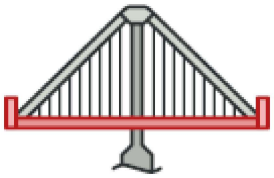

Bridge in C++

Bridge is a structural design pattern that divides business logic or huge class into separate class hierarchies that can be developed independently.

One of these hierarchies (often called the Abstraction) will get a reference to an object of the second hierarchy (Implementation). The abstraction will be able to delegate some (sometimes, most) of its calls to the implementations object. Since all implementations will have a common interface, they'd be interchangeable inside the abstraction.

Learn more about Bridge

Complexity:

Popularity:

Usage examples: The Bridge pattern is especially useful when dealing with cross-platform apps, supporting multiple types of database servers or working with several API providers of a certain kind (for example, cloud platforms, social networks, etc.)

Identification: Bridge can be recognized by a clear distinction between some controlling entity and several different platforms that it relies on.

Conceptual Example

This example illustrates the structure of the **Bridge** 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 Implementation defines the interface for all implementation classes. It
```

```
 * doesn't have to match the Abstraction's interface. In fact, the two
 * interfaces can be entirely different. Typically the Implementation interface
 * provides only primitive operations, while the Abstraction defines higher-
 * level operations based on those primitives.
 */

class Implementation {
 public:
  virtual ~Implementation() {}
  virtual std::string OperationImplementation() const = 0;
};

/**
 * Each Concrete Implementation corresponds to a specific platform and
 * implements the Implementation interface using that platform's API.
 */
class ConcreteImplementationA : public Implementation {
 public:
  std::string OperationImplementation() const override {
    return "ConcreteImplementationA: Here's the result on the platform A.\n";
  }
};
class ConcreteImplementationB : public Implementation {
 public:
  std::string OperationImplementation() const override {
    return "ConcreteImplementationB: Here's the result on the platform B.\n";
  }
};

/**
 * The Abstraction defines the interface for the "control" part of the two class
 * hierarchies. It maintains a reference to an object of the Implementation
 * hierarchy and delegates all of the real work to this object.
 */

class Abstraction {
  /**
   * @var Implementation
   */
 protected:
  Implementation* implementation_;

 public:
  Abstraction(Implementation* implementation) : implementation_(implementation) {
  }

  virtual ~Abstraction() {
  }

  virtual std::string Operation() const {
    return "Abstraction: Base operation with:\n" +
           this->implementation_->OperationImplementation();
```

```
  }
};
/**
 * You can extend the Abstraction without changing the Implementation classes.
 */
class ExtendedAbstraction : public Abstraction {
 public:
  ExtendedAbstraction(Implementation* implementation) : Abstraction(implementation) {
  }
  std::string Operation() const override {
    return "ExtendedAbstraction: Extended operation with:\n" +
           this->implementation_->OperationImplementation();
  }
};

/**
 * Except for the initialization phase, where an Abstraction object gets linked
 * with a specific Implementation object, the client code should only depend on
 * the Abstraction class. This way the client code can support any abstraction-
 * implementation combination.
 */
void ClientCode(const Abstraction& abstraction) {
  // ...
  std::cout << abstraction.Operation();
  // ...
}
/**
 * The client code should be able to work with any pre-configured abstraction-
 * implementation combination.
 */

int main() {
  Implementation* implementation = new ConcreteImplementationA;
  Abstraction* abstraction = new Abstraction(implementation);
  ClientCode(*abstraction);
  std::cout << std::endl;
  delete implementation;
  delete abstraction;

  implementation = new ConcreteImplementationB;
  abstraction = new ExtendedAbstraction(implementation);
  ClientCode(*abstraction);

  delete implementation;
  delete abstraction;

  return 0;
}
```

Output.txt: Execution result

```
Abstraction: Base operation with:
ConcreteImplementationA: Here's the result on the platform A.

ExtendedAbstraction: Extended operation with:
ConcreteImplementationB: Here's the result on the platform B.
```