



Decorator in C++

Decorator is a structural pattern that allows adding new behaviors to objects dynamically by placing them inside special wrapper objects, called *decorators*.

Using decorators you can wrap objects countless number of times since both target objects and decorators follow the same interface. The resulting object will get a stacking behavior of all wrappers.

[Learn more about Decorator](#)

Complexity:

Popularity:

Usage examples: The Decorator is pretty standard in C++ code, especially in code related to streams.

Identification: Decorator can be recognized by creation methods or constructors that accept objects of the same class or interface as a current class.

Conceptual Example

This example illustrates the structure of the **Decorator** design pattern. It focuses on answering these questions:

- What classes does it consist of?
- What roles do these classes play?
- In what way the elements of the pattern are related?

main.cc: Conceptual example

```

/**
 * The base Component interface defines operations that can be altered by
 * decorators.
 */
class Component {
public:
    virtual ~Component() {}
    virtual std::string Operation() const = 0;
};

/**
 * Concrete Components provide default implementations of the operations. There
 * might be several variations of these classes.
 */
class ConcreteComponent : public Component {
public:
    std::string Operation() const override {
        return "ConcreteComponent";
    }
};

/**
 * The base Decorator class follows the same interface as the other components.
 * The primary purpose of this class is to define the wrapping interface for all
 * concrete decorators. The default implementation of the wrapping code might
 * include a field for storing a wrapped component and the means to initialize
 * it.
 */
class Decorator : public Component {
    /**
     * @var Component
     */
protected:
    Component* component_;

public:
    Decorator(Component* component) : component_(component) {}
};

/**
 * The Decorator delegates all work to the wrapped component.
 */
std::string Operation() const override {
    return this->component_->Operation();
}
};

/**
 * Concrete Decorators call the wrapped object and alter its result in some way.
 */
class ConcreteDecoratorA : public Decorator {
    /**
     * Decorators may call parent implementation of the operation, instead of
     * calling the wrapped object directly. This approach simplifies extension of
     * decorator classes.
     */

```

```

    */
public:
    ConcreteDecoratorA(Component* component) : Decorator(component) {
    }
    std::string Operation() const override {
        return "ConcreteDecoratorA(" + Decorator::Operation() + ")";
    }
};
/**
 * Decorators can execute their behavior either before or after the call to a
 * wrapped object.
 */
class ConcreteDecoratorB : public Decorator {
public:
    ConcreteDecoratorB(Component* component) : Decorator(component) {
    }

    std::string Operation() const override {
        return "ConcreteDecoratorB(" + Decorator::Operation() + ")";
    }
};
/**
 * The client code works with all objects using the Component interface. This
 * way it can stay independent of the concrete classes of components it works
 * with.
 */
void ClientCode(Component* component) {
    // ...
    std::cout << "RESULT: " << component->Operation();
    // ...
}

int main() {
    /**
     * This way the client code can support both simple components...
     */
    Component* simple = new ConcreteComponent;
    std::cout << "Client: I've got a simple component:\n";
    ClientCode(simple);
    std::cout << "\n\n";
    /**
     * ...as well as decorated ones.
     *
     * Note how decorators can wrap not only simple components but the other
     * decorators as well.
     */
    Component* decorator1 = new ConcreteDecoratorA(simple);
    Component* decorator2 = new ConcreteDecoratorB(decorator1);
    std::cout << "Client: Now I've got a decorated component:\n";
    ClientCode(decorator2);
    std::cout << "\n";
}

```

```
delete simple;  
delete decorator1;  
delete decorator2;  
  
return 0;  
}
```

Output.txt: Execution result

Client: I've got a simple component:

RESULT: ConcreteComponent

Client: Now I've got a decorated component:

RESULT: ConcreteDecoratorB(ConcreteDecoratorA(ConcreteComponent))