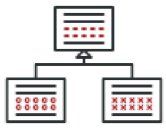

Template Method in C++

Template Method is a behavioral design pattern that allows you to define a skeleton of an algorithm in a base class and let subclasses override the steps without changing the overall algorithm's structure.

Learn more about Template Method

Complexity:

Popularity:

Usage examples: The Template Method pattern is quite common in C++ frameworks. Developers often use it to provide framework users with a simple means of extending standard functionality using inheritance.

Identification: Template Method can be recognized if you see a method in base class that calls a bunch of other methods that are either abstract or empty.

Conceptual Example

This example illustrates the structure of the **Template 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 Abstract Class defines a template method that contains a skeleton of some
```

```
 * algorithm, composed of calls to (usually) abstract primitive operations.
 *
 * Concrete subclasses should implement these operations, but leave the template
 * method itself intact.
 */
class AbstractClass {
  /**
   * The template method defines the skeleton of an algorithm.
   */
 public:
  void TemplateMethod() const {
    this->BaseOperation1();
    this->RequiredOperations1();
    this->BaseOperation2();
    this->Hook1();
    this->RequiredOperation2();
    this->BaseOperation3();
    this->Hook2();
  }
  /**
   * These operations already have implementations.
   */
 protected:
  void BaseOperation1() const {
    std::cout << "AbstractClass says: I am doing the bulk of the work\n";
  }
  void BaseOperation2() const {
    std::cout << "AbstractClass says: But I let subclasses override some operations\n";
  }
  void BaseOperation3() const {
    std::cout << "AbstractClass says: But I am doing the bulk of the work anyway\n";
  }
  /**
   * These operations have to be implemented in subclasses.
   */
  virtual void RequiredOperations1() const = 0;
  virtual void RequiredOperation2() const = 0;
  /**
   * These are "hooks." Subclasses may override them, but it's not mandatory
   * since the hooks already have default (but empty) implementation. Hooks
   * provide additional extension points in some crucial places of the
   * algorithm.
   */
  virtual void Hook1() const {}
  virtual void Hook2() const {}
};
/**
 * Concrete classes have to implement all abstract operations of the base class.
 * They can also override some operations with a default implementation.
 */
class ConcreteClass1 : public AbstractClass {
 protected:
```

```
  void RequiredOperations1() const override {
    std::cout << "ConcreteClass1 says: Implemented Operation1\n";
  }
  void RequiredOperation2() const override {
    std::cout << "ConcreteClass1 says: Implemented Operation2\n";
  }
};
/**
 * Usually, concrete classes override only a fraction of base class' operations.
 */
class ConcreteClass2 : public AbstractClass {
 protected:
  void RequiredOperations1() const override {
    std::cout << "ConcreteClass2 says: Implemented Operation1\n";
  }
  void RequiredOperation2() const override {
    std::cout << "ConcreteClass2 says: Implemented Operation2\n";
  }
  void Hook1() const override {
    std::cout << "ConcreteClass2 says: Overridden Hook1\n";
  }
};
/**
 * The client code calls the template method to execute the algorithm. Client
 * code does not have to know the concrete class of an object it works with, as
 * long as it works with objects through the interface of their base class.
 */
void ClientCode(AbstractClass *class_) {
  // ...
  class_->TemplateMethod();
  // ...
}

int main() {
  std::cout << "Same client code can work with different subclasses:\n";
  ConcreteClass1 *concreteClass1 = new ConcreteClass1;
  ClientCode(concreteClass1);
  std::cout << "\n";
  std::cout << "Same client code can work with different subclasses:\n";
  ConcreteClass2 *concreteClass2 = new ConcreteClass2;
  ClientCode(concreteClass2);
  delete concreteClass1;
  delete concreteClass2;
  return 0;
}
```

Output.txt: Execution result

```
Same client code can work with different subclasses:
AbstractClass says: I am doing the bulk of the work
ConcreteClass1 says: Implemented Operation1
AbstractClass says: But I let subclasses override some operations
ConcreteClass1 says: Implemented Operation2
AbstractClass says: But I am doing the bulk of the work anyway

Same client code can work with different subclasses:
AbstractClass says: I am doing the bulk of the work
ConcreteClass2 says: Implemented Operation1
AbstractClass says: But I let subclasses override some operations
ConcreteClass2 says: Overridden Hook1
ConcreteClass2 says: Implemented Operation2
AbstractClass says: But I am doing the bulk of the work anyway
```