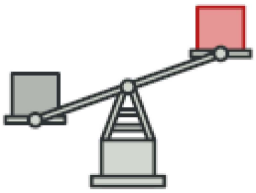

Flyweight in C++

Flyweight is a structural design pattern that allows programs to support vast quantities of objects by keeping their memory consumption low.

The pattern achieves it by sharing parts of object state between multiple objects. In other words, the Flyweight saves RAM by caching the same data used by different objects.

Learn more about Flyweight

Complexity:

Popularity:

Usage examples: The Flyweight pattern has a single purpose: minimizing memory intake. If your program doesn't struggle with a shortage of RAM, then you might just ignore this pattern for a while.

Identification: Flyweight can be recognized by a creation method that returns cached objects instead of creating new.

Conceptual Example

This example illustrates the structure of the **Flyweight** 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

```
/**
 * Flyweight Design Pattern
 *
 * Intent: Lets you fit more objects into the available amount of RAM by sharing
 * common parts of state between multiple objects, instead of keeping all of the
 * data in each object.
 */

struct SharedState
{
    std::string brand_;
    std::string model_;
    std::string color_;

    SharedState(const std::string &brand, const std::string &model, const std::string &color)
        : brand_(brand), model_(model), color_(color)
    {
    }

    friend std::ostream &operator<<(std::ostream &os, const SharedState &ss)
    {
        return os << "[ " << ss.brand_ << " , " << ss.model_ << " , " << ss.color_ << " ]";
    }
};

struct UniqueState
{
    std::string owner_;
    std::string plates_;

    UniqueState(const std::string &owner, const std::string &plates)
        : owner_(owner), plates_(plates)
    {
    }

    friend std::ostream &operator<<(std::ostream &os, const UniqueState &us)
    {
        return os << "[ " << us.owner_ << " , " << us.plates_ << " ]";
    }
};

/**
 * The Flyweight stores a common portion of the state (also called intrinsic
 * state) that belongs to multiple real business entities. The Flyweight accepts
 * the rest of the state (extrinsic state, unique for each entity) via its
 * method parameters.
 */
class Flyweight
{
private:
    SharedState *shared_state_;
```

```
public:
    Flyweight(const SharedState *shared_state) : shared_state_(new SharedState(*shared_state)
    {
    }
    Flyweight(const Flyweight &other) : shared_state_(new SharedState(*other.shared_state_))
    {
    }
    ~Flyweight()
    {
        delete shared_state_;
    }
    SharedState *shared_state() const
    {
        return shared_state_;
    }
    void Operation(const UniqueState &unique_state) const
    {
        std::cout << "Flyweight: Displaying shared (" << *shared_state_ << ") and unique (" <
    }
};
/**
 * The Flyweight Factory creates and manages the Flyweight objects. It ensures
 * that flyweights are shared correctly. When the client requests a flyweight,
 * the factory either returns an existing instance or creates a new one, if it
 * doesn't exist yet.
 */
class FlyweightFactory
{
    /**
     * @var Flyweight[]
     */
private:
    std::unordered_map<std::string, Flyweight> flyweights_;
    /**
     * Returns a Flyweight's string hash for a given state.
     */
    std::string GetKey(const SharedState &ss) const
    {
        return ss.brand_ + "_" + ss.model_ + "_" + ss.color_;
    }

public:
    FlyweightFactory(std::initializer_list<SharedState> share_states)
    {
        for (const SharedState &ss : share_states)
        {
            this->flyweights_.insert(std::make_pair<std::string, Flyweight>(this->GetKey(ss),
        }
    }

    /**
```

```
     * Returns an existing Flyweight with a given state or creates a new one.
     */
    Flyweight GetFlyweight(const SharedState &shared_state)
    {
        std::string key = this->GetKey(shared_state);
        if (this->flyweights_.find(key) == this->flyweights_.end())
        {
            std::cout << "FlyweightFactory: Can't find a flyweight, creating new one.\n";
            this->flyweights_.insert(std::make_pair(key, Flyweight(&shared_state)));
        }
        else
        {
            std::cout << "FlyweightFactory: Reusing existing flyweight.\n";
        }
        return this->flyweights_.at(key);
    }
    void ListFlyweights() const
    {
        size_t count = this->flyweights_.size();
        std::cout << "\nFlyweightFactory: I have " << count << " flyweights:\n";
        for (std::pair<std::string, Flyweight> pair : this->flyweights_)
        {
            std::cout << pair.first << "\n";
        }
    }
};

// ...
void AddCarToPoliceDatabase(
    FlyweightFactory &ff, const std::string &plates, const std::string &owner,
    const std::string &brand, const std::string &model, const std::string &color)
{
    std::cout << "\nClient: Adding a car to database.\n";
    const Flyweight &flyweight = ff.GetFlyweight({brand, model, color});
    // The client code either stores or calculates extrinsic state and passes it
    // to the flyweight's methods.
    flyweight.Operation({owner, plates});
}

/**
 * The client code usually creates a bunch of pre-populated flyweights in the
 * initialization stage of the application.
 */

int main()
{
    FlyweightFactory *factory = new FlyweightFactory({{"Chevrolet", "Camaro2018", "pink"}, {"
    factory->ListFlyweights();

    AddCarToPoliceDatabase(*factory,
                           "CL234IR",
                           "James Doe",
```

```
                            "BMW",
                            "M5",
                            "red");

    AddCarToPoliceDatabase(*factory,
                            "CL234IR",
                            "James Doe",
                            "BMW",
                            "X1",
                            "red");
    factory->ListFlyweights();
    delete factory;

    return 0;
}
```

Output.txt: Execution result

```
FlyweightFactory: I have 5 flyweights:
BMW_X6_white
Mercedes Benz_C500_red
Mercedes Benz_C300_black
BMW_M5_red
Chevrolet_Camaro2018_pink

Client: Adding a car to database.
FlyweightFactory: Reusing existing flyweight.
Flyweight: Displaying shared ([ BMW , M5 , red ]) and unique ([ CL234IR , James Doe ]) state.

Client: Adding a car to database.
FlyweightFactory: Can't find a flyweight, creating new one.
Flyweight: Displaying shared ([ BMW , X1 , red ]) and unique ([ CL234IR , James Doe ]) state.

FlyweightFactory: I have 6 flyweights:
BMW_X1_red
Mercedes Benz_C300_black
BMW_X6_white
Mercedes Benz_C500_red
BMW_M5_red
Chevrolet_Camaro2018_pink
```