Factory Method in C++

Factory method is a creational design pattern which solves the problem of creating product objects without specifying their concrete classes.

The Factory Method defines a method, which should be used for creating objects instead of using a direct constructor call (`new` operator). Subclasses can override this method to change the class of objects that will be created.

If you can't figure out the difference between various factory patterns and concepts, then read our **Factory Comparison**.

Learn more about Factory Method

Complexity:

Popularity:

Usage examples: The Factory Method pattern is widely used in C++ code. It's very useful when you need to provide a high level of flexibility for your code.

Identification: Factory methods can be recognized by creation methods that construct objects from concrete classes. While concrete classes are used during the object creation, the return type of the factory methods is usually declared as either an abstract class or an interface.

Conceptual Example

This example illustrates the structure of the **Factory Method** 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 Product interface declares the operations that all concrete products must
 * implement.
 */

class Product {
 public:
  virtual ~Product() {}
  virtual std::string Operation() const = 0;
};

/**
 * Concrete Products provide various implementations of the Product interface.
 */
class ConcreteProduct1 : public Product {
 public:
  std::string Operation() const override {
    return "{Result of the ConcreteProduct1}";
  }
};
class ConcreteProduct2 : public Product {
 public:
  std::string Operation() const override {
    return "{Result of the ConcreteProduct2}";
  }
};

/**
 * The Creator class declares the factory method that is supposed to return an
 * object of a Product class. The Creator's subclasses usually provide the
 * implementation of this method.
 */

class Creator {
  /**
   * Note that the Creator may also provide some default implementation of the
   * factory method.
   */
 public:
  virtual ~Creator(){};
  virtual Product* FactoryMethod() const = 0;
  /**
   * Also note that, despite its name, the Creator's primary responsibility is
   * not creating products. Usually, it contains some core business logic that
   * relies on Product objects, returned by the factory method. Subclasses can
   * indirectly change that business logic by overriding the factory method and
   * returning a different type of product from it.
   */

  std::string SomeOperation() const {
    // Call the factory method to create a Product object.
    Product* product = this->FactoryMethod();
    // Now, use the product.
    std::string result = "Creator: The same creator's code has just worked with " + product->
    delete product;
    return result;
  }
};

/**
 * Concrete Creators override the factory method in order to change the
 * resulting product's type.
 */
class ConcreteCreator1 : public Creator {
  /**
   * Note that the signature of the method still uses the abstract product type,
   * even though the concrete product is actually returned from the method. This
   * way the Creator can stay independent of concrete product classes.
   */
 public:
  Product* FactoryMethod() const override {
    return new ConcreteProduct1();
  }
};

class ConcreteCreator2 : public Creator {
 public:
  Product* FactoryMethod() const override {
    return new ConcreteProduct2();
  }
};

/**
 * The client code works with an instance of a concrete creator, albeit through
 * its base interface. As long as the client keeps working with the creator via
 * the base interface, you can pass it any creator's subclass.
 */
void ClientCode(const Creator& creator) {
  // ...
  std::cout << "Client: I'm not aware of the creator's class, but it still works.\n"
            << creator.SomeOperation() << std::endl;
  // ...
}

/**
 * The Application picks a creator's type depending on the configuration or
 * environment.
 */

int main() {
  std::cout << "App: Launched with the ConcreteCreator1.\n";
  Creator* creator = new ConcreteCreator1();
  ClientCode(*creator);
  std::cout << std::endl;
  std::cout << "App: Launched with the ConcreteCreator2.\n";
  Creator* creator2 = new ConcreteCreator2();
  ClientCode(*creator2);

  delete creator;
  delete creator2;
  return 0;
}
```

Output.txt: Execution result

```
App: Launched with the ConcreteCreator1.
Client: I'm not aware of the creator's class, but it still works.
Creator: The same creator's code has just worked with {Result of the ConcreteProduct1}

App: Launched with the ConcreteCreator2.
Client: I'm not aware of the creator's class, but it still works.
Creator: The same creator's code has just worked with {Result of the ConcreteProduct2}
```