Open-Closed Principle (OCP) in C++

Definition:

- A class should be **open for extension** (i.e., new functionality can be added).
- A class should be **closed for modification** (i.e., existing code should not be changed).

Example Scenario: Product Filtering System

1. Initial Problem:

- We need to filter products based on their **color** and **size**.
- A naive approach would be to add new filtering functions every time a new criterion is required.
- This violates OCP because we modify the existing class every time a new requirement comes in.

2. Bad Implementation (Violates OCP)

The `ProductFilter` class has separate functions for:

- `by_color()`
- `by_size()`
- `by_size_and_color()`

```
struct ProductFilter
{
  typedef vector<Product*> Items;

  Items by_color(Items items, const Color color)
  {
    Items result;
    for (auto& i : items)
      if (i->color == color)
        result.push_back(i);
    return result;
  }
};
```

Issues:

- Adding a new criterion (e.g., weight, material) requires modifying the class.
- Each combination of filters requires a new function.
- Maintenance becomes difficult and code duplication increases.

Better Approach: Using the Specification Pattern

To follow **OCP**, we use the **Specification Pattern**:

1. Define an abstract `Specification<T>` interface with `is_satisfied()`.
2. Define an abstract `Filter<T>` interface.
3. Implement concrete **specifications** for filtering by color, size, etc.
4. Implement `BetterFilter` that uses the specification.

Step 1: Create the Specification Interface

```
template <typename T>
struct Specification
{
  virtual ~Specification() = default;
  virtual bool is_satisfied(T* item) const = 0;
};
```

Step 2: Create the Filter Interface

```
template <typename T>
struct Filter
{
  virtual vector<T*> filter(vector<T*> items, Specification<T>& spec) = 0;
};
```

Step 3: Implement a Better Filter

```
struct BetterFilter : Filter<Product>
{
  vector<Product*> filter(vector<Product*> items,
                        Specification<Product> &spec) override
  {
    vector<Product*> result;
    for (auto& p : items)
      if (spec.is_satisfied(p))
        result.push_back(p);
    return result;
  }
};
```

Step 4: Implement Concrete Specifications

Filter by Color

```
struct ColorSpecification : Specification<Product>
{
  Color color;
  ColorSpecification(Color color) : color(color) {}

  bool is_satisfied(Product *item) const override {
    return item->color == color;
  }
};
```

Filter by Size

```
struct SizeSpecification : Specification<Product>
{
  Size size;
  explicit SizeSpecification(const Size size) : size(size) {}

  bool is_satisfied(Product* item) const override {
    return item->size == size;
  }
};
```

Combining Multiple Specifications

To support filtering by **multiple criteria** (e.g., "green and large"), we introduce an **AndSpecification**.

```
template <typename T>
struct AndSpecification : Specification<T>
{
  const Specification<T>& first;
  const Specification<T>& second;

  AndSpecification(const Specification<T>& first, const Specification<T>& second)
    : first(first), second(second) {}

  bool is_satisfied(T *item) const override {
    return first.is_satisfied(item) && second.is_satisfied(item);
  }
};
```

Using the Better Filter

```
int main()
{
  Product apple{"Apple", Color::green, Size::small};
  Product tree{"Tree", Color::green, Size::large};
  Product house{"House", Color::blue, Size::large};

  const vector<Product*> all { &apple, &tree, &house };

  BetterFilter bf;
  ColorSpecification green(Color::green);
  auto green_things = bf.filter(all, green);

  for (auto& x : green_things)
    cout << x->name << " is green\n";
}
```

Output:

```
Apple is green
Tree is green
```

Enhancing Readability with Operators

Instead of manually creating `AndSpecification`, we define an **operator overload**:

```
template <typename T>
AndSpecification<T> operator&&(const Specification<T>& first, const Specification<T>& second)
{
  return { first, second };
}
```

Now, we can write:

```
auto spec = green && large;
```

Instead of:

```
AndSpecification<Product> green_and_large(green, large);
```

Key Takeaways

1. **Avoid modifying existing code** when adding new features.
2. Use **inheritance and polymorphism** to extend functionality.
3. The **Specification Pattern** allows flexible filtering.
4. Operators (`&&`) improve readability and reduce boilerplate code.

By following **OCP**, our filtering system can be easily extended to **other attributes** (e.g., weight, material) **without modifying existing code**.

Potential Issue: Undefined Behavior

While implementing `AndSpecification`, one must be cautious when using temporary objects.

Undefined Behavior - Use After Free

The following expression may cause undefined behavior:

```
auto spec = ColorSpecification{Color::green} &&
SizeSpecification{Size::large};
```

Reason:

- `AndSpecification` holds references to temporary objects.
- These objects are destroyed after evaluation, leading to **use-after-free**.
- Some compilers may optimize it out, but others will crash or behave unexpectedly.

Solutions:

1. **Store Specifications as Variables**

   ```
   ColorSpecification green(Color::green);

   SizeSpecification large(Size::large);

   auto spec = green && large;
   ```

   This ensures the objects persist in memory.

2. **Use Smart Pointers (std::shared_ptr)**

   ```
   auto spec = std::make_shared<ColorSpecification>(Color::green) &&

          std::make_shared<SizeSpecification>(Size::large);
   ```

   This avoids dangling references and ensures memory safety.

3. **Avoid Overloading `&&` Operator**

   - Overloading `&&` leads to **loss of short-circuit evaluation**.
   - Instead, use a **variadic template function** to combine specifications.