final and access specifiers and virtual functions with default arguments

```
/*

    . Topics:
        .#1: Final
        .#2: Final and override are not keywords
        .#3: . Exploring the case where when the same function is public in the base class
                and private in the derived class
                . When polymorphism is used, the function in the derived class is called.
                . When static binding is used, the function in the base class is called.
        .#4: Virtual functions with default arguments
*/
```

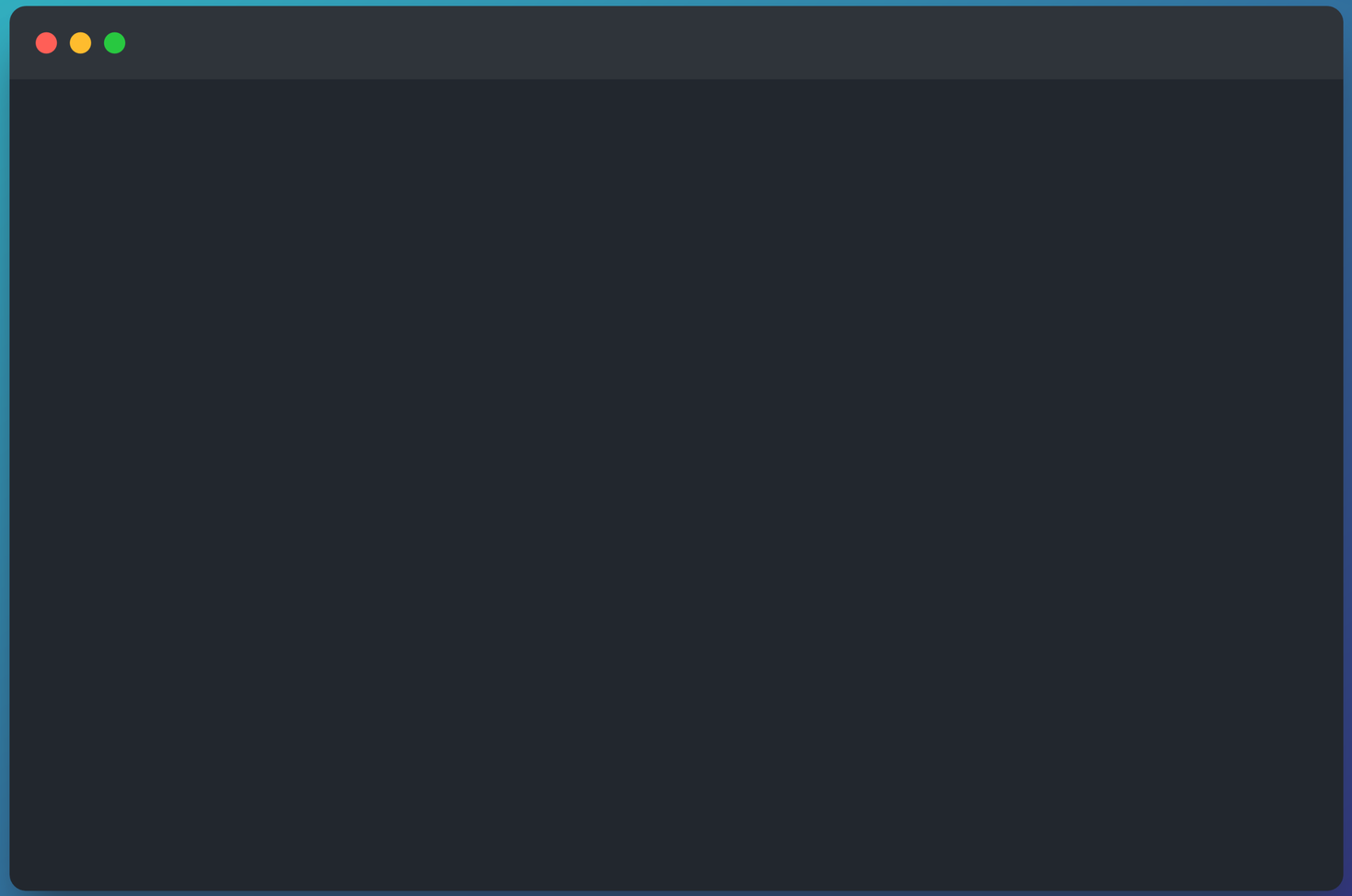
final

```
//This specifier is used to prevent further inheritance or overriding of a function.
Animal
 ├──> virtual void breathe() const                    // Base breathe()

Feline : public Animal
 ├──> virtual void run() const                         // Base run()

Dog : public Feline
 ├──> void run() const override final                  // Final run()

Cat final : public Feline
 ├──> void run() const override                        // Overridden run()
 ├──> virtual void miaw() const

Bird : public Animal
 ├──> virtual void fly() const final                   // Final fly()

Crow : public Bird
 ├──> // No final methods

Pigeon : public Bird
 ├──> // No final methods
```

final

```
//This specifier is used to prevent further inheritance or overriding of a function.
Animal
 ├──> virtual void breathe() const                    // Base breathe()

Feline : public Animal
 ├──> virtual void run() const                         // Base run()

Dog : public Feline
 ├──> void run() const override final                  // Final run()

Cat final : public Feline
 ├──> void run() const override                        // Overridden run()
 ├──> virtual void miaw() const

Bird : public Animal
 ├──> virtual void fly() const final                   // Final fly()

Crow : public Bird
 ├──> // No final methods

Pigeon : public Bird
 ├──> // No final methods
```

Demo time!



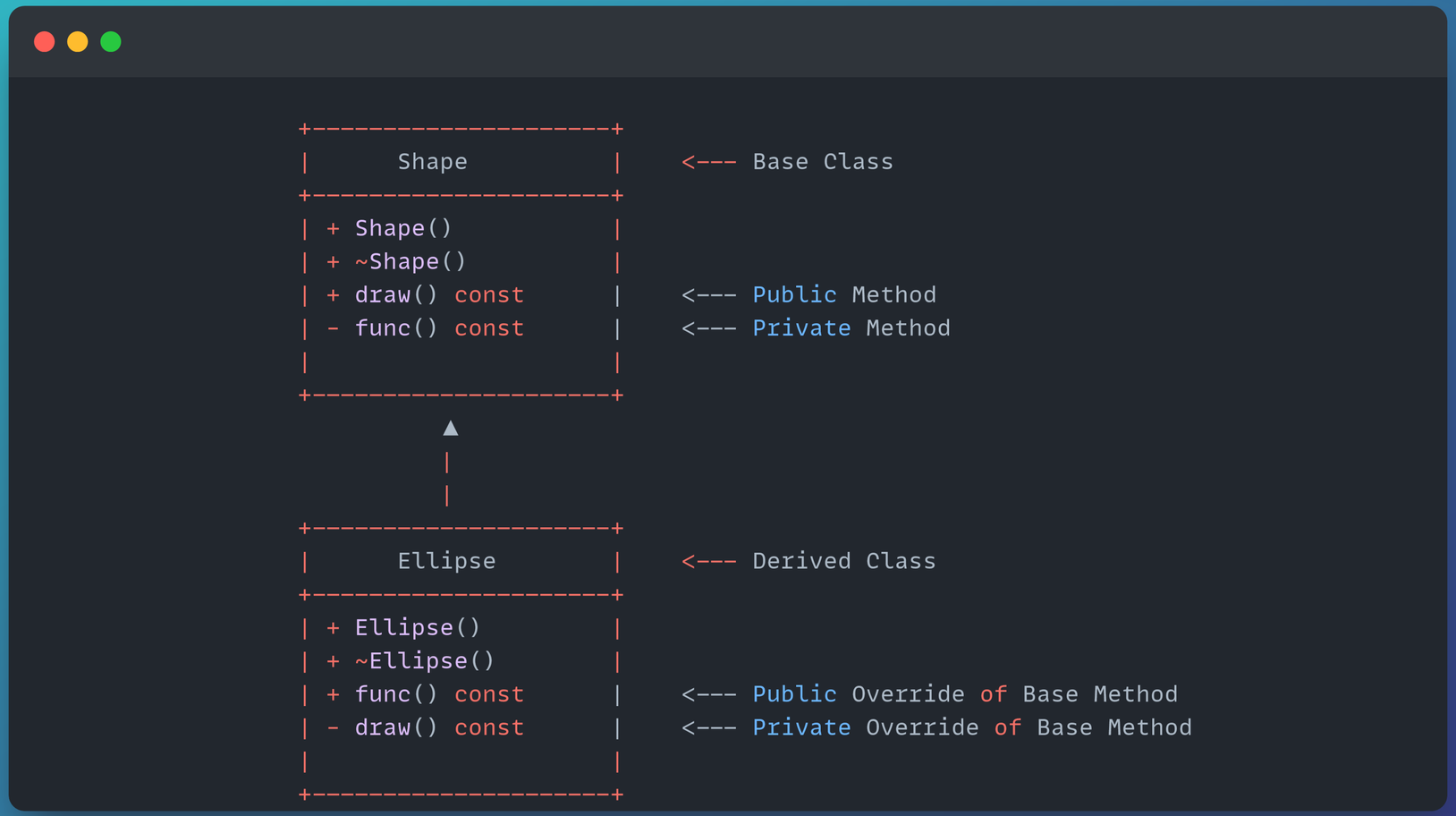
final and override are not reserved keywords

```
class final{
    //DONT' DO THIS
};

export void override(){ // DON'T DO THIS EITHER.
    fmt::println("The override function being called...");
}
```

Demo time!

```
class final{
    //DONT' DO THIS
};

export void override(){ // DON'T DO THIS EITHER.
    fmt::println("The override function being called...");
}
```

Access specifiers: static binding vs dynamic binding




```
/*
        . The question: if we go through a base class pointer, which classes's function will be called
          both for draw and func?
          . When polymorphism is used, the function in the derived class is called.
          . When static binding is used, the function in the base class is called.
*/
```

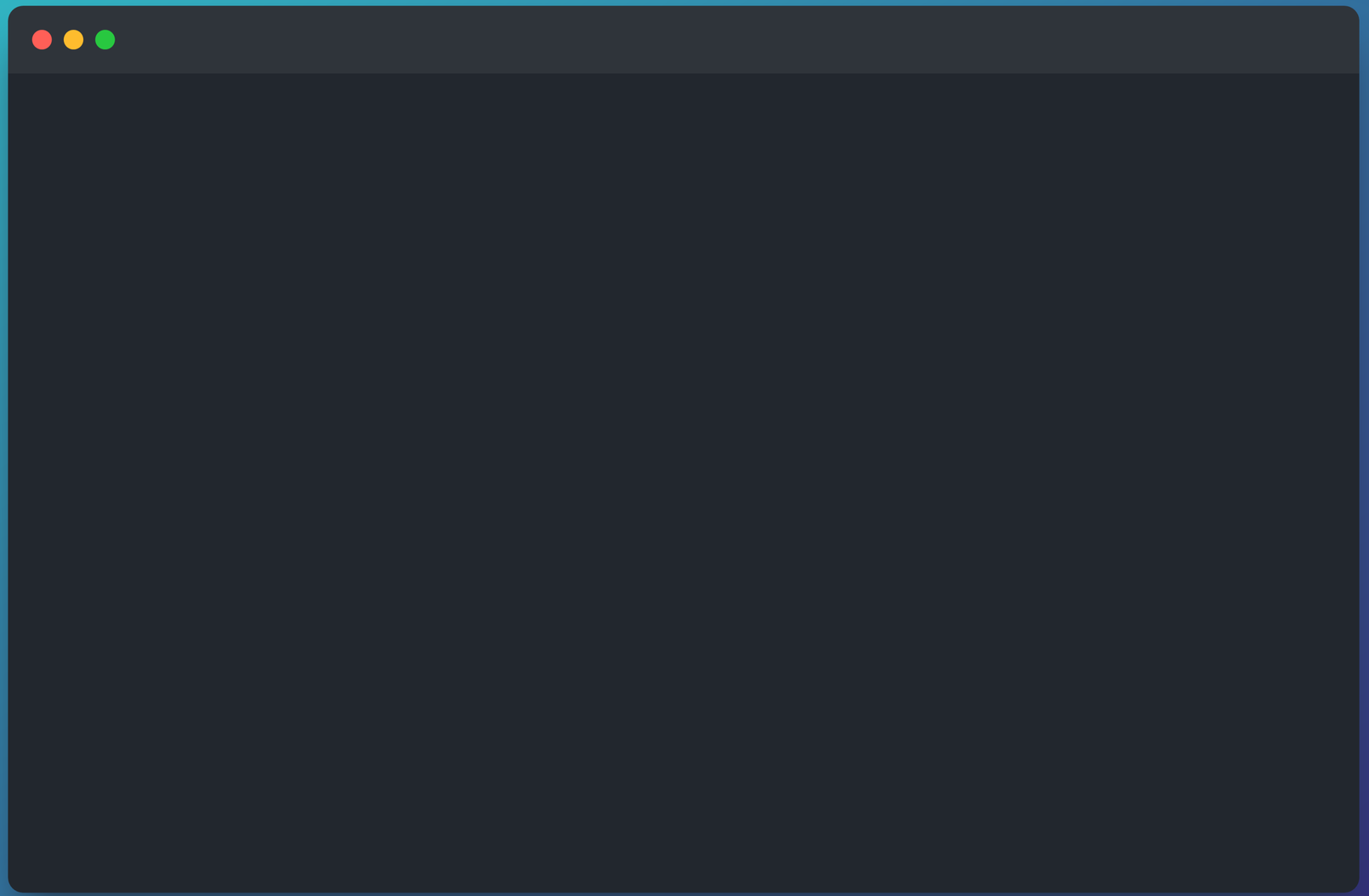
Access specifiers: static binding vs dynamic binding

```
//Dynamic binding
fmt::println("\nDynamic binding");
using inh_poly_3::dyn_bind::Shape;
using inh_poly_3::dyn_bind::Ellipse;

// Accessing stuff through the base class pointer
std::shared_ptr<Shape> shape0 =
                        std::make_shared<Ellipse>(1, 5, "ellipse0");
shape0->draw();// Polymorphism
// shape0->func(); // Error :  func is private in Shape

// Direct objects : static binding
Ellipse ellipse1(1, 6, "ellipse1");
ellipse1.func();// Works
//ellipse1.draw(); //Error : draw() is private in Ellipse•– Static binding

// Raw derived object assigned to raw base object
// Slicing will occur, Shape::draw will be called
Shape shape3 = Ellipse(2, 3, "ellipse3");
shape3.draw();// Shape::draw() called
// shape3.func(); // Error : func is private in shape
```

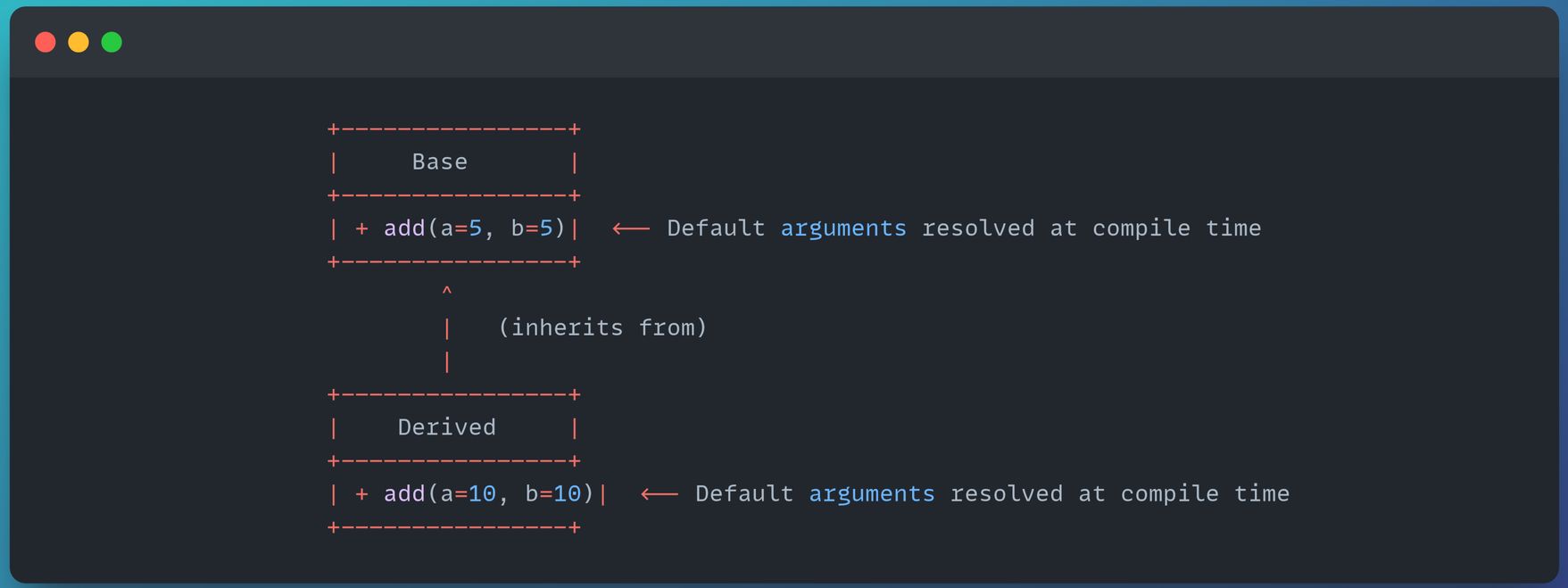
Access specifiers: static binding vs dynamic binding

```
//Static binding
fmt::println("\nStatic binding");
using inh_poly_3::stat_bind::Shape;
using inh_poly_3::stat_bind::Ellipse;

// Accessing stuff through the base class pointer
std::shared_ptr<Shape> shape0 =
        std::make_shared<Ellipse>(1, 5, "ellipse0");
shape0→draw();// Polymorphism
// shape0→func(); // Error :  func is private in Shape

// Direct objects : static binding
Ellipse ellipse1(1, 6, "ellipse1");
ellipse1.func();// Works
//ellipse1.draw(); //Error : draw() is private in Ellipse•– Static binding

// Raw derived object assigned to raw base object
// Slicing will occur, Shape::draw will be called
Shape shape3 = Ellipse(2, 3, "ellipse3");
shape3.draw();// Shape::draw() called
// shape3.func(); // Error : func is private in shape
```

Demo time!

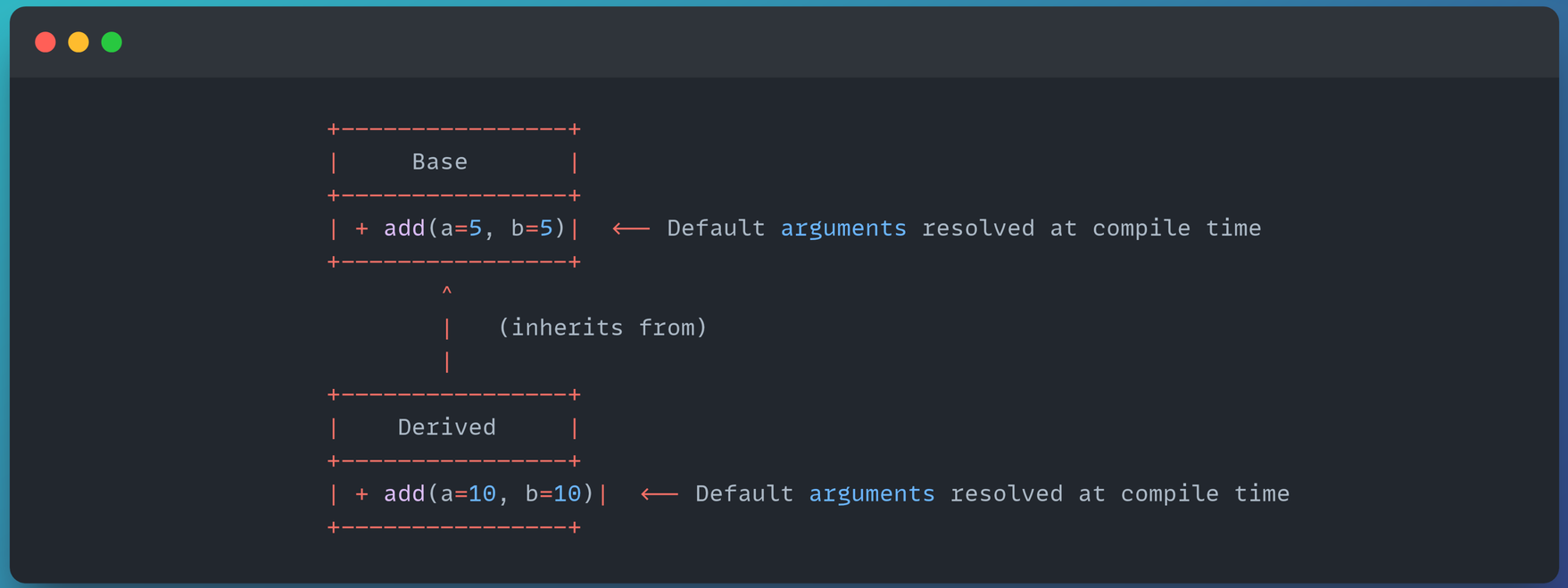


Virtual functions and default arguments




```
/*
    . If we use a base pointer to manage a derived object and call the add function, the default
      arguments of the base class will be used. This is probably what you don't want.

    . How I handle this:
      . Use the same default arguments in the derived class as in the base class.
        . probably even store them in a static constexpr variable in the base class,
          so that I manage it from one place.
      . Not using default arguments in virtual functions at all.
*/
```

Virtual functions and default arguments




```
fmt::println("\nBase ptr managing derived object: ");
inh_poly_4::Base *base_ptr1 = new inh_poly_4::Derived;
double result = base_ptr1->add();
fmt::println("Result (base ptr) : {}", result);// 12
```

Demo time!

```
+-----------------+
|      Base       |
+-----------------+
| + add(a=5, b=5)|   ⟵ Default arguments resolved at compile time
+-----------------+
         ^
         |   (inherits from)
         |
+-----------------+
|    Derived      |
+-----------------+
| + add(a=10, b=10)|   ⟵ Default arguments resolved at compile time
+-----------------+
```