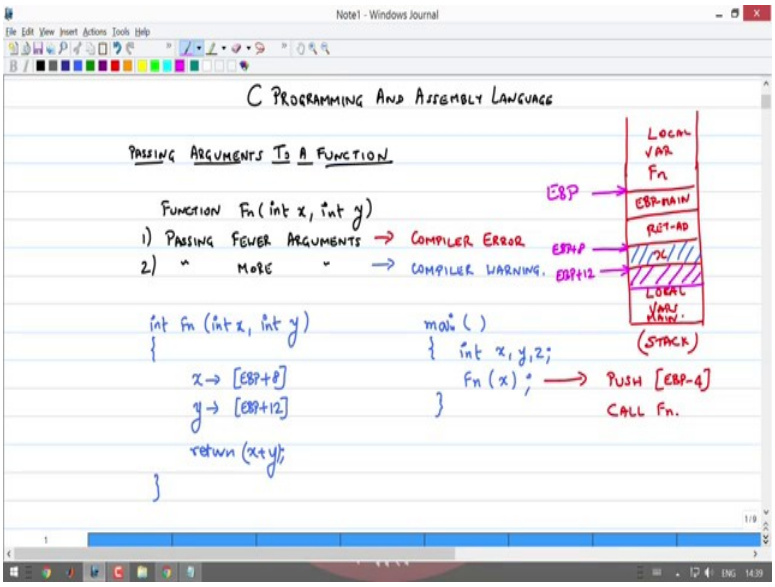
C Programming and Assembly language
Prof. Janakiraman Viraraghavan
Department of Electrical Engineering
Indian Institute of Technology, Madras

Lecture - 16

So, welcome back to this course on C Programming and Assembly language. In the last module we looked at how to compile a C program and how to translate local variables into you know EBP minus some offset, how to translate function parameters to EBP plus offset and so on . In this module, we will use that information that we have learnt in module 3. And look at the impact of certain corner cases in C programming . And we will also look at some special functions like print f and so on.

(Refer Slide Time: 00:49)

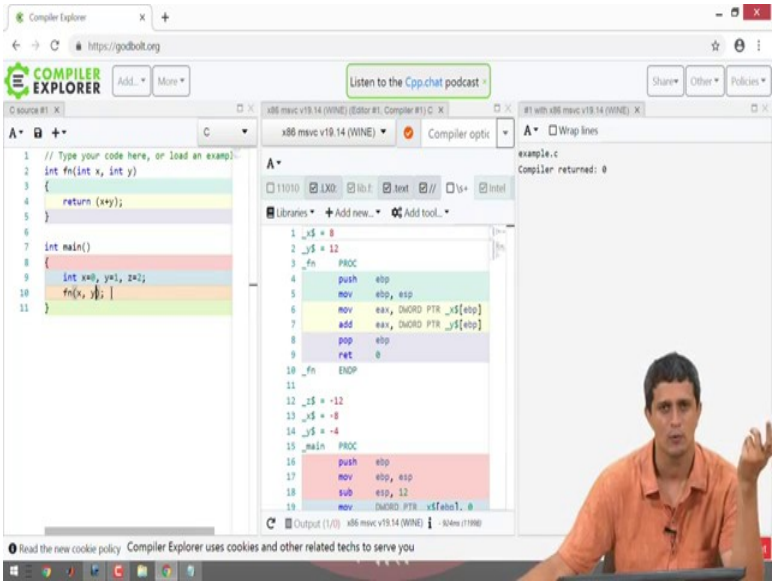


So, in this lecture I would like to discuss the following I want to look at passing arguments to a function . So, this is a precursor to studying what we have to do in print f . So, let us look at 2 cases specifically here . So, first is we have a function Fn, which takes some set number of arguments we have a so, let us say then the function Fn, which takes let us say two arguments int x and int y .

So, what I want to study is I want to see what happens, when I invoke this function with fewer arguments and when I invoke this function with more number of arguments, then

what has been defined . So, case one is passing fewer arguments . And case 2 is more arguments. So, let us actually go back and look at our compiler explorer tool .

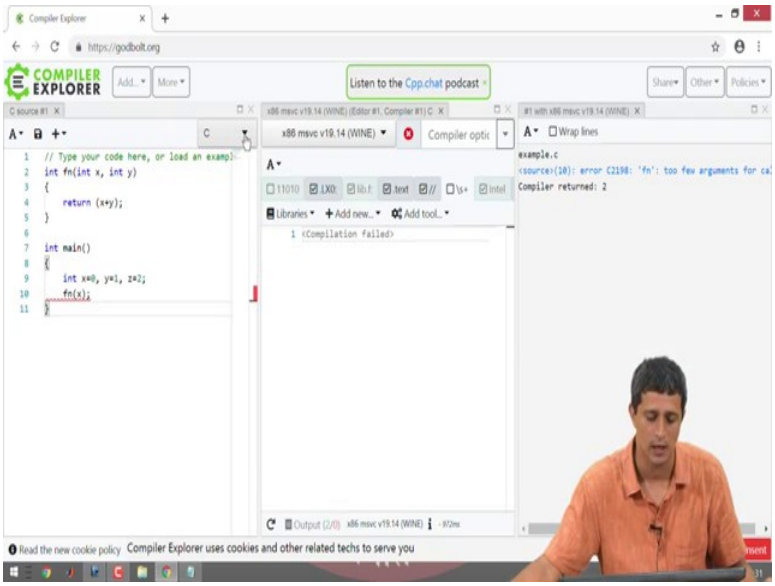
(Refer Slide Time: 02:33)



And understand what happens here. So, so, we have a function fn which takes two arguments x and y and let us say it just returns x plus y the function can actually do anything is this just a dummy function . The main focus is the arguments x and y here.

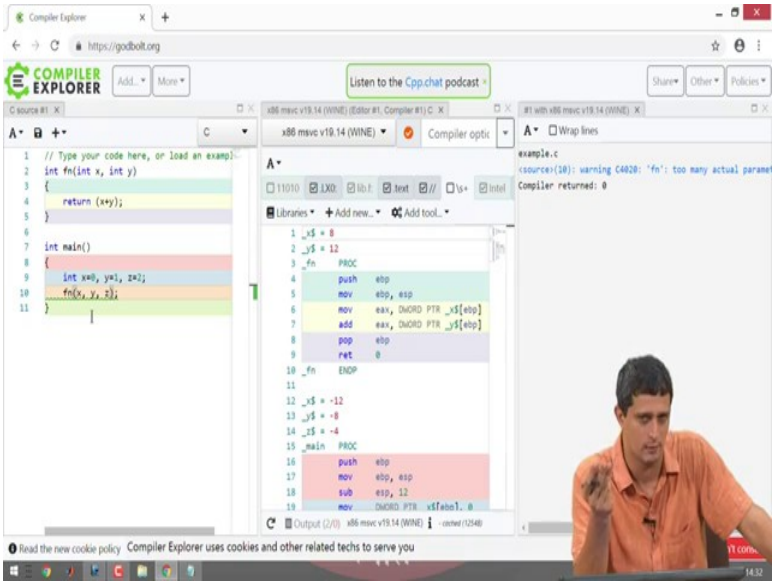
Now, in main what I am doing is I have 3 variables x y and z initialize to 0, 1 and 2 respectively. So, normally I would invoke this function fn as fn of say x and y . So, this is the right way to do it and as you can see there are no warnings or no errors on the right hand side here and the program has got compiled to it is assembly output appropriately . Now, the question is if I remove this y, then what happens.

(Refer Slide Time: 03:27)



So, clearly remember that here I am dealing with a C program , this is not C plus plus this is a pure C program. So, it is essentially saying that fn two few arguments for call right, which is understandable because the function fn has been defined to take 2 variables x and y as arguments. So, let us note this down, if you pass your arguments, then what has been defined this will give a compiler error. Now, on the other hand if I just call it as f of x comma y comma z , which means I am passing 3 variables to it .

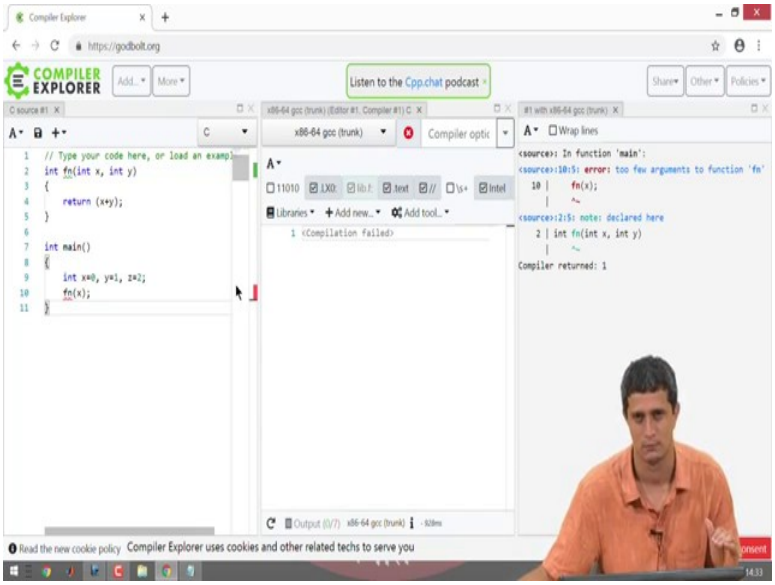
(Refer Slide Time: 04:19)



Then, what happened is that, I see that there is a warning fn too many actual parameters . There is just a warning here, but the program still gets compiled to it is appropriate output without any problem . So, if we instead passed more arguments than what has been defined, you would only you would get a compiler warning. The point is whether it is a warning or an error is up to how the person wanted to implement a compiler it really does not matter s.

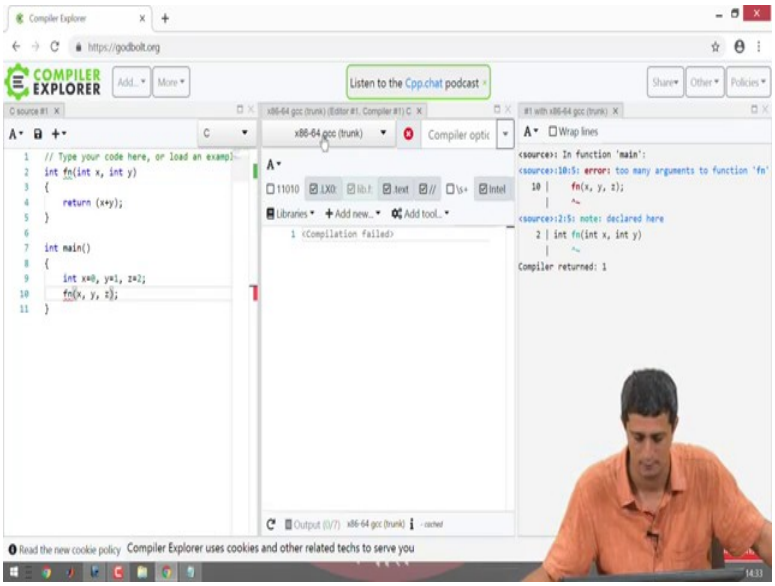
So, by the way this is not you know restricted only to msvc you could go and try your hand at compiling this even with GCC ..

(Refer Slide Time: 05:25)



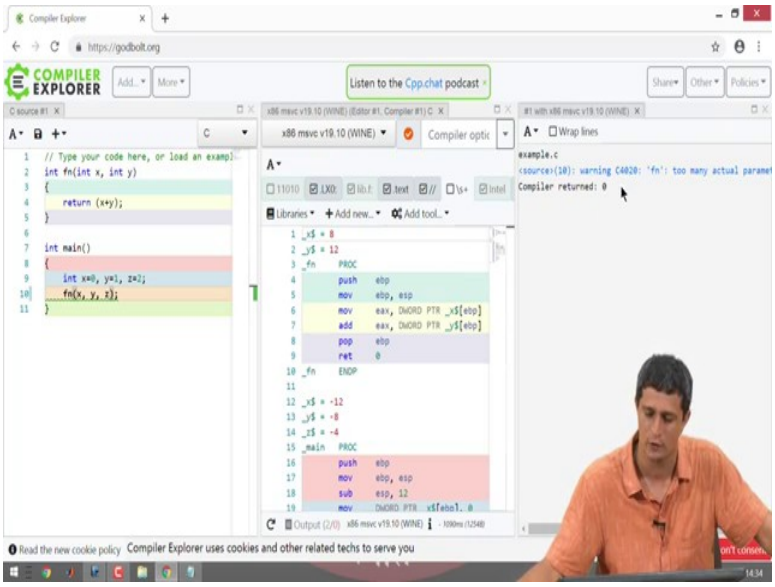
And so, it turns out that GCC gives an error too many arguments to fn s and GCC of course, if you call it with fewer arguments will also give you the error saying two few arguments to function fn . So, the point is whether it is an error or a warning is actually up to the implementation of the compiler. Now, if you pass fewer arguments then what has been defined, then all compilers will definitely give you an error , but if you pass more arguments than needed right like this x, y and z.

(Refer Slide Time: 06:05)



Then, it is the choice of whether to throw a warning or an error is compiler dependent . And, that is what we are going to analyse? And see what the impact of passing this extra arguments is at an assembly language level .

(Refer Slide Time: 06:21)



So, if you look at msvc we are back to msvc 19.10 version here. And, this only throws a warning if you pass more number of arguments to it. So, the question is now what happens, when I pass more number of arguments then what has been defined . So, let us look at our function Fn int Fn of int x comma int y . Now, we already know that the

arguments are passed pushed onto stack before calling a function. So, therefore, both x and y will be accessed as contents of EBP plus something at EBP plus 8 .

So, x will essentially be contents of EBP plus 8 and y will be contents of EBP plus 12 if it is a C program , if it is a C plus plus program then the offset will be slightly more as discussed earlier. Now, in my main I am going to call this function in two different ways. So, the first way is case one I am going to only call it with x , I am saying int x, y and z . So, how does this particular function call get translated to assembly. So, you say push x which is EBP contents of EBP minus 4 and then you are going to say call Fn.

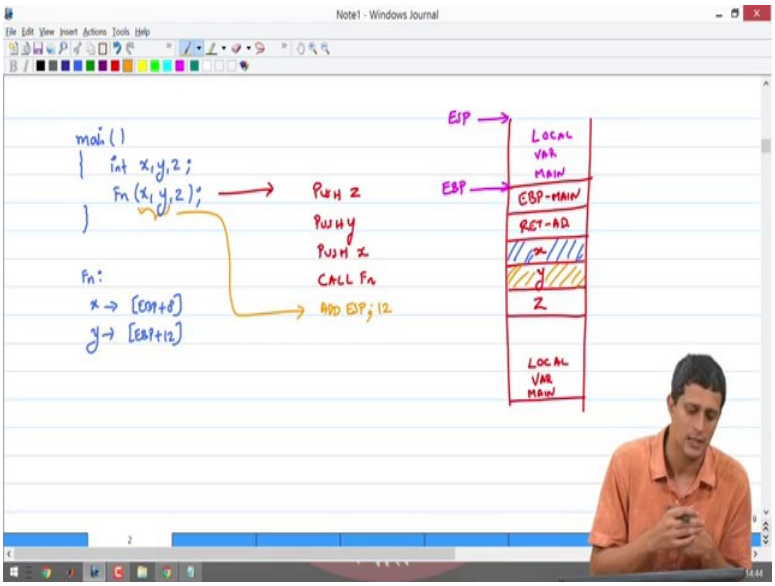
So, if you look at this particular stack what happens is you have the local variables of main . This is local variables of main . Then you are pushing EBP minus 4 onto stack contents of EBP minus 4, which is basically the local variable x and then are calling the function. So, what happens is there is a return address that gets pushed onto stack . And then the you enter the function Fn where as usual you push EBP to stack and you know all those prologue happens appropriately .

So, they are this is EBP of main . And then this is my local variable space of Fn. Now, if you look at this particular and by the way this is the stack. So, if you look at this particular implementation and let us assume that we are able to translate this program to assembly language without any compiler error . Then, what is going to happen is in the function Fn you are going to access these two locations EBP plus 8, which is basically this guy right. EBP plus 8 because when I enter the function Fn remember that my EBP will be pointing here .

Then in the prologue we would we would have move the EBP to 0.2 ESP before allocating local variable space . So, my EBP is here and therefore, EBP plus 8 is basically this location . Now, EBP plus 12 is also going to get accessed in this function, because remember that the function that we implemented out here is return x plus y . So, both x and y will get accessed and y is nothing, but EBP plus 12.

So, what is happening is EBP plus 12 is this particular location,, which is some garbage value because we have not really even initialized it . So, some random value from main is getting accessed in this function Fn, if you pass fewer arguments than what has been defined in the function Fn . Now, on the other hand let us look at the second case .

(Refer Slide Time: 11:43)



Now, let us draw the stack picture for the second case where in main, we are now passing Fn of x comma y comma z and this is int x comma y comma z . So, how does this get translated to assembly language it is push z let us not worry about EBP plus what you know for the local variable here push y, push x and then call function. So, let us now redraw the stack picture out here as well . So, again when I have the local variable space . And so, what do I do I first push z onto stack , then I push y onto stack and then I push x onto stack .

And then of course, I am going to do a call so, the return address gets pushed here. And, once I enter the function Fn the EBP of main will also get stored here . So, after all this EBP will point here . And this is my ESP local variable space for main has been allocated, local variable for main. So, now, let us look at what happens in the function Fn .

So, the function Fn is simply going to access only 2 locations . In Fn x is contents of EBP plus 8 and y is contents of EBP plus 12 . So, EBP plus 8 is this particular location . And EBP plus 12 is this particular location. So, what has happened is we have pushed an additional variables z onto the stack, but that is really not being accessed in the function Fn.

So, therefore, as far as Fn is considered there are 2 variable that have been passed to it appropriately of course, it takes only the first 2 variables x and y when it is been called .

So, here it is only these 2 variables that it will eventually take z will just get ignored even though it has been pushed onto the stack .

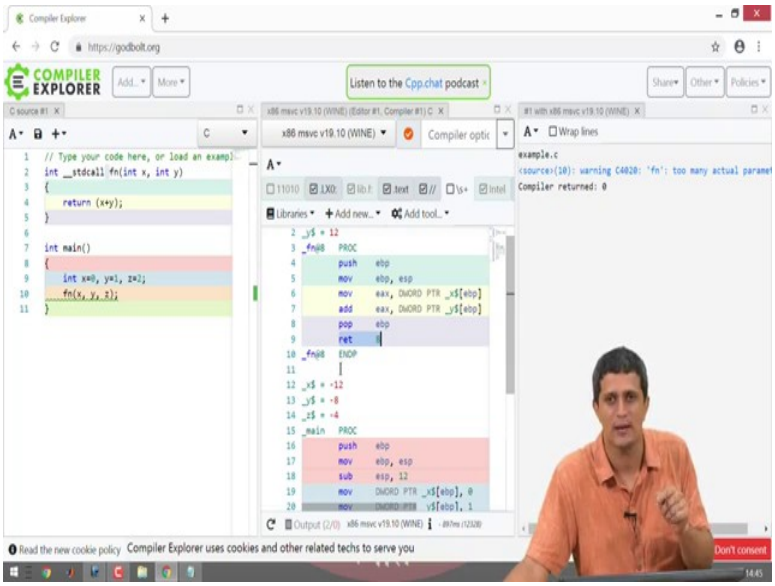
So, now, what happens is after the function Fn is done , when we return and ; come back to this function here, we do a stack cleanup . And if the caller is doing the stack cleanup , as is the case by default with underscore underscore C decl calling convention. Then what will happen is this function call Fn will get translated to ADD ESP comma 12. Whereas, if I did Fn of x alone which is basically the previous case here, if I did this then this would get translated to add ESP comma 4.

Now, in both cases whether I passed fewer arguments or more number of arguments stack cleanup is not a problem as long as the stack cleanup is happening in the collars domain . So, which is the case by default and hence there is no problem, but in the case when we pass fewer arguments then what is been defined, some garbage value gets accessed.

And therefore, you will see that all compilers return an error and do not even allow you to compile the program if you pass fewer arguments, but if you pass more number of arguments there is nothing functionally wrong, the extra argument gets ignored right. And therefore, nothing really happens as far as the function execution is considered.

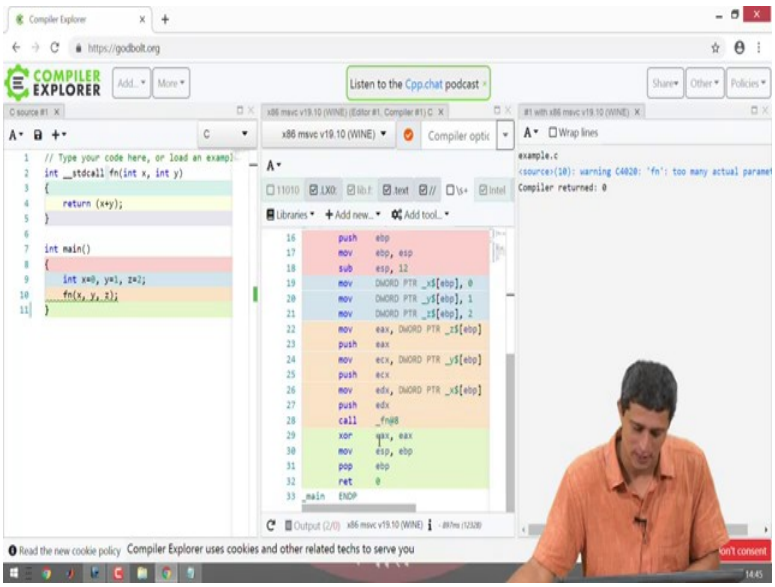
And therefore, some compilers may choose to just give you a warning and not worry about calling out the error at that point . So, at this point it is interesting to ask saying ok. So, is this always true you mean whereas where if I pass more number of function parameters to the function, then what is defined? Is it always true then the program can get executed correctly while ignoring the extra parameters that have been passed, well the answer is not you know is not it is not always true and I can show you where that will fail.

(Refer Slide Time: 17:13)



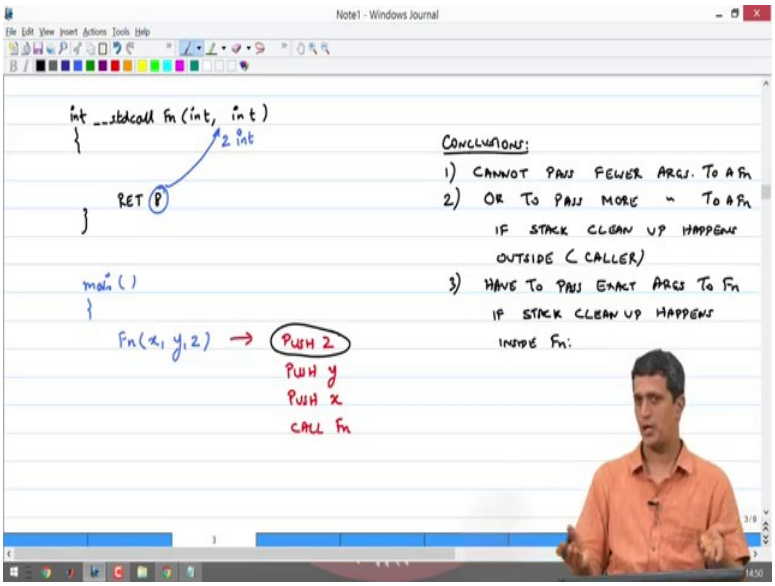
If for example, this is an underscore, underscore std call , fn of int x comma int y. And then I pass 3 variables to it x y and z. Then, you see that this particular function fn gets translated to these argument, which is push EBP you know the usual stuff and ultimately there is ret 8 which happens. Why is this because this is a standard call where the arguments are fixed ? And the stack cleanup is happening within the function fn .

(Refer Slide Time: 17:57)



So, if I now come here basically what has happened is I have 3 pushes happening on to stack , push z push y push x and then a function call to Fn is happening .

(Refer Slide Time: 18:21)



So, what happens is suppose Fn was an underscore underscore sorry std call of int comma int . Then, the function would get translated to assembly and eventually it would finish with RET 8. Why RET 8, this is tied to the fact that there are 2 integers, as arguments . Now, here the function itself is doing the calling the stack cleanup . And therefore, if this function gets called with either lesser or more number of arguments, the compiler should flag an error necessarily, because the functionality will be in trouble when we execute the program here .

So, here for example, if I had a main and I called this function as Fn of x comma, y comma, z. Then, this would get translated to push z, push y, push x and call Fn. The add ESP comma 12 will not happen here, because this is an underscore underscore std call and the stack cleanup is happening through RET 8. So, therefore, in this case the extra push that was done this particular push this start cleanup will not happen.

So, ESP instead of coming down by 12 after the function call Fn, it will come down only by 8. And therefore, one push would have not been accounted for when the exit main. And therefore, it will return to some random address after it completes main . So, with that we can write down some important conclusions that we have derived from this particular lecture .

So, cannot pass fewer ARGS to a function right. I will say it is to pass more args to a function if stack cleanup happens, basically at the caller , but of course, if it is a std call

then stack cleanup is happening within the callee, then you cannot; pass more number of arguments you have to pass exact number of arguments, exact args to function if stack clean up happens inside Fn.

So, this conclusion that we have come to is very important when we go ahead and look at the implementation of something like print f , where we have variable argument list, we need to make sure that it is possible to pass more number of arguments than, what has been defined . And yet have this program execute appropriately. And in the next lecture we will look at the implementation of print f in detail.

(Refer Slide Time: 23:09)