Object Oriented Programming (Spring2024)

Week05 (12/12/13-March-2024)

M Ateeq,
Department of Data Science, The Islamia University of Bahawalpur.

Polymorphism: A Core Principle of Object-Oriented Design

In object-oriented programming (OOP), polymorphism (derived from the Greek words "poly" meaning many and "morphe" meaning form) is a fundamental concept that empowers code to exhibit flexibility and adaptability. It essentially allows a single message or function call to have different behaviors depending on the object it's being applied to. This dynamic behavior offers significant advantages in terms of code reusability, maintainability, and the ability to create well-structured, extensible programs.

Understanding the Core Concept

Imagine a scenario where you're working with various geometric shapes in your C++ program. You might have classes representing squares, circles, triangles, and so on. Each shape has a distinct way of calculating its area. Polymorphism allows you to define a general function, say `calculateArea()` , in a base class named `Shape` . This function would be declared as a pure virtual function (using the `virtual` keyword), indicating that it must be implemented (overridden) in derived classes like `Square` , `Circle` , and `Triangle` to provide their specific area calculation formulas.

Here's a breakdown of the key aspects involved:

- **Single Message/Function Call:** You would call the `calculateArea()` function on objects of different derived shapes (e.g., `square.calculateArea()` , `circle.calculateArea()` , `triangle.calculateArea()`).
- **Different Behaviors:** Within each derived class, the `calculateArea()` function would be overridden to implement the appropriate area calculation for that particular shape.
- **Dynamic Binding (Runtime Dispatch):** When you call `calculateArea()` on a derived shape object, the C++ runtime system dynamically determines the correct implementation of the function to execute based on the object's actual type at runtime. This ensures that the most suitable area calculation is performed for each shape.

Benefits of Polymorphism

- **Code Reusability:** By defining a generic function in the base class, you avoid code duplication, as derived classes can provide their specialized implementations without rewriting the entire function.
- **Maintainability:** Modifications or enhancements to the area calculation logic can be made within the overridden functions of derived classes, keeping the base class code clean and

focused on the general concept of area calculation.

- **Extensibility:** New shapes can be easily added to your program by creating additional derived classes and overriding the `calculateArea()` function with their specific formulas. The existing code that interacts with shapes through the base class reference will automatically work with the new shapes without any modifications.
- **Flexibility:** Polymorphism enables you to write more generic code that can operate on a collection of objects of different but related types, enhancing the adaptability and flexibility of your program.

Real-World Analogy

Consider a zoo with various animals. You might have a base class `Animal` with a virtual function `makeSound()`. Derived classes like `Dog`, `Cat`, and `Lion` would override `makeSound()` to produce their characteristic sounds (bark, meow, roar). When you call `makeSound()` on an animal object (whose exact type might not be known beforehand), the correct sound is played based on the animal's specific type at runtime.

Illustrative Code Example

```
#include <iostream>

class Shape {
public:
    virtual double calculateArea() const = 0; // Pure virtual function
(must be overridden)
};

class Square : public Shape {
private:
    double side;
public:
    Square(double side) : side(side) {}
    double calculateArea() const override { return side * side; }
};



class Circle : public Shape {
private:
    double radius;
public:
    Circle(double radius) : radius(radius) {}
    double calculateArea() const override { return 3.14159 * radius *
radius; }
};
```

```
int main() {
    Shape* shape1 = new Square(5);
    Shape* shape2 = new Circle(3);

    // Polymorphic behavior: Calls the appropriate calculateArea() bas
ed on object type
    std::cout << "Area of Square: " << shape1->calculateArea() << st
d::endl;
    std::cout << "Area of Circle: " << shape2->calculateArea() << st
d::endl;

    delete shape1;
    delete shape2;

    return 0;
}
```

In this example, the `calculateArea()` function exhibits polymorphic behavior. When called through the base class pointers `shape1` and `shape2` , the correct area calculation for the specific square or circle object is executed at runtime due to dynamic binding.

Virtual Functions Explained

Imagine a Zoo with Different Animals:

- You have a base class called `Animal` that represents the general concept of an animal.
- Derived classes like `Dog` , `Cat` , and `Lion` inherit from `Animal` to represent specific types of animals.

Making Sounds:

- All animals make sounds, but each has a unique vocalization.
- You want a function named `makeSound()` to be part of every animal. However, the specific sound each animal makes should be different.

Virtual Functions to the Rescue:

1. **Declare `makeSound()` as virtual in the `Animal` class:** This tells the compiler that derived classes can provide their own versions of this function.
2. **Override `makeSound()` in derived classes:** Each derived class (e.g., `Dog` , `Cat` , `Lion`) creates its own `makeSound()` function that defines its specific sound (e.g., "Woof!", "Meow!", "Roar!").

The Magic of Polymorphism:

- When you call `makeSound()` on an animal, something special happens:

    - The compiler doesn't know the exact animal type beforehand (it might be a dog, cat, or lion).
    - At runtime, based on the actual animal object being used (e.g., a `Dog` object), the correct version of `makeSound()` (in this case, `Dog::makeSound()`) is executed. This is called **dynamic binding**.

Think of it like this:

- You have a single remote control (the base class reference) that can interact with different TVs (derived class objects).
- Pressing the "volume up" button on the remote (calling `makeSound()`) does the appropriate action based on the specific TV it's pointed at (the actual animal object).

Benefits:

- **Flexible code:** You can write generic code (the `makeAnimalsSpeak()` function) that works with various animals without needing separate functions for each type.
- **Extensible:** Adding new animals (derived classes) only requires overriding `makeSound()` in the new class, keeping the existing code maintainable.

Key Points:

- Virtual functions are declared with the `virtual` keyword in the base class.
- Derived classes override virtual functions to provide their specific implementations.
- Dynamic binding ensures the correct function version is called at runtime based on the object's actual type.

Simple Analogy:

Think of shapes like squares, circles, and triangles. A base class `Shape` could have a virtual function `calculateArea()` . Derived classes would override this function with their specific area calculation formulas. Calling `calculateArea()` on a shape object (without knowing its exact type beforehand) would result in the correct area calculation due to dynamic binding.

Formalizing the Concept: Overloading vs Overriding

Static vs Dynamic Polymorphism

Function Overloading: Compile Time Polymorphism

Function overloading is a powerful concept in C++ that allows you to define multiple functions with the same name, but differentiated by their parameter lists. This enables the compiler to select the most appropriate function to call based on the arguments you provide during function invocation. It's considered a form of compile-time polymorphism, as the decision is made during compilation rather than runtime.

Understanding Function Overloading

- **Multiple Functions, Same Name:** You can create several functions with the same name within the same scope (typically a class or a translation unit).
- **Differing Parameter Lists:** The key distinction lies in the number, types, and order of the parameters each function accepts. This variation in parameter lists allows the compiler to

Compile-Time Resolution

- **Argument Matching:** When you call an overloaded function, the compiler examines the arguments you've supplied.
- **Function Selection:** It then compares the argument types and number with the parameter lists of all the overloaded functions with that name.
- **Best Match:** The function whose parameter list exactly matches the arguments in terms of type and order is chosen for execution.

Benefits of Function Overloading

- **Improved Code Readability:** Function names often convey the intended operation. Overloading allows you to maintain that name for functions that perform similar operations on different data types, enhancing code clarity.
- **Maintainability:** By grouping related operations under the same function name, you make code easier to understand and modify.
- **Convenient Data Type Handling:** You can handle various data types within a single function name, promoting code conciseness and reducing the need for multiple functions with slightly different purposes.

Example: The Versatile `sum()` Function

Here's a demonstration of function overloading with a `sum()` function that can handle integers, floats, and even strings:

```cpp
#include <iostream>
#include <string>

using namespace std;

// Overloaded sum() function for integers
int sum(int a, int b) {
    return a + b;
}

// Overloaded sum() function for floats
double sum(double a, double b) {
    return a + b;
}
```

```
// Overloaded sum() function for string concatenation
string sum(string a, string b) {
    return a + b; // Concatenates the strings
}

int main() {
    int x = 5, y = 3;
    double d1 = 2.5, d2 = 1.7;
    string s1 = "Hello ", s2 = "World!";

    cout << "Sum of integers: " << sum(x, y) << endl;
    cout << "Sum of floats: " << sum(d1, d2) << endl;
    cout << "Concatenation of strings: " << sum(s1, s2) << endl;

    return 0;
}
```

In this example, the `sum()` function name is overloaded for three different scenarios:

- Adding integers
- Adding floats
- Concatenating strings

When you call `sum(x, y)`, the compiler identifies the `sum(int, int)` version as the best match due to the integer arguments. Similarly, for `sum(d1, d2)`, the `sum(double, double)` version is chosen, and for `sum(s1, s2)`, the `sum(string, string)` version is selected.

Key Points to Remember

- Function overloading is based on parameter lists, not return types. You cannot overload functions solely based on return types.
- The order of parameter lists matters. Functions with the same parameter types but in a different order are considered distinct.
- Function overloading cannot be used with member functions (methods) within the same class that have the same name. You'll need to use unique names for member functions.

By effectively using function overloading, you can create more readable, maintainable, and flexible C++ code that caters to various data types and operations under intuitive function names.

Function Overriding: Run Time Polymorphism

Specialization Through Inheritance in C++

Function overriding is a cornerstone of object-oriented programming in C++. It empowers you to redefine a virtual function inherited from a base class in a derived class to provide a more specific implementation tailored to the derived class's needs. This mechanism fosters code flexibility and polymorphism, allowing a single function call to exhibit different behaviors depending on the object's type at runtime.

Understanding Function Overriding

- **Redefining Virtual Functions:** In a derived class, you can override a virtual function declared in the base class. This means you provide a new implementation for that function that's specific to the derived class.
- **Inheritance Prerequisite:** Function overriding necessitates inheritance. The derived class must inherit the virtual function from the base class in order to redefine it.
- **Virtual Keyword:** The virtual keyword in the base class declaration signals to the compiler that this function is intended for potential overriding in derived classes.

Dynamic Binding: The Powerhouse

- **Runtime Determination:** When you call a virtual function through a base class reference or pointer that points to a derived class object, the decision of which function implementation to execute is made at runtime, not compile time. This is known as dynamic binding.
- **Object Type, Not Reference/Pointer Type:** The crucial factor determining which function to call is the actual type of the object being referenced, not the type of the reference or pointer variable itself.

Example: The Vocal Animal Kingdom

Consider an animal hierarchy modeled with inheritance:

- Base class: `Animal`
- Derived classes: `Dog` , `Cat`

The `Animal` class might have a virtual function named `makeSound()` declared with the `virtual` keyword. This function represents the general concept of an animal making a sound, but the specific sound varies for different animals.

```
class Animal {
public:
    virtual void makeSound() const {
        std::cout << "Generic animal sound" << std::endl;
    }
};

class Dog : public Animal {
public:
    void makeSound() const override { // Override with specific sound
        std::cout << "Woof!" << std::endl;
    }
};
```

```cpp
class Cat : public Animal {
public:
    void makeSound() const override { // Override with specific sound
        std::cout << "Meow!" << std::endl;
    }
};
```

In this example, the `makeSound()` function is virtual in the `Animal` class. The derived classes `Dog` and `Cat` override this function to provide their specific sounds ("Woof!" and "Meow!"). Now, when you call `makeSound()` on an `Animal` reference or pointer that points to either a `Dog` or `Cat` object, dynamic binding ensures that the appropriate overridden function (`Dog::makeSound()` or `Cat::makeSound()`) is executed at runtime based on the object's actual type.

Benefits of Function Overriding

- **Polymorphism in Action:** It enables polymorphic behavior, where a single function call can manifest differently based on the object's type, promoting code flexibility.
- **Code Reusability:** You define the general behavior in the base class and specialize it in derived classes, fostering code reuse and maintainability.
- **Extensibility:** The ability to add new derived classes with their own overridden implementations of virtual functions makes the code extensible to accommodate new types of objects.

Key Points to Remember

- Function overriding can only occur with virtual functions.
- The base class virtual function declaration serves as a template for overriding in derived classes. The parameter list and return type must be identical in the overridden function.
- Dynamic binding ensures that the most suitable implementation is executed at runtime based on the object's actual type.

By effectively leveraging function overriding, you can create well-structured, extensible object-oriented programs that exhibit flexibility and code reusability.

Polymorphic References and Pointers: Unleashing Dynamic Behavior

Polymorphic references and pointers are powerful tools in C++ object-oriented programming that enable you to achieve dynamic binding, a core concept behind polymorphism. This allows a single function call through a base class reference or pointer to execute the most suitable implementation based on the object's actual type at runtime, not the reference/pointer type itself.

Declaring References and Pointers to Base Classes

- **Base Class References:** You can declare references that refer to objects of the base class or any of its derived classes.

- **Base Class Pointers:** Similarly, you can declare pointers that can point to objects of the base class or any of its derived classes.

```
class Animal {
public:
    virtual void makeSound() const = 0; // Pure virtual function
};

class Dog : public Animal {
public:
    void makeSound() const override {
        std::cout << "Woof!" << std::endl;
    }
};



class Cat : public Animal {
public:
    void makeSound() const override {
        std::cout << "Meow!" << std::endl;
    }
};
```

In this example, `Animal` is a base class, and `Dog` and `Cat` are derived classes. Here's how to declare references and pointers to `Animal` :

```
Animal& animalRef;  // Reference to Animal (can refer to Animal or derived class objects)
Animal* animalPtr;  // Pointer to Animal (can point to Animal or derived class objects)
```

Assigning Derived Class Objects

- **Upcasting:** You can assign objects of derived classes to base class references or pointers. This is known as upcasting.

```
Dog dog;
Cat cat;

animalRef = dog;  // Assigning a Dog object to Animal reference (upcasting)
animalPtr = &cat; // Assigning address of a Cat object to Animal pointer (upcasting)
```

Function Calls with Dynamic Binding

- **Focus on Object Type:** When you call a virtual function through a base class reference or pointer, the **object's type** at runtime determines which function implementation is executed, not the reference/pointer type itself. This is the essence of dynamic binding.

```
void makeAnimalsSpeak(Animal& animal) {
    animal.makeSound(); // Dynamic binding: Calls Dog::makeSound() or Cat::makeSound()
}

int main() {
    Dog dog;
    Cat cat;

    makeAnimalsSpeak(dog);  // Outputs "Woof!"
    makeAnimalsSpeak(cat);  // Outputs "Meow!"
}
```

In the `makeAnimalsSpeak()` function, although the parameter is an `Animal` reference, dynamic binding ensures that the appropriate overridden function (`Dog::makeSound()` or `Cat::makeSound()`) is called based on the actual type of the object being referenced (`dog` or `cat`).

Benefits of Polymorphic References and Pointers

- **Generic Code:** You can write generic code that operates on base class references or pointers, allowing it to work with various derived class objects without modifying the code for each specific type.
- **Flexibility:** This approach promotes a flexible and extensible design, as you can add new derived classes without needing to modify existing functions that use base class references or pointers.
- **Code Reusability:** By focusing on the base class interface, you can write reusable code that can interact with a variety of derived class objects.

Key Points to Remember

- While you can assign derived class objects to base class references/pointers (upcasting), be cautious of downcasting (assigning a base class reference/pointer to a derived class variable) to avoid potential type mismatches.
- Polymorphic references and pointers work effectively when combined with virtual functions and function overriding to enable dynamic behavior.

Reflection MCQs:

Q1. What is polymorphism in C++?

A) A way to call functions based on their return type
B) The ability of a message to be displayed in more than one form
C) The ability of different classes to have methods with the same name and functionality

D) The concept of allowing objects of different classes to be treated as objects of a common superclass
Correct Answer: D

Q2. Which of the following is an example of static polymorphism?

A) Function overloading
B) Virtual functions
C) Dynamic binding
D) Pure virtual functions
Correct Answer: A

Q3. What is a virtual function in C++?

A) A function that can be overridden in any derived class
B) A function that cannot be implemented in the base class
C) A function that is used to achieve dynamic polymorphism
D) A special function used for dynamic memory allocation
Correct Answer: C

Q4. What is a pure virtual function?

A) A virtual function with no body defined in the base class
B) A function that deletes objects
C) A virtual function that cannot be overridden
D) A function defined in the derived class only
Correct Answer: A

Q5. What does overriding a function mean?

A) Changing the function definition in the same class
B) Providing a new implementation for a virtual function in a derived class
C) Changing the number of parameters in a function
D) Increasing the visibility of a function in the derived class
Correct Answer: B

Q6. What is dynamic binding?

A) Binding of a function call to its definition at compile time
B) The process of linking a function call with the function definition at runtime
C) Assigning a new value to a variable
D) Changing the type of an object dynamically
Correct Answer: B

Q7. Which feature in C++ enables creating a child class object through a parent class pointer?

A) Encapsulation
B) Inheritance
C) Polymorphism
D) Composition
Correct Answer: C

Q8. Function overloading is an example of:

A) Dynamic polymorphism
B) Static polymorphism
C) Encapsulation
D) Abstraction

Q9. Dynamic polymorphism is implemented by using:

A) Virtual functions
B) Function overloading
C) Templates
D) Operator overloading
Correct Answer: A

Q10. Which of the following correctly describes function overriding?

A) Functions that have the same name and different parameters in the same class
B) Functions that have different names but the same parameters in different classes
C) A derived class function that has the same signature as a base class function
D) Changing the default behavior of an operator
Correct Answer: C

Q11. Which concept is used when the exact type of the object is not known until runtime?

A) Static binding
B) Dynamic binding
C) Function overloading
D) Operator overloading
Correct Answer: B

Q12. The act of representing one form in multiple forms is known as:

A) Encapsulation
B) Inheritance
C) Polymorphism
D) Modularity
Correct Answer: C

Q13. In C++, if a function in the base class is declared as virtual, then:

A) The function cannot be overridden in the derived class
B) The function must be overridden in the derived class
C) The function can be overridden in the derived class
D) The derived class becomes abstract
Correct Answer: C

Q14. Which of the following statements is true about pure virtual functions?

A) A class with at least one pure virtual function is called a concrete class
B) Pure virtual functions can have a body
C) Pure virtual functions must be overridden by any non-abstract derived class
D) Pure virtual functions prevent dynamic binding

Q15. When using a parent class pointer to a child class object, which type of polymorphism is being utilized?

A) Static polymorphism
B) Dynamic polymorphism
C) Both A and B
D) None of the above
Correct Answer: B

Code Exercise: Zoo Animal Management System

This is a code exercise to explore polymorphism, focusing on virtual functions, pure virtual functions, function overloading and overriding, and dynamic vs. static binding in C++. This exercise will guide you through creating a simple class hierarchy for a zoo application, showcasing how these concepts are applied in practice.

Objective: Implement a system for managing different types of animals in a zoo using polymorphism. This system will demonstrate the use of virtual functions, pure virtual functions, function overloading and overriding, and the concepts of dynamic and static polymorphism.

Step 1: Base Class - Animal

- Create a base class named `Animal` with the following:
  - Protected data member: `name` (string).
  - Public methods: A parameterized constructor to initialize `name` , a virtual function `makeSound()` that prints a generic sound, and a pure virtual function `getType()` that returns the type of the animal as a string.

Step 2: Derived Classes - Lion and Bird

- Derive two classes from `Animal` : `Lion` and `Bird` .
  - For each derived class, implement the constructor, override `makeSound()` to print an animal-specific sound, and implement `getType()` to return the type of the animal.

Step 3: Demonstrate Static Polymorphism

- Within the `Animal` class, demonstrate static polymorphism by overloading a function named `info()`. Create two versions: one that takes no parameters and another that takes an integer age parameter.

Step 4: Dynamic Polymorphism

- In your `main()` function, create an array or vector of `Animal` pointers. Populate it with instances of `Lion` and `Bird`.
  - Loop through the array/vector and demonstrate dynamic polymorphism by calling `makeSound()` and `getType()` on each animal.

Step 5: Cleanup

- Remember to delete any dynamically allocated memory to prevent memory leaks.

Starter Code with TODOs Indicated:

```
#include <iostream>
#include <vector>
#include <string>

// Step 1: Define the Animal class
class Animal {
protected:
    std::string name;

public:
    // TODO: Implement a parameterized constructor for Animal
    // TODO: Declare and define a virtual function makeSound()
    // TODO: Declare a pure virtual function getType()

    // TODO: Demonstrate static polymorphism with an overloaded functi
on info()
};
```

```
// Step 2: Define the Lion and Bird classes derived from Animal
// TODO: Implement the Lion class with overridden makeSound() and getT
ype()
// TODO: Implement the Bird class with overridden makeSound() and getT
ype()

// Step 4: Demonstrate Dynamic Polymorphism in main
int main() {
    // TODO: Create an array or vector of Animal pointers
    // TODO: Populate it with Lion and Bird objects
    // TODO: Loop through and demonstrate polymorphism with makeSound
() and getType()
    // TODO: Cleanup dynamically allocated objects

    return 0;
}
```

Make sure that you have:

- **completed all the concepts**
- **ran all code examples yourself**
- **tried changing little things in the code to see the impact**
- **implemented the required program**

Please feel free to share your problems to ensure that your weekly tasks are completed and you are not staying behind

In [ ]: