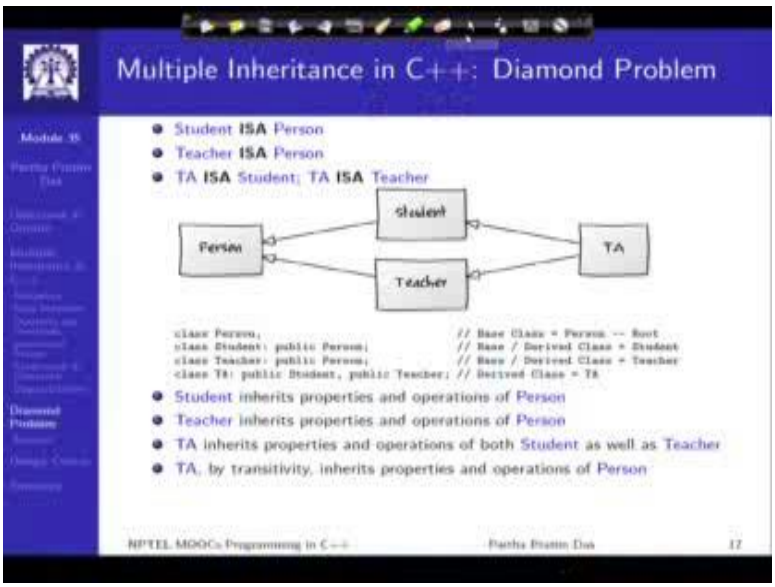
Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 51
Multiple Inheritance (Contd.)

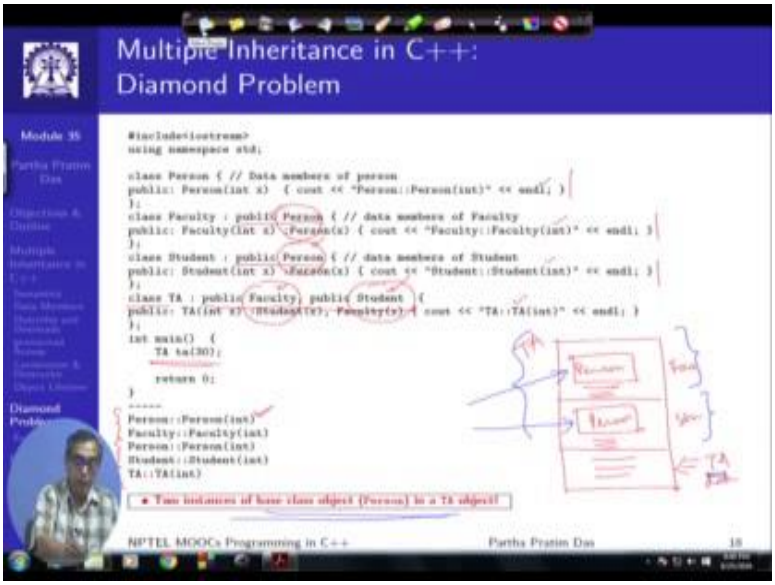
Welcome back to module 35 of Programming in C++. We have been discussing multiple inheritance and we have seen the basic mechanism of construction, destruction, layout, data members, member functions and what happens, if between the multiple base classes, if the data member or the member functions have duplicate same names.

(Refer slide Time: 00:40)



Now, we will look into some of the more integrate use scenarios and let us say that we have the students teacher TA scenario, where a TA is a student, TA is a teacher and both of them are persons. So, we have a diamond kind of situation and we call this as a diamond problem we will see why we call this is as diamond problem. We have already explained that it is very common that you will have a common base class to the base classes of multiply inherited class.

(Refer slide Time: 01:13)

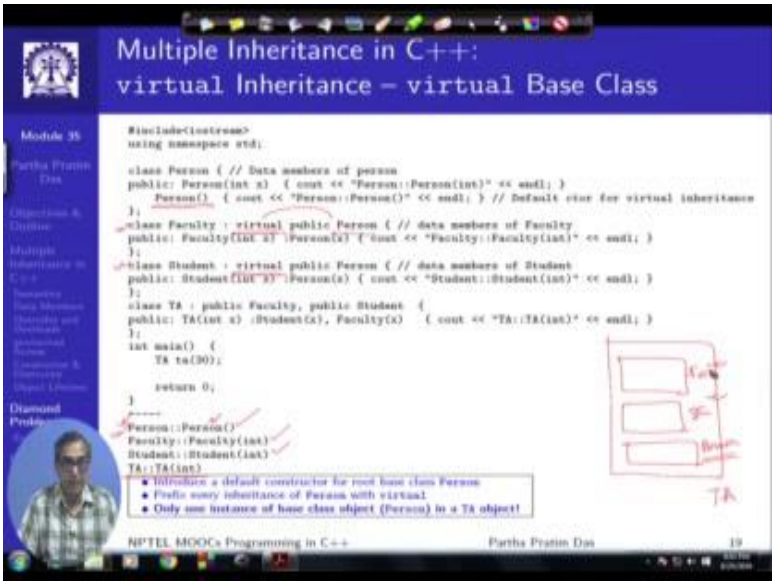


So, let us try to look into the code person is a class, here I have call it faculty, in the example in the code I have called it faculty, which means teacher is a class student is a class so they inherit from a person. So, and then TA inherits from faculty and student both, so this is the scenario. And just look at the every constructor has a message to see what is happening at the construction. So, in the construction happens certainly the base class will have to get constructed. So, what will have to get constructed for constructing a TA object is a TA object the faculty has to get constructed.

Now, faculty has a specialising from person. So, what will need to be done for constructing a faculty object a person has to get constructed, so these two construct a faculty object. Then the student has to get constructed and student is specialising from person. So, if I want to construct a student I will need to construct a person again. So, another person will get constructed and then student will get constructed, then TA will get constructed. So, if I look into the total object scenario, then I have two base classes this is faculty and this is a student. And this will have a person, this will have a person other than that you will have different faculty data here, you will have the student data here, you will have the derive class this is the TA object. So, you will have TA data here these are the TA data.

But this is how the construction will actually happen. So, what is interesting to note that when you construct like this our basic principle of single inheritance tell us that there will have to be two person objects in the same TA object. Because, otherwise the faculty cannot be constructed because it needs the base object to be embodied that, student cannot be constructed because it needs the person to be embodied that. Therefore, and TA needs both faculty and student to be embodied that. So, you will have two instances of the base class object and this is certainly not a very desirable situation, because certainly TA as a person will have only one set of attributes if I have two instances of the person class and how do I going to resolve in that, how would are you going to maintain the data.

(Refer slide Time: 03:54)



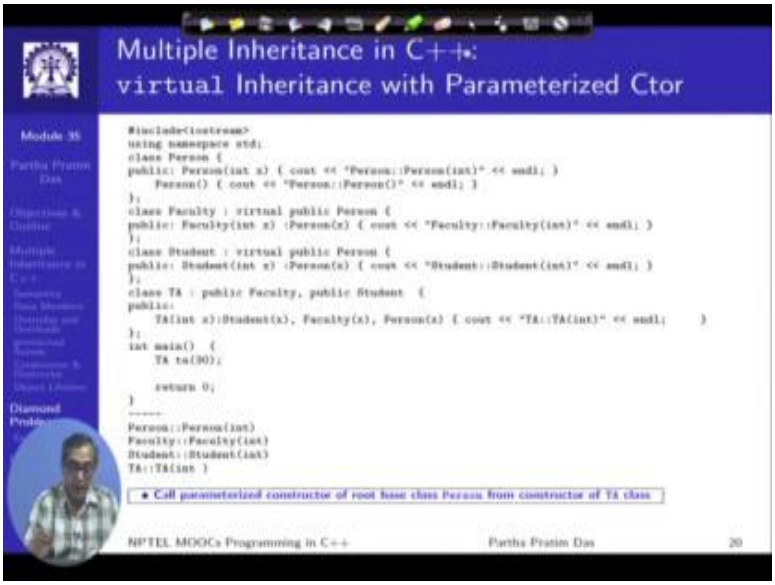
So, this needs, that leads to that what is known as virtual inheritance and multiple inheritance. What it say is we use the keyword virtual, let me use a red again, use a keyword virtual before or after it does not matter as to, whether you write it you can write it after public also, you can write it before public also, but you use a keyword virtual, where you are inheriting. When you do that then the inheritance become virtual which means that then, when TA has to be constructed your first thing is faculty will have to be constructed.

Now, if you say that faculty inherits person in a virtual way then it knows that some other specialised class is being constructed for which there could be multiple base classes. So, the faculty does not construct its person class, it does not where construct is person object, does not construct that instance. Similarly, when we do student it does not construct the person instance of the student class, which otherwise I would require. But the process constructs one common person instance for both faculty as well as student.

Now, how will this get constructed, now certainly faculty is not constructing a student is not constructing is because they are virtually inheriting from person. So, this has to be a default construction, this has to be a process of virtual inheritance that one single base class, unique base class will instance will get constructed. Therefore, I have introduced a default constructor for the person class here. So, now, we can see that person class is construct instances is constructed only once and based on that the faculty and student instances have been. So, if we look into the base class part of the faculty you get this person, if you look into the base class part of the student you again get the same person instance, if you look into certainly the base class part of the TA then of course, you get the same person. So, you kind of uniquify the instance of the base class.

So, this when we use virtual inheritance and construct the hierarchy in this way to avoid having multiple instances of the route class, multiple instance of the diamond class into the derive class instance, we use virtual inheritance and such classes as known as virtual base class. So, this is a virtual base class, this is a virtual base class, these are virtual base classes VBCs. Because they will not directly construct their base class part instance of the base class part that will be common with the total, derive class object. Now, in this process this solves a basic problem of object layout in case of diamond that is this will happen if you have diamond. So, that this class is getting constructed here as well as getting constructed here. So, virtual inheritance basically eliminates that problem, but with one small restriction that for doing this since we have done this automatically we could use only the default constructor the person class. So, what if I want to pass parameters to the person class that is what if I want to call this constructor.

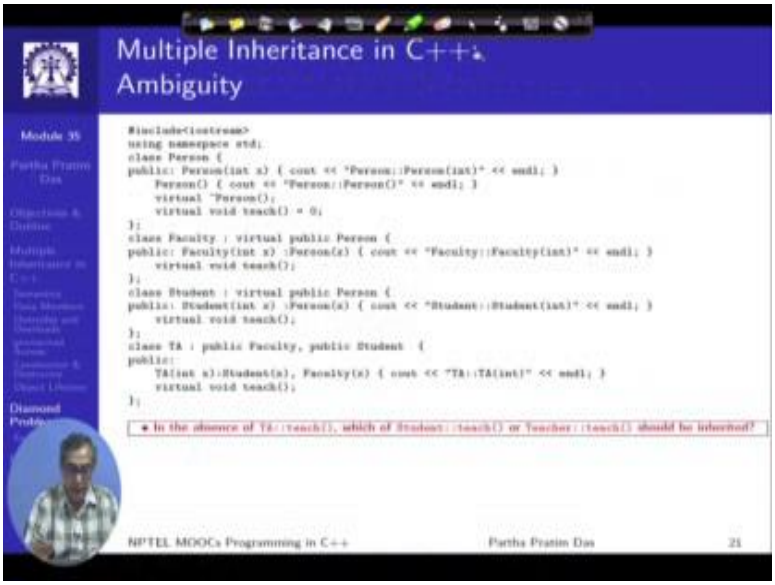
(Refer slide Time: 07:40)



It is that is the very simple way of doing this in multiple inheritance, all that you need to do is these remains same you are virtually inheriting, this virtually inheriting, this these remain to be VBCs virtual base classes all that remain same. The only difference is in the constructor of the TA class, in the constructor of that is, in the constructor of the glyphs derive class you explicitly call the constructor of the route class and there you pass the parameters. So, what will happen if you do this then, this will be the first to get constructed which will call this constructor which has parameter.

Then according to this order this will be the next to get called and this will be the last to get call that is why you will see that it is person faculty student and TA constructor in that order that it will get constructed. And you still have now, have that original problem resolved you have a unique instance of the base class, but you have been able to pass parameters to the constructor. So, that is the basic way to construct a multiple inheritance hierarchy.

(Refer slide Time: 08:50)



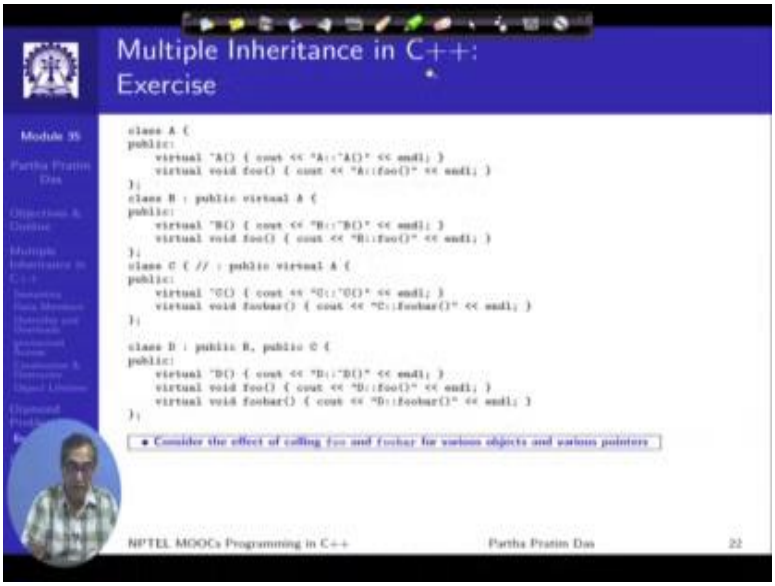
And you can now go ahead and make that polymorphic and a teach member method say to the base class of person make it purely virtual, because a person you do not know how a person can teach, so instantiate that, implement that in faculty as a just a non-pure virtual function, instantiate that again in student and so on. But as you do that certainly you will have a conflict in terms of whether you will use this or you will use this and on a purely virtual on a polymorphic hierarchy certainly you cannot specify us to which member function you will use like you did in case of a non-polymorphic one using the using method. So, the only way you can leverage this is actually override both of them using a new member function teach in the TA class as well. If you do not do this then certainly, when you do when you try to do an instance of TA it will say that we cannot do that because you do not know which of these two teach function should this object use.

So, once you have over written this so that means, you directly hide this in the virtual function table then you can that the code will work and then depending on what you want to do. If you want to reuse this function then within the implementation of teach in the TA class you could certainly refer to it by faculty colon colon teach or if you want to reuse teach of the student class you can use student colon colon teach or you could implement it on its own. So, this is the basic issue of ambiguity in case of multiple

inheritance with diamond structure, we will not have this unless you have a diamond structure the reason you are you are getting this is, because you have the diamond.

Therefore, since you have the diamond there are two ways that the teach function can reach here, the two ways that teach methods can reach here, if you have more base classes and a common diamond then you will have more ways of doing that, but two is enough for the confusion. So, here you do not know whether you should use teach of this or you should use teach of this. So, that is a basic problems and that leads to if you want to really have a generic multiple inheritance hierarchy that leads to several issues. And more often many of the organisations prescribe that you should not actually use this should not use diamond in the multiple inheritance scenarios of course, that restricts the total use of multiple inheritance in a severe way. But this is a serious interpolation problem which will have to live with.

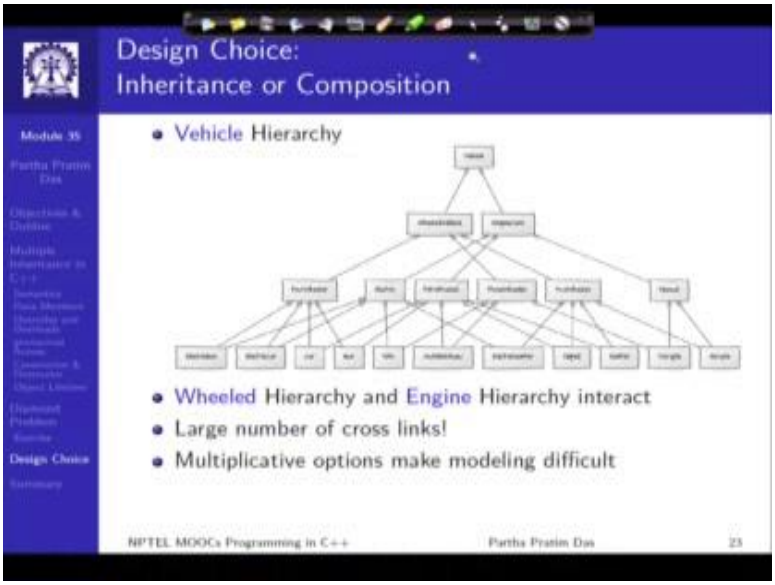
(Refer slide Time: 11:44)



So, in this context, there is a; this is an exercise, where there is a class A, there is a class B. So, there is a class A, class B and then there is a class C and then there is a class D. So, this is a scenario like this. So, this is a again a scenario of multiple inheritance, but this is not exactly a diamond. So, on this there are different member functions foo and foobar defined. So, I would just suggest that you having known all that you have studied

in an inheritance in casting as well as in multiple inheritance. You try to create instances of different class object, try to take pointers of different class types and try to shake out as to which how you can invoke the different member functions on this hierarchy. Of course, you can finally, make the whole thing complicated by also letting C derive from A the moment you allow C derive from A you get into diamond and you will have lot of interesting problem of ambiguity that come in.

(Refer slide Time: 12:58)



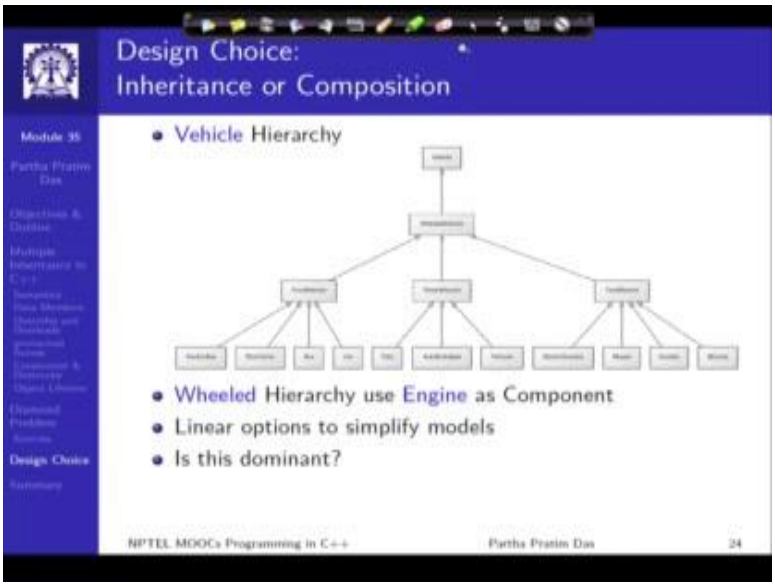
So, finally, before I close, I would like to just give you glimpse of a issue of design choice, as to whether I am we can always model in terms of inheritance and in place of that we could also do composition the other way hierarchy. And there is always a trade of in the designers to whether you should do inheritance or you should do composition. So, just may have created an example here to show you what kind of difficulties you accept. For example, I am looking at vehicle hierarchy and these are the primary you know sub classes of the vehicle that is you are checking out is the class of wheeled vehicles that exist in the world.

And where different types of driving mechanism, the engine class are basically the different drive mechanism that can happen for that and if you look into that then in terms of the wheeled one you may have a two wheeler, you may have a three wheeler and you

may have a four wheeler these are the different options. And in terms of the engine class you may have a manual drive you could have a petrol fluid drive you could have a electric drive these are the all different. So, if you have all these, so is basically you have two major hierarchies one based on the wheel drive, one based on the engine class. So, if you look at that then based on these you have a whole lot of you know IF classes that come in based on different combinations, like I can talk about a bicycle which is manual and two wheeler, tricycle it is manual and three wheeler. So, that is the multiple inheritance happening I have a car which is a four wheeler and petrol fluid I have an electric car, which is four-wheeler but electric fluid and so on.

So, depending on I have three types of wheeled vehicle I have three types of a engine drive. So, actually I have a combination of three into three nine combinations and a there may be several live class which have the same base class parent. So, you will actually have lot more than nine explosive kind of combine multiply get the evenly explosive kind of combinations at the live level. So, this is quite in terms of modelling, but when you actually have to manage the code, and remember these whole hierarchies and like the logic on top this, deciding at every stage as to what is inherited. And you know what using you should use and what you should override this becomes not a very good aid in the design rather it becomes a hindrance.

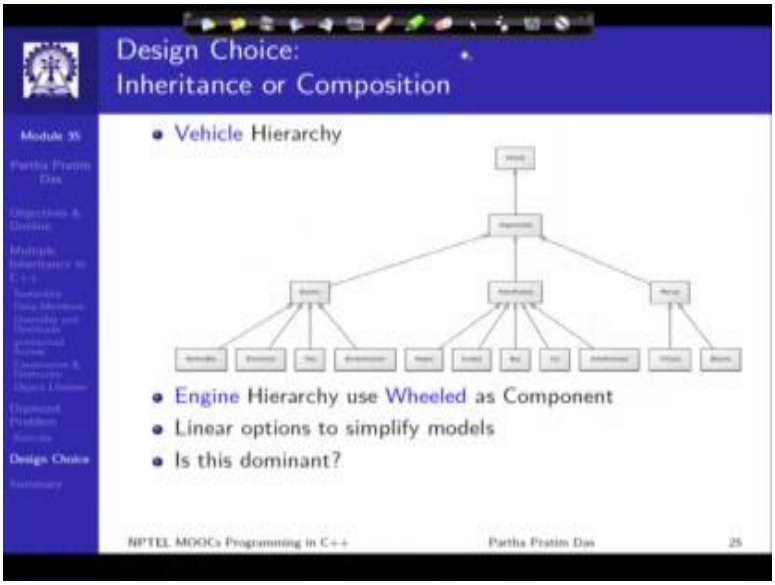
(Refer slide Time: 15:48)



So, what is a general prescription is that if you have multiple possible hierarchies in a domain then you should choose one which is the dominant one, that is one way of looking at even that may help you get read of multiple inheritance in in many places is as the vehicle hierarchy. So, I am just looking at the hierarchy of the wheeled vehicles. So, I am not looking at the engine types. So, what I will do is whenever I make a vehicle object of any derive types.

So, these are different wheeled vehicles four, three, two and then you have all the base class, for that I will have in this object I will have an engine star say pointed to engine and this will basically point to the engine class having all the different specialization. So, whether I have manual, whether I have electric, whether I have petrol fluid and so on, instead of deriving this object jointly from two hierarchies I will derive it on one here on this particularly wheel side and then refer to the engine hierarchy as a member, as a composition. So, this pointer will then lead need to the particular type of engine as I want.

(Refer slide Time: 17:13)



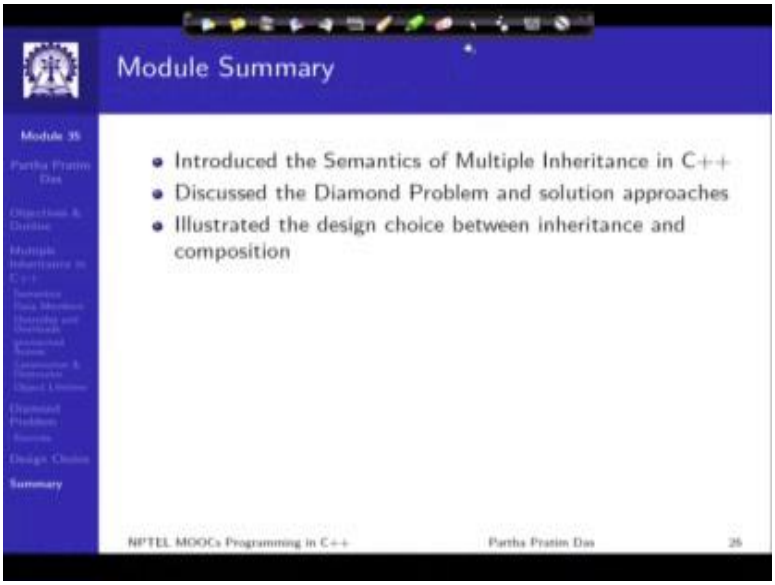
Of course, if I am doing this, then of course I can do the other way round, I can do a vehicle hierarchy again, which is basically based on the engine class. So, I have the electric engine, petrol fuel engine, manual engine I have my (Refer Time: 17:27) classes

on that, but now in a wheel I have a wheel pointer, which basically tells me the hierarchy of two wheeler, three wheeler, four wheeler and so on. And I primarily work on the engine type as a dominant hierarchy and use the other one as or component of this vehicle type.

So, either of that can be done, if there are multiple notations in the in the multiple inheritance structure that exist then you will have several such options. And the most prescribed and more commonly used style is you identify one of them, one of the different options of inheritance is a hierarchy that you are getting take identify one of them which is dominant and put a single multi-level and you know are kind of a hierarchy on that. And all the other hierarchies that you are getting all the other is a relationship like the engine one we saw here make them into their components. And then you travels that components and going to the particular hierarchy like this. And do the reasoning about the engine type on this hierarchy if you want say you want to have another hierarchy in terms of the carriage type as to whether it is for passengers, whether it is for a goods, whether it is it is for customer and so on.

So, accordingly you have to it; now this is a design choice to what is a dominant one which must be on a hierarchy. So, that you can actually write polymorphic code on that and then refer to the different components in the alternate hierarchies. And there are several design techniques by which you can make what is known as a double dispatch like jointly you can a decide on two independent hierarchies and actually dispatch a polymorphic function that will be beyond the scope of this course as you become more and more expert, you will able to learnt and see all that.

(Refer slide Time: 19:37)



So, to summarize in two lectures in this module, we have introduced the concepts of multiple inheritance, we have explained the basic semantics. And I have tried to just give you glimpse of what are the different pitfalls that may happen, when you use multiple inheritance. And at the end up, try to give you some idea about a design choice between using inheritance as only mechanism of object structuring vis-a-vis making a mix of inheritance and composition and deciding on a dominant hierarchy of object as your major polymorphic representation of the objects. And using the other ones as composition in terms of reference and then making your polymorphic dispatch according to that.