

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

Lecture - 43
Dynamic Binding (Polymorphism): Part III

Welcome to Module 28 of Programming in C++. We have been discussing about Static and Dynamic Binding. In this context in module 26, we discussed about casting particularly on a hierarchy up cast and down cast issues. And then in the last module, we have formally introduced a Notion of Binding. While I will invoking a function use in a pointer or a reference, how does the compiler resolve the actual function that will be invoked, whether it does it based on static strategy which will be done for non-virtual functions or it invokes involves a dynamic strategy for virtual functions. And we have seen what are the basic rules of static and dynamic binding that is engaged in C++.

And we have seen that whenever a particular class has a virtual function whether that function is inherited or that function is introduced in that class it becomes a polymorphic type. And based on that, we will be typically discussing about polymorphic hierarchies and thoroughly try to illustrate to you as to how they become very useful modelling and programming tool in terms of the object oriented paradigm.

(Refer Slide Time: 01:41)

The slide is titled "Module Objectives" and features a blue header with a logo on the left. A vertical sidebar on the left lists the following items: "Module 28", "Partha Pratim Das", "Objectives & Outline", "Virtual Destructor", "Pure Virtual Function", "Abstract Base Class", and "Summary". The main content area contains two bullet points: "• Understand why destructor must be virtual in a class hierarchy" and "• Learn to work with class hierarchy". At the bottom, it reads "NPTEL MOOCs Programming in C++" and "Partha Pratim Das" with a page number "2".

In this particular module, we will continue on that polymorphic type. And we specifically try to understand why destructors must be virtual in a polymorphic hierarchy. And we will also try to start learning as to how to work with the polymorphic hierarchy.

(Refer Slide Time: 02:03)

The slide is titled "Module Outline" and features a blue header with a logo on the left. A vertical sidebar on the left lists the following items: "Module 28", "Partha Pratim Das", "Objectives & Outline", "Virtual Destructor", "Pure Virtual Function", "Abstract Base Class", and "Summary". The main content area contains three bullet points: "• Virtual Destructor", "• Pure Virtual Function", and "• Abstract Base Class". At the bottom, it reads "NPTEL MOOCs Programming in C++" and "Partha Pratim Das" with a page number "3". A small circular portrait of the presenter is visible in the bottom left corner.

So, in a specific we have three topics to discuss which form our outline and we will be visible on the left hand side.

(Refer Slide Time: 02:12)

The slide is titled "Virtual Destructor" and shows a C++ code example. The code defines a base class B and a derived class D. Class B has a constructor that takes an integer and prints "B()", and a virtual destructor that prints "B()". Class D inherits from B, has a constructor that takes two integers and a pointer, calls the base constructor, and prints "D()", and a destructor that prints "D()" and deletes the pointer. The main function creates objects of B and D, prints them, and deletes them. The output shows the sequence of constructor and destructor calls, with a note that the destructor of D (type D) is not called.

```
#include <iostream>
using namespace std;

class B {
    int data;
public:
    B(int d) : data(d) { cout << "B()" << endl; }
    ~B() { cout << "~B()" << endl; }
    virtual void Print() { cout << data; }
};

class D : public B {
    int *ptr;
public:
    D(int d1, int d2) : B(d1), ptr(new int(d2)) { cout << "D()" << endl; }
    ~D() { cout << "~D()" << endl; delete ptr; }
    void Print() { B::Print(); cout << " " << *ptr; }
};

int main() {
    B b = new B(2);
    D d = new D(3, 4);

    b->Print(); cout << endl;
    d->Print(); cout << endl;

    delete b;
    delete d;

    return 0;
}
```

Output:

```
B()
D()
D()
3 4
~B()
~D()
Destructor of d (type D) not called!
```

So, first is a Virtual Destructor. So, let us start with an example. So, this is the example. There is a class B, the base class, which has some int type of data. There is a class D which specializes from B and it is a pointer to integer type of data. Look at the constructor, the constructor simply takes a value and assigns sets to the member. The destructor does nothing. We have put messages in the construct and destructor, so that we can trace and understand what is going on. Similarly, if I have a the derived class constructor, it takes two numbers; first it uses to construct the base part, calls a base part constructor; and the second, it uses to initialize its own pointer data, it dynamically allocates an integer with the value of D 2 and sets a pointer to the ptr member. It also has a message to say that it has happened.

Coming to the destructor of the derived class, we have a message saying that the destructor is being used. And certainly since now the object is about to be destroyed; it had a dynamic allocation in the ptr pointer, so that allocation will have to be released. So, we do a delete here. In addition to be able to see what is insight the classes, we use a use print functions and we make the print function virtual, so that it can be invoked from the

pointer and depending on the object type, it will do an appropriate print of either the data or the data and the pointer. Since, data is private here the overridden print function in D we cannot actually access data. So, this function I cannot write C out; even though data is a member of the D class, I cannot write this because data is private here, and therefore it cannot be accessed.

So, what we do is we follow a simple trick. We in turn invoke the corresponding member function in the B class. So, I directly invoke B colon-colon print, which means that it will invoke this member function print of the B class with the this pointer of the derived class D. And since this invocation actually needs this pointer of type B, what will happen is we are going up the hierarchy, so an up cast will automatically happen. So, that is what is the meaning of this. So, this is about the code that you get to see you can go through further literature.

(Refer Slide Time: 05:02)

The slide, titled "Virtual Destructor", displays the following C++ code:

```
#include <iostream>
using namespace std;

class B {
    int data_;
public:
    B(int d) : data_(d) { cout << "B()" << endl; }
    ~B() { cout << "~B()" << endl; }
    virtual void Print() { cout << data_; }
};

class D : public B {
    int *ptr_;
public:
    D(int d1, int d2) : B(d1), ptr_(new int(d2)) { cout << "D()" << endl; }
    ~D() { cout << "~D()" << endl; delete ptr_; }
    void Print() { B::Print(); cout << " " << ptr_; }
};

int main() {
    B *p = new B(2);
    B *q = new B(3, 4);

    p->Print(); cout << endl;
    q->Print(); cout << endl;

    delete p;
    delete q;

    return 0;
}
```

The output of the program is shown on the right side of the slide:

```
Output:
B()
B()
~B()
~B()
2
3 4
~D()
~B()
Destructor of D (type D) not called!
```

The slide also includes a navigation menu on the left with items like "Module 20", "Partha Pratim Das", "Destructor & Destruct", "Virtual Destructor", "Pure Virtual Function", "Abstract Base Class", and "Summary". At the bottom, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

And now then in the application we create a B class object, we create a D class object put them to two pointers, and we print. And certainly if we do that from the creation, this is the construction of the B class object that is pointed to by p. These two come from the construction of the D class object because first B part of the D class object is constructed then D part is, then you do print you see 2, 3, 5 as you see this member, so up to this

point there is nothing interesting. So, forget about this was all setting up. And then we have the delete for these pointers. We want to basically delete the object. So, if I delete this what will happen this should invoke my B class destructor which it does, so that that is pretty fine. Finally, I invoke the destructor of invoke delete on q which should invoke the destructor of D and see what I get. I do not get the destructor of D, I do not get this printed fine. So, why is it that?

Now if we apply our mind and think about what is going on in terms of the binding see this destructor, destructors are in a way are member functions which are defined here as non-virtual. So, what will happen when I want to call delete on q then it has to decide based on what is the type of q, is that is what the compiler we has to see. So, compiler wants to decide based on q and q is of type B. So, it has to go to class B and decide that what destructor should be called it knows there is a destructor and it finds that this destructor is non-virtual. So, it calls it done. The destructor of D will never get called, because actually what we needed to be able to destroy this object, we needed the call to land in the destructor of D which in turn would call the destructor of B and do things right that is how it happens. So, here the call landed at a wrong place. So, this is the basic problem of a destruction way for which we need to make the destructor virtual.

(Refer Slide Time: 07:40)

Virtual Destructor

```

#include <iostream>
using namespace std;

class B {
    int data_;
public:
    B(int d) :data_(d) { cout << "B()" << endl; }
    ~B() { cout << "~B()" << endl; }
    virtual void Print() { cout << data_; }
};

class D: public B {
    int *ptr_;
public:
    D(int d1, int d2) :B(d1), ptr_(new int(d2)) { cout << "D()" << endl; }
    ~D() { cout << "~D()" << endl; delete ptr_; }
    void Print() { B::Print(); cout << " " << *ptr_; }
};

int main() {
    B *p = new B(2);
    D *q = new D(3, 5);

    p->Print(); cout << endl;
    q->Print(); cout << endl;

    delete p;
    delete q;

    return 0;
}

```

Output:
B()
B()
D()
2
3 5
~B()
~B()
Destructor of d (type B) not called!

NPTEL MOOCs Programming in C++ Partha Pratim Das 4

So, it is the fix is very simple. All that you do is write the word virtual between in front of the destructor of p; up to this point there is nothing different, there is nothing different in the expected behavior and the output. Now what happens in if you to delete q, q is of type B, so it goes to class B, the compiler sets it to class B. In class B, you find that the destructor is virtual which means now the dispatch has to happen not based on the type of the pointer, but the type of the object it is pointing to. And what is the type of the object, what is a dynamic type the dynamic type is D. So, when we do this, actually this gets involved this does not this invokes the destructor of D, which in turn will invoke the destructor of B. So, you see that the destructor of D is invoked this, this gets deleted and then at this point the destructor of B is invoked. So, you get to see the B has got invoked. By the rule of polymorphism hierarchy that we have seen earlier, since the destructor of B is virtual by inheritance the destructor of D is also virtual. I do not need to write that it is virtual.

(Refer Slide Time: 08:58)

Virtual Destructor

```

#include <iostream>
using namespace std;

class B {
    int data_;
public:
    B(int d) : data_(d) { cout << "B()" << endl; }
    virtual ~B() { cout << "~B()" << endl; } // Destructor made virtual
    virtual void Print() { cout << data_; }
};

class D : public B {
    int *ptr_;
public:
    D(int d1, int d2) : B(d1), ptr_(new int(d2)) { cout << "D()" << endl; }
    ~D() { cout << "~D()" << endl; delete ptr_; }
    void Print() { B::Print(); cout << " " << *ptr_; }
};

int main() {
    B *p = new B(2);
    B *q = new D(3, 1);

    p->Print(); cout << endl;
    q->Print(); cout << endl;

    delete p;
    delete q;

    return 0;
}

```

Handwritten notes: A diagram shows a box labeled 'q' pointing to a box labeled 'B'. Another box contains a question mark. The word 'Slice' is written in blue.

Output:

```

B()
D()
D()
2
3 1
~B()
~D()
~B()
Destructor of B (type B) is called!

```

NPTEL MOOCs Programming in C++ Partha Pratim Das 5

But without that the major problem that I was getting into that if this virtual were not there then this was not getting called; and the consequence of that, the pointer that was created in the D object was not being released. So, I can say that when I have an object from a derived class, which has a base class part. And I am holding that from a base class pointer, if the destructor is not virtual then it is simply looking at the base class part of

the pointer. So, it only deletes this part of the object. It does not delete the remaining part of the object. So, we say that the destructor tries to slice the object; it tries to chop off that object at this point, and only does releases one part. So, this is something very, very dangerous because we will have certain you know chopped off sliced object remaining in the system, and the system consistencies will go here where. So, if you are on hierarchy then make sure that the destructor in the base class will would be a virtual function.

And now you can understand the moment you make the destructor of the base class virtual, because otherwise this whole mechanism will not work, the whole cleanup consistency will not work. But as soon as you make the destructor of the base class virtual, that means, that the base class become polymorphic irrespective of whether you have another polymorphic function like this or not; like here, we had another polymorphic function. But even if we did not have that, even if we did not have this print function, the moment I make this virtual class B becomes a polymorphic type, and therefore, since it is the root the all classes that are derived from that directly or indirectly all become polymorphic. So, the whole hierarchy becomes polymorphic. So, this is one of the reasons that I had mentioned earlier that if I have a hierarchy then it does not make sense to make it non-polymorphic, it is of not much of interest.

(Refer Slide Time: 11:05)

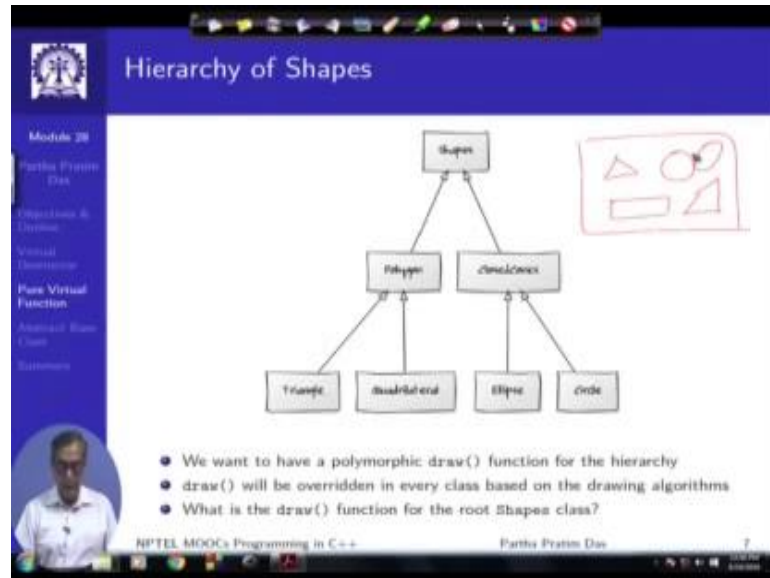
Virtual Destructor

- If the destructor is not virtual in a polymorphic hierarchy, it leads to **Slicing**
- **Destructor must be declared virtual in the base class**

NPTEL MOOCs Programming in C++ Partha Pratim Das

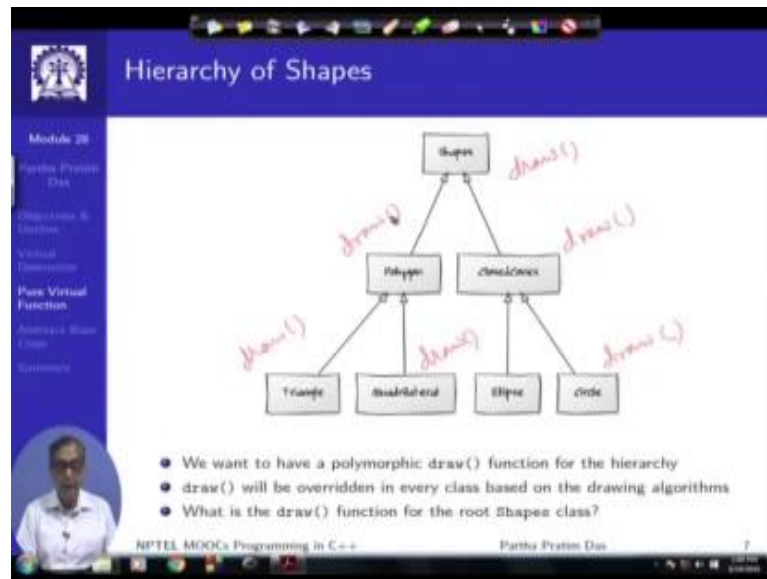
So, what we learn is if the destructor is not virtual in a polymorphic hierarchy it leads to slicing. So, destructor must be declared virtual in the base class always.

(Refer Slide Time: 11:16)



Let us look into some other interesting problem. Let us consider a hierarchy of shapes and our basic objective is we are trying to build a graphic system where these shapes can be there on the canvas. So, it is as if there is a canvas, and on that canvas, I want to draw objects of this shape different kinds of objects of this shape and so on, so that is a in deep objective. And there could be several other graphic things that we should be doing, but based on that we create a hierarchy. So, there is a shape, there are two kinds of shape that is basically polygonal shapes and closed curves like closed conics, there could be others also. In polygon, we have triangle, quadrilateral; there could be many more in closed conics, we have ellipse, circle and so on. So, this is just a simple.

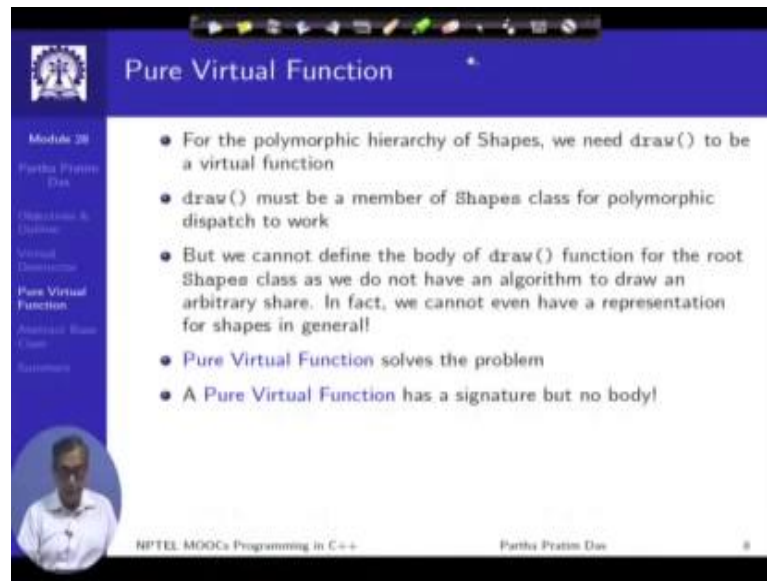
(Refer Slide Time: 12:12)



And what we want to do is we want to have a polymorphic draw function here, because the way you draw a triangle say here drawing a triangle is basically drawing three segments of line. But drawing a circle is a pretty involved operation it use you some equation some algorithm drawing an ellipse is even more complicated, but in contrast drawing a quadrilateral would be. So, we want to have a hierarchy, where we have a draw function everywhere, so that we can we can simply we may not really need to bother about which particular object we have we will just hold a pointed to that object and call draw. And with that, we should be able to land by the use of dynamic binding or you should be able to land with the right draw function of the corresponding class, so that is what we want to do.

So, you want to have a polymorphic draw function for the hierarchy the draw will be overridden in every class based on the drawing algorithm. Now we get stuck because if I have to have this then certainly I need the draw function in the route, I need a draw function in the shape class. So, but the question is if I just a shape can you draw it, it is not possible to draw in arbitrary shape. In fact, which is words is for an arbitrary shape we cannot even represent it. So, we lead need some mathematical curves, some definitions to be able to represent shape, so that is the genesis of the problem that we are trying to address.

(Refer Slide Time: 13:39)



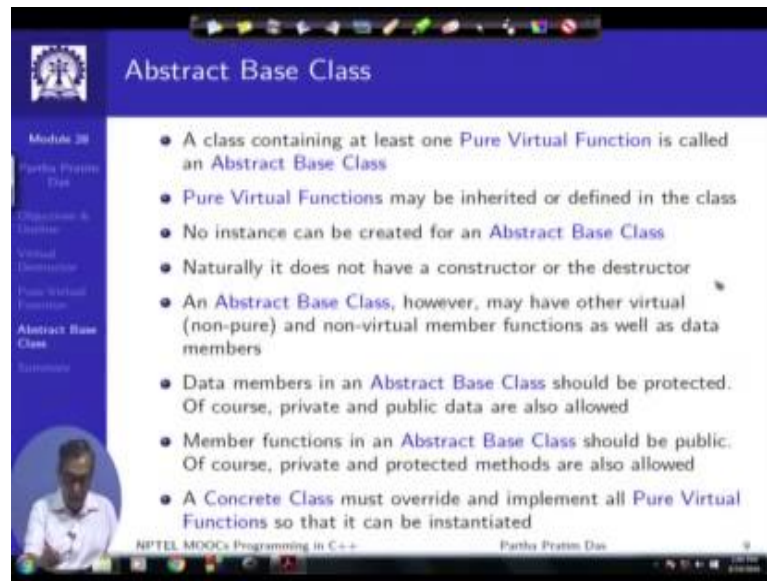
The slide is titled "Pure Virtual Function" and features a blue header with a navigation bar at the top. On the left side, there is a vertical menu with the following items: "Module 28", "Partha Pratim Das", "Introduction to C++", "Virtual Dispatch", "Pure Virtual Function", "Abstract Base Class", and "Namespace". A small circular portrait of the speaker is located at the bottom left of the slide. The main content area contains a list of five bullet points:

- For the polymorphic hierarchy of Shapes, we need `draw()` to be a virtual function
- `draw()` must be a member of Shapes class for polymorphic dispatch to work
- But we cannot define the body of `draw()` function for the root Shapes class as we do not have an algorithm to draw an arbitrary shape. In fact, we cannot even have a representation for shapes in general!
- Pure Virtual Function solves the problem
- A Pure Virtual Function has a signature but no body!

At the bottom of the slide, the text "NPTEL MOOCs Programming in C++" and "Partha Pratim Das" is visible, along with a small number "8" in the bottom right corner.

So, for polymorphic hierarchy of shapes we need a draw to be a virtual function, draw must be a member of shapes class, so that the polymorphic dispatch can work. So that I can have a pointer to the shape class type which can hold actual object instances of any of the different types of object that exist on the hierarchy of a triangle, of a rectangle, of anything. And we should be able to just from the pointer to shape, we should be able to just invoke draw and it should end up polymorphically dispatch to the particular draw function of the particular class of which the object I am pointing too. But certainly we cannot write the body of the draw function in the shape class, because we do not know the algorithm. So, for that a new notion is introduced which is called the pure virtual function. And a pure virtual function is characterized by it has a signature, but no body. We will see some exceptions to that, but the basic notion is it has the signature, so that I can call it, but it does not have a body because I cannot implement it; sounds weird, but let us see how does that fit in.

(Refer Slide Time: 14:49)



The slide is titled "Abstract Base Class" and features a list of seven bullet points. On the left side, there is a vertical navigation menu with the following items: "Module 20", "Partha Pratim Das", "Classroom & Online", "Virtual Destructor", "Pure Virtual Function", "Abstract Base Class", and "Summary". The "Abstract Base Class" item is highlighted. At the bottom left of the slide, there is a small circular inset image of a man in a white shirt. The bottom of the slide contains the text "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

- A class containing at least one Pure Virtual Function is called an Abstract Base Class
- Pure Virtual Functions may be inherited or defined in the class
- No instance can be created for an Abstract Base Class
- Naturally it does not have a constructor or the destructor
- An Abstract Base Class, however, may have other virtual (non-pure) and non-virtual member functions as well as data members
- Data members in an Abstract Base Class should be protected. Of course, private and public data are also allowed
- Member functions in an Abstract Base Class should be public. Of course, private and protected methods are also allowed
- A Concrete Class must override and implement all Pure Virtual Functions so that it can be instantiated

Now, if I have a pure virtual function in a class, at least one, then I call that to be an abstract base class. Now this virtual function - pure virtual function may be inherited or might may be defined in the class, it does not matter which I get. But if a class has a pure virtual function, it is a abstract base class, what does that mean, what is abstract about this. So, the third point says it all. No instance can be created for a abstract base class; an abstract base has everything, but it cannot create an object instance. Why it cannot create an object instance. Conceptually, it is very clear because if I have a pure virtual function of which I just know the signature, I do not know the body, I do not know the algorithm, if I could create an object of the abstract base class then using that object I could invoke the pure virtual function which is a member of that abstract base class, but I do not have an algorithm for that so what do I execute. So, the way we restrict we say that the no instance can be created. Since, no instance can be created an abstract base class does not have a constructor or a destructor.

But it can have other virtual functions, it can have more pure virtual functions, it can have non-pure other virtual function, it may have non-virtual functions, it may have a data members and so on and so forth. Naturally, if it has data members then we would expect the data members to be protected, but it is possible that you have them as private and public I am saying it should be preferably protected because you do not expect an

instance of this class. So, if you do not expect an instance of this class, you do not expect that the class itself is doing some computation using the data members. So, the data members are there, so that the derive classes can use them. So, it is reasonable that they should be protected. Of course, you can meet them public, but that violates the basic encapsulation rule.

Similarly, the member functions of the class should normally be public, because certainly we will not have an instance, so if you do not have an instance then certainly it is a derive classes the hierarchy that we are going to use these functions. But, it is possible that you can have private or protected methods as well; and do different kind of tricks to hide the encapsulation. And since, we are talking about an abstract base class in contrast there are concrete class that must override an implement all pure virtual functions. Because now if you derive from, if you specialize from an abstract base class, naturally in the derived class, you do not have a default you will inherit the function, you will inherit the pure virtual function. But again in the derive class, you do not have an option of invoking that function, because that function by itself does not have a body. So, you will have to be able to create objects finally, you must have some derived classes which override the pure virtual function as non-pure virtual function, and implements them that is provides the body, such classes will be known as concrete classes. So, certainly for concrete classes, the instances can be created. This was a lot of set of rules.

(Refer Slide Time: 18:17)

```
#include <iostream>
using namespace std;

class Shape { public:
    virtual void draw() = 0; // Pure Virtual Function // Abstract Base Class
};

class Polygon: public Shape { // Concrete Class
    void draw() { cout << "Polygon: Draw by Triangulation" << endl; }
};

class ClosedConics : public Shape { // Abstract Base Class
    // draw() inherited - Pure Virtual
};

class Triangle : public Polygon { public: // Concrete Class
    void draw() { cout << "Triangle: Draw by Lines" << endl; }
};

class Quadrilateral : public Polygon { public: // Concrete Class
    void draw() { cout << "Quadrilateral: Draw by Lines" << endl; }
};

class Circle : public ClosedConics { public: // Concrete Class
    void draw() { cout << "Circle: Draw by Bresenham Algorithm" << endl; }
};

class Ellipse : public ClosedConics { public: // Concrete Class
    void draw() { cout << "Ellipse: Draw by ..." << endl; }
};

int main() {
    Shape *arr[] = { new Triangle, new Quadrilateral, new Circle, new Ellipse };
    for (int i = 0; i < sizeof(arr) / sizeof(Shape *); ++i) arr[i]->draw();
    // ...
    return 0;
}
```

So, let's go into an example and try to understand. So, I have shown the hierarchy of shapes. At this point, you have feeling little lost in terms of the hierarchy, it will be good to take a print out of that slide and keep it by the side, so that you can quickly refer to it. I did not have enough space to show it together, but it is conceptually very clear at the root class, base class in shapes. So, it has one virtual function draw, which does not take anything, does not return anything, but it is a virtual function. And we use this special notation of saying the assignment symbol then zero, which says that it is a pure virtual function. So, if you do not have this, then it is just a virtual function, but when you put this, this is called a pure virtual function which means that it is not expected now that there you will need an implementation for this function for the whole code to run. And since I have a pure virtual function, this shape becomes an abstract base class that is the basic.

Now, let us go down the shape had two specializations; one is polygon, and one is closed conics. Now in polygon, as I inherit we override the draw function, and polygon now has an implementation. Of course, this in a print message is just indicative of the implementation. So, possibly you implement an algorithm that the polygon is triangulated, and every triangle is drawn possible that way whatever. But, the point is we have now inherited and implemented the pure virtual function. So, this becomes an

ordinary virtual function now. And therefore, the class becomes a concrete class that is it will be possible to create instances of the polygon class. Look at the other class specialize from shapes, which is closed conics. In this, if it a closed conics, if you just say it a closed conic, then it could be a circle, it could be eclipse. So, I do not really know how to draw them through a generic algorithm. So, the closed conic also does not have a draw function. So, what it does, it does not override, it does not just have any signature of the draw function. So, what will happen, since this is a specialization of shapes from the shapes, it inherits the draw function, which is purely virtual in shapes, therefore, it is purely virtual in closed conics also, because no implementation has been provided. So, closed conic also continue to be the abstract base class.

Then you have triangle specializing from polygon, quadrilateral specializing from polygon, as they specialize they have their own implementation. So, they are inheriting and overriding the function, so these are the specific draw functions for triangle quadrilateral base of classes. And then in terms of the other side, in terms of the closed conics, circle specializes from closed conics, ellipse specialize from closed conics, and they override the draw function with the circle specific drawing algorithm or ellipses specific drawing algorithm and so on. So, they become concrete class. So, now, as it turns out that the route is an abstract base class, the closed conics are the abstract base class, because both of them have one pure virtual function draw. All other five classes have become concrete, because each one of them have introduced an implementation have promise an implementation of the inherited draw function, which it has overridden.

(Refer Slide Time: 22:15)

```
#include <iostream>
using namespace std;

class Shape { public:
    virtual void draw() = 0; // Pure Virtual Function // Abstract Base Class
};

class Polygon : public Shape { // Concrete Class
    void draw() { cout << "Polygon: Draw by Triangulation" << endl; }
};

class ClosedConvex : public Shape { // Abstract Base Class
    // draw() inherited, Pure Virtual
};

class Triangle : public Polygon { public: // Concrete Class
    void draw() { cout << "Triangle: Draw by Lines" << endl; }
};

class Quadrilateral : public Polygon { public: // Concrete Class
    void draw() { cout << "Quadrilateral: Draw by Lines" << endl; }
};

class Circle : public ClosedConvex { public: // Concrete Class
    void draw() { cout << "Circle: Draw by Brushstrokes Algorithm" << endl; }
};

class Ellipse : public ClosedConvex { public: // Concrete Class
    void draw() { cout << "Ellipse: Draw by ..." << endl; }
};

int main() {
    Shape *arr[] = { new Triangle, new Quadrilateral, new Circle, new Ellipse };
    for (int i = 0; i < sizeof(arr) / sizeof(Shape *); ++i) arr[i]->draw();
    // ...
    return 0;
}
// NPTEL MOOCs Programming in C++
// Partha Pratim Das
```

So, with this now, if we look at the way we could create the canvas. So, the canvas say is now an array of a pointer to shape, so these are pointer to shape. And I have different kinds of shape, I have a triangle, I have a quadrilateral, I have a circle, I have an ellipse. So, I create all of these of course, in actual code, there will be lot of parameters and all that, but I am just demonstrating the whole process. So, I create pointers to variety of different objects; each one of them is eventually a specialization of shape. So, when I create a dynamically create a triangle, I get a pointer of a triangle type. And since triangle is a polygon, and a polygon is a shape, so I can up cast this pointer to the shapes pointer I can do that for each one of these pointers and they are all shape.

(Refer Slide Time: 23:14)

```
Shape Hierarchy

#include <iostream>
using namespace std;

class Shapes { public:
    virtual void draw() = 0; // Pure Virtual Function // Abstract Base Class
};

class Polygon: public Shapes { // Concrete Class
    void draw() { cout << "Polygon: Draw by Triangulation" << endl; }
};

class ClosedConvex : public Shapes { // Abstract Base Class
    // draw() inherited - Pure Virtual
};

class Triangle : public Polygon { public: // Concrete Class
    void draw() { cout << "Triangle: Draw by Lines" << endl; }
};

class Quadrilateral : public Polygon { public: // Concrete Class
    void draw() { cout << "Quadrilateral: Draw by Lines" << endl; }
};

class Circle : public ClosedConvex { public: // Concrete Class
    void draw() { cout << "Circle: Draw by Brachudon Algorithm" << endl; }
};

class Ellipse : public ClosedConvex { public: // Concrete Class
    void draw() { cout << "Ellipse: Draw by ..." << endl; }
};

int main() {
    Shapes *arr[] = { new Triangle, new Quadrilateral, new Circle, new Ellipse };
    for (int i = 0; i < (sizeof(arr) / sizeof(Shapes *)); ++i) arr[i]->draw();
    return 0;
}

NPTEL MOOCs Programming in C++ Partha Pratim Das 10
```

So, now, I have uniform array of pointers of shape through which I can actually invoke methods of all of these functions. Then I simply to draw I simply write a for loop, but I start from the zero, this code must be familiar with you this basically tries to find out how many elements are there in the array of shapes and you go over them. So, if we have the ieth element arr_i this of type shape pointer. So, on that I invoke draw function. So, when i will be 0, at it will actually have the value of new triangle. So, this will this call will take me the compiler will map it here first, because arr is of shape pointer type and then it finds that this is virtual, so that is to be dynamic dispatch. So, the dispatch will happen based on the actual object it is pointing to. For 0, this will be triangle. So, this particular function will get called. Then when becomes, so this for 0, for 1, this is a quadrilateral. So, the same dispatch will happen and this function will get called and so on.

(Refer Slide Time: 24:23)

The slide is titled "Shape Hierarchy" and features a blue header with a logo on the left. A sidebar on the left contains navigation links: "Module 28", "Partha Prasad Das", "Abstract Base Class", and "Summary". The main content area displays C++ code for a program that creates an array of shapes and calls a draw method on each. The output shows the draw method being called for a triangle, quadrilateral, circle, and ellipse. A red bullet point indicates that instances for the abstract classes "Shapes" and "ClosedConics" cannot be created. The footer includes "NPTEL MOOCs Programming in C++", "Partha Prasad Das", and the slide number "11".

```
int main() {
    Shapes *arr[] = { new Triangle, new Quadrilateral, new Circle, new Ellipse };
    for (int i = 0; i < sizeof(arr) / sizeof(Shapes *); ++i) arr[i]->Draw();
    // ...
    return 0;
}
```

Output:

Triangle: Draw by Lines
Quadrilateral: Draw by Lines
Circle: Draw by Bresenham Algorithm
Ellipse: Draw by ...

- Instances for class Shapes and class ClosedConics cannot be created

So, if we try to quickly look at the output then for this code, the output will look like this. So, first was the new triangle, so in the output you see. The triangle draw has been called then the quadrilateral draw has been called, then the circle draw has been called, the ellipse draw has been called and so on. And all that we could do we needed to do is just call the function at the route, the polymorphic function at the route. And because of the ability to introduce pure virtual function and abstract base classes, we have been able to do this whole module together; otherwise, without that there is no way to generalize between the polygons and the closed conics into a same hierarchy we would have required actually closed conics itself is cannot be generalize in that way.

So, this is the basic advantage, the basic way you start using the polymorphic hierarchy to be able to write codes which are extremely compact in nature and we will see lot of other benefits like extensibility and so on. And but certainly for this case the instances of the shape and the closed conic classes cannot be created because they are abstract base class.

(Refer Slide Time: 25:35)

Shape Hierarchy:
A Pure Virtual Function may have a body!

```
#include <iostream>
using namespace std;
class Shape { public:
    virtual void draw() = 0 // Pure Virtual Function
    { cout << "Shape: Tail Hook" << endl; }
};
class Polygon: public Shape {
    void draw() { Shape::draw(); cout << "Polygon: Draw by Triangulation" << endl; }
};
class ClosedConvex : public Shape {
    // draw() inherited - Pure Virtual
};
class Triangle : public Polygon { public:
    void draw() { Shape::draw(); cout << "Triangle: Draw by Lines" << endl; }
};
class Quadrilateral : public Polygon { public:
    void draw() { Shape::draw(); cout << "Quadrilateral: Draw by Lines" << endl; }
};
class Circle : public ClosedConvex { public:
    void draw() { Shape::draw(); cout << "Circle: Draw by Brahmaghosa Algorithm" << endl; }
};
class Ellipse : public ClosedConvex { public:
    void draw() { Shape::draw(); cout << "Ellipse: Draw by ..." << endl; }
};
int main() {
    Shape *arr[] = { new Triangle, new Quadrilateral, new Circle, new Ellipse };
    for (int i = 0; i < sizeof(arr) / sizeof(Shape *); ++i) arr[i]->draw();
    // ...
    return 0;
}
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 12

Now, one more point that you should note is if I define a function - virtual function to be pure, then the class becomes abstract by definition. And you cannot create an instance, but the fact that a function is purely virtual says that an implementation for that function is not necessary, but it does not say that I cannot have an implementation. A function would be purely virtual, and in addition, I could have an implementation for that. Now if I provide the implementation, also its purity does not go away, because I am saying that it is it is a pure function. So, it will continue to remain pure, which means that no the class will continue to be abstract and no instance of that class can be created, but the function has an implementation which can be used. Now why would I need that, certainly the reason I will need that is for a code reuse for code factoring because a certainly the pure virtual function are at the route.

(Refer Slide Time: 26:50)

```
#include <iostream>
using namespace std;
class Shape { public:
    virtual void draw() = 0 // Pure Virtual Function
    { cout << "Shape: Init Brush" << endl; };
};
class Polygon: public Shape {
    void draw() { Shape::draw(); cout << "Polygon: Draw by Triangulation" << endl; };
};
class ClosedConvex : public Shape {
    // draw() inherited - Pure Virtual
};
class Triangle : public Polygon { public:
    void draw() { Shape::draw(); cout << "Triangle: Draw by Lines" << endl; };
};
class Quadrilateral : public Polygon { public:
    void draw() { Shape::draw(); cout << "Quadrilateral: Draw by Lines" << endl; };
};
class Circle : public ClosedConvex { public:
    void draw() { Shape::draw(); cout << "Circle: Draw by Bresenham Algorithm" << endl; };
};
class Ellipse : public ClosedConvex { public:
    void draw() { Shape::draw(); cout << "Ellipse: Draw by ..." << endl; };
};
int main() {
    Shape *arr[] = { new Triangle, new Quadrilateral, new Circle, new Ellipse };
    for (int i = 0; i < sizeof(arr) / sizeof(Shape *); ++i) arr[i]->draw();
    // ...
    return 0;
}
```

So, think about this; suppose in the first case, I needed to do a draw for a triangle. Now if I need to do a draw triangle, then certainly if you think about the drop, now there is there is one part is a triangle algorithm which comes here. But to be able to draw a triangle, I need to pick up a brush, I need to decide the color, I need to know the thickness of the brush, I need to know the position, and all those. Now that part of the algorithm does it change between whether I am drawing a triangle or I am drawing a quadrilateral, drawing a quadrilateral also I need to pick a brush, I need to pick a color, I need to pick a thickness. In drawing a circle also, I will need to do that; in drawing an ellipse also, I need to do that. So, all of these will be a common part of code which goes into all of this different draw functions. What would be best is I take that common part of the code out and put it in the route.

So, what I simply need to do is when I am implementing draw function for the triangle, I simply invoke the draw function of shape. We have already seen this in earlier example, in the print example we saw this that you can always call the inheritable function from your parent directly by using the class of a parent. I could have made it propagate through polygon also that is I could have done it that this actually is a polygon colon-draw. So that it calls this and then that calls shape it could have been like that, but for the draw I did not find any reasonable logic as to what can triangle and quadrilateral

share in common form the draw of the polygon function, which actually has to draw. But certainly it has a lot that it can share from the draw of the shape function which can do all the required brush initialization and stuff like that before drawing a particular geometric object.

So, it is possible that pure virtual functions may have a body, and that does not take away the virtuality of it. So, pure virtual functions, I would am particularly emphasizing because I have seen quite a few textbooks, do not clarify this point. They just say that the pure virtual functions do not have a body, but I am just clarifying that it is not the pure virtual functions do not have a body; pure virtual functions may not have a body. They can if they want, but the purity lies in terms of whether you define it has a pure and that will lead the corresponding class to be an abstract base class.

(Refer Slide Time: 29:29)

Shape Hierarchy

```
int main() {
    Shapes *arr[] = { new Triangle, new Quadrilateral, new Circle, new Ellipse };
    for (int i = 0; i < sizeof(arr) / sizeof(Shapes *); ++i) arr[i]->Draw();
    // ...
    return 0;
}
```

Output:

Shapes: Init Brush
Triangle: Draw by Lines
Shapes: Init Brush
Quadrilateral: Draw by Lines
Shapes: Init Brush
Circle: Draw by Bresenham Algorithm
Shapes: Init Brush
Ellipse: Draw by ...

• Instances for class Shapes and class ClosedConics cannot be created

NPTEL MOOCs Programming in C++ Partha Pratim Das 13

So, if I do this then naturally my earlier output will change. And now for the same code, I have I am basically when I am here with index 0 - the arr0 pointed draw is actually calling for this function, which is calling the draw function of triangle which in turn is calling the draw function of shape. So, these two, first the draw function of shape executes and then that of because it has been called that way, it is not by structure. Similarly, for quadrilateral, similarly for circle, for ellipse, so every time you first call

this. So, this is one of the though it is not the only way to use the implementation of a polymorphic function of a pure virtual function, but this is one of the ways where you might use the implementation for a pure virtual function as well. But of course, that does not change the abstract notion of the classes, and shape and class closed conics continue to be abstract, and no instances can be created for them.

(Refer Slide Time: 30:35)

The image shows a presentation slide with a blue header and a white content area. The header contains the text 'Module Summary' and a small logo on the left. The content area lists three bullet points: 'Discussed why destructors must be virtual', 'Introduced Pure Virtual Functions', and 'Introduced Abstract Base Class'. On the left side, there is a vertical navigation menu with items: 'Module 28', 'Partha Pratim Das', 'Object-Oriented & Classes', 'Virtual Destructors', 'Pure Virtual Functions', 'Abstract Base Class', and 'Summary'. At the bottom left, there is a small circular video inset showing a man in a white shirt. At the bottom, there is footer text: 'NPTEL MOOC - Programming in C++', 'Partha Pratim Das', and '14'.

So, to conclude, we have discussed about why destructors must be virtual. And to be able to work on the polymorphic hierarchy, we have introduced pure virtual functions and introduced the notion of abstract of base class. In the next module, we will take up more examples to show how these tools can be used to actually do certain design and code processing on the class hierarchy.