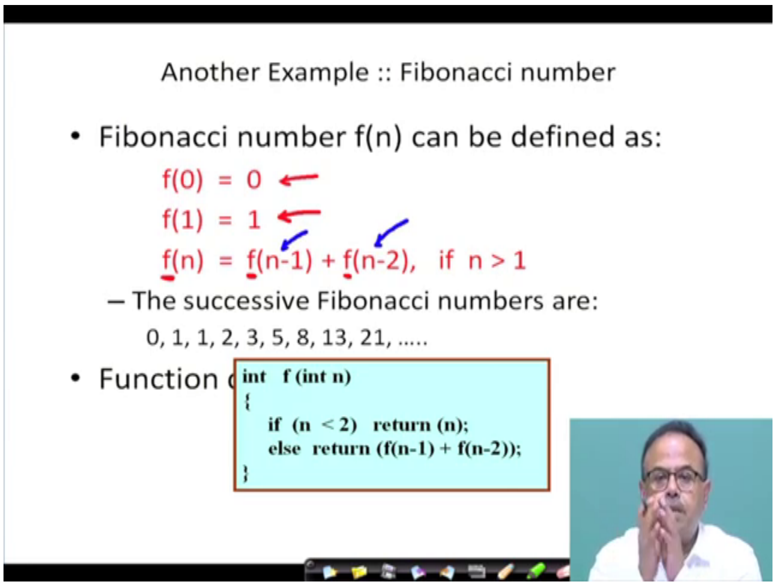
**Problem Solving Through Programming In C**
**Prof. Anupam Basu**
**Department of Computer Science and Engineering**
**Indian Institute of Technology, Kharagpur**

# Lecture - 54
# Recursion (Contd.)

So, we are looking at recursion and that is a new style of programming where we can express the particular function in terms of itself like I can express factorial n in terms of factorial itself factorial n minus 1 and n into factorial n minus 1 ok. Another very common example of a and easy example of recursion is Fibonacci numbers, we have already told you; what Fibonacci sequence is the Fibonacci sequence can be expressed as f 0 Fibonacci 0th Fibonacci number is 0. The next one is also 1.

(Refer Slide Time: 00:57)



Another Example :: Fibonacci number

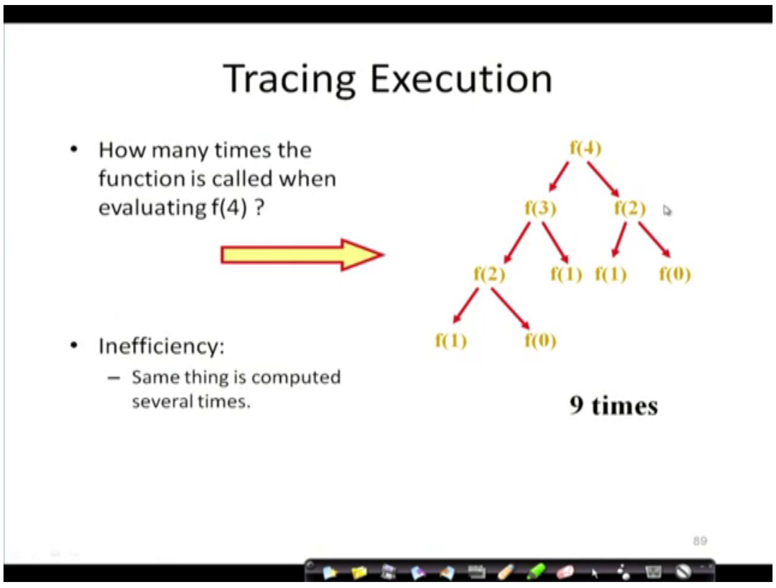
- Fibonacci number f(n) can be defined as:
  - $f(0) = 0$
  - $f(1) = 1$
  - $f(n) = f(n-1) + f(n-2)$, if $n > 1$
  - The successive Fibonacci numbers are:
    0, 1, 1, 2, 3, 5, 8, 13, 21, .....
- Function

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

And henceforth, all other ones are sum of the previous 1; so, 0, 1, 1, 2, 3, 5, 8, 13, 21, so and so forth. Therefore, we should be able to define it in terms of a recursive function because you can see this function f and this function f are the same only variations are in these parameters, right, I am expressing the same function in terms of these parameters. So, the function definition will be is very simple f; intend some integer if n is less than 2, then return n if n is less than 2 if it is 0 then 0.

If it is 1, then it is 1, otherwise what did you return; return f n minus 1 plus fn minus 2 sum of the previous 2 Fibonacci numbers. Now this is interesting because again if you see how this will be computed, it will be first expanding what are the things I have to compute and when it meets stopping condition then it starts collecting back and come back.

(Refer Slide Time: 02:31)



## Tracing Execution

- How many times the function is called when evaluating f(4) ?
- Inefficiency:
  - Same thing is computed several times.

9 times

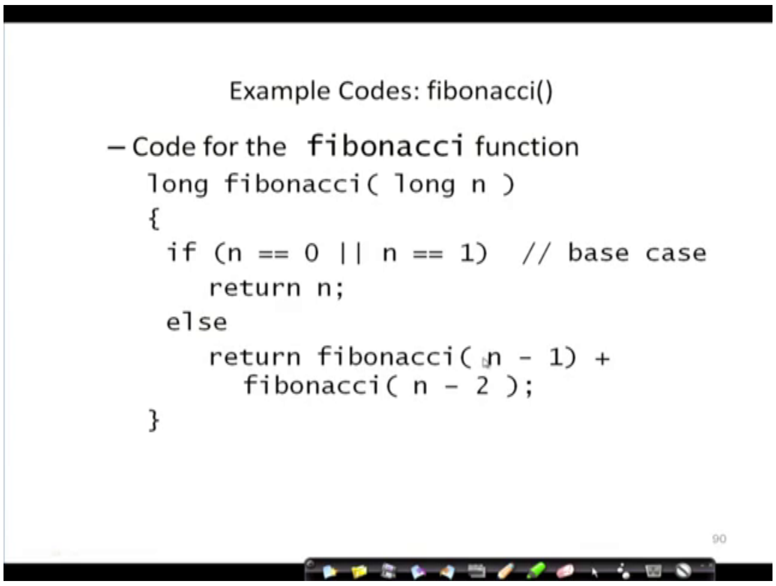
So, how many times say if I say if I want to compute Fibonacci of 4 how many times will that function be called let us look at the expansion of this. So, how will it happen Fibonacci of 4 will be I want to compute Fibonacci of 4 is Fibonacci of 3 Fibonacci of 2, these 2 should be added. So, I have not yet have found out anything, I am just decomposing the problem ok.

There is a very very important concept that in order to solve the problem, I want to decompose it into smaller sub problems for f 4, I have to solve it by solving f 3 and f 2 ok. Now for solving f 3, I have to solve f 2 and f 1, I further decomposed it. Now for solving f 2 I have to solve f 1 and f 0 fortunately and for itself now the even now the entire thing has not been broken down f 2, for that I have to solve f 1 and f 0. Now I have expanded the whole thing.

Now, I know that f 1 is 1, f 0 is 1. So, f 2 will be 1 plus 1; 2, this is known all these endpoints of this structure are known 1 0, 1, 1, 0, 1. So, I go on adding them and ultimately, I will get f 4. Now it is in a way inefficient, but because the same thing is being computed repeatedly ok, but it has it will make for to annex to a practiced programmer, it will make your programming writing the program much more I mean less line lines of codes if you can express it in much more better ways.

So, you can see here; how many times the function was called 1, 2, 3, 4, 5, 6, 7, 8, 9 times; 9 times the same function was called in order to compute f 4.

(Refer Slide Time: 05:02)



Example Codes: fibonacci()

– Code for the `fibonacci` function

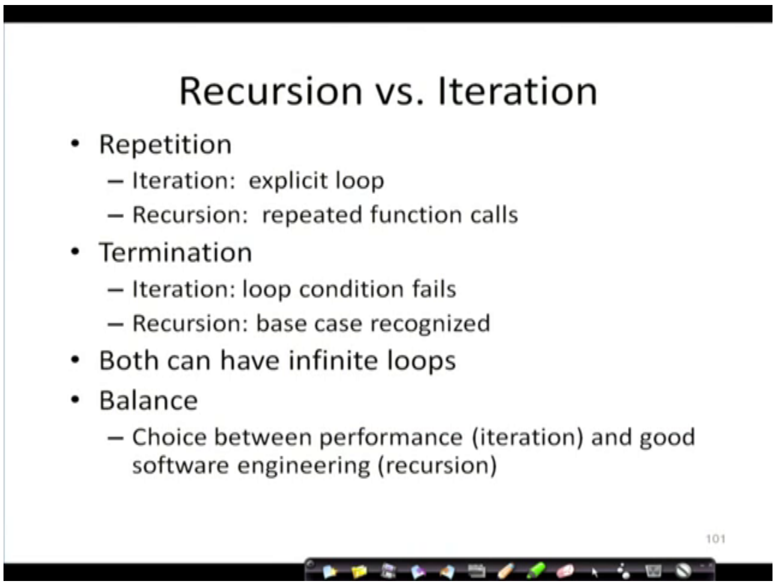
```
long fibonacci( long n )
{
  if (n == 0 || n == 1)  // base case
     return n;
  else
     return fibonacci( n - 1) +
       fibonacci( n - 2 );
}
```

90

So, the code for the Fibonacci sequence will be; if now the stopping condition of the base condition is very very important is very very important. So, if n is 0 or n is 1, then I return 1, this is the base case, unless I reach at this point, I will not be able to compute the entire solution.

Otherwise, return Fibonacci of n minus 1 plus Fibonacci of n minus 2 that is the code for the Fibonacci number. Now I mean this sort of whenever I will have too many calls in that case, I should be should avoid them as much as possible and. So, what is the difference?

(Refer Slide Time: 06:00)



## 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)

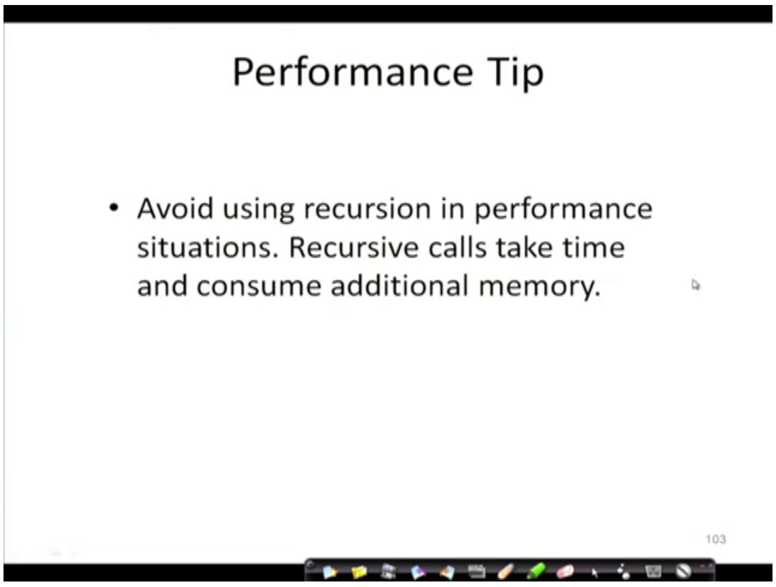
101

Between recursion and iteration in recursion, we have the repetition sorry in iteration it is both reputation in iteration there is an explicit loop explicit loop for i equals 0, i less than equal to n minus 1 i plus plus.

So, there is an explicit loop whereas, in case of recursion, it is a repeated function calls, all right, termination iteration if the loop condition is no longer satisfied while this condition do if that condition fails, then we come out of the loop. In the case of recursion, the base condition must be recognized whenever we are getting the base condition factorial one or factorial 0 Fibonacci of 0 or Fibonacci of 1.

So, those are the base conditions, both can have if wrongly program both can have infinite loop. So, the performance wise iteration often gives faster result, but it is a good software engineering practice to gradually get accustomed to recursion as you do more and more programming, you will see that you will be able to express the things in a much settler way.

(Refer Slide Time: 07:29)

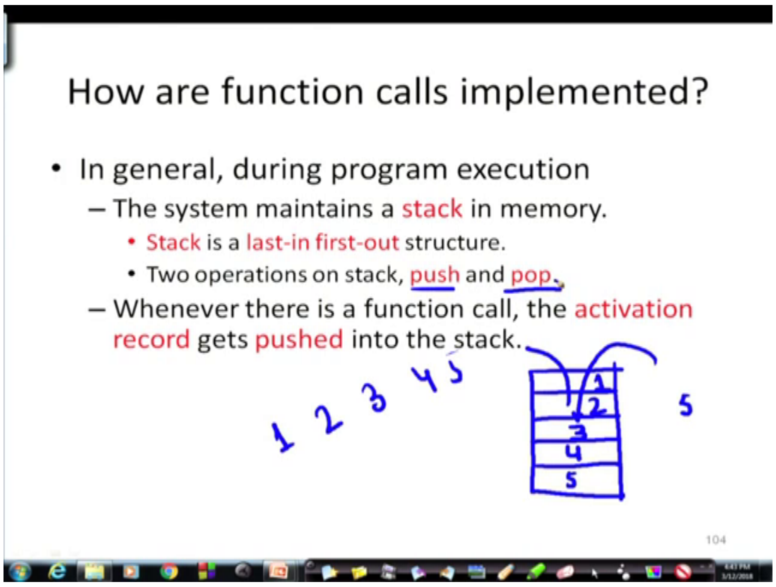


## Performance Tip

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

103

(Refer Slide Time: 07:37)



## 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.

104

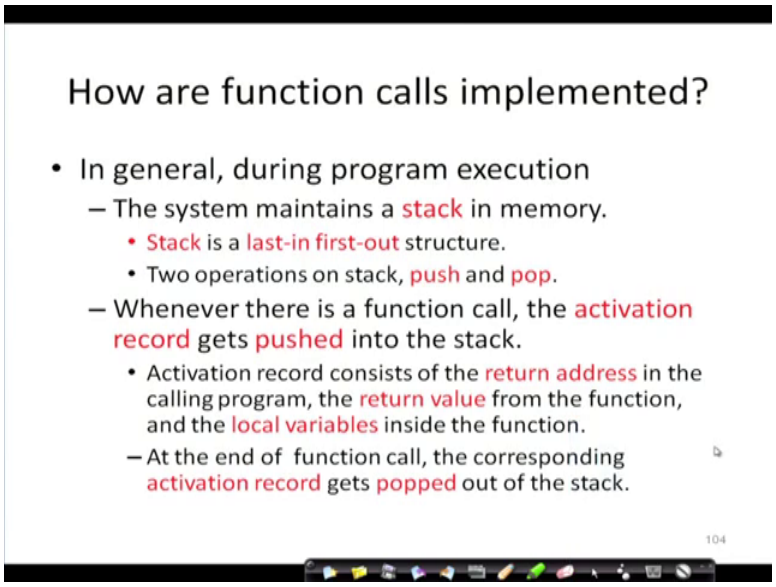
So, whenever there is a performance issue try to avoid recursion, it will require additional memory also there is a there is a particular type of storage that is required in recursion that is known as stack. Stack is a last in first out type of structure.

So, those things, briefly, let me tell you; how this thing is done because stack is nothing, but a structure where we can push in data from one side say I put say 5, first I push. So, 5 comes here, then I push 4, then I do n minus 1 3 is pushed, then 2 is pushed then n minus 1 again and 1 is pushed. Now when I take out the data, the data will come out as 1, then

2, then 3, then 4, then 5, in the reverse order ok. So, the 2 operations are push and popped ok.

Popping out from the stack and pushing inside the stack.

(Refer Slide Time: 08:52)



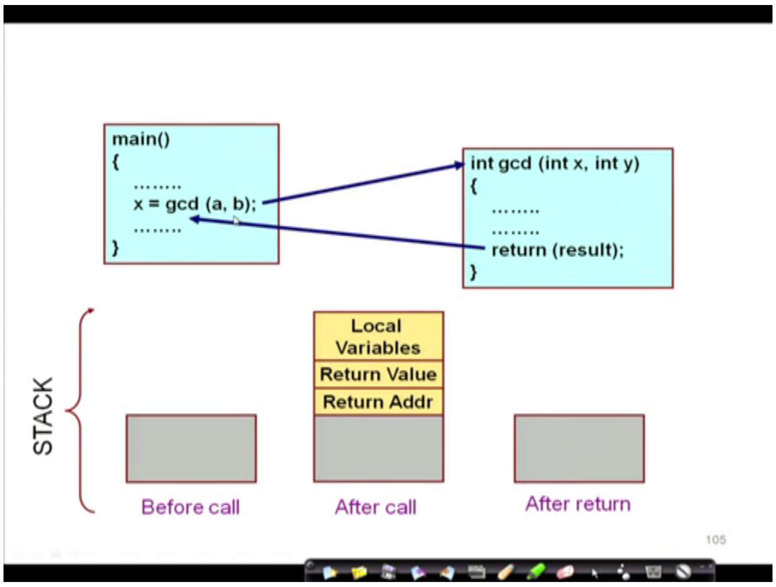
## How are function calls implemented?

- 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.

104

So, this stack data structure becomes very handy for implementation of function recursive functions, we will show some examples.

(Refer Slide Time: 08:59)



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

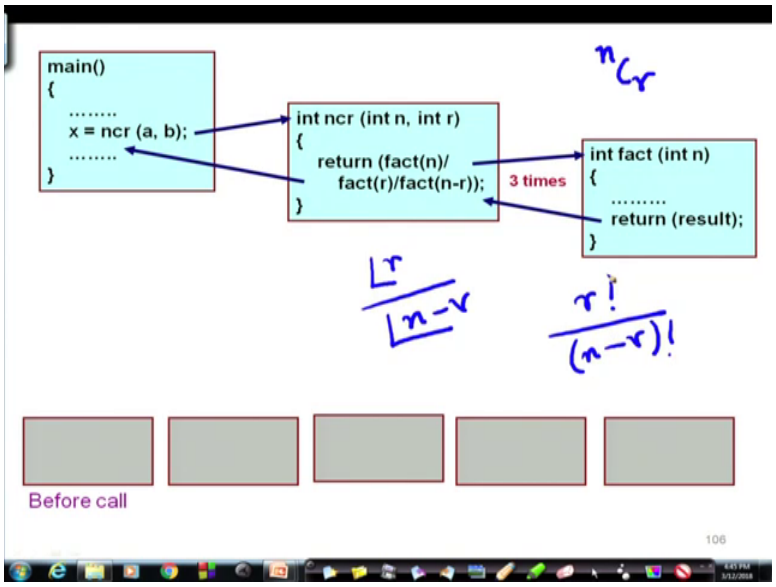
```
int gcd (int x, int y)
{
   ........
   ........
   return (result);
}
```

Like here for example, I want to compute the gcd of a b. Now typically what happens is I call this city is computed I return ok.

Now, here when I call something I call something all the say; here was I; here I was here my program flow was here I went in. So and when I went in all the local variables and everything here was stored and I had to remember where I will be returning back. So, all those things are stored in the stack and without a stack data structure, it is very difficult to implement recursion and for that matter any function call.

So, you see here is a function and so, all those return address is stored before called this stack was empty now after when its returning it is taking out from the top of the stack and again I come to know where I was ok.

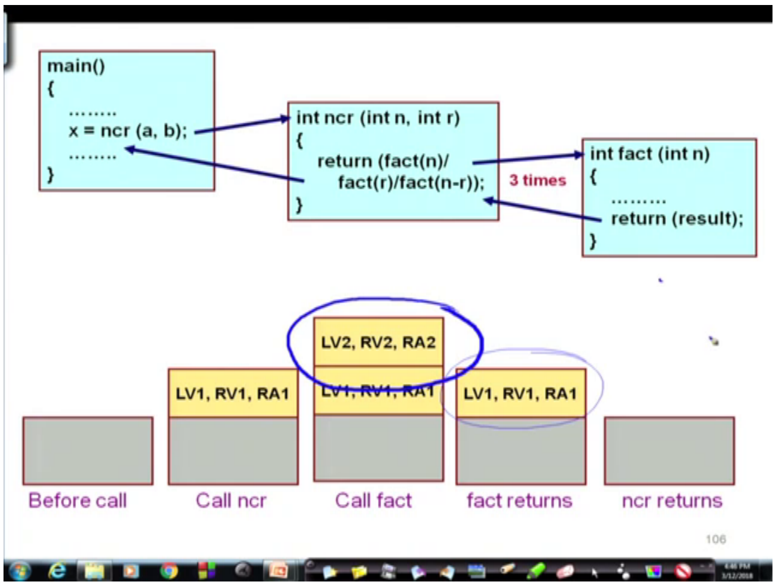
(Refer Slide Time: 10:09)



So, that is so, similarly you see here. It is a inches are a b; that means, n C r what you compute n choose r, if I compute, then you know n choose r is factorial r divided by factorial of n minus r or some people write it in this way by factorial of n minus r. So, how did I do that? So, how can I implement it?

So, here n C r has been called from here factorial has been called and then where do I return I return I have to come here ultimately I have to return here. So, I should not lose the path. So, what the stack does is when I make the call first call then when I make the call n C r.

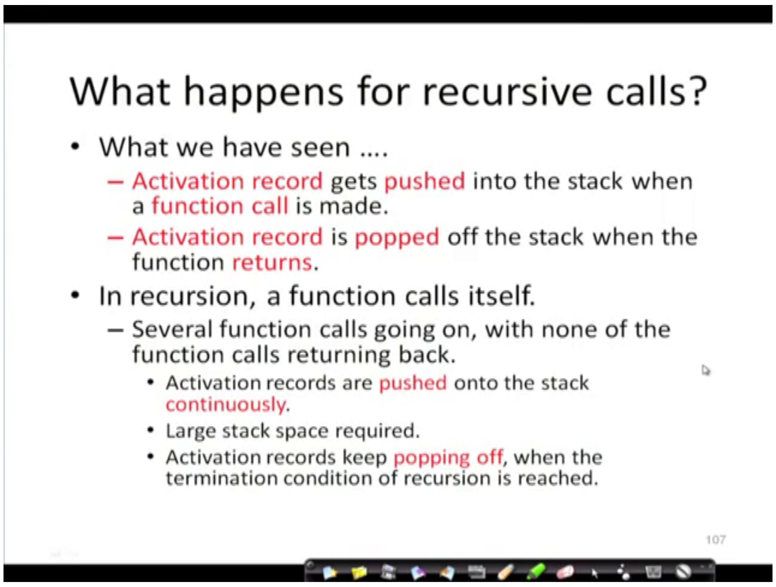
(Refer Slide Time: 11:10)



Then the local variables here will be stored on the stack and again I make another call from here. So, local variables are there, I am calling fact as I go in here, the local variables here has stacked up and then when I return this part this part this part will be taken out and passed on to this.

So, what happens is this part is as it returns this part is taken out and I am here, it can again continue and then it returns when it returns here this part will be taken out this part will be taken out .

(Refer Slide Time: 12:13)



## What happens for recursive calls?

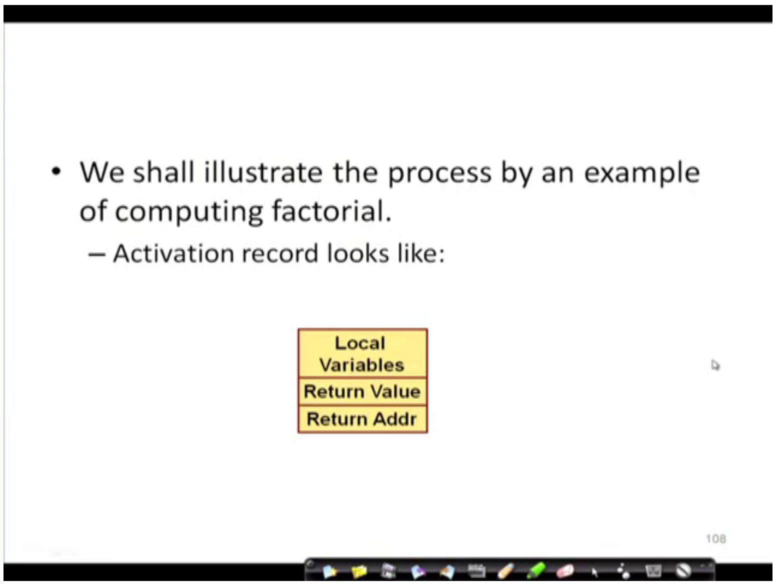
- What we have seen ....
  - Activation record gets pushed into the stack when a function call is made.
  - Activation record is popped off the stack when the function returns.
- In recursion, a function calls itself.
  - Several function calls going on, with none of the function calls returning back.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

107

This part will be taken out and. So, that is for normal function called how the stack remembers where I should go back all right in the case of recursive calls what happens what we have seen is activation record gets pushed into the stack when a function I will call is made in recursion a function calls itself.

So, several function calls are going on with none of the calls are getting back. So, all the activation records are collected. So, you need not delve into that too much let us see I will.

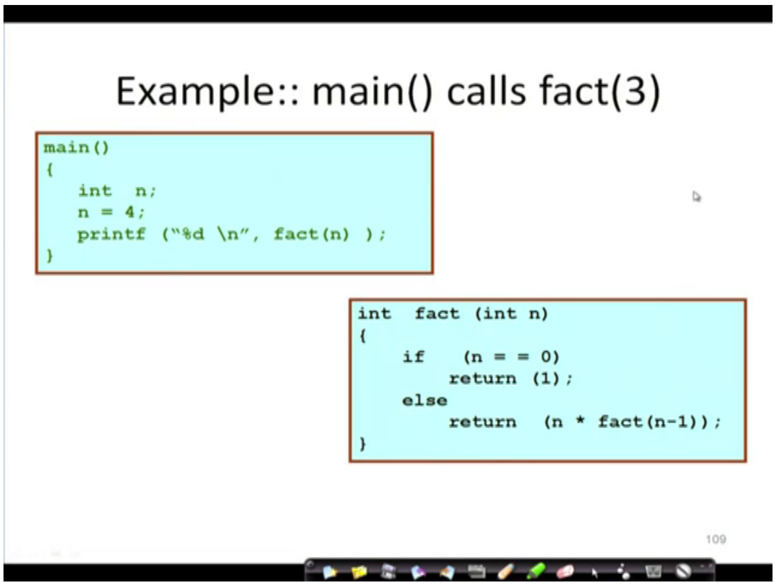
(Refer Slide Time: 12:54)



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

Show it by an example of computing factorial. So, an activation record is the local variables and the return value what the function should return and where it should return.

(Refer Slide Time: 13:11)



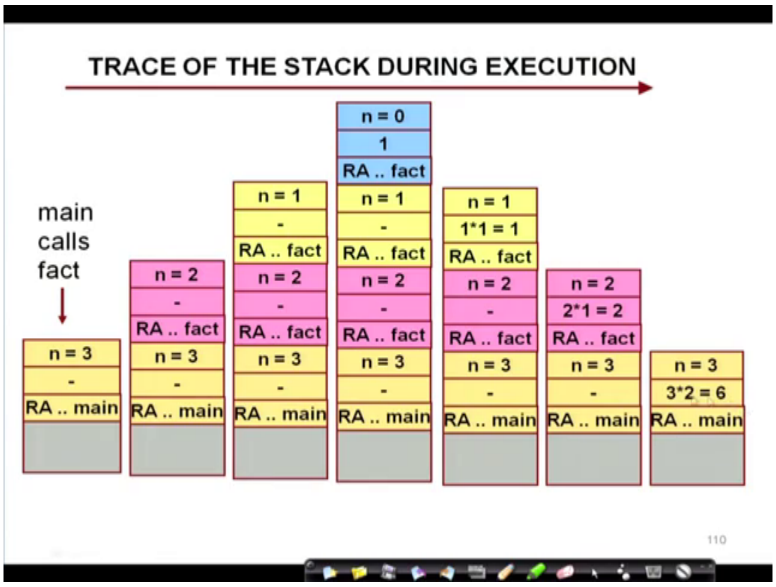
## Example:: main() calls fact(3)

```
main()
{
   int  n;
   n = 4;
   printf ("%d \n", fact(n) );
}
```

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

So, that with that say the main function is calling fact 3, I will track them and here is a fact n if n equal to 0 return one otherwise n times fact n minus 1.

(Refer Slide Time: 13:28)



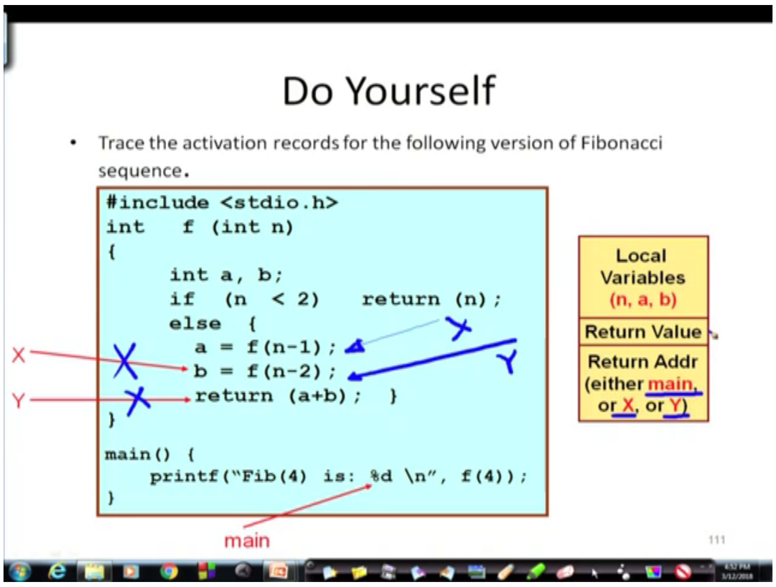
## TRACE OF THE STACK DURING EXECUTION

So, main calls fact. So, when it calls the value is n equal to 3 there is no return value return address is in main I am remembering that next again fact is calling itself; so, now, fact is calling fact 3 is calling fact 2 and my return address is fact. So, you see it has been stacked up next fact 2 will call fact one so that is what the stack is growing and if its return is in fact, all right.

Now, next time, it will be fact 0. Now the returned value till now, there was no return value. Now the return value is 1 and return addresses fact. So, as I do that I return, then I have got a return value because now I have come to this point. So, 1 into 1 will be 1 and I am returning to fact, as I return the stack will contract and what is the return result? 2 times 1 that is 2 that is a factorial 2 and return I am returning to fact. So, I return again, now I am coming to the last time in the fact with n equal to 3 that started here.

So, and the result is 6. Now I returned to main. So, at every stage look at this, I have I know I remember from where I started and from where I am returning back nothing is lost using this stack. So, stack is a very interesting data structure, that helps us in many ways, especially in implementing things like recursion and although.

(Refer Slide Time: 15:26)



## 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));
}
```

| Local Variables (n, a, b) |
|---|
| Return Value |
| Return Addr (either main, or X, or Y) |

So, one assignment that I am leaving to you do it yourself, trace the activation record for the following version of Fibonacci. Please note down the code include stdio dot h in def f is the Fibonacci function a and b if n is less than 2 return n if it is 0 then return 0.

If it is 1 return 1, otherwise a is fn minus 1, b is fn minus 2, I have done it in a different way, fn minus 1 and fn minus 2. So, if n minus 1 has to be solved separately and fn minus 2 should be solved separately, then we will return a plus b all right and then the main will print. So, just as a fun you try to draw the activation record of this version of the function, please note it down, take some time and note it note down this function and you see on this side I have shown.

How the activation record will look like local variables, you can see n a and b return value you have to keep whether it is in Fibonacci or in main either in main or in x or y, what is x and what is y this is x and this sorry this is the there is an problem in this I am drawing it again. So, you see this; this is x, this is x and this is y, alright not these 2 these are not these are not aligned properly, all right.

So, either where do I return and here is the return to the main either a return to main or to x or to y and what is the return value draw the activation record of this and then, we will see how much you could do it I am sure you will be able to do it and because. So, today we have learnt a new style of programming that is recursion and also we discussed in the last class. So, recursion is a type of writing functions where the function calls itself and that makes many functions to be written much more. Secondly, much more subtly and that is a very good software engineering practice although as a beginner.

If you find difficulty in that; you need not bother too much about it, you have got iteration at your disposal and you can solve most of the problems with iteration practically all the problems, you can do may be in some cases, it may be a little more difficult to write, but ultimately it is will be possible ok. So, if you find difficulty with recursion, you can set it aside for the time being, but we have to discuss it because that is a very nice way of writing functions, we will continue with the concept of structures in the next lecture a new thing will be introduced that is called structure ok.

Thank you.