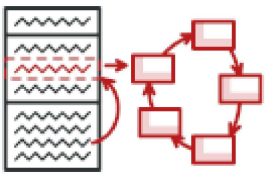

State in C++

State is a behavioral design pattern that allows an object to change the behavior when its internal state changes.

The pattern extracts state-related behaviors into separate state classes and forces the original object to delegate the work to an instance of these classes, instead of acting on its own.

Learn more about State

Complexity:

Popularity:

Usage examples: The State pattern is commonly used in C++ to convert massive `switch`-base state machines into objects.

Identification: State pattern can be recognized by methods that change their behavior depending on the objects' state, controlled externally.

Conceptual Example

This example illustrates the structure of the **State** 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

```
#include <iostream>
#include <typeinfo>
/**
 * The base State class declares methods that all Concrete State should
 * implement and also provides a backreference to the Context object, associated
 * with the State. This backreference can be used by States to transition the
```

```
 * Context to another State.
 */

class Context;

class State {
  /**
   * @var Context
   */
 protected:
  Context *context_;

 public:
  virtual ~State() {
  }

  void set_context(Context *context) {
    this->context_ = context;
  }

  virtual void Handle1() = 0;
  virtual void Handle2() = 0;
};

/**
 * The Context defines the interface of interest to clients. It also maintains a
 * reference to an instance of a State subclass, which represents the current
 * state of the Context.
 */
class Context {
  /**
   * @var State A reference to the current state of the Context.
   */
 private:
  State *state_;

 public:
  Context(State *state) : state_(nullptr) {
    this->TransitionTo(state);
  }
  ~Context() {
    delete state_;
  }
  /**
   * The Context allows changing the State object at runtime.
   */
  void TransitionTo(State *state) {
    std::cout << "Context: Transition to " << typeid(*state).name() << ".\n";
    if (this->state_ != nullptr)
      delete this->state_;
    this->state_ = state;
    this->state_->set_context(this);
```

```
  }
  /**
   * The Context delegates part of its behavior to the current State object.
   */
  void Request1() {
    this->state_->Handle1();
  }
  void Request2() {
    this->state_->Handle2();
  }
};

/**
 * Concrete States implement various behaviors, associated with a state of the
 * Context.
 */

class ConcreteStateA : public State {
 public:
  void Handle1() override;

  void Handle2() override {
    std::cout << "ConcreteStateA handles request2.\n";
  }
};

class ConcreteStateB : public State {
 public:
  void Handle1() override {
    std::cout << "ConcreteStateB handles request1.\n";
  }
  void Handle2() override {
    std::cout << "ConcreteStateB handles request2.\n";
    std::cout << "ConcreteStateB wants to change the state of the context.\n";
    this->context_->TransitionTo(new ConcreteStateA);
  }
};

void ConcreteStateA::Handle1() {
  {
    std::cout << "ConcreteStateA handles request1.\n";
    std::cout << "ConcreteStateA wants to change the state of the context.\n";

    this->context_->TransitionTo(new ConcreteStateB);
  }
}

/**
 * The client code.
 */
void ClientCode() {
  Context *context = new Context(new ConcreteStateA);
```

```
  context->Request1();
  context->Request2();
  delete context;
}

int main() {
  ClientCode();
  return 0;
}
```

Output.txt: Execution result

```
Context: Transition to 14ConcreteStateA.
ConcreteStateA handles request1.
ConcreteStateA wants to change the state of the context.
Context: Transition to 14ConcreteStateB.
ConcreteStateB handles request2.
ConcreteStateB wants to change the state of the context.
Context: Transition to 14ConcreteStateA.
```