Object Oriented Programming (Spring2024)

Week06 (18/19/20-March-2024)

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

Demonstrating Inheritance and Polymorphism: A Grocery Store Example

In the world of object-oriented programming (OOP), inheritance and polymorphism are fundamental concepts that enable more flexible and reusable code. To explore these concepts, we'll dive into a practical example set in a familiar environment: a grocery store. This scenario will help us understand how different items in the store, such as fruits and vegetables, can be represented in code while sharing common characteristics and behaviors, yet also demonstrating unique properties.

Inheritance allows us to define a general class (in our case, `GroceryItem`) with properties and methods common to all items in the grocery store, such as a name and price. Then, we can create more specific classes (`Fruit` and `Vegetable`) that inherit from this base class, adding their unique attributes (like `color` for fruits and `isOrganic` for vegetables) while also leveraging the shared features without needing to redefine them.

Polymorphism, on the other hand, gives us the flexibility to treat objects of these derived classes as objects of the base class, enabling us to write more general and reusable code. For example, we can use a base class pointer to refer to objects of any derived class, calling overridden methods like `displayInfo`, which will behave differently depending on the object's actual class. This allows us to manage a collection of mixed `GroceryItem` types (fruits, vegetables, etc.) seamlessly.

In this example, we'll implement a simple console application that creates instances of `Fruit` and `Vegetable`, demonstrating both inheritance (as they derive from `GroceryItem`) and polymorphism (through the use of virtual functions). This approach not only showcases how OOP principles can be applied to solve real-world problems but also emphasizes code reuse and the power of treating different objects through a common interface.

Let's see inheritance and polymorphism in action, making our grocery store inventory system organized and flexible.

Simplified Grocery Store Scenario

- **`GroceryItem` (Base Class):** Holds common attributes and declares a pure virtual function for displaying item information and a virtual function for updating the item's price.
- **`Fruit` (Derived Class):** Inherits from `GroceryItem` and implements the pure virtual function; it may also override other virtual functions.

- **`Vegetable` (Derived Class):** Similar to `Fruit`, it inherits from `GroceryItem` and provides its own implementation for the pure virtual function.

Step 1: Define the Base Class `GroceryItem`

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

class GroceryItem {
protected:
    string name;
    double price;



public:
    GroceryItem(const string& n, double p) : name(n), price(p) {}

    // Pure virtual function for displaying information
    virtual void displayInfo() const = 0;

    // Virtual function for updating the price
    virtual void updatePrice(double newPrice) {
        price = newPrice;
    }

    virtual ~GroceryItem() {}
};
```

Constructor Syntax Explained

- **`GroceryItem(const string& name, double price)`**:
  - This is the declaration of the constructor for the `GroceryItem` class. A constructor is a special member function that is automatically called when an object of the class is created.
  - The constructor takes two parameters: a constant reference to a `string` (`const string& name`), representing the name of the grocery item, and a `double` (`double price`), representing the price of the item. Using a reference to a `string` (with `const` to prevent modification) is efficient because it avoids copying the string when the constructor is called, yet ensures the original string passed in is not altered.

- **`: name(name), price(price) {}`**:
  - This part of the constructor is known as the member initializer list. It directly initializes the class's member variables with the values provided as arguments to the constructor.

- `name(name)` initializes the class's `name` member variable with the value of the `name` parameter passed to the constructor. Although both have the same name, the context makes their roles clear: the first `name` (before the parenthesis) refers to the member variable, while the second `name` (inside the parenthesis) refers to the parameter.
- Similarly, `price(price)` initializes the class's `price` member variable with the value of the `price` parameter.
- The use of the member initializer list is preferred in C++ for initializing members because it is more efficient than assignment inside the constructor body. It allows direct initialization of the members rather than default-initializing them first and then assigning them a value.
- The curly braces `{}` at the end of the member initializer list indicate the body of the constructor. In this case, the body is empty because all necessary initializations have been performed in the initializer list.

The code snippet

```
virtual void displayInfo() const = 0;
```

from the context of the `GroceryItem` class in C++ is a declaration of a pure virtual function. This line is crucial for establishing the polymorphic behavior of the class hierarchy in the grocery store system example. Let's break down each part of this declaration to understand its significance:

Pure Virtual Function Explained

- **`virtual`** :
  - The keyword `virtual` indicates that the function can be overridden in a derived class. It enables polymorphism, allowing the derived class to provide a specific implementation of the function that will be called based on the object's actual type, even when accessed through a pointer or reference to the base class. This keyword is the foundation of dynamic polymorphism in C++.

- **`void`** :
  - This specifies the return type of the `displayInfo` function. `void` means that this function does not return any value.
- **`displayInfo()`** :
  - This is the name of the function. In this context, it suggests a functionality that displays or prints information about the grocery item to some output (typically the console). The empty parentheses `()` indicate that this function takes no parameters.
- **`const`** :
  - Placing `const` after the function declaration means that the function is a constant member function. It cannot modify any non-static member variables of the class. This promises that calling `displayInfo` on a `GroceryItem` object will not alter that object's state, making it safe to call on `const` instances of `GroceryItem` .
- **`= 0;`** :
  - The `= 0` syntax at the end of the function declaration makes this function a pure virtual function. This declaration does not provide any implementation for the function within the `GroceryItem` class; instead, it indicates that any derived class must provide its

own implementation of the `displayInfo` function. A class with one or more pure virtual functions is considered an abstract class, meaning you cannot instantiate objects of this class directly. It serves as a base for other classes to build upon, ensuring they implement this essential function.

Virtual Function: `void updatePrice(double newPrice)`

- **`virtual`** :
  - The `virtual` keyword indicates that this function can be overridden in a derived class. This is a foundational aspect of polymorphism in C++, allowing for dynamic dispatch - meaning, the decision of which function version to call (the one in the base class or an overridden version in the derived class) is made at runtime based on the actual type of the object.

Return Type

- **`void`** :
  - This specifies the return type of the `updatePrice` function. `void` means that this function does not return any value. Its sole purpose is to perform an action, which, in this case, is updating the price of an item.

Function Name and Parameter

- **`updatePrice(double newPrice)`** :
  - `updatePrice` is the name of the function, indicating its purpose: to update or change the price of a grocery item.
  - The function takes a single parameter, `newPrice` , of type `double` . This parameter represents the new price that the grocery item should be updated to. Using `double` for currency can lead to precision issues in more complex financial applications, but it is common in simple examples for its ease of use.

Function Body

- **`{ price = newPrice; }`** :
  - The function body contains a single statement, `price = newPrice;` . This statement assigns the value of the `newPrice` parameter to the `price` member variable of the `GroceryItem` class. It effectively updates the price of the item to the new value provided.
  - The use of this function allows the price of an item to be changed after the object has been created, providing flexibility in managing grocery item data.

Virtual Destructor

- **`virtual`** :
  - The `virtual` keyword indicates that the destructor can be overridden in derived classes. This is crucial for polymorphism, particularly when objects are deleted through

pointers to the base class. It ensures that the appropriate destructor is called for the object being deleted, starting with the most derived class's destructor and then calling base class destructors in reverse order of inheritance.

Destructor

- **`~GroceryItem()`** :
  - The tilde `~` followed by the class name `GroceryItem` signifies a destructor in C++. A destructor is a special member function that is automatically called when an object goes out of scope or is explicitly deleted (for dynamically allocated objects). Its main purpose is to perform cleanup operations required by the class, such as releasing memory or other resources.
  - In the context of `GroceryItem` , there are no specific resources that need to be released, which is why the destructor's body is empty `{}` . However, declaring the destructor is still important for proper resource management in derived classes.

Empty Body

- **`{}`** :
  - The curly braces `{}` contain the definition of the destructor. In this case, the braces are empty because there's no specific cleanup required for `GroceryItem` objects. The class doesn't manage any dynamic memory or other resources that would require explicit release in the destructor.

Importance of Virtual Destructors

The declaration of a virtual destructor is essential in a base class for a few reasons:

1. **Memory Leak Prevention**: If a derived class allocates dynamic memory or acquires other resources that need to be released, not having a virtual destructor in the base class can lead to resource leaks. Deleting an object through a base class pointer without a virtual destructor will result in only the base class destructor being called, potentially skipping the cleanup logic in the derived class's destructor.
2. **Correct Cleanup Hierarchy**: The virtual destructor ensures that destructors for all classes in the inheritance hierarchy are called in the correct order, starting from the most derived class to the base class. This orderly cleanup is critical for the correctness of the program, especially when dealing with complex class hierarchies.

Step 2: Derived Class `Fruit`

```
class Fruit : public GroceryItem {
    string color;

public:
    Fruit(const string& name, double price, const string& color)
```

Constructor Declaration

- **`Fruit(const string& name, double price, const string& color) :`**
- This part declares the constructor of the `Fruit` class. The constructor is designed to take three parameters:
  - `const string& name` : A constant reference to a `string` representing the name of the fruit. Using a reference avoids copying the string, and `const` ensures the string cannot be modified by this constructor.
  - `double price` : A `double` representing the price of the fruit. The use of `double` allows for fractional values, common in representing currency.
  - `const string& color` : A constant reference to a `string` representing the color of the fruit. Like `name` , this uses a reference to avoid copying and is marked `const` to prevent modifications.

Member Initializer List

- **`: GroceryItem(name, price), color(color) {}`** :
  - This is the member initializer list, and it performs direct initialization of the base class (`GroceryItem`) and the member variable (`color`) with the values passed to the constructor.
    - `GroceryItem(name, price)` : This initializes the base class part of a `Fruit` object. The `name` and `price` arguments are passed to the constructor of the `GroceryItem` class, ensuring that these properties are correctly set in the base class portion of the object.
    - `color(color)` : This initializes the `color` member variable of the `Fruit` class with the `color` parameter passed to the constructor. The first `color` refers to the member variable, and the second `color` refers to the parameter. This is an example of parameter shadowing, where the local variable or parameter has the same name as a member variable.

Constructor Body

- **`{}`** :
  - The curly braces define the body of the constructor. In this case, the body is empty because all necessary initialization is performed in the member initializer list. This is a common practice in C++ when the constructor's sole purpose is to initialize member variables and no additional logic needs to be executed.

The code snippet from `Fruit` class:

```
void displayInfo() const override {
    cout << "Fruit Name: " << name << ", Price: $" << price << ", Colo
r: " << color <<endl; }
```

from the context of the `Fruit` class in C++ illustrates the overriding of a base class function to provide a specific implementation for `Fruit` objects. This method is a quintessential example of polymorphism in action. Let's dissect each part of this function definition:

Function Signature

- **`void`** :
  - This indicates the return type of the `displayInfo` function. In this case, `void` means that the function does not return any value. Its purpose is solely to perform an action, which is displaying information about the object.
- **`displayInfo()`** :
  - This is the name of the function. The function's goal, as indicated by its name, is to display (or print out) information about the `Fruit` object. The empty parentheses `()` signify that this function takes no parameters.
- **`const`** :
  - The `const` keyword at the end of the function declaration means that this is a constant member function. It cannot modify any member variables of the `Fruit` object for which it is called. This guarantees that calling `displayInfo` on a `Fruit` object is a read-only operation that does not alter the object's state, making it safe to call on `const` instances of `Fruit` .
- **`override`** :
  - The `override` keyword indicates that this function is intended to override a virtual function of the same signature in the base class (`GroceryItem` in this case). This keyword is part of C++11 and later standards, providing a clear indication that the function is overriding a base class function and allowing the compiler to check that the overridden base class function exists and matches in signature. This helps prevent common errors, such as mistyping the function name or parameters.

Function Body

- **`{ cout << "Fruit Name: " << name << ", Price: $" << price << ", Color: " << color <<endl; }`** :
  - The function body is enclosed in curly braces `{}` . Inside, it uses `cout` to print information about the `Fruit` object to the standard output (usually the console). This line constructs a formatted string containing the fruit's name, price, and color, which are member variables inherited from `GroceryItem` (for `name` and `price`) and defined within `Fruit` (for `color`).
  - `endl` is used to insert a newline character after the output and flush the output buffer, ensuring that the information is displayed immediately.

Step 3: Derived Class `Vegetable`

```
class Vegetable : public GroceryItem {
    bool isOrganic;

public:
    Vegetable(const string& name, double price, bool isOrganic)
        : GroceryItem(name, price), isOrganic(isOrganic) {}

    void displayInfo() const override {
        cout << "Vegetable Name: " << name << ", Price: $" << price <<
", Organic: " << (isOrganic ? "Yes" : "No")<<endl;
    }
};
```

Constructor Declaration

- **`Vegetable(const string& name, double price, bool isOrganic) :`**
- This is the declaration of the `Vegetable` class constructor. Constructors are special member functions that are called when a new instance of a class is created. This constructor takes three parameters:
  - `const string& name` : A constant reference to a `string` , representing the name of the vegetable. Using a reference avoids copying the string, and `const` ensures the original string passed in cannot be modified by the constructor.
  - `double price` : A `double` representing the price of the vegetable.
  - `bool isOrganic` : A `bool` indicating whether the vegetable is organic (`true`) or not (`false`).

Member Initializer List

- **`: GroceryItem(name, price), isOrganic(isOrganic) {}`** :
  - This part of the constructor is known as the member initializer list, which is used for directly initializing member variables and base class components of an object.
    - `GroceryItem(name, price)` : This initializes the base class (`GroceryItem`) part of the `Vegetable` object. The `GroceryItem` constructor is called with the `name` and `price` arguments provided to the `Vegetable` constructor. This demonstrates inheritance and how derived classes can utilize constructors of their base class to initialize parts of the object that are defined by the base class.
    - `isOrganic(isOrganic)` : This initializes the `isOrganic` member variable of the `Vegetable` class with the value of the `isOrganic` parameter passed to the constructor. It's an example of direct member initialization, where the member variable name and the constructor parameter name are the same. In such cases, the context (the member variable versus the constructor parameter) is clear from their usage.

Constructor Body

- **`{}`**:
  - The curly braces `{}` denote the body of the constructor. In this case, the body is empty because all necessary initializations are already done in the member initializer list. This is a common practice in C++ where the initializer list is preferred for initializing members and base classes because it can be more efficient and expressive, especially for complex types or when base class constructors need to be called with specific arguments.

The code snippet from `Vegetable` class:

```
    void displayInfo() const override {
        cout << "Vegetable Name: " << name << ", Price: $" << price <<
", Organic: " << (isOrganic ? "Yes" : "No")<<endl;
```

from the `Vegetable` class in C++ demonstrates an overriding function that outputs information specific to `Vegetable` objects. This method exemplifies the concept of polymorphism, particularly the use of method overriding. Let's analyze each component:

Function Declaration

- **`void`**:
  - This specifies the return type of the `displayInfo` function, indicating that it does not return any value.
- **`displayInfo()`**:
  - The function's name, which clearly states its purpose: to display or print out information about the object. The empty parentheses `()` indicate that this function takes no parameters.
- **`const`**:
  - This keyword at the end of the function declaration signifies that `displayInfo` is a constant member function. It guarantees that the function does not modify any of the object's member variables, ensuring it's safe to call on `const` instances of `Vegetable`. This property is crucial for functions intended only to read from and report on the object's state.
- **`override`**:
  - This keyword confirms that the function is overriding a virtual function with the same signature in the base class (`GroceryItem`). It's part of modern C++ (C++11 onwards), helping to catch errors at compile time, such as mismatches in the function signature or accidentally not overriding when intended. The compiler verifies that there is indeed a virtual function in one of the base classes that matches the override.

Function Body

- **`{ cout << "Vegetable Name: " << name << ", Price: $" << price << ", Organic: " << (isOrganic ? "Yes" : "No") <<endl; }`**:
  - The function body is encapsulated within curly braces `{}` . Within this block, the function utilizes `cout` for outputting a formatted message to the console. This message includes the vegetable's name, price, and whether it is organic.

- The `name` and `price` member variables are inherited from the `GroceryItem` base class and accessed directly here, demonstrating inheritance.
- The `isOrganic` member variable is specific to the `Vegetable` class and indicates whether the vegetable is organic. It's used in a conditional expression `(isOrganic ? "Yes" : "No")` to print "Yes" if `isOrganic` is true, and "No" otherwise, showcasing a simple way to include conditional logic in the output.
- `endl` is used at the end of the output line to insert a newline character and flush the output buffer, ensuring that the printed information appears immediately on the console.

Step 4: Demonstration of Inheritance and Polymorphism

```
int main() {
    Fruit apple("Apple", 1.99, "Red");
    Vegetable carrot("Carrot", 0.99, true);

    // Demonstrating polymorphism: A pointer to GroceryItem can point
to objects of derived classes
    GroceryItem* items[] = {&apple, &carrot};

    for (int i = 0; i < 2; ++i) {
        items[i]->displayInfo(); // Polymorphic call
    }



    // Updating price using a polymorphic call
    items[0]->updatePrice(2.49);
    items[1]->updatePrice(1.09);

    cout << "\nAfter updating prices:\n";
    for (int i = 0; i < 2; ++i) {
        items[i]->displayInfo(); // Polymorphic call to show updated p
rices
    }

    return 0;
}
```

The code snippet from `main()`

```
GroceryItem* items[] = {&apple, &carrot};
```

from the context of demonstrating inheritance and polymorphism in a grocery store system in C++ is an example of creating an array of pointers to `GroceryItem` objects. Let's dissect each part of this line to understand its functionality and significance:

Array of Pointers to Base Class

- **`GroceryItem* items[]`** :
  - This declares an array named `items` that can store pointers to `GroceryItem` objects. `GroceryItem*` indicates that each element of the array is a pointer to a `GroceryItem` instance. The `[]` denotes that `items` is an array, but without a specified size—the size is inferred from the number of elements in the initialization list that follows.

Initialization List

- **`= {&apple, &carrot};`** :
  - This part initializes the `items` array with specific values. The initialization list is enclosed in curly braces `{}` .
  - `&apple` and `&carrot` are the addresses of `apple` and `carrot` objects, respectively. The ampersand `&` is the address-of operator, which gets the memory address of an object, effectively creating a pointer to that object.
  - `apple` and `carrot` are assumed to be objects of derived classes from `GroceryItem` (`Fruit` and `Vegetable` , respectively, based on the context). This line demonstrates polymorphism in action: although `apple` and `carrot` are of different derived types, they can be treated as their base type (`GroceryItem`) through pointers. This allows for generic handling of various types of grocery items in the array.

Implications and Usage

This array of base class pointers is a fundamental concept in object-oriented programming, particularly for polymorphic behavior. By storing pointers to objects of derived classes in an array typed to the base class, you can:

- **Treat objects of different derived classes uniformly**: Access and manipulate `Fruit` , `Vegetable` , or any other `GroceryItem` -derived objects through a common interface (the `GroceryItem` base class). This is crucial for writing generic code that operates on collections of objects from a class hierarchy.
- **Invoke overridden methods polymorphically**: When methods are called on these pointers, the actual method that gets invoked is the one corresponding to the object's dynamic type (the derived class type of the object being pointed to), thanks to virtual function mechanics in C++. For instance, calling a virtual `displayInfo()` function on each element in the `items` array will execute the version of `displayInfo()` specific to each object's class (`Fruit` or `Vegetable`), even though the array type is of the base class `GroceryItem` .

This approach exemplifies the power of polymorphism in C++, enabling flexible and scalable design patterns, like treating all grocery items in a uniform way despite their specific differences.

The code snippet from `main()` :

```
items[0]->updatePrice(2.49);
items[1]->updatePrice(1.09);
```

performs operations on an array of pointers to `GroceryItem` objects, specifically updating the prices of the items at index 0 and index 1. Let's break down each aspect of these lines:

Array of Pointers

- **`items`** is an array of pointers to `GroceryItem` objects. In C++, arrays are a way to store multiple objects of the same type. When you have an array of pointers, like `GroceryItem* items[]`, each element of the array can store the address of a `GroceryItem` object. This allows for polymorphic behavior, as the actual objects pointed to can be instances of any class derived from `GroceryItem`.

Accessing Elements

- **`items[0]`** and **`items[1]`** access the first and second elements of the array, respectively. Since `items` is an array of pointers, `items[0]` and `items[1]` are pointers to `GroceryItem` objects. In this context, `items[0]` points to `apple`, and `items[1]` points to `carrot`.

Arrow Operator

- **`->`** is the arrow operator used in C++ to access members (methods or variables) of an object through a pointer. Here, it is used to call the `updatePrice` method on the objects that `items[0]` and `items[1]` point to.

Calling the Method

- **`updatePrice(2.49);`** and **`updatePrice(1.09);`** call the `updatePrice` method on the `apple` and `carrot` objects, respectively. The `updatePrice` method is designed to set the `price` member variable of the object to a new value. The value `2.49` is passed as the new price for the `apple`, and `1.09` for the `carrot`.

Method Parameters

- **`2.49`** and **`1.09`** are the arguments passed to the `updatePrice` method. They represent the new prices for the apple and carrot, respectively, in units consistent with the `price` variable (typically dollars or another currency).

Overall Explanation

- **Inheritance:** `Fruit` and `Vegetable` classes inherit from `GroceryItem`, gaining access to its protected attributes (`name`, `price`) and the obligation to implement its pure virtual function (`displayInfo`).
- **Polymorphism:** Through the use of virtual functions, `GroceryItem` pointers can invoke `displayInfo` on `Fruit` and `Vegetable` objects, allowing for different behaviors depending on the object's actual type. The `updatePrice` method demonstrates how a

base class's virtual method can be overridden (though not done here for simplicity) and called polymorphically.

- **Pure Virtual Function:** `displayInfo` is a pure virtual function in `GroceryItem`, making it an abstract class that cannot be instantiated. This enforces that derived classes must

Type *Markdown* and LaTeX: α^2

Type *Markdown* and LaTeX: α^2

Type *Markdown* and LaTeX: α^2

Type *Markdown* and LaTeX: α^2

Type *Markdown* and LaTeX: α^2

Type *Markdown* and LaTeX: α^2

Type *Markdown* and LaTeX: α^2

Type *Markdown* and LaTeX: α^2

Type *Markdown* and LaTeX: α^2

Type *Markdown* and LaTeX: α^2

Type *Markdown* and LaTeX: α^2

Type *Markdown* and LaTeX: α^2

Type *Markdown* and LaTeX: α^2

Type *Markdown* and LaTeX: α^2

Type *Markdown* and LaTeX: α^2

Type *Markdown* and LaTeX: α^2

Type *Markdown* and LaTeX: α^2

Type *Markdown* and LaTeX: α^2

Type *Markdown* and LaTeX: α^2

Type *Markdown* and LaTeX: α^2

Type *Markdown* and LaTeX: α^2

Type *Markdown* and LaTeX: α^2

Type *Markdown* and LaTeX: α^2

Type *Markdown* and LaTeX: α^2

Type *Markdown* and LaTeX: α^2

Type *Markdown* and LaTeX: α^2

Type *Markdown* and LaTeX: α^2

Type *Markdown* and LaTeX: α^2

Type *Markdown* and LaTeX: α^2

Type *Markdown* and LaTeX: α^2

Type *Markdown* and LaTeX: α^2

Type *Markdown* and LaTeX: α^2

Type *Markdown* and LaTeX: α^2

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 [ ]: