Modern Programming Principles & Practice

Feb-May 2025
sruthi.padathara@dorset.ie

Contents

- **Using super() to Call Parent Class Constructor**
- Static Binding vs Dynamic Binding in C++
- Real-World Example of Static vs Dynamic Binding in C++
- Virtual Functions & Abstract Classes
- Requirement of Virtual Functions
- How to Implement Virtual Functions
- Pure Virtual Functions
- Abstract Classes
- Why Use Abstract Classes?
- When to Use Abstract Classes?
- Difference Between Virtual Functions & Abstract Classes

Using super() to Call Parent Class Constructor

Create a class Person with a parameterized constructor that takes name and age. Create a class Student that inherits from Person and uses super() to call the Person constructor. The Student class should also have a method display() to print the name and age.

- Implement a main() function that creates a Student object and calls display().

static Binding (Early Binding)

Definition:

- Static Binding (also known as Early Binding) refers to the process where the function call is resolved at compile time, rather than at runtime.
- In other words, the decision about which function to call is made while the program is being compiled. The compiler uses information about the object types to resolve which function to call.

How It Works:

- **Function Overloading:** Static binding occurs when you have multiple functions with the same name but different parameter types. The compiler will decide which function to call based on the parameters passed during the function call.
- **Function Overriding (Without Virtual Functions):** Static binding also happens in function overriding if the function is not marked as virtual. In this case, the compiler determines which function to call based on the object type at compile time.

Process:

- The compiler looks at the type of the object or the arguments passed to the function at compile time.
- Based on this information, it binds the function call to the appropriate function.

Advantages of Static Binding:

1. **Faster Execution:**
    - Since function calls are resolved at compile time, the program runs faster. The overhead of runtime function lookup is eliminated.
2. **Less Overhead:**
    - The compiler does all the work of determining which function to call at compile time. This avoids the need for dynamic lookups at runtime.

Disadvantages of Static Binding:

1. **No Runtime Polymorphism:**
    - With static binding, polymorphism (the ability for a base class reference to call a derived class method) does not work. You cannot override functions at runtime, which limits the flexibility of object-oriented programming.
2. **Limited Flexibility:**
    - Static binding does not allow for flexibility in handling dynamic changes in behavior. For example, if you want to call the Derived class function through a base class pointer, static binding would always call the base class function instead of the overridden derived function.

Write a **C++ program** that demonstrates **function overloading** by implementing a Payment class. The class should have multiple pay() functions to handle different types of payments.

Requirements:

1. **Overload the pay() function**:
   - One function should accept only an int amount (for cash payments).
   - Another function should accept an int amount and a string cardNumber (for credit card payments).
2. In the main() function:
   - Create an object of the Payment class.
   - Call the pay() function for both cash and credit card payments.
3. Display appropriate messages when a payment is made.

Dynamic Binding (Late Binding)

- Function call is **resolved at runtime**.
- Achieved using **virtual functions** and **pointers/references to base class**.
- Function execution depends on the **actual object type at runtime**.
- Enables **polymorphism**, where a base class pointer can call a derived class function.
- Slower than static binding due to runtime decision-making.

Advantages:

- Supports **polymorphism**
- Increases **flexibility** in OOP

Disadvantages:

- Slight performance overhead due to runtime resolution

Write a **C++ program** that demonstrates **runtime polymorphism** using **virtual functions**. Implement a base class Payment and derived classes CreditCard and PayPal to process different types of payments.

Requirements:

1. Create a **base class** Payment with a **virtual function** pay(int amount), which prints a generic payment processing message.
2. Derive two classes from Payment:
   - **CreditCard**: Overrides pay(int amount) to display a credit card payment message.
   - **PayPal**: Overrides pay(int amount) to display a PayPal payment message.
3. In the main() function:
   - Declare a **pointer to Payment**.
   - Assign it to objects of CreditCard and PayPal classes.
   - Call the pay() function and observe **dynamic method dispatch**.

Virtual Functions

- A **member function** in a base class that can be **overridden** in derived classes.
- Declared using the virtual keyword.
- Enables **runtime polymorphism** (method calls are resolved dynamically).
- Helps in **achieving dynamic binding** (late binding).
- When overridden, the derived class version is invoked even when using a base class pointer/reference.

Requirement of Virtual Functions

- Needed for **polymorphism** (same function behaves differently in derived classes).
- Used when **base class provides a common interface**, but derived classes need their own implementation.
- Prevents **early binding** (compile-time resolution).
- Makes **function overriding meaningful** in inheritance.

How to Implement Virtual Functions

- Declare function as virtual in the base class.
- Override it in the derived class.
- Use a **base class pointer/reference** to call derived class functions.
- Allows for **dynamic dispatch** (deciding function execution at runtime).

```
class Base {
public:
    virtual void functionName() {
        // Base class implementation
    }
};

class Derived : public Base {
public:
    void functionName() override {
        // Overridden function in derived class
    }
};

int main() {
    Base* obj = new Derived();
    obj->functionName();  // Calls Derived class function (dynamic binding)
    delete obj;}
```

Why Use Virtual Functions?

You are building an **audio processing system** where different **audio formats (MP3, WAV, FLAC)** have a `playSound()`function.

- If the `playSound()` function is **not virtual**, which function gets executed when calling it through a base class pointer?
- Explain how using virtual functions **resolves this issue**.

If `playSound()` is Not Virtual:

If `playSound()` is not declared as a **virtual function** in the base class, **compile-time (static) binding** occurs. This means that when calling `playSound()` through a base class pointer, the **base class version** of the function will always be executed, even if the actual object is of a derived class.

How Virtual Functions Solve This Issue:

Declaring `playSound()` as **virtual** in the base class enables **runtime (dynamic) binding**. This ensures that the correct derived class function is executed based on the actual object type.

Conclusion:

- **Without virtual**, the base class function is always executed due to **static binding**.
- **With virtual**, the correct derived class function is executed based on **dynamic binding (polymorphism)**.
- **Virtual functions are essential** for ensuring the correct function execution when working with **inheritance and base class pointers/references**.

Abstract Classes

An **abstract class** in C++ is a class that **cannot be instantiated** and is designed to be a base class for other derived classes. It **contains at least one pure virtual function**, which must be implemented by any subclass.

```
class Shape {
public:
    virtual void draw() = 0; // Pure virtual function
};
```

Feature	Normal Class	Abstract Class
Instantiation	Can create objects	Cannot create objects
Virtual Functions	Optional	Must have at least one pure virtual function
Purpose	Can be standalone or base class	Only meant to be a base class

Why do we need abstract classes when we have normal base classes?

- **Ensures Derived Classes Implement Required Methods** – If a base class has common behavior but should not be instantiated, an **abstract class forces derived classes to define specific behavior**.
- **Prevents Direct Instantiation** – Abstract classes **prevent incomplete objects from being created**.
- **Improves Code Maintainability** – Clearly defines a common interface for multiple derived classes.

Can we create an object of an abstract class? Why or why not?

No, we cannot create an object of an abstract class.
Since an **abstract class contains at least one pure virtual function**, it has **incomplete functionality**, making it impossible to instantiate.

```
Shape s;  // Error: Cannot instantiate abstract class
```

When should you use an abstract class instead of a normal base class?

Use an **abstract class** when:
You want to **define a common interface** for multiple derived classes.
You need to **force derived classes to implement specific functions**.
You want to **prevent creating objects of the base class**.

Use a **normal base class** when:
You **don't need to enforce** function implementation in derived classes.
The base class **can have independent instances**.

How does an abstract class enforce code consistency?

- **Defines a Standard Interface** – Every subclass must implement the same function signatures.
- **Prevents Partial Implementations** – A derived class **must override** all pure virtual functions.
- **Encourages Code Reusability** – Abstract classes **store common functionality** to reduce redundant code in derived classes.

```
class Animal {
public:
    virtual void makeSound() = 0; // Every derived class must implement this
};
```

Now, every class that inherits from Animal **must** implement makeSound(), ensuring consistency across different animal types.