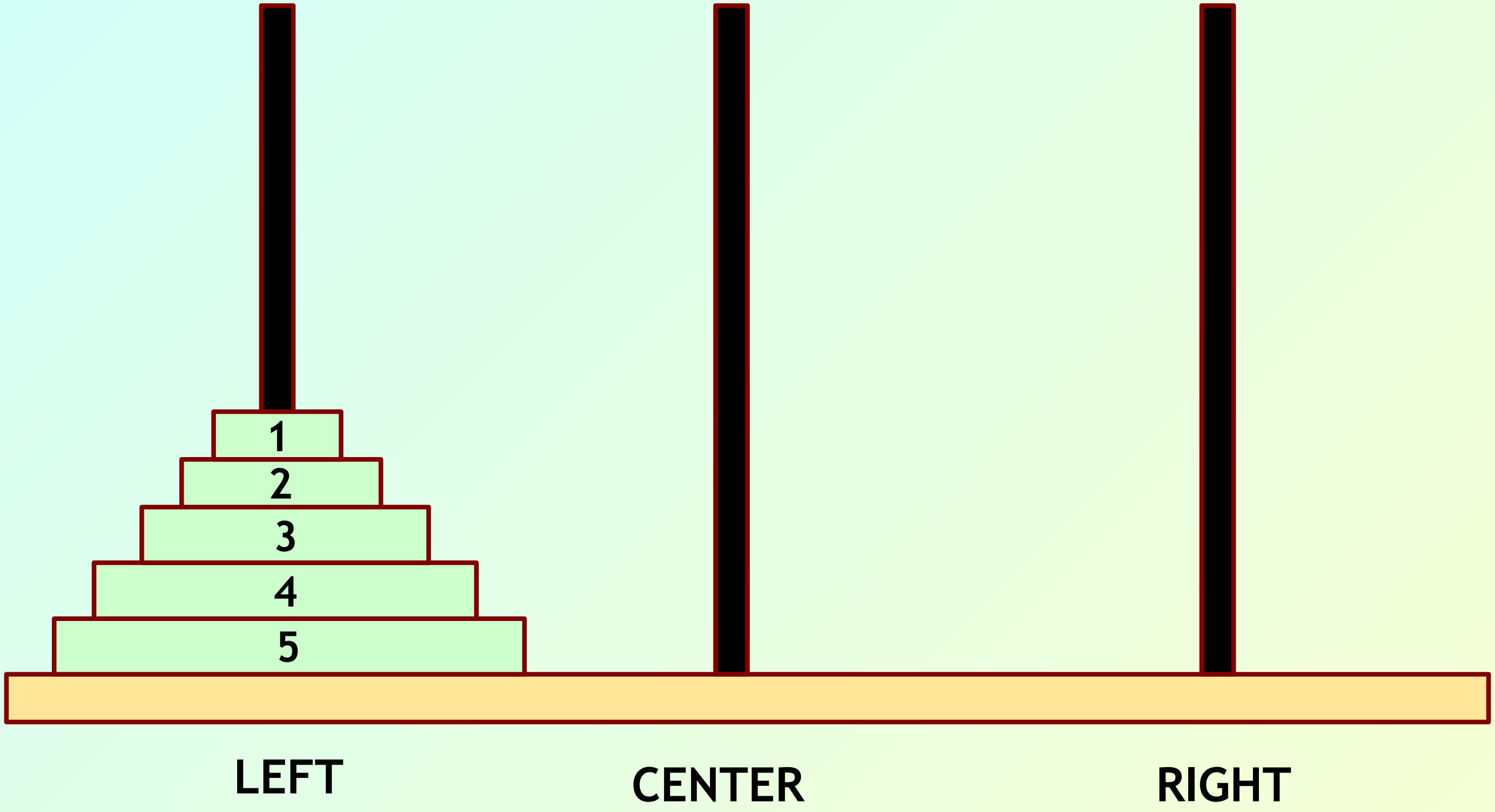
- Avoid Fibonacci-style recursive programs which result in an exponential "*explosion*" of calls.
- Avoid using recursion in performance situations.
- Recursive calls take time and consume additional memory.

Fibonacci number: iterative version

```c
#include <stdio.h>
int f (int x);

int main()
{
   printf (“\n %d %d %d %d”, f(2), f(3), f(4), f(5));
}

int  f (int n)
{
   int a = 0, b = 1, temp, i;
   for (i=2; i<=n; i++)
   {
       temp = a + b;
       a = b;
       b = temp;
   }
   return (b);
}
```

```
Output:
1 2 3 5
```

Example 4 :: 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;
}
```

```
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
```

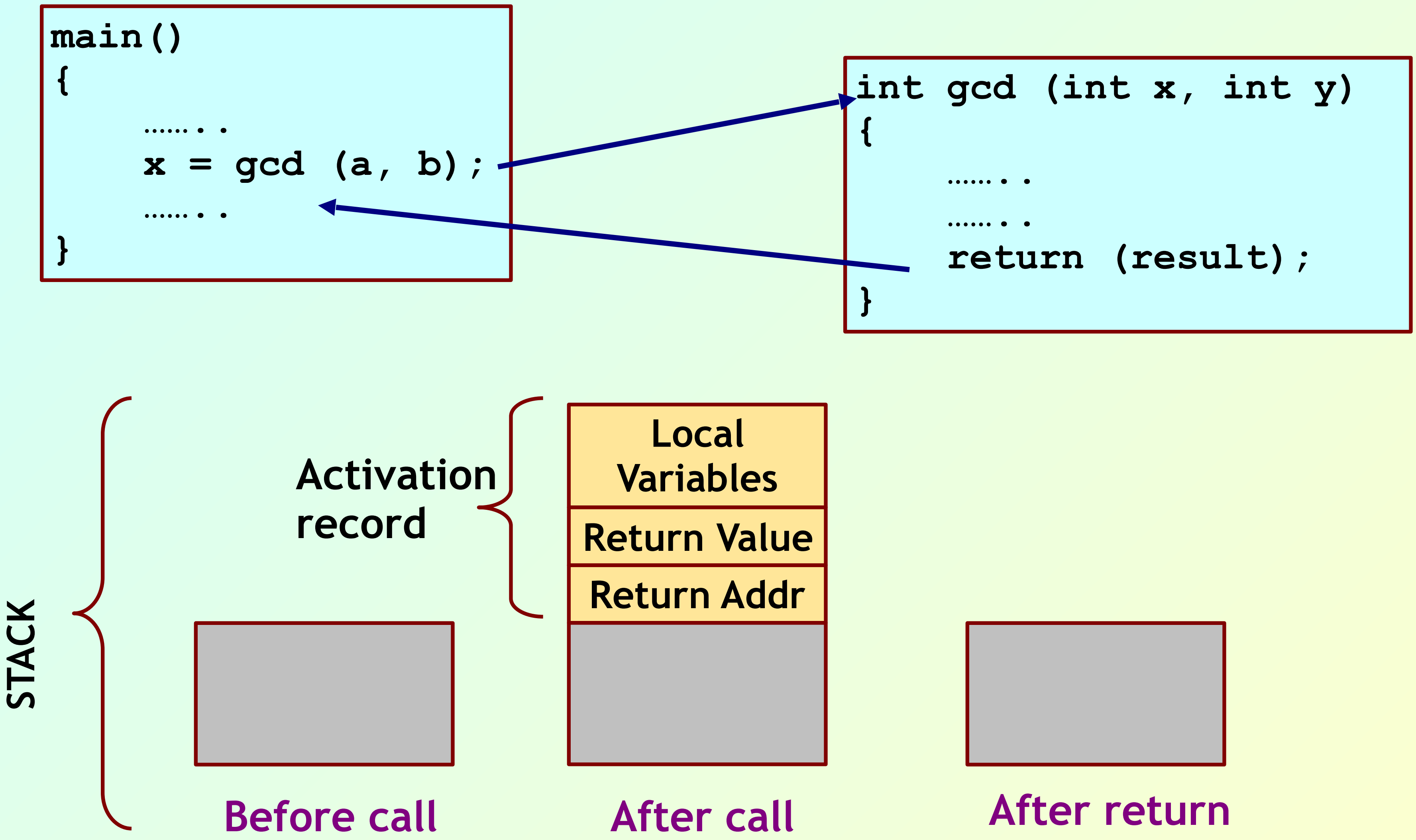
```
Move disk 5 from L to R
Move disk 1 from C to L
Move disk 2 from C to R
Move disk 1 from L to R
Move disk 3 from C to L
Move disk 1 from R to C
Move disk 2 from R to L
Move disk 1 from C to L
Move disk 4 from C to R
Move disk 1 from L to R
Move disk 2 from L to C
Move disk 1 from R to C
Move disk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

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

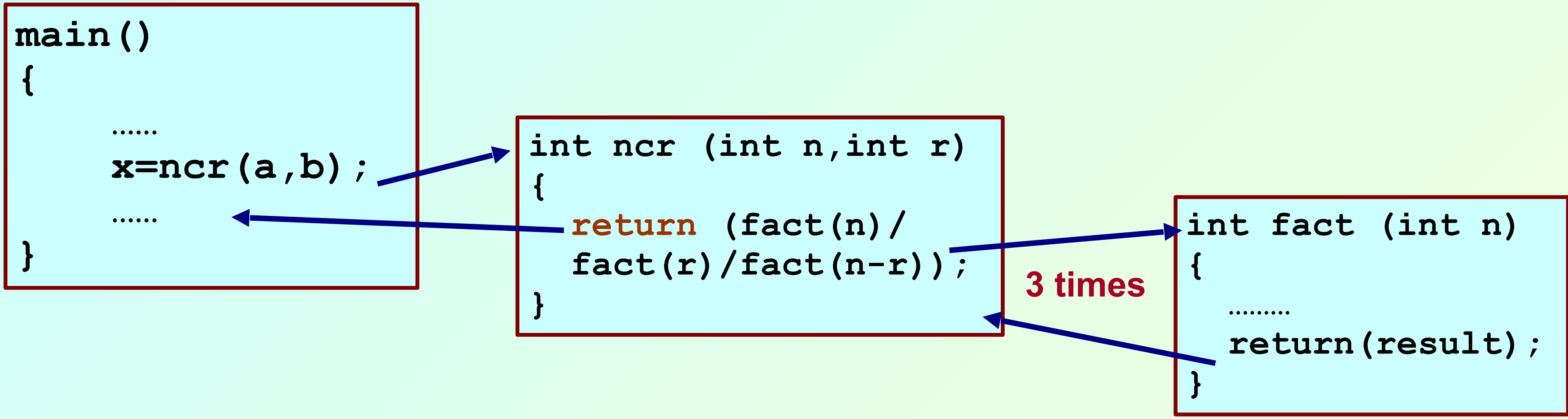
Return Value

Return Addr

Before call

After call

After return

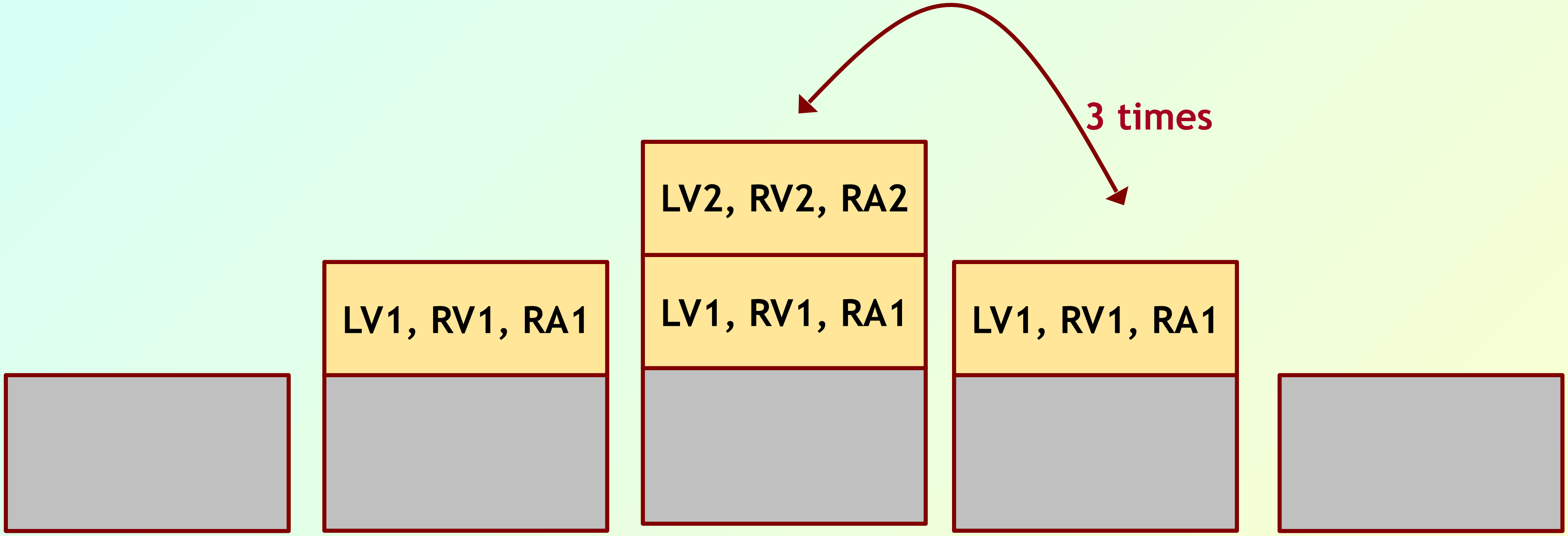


```
main()
{
      ......
      x=ncr(a,b);
      ......
}
```

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

3 times

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



3 times

LV2, RV2, RA2

LV1, RV1, RA1

Before call

Call ncr

Call fact

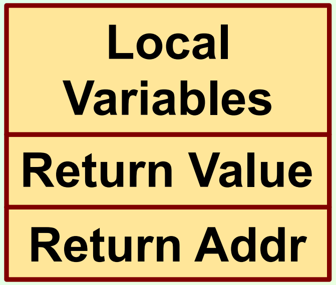
fact returns

ncr returns

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

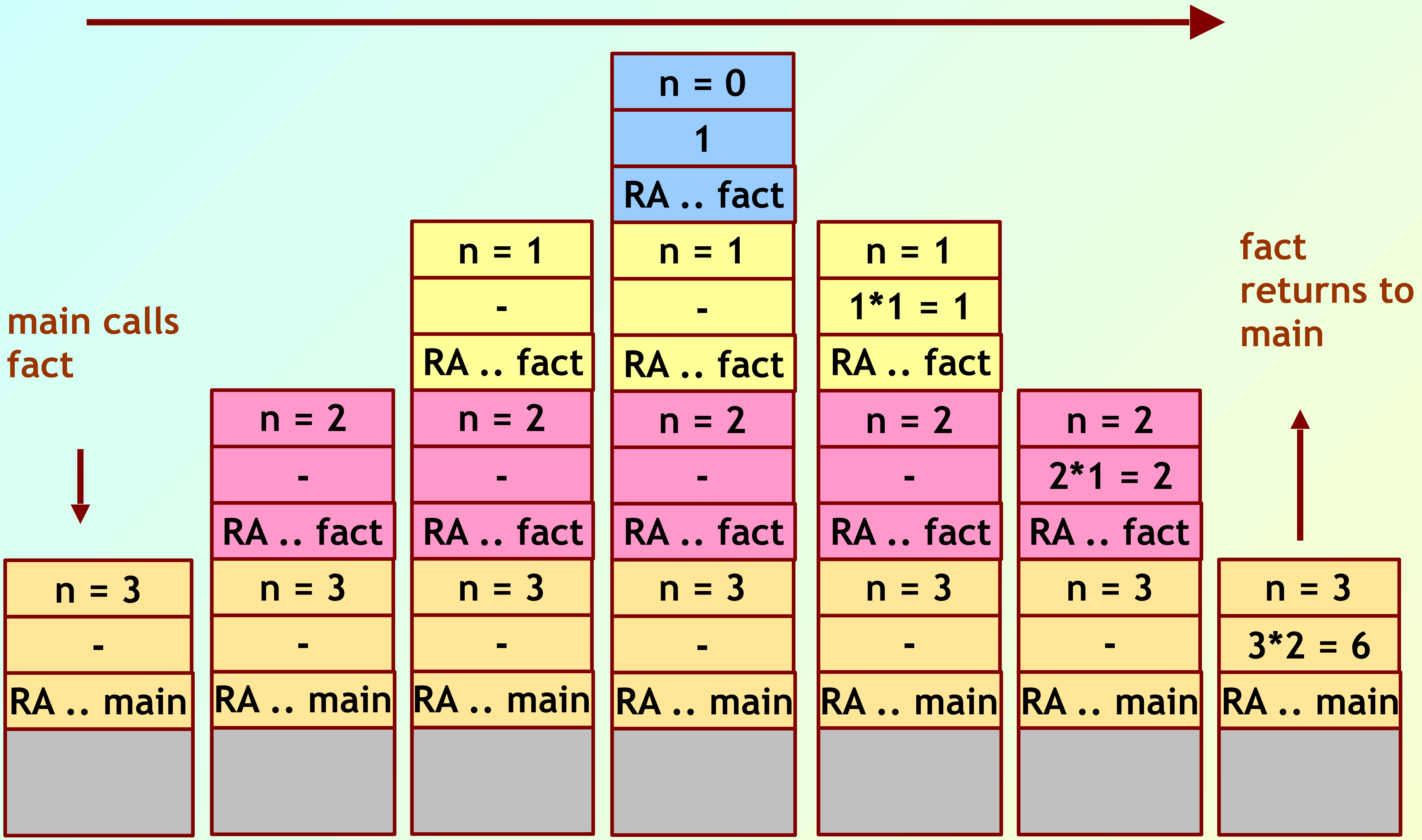
Return Addr

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



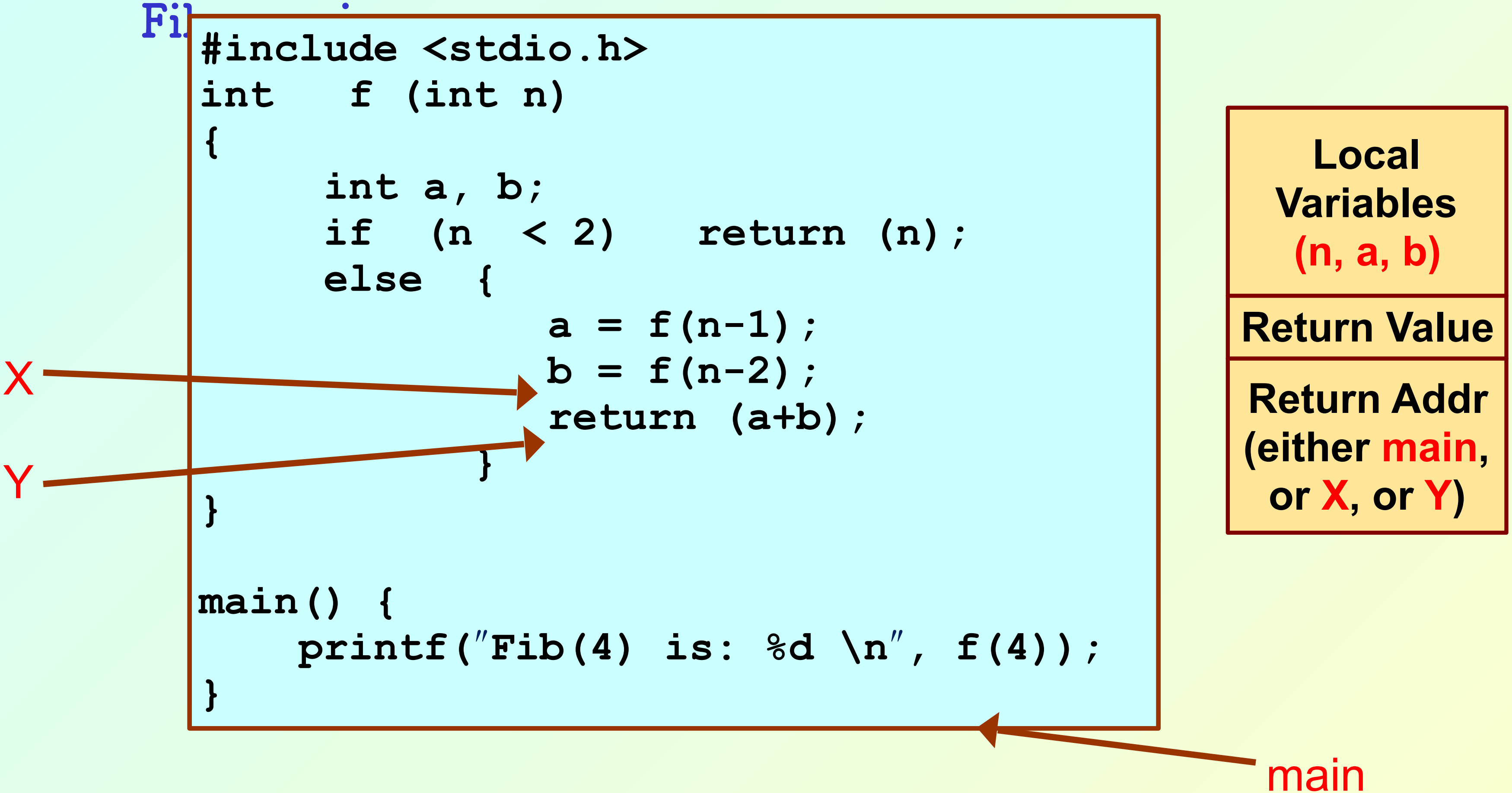
main calls fact

fact returns to main

			n = 0			
			1			
			RA .. fact			
		n = 1	n = 1	n = 1		
		-	-	1*1 = 1		
		RA .. fact	RA .. fact	RA .. fact		
	n = 2	n = 2	n = 2	n = 2	n = 2	
	-	-	-	-	2*1 = 2	
	RA .. fact	RA .. fact	RA .. fact	RA .. fact	RA .. fact	
n = 3	n = 3	n = 3	n = 3	n = 3	n = 3	n = 3
-	-	-	-	-	-	3*2 = 6
RA .. main	RA .. main	RA .. main	RA .. main	RA .. main	RA .. main	RA .. main

Do Yourself

- Trace the activation records for the following version of Fi



```
#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));
}
```

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
```

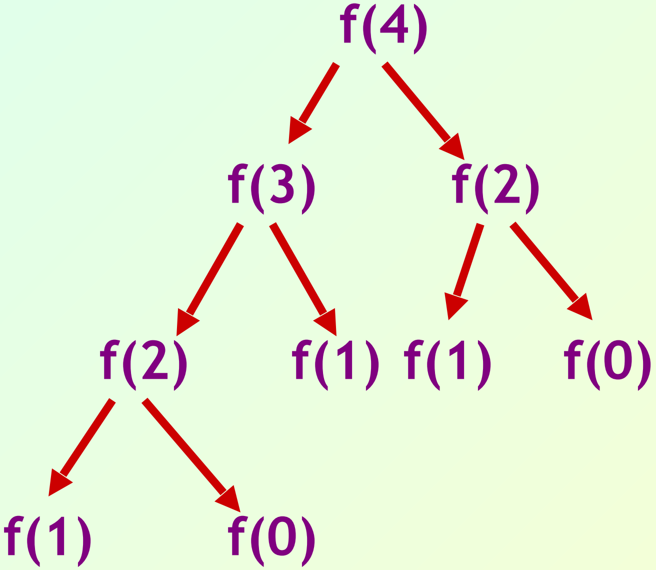
EXAMPLE 2

```
#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));
}
```

- 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
```



f(4)

f(3) f(2)

f(2) f(1) f(1) f(0)

f(1) f(0)

[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);
}
```

Some Examples on Recursion

GCD Computation … Correct Version

```
#include <stdio.h>
int  gcd (m, n)
int  m, n;
{
    if (m == 0) return n;
    if (n == 0) return m;
    if (m == n) return (m);
    if (m > n)
       return gcd (m%n, n);
    else
       return gcd (m, n%m);
}

int main()
{
  int num1, num2;
  scanf ("%d %d", &num1, &num2);
  printf ("\nGCD of %d and %d is %d", num1, num2, gcd(num1,num2));
}
```

```
GCD of 12 and 12 is 12
GCD of 15 and 0 is 15
GCD of 0 and 25 is 25
GCD of 156 and 66 is 6
GCD of 75 and 925 is 25
```

Compute power a^b

```
// Compute a to the power b
#include <stdio.h>

long int power (int a, int b)
{
  if (b == 0) return (1);
  else return (a * power(a,b-1));
}

int main()
{
  int x, y;
  long int result;
  scanf ("%d %d", &x, &y);
  result = power (x, y);
  printf ("\n%d to the power %d is %ld", x, y, result);
}
```

```
3 to the 4 is 81
2 to the power 16 is 65536
2 to the power 8 is 256
17 to the power 4 is 83521
436 to the power 0 is 1
```

Sum of digits of a number

```
// Find sum of the digits of a number
#include <stdio.h>

int digitsum (int num)
{
  int digit;
  if (num == 0) return (0);
  else {
         digit = num % 10;
         return (digit + digitsum(num/10));
       }
}

int main()
{
  int a;
  scanf ("%d", &a);
  printf ("\nSum of digits of %d is %d", a, digitsum(a));
}
```

```
Sum of digits of 25 is 7
Sum of digits of 23863 is 22
Sum of digits of 11111 is 5
Sum of digits of 0 is 0
Sum of digits of 9999 is 36
```

Decimal to Binary

```c
// Print a decimal number in binary
#include <stdio.h>

void dec2bin (int n)
{
  if (n == 0) return;
  else {
         dec2bin (n/2);
         printf ("%2d", n%2);
       }
}

int main()
{
  int dec;
  scanf ("%d", &dec);
  printf ("\nBinary of %d is", dec);
  dec2bin (dec);
}
```

```
Binary of 25 is 1 1 0 0 1
Binary of 12 is 1 1 0 0
Binary of 128 is 1 0 0 0 0 0 0 0
Binary of 254 is 1 1 1 1 1 1 1 0
```