

Object Oriented Advanced

Basics

Syntax of Inheritance

```
#include <iostream>
using namespace std;
// Base class (Parent class)
class Base {
public:
    void parent_class_function() {
        cout << "This is the base class method." << endl;
    }
};
// Derived class (Child class) that inherits from the Base class publically
class Derived : public Base {
public:
    void child_class_function() {
        cout << "This is the derived class method." << endl;
    }
};
// Derived -> Base
int main() {
    Derived derived_obj;
    // Calling method from the base class
    derived_obj.parent_class_function();
    // Calling method from the derived class
    derived_obj.child_class_function();
    return 0;}
```

Advanced: Adapter pattern

We can get a little creative with the inheritance syntax and deploy an "Adapter pattern":

```
#include <iostream>
#include <string>
using namespace std;
// Existing class (Adaptee)
class LegacyMediaPlayer {
public:
    void play_mp3(const string& filename) {
        cout << "Playing MP3 file: " << filename << endl;
    }
};
// Target interface
class AdvancedMediaPlayer {
public:
    void play_mp4(const string& filename) {
        cout << "This player does not support MP4 files." << endl;
    }
    void play_vlc(const string& filename) {
        cout << "This player does not support VLC files." << endl;
    }
};
// Adapter class using private inheritance
class MediaPlayerAdapter : private LegacyMediaPlayer, public AdvancedMediaPlayer {
public:
    void play_mp4(const string& filename) {
        // For simplicity, convert MP4 to MP3 (dummy implementation)
        cout << "Converting MP4 to MP3..." << endl;
        LegacyMediaPlayer::play_mp3(filename + ".converted.mp3");
    }
    void play_vlc(const string& filename) {
        // For simplicity, convert VLC to MP3 (dummy implementation)
        cout << "Converting VLC to MP3..." << endl;
        LegacyMediaPlayer::play_mp3(filename + ".converted.mp3");
    }
};
```

```

    }
};
// Client code
int main() {
    MediaPlayerAdapter player;
    player.play_mp4("example.mp4");
    player.play_vlc("example.vlc");
    return 0;
}

```

Basic Inheritance pattern

Let us learn by studying an example.

```

#include <iostream>
#include <string>
using namespace std;
class Vehicle {
    // protected says that inheriting class will have access to these members
    // but users of the class will not
protected:
    string brand;
    int year;
    void protected_method() { std::cout << "protected method" << std::endl; }
public:
    Vehicle(string b, int y) : brand(b), year(y) {}
    virtual void display_info() {
        cout << "Brand: " << brand << ", Year: " << year << endl;
    }
};

void example() {
    Vehicle v("name", 2020);
    //v.brand = "123"; // Can't do because brand is private, only inheriting child class can access it
    //v.protected_method(); // Can't do this because this method is not public
}

class Car : public Vehicle {

```

```

private:
    int doors;
public:
    Car(string b, int y, int d) : Vehicle(b, y), doors(d) {}
    void display_info() override {
        cout << "Car - Brand: " << Vehicle::brand <<
            ", Year: " << Vehicle::year <<
            ", Doors: " << doors << endl;
    }
};

class Motorcycle : public Vehicle {
private:
    string type;
public:
    Motorcycle(string b, int y, string t) : Vehicle(b, y), type(t) {}
    void display_info() override {
        cout << "Motorcycle - Brand: " << brand << ", Year: " << year << ", Type: " << type << endl;
    }
};

int main() {
    Car car("Toyota", 2020, 4);
    Motorcycle motorcycle("Yamaha", 2019, "Sport");
    car.display_info();
    motorcycle.display_info();
    return 0;
}

```

Abstract classes

Override

A base class can define method(s) that could have override behavior:

```

class Animal {
public:
    virtual void speak() {

```

```

        std::cout << "Some generic animal sound" << endl;
    }
};

class another_animal : public Animal {
public:
    void speak() { /* Intent is to override the base, parent class */
        std::cout << "I speak differently" << std::endl;
    }
};

```

One question to ask in the code snippet above is that what happens if the base class changes its method name? In this case, the intent was for `another_animal` to override, but it no longer overrides anything. This is the reason the `override` keyword was invented.

```

class Animal {
public:
    //virtual void speak() { /* Case changes: */
    virtual void Speak() {
        std::cout << "Some generic animal sound" << endl;
    }
};

class another_animal : public Animal {
public:
    void speak() { /* This overrides the base class function */
        std::cout << "I speak differently" << std::endl;
    }
};

```

Let us use C++11 syntax to use the `override` keyword and see what happens when a method is intending to override a function but it doesn't do so:

```

class Animal {
public:
    //virtual void speak() { /* Case changes: */
    virtual void Speak() {
        std::cout << "Some generic animal sound" << endl;
    }
};

```

```
class another_animal : public Animal {
public:
    void speak() override { /* This overrides the base class function */
        std::cout << "I speak differently" << std::endl;
    };
};
```

Pure Virtual

A base class can provide default functionality or mandate that the child class inheriting the base (parent) class has to provide a certain functionality. It would look like this:

```
class Animal {
public:
    virtual void speak() = 0; /* I do not know how to speak, someone else tell me how */
};
class dolphin : public Animal {
public:
    void speak() override {
        std::cout << "I am a dolphin and I can whistle" << std::endl;
    };
};
```

Example 1

```
#include <iostream>
using namespace std;
class Animal {
public:
    virtual void speak() = 0;
};
class Dog : public Animal {
public:
    void speak() override { // Overriding the base class function
        cout << "Woof!" << endl;
    }
};
class Cat : public Animal {
```

```

public:
    void speak() override { // Overriding the base class function
        cout << "Meow!" << endl;
    }
};

void speak(Animal& animal) {
    animal.speak();
}

```

Example 2

```

#include <iostream>
#include <vector>
class Shape {
public:
    // Virtual function to be overridden by derived classes
    virtual void draw() const {
        std::cout << "Drawing a shape\n";
    }
    // Virtual destructor to ensure proper cleanup of derived objects
    virtual ~Shape() {}
};

class Circle : public Shape {
public:
    void draw() const override {
        std::cout << "Drawing a circle\n";
    }
};

class Rectangle : public Shape {
public:
    void draw() const override {
        std::cout << "Drawing a rectangle\n";
    }
};

int main() {
    // Create a vector of Shape pointers

```

```

std::vector<Shape*> shapes;
// Add different shapes to the vector
shapes.push_back(new Circle());
shapes.push_back(new Rectangle());
shapes.push_back(new Circle());
// Iterate through the vector and call the draw method
for (const auto& shape : shapes) {
    shape->draw(); // Polymorphic call
}
// Clean up the allocated memory
for (auto& shape : shapes) {
    delete shape;
}
return 0;}

```

Video Game Example

```

#include <iostream>
#include <vector>
#include <memory>
// Base class
class Character {
public:
    virtual void attack() const {
        std::cout << "Character attacks in a generic way.\n";
    }
    virtual ~Character() {}
};
// Derived class: Warrior
class Warrior : public Character {
public:
    void attack() const override {
        std::cout << "Warrior swings a sword!\n";
    }
};
// Derived class: Mage

```



```

class Mage : public Character {
public:
    void attack() const override {
        std::cout << "Mage casts a fireball!\n";
    }
};

// Derived class: Archer
class Archer : public Character {
public:
    void attack() const override {
        std::cout << "Archer shoots an arrow!\n";
    }
};

int main() {
    // Create a vector of unique_ptr to Character
    std::vector<std::unique_ptr<Character>> characters;
    // Add different types of characters to the vector
    characters.push_back(std::make_unique<Warrior>());
    characters.push_back(std::make_unique<Mage>());
    characters.push_back(std::make_unique<Archer>());
    // Iterate through the vector and call the attack method
    for (const auto& character : characters) {
        character->attack(); // Polymorphic call
    }
    // No need to delete characters; unique_ptr handles it automatically
    return 0;
}

```

Interfaces

An interface abstracts the implementation and enables unit-testing on the code. In other words, an interface is an abstract class that decouples the implementation from the interface, which allows changing or adding new implementations without affecting the rest of the program.

Polymorphism: The code below demonstrates polymorphism through the use of the interface. The

`test_communication` function can take any object that implements the Communicable interface, showcasing how polymorphism enables the writing of more flexible and reusable code.

Example 1

In the example below, the interface ensures that any class that implements it will have a `send` method, but does not commit to how the sending is done, which is polymorphic.

```
#include <iostream>

// Interface class with a single responsibility
class Communicable {
public:
    virtual void send(const std::string& message) = 0; // Only one pure virtual function
    virtual ~Communicable() {} // Virtual destructor for proper cleanup
};

class WiFiDevice : public Communicable {
public:
    void send(const std::string& message) override {
        std::cout << "Sending over WiFi: " << message << std::endl;
    }
};

class BluetoothDevice : public Communicable {
public:
    void send(const std::string& message) override {
        std::cout << "Sending over Bluetooth: " << message << std::endl;
    }
};

void test_communication(Communicable& device, const std::string& message) {
    device.send(message); // Use the send method to demonstrate communication
}

int main() {
    WiFiDevice wifi;
    BluetoothDevice bluetooth;
    testCommunication(wifi, "Hello WiFi World");
    testCommunication(bluetooth, "Hello Bluetooth World");
    return 0;
}
```

Example 2

```

#include <iostream>
#include <vector>
#include <memory>
// Interface class
class GameCharacter {
public:
    virtual void attack() = 0;           // Pure virtual function
    virtual void defend() = 0;          // Pure virtual function
    virtual void heal() = 0;            // Pure virtual function
    virtual int get_health() const = 0; // Pure virtual function to get health
    virtual ~GameCharacter() {}
};

void perform_action(GameCharacter& character) {
    if (character.get_health() < 30) {
        character.heal();
    } else if (rand() % 2 == 0) { // 50% chance to attack or defend
        character.attack();
    } else {
        character.defend();
    }
}

```

Exercise

Let us continue working on the examples below and practice more uses of interfaces.

1. Create an interface that returns low threshold level for health
2. Create an interface that returns when a character should attack
3. Write two classes that inherit and implement the GameCharacter class
4. Write a main function to invoke `perform_action()` on the two classes

Revision #11

Created 1 month ago by [Preet Kang](#)

Updated 1 month ago by [Preet Kang](#)