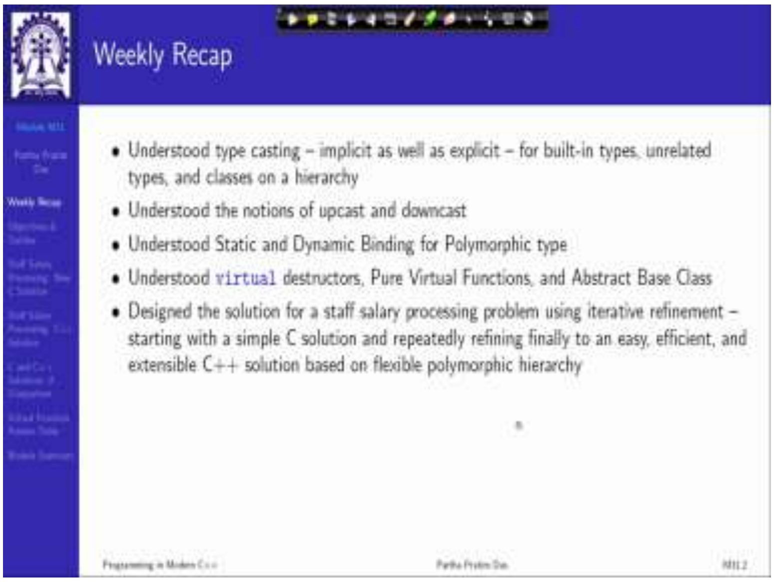
Programming in Modern C++
Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 31
Virtual Function Table

(Refer Slide Time: 0:34)

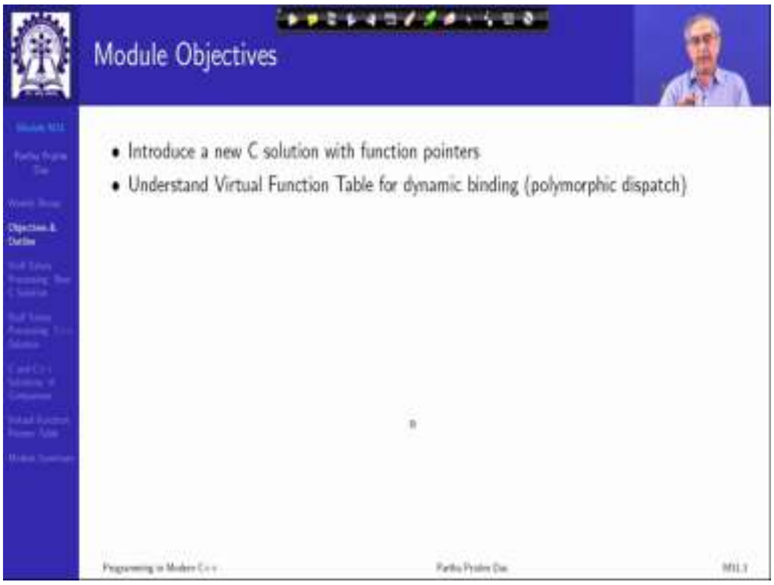


Welcome to Programming in Modern C++. We are in Week 7 and we are going to discuss module 31. In the last week, we started with understanding the typecasting - implicit as well as explicit casting - for built-in types, unrelated types, and we primarily discussed it in the context of C casting style.

And in the process, we understood the basic notion of upcast and downcast in a C++ hierarchy. And then we moved on to discuss about different kinds of binding, static or compiled time and dynamic and runtime binding for polymorphic type. Introduced virtual destructors, talked about pure virtual function, abstract base classes.

And in the last 2 modules of last week, we have done an extensive treatment of a salary processing problem to show how starting from a very simple problem statement and a flat C solution, how we can iteratively refine to get to a very good, flexible polymorphic hierarchical solution. So, it was all primarily about polymorphism.

(Refer Slide Time: 1:52)

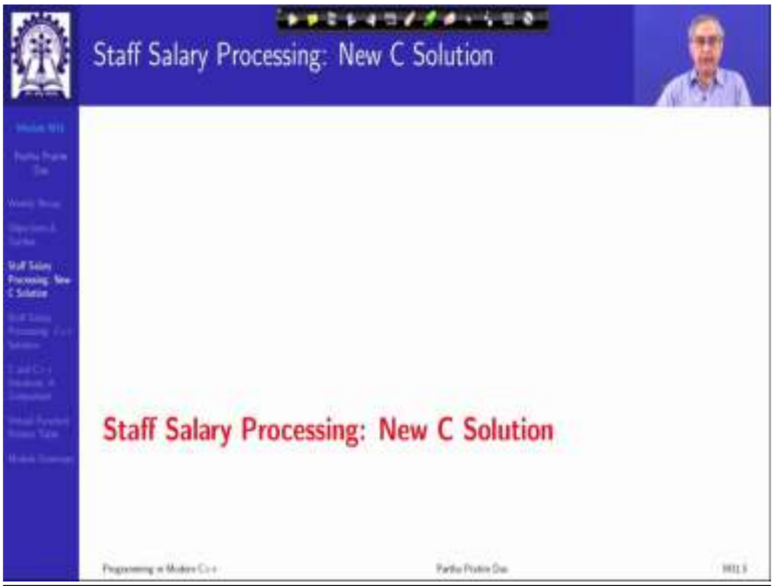


So, moving on, we will hover around that itself but go into more specificities in this week. Talking first, in this module about a new C solution. You will be a little surprised that again, having done all sorts of C++ solution, I am coming back to introducing another C solution. But this C solution will be unique in the fact that it will use function pointers. All previous, the previous C solution had used function switch basically. And then we will understand the virtual function table for dynamic binding which is the backbone of polymorphic dispatch, the backbone of polymorphism in a C++ polymorphic hierarchy.

(Refer Slide Time: 2:38)

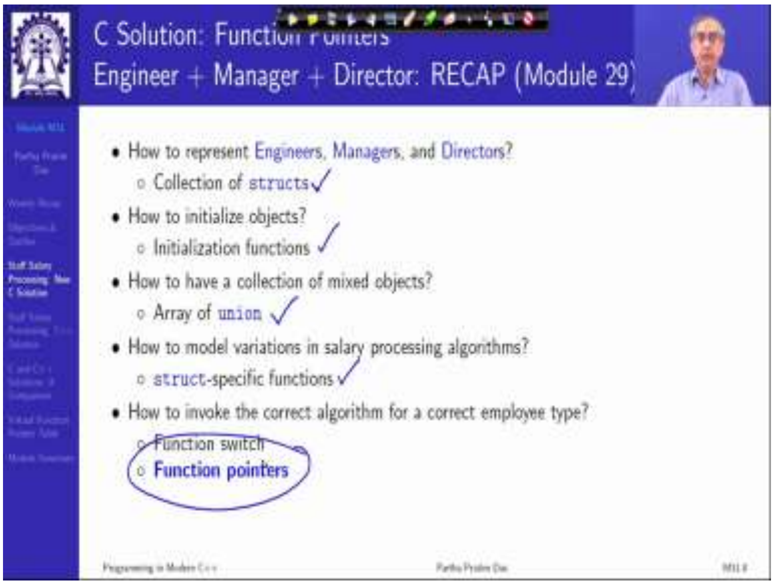
This is the outline which will be there with you all the time on the left panel.

(Refer Slide Time: 02:48)



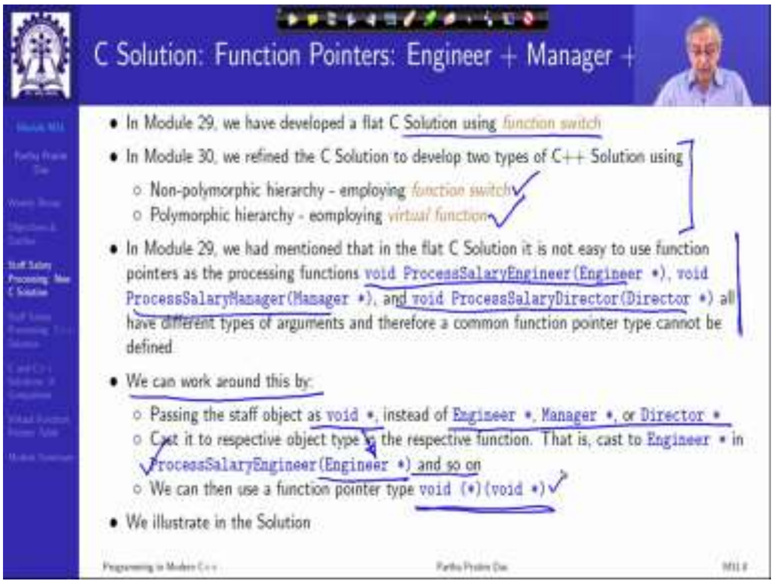
So, let me just start with as I said, the staff salary processing. Just a quick recap and introduction to a new solution. So, we have talked about an organization which needs salary processing. There are engineers, managers in the engineering division. Then we have added directors to that. We want to keep provision for adding new divisions in future and so on. The salary processing logic for each type of employee is different. So, we want to model that, model these employees and the salary processing. And to make sure that when given an aggregate of employees, we are able to process their salaries through a suitable and extensible design.

(Refer Slide Time: 3:33)



Now, these are the basic questions that we started asking with and nothing much changes here. The modelling still continues to be in C it is with structures. Initialization function, array of union, structures specific functions. But what we are going to see differently here is instead of using functions which we will now see that use of function pointers.

(Refer Slide Time: 4:07)



So, let us understand what we have done so far. If you refer to Module 29, then we developed a C solution using functions switch, separate set of structured types and functions switch. Then we had a number of C++ solutions. But primarily of two categories, one that was on the non-polymorphic hierarchy which gave the advantages of encapsulation and inheritance to refactor the common data members and so on. But still that dispatch was in terms of a functions switch pretty much like C.

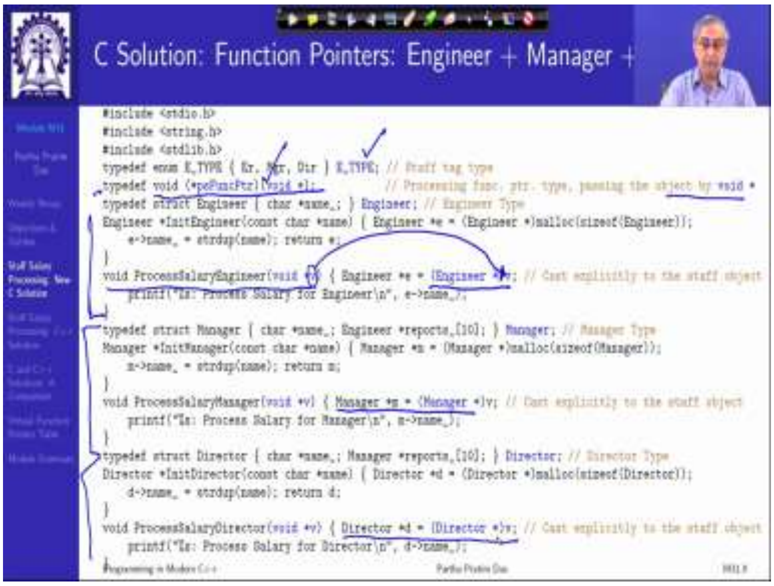
So, we moved on to a polymorphic hierarchy and we employed virtual functions. Then we did further refinements on that: used vectors and all that. That is not very relevant. Now, if you think back in terms of module 29, the C solution then what we made is since we need a different kind of processing, we assume that for every type of employee, there is a separate processing function which has a lot of similarity. In that it takes a pointer to the employee type record as an attribute, as an argument. Processes the salary, does you know whatever database updates, prints and so on and returns nothing.

Now, we had observed and you will recall that in that discussion that it is we had been mentioning that it is not easy to do a function pointer-based solution in C particularly because all these functions have different types of parameters. One is engineer star, one is manager star, one is director star. So, how do you combine them into one function point at time?

So, right now we will show that this can be worked around by doing some little trick that we can assume that this processing function actually takes a void*. This is where you see the use of void* particularly in C in terms of combining effects of multiple types instead of the specific Employee type pointers. And then, if we take it as a void*, then say I am in processing salary for Engineer, then I have got a void* pointed but I know that I am processing salary for Engineer. So, I can take this void* and explicitly cast that pointer to an Engineer* pointer.

Because I am assuming that processing function Engineer will never be called with the pointer to any other staff type. Like it will never be called with a pointer to a Manager structure passed as a void*. That trust is what I will have to assume. And given that I can assume that the processing function simply takes a void* and returns nothing. So, the processing function can generically have an interface which is void* means the function pointer. And void* takes a void* parameter. So, if we assume that, then let we can easily put together the solution here.

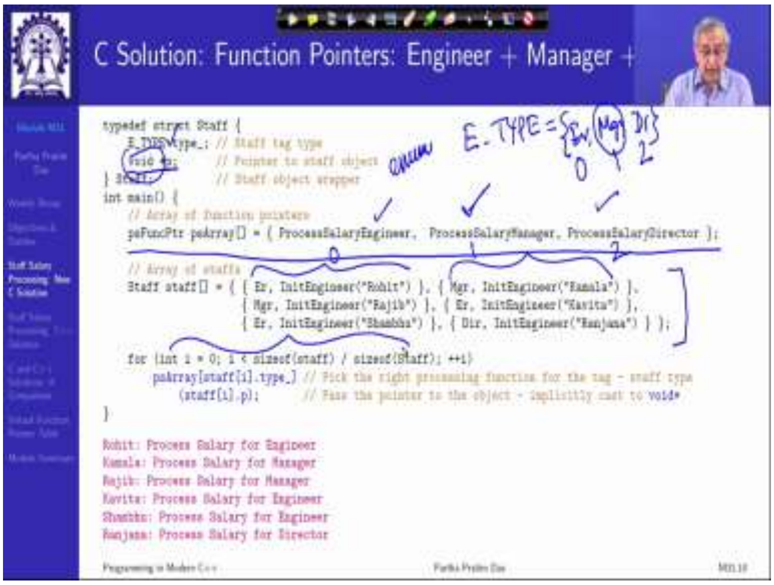
(Refer Slide Time: 7:19)

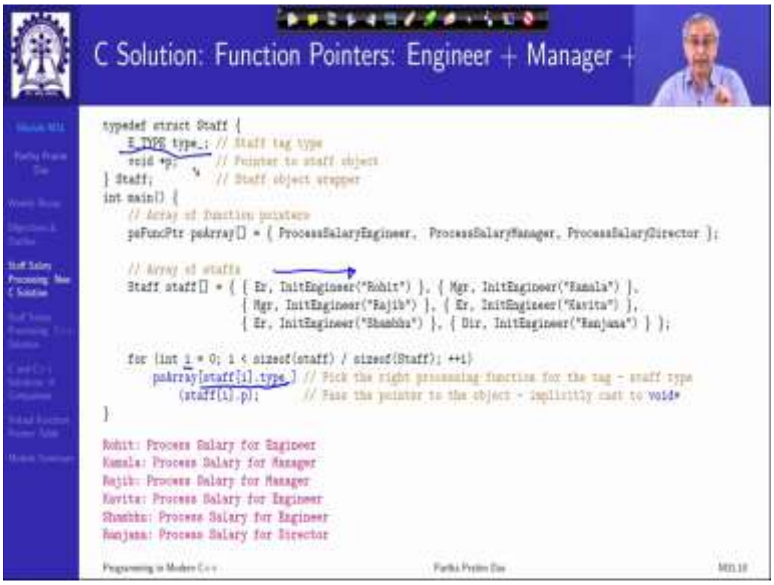


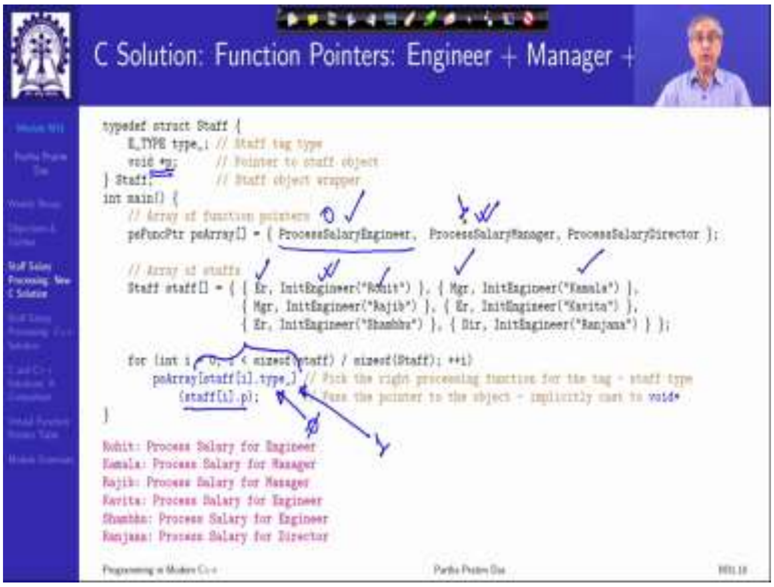
So, here is the tag type which we had earlier. This is where I am defining the function pointer for the processing type, psFuncPtr pointer pointed to this function pointer which is processing functions which takes void*. And the objects are passed by void*. Then we have specific types for Engineer, for Manager and for Director. The only point to note is when I come to processing for Engineer, now my parameter type is not Engineer*, it is void* because I want to unify all of them in this function pointer type. So, what I get as v is a void, is a pointer to a I do not know which type.

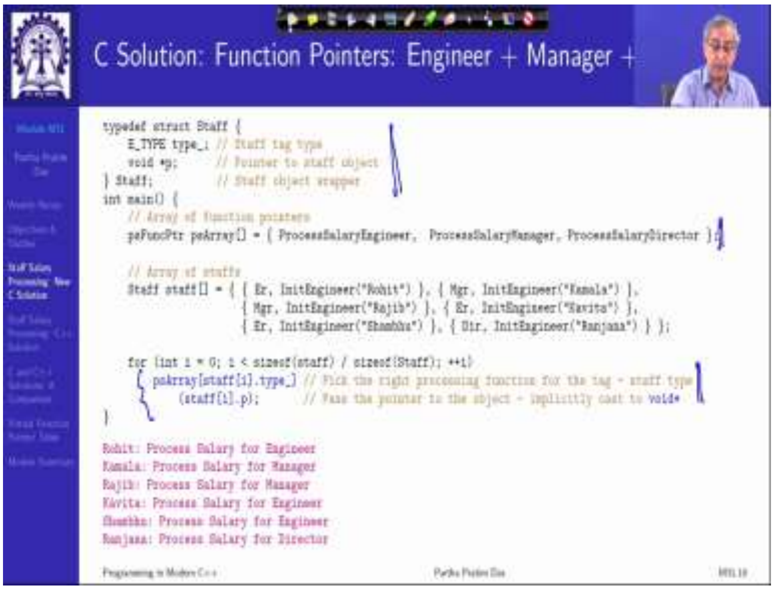
But I know if this function has been called, I trust that it must be for an engineer record. So, I cast it to Engineer*. And from that point onward, rest of the logic will follow as before. Similarly, I do it for manager. I do it for director. So, this is the only trick I do to unify all of this processing function in terms of a single function pointer type.

(Refer Slide Time: 8:41)









Now, once I have got that, then I can write my rewrite my unifying structure. What did we have earlier? Earlier we had union of different pointer types and we had the tag type to identify which pointer we are using instead. Now, I just use the void* pointer. Because once I know this type, I do not need to really know the type of the pointer to the particular Engineer or Manager or Director because the E_TYPE will tell me everything. So, I just take it a void*.

I create an array of function pointers. These are the array of processing functions. And here is an underlying trick which is what or I should say protocol that has to be maintained. If you remember enum. This enum was done as Er, Mgr, Dr, director like that. So, this basically in the internal representation, this is 0, 1, 2. So, I placed the corresponding processing functions at the corresponding index locations of the array. This is very, very important.

If I mess this up, things will not work. Because given any, say given a staff type Mgr, I would consider Mgr I would know Mgr is value 1. So, I would consider that whatever function point exists in this ps array is the function to process this particular M staff record. Then I have the staff array which is pretty much like before.

Only thing I have used a compact initialization notation to initialize it with pairs of values. One is the E_TYPE, other is a pointer to the Engineer, Manager or Director. But actually, since the type is void*, it will implicitly get cast to void star and will all be populated in the array of staff.

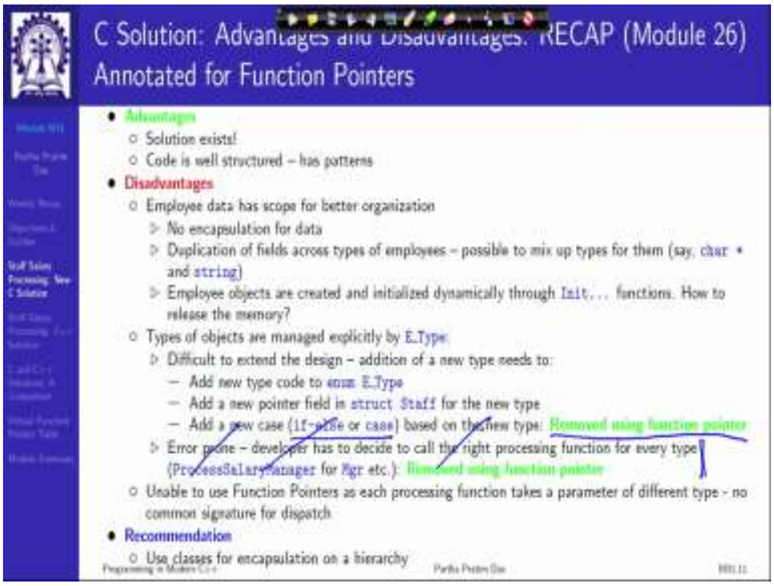
Now, this for loop does not change. So, what do I have to do? I have to go over this. Let me clear this out. So, I have to go over this array, pick up each and every staff, check what is the type and call the corresponding processing function. So, I go over this, that is my i. I pick up staff[i].type, staff[i].type which means for that particular ith staff what is the Employee type. So, I will get Er or Mgr or Dr given that. So, I get 0, 1 or 2. So, what I do is I am indexing the array with that tag type.

So, suppose the first one is Er. So, what I will get here for Rohit is a 0. So, I am indexing ps array with 0. So, I get this processing function which is a right processing function for the engineer. Now, what I will have to pass it? I will have to pass a pointer to the particular staff record that is appointed to this engineer Rohit which is this void* pointer, p.

So, I do staff[i].p. So, at that function pointer, I pass that staff[i] or staff[0].p which is Rohit's, pointed to Rohit's record which internally gets cast to the engineer because I have this engineer tag and the engineer processing function has been invoked and the processing is done. When I go to the next one, that for Kamala, my type is Mgr. So, my type value here is 1 which means that I will pick up this function which resides at 1. So, the manager's function will be called with pointed to Kamala's record and so on.

So, you can see that the biggest advantage here additionally we have got is not only we have unified this getting rid of the union. We have a nice array of function pointers. And this has become a single line code instead of all that switch I was having. So, this such an elegant solution is also possible in C and this is kind of the best solution you can have in C involving multiple types in this way.

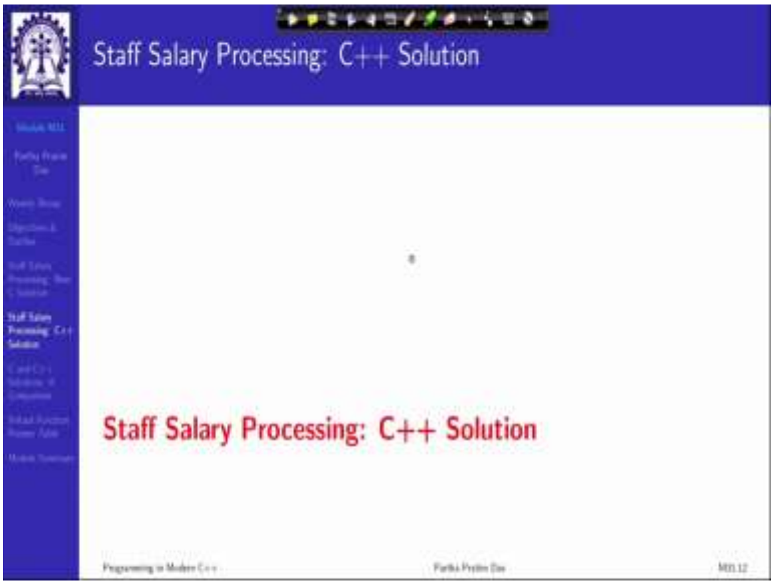
(Refer Slide Time: 13:18)

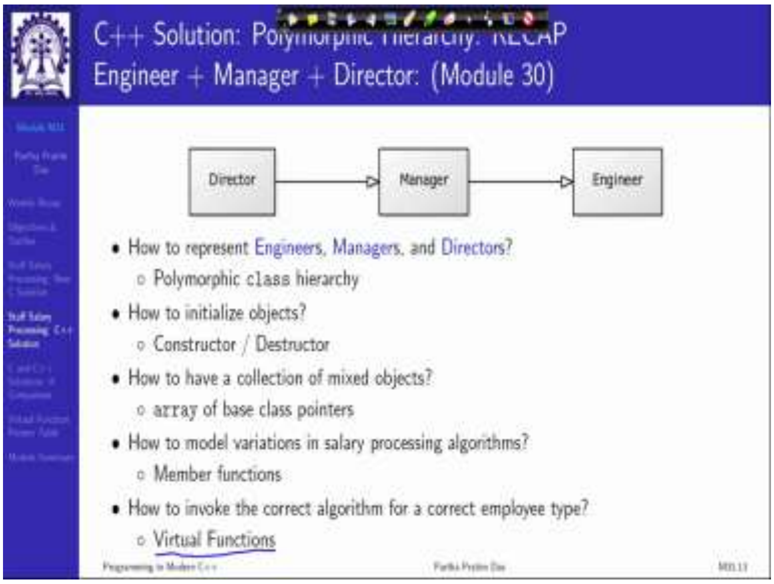


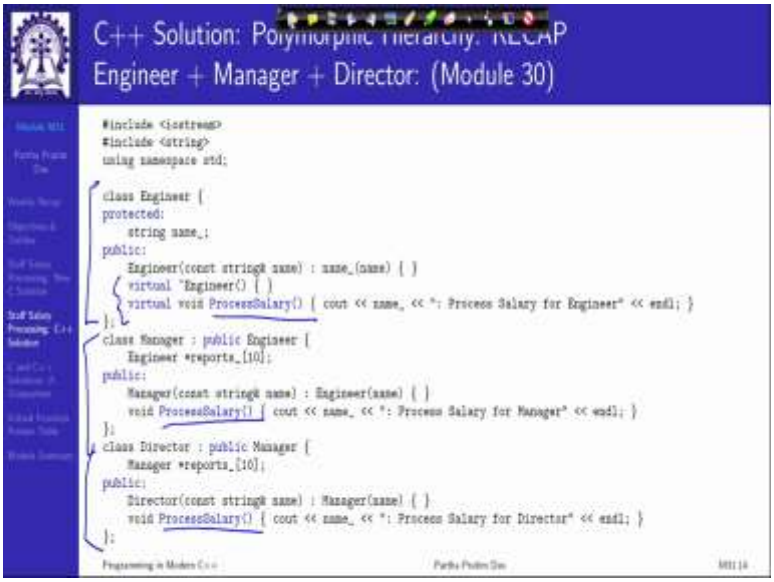
So, if you just look back in the original advantage-disadvantages, every disadvantage of the solution remains like not having encapsulation and so on. But two major disadvantages have been removed. One is the switch is gone. I do not need the switch anymore. That is happening automatically through the function pointer array. Right? So, this if-else, this switch is gone which is a great relief.

Second is there is no chance of calling a wrong function now. Because we have now the function pointer strongly bound to the type for which it has been created. So, these two disadvantages disappeared which is a big gain. But certainly, I still need to add a type. I still need to have, still do not have encapsulation. I still have repetition of field values and so on for which the C++ solution would be the ideal as we have seen.

(Refer Slide Time: 14:25)



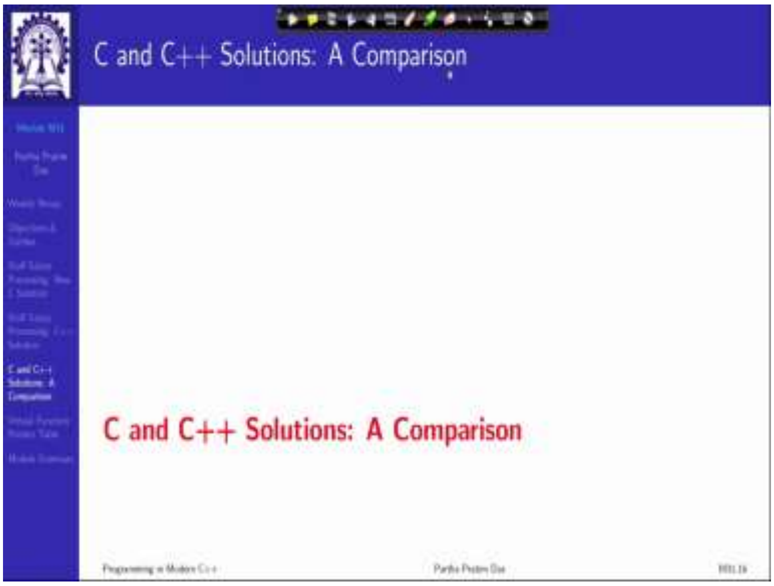


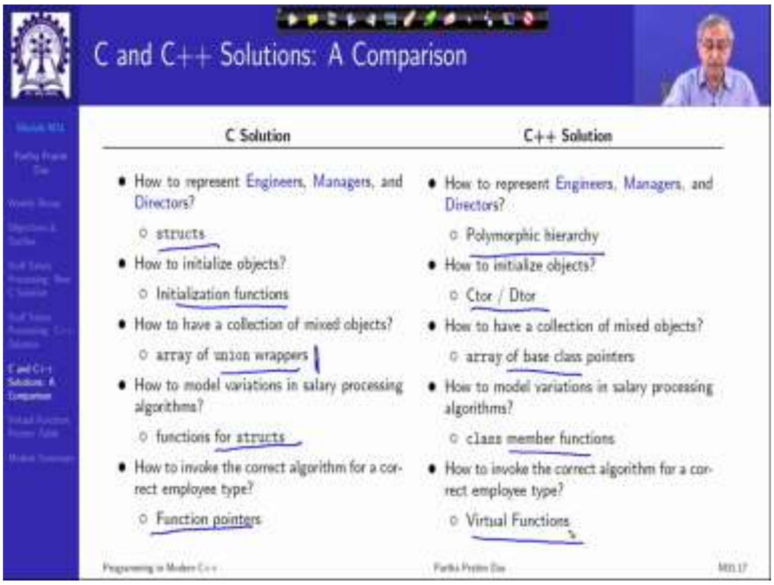


Now, if we look at the C++ solution that we did in relation to this, these are the basic points we had discussed in Module 30. And we said that for dispatch we will use virtual function. So, if we now look at the C++ solution, this is what we had. We had the Engineer class, the Manager class, the Director class coming is as a hierarchy and that is polymorphic. All of these processing functions are called ProcessSalary. So, which is the very simple way of depicting it.

And again, we have the array of employees. I am not considering the vector solution. And I have a single line called to this ProcessSalary function which by polymorphic dispatch will get me to the correct function of for the Engineers, for Rohit. For again Engineer for Kavita. No sorry, this m1 is Kamala. So, it will get to by Managers processing function for Kamala and so on. So, this is the C++ solution we saw.

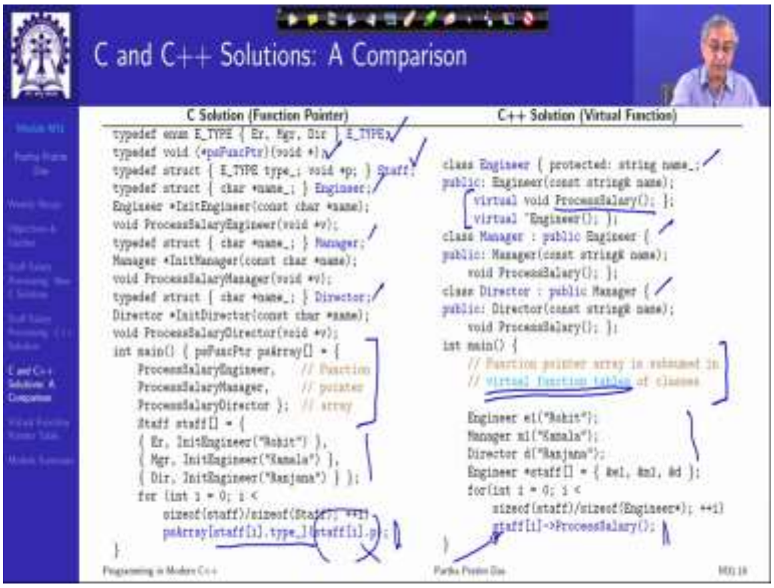
(Refer Slide Time: 15:43)





So, now, if we put them side by side to see, you know, how really they compare, how really they look like. This is in terms of their proposition. Struct versus polymorphic hierarchy. Initialization released functions versus constructor/destructor. Array of union wrappers versus base class pointers. We have partly been able to get rid of this. Functions for structure replaced by class member functions. And we have function pointers in parallel to virtual functions. Very similar in functionality. So, that is what is the focal point that we are, I am trying to get you to.

(Refer Slide Time: 16:22)

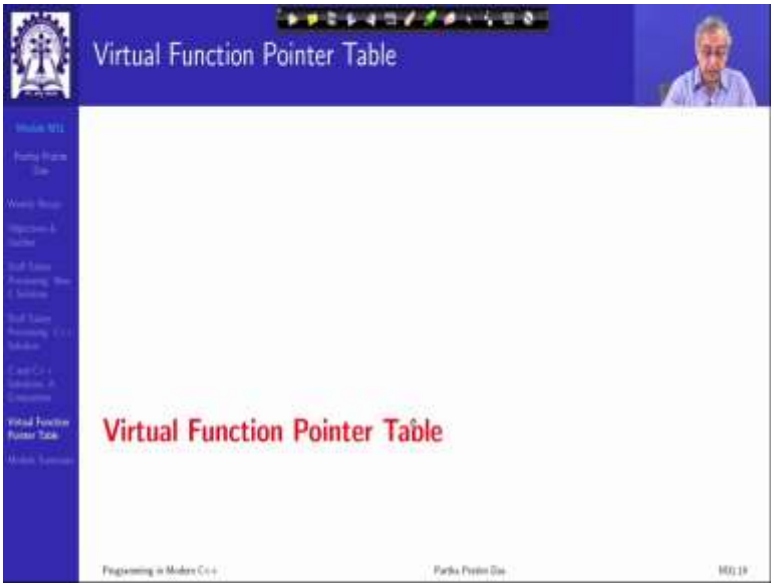


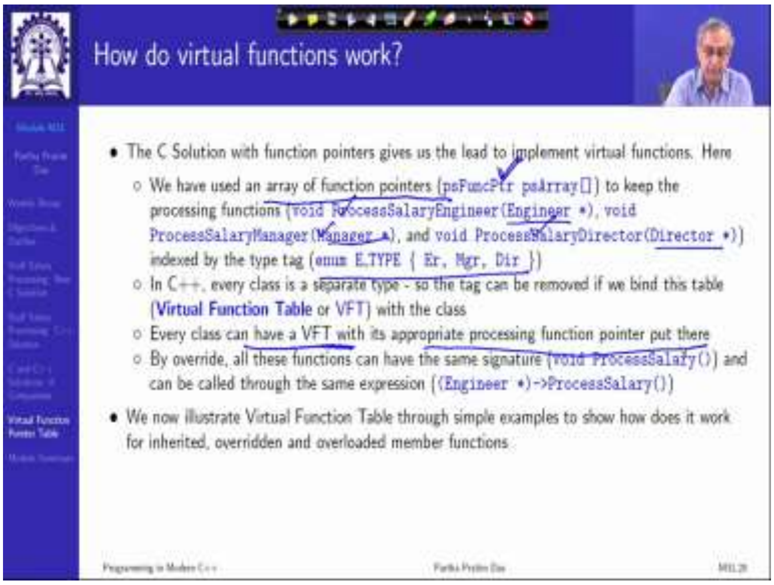
So, let us look at the solutions in parallel. Right? You have the Employee type. You have the function pointer type declaration. You have the Staff type which is a wrapper of the Employee type and the pointer to the particular staff. And you have Engineer, Managers structure records. Here you just have the classes. The class hierarchy which is polymorphic. Same processing function for all. We do not need the tag because every class is a different type.

In terms of the C solution using function pointers, we have this function pointer array. Here we do not have anything corresponding to that. And that is what is the key point that we will we are going to discuss here is, “How does that happen in C++?”. So, the function pointer array that we explicitly have to manage in C gets subsumed as a function pointer table which is also an array corresponding to the classes. That is the key point that I want to drive in. Rest of the solutions look the same except for the syntax and all that.

And the calls also look very similar. But C++ has the advantage that I am not having to maintain the explicit type. So, all that I need to do is basically, this is my object and on which I call the ProcessSalary function which automatically gets dispatched. So, having seen this parallel, now you will get to have an idea of how really C++ does this polymorphic dispatch or how really does it implement virtual functions.

(Refer Slide Time: 18:15)



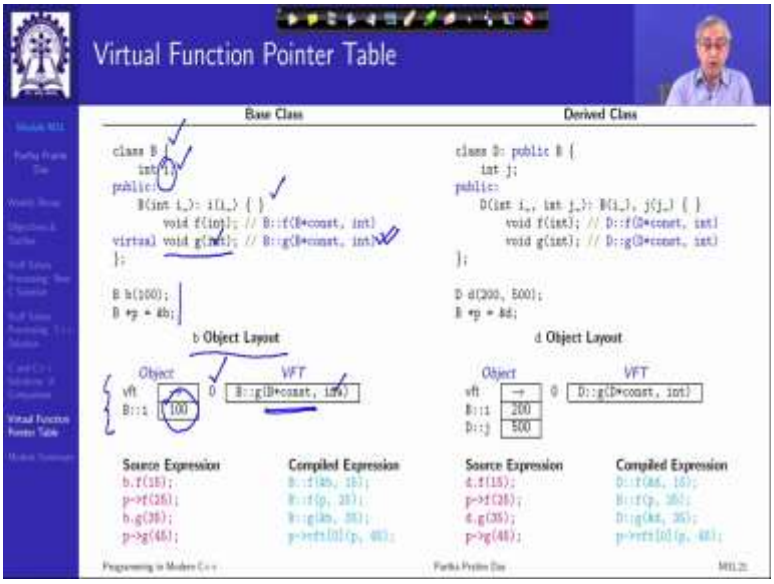


So, virtual functions are implemented in terms of virtual function pointer table. This is to note that if you observe the C solution very carefully, the function pointer-based solution, you get a complete insight into how to implement the virtual functions. What we have used here? First thing we have used is a array of function pointers which had in this case, this is the array which had 3 different function pointers.

We had simplified them by making these parameters void*. And they were indexed by an enum type. This was the backbone of the C solution with function pointers. In C++, every class is a separate type. So, I do not need that E_TYPE knowing the class itself because I am constructed the object. When the class itself is tells me what is the type of that object.

Now, somewhere I need this table. So, what where does this table go? The class can have a virtual function table which in which its appropriate processing function is put. So, it is not a common table where you are indexing and finding out which one I want. Rather we say that for every class, we have a different table. And every table keeps the processing function for that class. And those tables are called virtual function tables. Now, let us look at in the actual C++ context.

(Refer Slide Time: 20:04)

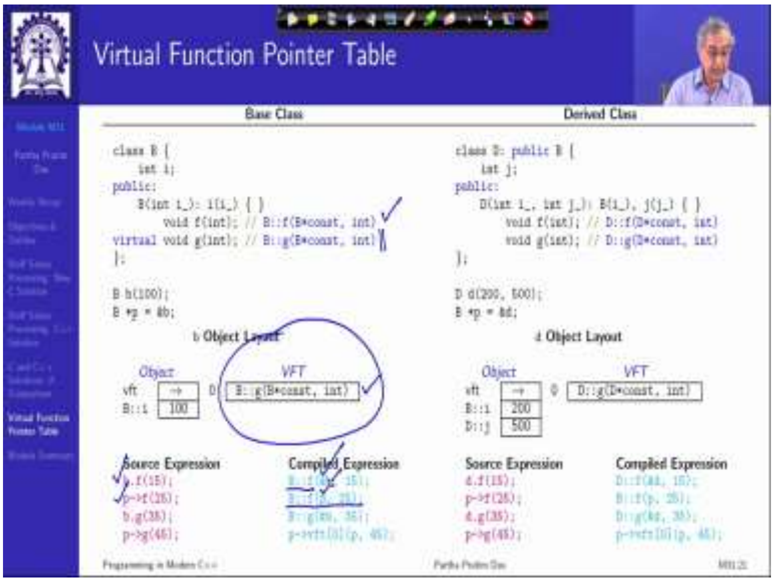


So, I have a base class with a member i. I have its constructor. I have a non-virtual function f and a virtual function g. I instantiate that and have a pointer to the base class which is initialized with the address. So, how will that object layout, the object layout b look like? Earlier we would have known that it will only have the data member 100.

But now, what it will have? It will have one more field which is a pointer to a; earlier it was having only the value 100. Now, it will have a additional field which is a pointer to a table. What does that table keep? The table keeps a function pointer which is basically a kind of an address.

So, it is a linear table, 1-dimensional. Since I have only one virtual function, there is only one entry at index 0. What is that function? That function is the only virtual function that I have defined here which is g. So, what is, what will be the signature for this function? As you know that every member function has an implicit this pointer. So, which is a pointer to the class. So, it will have B* const because this pointer is a constant pointer. And it takes a parameter int. So, it's second parameter is int.

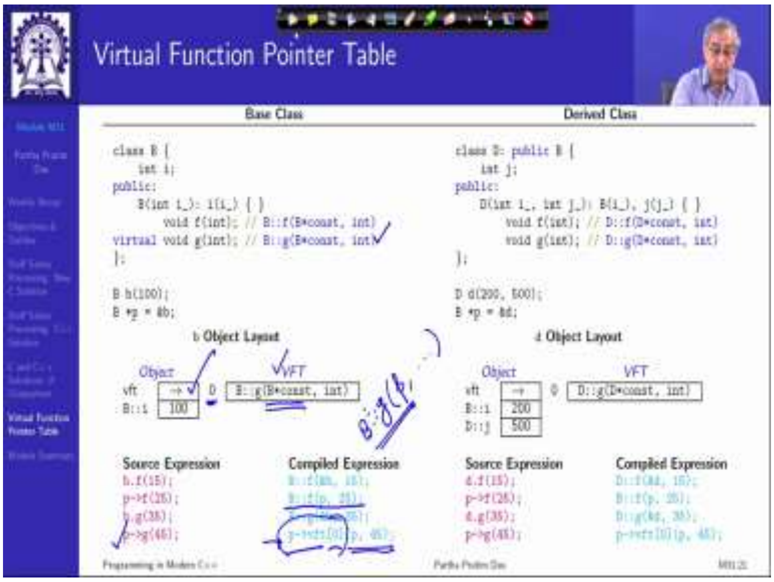
(Refer Slide Time: 21:50)



So, here I have in the virtual function table, I have this entry. And the point to note that this particular virtual function table entry does not vary with the instances of objects. Every instance of the object will have the same virtual function. Because there is nothing specific to the object that is here. It is all specific to the class and virtual function signature, name and signature that we have specified. So, this is the basic structure.

So, what happens if I look at different invocations? What if I do b.f()? It will always invoke the function in that class. So, b.f(), this is a non-virtual function anyway, will call B::f passing the address of B and then the parameter 15. What is p pointer f? f is non-virtual. So, it is static time binding. So, it uses the value of the pointer in the place of the this pointer.

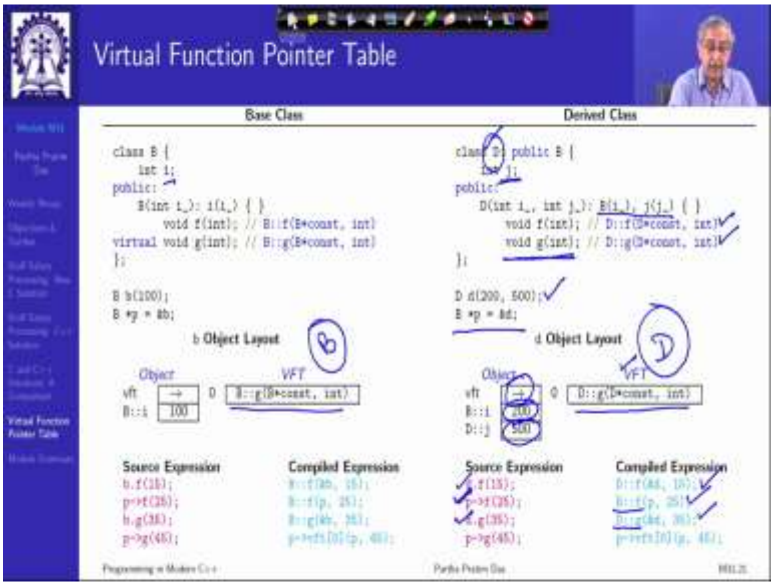
(Refer Slide Time: 22:55)



Now, what happens for g which is a virtual function? If I do b->g, I know it is a static type invocation, is a static binding. So, it will still invoke g with the address of b. Right? No difference. Up to this point there is no difference. But what happens if I do p->g? Then it will try to do a runtime binding, dynamic binding. How will it do? What it will do, it does not know what is object inside. So, what it does? It instead of doing like here, like here passing that is doing B::g passing p and so on. Instead of doing this, it actually does this.

It says it is pointing to this object. It goes to the virtual function pointer table. It goes to that pointer which is pointed VFT. It is this table. It knows this is the first or the 0th virtual function in the class. So, it picks up the index 0, index 0. So, what it gets? It gets a function. Now, it gets a function. And what does it pass to the function? The pointer. The pointer p. But what function it gets might change. In this case it will not because there is only one, one class. So, let us let us look at it on the right side where I have a derived class.

(Refer Slide Time: 24:34)

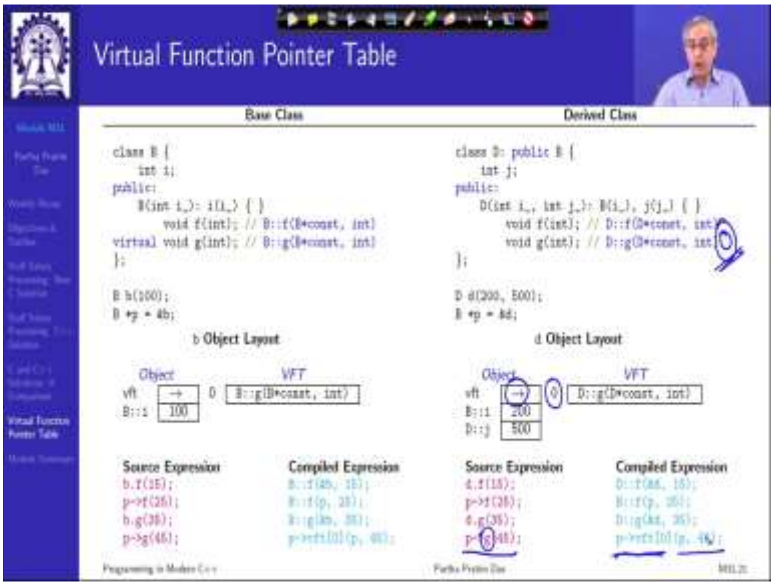


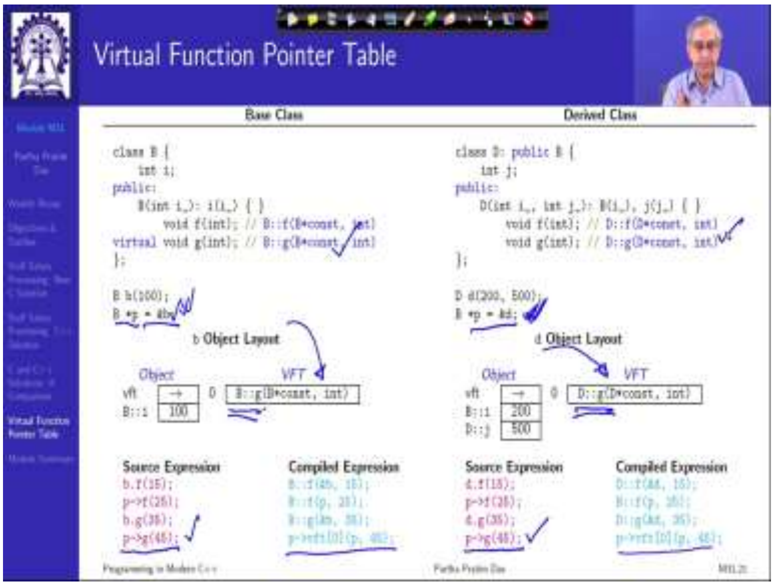
Now, D is derived from B. It has another data member j. I have instantiated d. Naturally d is constructed in the proper way. What D does? It over, it is overriding the function f, the non-virtual function in D. That is fine. It also overrides the virtual function g from B. So, it has these two functions now. I make a B pointer taking the address of the d object.

Now, the pointer static time is B, dynamic type is D. So, what will I have? I will have in this object i and j. So, for i, B::i, I have 200. I have 500. And this function pointer, virtual function pointer table, this pointer points to the VFT of Class D. This is of Class D. This was of class B. Why? Because class D may have overridden the virtual function that it inherited which actually it has done. So, it has overridden g. And therefore, I have D colon. Here I had B colon. So, it is a different function here.

Now, what happens? If I invoke the static function, I am sorry, if I invoked the non-virtual function with d, it will call the function for D. If I invoke it with p since it is non-virtual and type of p is of B, the base type it will invoke the function for B. If I invoked the virtual function g with the object d dot, it will again do compile time. So, it knows it is D. So, it is D colon g. All these are absolutely fine.

(Refer Slide Time: 26:54)



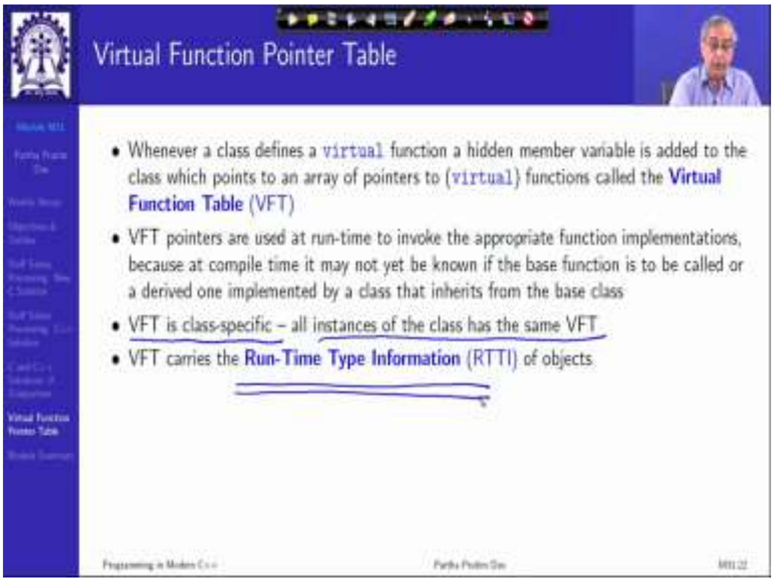


But when it does b-> g, it again has a virtual function g to deal with. So, what it will do? It will not put directly the member function. Because it does not know which one. What it does? It goes to p, picks up the virtual function table pointer, goes to index 0 because this still happens to be the 0th or the first virtual function and then takes this function passes the p and 45, the parameters to it.

So, you can see that the difference is if you look at p->g here in the context of p being &b and p->g 45 here in the context of this, at the compile time, at the compile the call looks the same. In terms of the translated compiled expression, they are same. Because certainly the compiler does not know whether this has happened or whether this has happened.

But at the runtime, in the runtime if this is there, then the VFT that exists is this which calls the B::g, this function. Whereas, if at the runtime p is pointing to a D object, your VFT is this one which has D::g, it calls the function of the D class. So, that is how polymorphic dispatch based on the runtime type, based on the runtime object you are actually pointing to happens. This is a very simple scheme but very effective one.

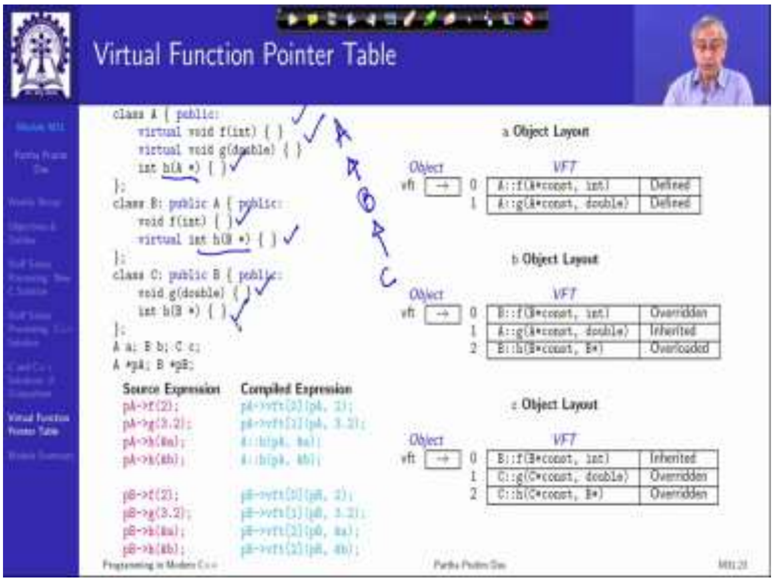
(Refer Slide Time: 28:47)

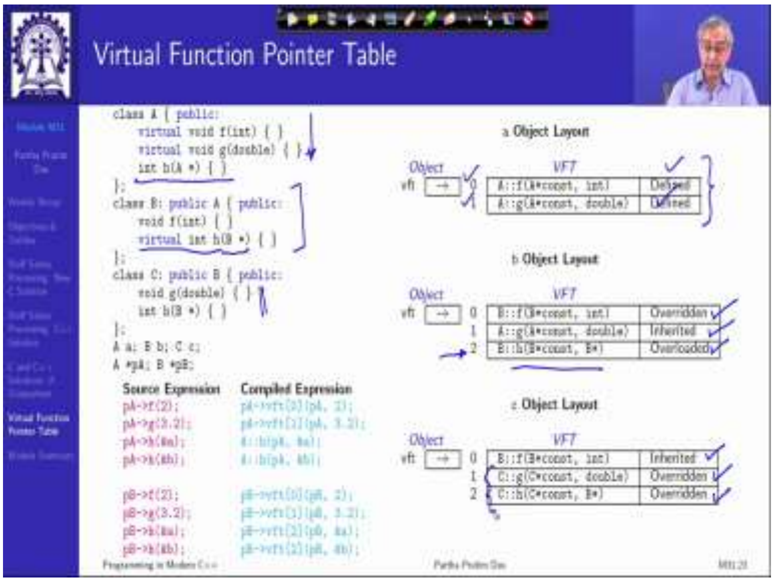


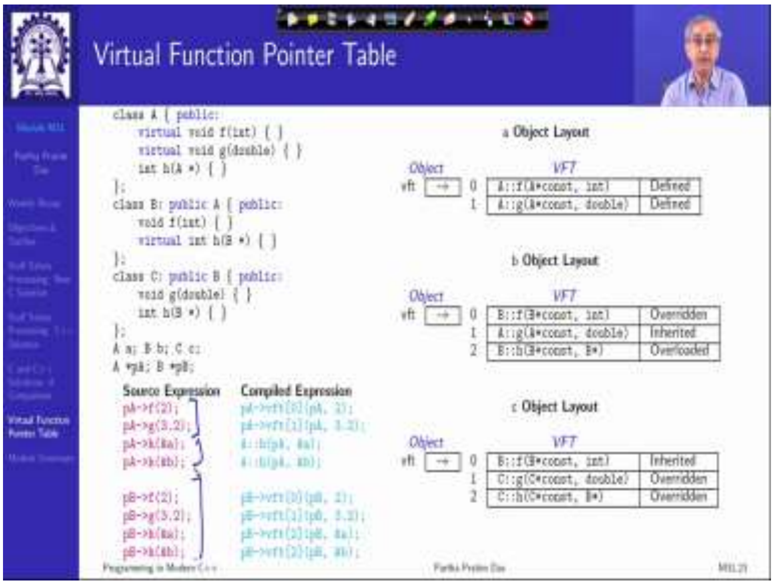
So, whenever a class defines a virtual function, a hidden member variable is added to the class which points to the array of functions which is the virtual function pointer table. VFT pointers are used at the runtime to invoke the appropriate function implementation. This point is most critical to remember that it is class specific and all instances of the class has the same VFT because it depends on the structure that exist.

And therefore, VFT carries, so this even though in C you never know when writing down what is the type of the object. But this pointer type, this pointed to the virtual function table will be different for every class but same for all instances of the same class. So, that at the runtime will always indirectly tell you which type you are dealing with.

(Refer Slide Time: 29:44)





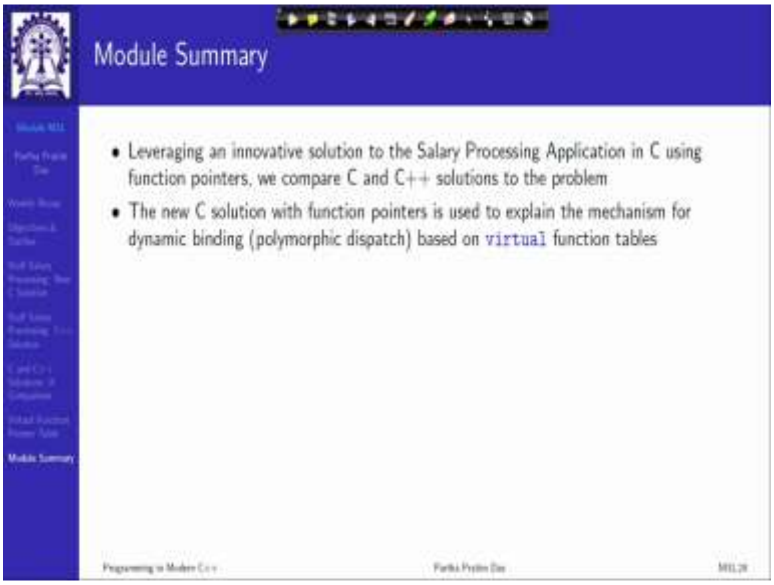


So, before I close here is a little bit longer example. This we had seen before. C is a B. B is an A. And we have two virtual functions here. One non-virtual. Here f has been overridden. g is simply inherited. And h has been overloaded, overridden plus overloaded because the parameter type has changed. Similarly coming to C, f is simply inherited. g is overridden. h is overridden as well.

So, in this context, if you look at the virtual function table, A will have 2 virtual functions, at 0 and at 1. It happens in the order in which you define them. The function f and function g. They are just defined. When you come to B, you are; note in A h does not make to the virtual function table because it is not virtual. But in B I have made it virtual. So, in B this comes in the location 2 where I have this h function which takes a B* as an argument. So, what I have got? I have inherited f and overridden. So, I have a new B::f, overridden. I have g simply inherited and I have h overloaded.

Similarly, I go to the next one. I have f, B::f simply inherited because I have not included it here. But C has overridden both g as well as h. So, here C of g and h coming together. In the in the left you can you how the calls will look like with the different virtual functions as well as with the non-virtual functions. I will leave that as your practice to check out. If you have any doubts, please get back to us. So, this again in one slide tell you almost everything about you need to know in terms of the virtual function pointer table.

(Refer Slide Time: 32:02)



So, to summarize, we have introduced an innovative solution to the salary processing application and C again using function pointers. And we have done that to show the parallel between the best possible C way of dealing with function pointers in flexible type design and the hugely advantages polymorphic hierarchy-based design in C++ which uses virtual functions. And from that we have introduced what are the different, what is the way virtual function table is actually implemented. I hope you have enjoyed it. Thank you very much for your attention. We will meet in the next module.