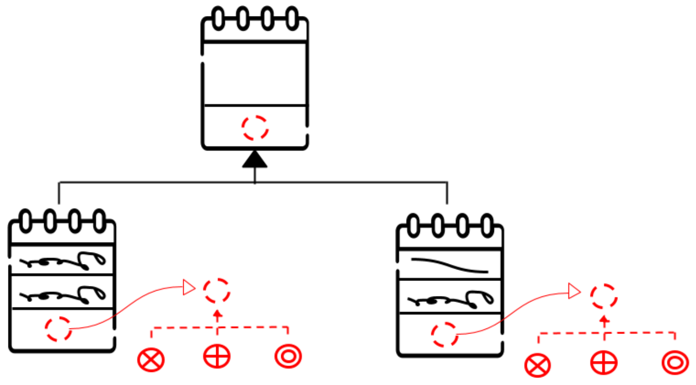
Bridge Design Pattern in Modern C++



Reading Time: 6 minutes

Bridge Design Pattern is a Structural Design Pattern used ***to decouple a class into two parts – abstraction and it's implementation*** – so that both can be developed independently. This promotes the loose coupling between class abstraction & its implementation. You get this decoupling by adding one more level of indirection i.e. an interface which acts as a bridge between your original class & functionality. Insulation is another name of Bridge Design Pattern in C++ world.

"All problems in computer science can be solved by another level of indirection."
– David Wheeler

By the way, If you haven't check out my other articles on Structural Design Patterns, then here is the list:

The code snippets you see throughout this series of articles are simplified not sophisticated. So you often see me not using keywords like `override`, `final`, `public`(while inheritance) just to make code compact & consumable(most of the time) in single standard screen size. I also prefer `struct` instead of `class` just to save line by not writing "`public:`" sometimes and also miss virtual destructor, constructor, copy constructor, prefix `std::`, deleting dynamic memory, intentionally. I also consider myself a pragmatic person who wants to convey an idea in the simplest way possible rather than the standard way or using Jargons.

Note:

- If you stumbled here directly, then I would suggest you go through What is design pattern? first, even if it is trivial. I

believe it will encourage you to explore more on this topic.
- All of this code you encounter in this series of articles are compiled using C++20(though I have used Modern C++ features up to C++17 in most cases). So if you don't have access to the latest compiler you can use https://wandbox.org/ which has preinstalled boost library as well.

Contents [hide]

Intent

To separate the interface from its implementation.

- In other words, It's all about connecting components together through flexible abstractions using aggregation/composition rather than inheritance/generalization.
- This pattern involves an interface which acts as a bridge. That makes the functionality of concrete classes independent from interface implementer classes. Both types of classes can alter structurally without affecting each other.

Motivation for Bridge Design Pattern

- Bridge Design Pattern prevents ***Cartesian Product*** complexity explosion. Don't be scared with this mathematical term, I have simplified it with an example below.
- So, for example, let's suppose that you have some base class called `Shape` and then the `Shape` can be `Circle` or `Square` and it can also be drawn by API 1 or API 2.

```
struct DrawingAPI_1 { };
struct DrawingAPI_2 { };

struct Shape { virtual void draw() = 0; };

/* 2 x 2 scenario */
struct Circle : Shape, DrawingAPI_1 { };
struct Circle : Shape, DrawingAPI_2 { };

struct Square : Shape, DrawingAPI_1 { };
struct Square : Shape, DrawingAPI_2 { };
```

- This way you end up having a **two by two(2×2) scenario**. So if you decide to implement it you have to implement four classes. One for you know the `Circle` with `API_1`, `Circle` with `API_2` and so on.

- The Bridge Design Pattern is precisely the pattern that actually avoids this whole entity explosion.
- So instead of having something like above, what we can do is we design the DrawingAPI interface(which later on used to derive API 1 & 2) and aggregate it in `Circle` & `Square`.

Bridge Design Pattern C++ Example

- So following is the typical implementation of the Bridge Design Pattern. We are not going to look at anything quite so complicated here. But essentially a ***bridge is a mechanism that decouples the interface or the hierarchy from the implementation.***

```
struct DrawingAPI {
    virtual void drawCircle() = 0;
};

struct DrawingAPI_1 : DrawingAPI {
    void drawCircle() { cout << "Drawn by API 1"<< endl; }
};

struct DrawingAPI_2 : DrawingAPI {
    void drawCircle() { cout << "Drawn by API 2"<< endl; }
};

struct Shape {
    Shape(DrawingAPI &drawingAPI) : m_drawingAPI{drawingAPI} {}
    virtual void draw() = 0;
protected:
    DrawingAPI &m_drawingAPI;   // Now Shapes does not need to worry about drawing APIs
};

struct Circle : Shape {
    Circle(DrawingAPI &drawingAPI) : Shape{drawingAPI} {}
    void draw() { m_drawingAPI.drawCircle(); }
};

int main() {
    DrawingAPI_1 API_1;
    DrawingAPI_2 API_2;
    Circle(API_1).draw();
    Circle(API_2).draw();
    return EXIT_SUCCESS;
}
```

- This way you don’t rely as much as on inheritance and aggregation. Rather you rely on the interface.

Bridge Design Pattern using C++ Idiom: Pointer to Implementation(PIMPL)

- How can we forget the PIMPLE idiom while we are discussing the Bridge Design Pattern! PIMPLE is the manifestation of the bridge design pattern albeit a slightly different one.
- PIMPL idiom is all about hiding the implementation details of a particular class by sticking it into separate implementation pointed by pointer just as the name suggests. Let me show you how this works:

Person.h

```
#pragma once
#include <string>
#include <memory>

struct Person {
    /* PIMPL ------------------------------------ */
    class PersonImpl;
    unique_ptr<PersonImpl>  m_impl; // bridge - not necessarily inner class, can vary
    /* ------------------------------------------ */
    string                  m_name;

    Person();
    ~Person();

    void greet();
private:
    // secret data members or methods are in `PersonImpl` not here
    // as we are going to expose this class to client
};
```

Person.cpp <- will be turned into shared library(.so/.dll), to hide the business logic

```
#include "Person.h"

/* PIMPL Implementation ------------------------------------ */
struct Person::PersonImpl {
    void greet(Person *p) {
        cout << "hello "<< p->name.c_str() << endl;
    }
};
/* --------------------------------------------------------- */

Person::Person() : m_impl(new PersonImpl) { }
Person::~Person() { delete m_impl; }
void Person::greet() { m_impl->greet(this); }
```

- OK, so this is the pimple idiom in it’s kind of concise form shall we say. And the question is Well why would you want to do this in the first place.

- Security purpose, you might have a question that any way we are going to expose header file to the client which contains API of a class, then how do we get security here? Well, just think about the data members & private methods. If you have trade secrets & having a data member which contains critical information. Why do you even let the client know the name of the object?
- One more benefit of PIMPL is compilation time which is critical for C++ as it widely criticized for it. But this is becoming less and less relevant as the compilers become more and more incremental. And they are really fantastic nowadays.

Secure & Faster PIMPL

- As we have to use all the API using indirection provided by `unique_ptr` which accounts for some run-time overhead as we have to dereference the pointer every time for access.
- Plus we also have construction & destruction overhead of `unique_ptr`because it creates a memory in a heap which involves many other functions calling along with system calls.
- Moreover, we also have to baer some indirection if we want to access the data member of `Person` in `PersonImpl` like passing `this` pointer or so.

Person.h
```
#pragma once
#include <string>
#include <cstddef>
#include <type_traits>

struct Person {
    Person();
    ~Person();
    void greet();

private:
    static constexpr size_t     m_size = 1024;
    using pimpl_storage_t = aligned_storage<m_size, alignment_of_v<max_align_t>>::type;

    string                      m_name;
    pimpl_storage_t             m_impl;
};
```

Person.cpp <- will be turned into shared library(.so/.dll), to hide the business logic
```
#include "Person.h"
#include <iostream>

struct PersonImpl {
    void greet(string &name) {
        cout << "hello "<< name << endl;
    }
};

Person::Person() {
  static_assert(sizeof(impl) >= sizeof(PersonImpl)); // Compile time safety
  new(&impl) PersonImpl;
}
Person::~Person() { reinterpret_cast<PersonImpl*>(&impl)->~PersonImpl(); }
void Person::greet() { reinterpret_cast<PersonImpl*>(&impl)->greet(name);  }
```

- So let’s address this issue with placement new operator & preallocated aligned memory buffer. `reinterpret_cast` is just compile-time substitute so there won’t be any other indirection.

Benefits of Bridge Design Pattern

1. Bridge Design Pattern provides flexibility to develop abstraction(i.e. interface) and the implementation independently. And the client/API-user code can access only the abstraction part without being concerned about the Implementation part.
2. It preserves the Open-Closed Principle, in other words, improves extensibility as client/API-user code relies on abstraction only so implementation can modify or augmented any time.
3. By using the Bridge Design Pattern in the form of PIMPL. We can hide the implementation details from the client as we did in PIMPL idiom example above.
4. The Bridge Design Pattern is an application of the old advice, “prefer composition over inheritance” but in a smarter way. It comes handy when you must subclass different times in ways that are orthogonal with one another(say 2×2 problem discuss earlier).
5. A compile-time binding between an abstraction and its implementation should be avoided. So that an implementation can select at run-time.

Summary by FAQs

What is the practical use case of Bridge Design Pattern?

Plugins in any internet browser leverage this pattern directly where browser specifies only abstraction & implementation varies by different types of plugins.

When to use Bridge Design Pattern?

- When you are unsure of implementation or its variations & still you want to move forward with development.
- In case of a behaviour permutation problem i.e. Cartesian Product Complexity Explosion.

What are the differences between Adapter & Bridge Design Pattern?

- Adapter is commonly used with an existing app to make some otherwise-incompatible classes work together nicely.
- Bridge is usually designed up-front, letting you develop parts of an application independently of each other.

What are the differences between Strategy & Bridge Design Pattern?

- Strategy is a single dimension problem like Multi-bit screwdriver.
- Bridge is a multi-dimension problem like communication types & devices.

Strategy
Bridge
Multi-bit screwdriver
Interchangeable bits
Strategy = bits
Communication Types: email,SMS,voice
Communication Devices: PC,tablet, phone
Bridge = Interface(having email,SMS, voice as pure virtual methods) & Implementation(PC,tablet,phone)

Do you like it☝️? Get such articles directly into the inbox…!?