Facade in C++

Facade is a structural design pattern that provides a simplified (but limited) interface to a complex system of classes, library or framework.

While Facade decreases the overall complexity of the application, it also helps to move unwanted dependencies to one place.

Learn more about Facade

Complexity:

Popularity:

Usage examples: The Facade pattern is commonly used in apps written in C++. It's especially handy when working with complex libraries and APIs.

Identification: Facade can be recognized in a class that has a simple interface, but delegates most of the work to other classes. Usually, facades manage the full life cycle of objects they use.

Conceptual Example

This example illustrates the structure of the **Facade** 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 Subsystem can accept requests either from the facade or client directly.
 * In any case, to the Subsystem, the Facade is yet another client, and it's not
 * a part of the Subsystem.
 */
```

```
class Subsystem1 {
 public:
  std::string Operation1() const {
    return "Subsystem1: Ready!\n";
  }
  // ...
  std::string OperationN() const {
    return "Subsystem1: Go!\n";
  }
};
/**
 * Some facades can work with multiple subsystems at the same time.
 */
class Subsystem2 {
 public:
  std::string Operation1() const {
    return "Subsystem2: Get ready!\n";
  }
  // ...
  std::string OperationZ() const {
    return "Subsystem2: Fire!\n";
  }
};

/**
 * The Facade class provides a simple interface to the complex logic of one or
 * several subsystems. The Facade delegates the client requests to the
 * appropriate objects within the subsystem. The Facade is also responsible for
 * managing their lifecycle. All of this shields the client from the undesired
 * complexity of the subsystem.
 */
class Facade {
 protected:
  Subsystem1 *subsystem1_;
  Subsystem2 *subsystem2_;
  /**
   * Depending on your application's needs, you can provide the Facade with
   * existing subsystem objects or force the Facade to create them on its own.
   */
 public:
  /**
   * In this case we will delegate the memory ownership to Facade Class
   */
  Facade(
      Subsystem1 *subsystem1 = nullptr,
      Subsystem2 *subsystem2 = nullptr) {
    this->subsystem1_ = subsystem1 ?: new Subsystem1;
    this->subsystem2_ = subsystem2 ?: new Subsystem2;
  }
  ~Facade() {
    delete subsystem1_;
    delete subsystem2_;
```

```
  }
  /**
   * The Facade's methods are convenient shortcuts to the sophisticated
   * functionality of the subsystems. However, clients get only to a fraction of
   * a subsystem's capabilities.
   */
  std::string Operation() {
    std::string result = "Facade initializes subsystems:\n";
    result += this->subsystem1_->Operation1();
    result += this->subsystem2_->Operation1();
    result += "Facade orders subsystems to perform the action:\n";
    result += this->subsystem1_->OperationN();
    result += this->subsystem2_->OperationZ();
    return result;
  }
};

/**
 * The client code works with complex subsystems through a simple interface
 * provided by the Facade. When a facade manages the lifecycle of the subsystem,
 * the client might not even know about the existence of the subsystem. This
 * approach lets you keep the complexity under control.
 */
void ClientCode(Facade *facade) {
  // ...
  std::cout << facade->Operation();
  // ...
}
/**
 * The client code may have some of the subsystem's objects already created. In
 * this case, it might be worthwhile to initialize the Facade with these objects
 * instead of letting the Facade create new instances.
 */

int main() {
  Subsystem1 *subsystem1 = new Subsystem1;
  Subsystem2 *subsystem2 = new Subsystem2;
  Facade *facade = new Facade(subsystem1, subsystem2);
  ClientCode(facade);

  delete facade;

  return 0;
}
```

Output.txt: Execution result

```
Facade initializes subsystems:
Subsystem1: Ready!
Subsystem2: Get ready!
Facade orders subsystems to perform the action:
Subsystem1: Go!
Subsystem2: Fire!
```