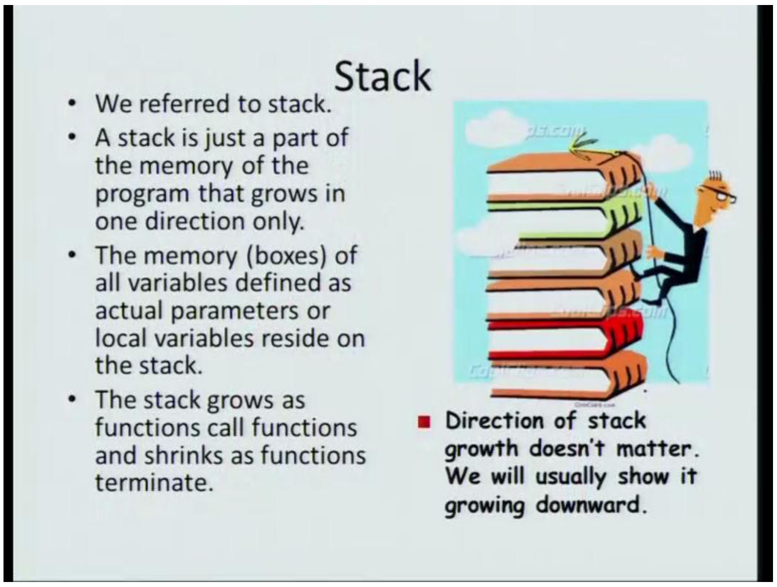
Introduction to Programming in C

Prof. Satyadev Nandakumar
Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur

Lecture - 26

When comes to C functions, we have seen the following concepts. One is the declaration in the definition of the function by which I mean the declaration is what type is the function? What are the input arguments? What types are the input arguments? And what is the result return type? So, these form the declaration. Definition is the logic of a function. So, this is what is known as the declaration and the definition of the function and we do it only once. So, function is defined only once. Once we define a function we can of course, call the function multiple times. So, definition is done only once and calling can be done any number of times.

(Refer Slide Time: 00:49)



Stack

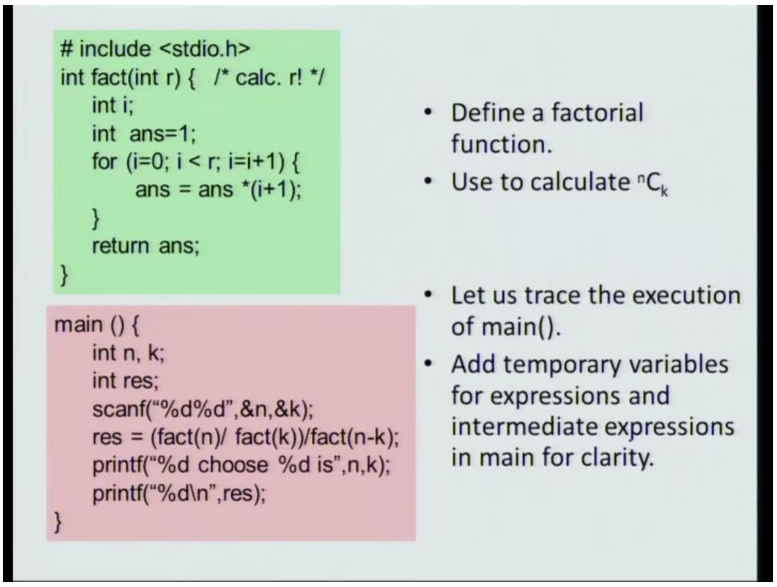
- We referred to stack.
- A stack is just a part of the memory of the program that grows in one direction only.
- The memory (boxes) of all variables defined as actual parameters or local variables reside on the stack.
- The stack grows as functions call functions and shrinks as functions terminate.

- Direction of stack growth doesn't matter. We will usually show it growing downward.

Now, we refer to a stack which is, what is the central concept in executing a function. Stack is just a part of the memory, that goes only in one direction. So, that is what it is supposed to mean. Basically, you can think of it as a stack of boxes or a stack of paper on a table or a stack of a plates. So, it grows in one direction. So, the stack grows as the main calls of a particular function, that function calls a different function and so, on.

And you can imagine the stack is growing upwards or growing downwards. It does not matter. As functions get called it either grow keeps growing upwards or keeps going downwards. We will usually represent it us keeping growing downward.

(Refer Slide Time: 01:44)



```
# include <stdio.h>
int fact(int r) {   /* calc. r! */
     int i;
     int  ans=1;
     for (i=0; i < r; i=i+1) {
           ans = ans *(i+1);
     }
     return ans;
}
```

```
main () {
     int n, k;
     int res;
     scanf("%d%d",&n,&k);
     res = (fact(n)/ fact(k))/fact(n-k);
     printf("%d choose %d is",n,k);
     printf("%d\n",res);
}
```

- Define a factorial function.
- Use to calculate ${}^{n}C_{k}$
- Let us trace the execution of main().
- Add temporary variables for expressions and intermediate expressions in main for clarity.

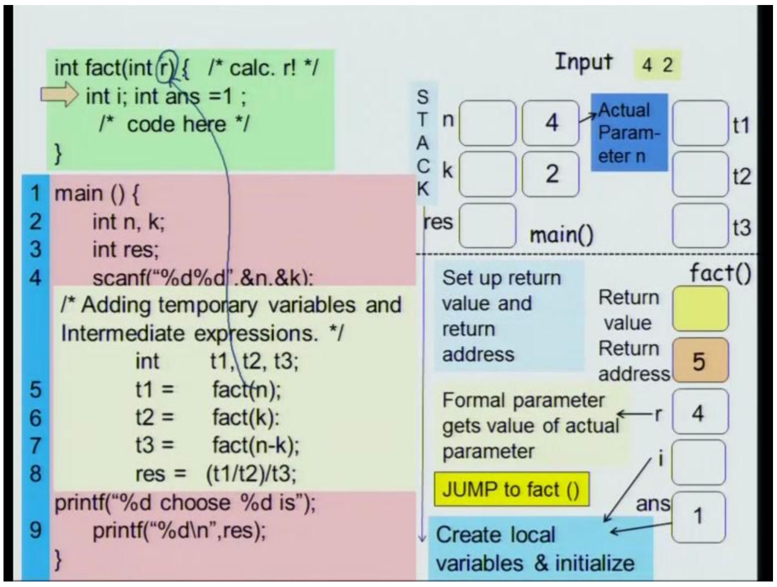
So, let us look at this function that we were talking about earlier. So, n choose k is n factorial upon k factorial times n minus k factorial and let us try to code this up. We know that factorial is something that we will need over and over in this program. So, let us say that I write factorial as a function. So, factorial takes an integer and returns an integers. So, the declaration is int fact int r, r is a input argument and the return type is int.

Now, inside that we will write the code for factorial. All variables declared inside the factorial or local or private to the factorial function, they cannot be seen outside. So, the input argument as well as any variables declared inside factorial or private or local to the factorial function. So, I have i and this encodes the logic of factorial that we have seen earlier.

So, you start with the product equal to 1 and keep on multiplying the numbers, till you reach r factorial. So, once you reach r you return the r factorial. This logic is something that we have seen before. Now, we will see how do we put this together in order to produce the function. So, what we need to do is? We will just encode this solution that we have. So, it is factorial n divided by factorial k divided by factorial n minus k. So, here are the encoded just a logic.

So, even though division is involved I know that when I do n c k the result is always going to be an integer. So, I can declare it us int res. So, this part is known as the definition of the factorial function. So, this part is what is known as definition and each of this are what are known as calls. Now, let us try to see what happens when we execute this program? So, regardless of how many functions have been defined, whenever you start executing a programming it always executes the first line of main. So, let us try to add some temporary variables. Because, we call this function three times in this main. Let us try to separate them out into three separate calls, just for the sake of clarity.

(Refer Slide Time: 04:54)



```
int fact(int r) {   /* calc. r! */
    int i; int ans =1 ;
      /*  code here */
}
1  main () {
2      int n, k;
3      int res;
4      scanf("%d%d",&n,&k);
   /* Adding temporary variables and
   Intermediate expressions. */
           int      t1, t2, t3;
5          t1 =     fact(n);
6          t2 =     fact(k);
7          t3 =     fact(n-k);
8          res =   (t1/t2)/t3;
   printf("%d choose %d is");
9      printf("%d\n",res);
   }
```

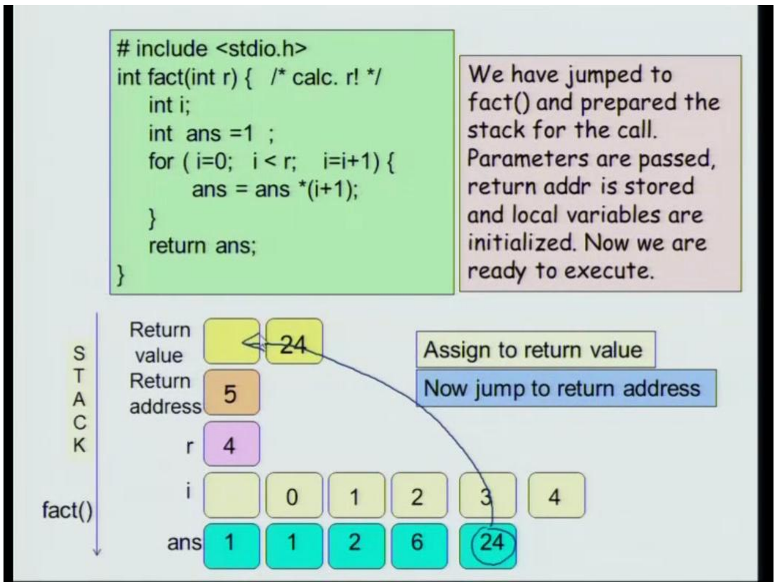
Input 4 2
STACK
n
4
k
2
res
Actual Param-eter n
t1
t2
t3
main()
fact()
Set up return value and return address
Return value
Return address
5
Formal parameter gets value of actual parameter
r
4
i
JUMP to fact ()
ans
1
Create local variables & initialize

So, I will add slightly larger code, this is not proper c code. I let us say that I have three extra variables which have declared of int t 1, t 2 and t 3. Now, t 1 will be factorial of n, t 2 will be factorial of k and t 3 will be factorial of n minus k, have separated this out. So, that, I can clearly explain what happens when the code executes. Let us say that I want to calculate 4 c 2. Now, first when the program starts executing, you start with code on the first line of the main.

So, you scanf n and k. So, n is 4 and k is 2. Now, use do t 1 equal to factorial of n. So, when t 1 equal to factorial of n is called, what you do is, you set up the return value and return address. So, return value is not yet decided to return address is 5, because, you have to go back to line 5 of the code. So, that is why the return value is 5. Also what you need to do, you need to copy the parameter value which is 4. So, this is the actual parameter 4 and you have to copy it to the input argument r.

So, r is the input argument, r should be assign to the value n here, n is 4. So, that is known as passing the argument. Now, once that is done the code can be seen us jumping to factorial. So, as soon as the function is called, you actually pass the execution to the factorial function. Now, inside the factorial function you have two in local variables i and ans which is answered. And we start executing the factorial function. So, let us see what happens, when we execute the factorial function. So, far we have passed the arguments and so, on.

(Refer Slide Time: 07:22)



```
# include <stdio.h>
int fact(int r) {   /* calc. r! */
    int i;
    int  ans =1  ;
    for ( i=0;   i < r;    i=i+1) {
          ans = ans *(i+1);
    }
    return ans;
}
```

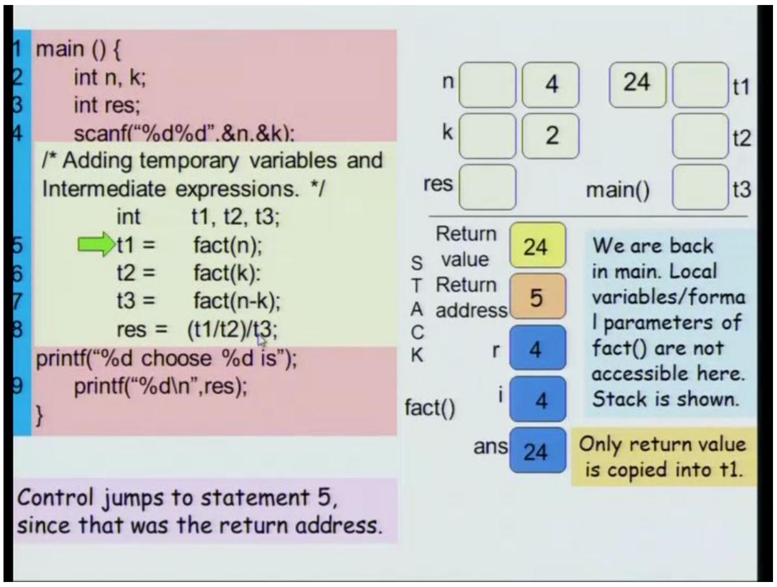
We have jumped to fact() and prepared the stack for the call. Parameters are passed, return addr is stored and local variables are initialized. Now we are ready to execute.

Assign to return value

Now jump to return address

Now, I have just hidden the part of the stack that was used for name. And let us focus just on the factorial function. This computes the factorial function, that we are familiar with this nothing new here. So, it has a variable I which keep track of how many times it has loop has executed and r is notice 4. So, you compute the factorial of 4. Finally, when r equal to 4 answer equal to 24 now, this 24 value we say return the answer value. So, answer value is 24. So, this will be copied to the return value location. So, the return value will get the value 24 and now jump back to return address. So, return address is line 5.

(Refer Slide Time: 08:28)



```
1 main () {
2      int n, k;
3      int res;
4      scanf("%d%d".&n.&k):
  /* Adding temporary variables and
  Intermediate expressions. */
           int       t1, t2, t3;
5          t1 =      fact(n);
6          t2 =      fact(k):
7          t3 =      fact(n-k);
8          res =  (t1/t2)/t3;
  printf("%d choose %d is");
9      printf("%d\n",res);
  }
```

Control jumps to statement 5, since that was the return address.

We are back in main. Local variables/formal parameters of fact() are not accessible here. Stack is shown.

Only return value is copied into t1.

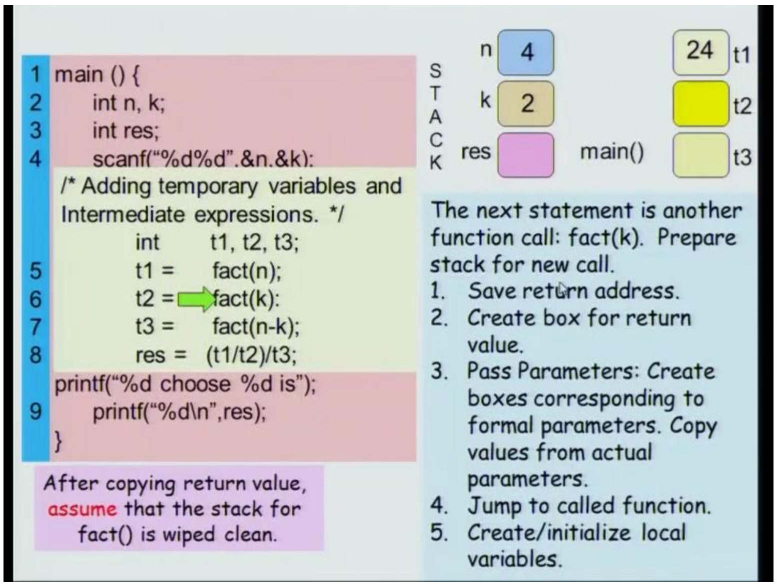
So, will jump back to line 5 and there we will say that t 1 equal to 24. Only the return value is copied back to the main program all other things are irrelevant. So, the correct way to imagine what happens. When the function has returned is that, the stack that was allocated to main to the execution of fact is completely erased. So, once we go back to main as soon as the function returns back to the main. You should imagine that the entire stack is deleted, and only the memory that was originally allocated to main remains.

So, the correct way to think about a function executing, you can imagine that, you are main and you have a friend, who can calculate factorial for you. Now, you can ask your friend to calculate factorial for you and things are done in a very hygienic manner. So, what you do is, you write on a piece of paper the number 4 and give it your friend. Now, your friend is another room. So, he has at his disposal some black board. So, he looks at the number 4 and using the private local variables that he has, which is I and a result or answer, he calculates the factorial of these numbers. Once see does that, he copies the result back on to a piece of paper. So, 4 factorial is 24 and brings it back to you. Before he does that, he erases the black board and he will bring back the number 24 on a piece of paper.

Now, you can imagine that the space that your friend used to compute 24 has now been wipe clean. And all that remains is the value 24 which you can copy back on to your note book. So, this allegory tells you exactly what happens in the case of function execution. You write down what you want the factorial of on a piece of paper, pass it your friend, he will go to a separate room. And he will calculate whatever he wants. Once he does that, he will clean his black board, right down the result on a piece of paper and bring that paper back to you.

So, as far as you are concerned you are least bothered with how he is computing the factorial function. All you want as the result. And this is the basic way to thing about functions. You should be able to reason out a bigger program by saying, what does as a smaller program, what does as a smaller function do regardless of how that function does it.

(Refer Slide Time: 11:45)



```
1 main () {
2       int n, k;
3       int res;
4       scanf("%d%d",&n,&k);
  /* Adding temporary variables and
  Intermediate expressions. */
            int     t1, t2, t3;
5           t1 =    fact(n);
6           t2 =    fact(k);
7           t3 =    fact(n-k);
8           res =   (t1/t2)/t3;
  printf("%d choose %d is");
9       printf("%d\n",res);
  }
```

After copying return value, assume that the stack for fact() is wiped clean.

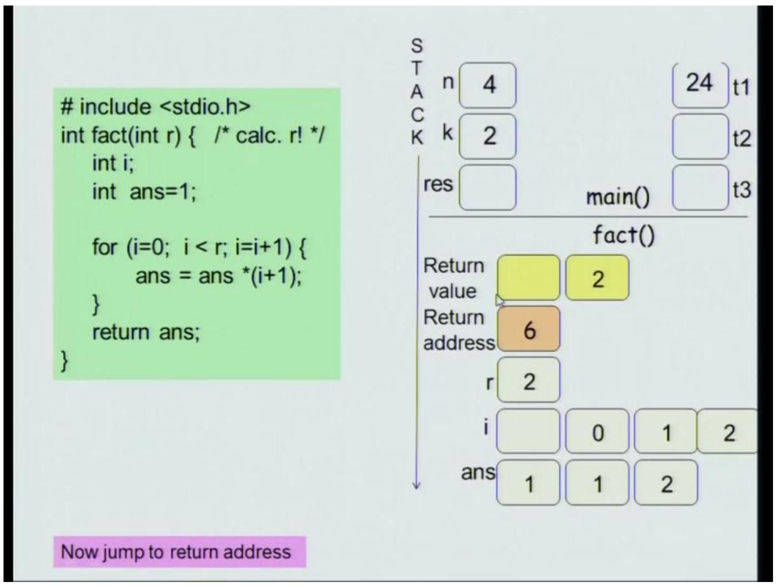
The next statement is another function call: fact(k). Prepare stack for new call.

1. Save return address.
2. Create box for return value.
3. Pass Parameters: Create boxes corresponding to formal parameters. Copy values from actual parameters.
4. Jump to called function.
5. Create/initialize local variables.

Now, let us get on with the remaining execution. We have just computed factorial of 4. Now, we need to calculate factorial of 2 and factorial of 4 minus 2. So, we go to the next line, the next line also involves the call to factorial of k. So, we do the same things again, we save the return address. Now, the return address is 6. Because, we are executing line 6, then we create a box for the return value and pass the parameters, and finally, jump to the called function.

So, we do all that we have some memory for main. But, we allocate a new space in the stack for executing factorial. At this point return address is 6. Because, it is a second factorial that is being called, r is 2, because, k is 2 and you execute the factorial function.

(Refer Slide Time: 12:43)



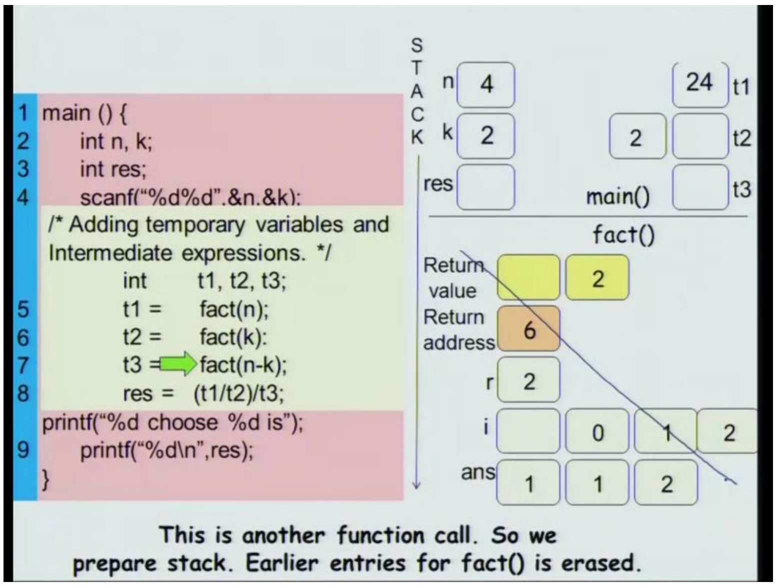
```
# include <stdio.h>
int fact(int r) {   /* calc. r! */
     int i;
     int  ans=1;

     for (i=0;  i < r; i=i+1) {
          ans = ans *(i+1);
     }
     return ans;
}
```

Now jump to return address

So, you again go to the factorial function and calculate 2 factorial, 2 factorial is 2. So, that will be transferred back to the return value

(Refer Slide Time: 13:00)



```
1 main () {
2      int n, k;
3      int res;
4      scanf("%d%d",&n,&k);
  /* Adding temporary variables and
  Intermediate expressions. */
          int       t1, t2, t3;
5         t1 =      fact(n);
6         t2 =      fact(k):
7         t3 =      fact(n-k);
8         res =   (t1/t2)/t3;
  printf("%d choose %d is");
9      printf("%d\n",res);
  }
```

This is another function call. So we prepare stack. Earlier entries for fact() is erased.

And now you can imagine that, you will get back to the address 6, where t 2 will have the value 2. So, once you do that again the thing to imagine is that, this slate is wiped clean. And all the memory that you allocated to the stack is now free. So, all once you are back in main all you have as the memory for me. Now, there is the third call to factorial. Factorial of n minus k and it is done in exactly the same manner without much elaboration. So, it will create n minus k is 4 minus 2 which is also 2 and the return address is 7 equation n.

And once you do that, it will execute the factorial code again, and calculate the factorial of 2 which is again 2 and return to line 7. So, 2 will be copied as the return value and once the execution finishes, you return to line 7 of main program. At this point, you say that t 3 equal to 2. And you can imagine that the stack allocated to factorial is now erased. So, at this point main has t 1 equal to 24, t 2 equal to 2 and t 3 equal to 2. You have all the information that you need in order to calculate your result.

So, you calculate 24 divided by 2 divided by 2 and the answer is 6 which is 4 choose two. So, this illustrates how do you write a function? How do you define a function? And how do you call it? And what actually happens when you execute a main function? So, the execution of a function can be visualized as a stack. A stack is a part of memory, that is allocated as private to a new function that is being called. Once that function finishes execution, the stack is erased and you go back to the previous function. And you go back to the calling function.