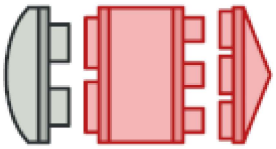

Adapter in C++

Adapter is a structural design pattern, which allows incompatible objects to collaborate.

The Adapter acts as a wrapper between two objects. It catches calls for one object and transforms them to format and interface recognizable by the second object.

Learn more about Adapter

Complexity:

Popularity:

Usage examples: The Adapter pattern is pretty common in C++ code. It's very often used in systems based on some legacy code. In such cases, Adapters make legacy code work with modern classes.

Identification: Adapter is recognizable by a constructor which takes an instance of a different abstract/interface type. When the adapter receives a call to any of its methods, it translates parameters to the appropriate format and then directs the call to one or several methods of the wrapped object.

Conceptual Example | Multiple Inheritance

Conceptual Example

This example illustrates the structure of the **Adapter** 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 Target defines the domain-specific interface used by the client code.
 */
class Target {
 public:
  virtual ~Target() = default;

  virtual std::string Request() const {
    return "Target: The default target's behavior.";
  }
};

/**
 * The Adaptee contains some useful behavior, but its interface is incompatible
 * with the existing client code. The Adaptee needs some adaptation before the
 * client code can use it.
 */
class Adaptee {
 public:
  std::string SpecificRequest() const {
    return ".eetpadA eht fo roivaheb laicepS";
  }
};

/**
 * The Adapter makes the Adaptee's interface compatible with the Target's
 * interface.
 */
class Adapter : public Target {
 private:
  Adaptee *adaptee_;

 public:
  Adapter(Adaptee *adaptee) : adaptee_(adaptee) {}
  std::string Request() const override {
    std::string to_reverse = this->adaptee_->SpecificRequest();
    std::reverse(to_reverse.begin(), to_reverse.end());
    return "Adapter: (TRANSLATED) " + to_reverse;
  }
};

/**
 * The client code supports all classes that follow the Target interface.
 */
void ClientCode(const Target *target) {
  std::cout << target->Request();
}

int main() {
  std::cout << "Client: I can work just fine with the Target objects:\n";
  Target *target = new Target;
```

```
  ClientCode(target);
  std::cout << "\n\n";
  Adaptee *adaptee = new Adaptee;
  std::cout << "Client: The Adaptee class has a weird interface. See, I don't unders
  std::cout << "Adaptee: " << adaptee->SpecificRequest();
  std::cout << "\n\n";
  std::cout << "Client: But I can work with it via the Adapter:\n";
  Adapter *adapter = new Adapter(adaptee);
  ClientCode(adapter);
  std::cout << "\n";

  delete target;
  delete adaptee;
  delete adapter;

  return 0;
}
```

Output.txt: Execution result

```
Client: I can work just fine with the Target objects:
Target: The default target's behavior.

Client: The Adaptee class has a weird interface. See, I don't understand it:
Adaptee: .eetpadA eht fo roivaheb laicepS

Client: But I can work with it via the Adapter:
Adapter: (TRANSLATED) Special behavior of the Adaptee.
```