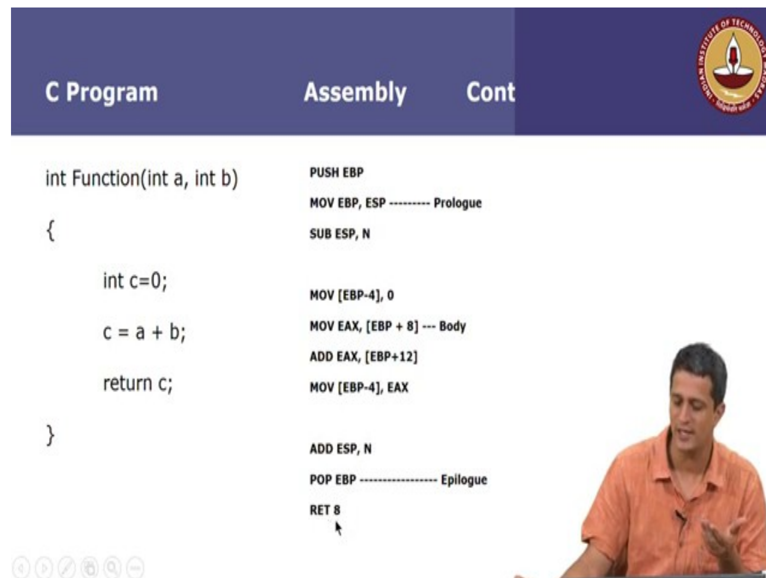
C Programming and Assembly language
Prof. Janakiraman Viraraghavan
Department of Electrical Engineering
Indian Institute of Technology, Madras

Lecture - 15

(Refer Slide Time: 00:11)

So, welcome back to this course on C Programming and Assembly language. We are in module 3. In the last couple of lectures, we have discussed how to translate a given C program into its assembly output .

(Refer Slide Time: 00:24)



In that regard we looked at a typical C function and we said that apart from the body of the instruction, which is shown in the middle here, you needed to add a couple of instructions in order to set up the context called the prologue and which basically was you know pushing EBP, moving EBP to the correct context of the called function, subtracting ESP comma N, which is basically allocating local variable space.

Then before you exit the function, you need to undo the exact same operations that we did in the prologue, which is adding ESP comma N and bringing ESP back to its original value. Then moving EBP back to its original context before that it had before it entered this function . And RET N is was needed, so that apart from returning and going back to the called calling function, we have to we had to clean the stack and undo the effect of pushing the parameters.

(Refer Slide Time: 01:29)

Calling conventions

➢ __cdecl
» Default calling convention of C functions
» Needed for variable argument list
» Caller cleans the stack - ADD ESP, N instruction

➢ __stdcall
» Faster than the __cdecl call.
» Callee cleans the stack - RET N instruction

Contd......

In this regard we had discussed two different calling conventions, the underscore underscore stdcall is the standard call which is a faster call than the other calling convention and this is where the RET N instruction is executed. The necessary condition is the function knows exactly, how many parameters and what the parameter sizes are.

For functions like printf, where the function parameters could be variable and is determined only at runtime, then it is the calling function that knows how many parameters are being passed on to stack or being pushed on to stack. And therefore, you can undo this effect only in the calling function and not within the function itself. And that undoing is implemented using the ADD ESP comma N instruction.

(Refer Slide Time: 02:23)

So, in this lecture what I want to look at is I want to look at the C plus plus functions. So, C plus plus is a very powerful programming language and really its quite similar to C in syntax and other things, but it offers a very powerful programming concept called object oriented programming , so that is a programming paradigm by itself and we are not going into any of those details, .

What I want to discuss in this lecture is at an assembly level, assembly implementation level. If I compile a C plus plus program, do I need to add any extra instruction to my prologue or epilogue that is the only thing that I want to focus on in this lecture, . So, let us take a typical example which is basically class you know, cls test some test ; I am going to say it has two member variables int a comma b ok. And then I am going to say public let me do, int add of int add and I am going to do return a plus b , this is my class definition.

So, clearly there is one member function here right and these are member variables. So, if I look at a typical C function , how different is it from this. So, the primary difference is that I am not passing any function parameters to this function add , but I am still able to access the member variables a and b in this function. So, remember that this public, private and protected and this inheritance and all these concepts are verified and checked by the compiler or the pre-compiler, .

None of these actually has an effect at an assembly language level. So, does not mean that, because this is a public function suppose instead of a public suppose this was private , does not mean that this function cannot be accessed at assembly level, . All these checks are done at compile time, but at an assembly level there is no check that is enforced to make sure that you do not access this function there.

So, let us now look at the main implementation where I use an object of this class, I am going to say main and I will say c l s test x . And I am going to do x dot add and maybe I can do int z equals 0 and I will do z equal to x dot add . Let us assume that the constructor of this class has initialized a and b to some reasonable values , so that we do not get some junk when we do this addition operation .

So, now the key point is that when I do x dot add , x is an object that is sitting in the stack of main. So, if you look at the picture of the stack here, I would have my stack sitting like this and I have the object x, which basically has two variables a and b, this together is my object x . Now, I am trying to call the function add .

So, if I look at the assembly translation of this particular function, then it will happen exactly like how a normal C program would have been translated. For example, this does not take any it does not take any parameters therefore; it will get translated simply to call ADD . And of course, this function out here would get translated exactly in the same way that we had discussed earlier ; where we do a push EBP and we do a MOV EBP comma ESP and we do subtract ESP comma 0 X 4 0, some random number that will ensure that we have enough local variable space, .

And then we go ahead and implement the body of this function , which is basically you know whatever the instructions that are needed to do add a plus b , so this is instruction for a plus b . And then I do my epilogue, the question is because this is a C plus plus program and there is an object x, what I need to do now is to access the local variable x which is sitting in main, when I am actually executing the function ADD .

So, if you look at this when you do the call of this function ADD , then obviously the return address gets let me put that in blue gets pushed on to stack , this is return address . And after that you know I will go ahead push my EBP on stack of MAIN right. Then I am going to move my EBP to this point, . And my ESP then will get moved somewhere up here. This is how my stack will look, when I call the function ADD.

Now, all I have to do is to ensure that when I am accessing the variables a and b, I do not access them locally in my local variable space out here , this is my local variable space of F n of ADD. I need to access the original object which is sitting in main, so somehow I have to get access to this particular address and that is available through something known as the this pointer .

So, the C plus plus you have something called the this, what is the this pointer; so in the function add , actually these variables here will get translated to this arrow a plus this arrow b , which is nothing but the pointer to the object x of itself and that is why it is called the this pointer and then you refer to the variables a and b. So, therefore, when I am dealing with a C plus plus program, I need to additionally pass the this pointer whenever a function F n is called . A member function F n or a member function add is called, I need to pass the this pointer somehow.

So, now the question is how do I pass the this pointer, well. One way is you could maybe push it onto stack or you could pass it through a register , because unlike function parameters that this pointer is just 1 in number, there is only 1 this pointer that I need to pass to every function. So, therefore I can actually pass this through a register and ECX is the register that is assigned for this job.

So, this pointer is passed ECX. So, what happens is before I actually call this function with call ADD , what you would do is you would get the address of x and load it into ECX. So, let me move that here, so this would now you would MOV ECX comma address of x , which by the way is just you know some EBP minus 4 or EBP minus 8; where EBP is the value in main and not the value after going into the function ADD .

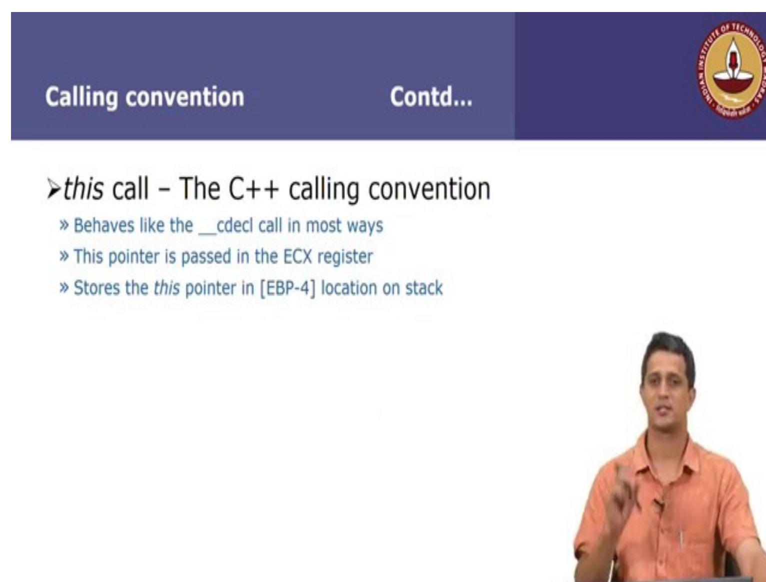
So, you are going to just pass this address of x into ECX and call the function ADD. Now, when I enter the function ADD, I might want to use the register ECX for some operations. Like for example, the string related operations or the c m p s b or something I might want to use ECX to do these special operations and therefore, I have to free that register ECX. And therefore, an extra operation that needs to be added to the prologue is to PUSH ECX onto stack, before you start anything in this function .

So, ECX comes in with the this pointer that is pushed onto stack, then you do the remaining prologue that has to be done for any C function and then proceed. So, what happens is now apart from EBP, I am also pushing my this pointer on to stack before I

start my program. And therefore, function parameters will now get addressed not as EBP plus 8 and EBP plus 12; it will start directly with EBP plus 12 . So, what is happening here is in between I am pushing my ECX, which is this pointer.

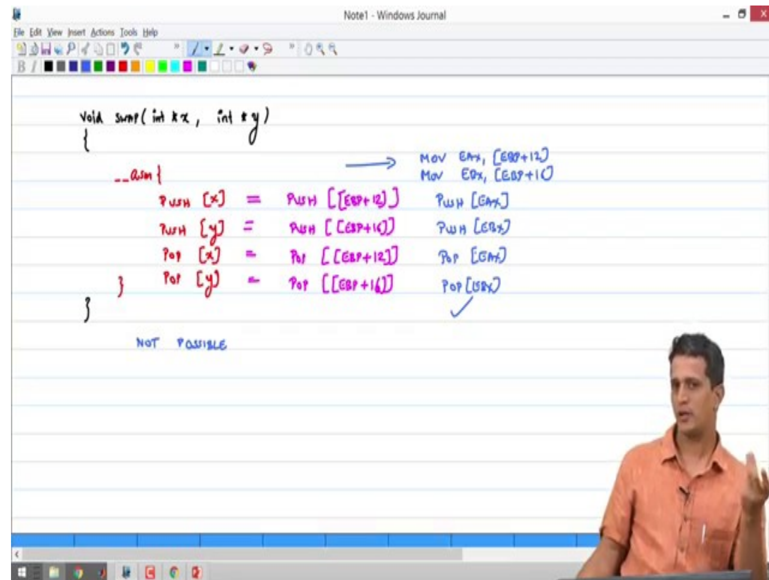
And therefore, my EBP which is pointing here, can now refer to function parameters only 3 into 4 bytes away. And therefore, the function parameters will now start as EBP plus 12 and EBP plus 16 and so on , so that is the only change at an assembly level when we deal with a C plus plus function as opposed to a C function.

(Refer Slide Time: 15:46)



And in fact, that is known as a this call. So, if you deal with g plus plus instead of gcc, then every function will get translated to a this call and it behaves exactly like the cdecl call, . What does behaving like a cdecl call mean, it means that the stack cleanup happens outside right using the ADD ESP comma N instruction . The this pointer is passed in the ECX register and it stores the this pointer in contents of EBP minus 4, which means the first local variable in the function is actually the this pointer, which is on stack.

(Refer Slide Time: 16:32)



So, let us now go back to our original question that we started off with in module – 2 , we had a function void swap of int star x comma int star y . And the question we asked was cant we do the following . We want to swap, so can we do this PUSH contents of x, PUSH contents of y, POP contents of x and POP contents of y.

The question was could we have done this and we said, it was not possible so now we will explain, why that was not possible . So, this clearly was not possible . So, instead what we said was we had to add two other instructions, we will come to it . So, now let us now that we know how the function parameters and local variables are translated to assembly language. Let us look at what this actually means at an assembly language level, x and y are function parameters to this function swap.

And therefore, this actually is PUSH contents of remember x itself is contents of EBP plus you know maybe which is a this call it will be EBP plus 12 , this will be PUSH contents of EBP plus 16, this will be POP EBP plus 12 and POP EBP plus 16. So, why is this not possible, the answer is very evident from the assembly instructions that we have here; we do not have an ability to do a double in direction of address. What we are saying is get the contents pointed to by EBP plus 12 and then use that as an address and push that onto stack. So, this is a double indirection which is not possible.

And therefore, we had to correct this by adding two instructions which was MOV x MOV into EAX you know the value of x, MOV EBX the value of y and then we said we

will do PUSH contents of EAX , PUSH contents of EBX, POP contents of EAX and POP contents of EBX. And this was indeed possible or is indeed possible, because we have moved x and y into registers .

And of course, I can in fact write this as EBP plus 12 and this as EBP plus 16 . So, we have translated this particular swapping operation using just a single indirect addressing mode as opposed to the double indirect addressing mode that would have been translated to if we had done push contents of x and contents of y, as shown on the left hand side here.

So, with that we conclude module 3 and in the next module we will look at some optimized implementations of C programs. We will also look at why recursion is not a great idea from a assembly point of view and from a performance point of view. And we will also look at a few other special functions to wind up this course.

(Refer Slide Time: 21:19)

Topics Covered

- Compiling a C++ member function
  - Passing the "this" pointer
  - Prologue and Epilogue change
- Back to our swap function