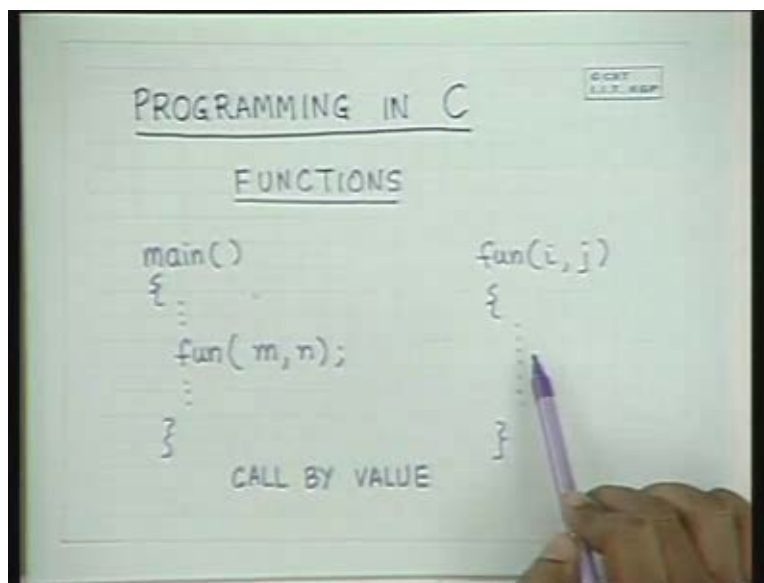


Programming and Data Structure
Dr. P.P.Chakraborty
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture # 04
C Programming - 3

We shall continue our study of programming in C and we shall pick up from where we left of in the last class. In the last class we were discussing functions and C allows the definition of large number of variety of functions but the format in which functions are called is very simple. A function, there are no individual items like subroutines and functions there is no difference between a subroutine and function in C, everything is a function. and you have the actual arguments of the function and the formal parameters in the function declaration. And as we mentioned previously, the calling is call by value. In call by value, the value, the idea can be understood by the following scheme that whenever a function is called, the values **of the formal** of the actual parameters are copied into the instantiated values as instantiated values of the formal parameter in the called function. That is these form local variables of the new function and the values of the main program from where it is called or any other function from where it may be called are copied here and executed. And after the execution, we are not copied back therefore whatever you call it with the actual values of these variables will not change.

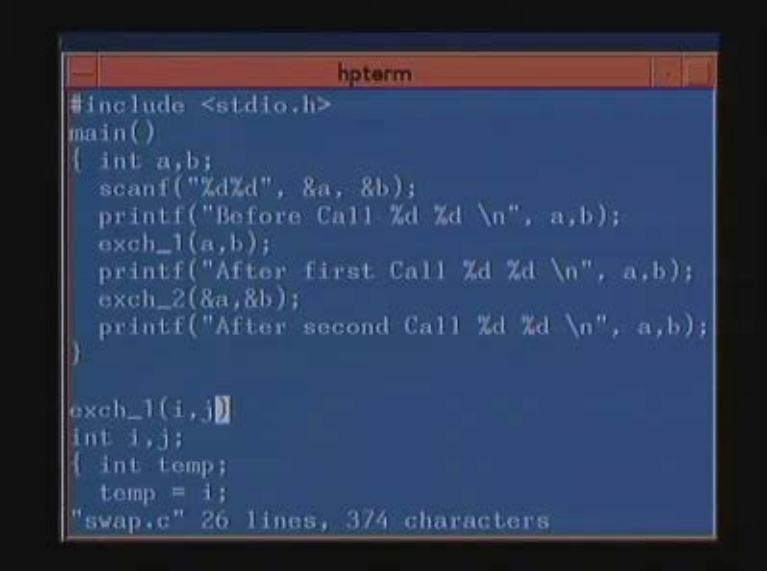
(Refer Slide Time 02:57 min)



And we understood in the previous class that if we require to change the contents of m and n, we would actually have to pass the addresses and once we pass the address, we would have to change the contents specifically from the addresses. So we will pick up from there and we will see another example and then move on to something else. So let's go back and watch an example of an exchange program. Now this is the program, we have got the main function here up to here and in the main function, we declare two variables a and b which are integers and then we read them and before the function is called, we print the variable of a and b. We call exchange 1 a b,

this is `exch_1` a b and after the call of this, we again print the values of a and b. Here we are passing the actual values and here in `exch_2` a b, we pass the addresses of a and b by saying `&a` and `&b` and after this call we return and in both we attend to swap.

(Refer Slide Time 04:03 min)

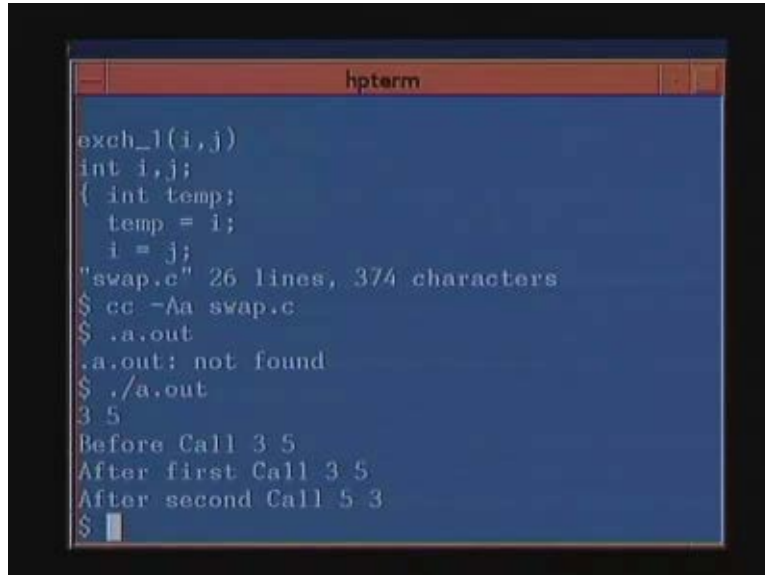


```
hpterm
#include <stdio.h>
main()
{ int a,b;
  scanf("%d%d", &a, &b);
  printf("Before Call %d %d \n", a,b);
  exch_1(a,b);
  printf("After first Call %d %d \n", a,b);
  exch_2(&a,&b);
  printf("After second Call %d %d \n", a,b);
}

exch_1(i,j)
int i,j;
{ int temp;
  temp = i;
  "swap.c" 26 lines, 374 characters
```

So in the first one `exch_1` i j the actual values are passed and we do a swap like this. Therefore since a and b are integers in the call `exch_1`, here in `exch_1` i and j are also integers and we do a simple exchange here, whereas in `exch_2` we pass the addresses of a and b and therefore we declare i and j here to be pointers or address locations in the sense that the content of i is an integer and the content of j is an integer. Star as we mentioned before is the content of a variable. Sometimes they are also called addresses; sometimes they are also called pointers. In C the language often used is pointers and we shall slowly adjust to this language as we move through the course. But it is always easy to understand the meaning of variable declaration this way. this way that the star i which means the content of i is of type integer. And here we specifically change the contents because we got only the addresses of the pointers to the location. So we do a swap temp is equal to content of i, content of i is equal to content of j and content of j is equal to 10. And as it is a call by value what do we expect here whatever we got for a and b after the call the values of a and b will not change. On the other hand here since we are passing the addresses and we doing an exchange, the value of a and b should change here, so just we will execute it and see how it works.

(Refer Slide Time 06:13 min)

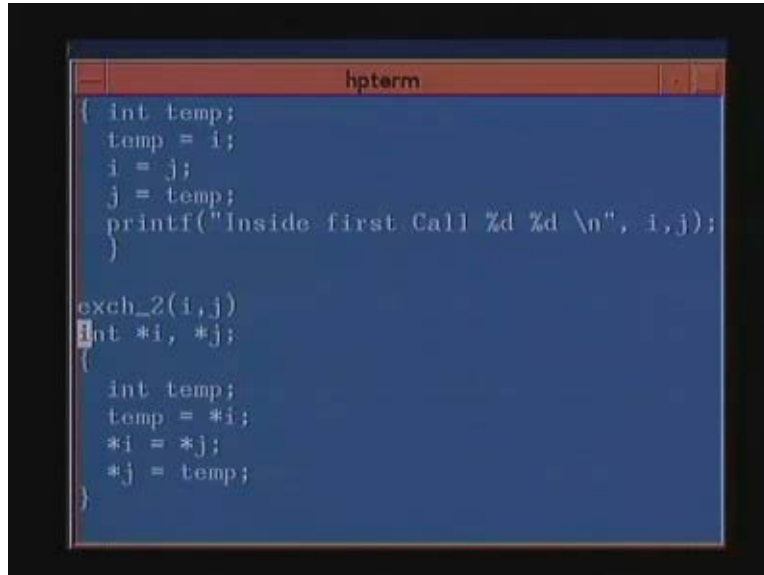


```
hpterm
exch_1(i, j)
int i, j;
{ int temp;
  temp = i;
  i = j;
}
"swap.c" 26 lines, 374 characters
$ cc -laa swap.c
$ ./a.out
.a.out: not found
$ ./a.out
3 5
Before Call 3 5
After first Call 3 5
After second Call 5 3
$
```

So if we pass 3 and 5 after the first call we get back 3 and 5 and after the second call we get back 5 and 3. This does not mean that the exchange did not take place here. If we want to see whether the exchange really took place here, we will just see what happened here and that will help us to understand that whether the exchange actually took place. So inside the first call, we will print the values of i and j. If you print the values of i and j inside the first call, we should be sure whether the value exchanges actually took place or not and we will see that the exchange actually does take place.

You see here before the first call it is this, inside the first call exchange takes place but after the first call it will again, the values of a and b remains the same. This is due to the fact that the call is by call by reference or call by value **sorry it's not call by reference** its call by value. Therefore even though the exchange takes place here, **the values of a and b do not affect the**, i and j do not affect the values of a and b.

(Refer Slide Time 07:51 min)



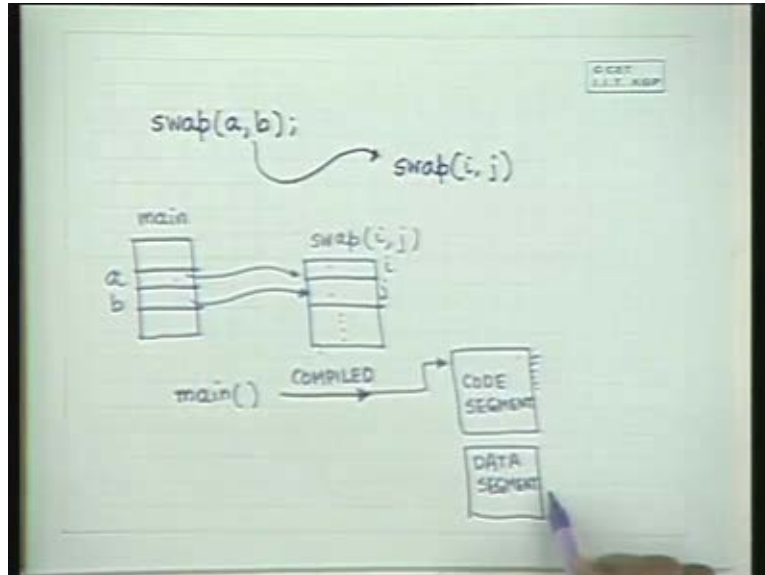
```
hpterm
{
    int temp;
    temp = i;
    i = j;
    j = temp;
    printf("Inside first Call %d %d \n", i,j);
}

exch_2(i,j)
int *i, *j;
{
    int temp;
    temp = *i;
    *i = *j;
    *j = temp;
}
```

On the other hand if you pass the addresses and change the contents of the location, the values of the addresses do not change but the contents are changed. Therefore even it's a call by value by passing address locations and changing the contents, we can make modifications in the fields. And this is how we do call by reference, if we want to work it out in C. So in C language what we have is we have the swap routine like this, a call to the swap routine was made like this swap a b and this generated a call to swap i j. And the values of the variables here a and b of the main function were copied into the local variables of the data segment of the swap function. This was i, this was j and you can have several other local variables. This was copied here and it goes back and once the call is over, the values are not copied back.

Now in any function call in C, a schematic way of understanding how the functions operate and how the data operates is that for every program there is a static data segment associated with the function itself. That is the code when you have the, suppose you are given a function main, now when it is compiled, it is compiled into two parts. One is called the code segment which contains the machine code for the program and some locations for certain types of data and the other part is called the data segment which contains all the local variables of a particular function whether it is main or whether it is something else. So there is a code segment and there is a data segment.

(Refer Slide Time 10:52 min)



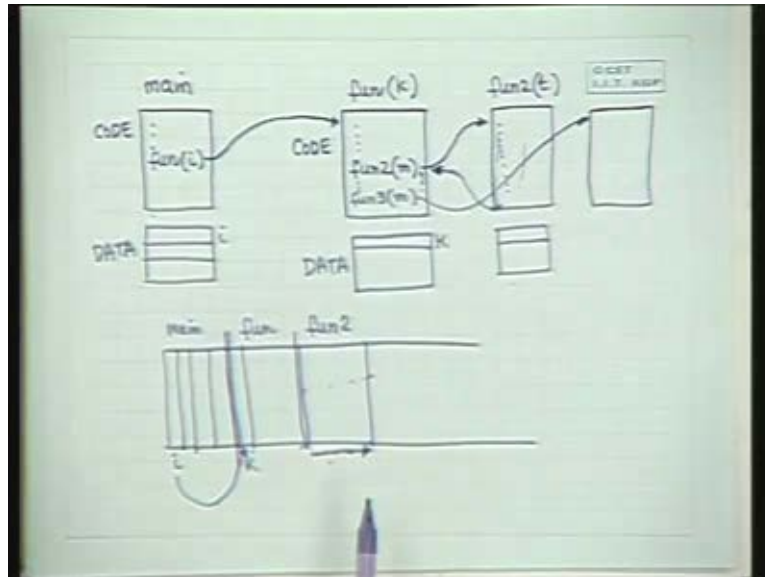
Now whenever a function is called, the code segments starts executing and new data segment is created for that function. So let us see slightly in more details how this works. So suppose we have a code segment for a main program and a data segment for the main program, so this is main and for a function `fun` let us say we have a code segment and a data segment. So this is the code, this is the data, this is the code, this is the data. So in the beginning, it will start executing from the main program and maybe here there is a call to fun. Now as soon as there is a call to fun, so initially when its starts executing in the main program there is a dynamic data segment which is created. So the data segments of the main program, the variables of the main program which are part of the data segment for example say this is fun (i) and this is a local variable, so this gets loaded into a portion of the memory.

Now whenever there is a call to this, a data segment corresponding to this gets stacked up, so this is main, this is fun gets stacked up here. So this may be fun k, so corresponding to k there was a block which got loaded up here and due to the call this value got copied here. And now suppose fun calls another one say fun2 (m) and fun 2 is another function which has its own code and data segment then whenever there is a call here, this starts executing and as soon as this starts executing a data segment corresponding to fun 2 gets stacked up at this point. And when this code completes execution, the control returns here to from where it was called and this starts executing. And this data segment is no longer useful, so this data segment is now as we can say in stack language is removed or popped off from this position.

And suppose there was another call say to fun 3 (m) and there was a third function fun 3, then as soon as this call is make after this call a new data segment from fun 3 will start from here just after fun. So it works like this, when this is called a data segment is put here, when this calls another one the data segment of this is put beyond it. When suppose this completes execution and goes back then this data segment gets popped off and then if this calls it this one then again a data segment for fun 3 starts from here. So this is a schematic, this is very useful in understanding exactly how the program works, a sequence of calls work. So we will keep this

schematic structure in mind and we will see that most of the data which are variables are dynamically allocated in the data segments. For example they just go on being stacked one after another.

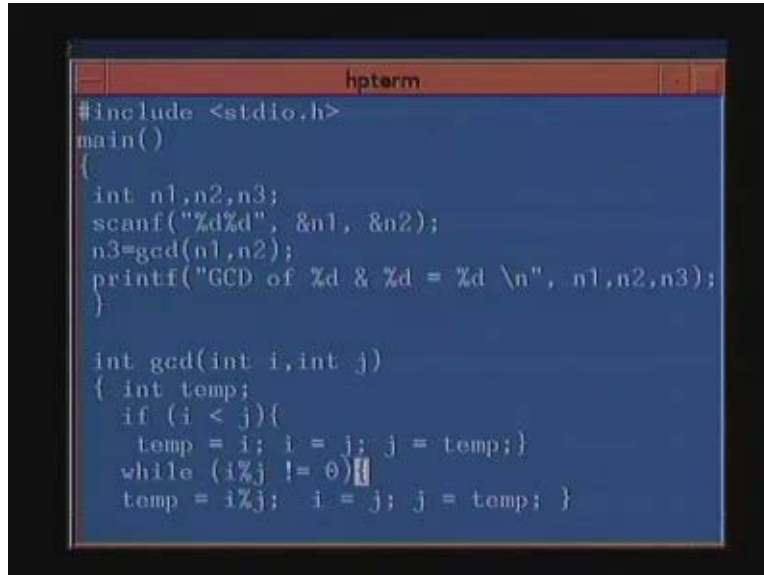
(Refer Slide Time 15:01 min)



However there are certain types of variables in C which you can declare to be fixed and they will exist in the code segment. We will come to such examples later on but this dynamism of the data segment should be understood more clearly. And we shall see some other programs to understand how exactly this works. This will be especially important when we come to recursive programs, when we know that various data segments of the same function may exist in the stack because the function maybe calling itself directly or indirectly. So we will come to such functions in this class but before we go to that int c a function can be used like we have used as a subroutine before in swap but a function can also return a value.

So, in C you can make a function return a value and put it on the right hand side of any expression other than making it like a subroutine call which we did in the example of swap. And we shall see such an example for the computation of gcd before we move on to another more involved in some. So let's go back and see. This is a program which computes the gcd of two numbers. This is the main program, it has got three integer declarations int n1, n2 and n3. It reads in n1 and n2 and computes n3 is equal to gcd n1 n2. So now here we have used it as a function which is going to return an integer value and then we print the value of gcd saying gcd of this and this is this, n1 n2 n3.

(Refer Slide Time 16:59 min)



```
hpterm
#include <stdio.h>
main()
{
    int n1,n2,n3;
    scanf("%d%d", &n1, &n2);
    n3=gcd(n1,n2);
    printf("GCD of %d & %d = %d \n", n1,n2,n3);
}

int gcd(int i,int j)
{ int temp;
  if (i < j){
    temp = i; i = j; j = temp;}
  while (i%j != 0){
    temp = i%j; i = j; j = temp; }
```

And here we have declared the function gcd. The function gcd takes in two arguments i and j and we can declare the arguments to be integers inside here we have seen how to declare it outside the parameter list, we can also declare it here. And we declared the function itself to be an integer and we compute the gcd by the standard well known technique of computing gcd. That is if i is less than j, we exchange i and j so that i is greater than or equal to j and then we divide i by j and take the remainder. This one finds out the remainder between i and j and if the remainder is not equal to zero, we continue by making j moving i to j and moving j into the temporary location. The temporary location is the remainder. So this is the standard and well known technique for computing gcd of two numbers. And here we have used it as a function and we have used the modulus operation which is i percentage j which returns the remainder of the division between i and j.

So as long as the division is not equal to zero, the remainder does not becomes zero. Whatever we do is **we make the dividend, we make it the divisor sorry**, we make the divisor the dividend and we make the remainder the divisor and continue. So I hope this is okay and we just execute this. Therefore the gcd of 45 and 105 is 15. Gcd of 99 and 33, gcd of 37 and 53 is 1. So this is the function which returns a value and you can make this function into returning anything, it can return integer, it can return character, it can return several other things and we shall see how to declare even complex such returns for a particular function. And in order to make it return something, you have to specifically give the statement return. So we have given the statement return j, it will therefore when the function completes the execution, the value of j will be returned.

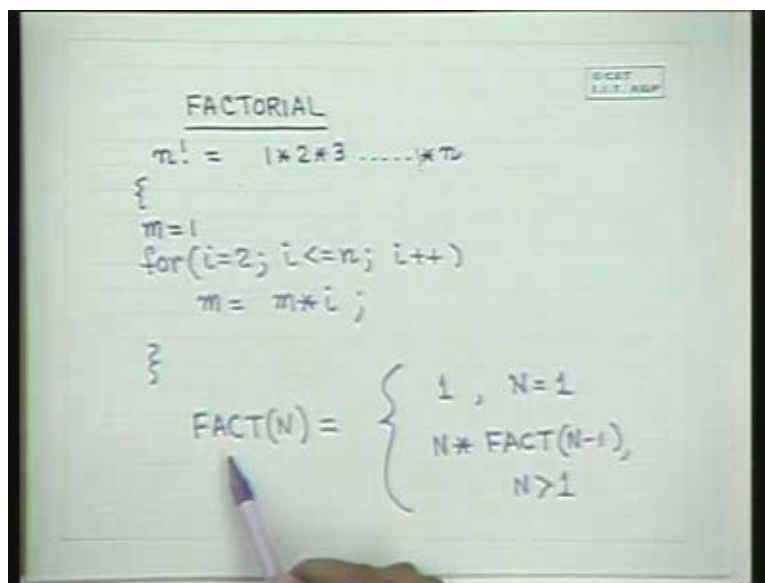
Here just to introduce the while statement, so this is the while statement and in this while statement is like any other standard while in another programming languages, the loop starts here and continues up to this point and in that while statement we have just converted the divisor and the remainder and adjusted their values and the meaning of while is as long as this condition is true, it will execute. when this condition is false it will jump out of the loop. So we have seen the

for statement for looping and here is the example of a while statement for looping. There is another statement called the do statement which we shall come to later on.

So we have seen that a function can be used to return values, how to change the values of the parameters and we have seen how functions can be used to return values themselves and be used in any expression. Now we shall see slightly advanced feature of a C language which does not exist in languages like FORTRAN and we will take up the classical example of computing the factorial. All of us know that n factorial is equal to 1 into 2 into 3 into n. And an easy way in which we can write out the factorial program or program which those who have written in FORTRAN would have written down always was something like this, m is equal to 1, its C equivalent would look like this. This would be the C equivalent for the program; just in a loop you multiply 1 into 2 then into 3 then into 4 then into 5.

However factorial has got another interesting in that definition and the definition of factorial can be written out inductively like this. Factorial of n is equal to 1 for n equal to 1 and is equal to n into factorial of n minus 1 for n greater than 1. So factorial of n is equal to 1 if n is equal to 1 which is the base of the inductive definition, otherwise it is n multiplied by the factorial of n minus 1 which you can obviously see if you look at it backwards, it is n multiplied the factorial of n minus 1. Therefore we get here an inductive definition and in many cases the problem solving you will see that induction is a way of decomposing a problem into many parts.

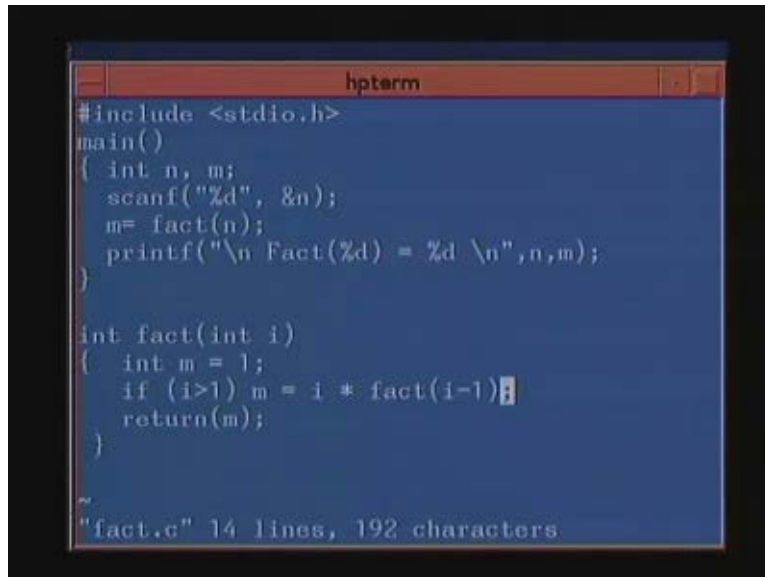
(Refer Slide Time 23:40 min)



And C language provides you a mechanism for directly working out such an inductive definition. That is we are able to get an inductive definition if the programming language allows a function to call itself. And we shall see that we can write down such functions directly in C language. So we will write out a factorial function in C using this idea and see how it executes. So let's go back and see how factorial executes. So this is the main program. We have declared 2 int n and m, we have read in the value of n and we have called in the main program m is equal to factorial of n. And we have printed that the factorial of n is equal to m and what have you done in the

main program in the function fact. We have declared a function called int fact which takes an integer parameter; we have initialized int to 1. Now look here we can do an initialization even during declaration. So we are slowly introducing various concept of C without just learning them one by one, I will just introduce them as and when we continue our programming. So we can initialize a variable like this but the main part is this. If i is greater than 1 then m is equal to i into fact of i minus 1. So here we have done the calling of a function and this function calls fact, fact calls itself and we return the value of m.

(Refer Slide Time 25:26 min)



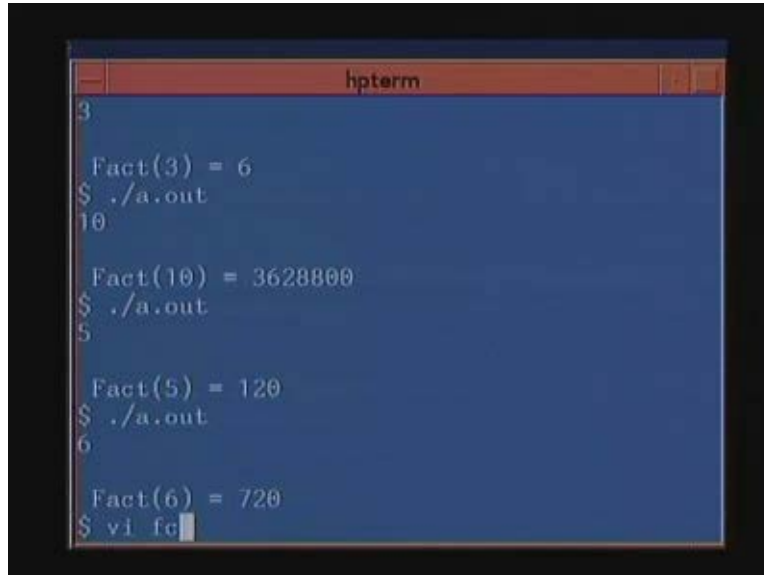
```
hpterm
#include <stdio.h>
main()
{ int n, m;
  scanf("%d", &n);
  m= fact(n);
  printf("\n Fact(%d) = %d \n",n,m);
}

int fact(int i)
{ int m = 1;
  if (i>1) m = i * fact(i-1);
  return(m);
}

~
"fact.c" 14 lines, 192 characters
```

So if m is equal to 1, it returns 1 otherwise it computes this value. So we have written out a definition of a function which calls itself. Such a function is called a recursive function. So let us first see whether it works. So if you give 1, we get fact 1 is equal to 1. If you give 2, if you give 3, if you give 10, I think 5 is a value which we know.

(Refer Slide Time 26:18 min)

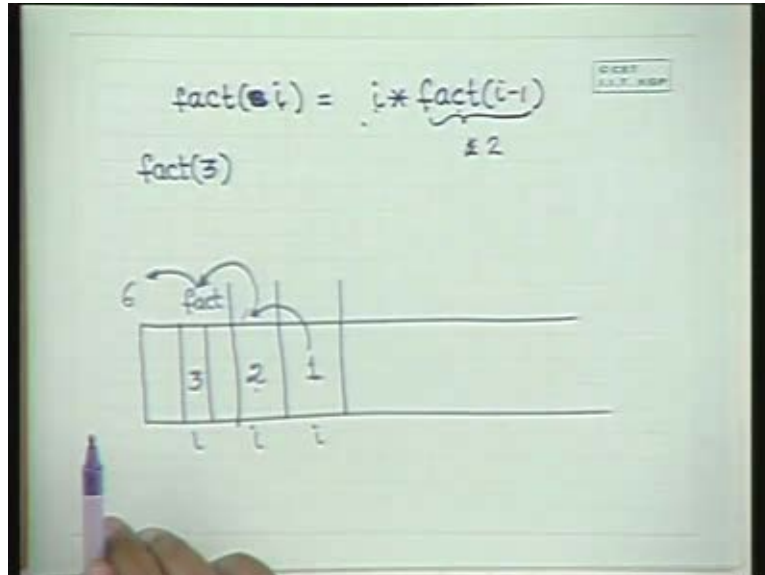
A screenshot of a terminal window titled 'hpterm' with a blue background. The terminal shows a sequence of recursive factorial calculations. Each calculation starts with a number, followed by the output 'Fact(n) = value', and then a command to run a program './a.out' which prints the next number. The sequence is: 3, Fact(3) = 6, ./a.out, 10, Fact(10) = 3628800, ./a.out, 5, Fact(5) = 120, ./a.out, 6, Fact(6) = 720, and finally the command '\$ vi fc' is entered at the prompt.

```
hpterm
3
Fact(3) = 6
$ ./a.out
10
Fact(10) = 3628800
$ ./a.out
5
Fact(5) = 120
$ ./a.out
6
Fact(6) = 720
$ vi fc
```

So we do get the factorial value, if we write out such a program or function which calls itself. So such a recursive function which calls itself. Now how does such a function execute? That is the most interesting part and we will go back to our concept of the data segment stacking one after another and we will see how it executes. So our basic part of the recursion was fact n or say fact i was equal to i into fact of i minus 1 that is if i is greater than 1. Now suppose it is called with a value. So in the main program, suppose you read in the value 3 and you call it with the value of fact 3. So now what happens here for a data segment for fact is called and here you have got the variable i which takes in the value 3. Now once we get the value of 3, you again call fact.

So another data segment here for i is created with i minus with value i minus 1, so you get 2. Now once you get two, so this is still greater than one, again you call fact. So another data segment is created **with value** with the value of i to be 1. So the data segment, new copies of the data segment continue to be created one after another like this. Now when i is equal to 1, this condition does not hold and you return 1. So once you return 1 you come back here, you come back to this program and this value, the value of here computes to 1 and so this computes this comes back here and compute 2 into 1. And this value is returned back here which computes, which goes back and the computation there now returns 2.

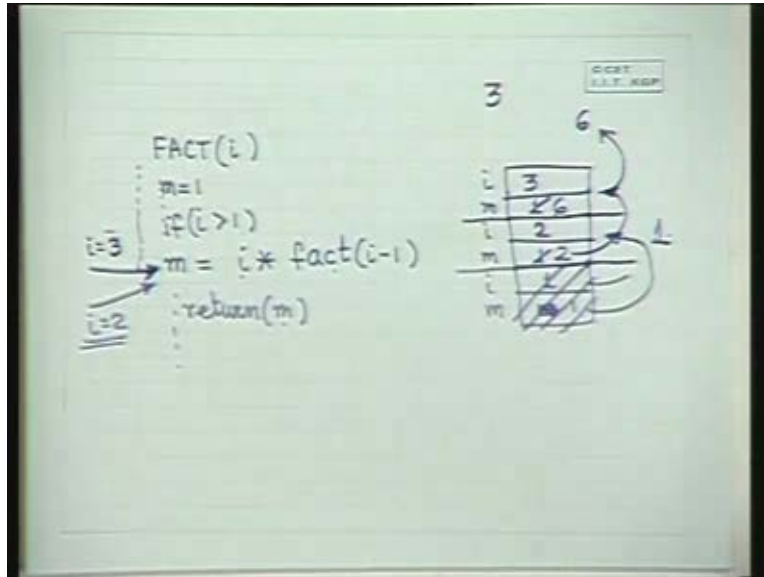
(Refer Slide Time 28:50 min)



Now this value **go back** goes back here and $i = 3$ into 2 returns 6 . So what happens when a program is called is that you have got the fact program which in some place says m is equal to $i * fact(i - 1)$, if this is, if i is greater than 1 . So whenever this starts executing say as we said with 3 , at this point of time it makes a call to another function. So now we will understand it in a slightly smaller schematic then it's one variable only which is i , another variable which is m . so **there is** there are two variables in the data segment of fact i and m . So **this is** there is a pointer which weights here and calls it and it is waiting for the return for the call when i is equal to 3 . So here i is equal to 3 and m we are initialized it to 1 , so it's just 1 . And at this point, it calls fact with fact 2 , so again two local variables for i and m are created, this with two and this when its starts execution it initializes it to one comes here, here i is 2 and it again calls fact. So another segment of two variables are called i and m , here i is 1 , m is initialize to 1 . This condition does not whole true **so it returns**, so it returns this value. And once it returns this value, control goes back to this and this goes out. control goes back here and once controls goes back here, immediately this value is returned, this value is returned from m sorry m was 1 , so this value is returned as 1 .

So now m into 1 , m becomes, this is 1 , 2 into 1 is computed here and it returns m , so m now becomes 2 and this value is returned back. So control now goes back to this pointer for i equal to 3 which was waiting here and this now computes 3 into 2 and returns 6 . So this will return 6 after making this 6 , it will return 6 . So this is the way in which you can understand it, you only have one recursion. We will have recursion of various types later on.

(Refer Slide Time 31:56 min)



In order to understand such things, it is also useful to just put in some diagnostic statements in your program. For example I have just in the program, this is the same as the previous program. We read in `n` and then `m` is equal to `fact` of `n` and then we print it. Here as soon as it enters, I printed in `fact` inside the factorial program and I print in the value of `i` and `m`. And then we compute this and after it comes out, again I print in the value of `i` and `m` and we know that `m` is going to be returned.

(Refer Slide Time 32:36 min)

```
hpterm
#include <stdio.h>
main()
{
    int n, m;
    scanf("%d", &n);
    m = fact(n);
    printf("\n Fact(%d) = %d \n", n, m);
}

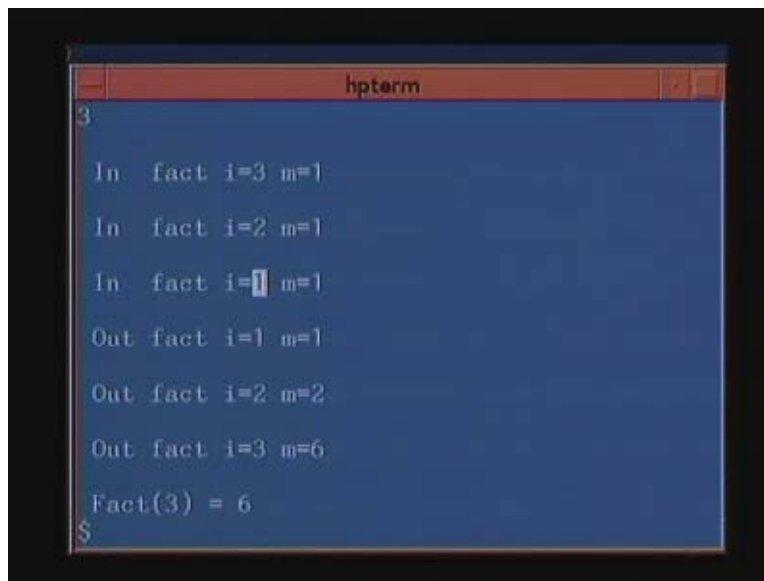
int fact(int i)
{
    int m = 1;
    printf("\n In fact i=%d m=%d \n", i, m);
    if (i>1) m = i * fact(i-1);
    printf("\n Out fact i=%d m=%d \n", i, m);
    return(m);
}

"fact2.c" 16 lines, 280 characters
```

So if we execute this, we will see exactly the sequence in which calls are made and the values of the variables are occurring.

So suppose you put in three as we put in previously, now see what happens.

(Refer Slide Time 33:08 min)

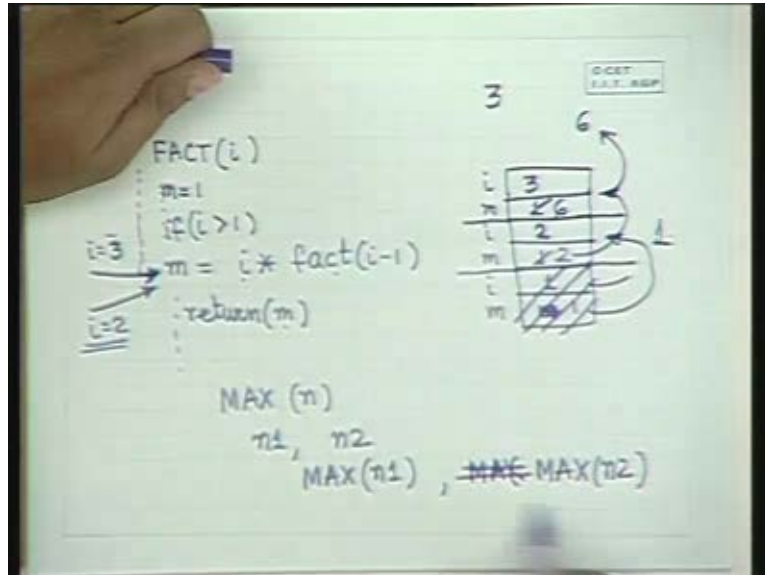


```
hpterm
3
In fact i=3 m=1
In fact i=2 m=1
In fact i=1 m=1
Out fact i=1 m=1
Out fact i=2 m=2
Out fact i=3 m=6
Fact(3) = 6
$
```

You first go inside the fact routine with i equal to 3 and then you call it with i equal to 2 then you call it with the i equal to 1. This one terminates first, it goes back to the call which was made with i equal to 2. This one then terminates and returns to and then this one terminates and returns 6 and finally you get your value. And if you put in larger values, you will always be able to see exactly the sequence of calls and we can print in all the local variables to see what are the values of the variables before the call is made that is just as you enter the call and just as you go out. So this is the sequence in the factorial program.

And finally the factorial program, if you have understood the factorial program then you need not write down m is equal to fact n . You can simply put in the function call even in the print statement expression. It works like any other expression and you can put in the factorial call here. Similarly you can put in a factorial call here in the return statement also because a function can take part in an expression. So here what we have done is we have re-return the factorial routine with i equal to 1, return one else return i into fact of i minus 1. So you just put in a recursive function here which is an expression and still it can be executed inside the factorial return. So this is the same as the original factorial function just that I have introduced to the concept that since a function returns a value it can take part in an expression and it can take part in an expressions like this. So this is how recursive programs work and we shall see more complex recursive programs in future classes but before terminating this lecture, I would like to reiterate the fact that recursive programming is a very very useful thing in problems solving. Especially problem decomposition can be done mentally in a large number of problems; you can mentally decompose the problem into two parts.

(Refer Slide Time 35:54 min)



As we said even that minimum of numbers even divided into two parts where splitting the set of numbers that you are supposed to read into say two parts and then recursively solving one part and recursively solving the other part. That is if you are asked to solve the maximum of n numbers, we will split it up into two parts of n_1 and n_2 numbers. And then compute recursively the maximum of the first n_1 numbers, **the maximum** the maximum of the next n_2 numbers and then find out the larger of the two. So recursive decomposition inductive principles will be very useful in our study of programming methodology. And it is very nice that if programming, advanced programming languages now do support recursive programming. And we shall make ample use of such recursive functions while solving more complex problems. We shall also see more of recursion in future classes but I think today we will stop here.