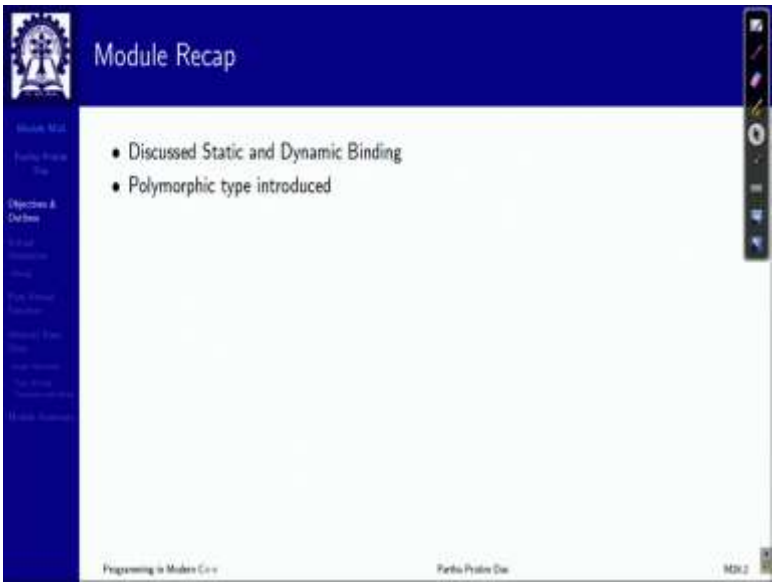
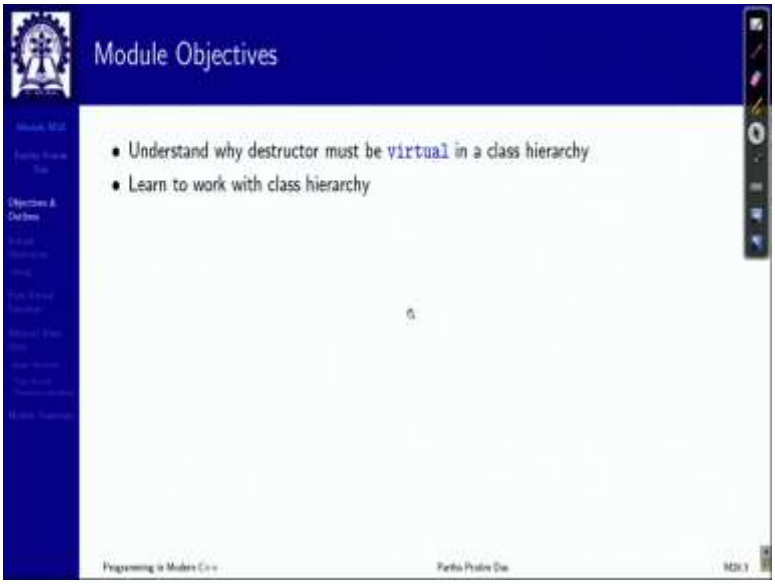
Programming in Modern C++
Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 28
Polymorphism: Part 3: Abstract Base Class

(Refer Slide Time: 0:37)



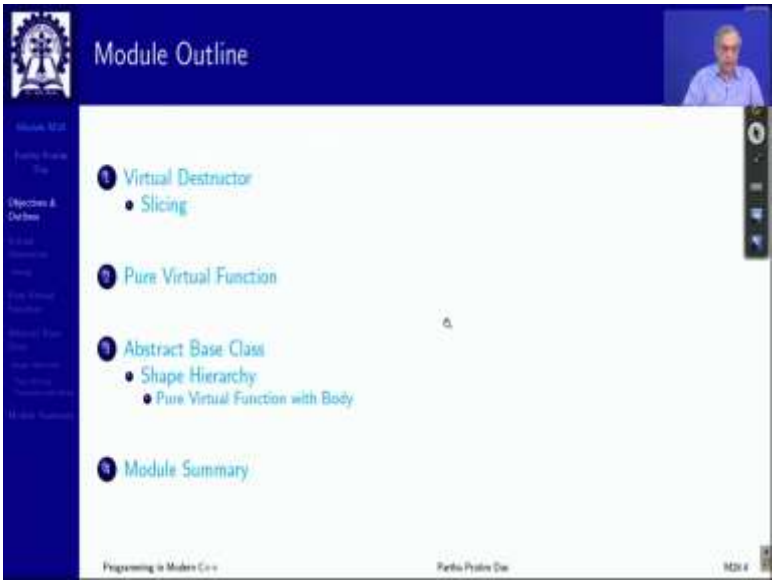
Welcome to Programming in Modern C++. We are in week 6, and we are going to discuss module 28. In the last module, we have discussed about static and dynamic binding. We have learned that an object may have a compile time type and a runtime type based on the type of binding it receives. And based on that, we have polymorphic type introduced in C++.

(Refer Slide Time: 1:09)



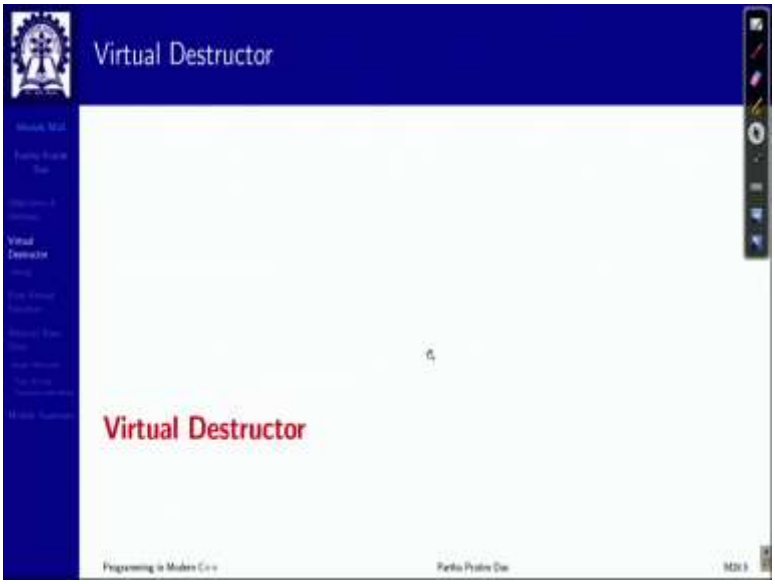
In the module today, we will build up on that polymorphic types and first try to understand why the destructor must be virtual in a polymorphic class hierarchy. And then we move on to work, learn to work with class hierarchy for a nice design.

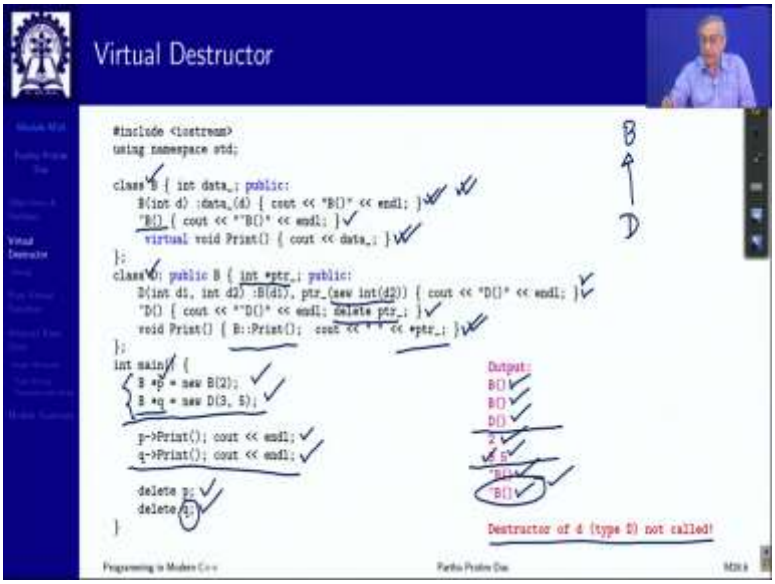
(Refer Slide Time: 1:35)



Here is an outline which will be available on your left panel all the time as it does.

(Refer Slide Time: 1:44)





So what is a virtual destructor? Let us look at a tiny hierarchy of generalization specialization and identify some problem. So let us, we start with the simple design where there is a base class B and a derived class D. So you have B. you have D. Now we have the constructor for B, destructor for B. Constructor for D, destructor for D.

We have a pointer data in class D which needs to be deleted which is allocated in the destructor, constructor and needs to be deleted in the destructor. And just to illustrate the problem, we introduce a virtual function: Print(). Naturally override it in the derived class D. So this is the simple design.

We construct, dynamically construct two instances: one of B and the other of D and put the pointers to B types pointers. As we do. We have seen how to do this by keeping doing an

upcast and keeping the pointer to the object by the base class pointer. So for p the static as well as the dynamic type is pointed to class P. Whereas, the static type of q is B. Whereas, the dynamic type of q is D. Because it is actually pointing to D.

So as we do this construction, what will happen? For new B (2), the constructor of Class B will get invoked. So this output will come. For the construction of D (3,5) first again the constructor of class B, the base class part of the class D object will be constructed by the base class constructor.

So B will be constructed. B will be called. Then we will have the completion through the construction of D. So this is what, how the two objects will get constructed up to this point, up to this point.

Then we are calling the virtual function Print() from the p pointer. Obviously, it will print for the object that has been created as B (2). So we will print 3 2. And then we call the virtual function Print() with the q pointer which is overridden.

And because it is a virtual function, even though the type of q is B, it will actually call the function based on the actual object existing, the type of the actual object present which is the D type object. And therefore, it will call this Print() function. And which in turn will first print the base part which is 3 and then whatever has been created as a pointed data. Yes? 5. Done up to this point.

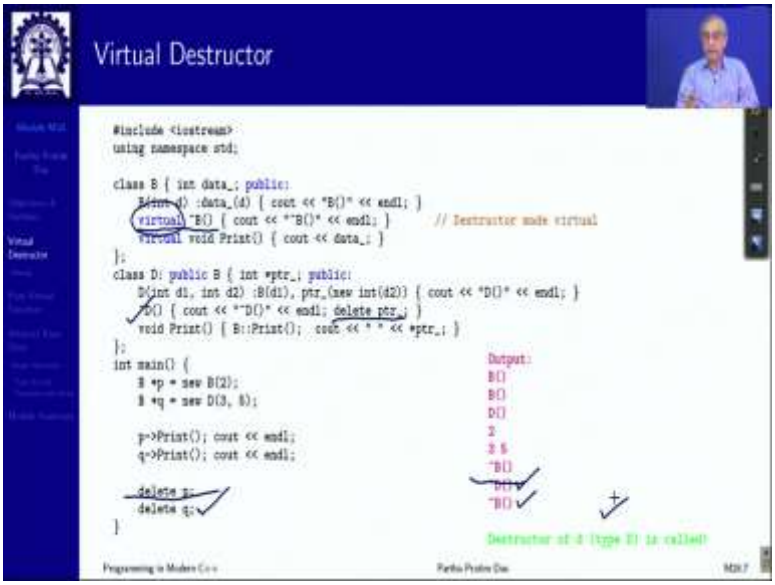
Then we will have delete p. So what does delete p do? delete p will try to dynamically deallocate the pointed p which will have to call the destructor of B, destructor of B. Now, then we call delete q. Delete q should actually call the destructor of B followed by the destructor of D. Because the actual object is of D type. But if we see in the output, we see only the destructor of B. The destructor of D is not called. Why is it that?

It is very simple. Because the we know that the polymorphic dispatch or runtime dispatch can happen only for functions that are virtual. Now the destructor in D is not a virtual function. Because we have not defined it to be a virtual function. The destructor in D is not a virtual function.

So what happens? When I call the delete, when I do delete q, the function the destructor function is statically bound. Since q has a type of B, it is bound to the destructor of B. And that just is called. Whereas, it should have called destructor of D by dispatch which in turn

would have called the destructor of B and completed this. So what means that only a part of the object has been deleted. The rest is remaining. So this is the basic problem that arise.

(Refer Slide Time: 7:18)



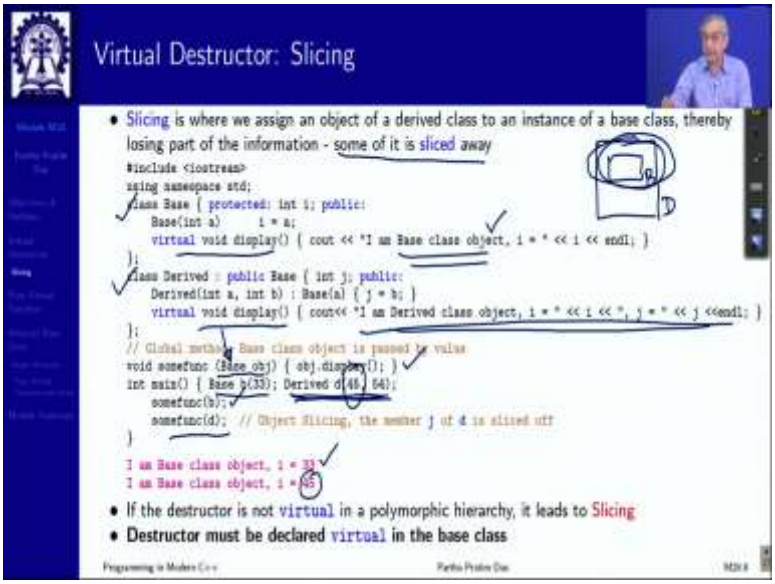
Now let us see how to solve that. It is it has a very easy fix. All that we need to do is to make the destructor of the base class B to be virtual. Everything else remains the same. There is no other change. Up to this point, up to this point everything is identical.

Now what happens? When I call delete q, though q is a pointer to the base type but that base type destructor is virtual. So it dispatches based on the actual type of the object, the dynamic type of the object. Not the static type that the pointer is having. So it will actually call the destructor of D.

And this is possible, this is happening because I have said that the destructor of B is virtual. I do not need to explicitly say that the destructor of the derived class is virtual because you know that on a hierarchy once a function is virtual, all its overrides are becoming do become virtual.

So now D is, destructor of D is called and then that will in turn call the destructor of B. So first the B part of the object is cleaned and then the D part is clean. So delete pointer will now correctly happen and the destructor is called. So this is the basic design. This is what is called a virtual destructor and is always required in a polymorphic hierarchy.

(Refer Slide Time: 8:54)



So the problem that we are we are specifically talking of here is that of slicing. So slicing is like, you have the derived class object which has a base part. Now, if you just use a type which is of base type without polymorphism, then only that base type will be used and rest of the object will not be used. Rest of the object will be kind of sliced away.

So you have a base class here. You have the derived class in this. You have a display, polymorphic display for that. And some global function, somefunction which takes a base class object. And we have base class and derived class objects. We called somefunction on the base class object. We get what is expected. We call the somefunction on the derived class object but it does not get, this function does not get called.

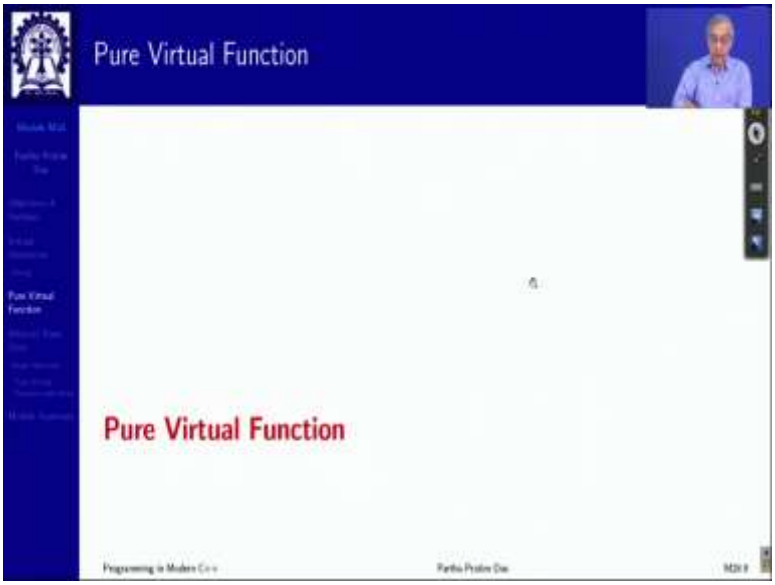
Rather this function is getting called because you have passed the object as base class object. So what happens? When you call it with the derived class object, it copy no, it cannot copy the entire object. It copies only the base part of it. It copies only the base part of it, not the entire object.

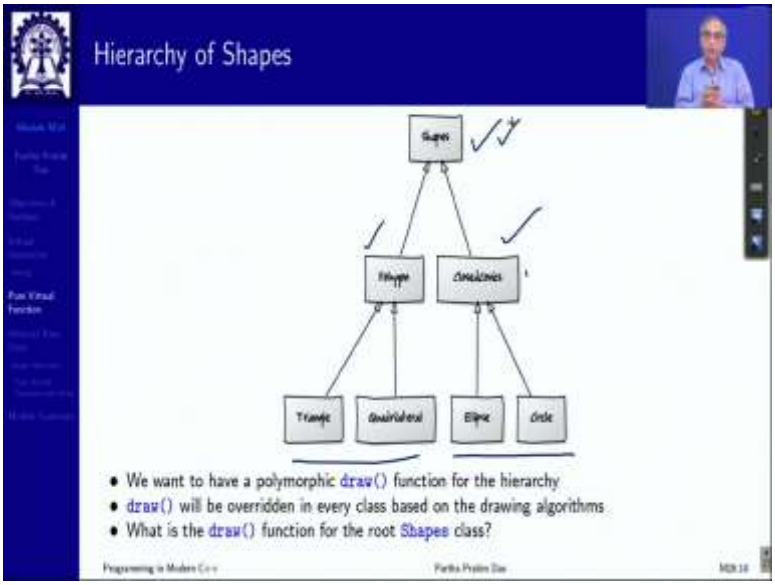
So it gets only the 45. Its runtime type remains to be of base. So therefore, the function actually called is the display of the base class and you just have 45 printed. So, this is a very standard problem of slicing.

So whenever you deal with the hierarchy, we will have to be very careful about this problem. Make all functions properly virtual and pass them with making sure that you do not have slicing coming in. So if the destructor in a polymorphic hierarchy is not virtual, it is leading

to such slicing so that the base part is destroyed but this remaining are not. So the destructor must be virtual in the base class.

(Refer Slide Time: 11:17)



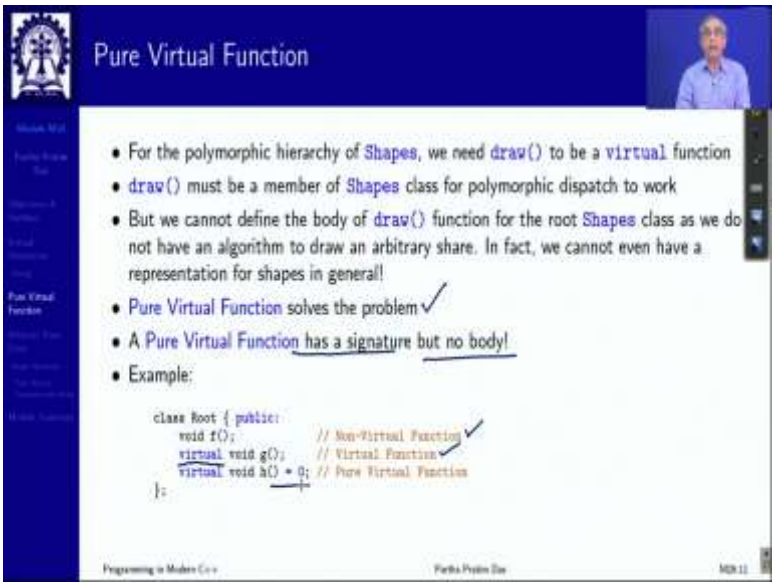


Next talk about pure virtual functions. Suppose, we have a class hierarchy of shapes. This is a shape: Polygon, ClosedConics, different types of polygons, different types of conics and so on. And what we want is we want a draw() function which is polymorphically available all along this hierarchy.

So we will define it in the base class, route class. And then, as we get to know because the way you draw a polygon is different from the way you draw closedconic or a triangle or an ellipse and so on. So we will keep on overriding the draw() function. So that whatever is a I mean if I have a pointer to shape, I will be able to call the draw() function. It will get

dispatched. It will call the actual draw() function of the object for the class. Now the question here is, what is the draw() function of the route class Shapes? Do we know that? Can we write that function ever?

(Refer Slide Time: 12:18)



The problem is the route class, the Shapes class need to have the draw() function so that it can be properly dispatched along the hierarchy. But we cannot define this draw() function because there is no algorithm to draw an arbitrary shape.

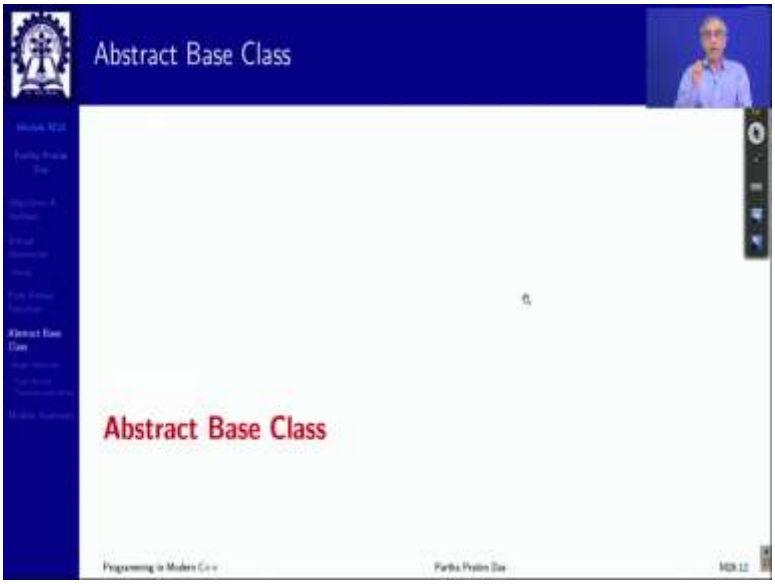
Shapes is not saying what it is. It is some shape. In fact, we cannot even have a representation of shapes in general. So here is an requirement that I need this function draw() to be present in the shapes class but I do not know how to write the body of that function.

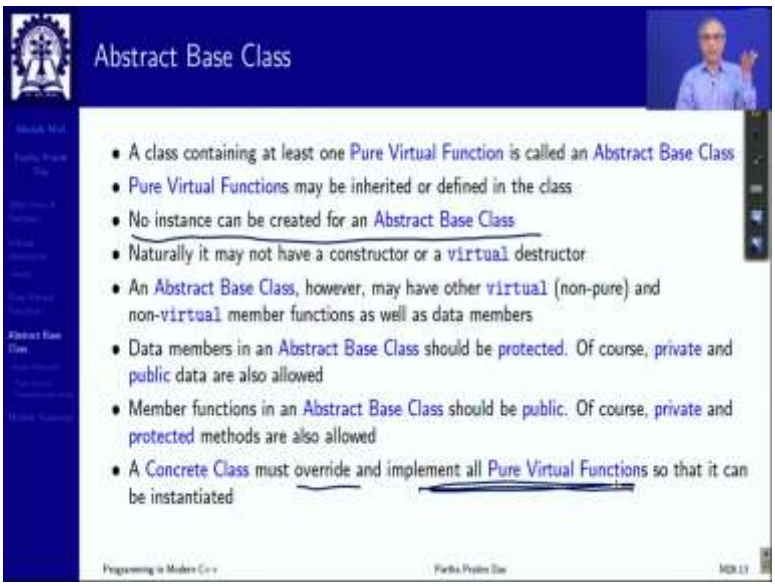
So, pure virtual functions basically solves that problem. It has a signature but no body. And how do you write it is very simple. This is just a complete example of member function types. You have a, this is a non-virtual function you have known.

If you put the keyword virtual it becomes a virtual function. If after making a virtual function, if you put this notation as if equal to 0, then it is taken to be a pure virtual function.

So the compiler does not expect that you will give a function body for this function but it will know this signature to be used in all along the hierarchy and these functions can be overridden in the subsequent derived classes as and when necessary.

(Refer Slide Time: 13:49)





So there is a big consequence of this. What? So now we have a class which has a member function which we know but we do not know how to implement it. So if we have an object of that class, naturally that object would like to call that member function. But it does not have a body. So what does it do?

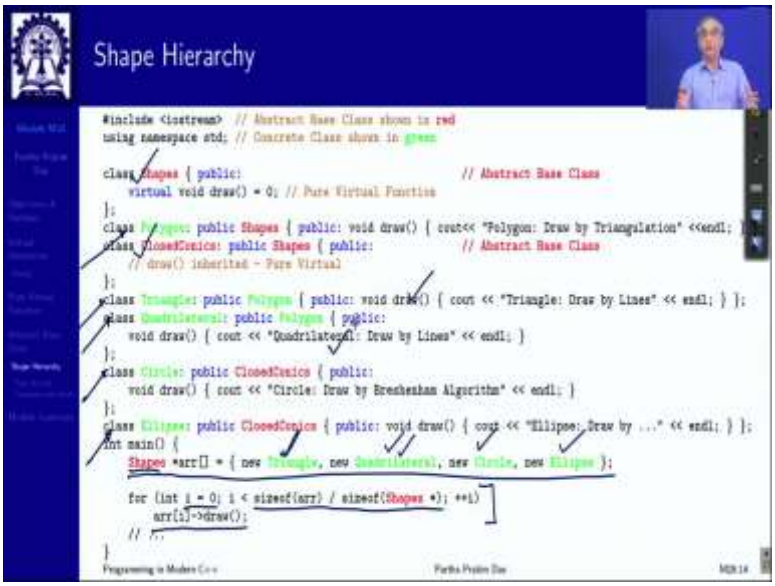
So the consequence of pure virtual function is that if a class has a pure virtual function, you cannot instantiate any object. It is more like an interface. It is an interface. So you cannot have an instance of such a class which is therefore known as an abstract base class in contrast to other classes where you can create instances and you call them as concrete classes.

So an abstract base class is one which has at least one pure virtual function. Because if you have at least one, then you cannot instantiate an object but it can have other data members. It

can have virtual and non-virtual functions. You can have them in all kinds of specifiers and all that. But when a concrete class is derived from this, it will have to override and implement all pure virtual functions.

If the abstract base class has two virtual, pure virtual functions. A concrete class derives from it and overrides only one and implements it but does not do the other, then that class also is containing one pure virtual function. And therefore that class also will remain an abstract base class. So that is the whole idea.

(Refer Slide Time: 15:40)



So now let us look into the Shapes with this glasses of pure virtual function and abstract base class. To solve the original problem that I want to have the hierarchy but at the root I do not know how to draw shape. So, here is what do we do. The colors used here for class names have a meaning that anything in red is a abstract base class. Anything which is in green is a concrete class. This is just to make it clearer for you.

So in Shapes, we are not putting any data member or anything right now just to make our understanding clear. So I have a pure virtual function, draw(). So by that what I am saying? All that I am saying is on this hierarchy, it is possible to call a function draw() on an object without passing any parameter and it will draw. It does not return anything.

But to be able to do that, I have to specialize and override and implement that function. So Shapes is an abstract base class. Polygon is derived from that. And in Polygon as if I have implemented this function. So I have overwritten and implemented this function. Since, you

know, drawing is a complex code, I have just, you know, put as placeholder certain cout statements so that we can know that this function is getting called and so on.

So this draw() is overridden from overrides this, but implements it. So, Polygon now becomes a concrete class. In other terms, let us say I have ClosedConics which I say is a specialization of shapes but also in general a ClosedConic I do not know how to draw it. So I do not, I have specialized but I have not overridden the draw() function. So what happens?

ClosedConics inherit the pure virtual function from Shapes, the pure virtual function draw() from Shapes class. And therefore, it contains a pure virtual function and therefore, it is still an abstract base class.

Then Triangle is derived from Polygon. I overwrite to "Draw triangle by Lines". Quadrilateral is derived from Polygon. I overwrite draw() to draw a quadrilateral because all these will be different algorithms. I derived from Circle from ClosedConics and now I overwrite and implement that function, how to draw the circle. So now Circle becomes a concrete class.

Similarly, I do that for Ellipse deriving from ClosedConics and my whole hierarchy is now ready. You can even see that there are two abstract classes of which I cannot have instances and there are four polygon actually five, five different concrete classes of which I can create instances.

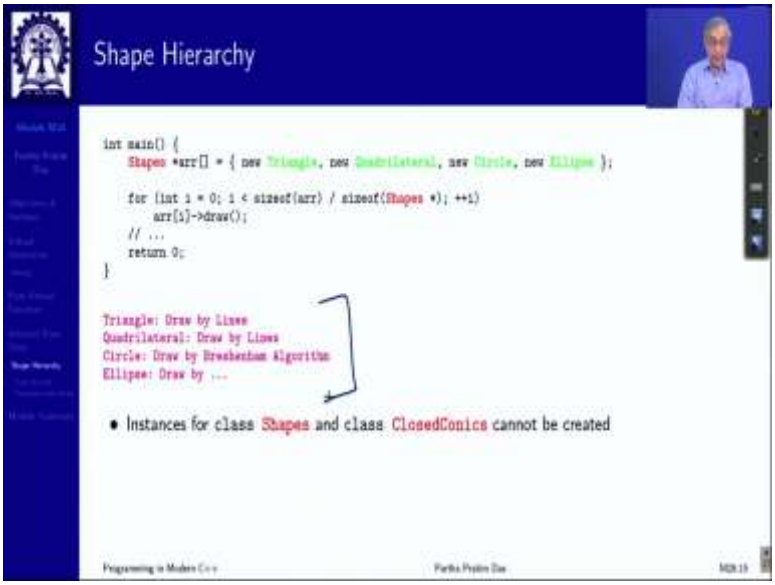
So now I create a Shape array with four types of objects: a Triangle, a Quadrilateral, a Circle, and Ellipse. So I create the objects dynamically and have an array of Shape pointers created. And by the seamless upcast, the pointers which are of Triangle pointed to Triangle, pointed to Quadrilateral and all that can be upcast to be pointers of Shape.

And then this in this whole panorama, suppose I have a canvas on which all these shapes exists. I will be able to just go over the Shapes array and draw all the shapes in a very simple you know, two line for loop that is I go over from i to the number of elements in the shape and it counts that. And then for the ith element, which is pointed to a Shape, I invoke draw(). And this draw is a polymorphic function. So for the array[0], it will call the draw() of Triangle. For array[1], it will call the draw() of quadrilateral and so on.

So, that is how very nicely I can put this entire hierarchy code in a very uniform compact manner. And the ability to have pure virtual function, abstract base classes allow me to have

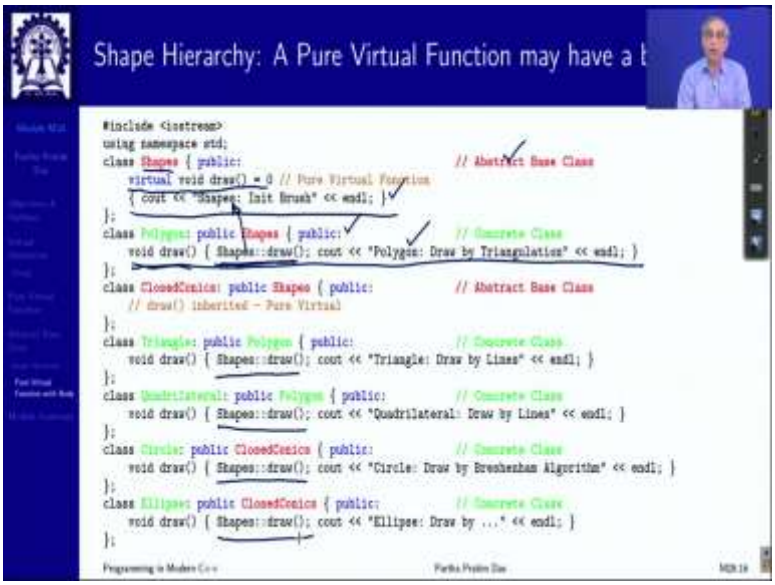
this uniform design. Otherwise, if I did not have draw() in the root of Shape, root Shape, class Shape, then I would not have been able to write such a compact code for this entire design.

(Refer Slide Time: 20:33)



So this is just you know repeating the main function here just to show you what are the different outputs that you will get. And again, I reiterate that Shape and ClosedConics cannot be instantiated because they are pure, they are abstract base classes.

(Refer Slide Time: 20:55)



Now, the pure virtual function does not have a body. That is what we started with. But there is a little twist to that. Actually, compilers do allow pure virtual functions also to have a body but it is not considered implemented. I mean it is a pure virtual function then even though it

has a body, it will not be considered to be an implemented function and the class will continue to remain an abstract base class.

So, let us see how we do this. So in Shapes I have a pure virtual function. Therefore, Shapes is an abstract base class. But I have provided a body for the draw() function in shape. Now, the question is why do I want to do that? The common reason would be several times there is even when I overwrite the functions all over the hierarchy, there maybe some common part of the code which every overridden function needs to implement.

Now, instead of putting that, you know, repeatedly copying it on all overridden functions, I can put all of these in the pure virtual function at the root and just call that function. So you see is something interesting. You are not able to construct an object of Shapes because it has a pure virtual function. It becomes an abstract base class because it has a pure virtual function. But it is still possible to provide an implementation for it and call that function.

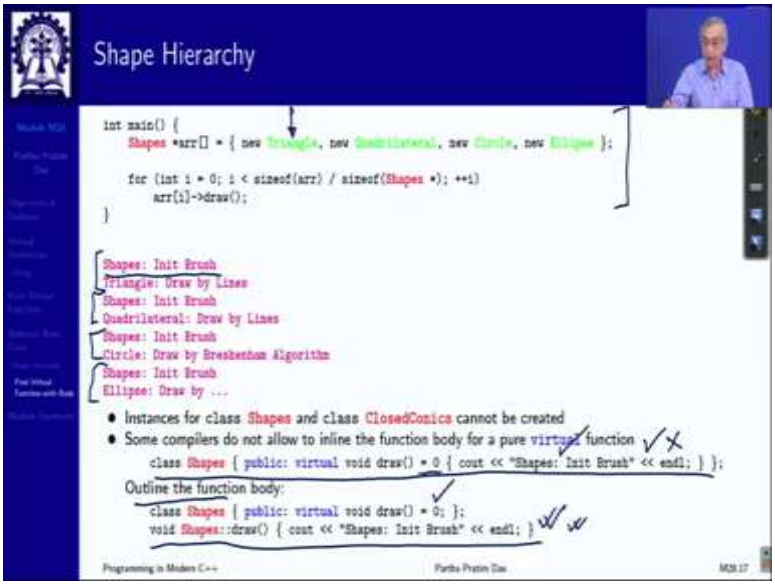
So what I do here can see here, conceive here is anything you draw, you need to initialize your brushes, your palettes and all that. So I say when Polygon is a concrete class. So when I override and implement the draw() for Polygon, I actually before whatever I was doing, I actually call Shapes::draw(). Note, I cannot call it by the object notation because I cannot construct any object of the Shapes class. But I can always call it by the by its fully qualified name.

So what happens is, when the draw() of Polygon is called, it will first call this pure virtual function which incidentally has a body. It will collect the, do the initialization to the brushes and all that. And then do the actual drawing.

Everything else remains same except that whenever I overwrite and implement the draw() function in the concrete class, I am calling Shapes::draw() to very compactly have all my initialization done before the actual draw() starts.

So that is the basic advantage. Otherwise, whatever this code, whatever this initialization code need to do, I would have required to you know, copy all of these here. Or I would have required to have another function in Shapes which have this initialization and call it all over. So it is not that you will necessarily have to follow this style but this is something which is possible. So I just wanted you to know this.

(Refer Slide Time: 24:24)



So if you if you do that run the same code now with the same main function that does not change, you can see for every case before the function; For example, first for the Triangle. This one. First the Init Brush in Shapes is getting printed because the draw() of Shapes is invoked. Similarly for Quadrilateral, for Circle, for Ellipse and all that. So this is, this is what is additionally available for you in the design.

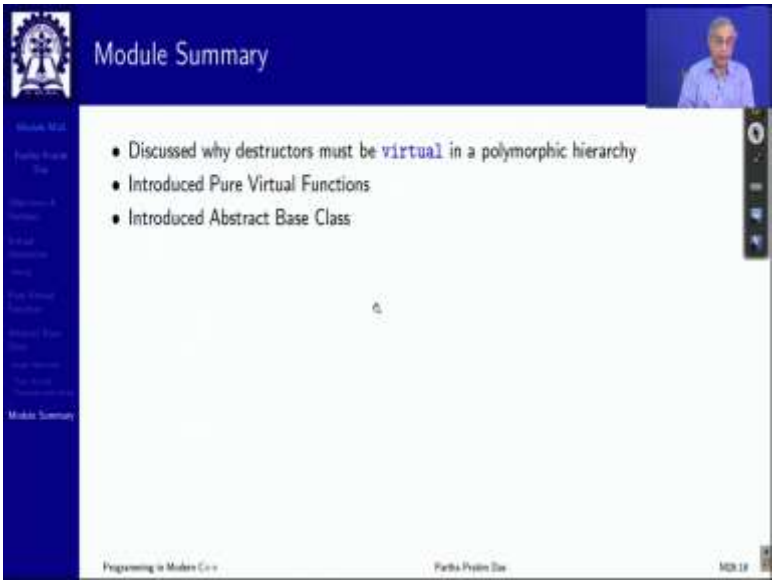
Again, this does not change the abstract base class status of the classes is something clearer to deal with. That since I have told that this is a pure virtual function, no instantiation will be allowed even if I provide a implementation of that pure virtual function.

Now, the way I have written it here which is kind of an inline style that after saying this pure virtual function right there, I have put the implementation which is in-situ, inline style of writing member functions. Not all compilers will accept this.

So if your compiler does not accept that, you can outline the function body. That is within the class just define the signature of the pure virtual function and outside the class as you can normally do, you write Shapes::draw() and implement that function.

This will, this second version will work with all compilers. This will, this compact version will work with some compilers. I tested with GCC. This, the first version does not work for GCC but the second one does. Whereas in Visual C++, in Microsoft Visual Studio both these versions work. So this is this is for your information. And you can check out in the compiler you are using. Just try this out. It is good fun and this is a good compact way to design.

(Refer Slide Time: 26:29)



So to summarize, we have discussed first very key ideas in a polymorphic design in a generalization, specialization, hierarchy in C++. The inheritance hierarchy. The first concept is why destructors must be virtual at the, in the polymorphic hierarchy. And the way to make them virtual is simply to make the destructor of the root class, the base class virtual.

Then by the rule of polymorphism, all destructors will be virtual and it will properly clean up the object by calling the destructors starting from the root and will not lead to slicing. Otherwise, you will have slicing.

We have also introduced pure virtual function which are virtual functions which have signature but not a body. So that if I know that I need a function like this conceptually in a base class but I do not know how to implement that, I will be able to model that.

So any polymorphic hierarchy is likely to have a number of pure virtual functions in it at its root. Certainly, if a class has a pure virtual function, it cannot be instantiated. Because it does not have a body and or notionally the compiler knows that you are not going to, you are, you are rather saying that I am not going to instantiate any object of this class. So it is called the abstract base class.

And the last point that we have noted is, it is still possible to provide an implementation of a pure virtual function to refactor your common code into that. So with that, we close on this module. We will continue in the next module. Using all these we will start looking at how to actually do a good hierarchy-based design. Thank you very much for your attention. See you in the next module.