#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)

- **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)

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

24

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

6

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

2

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

1

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

Example 2 :: Fibonacci number

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

  f(0) = 0

  f(1) = 1

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

  - The successive Fibonacci numbers are:

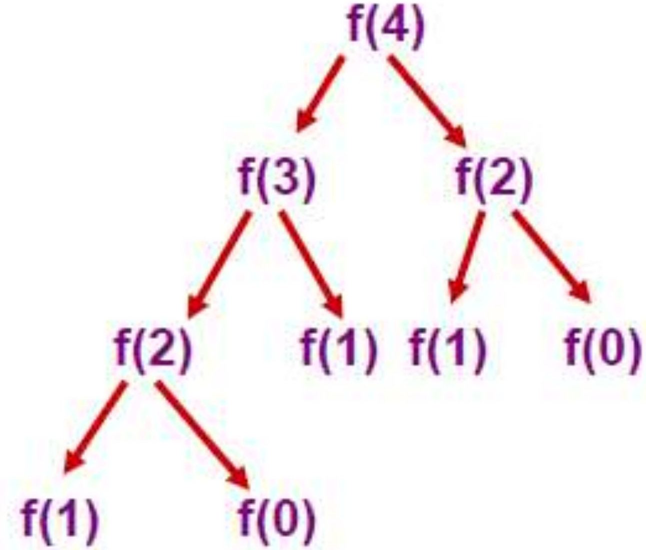
    0, 1, 1, 2, 3, 5, 8, 13, 21, …..

- Function definition:

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

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

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).

```c
#include<stdio.h>
int fib(int n)
{
          int a = 0, b = 1, c, i;
          if( n == 0)    return a;
         if(n == 1)       return b;
         for (i = 2; i <= n; i++)
         {
                        c = a + b;
                        a = b;
                        b = c;
         }
         return b;
}

int main ()
{
         int n = 9;
         printf("%d", fib(n));
         return 0;
}
```

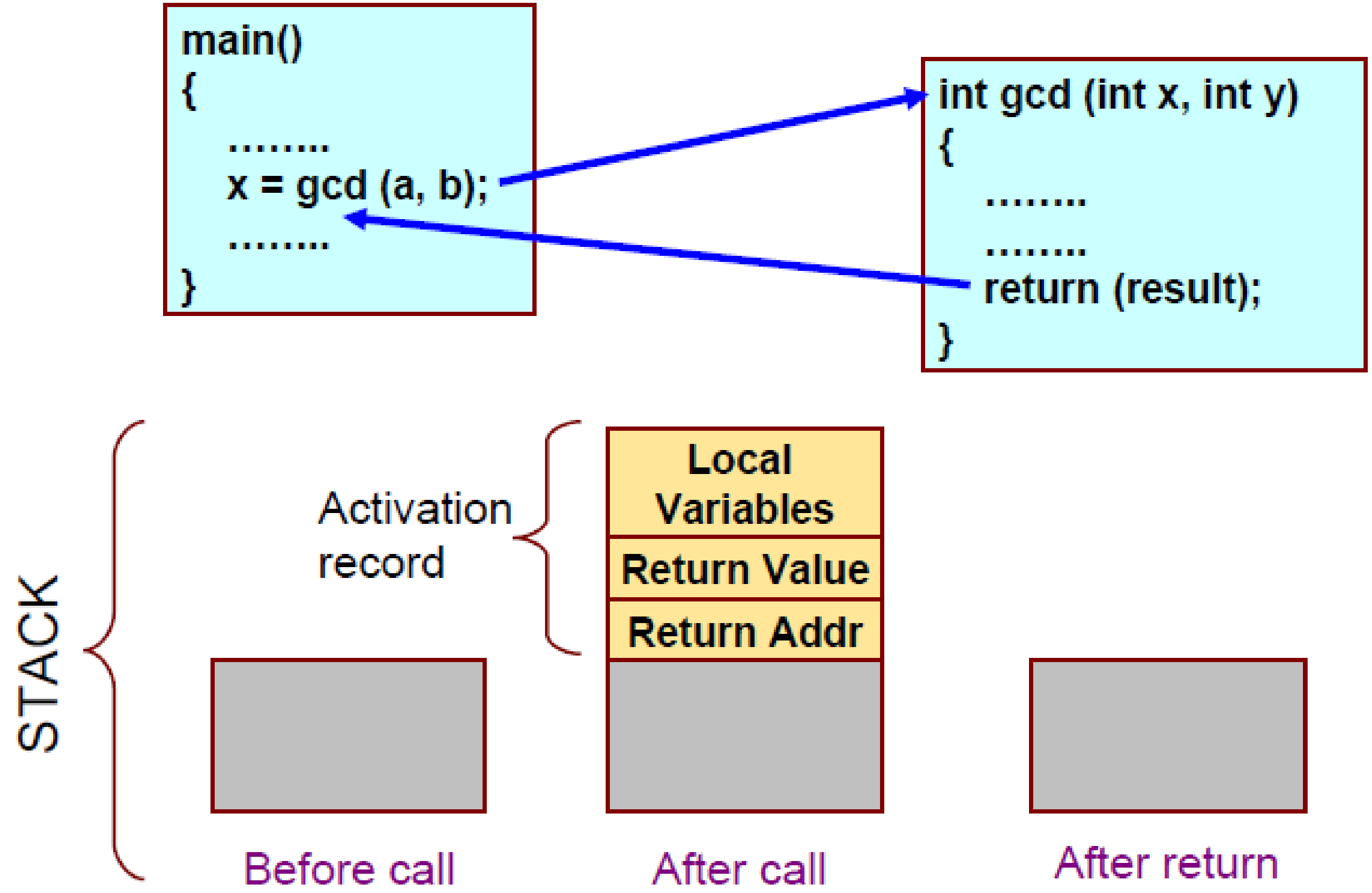
- Is this one efficient?

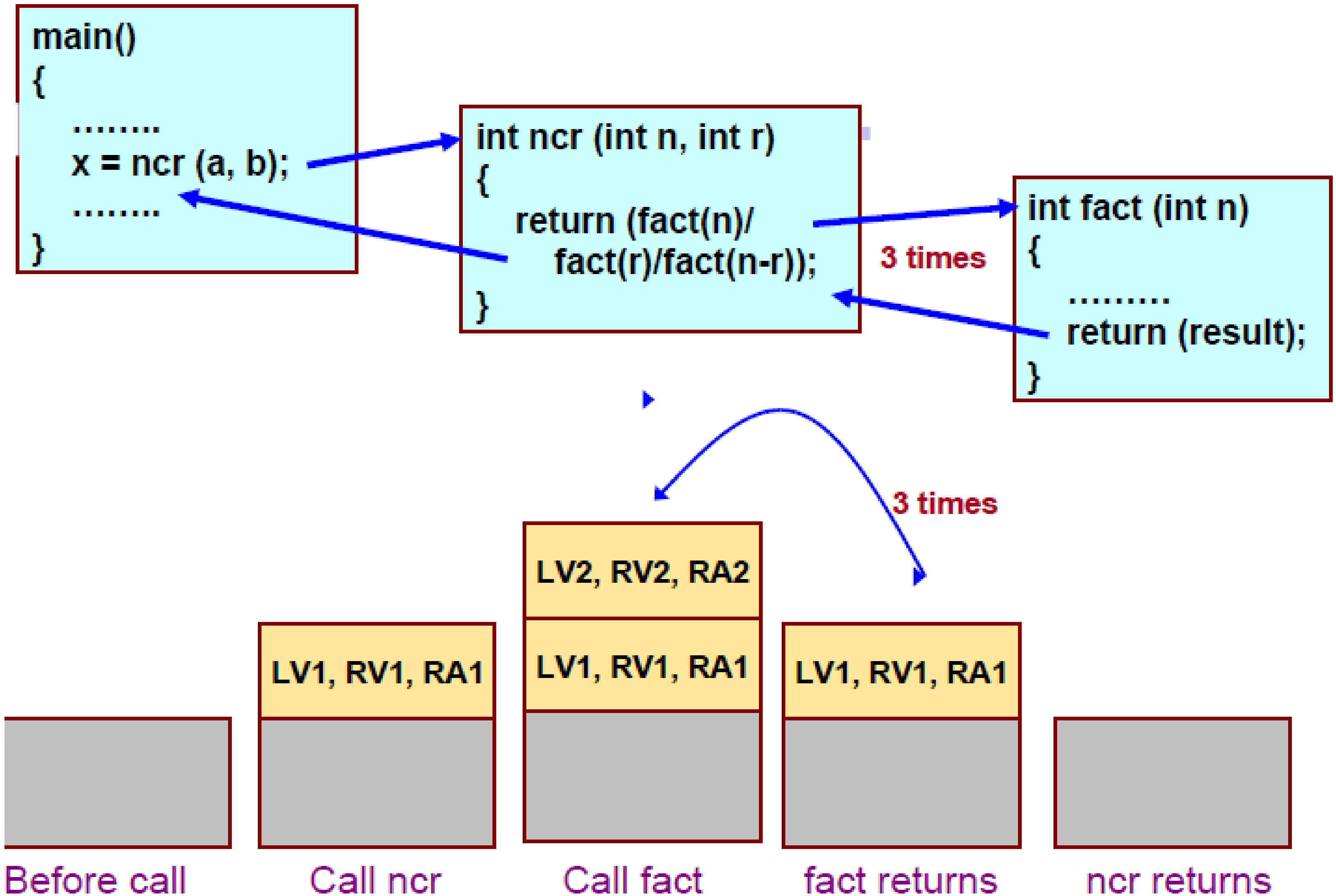
How are function calls implemented?

- 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
LV1, RV1, RA1
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.

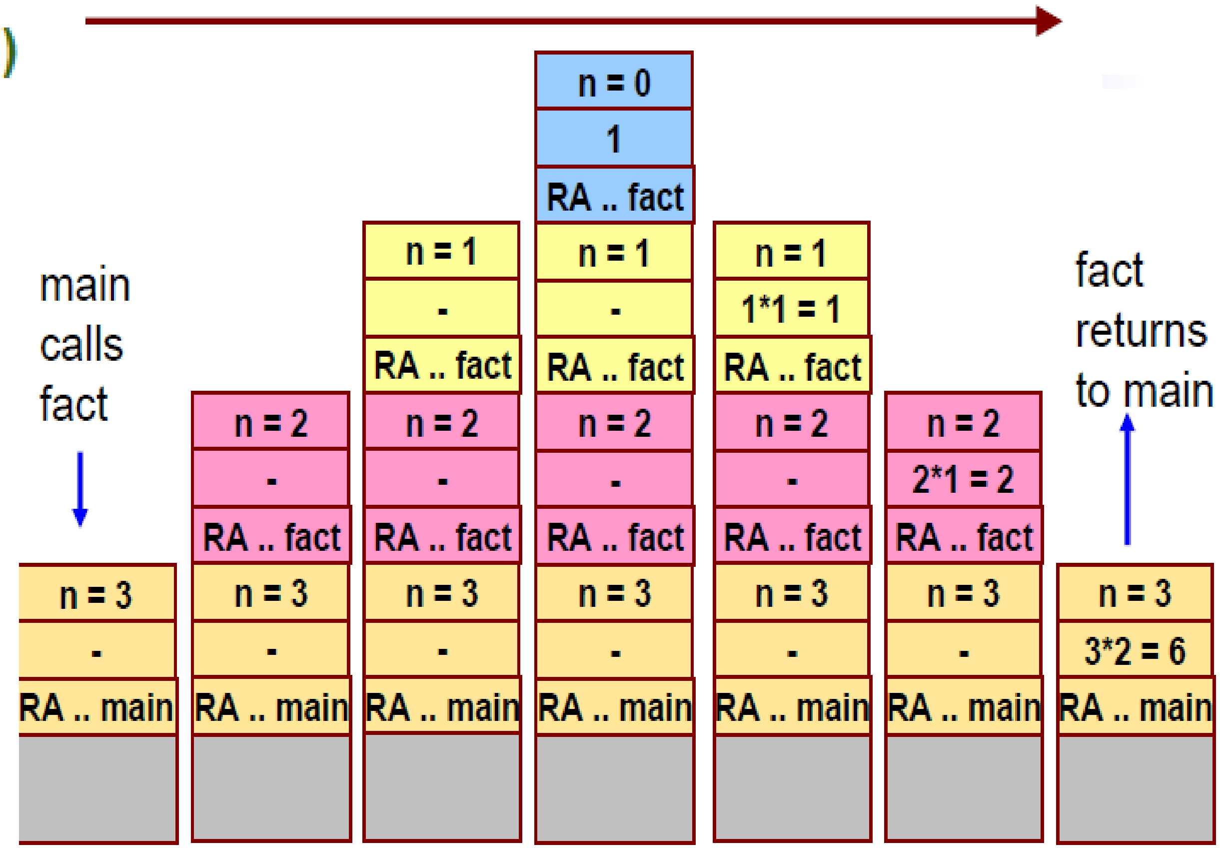
- Activation records keep popping off, when the termination condition of recursion is reached.

- We shall illustrate the process by an example of computing factorial.
  - Activation record looks like:

Local Variables
Return Value
Return Addr

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

TRACE OF THE STACK DURING EXECUTION




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

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);
X         b = f(n-2);
Y         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)

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
```

```c
#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));
}
```

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