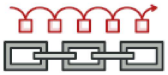

Chain of Responsibility in C++

Chain of Responsibility is behavioral design pattern that allows passing request along the chain of potential handlers until one of them handles request.

The pattern allows multiple objects to handle the request without coupling sender class to the concrete classes of the receivers. The chain can be composed dynamically at runtime with any handler that follows a standard handler interface.

Learn more about Chain of Responsibility

Complexity:

Popularity:

Usage examples: The Chain of Responsibility is pretty common in C++. It's mostly relevant when your code operates with chains of objects, such as filters, event chains, etc.

Identification: The pattern is recognizable by behavioral methods of one group of objects that indirectly call the same methods in other objects, while all the objects follow the common interface.

Conceptual Example

This example illustrates the structure of the **Chain of Responsibility** 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 Handler interface declares a method for building the chain of handlers.
 * It also declares a method for executing a request.
 */
class Handler {
 public:
  virtual Handler *SetNext(Handler *handler) = 0;
  virtual std::string Handle(std::string request) = 0;
};
/**
 * The default chaining behavior can be implemented inside a base handler class.
 */
class AbstractHandler : public Handler {
  /**
   * @var Handler
   */
 private:
  Handler *next_handler_;

 public:
  AbstractHandler() : next_handler_(nullptr) {
  }
  Handler *SetNext(Handler *handler) override {
    this->next_handler_ = handler;
    // Returning a handler from here will let us link handlers in a convenient
    // way like this:
    // $monkey->setNext($squirrel)->setNext($dog);
    return handler;
  }
  std::string Handle(std::string request) override {
    if (this->next_handler_) {
      return this->next_handler_->Handle(request);
    }

    return {};
  }
};
/**
 * All Concrete Handlers either handle a request or pass it to the next handler
 * in the chain.
 */
class MonkeyHandler : public AbstractHandler {
 public:
  std::string Handle(std::string request) override {
    if (request == "Banana") {
      return "Monkey: I'll eat the " + request + ".\n";
    } else {
      return AbstractHandler::Handle(request);
    }
```

```
  }
};
class SquirrelHandler : public AbstractHandler {
 public:
  std::string Handle(std::string request) override {
    if (request == "Nut") {
      return "Squirrel: I'll eat the " + request + ".\n";
    } else {
      return AbstractHandler::Handle(request);
    }
  }
};
class DogHandler : public AbstractHandler {
 public:
  std::string Handle(std::string request) override {
    if (request == "MeatBall") {
      return "Dog: I'll eat the " + request + ".\n";
    } else {
      return AbstractHandler::Handle(request);
    }
  }
};
/**
 * The client code is usually suited to work with a single handler. In most
 * cases, it is not even aware that the handler is part of a chain.
 */
void ClientCode(Handler &handler) {
  std::vector<std::string> food = {"Nut", "Banana", "Cup of coffee"};
  for (const std::string &f : food) {
    std::cout << "Client: Who wants a " << f << "?\n";
    const std::string result = handler.Handle(f);
    if (!result.empty()) {
      std::cout << "  " << result;
    } else {
      std::cout << "  " << f << " was left untouched.\n";
    }
  }
}
/**
 * The other part of the client code constructs the actual chain.
 */
int main() {
  MonkeyHandler *monkey = new MonkeyHandler;
  SquirrelHandler *squirrel = new SquirrelHandler;
  DogHandler *dog = new DogHandler;
  monkey->SetNext(squirrel)->SetNext(dog);

  /**
   * The client should be able to send a request to any handler, not just the
   * first one in the chain.
   */
  std::cout << "Chain: Monkey > Squirrel > Dog\n\n";
```

```
  ClientCode(*monkey);
  std::cout << "\n";
  std::cout << "Subchain: Squirrel > Dog\n\n";
  ClientCode(*squirrel);

  delete monkey;
  delete squirrel;
  delete dog;

  return 0;
}
```

Output.txt: Execution result

```
Chain: Monkey > Squirrel > Dog

Client: Who wants a Nut?
  Squirrel: I'll eat the Nut.
Client: Who wants a Banana?
  Monkey: I'll eat the Banana.
Client: Who wants a Cup of coffee?
  Cup of coffee was left untouched.

Subchain: Squirrel > Dog

Client: Who wants a Nut?
  Squirrel: I'll eat the Nut.
Client: Who wants a Banana?
  Banana was left untouched.
Client: Who wants a Cup of coffee?
  Cup of coffee was left untouched.
```