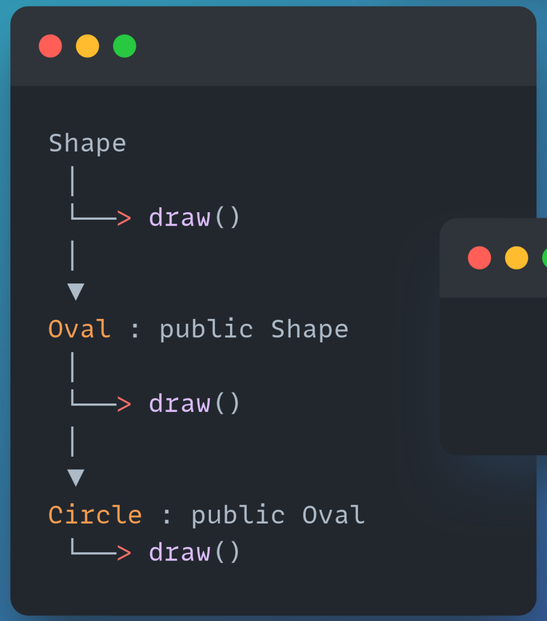
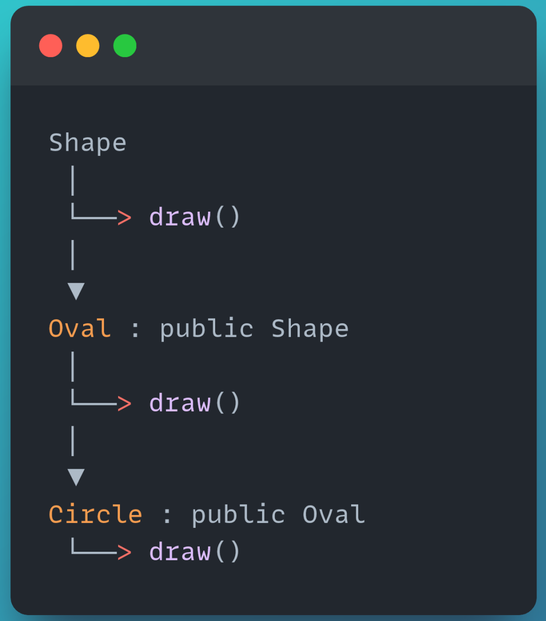
Polymorphism: The basics

```
/*

    . Topics:
        . Showing static binding in action in preparation for polymorphism.
        . We would want to manage derived objects through base class pointers and references.
        . But when we call the draw() function, we want the derived class's draw() to be called.
        . This is not the default behavior in C++.
        . We will see how to achieve this with polymorphism, in a later lecture.

*/
```



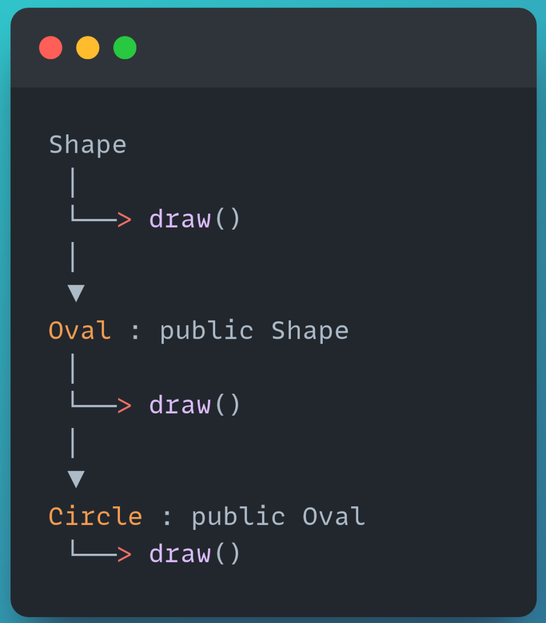

```
std::unique_ptr<Circle> circle = std::make_unique<Circle>(7.0, "Circle Shape");
circle→draw();
```

Polymorphism: The basics - static binding



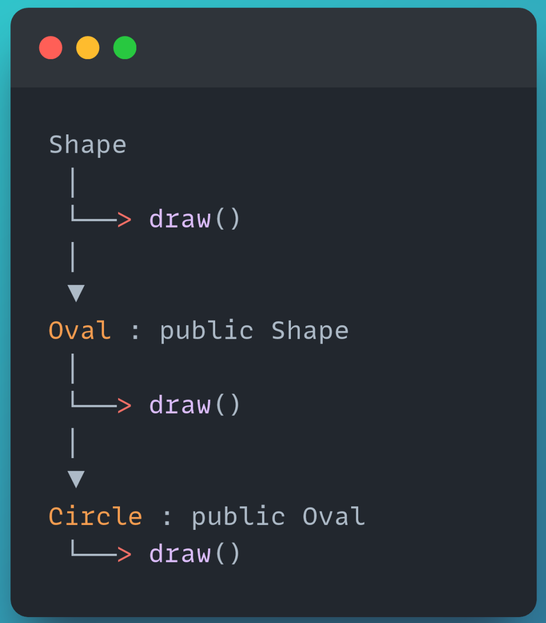

```cpp
// Shape class
export class Shape {
public:
    Shape() = default;
    Shape(std::string_view description) : m_description(description) {}

    void draw() const {
        fmt::println("Shape::draw() called. Drawing {}", m_description);
    }

protected:
    std::string m_description{ "" };
};
```

Polymorphism: The basics - static binding



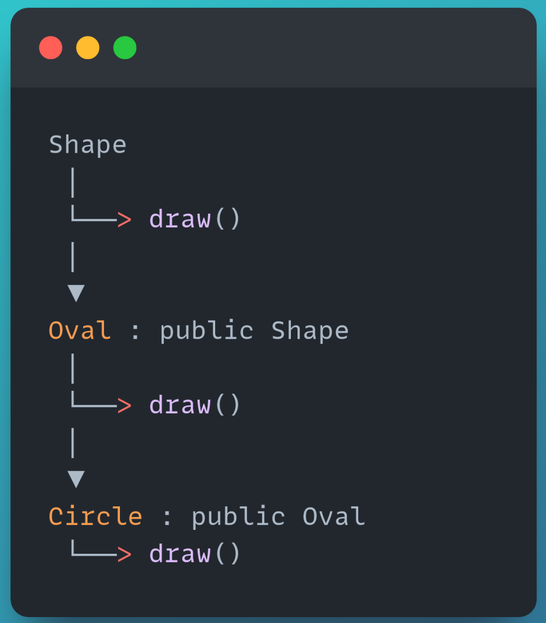

```cpp
// Oval class
export class Oval : public Shape {
public:
    Oval(double x_radius, double y_radius, std::string_view description)
        : Shape(description), m_x_radius(x_radius), m_y_radius(y_radius) {}

    void draw() const {
        fmt::println("Oval::draw() called. Drawing {} with m_x_radius: {} and m_y_radius: {}",
                     m_description, m_x_radius, m_y_radius);
    }

protected:
    double m_x_radius{ 0.0 };
    double m_y_radius{ 0.0 };
};
```

Polymorphism: The basics - static binding



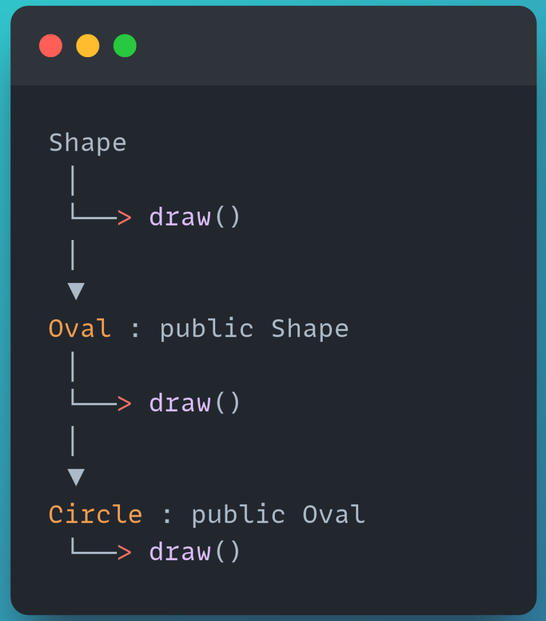

```cpp
// Circle class
export class Circle : public Oval {
public:
    Circle(double radius, std::string_view description)
        : Oval(radius, radius, description) {}

    void draw() const {
        fmt::println("Circle::draw() called. Drawing {} with radius: {}", m_description, m_x_radius);
    }
};
```

Polymorphism: The basics - static binding




```cpp
//Trying it out
// Creating instances of Shape, Oval, and Circle
poly_1::Shape shape("Generic Shape");
poly_1::Oval oval(5.0, 3.0, "Oval Shape");
poly_1::Circle circle(7.0, "Circle Shape");

// Managing objects via pointers
poly_1::Shape* shape_ptr = &shape;
poly_1::Shape* oval_ptr = &oval;    // Pointer to base class, but points to Oval object
poly_1::Shape* circle_ptr = &circle; // Pointer to base class, but points to Circle object

fmt::println("Calling draw() using Shape pointer (static binding):");
shape_ptr->draw();   // Calls Shape's draw() (as expected)
oval_ptr->draw();    // Still calls Shape's draw(), static binding
circle_ptr->draw();  // Still calls Shape's draw(), static binding
```

Polymorphism: The basics - static binding

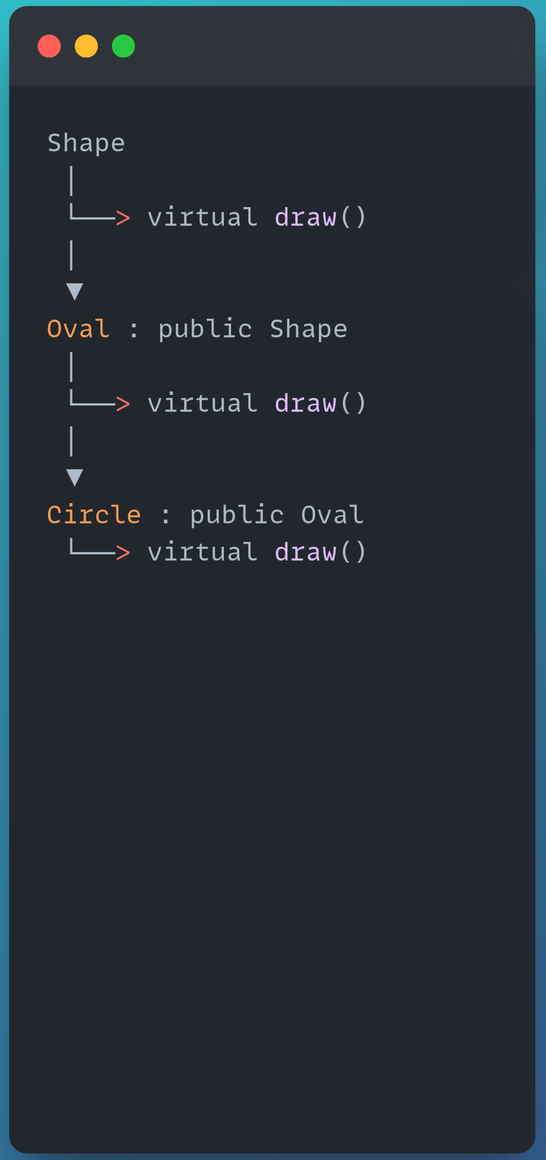



```cpp
namespace poly_1{

  //Helper functions for drawing.
  export void draw_shape(Shape* shape){
      shape→draw();
  }
  export void draw_shape(Shape& shape){
      shape.draw();
  }
}

int main(){
  //Trying it out
  // Managing objects via references
  poly_1::Shape& shape_ref = shape;
  poly_1::Shape& oval_ref = oval;    // Reference to base class, refers to Oval object
  poly_1::Shape& circle_ref = circle; // Reference to base class, refers to Circle object

  fmt::println("\nCalling draw() using Shape reference (static binding):");
  shape_ref.draw();   // Calls Shape's draw() (as expected)
  oval_ref.draw();    // Still calls Shape's draw(), static binding
  circle_ref.draw();  // Still calls Shape's draw(), static binding

  //Calling the draw functions
  fmt::println("\nCalling the draw functions");
  poly_1::draw_shape(&circle);
  poly_1::draw_shape(circle);

  }
```

Demo time!

```
Shape
  |
  └──> draw()
  |
  ▼
Oval : public Shape
  |
  └──> draw()
  |
  ▼
Circle : public Oval
  └──> draw()
```

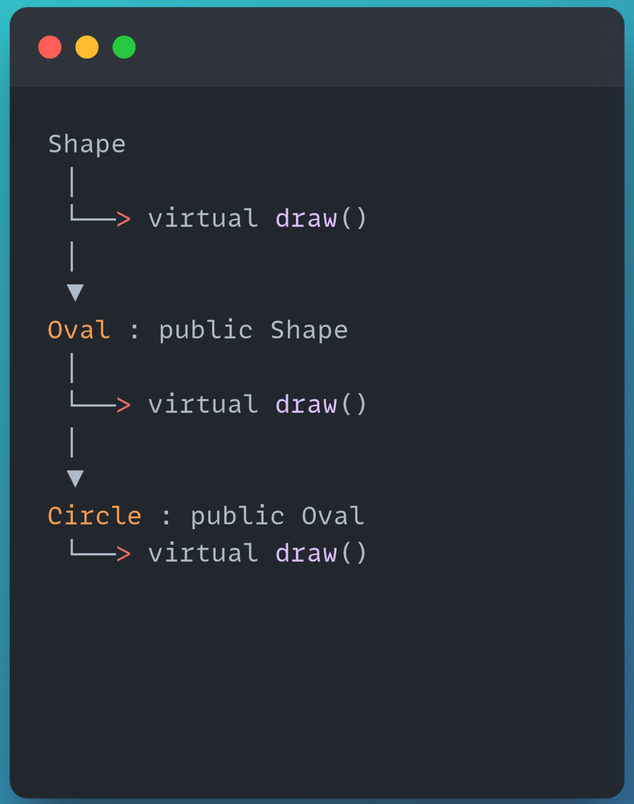
Polymorphism with virtual functions




```cpp
std::unique_ptr<Circle> circle = std::make_unique<Circle>(7.0, "Circle Shape");
circle->draw();
```

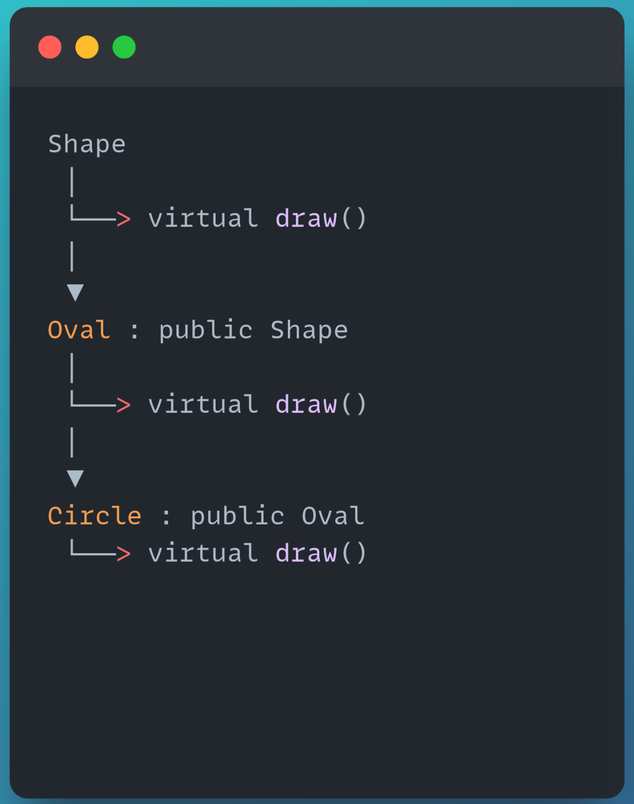
```cpp
/*

    Important facts:
      . We opt into polymorphism by setting up at least one virtual function
      . A virtual function is a function in the base class that can be overridden
            in derived classes.
      . Virtual functions enable runtime polymorphism by ensuring that the correct
        function (based on the actual object type) is called, WHEN accessed
        through a base class pointer or reference.
      . When a derived class provides its own implementation of a virtual function
        from the base class, this is called function overriding.
      . Overriding allows derived classes to provide specific behavior while keeping
        the same interface (method signature) as the base class.
      . The override keyword is used to indicate that a function is intended to
        override a virtual function in the base class. It's optional,
        but highly recommended.



*/
```

Polymorphism with virtual functions



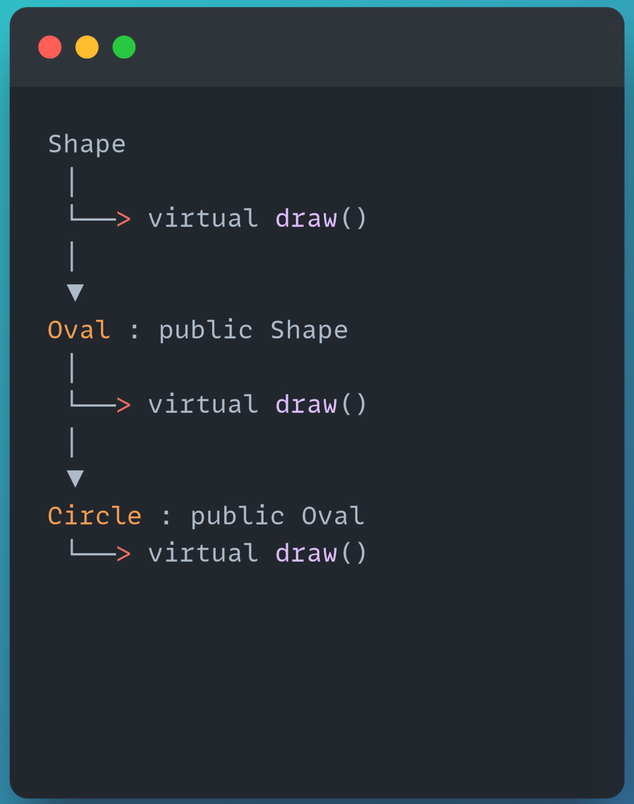

```cpp
// Shape class
export class Shape {
public:
    Shape(std::string_view description = "") : m_description(description) {}

    virtual void draw() const {
        fmt::println("Shape::draw() called. Drawing {}", m_description);
    }

    virtual ~Shape() = default;  // Virtual destructor for proper cleanup

protected:
    std::string m_description;
};
```

Polymorphism with virtual functions



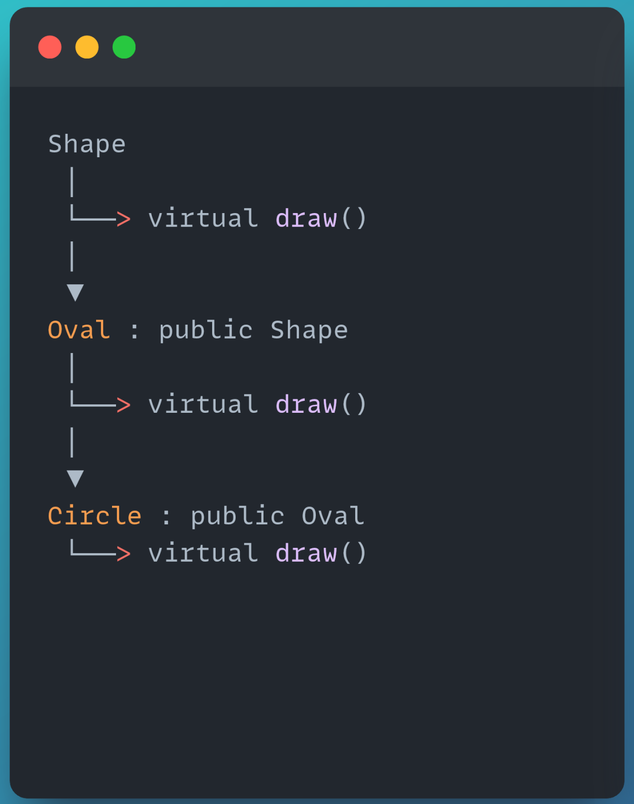

```cpp
// Oval class
export class Oval : public Shape {
public:
    Oval(double x_radius, double y_radius, std::string_view description)
        : Shape(description), m_x_radius(x_radius), m_y_radius(y_radius) {}

    virtual void draw() const override {
        fmt::println("Oval::draw() called. Drawing {} with m_x_radius: {} and m_y_radius: {}",
                     m_description, m_x_radius, m_y_radius);
    }

protected:
    double m_x_radius{0.0};
    double m_y_radius{0.0};
};
```

Polymorphism with virtual functions



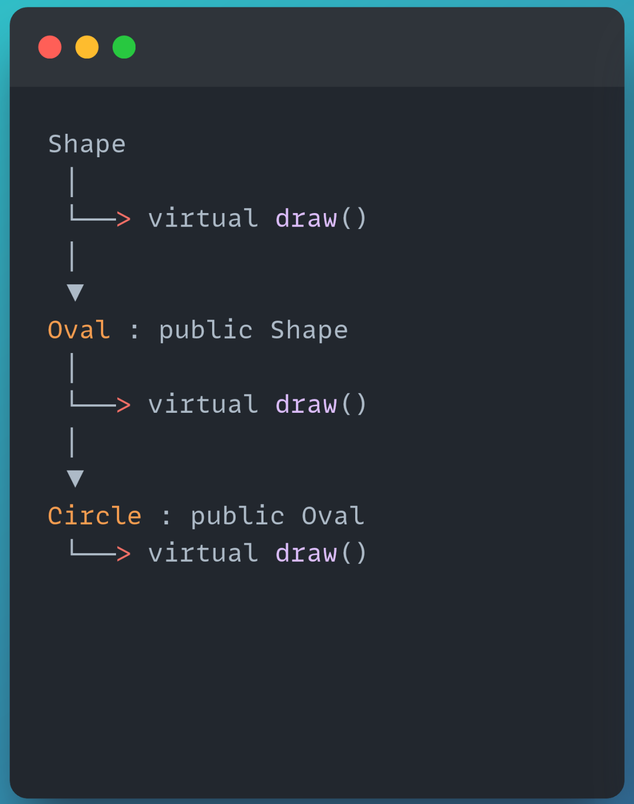

```cpp
// Circle class
export class Circle : public Oval {
public:
    Circle(double radius, std::string_view description)
        : Oval(radius, radius, description) {}

    virtual void draw() const override {
        fmt::println("Circle::draw() called. Drawing {} with radius: {}",
                                                    m_description, m_x_radius);
    }
};
```

Polymorphism with virtual functions



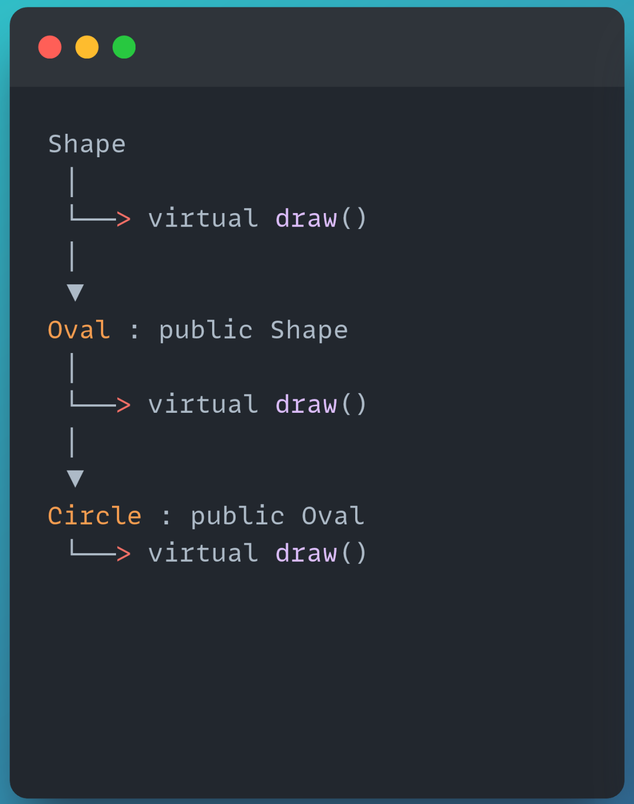

```
//Trying it out
// Creating instances of Shape, Oval, and Circle
poly_2::Shape shape("Generic Shape");
poly_2::Oval oval(5.0, 3.0, "Oval Shape");
poly_2::Circle circle(7.0, "Circle Shape");

// Managing objects via pointers and references
poly_2::Shape* shape_ptr = &shape;
poly_2::Shape* oval_ptr = &oval;    // Pointer to base class, but points to Oval object
poly_2::Shape* circle_ptr = &circle; // Pointer to base class, but points to Circle object

fmt::println("Calling draw() using Shape pointer (polymorphism):");
shape_ptr->draw();   // Calls Shape's draw() (as expected)
oval_ptr->draw();    // Calls Oval's draw(), polymorphism
circle_ptr->draw();  // Calls Circle's draw(), polymorphism
```

Polymorphism with virtual functions



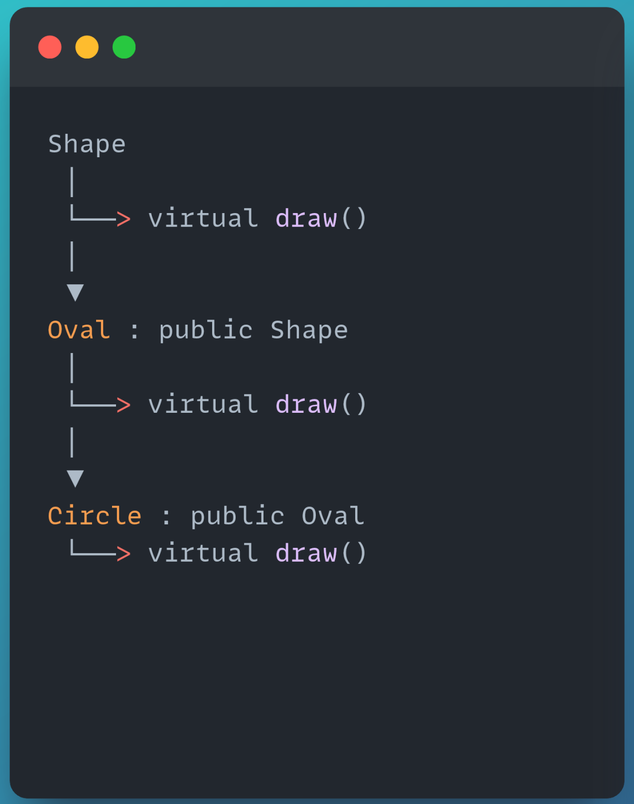

```
//Trying it out
// Creating instances of Shape, Oval, and Circle
poly_2::Shape shape("Generic Shape");
poly_2::Oval oval(5.0, 3.0, "Oval Shape");
poly_2::Circle circle(7.0, "Circle Shape");

// Managing objects via pointers and references
poly_2::Shape* shape_ptr = &shape;
poly_2::Shape* oval_ptr = &oval;    // Pointer to base class, but points to Oval object
poly_2::Shape* circle_ptr = &circle; // Pointer to base class, but points to Circle object

fmt::println("Calling draw() using Shape pointer (polymorphism):");
shape_ptr->draw();   // Calls Shape's draw() (as expected)
oval_ptr->draw();    // Calls Oval's draw(), polymorphism
circle_ptr->draw();  // Calls Circle's draw(), polymorphism
```

Polymorphism with virtual functions




```cpp
//Trying it out
// Managing objects via references
poly_2::Shape& shape_ref = shape;
poly_2::Shape& oval_ref = oval;    // Reference to base class, refers to Oval object
poly_2::Shape& circle_ref = circle; // Reference to base class, refers to Circle object

fmt::println("\nCalling draw() using Shape reference (polymorphism):");

shape_ref.draw();   // Calls Shape's draw() (as expected)
oval_ref.draw();    // Calls Oval's draw(), polymorphism
circle_ref.draw();  // Calls Circle's draw(), polymorphism
```

Polymorphism with virtual functions




```cpp
//If dynamic memory is involved, use smart pointers.
//Bring the classes into scope, to sharten the code.
using poly_2::Shape;
using poly_2::Oval;
using poly_2::Circle;

fmt::println("\nUsing smart pointers:");
{
    std::unique_ptr<Shape> shape = std::make_unique<Shape>("Basic Shape");
    shape->draw(); // Calls Shape's draw()
}  // shape goes out of scope and is destroyed here

{
    std::unique_ptr<Oval> oval = std::make_unique<Oval>(5.0, 10.0, "Oval Shape");
    oval->draw();     // Calls Oval's draw()
}  // oval goes out of scope and is destroyed here

{
    std::unique_ptr<Circle> circle = std::make_unique<Circle>(7.0, "Circle Shape");
    circle->draw();         // Calls Circle's draw()
}  // circle goes out of scope and is destroyed here
```

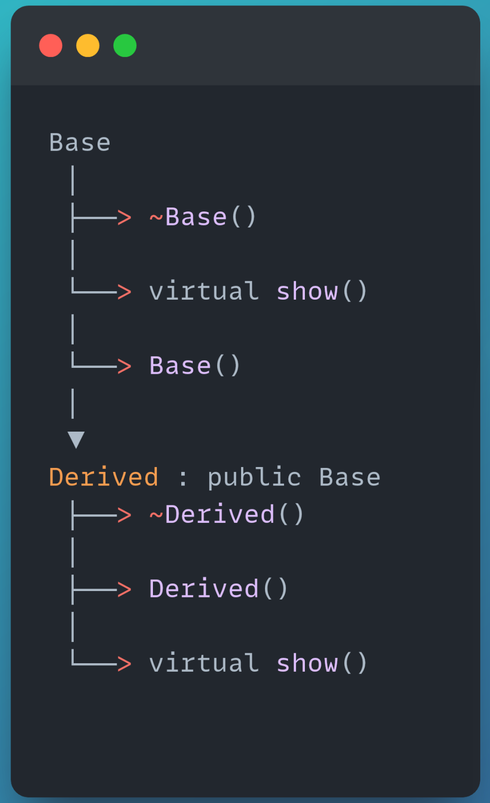
Demo time!

```
Shape
  |
  └──> virtual draw()
  |
  ▼
Oval : public Shape
  |
  └──> virtual draw()
  |
  ▼
Circle : public Oval
  └──> virtual draw()
```

Polymorphism: The need for virtual destructors

```
Base
  |
  ├──> ~Base()          // Non-virtual destructor, fmt::println("Base class destructor")
  |
  └──> virtual show()           // Virtual function, fmt::println("Base class show()")
  |
  └──> Base()           // Default constructor (implicitly called by Derived), no custom message
  |
  ▼
Derived : public Base
  ├──> ~Derived()       // Destructor, fmt::println("Derived class destructor")
  |                     // Deletes dynamically allocated memory (int* data)
  ├──> Derived()        // Constructor, allocates dynamic memory for `data`
  |                     // fmt::println("Derived class constructor")
  └──> virtual show()           // Overrides show(), fmt::println("Derived class show()")
```

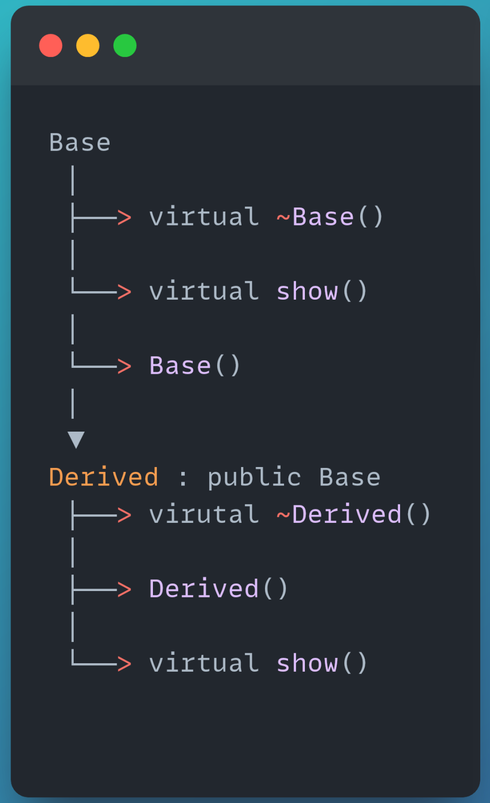
Polymorphism: The need for virtual destructors




```
        std::unique_ptr<Base> base_ptr = std::make_unique<Base>(param1, param2);
        //Goes out of scope: Only the base destructor is called.
```

```
/*

        . If the base class destructor is not virtual, when you delete an object
            through a pointer to the base class, the compiler will only call the
            destructor of the static type (the base class). This is because non-virtual
            functions are resolved at compile-time based on the static type of the pointer.



*/
```

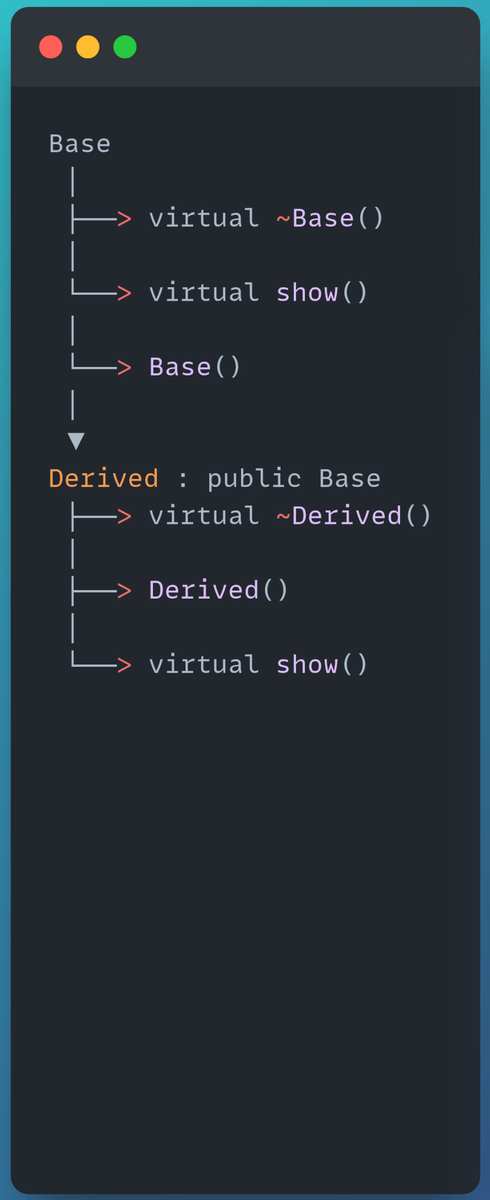
Polymorphism: The need for virtual destructors




```cpp
std::unique_ptr<Base> base_ptr = std::make_unique<Base>(param1, param2);
//Goes out of scope: Only the base destructor is called.
```

```cpp
//Solution: Mark the destructors virtual
export class Base {
public:
    // Virtual destructor
    virtual ~Base() {
        fmt::println("Base class destructor");
    }

    virtual void show() const {
        fmt::println("Base class show()");
    }
};
```

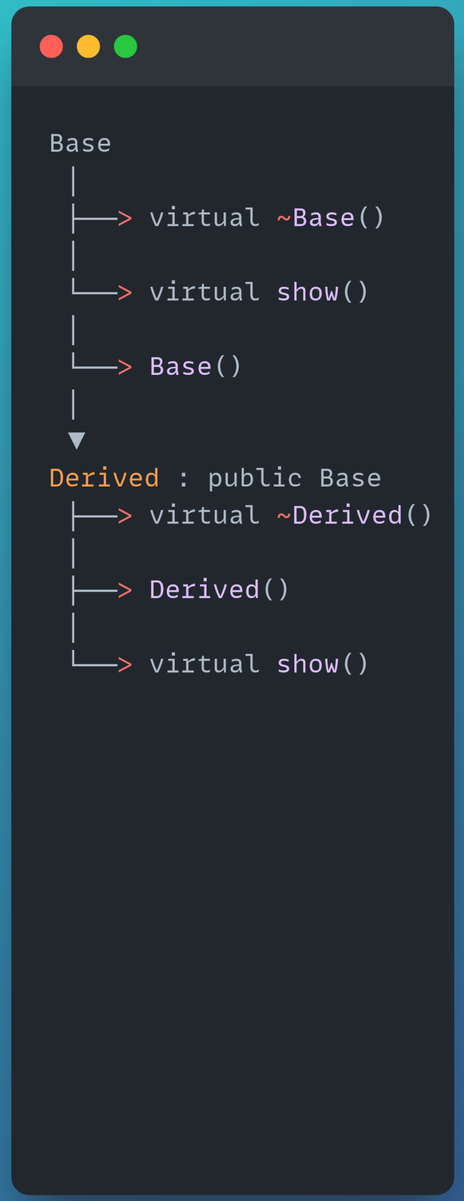
Polymorphism: The need for virtual destructors




```cpp
std::unique_ptr<Base> base_ptr = std::make_unique<Base>(param1, param2);
//Goes out of scope: Only the base destructor is called.
```

```cpp
//Solution: Mark the destructors virtual
export class Derived : public Base {
public:
    int* data;

    Derived() {
        data = new int[100];
        fmt::println("Derived class constructor");
    }

    ~Derived() override {
        delete[] data;  // Memory cleanup
        fmt::println("Derived class destructor");
    }

    void show() const override {
        fmt::println("Derived class show()");
    }
};
```

Demo time!




```cpp
std::unique_ptr<Base> base_ptr = std::make_unique<Base>(param1, param2);
//Goes out of scope: Only the base destructor is called.
```

Object sizes and slicing

```
early_binding::Shape
 |
 └──> draw()
 |
 └──> Shape()
 |
 ▼
early_binding::Oval : public Shape
 └──> draw()
 |
 └──> Oval()
 |
 ▼
early_binding::Circle : public Oval
 └──> draw()
 └──> Circle()
```

```
late_binding::Shape
 |
 ├──> virtual draw()
 |
 └──> Shape()
 |
 ▼
late_binding::Oval : public Shape
 └──> draw() override
 |
 └──> Oval()
 |
 ▼
late_binding::Circle : public Oval
 └──> draw() override
 └──> Circle()
```

```
// Polymorphism increases the sizes of your objects by a bit because of the vtable
```

Object sizes and slicing

```
 //early binding
fmt::println("\nObject sizes with static binding:");
fmt::println("sizeof(Shape) : {}", sizeof(poly_4::early_binding::Shape));
fmt::println("sizeof(Oval) : {}", sizeof(poly_4::early_binding::Oval));
fmt::println("sizeof(Circle) : {}", sizeof(poly_4::early_binding::Circle));

//late binding
fmt::println("\nObject sizes with dynamic binding:");
fmt::println("sizeof(Shape) : {}", sizeof(poly_4::late_binding::Shape));
fmt::println("sizeof(Oval) : {}", sizeof(poly_4::late_binding::Oval));
fmt::println("sizeof(Circle) : {}", sizeof(poly_4::late_binding::Circle));

//Slicing
fmt::println("\nSlicing:");
poly_4::late_binding::Circle circle1(3.3, "Circle1");
poly_4::late_binding::Shape shape = circle1;
shape.draw();// Shape::draw
```