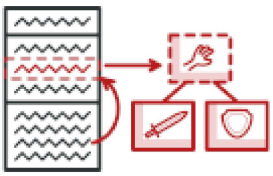

Strategy in C++

Strategy is a behavioral design pattern that turns a set of behaviors into objects and makes them interchangeable inside original context object.

The original object, called context, holds a reference to a strategy object. The context delegates executing the behavior to the linked strategy object. In order to change the way the context performs its work, other objects may replace the currently linked strategy object with another one.

Learn more about Strategy

Complexity:

Popularity:

Usage examples: The Strategy pattern is very common in C++ code. It's often used in various frameworks to provide users a way to change the behavior of a class without extending it.

Identification: Strategy pattern can be recognized by a method that lets a nested object do the actual work, as well as a setter that allows replacing that object with a different one.

Conceptual Example

This example illustrates the structure of the **Strategy** 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 Strategy interface declares operations common to all supported versions
 * of some algorithm.
 *
```

```
 * The Context uses this interface to call the algorithm defined by Concrete
 * Strategies.
 */
class Strategy
{
public:
    virtual ~Strategy() = default;
    virtual std::string doAlgorithm(std::string_view data) const = 0;
};

/**
 * The Context defines the interface of interest to clients.
 */

class Context
{
    /**
     * @var Strategy The Context maintains a reference to one of the Strategy
     * objects. The Context does not know the concrete class of a strategy. It
     * should work with all strategies via the Strategy interface.
     */
private:
    std::unique_ptr<Strategy> strategy_;
    /**
     * Usually, the Context accepts a strategy through the constructor, but also
     * provides a setter to change it at runtime.
     */
public:
    explicit Context(std::unique_ptr<Strategy> &&strategy = {}) : strategy_(std::move(strateg
    {
    }
    /**
     * Usually, the Context allows replacing a Strategy object at runtime.
     */
    void set_strategy(std::unique_ptr<Strategy> &&strategy)
    {
        strategy_ = std::move(strategy);
    }
    /**
     * The Context delegates some work to the Strategy object instead of
     * implementing +multiple versions of the algorithm on its own.
     */
    void doSomeBusinessLogic() const
    {
        if (strategy_) {
            std::cout << "Context: Sorting data using the strategy (not sure how it'll do it)
            std::string result = strategy_->doAlgorithm("aecbd");
            std::cout << result << "\n";
        } else {
            std::cout << "Context: Strategy isn't set\n";
        }
    }
```

```
};

/**
 * Concrete Strategies implement the algorithm while following the base Strategy
 * interface. The interface makes them interchangeable in the Context.
 */
class ConcreteStrategyA : public Strategy
{
public:
    std::string doAlgorithm(std::string_view data) const override
    {
        std::string result(data);
        std::sort(std::begin(result), std::end(result));

        return result;
    }
};
class ConcreteStrategyB : public Strategy
{
    std::string doAlgorithm(std::string_view data) const override
    {
        std::string result(data);
        std::sort(std::begin(result), std::end(result), std::greater<>());

        return result;
    }
};
/**
 * The client code picks a concrete strategy and passes it to the context. The
 * client should be aware of the differences between strategies in order to make
 * the right choice.
 */

void clientCode()
{
    Context context(std::make_unique<ConcreteStrategyA>());
    std::cout << "Client: Strategy is set to normal sorting.\n";
    context.doSomeBusinessLogic();
    std::cout << "\n";
    std::cout << "Client: Strategy is set to reverse sorting.\n";
    context.set_strategy(std::make_unique<ConcreteStrategyB>());
    context.doSomeBusinessLogic();
}

int main()
{
    clientCode();
    return 0;
}
```

Output.txt: Execution result

```
Client: Strategy is set to normal sorting.
Context: Sorting data using the strategy (not sure how it'll do it)
abcde

Client: Strategy is set to reverse sorting.
Context: Sorting data using the strategy (not sure how it'll do it)
edcba
```