Table of contents

MVC Dependency Injection

Introduction

This project demonstrates the implementation of the **Model-View-Controller (MVC)** pattern in C++ using three different types of **Dependency Injection (DI)** strategies:

1. **Manual Dependency Injection** - Dependencies are manually injected into the controller class.
2. **Custom Dependency Injection Framework** - A lightweight custom-built DI container is used to manage and inject dependencies.
3. **Boost.DI Framework** - An industry-standard, header-only library, `Boost.DI`, is utilized to automatically handle dependency injection.

The project contains the following files:

- **MVC.h / MVC.cpp**: These files implement the basic MVC architecture.
- **DI-manually.cpp**: This file demonstrates how manual dependency injection is implemented.
- **DI-CustomFramework.cpp**: This file showcases how a custom-built DI framework is used to manage dependencies.
- **DI-Framework_Boost.cpp**: This file integrates the `Boost.DI` library for automatic dependency injection.
- **BoostDI.hpp**: This file includes the necessary setup and configurations for `Boost.DI` (already provided, doesn't need explanation).

Each section of this documentation will describe how each dependency injection strategy is implemented, along with detailed explanations of the corresponding code. This will help you understand the pros and cons of each approach and how DI enhances modularity, scalability, and maintainability in MVC applications.

MVC Implementation

MVC Implementation

This section outlines the implementation of the **Model-View-Controller (MVC)** architecture, which is the foundation of the project. The MVC design pattern separates concerns into three core components: **Model**, **View**, and **Controller**, each with its own responsibility. Additionally, we utilize a **Logger** and **Configuration** for logging events and managing app configuration, respectively.

The implementation is contained in two files: `MVC.h` (header file) and `MVC.cpp` (implementation file).

1. MVC.h

The `MVC.h` file defines the classes and their public interfaces for the core components of the MVC architecture: **Model**, **View**, **Controller**, and additional helper classes like **Logger** and **Configuration**.

Logger Declaration

The `Logger` class provides basic logging functionality for the application. It is responsible for printing log messages to the console.

```
class Logger {
public:
    void log(const std::string& message);
};
```

- `log(const std::string& message)`: Logs the given message to the console.

Configuration Declaration

The `Configuration` class stores application-level configuration data. In this example, it holds the application name (`app_name`).

```
class Configuration {
public:
```

```
    Configuration() : app_name("HelloApp") {}
    std::string app_name;
};
```

- `app_name`: Stores the name of the application, initialized to `"HelloApp"`.

Model Declaration

The `Model` class holds the data for the application. In this case, it stores the user's name.

```
class Model {
public:
    void setName(const std::string& name);
    std::string getName() const;

private:
    std::string name_;
};
```

- `setName(const std::string& name)`: Sets the user's name in the `name_` variable.
- `getName() const`: Retrieves the user's name from the `name_` variable.
- `name_`: A private member variable that stores the user's name.

View Declaration

The `View` class interacts with the user. It prompts for user input and displays messages.

```
class View {
public:
    std::string askForName();
    void displayGreeting(const std::string& name);
};
```

- `askForName()`: Prompts the user for their name and returns the input.

- `displayGreeting(const std::string& name)`: Displays a greeting message using the provided name.

Controller Declaration

The `Controller` class ties together the **Model**, **View**, and **Logger** components. It handles the application's workflow.

```
class Controller {
public:
    // Constructor that uses shared_ptr for dependencies
    Controller(std::shared_ptr<Model> model, std::shared_ptr<View> view,
std::shared_ptr<Logger> logger)
        : model_(model), view_(view), logger_(logger) {}

    void run() const;

private:
    std::shared_ptr<Model> model_;
    std::shared_ptr<View> view_;
    std::shared_ptr<Logger> logger_;
};
```

- **Constructor**: Takes in `std::shared_ptr` to **Model**, **View**, and **Logger**, ensuring dependency injection is possible and memory is managed safely.
- `run() const`: The core function that runs the application, which interacts with the model, view, and logger.

2. MVC.cpp

The `MVC.cpp` file contains the implementation of the methods defined in `MVC.h`.

Logger Definition

```
void Logger::log(const std::string& message) {
    std::cout << "[LOG]: " << message << "\n";
}
```

- `log()`: Logs the given message to the console, prefixed with `[LOG]:`.

Model Definition

```
void Model::setName(const std::string& name) {
    name_ = name;
}

std::string Model::getName() const {
    return name_;
}
```

- `setName()`: Stores the name passed as an argument in the `name_` member variable.
- `getName()`: Returns the stored name.

View Definition

```
std::string View::askForName() {
    std::string name;
    std::cout << "Enter your name: ";
    std::getline(std::cin, name);
    return name;
}

void View::displayGreeting(const std::string& name) {
    std::cout << "Hello " << name << "!" << "\n";
}
```

- `askForName()`: Prompts the user for their name and captures it using `std::getline()`.
- `displayGreeting()`: Displays a greeting message including the user's name.

Controller Definition

```
void Controller::run() const {
    logger_->log("Starting application...");
    std::string name = view_->askForName();
```

```
    model_->setName(name);
    view_->displayGreeting(model_->getName());
    logger_->log("Application finished.");
}
```

- `run() const`:
  - Starts by logging a message that the application is starting.
  - Interacts with the **View** to get the user's name, then stores it in the **Model**.
  - Displays the greeting using **View** and logs when the application finishes.

Summary of the MVC Components:

- **Model**: Stores and manages application data (user's name).
- **View**: Interacts with the user, handling input/output (asking for the name and displaying the greeting).
- **Controller**: Glues the **Model** and **View** together, handling the application's logic and flow.
- **Logger**: Used for logging significant events in the application's lifecycle.
- **Configuration**: Holds configuration settings (e.g., app name) for the application.

Manual Dependency Injection

Manual Dependency Injection

In the file `DI-Manually.cpp`, the **Manual Dependency Injection** approach is demonstrated. In this method, the dependencies of the `Controller` (i.e., `Model`, `View`, and `Logger`) are created and injected manually in the `main()` function.

This method provides a simple, explicit way to manage dependencies, but it can become difficult to scale as the number of dependencies grows. The programmer is responsible for manually managing the creation and wiring of dependencies.

1. Main Entry Point (DI-Manually.cpp)

This file implements the manual dependency injection of the **Model**, **View**, and **Logger** objects into the **Controller**.

Code Breakdown

```
#include "MVC.h"

// Main entry point with manual DI
int main() {
    Configuration config;
    auto logger = std::make_shared<Logger>();
    auto model = std::make_shared<Model>();
    auto view = std::make_shared<View>();
    Controller controller(model, view, logger);

    logger->log("App Name: " + config.app_name + " - Manual DI");
    controller.run();
    return 0;
}
```

Key Steps in Manual Dependency Injection:

1. **Configuration Object**:

- A `Configuration` object is instantiated to hold application settings (like the application name).

```
Configuration config;
```

2. **Dependency Creation**:

- The dependencies `Logger`, `Model`, and `View` are instantiated using `std::make_shared`, creating smart pointers to these objects.

```
auto logger = std::make_shared<Logger>();
auto model = std::make_shared<Model>();
auto view = std::make_shared<View>();
```

3. **Controller Initialization**:

- The `Controller` object is created with its dependencies manually passed via its constructor. The `Model`, `View`, and `Logger` objects are injected into the `Controller` at the time of creation.

```
Controller controller(model, view, logger);
```

4. **Logging and Application Execution**:

- The `Logger` logs the application name along with a note that manual dependency injection is being used.

```
logger->log("App Name: " + config.app_name + " - Manual DI");
```

- The `Controller::run()` function is called to execute the application logic, which handles user input and displays a greeting message.

```
controller.run();
```

Explanation of Manual DI

Manual dependency injection is a straightforward approach that allows complete control over object instantiation and dependency management. This means you explicitly create and pass dependencies, ensuring tight coupling between objects.

While this method is simple, it becomes unwieldy in larger applications where multiple components have many dependencies. Scaling can lead to bloated code and the risk of overlooking the proper instantiation of certain components.

Advantages of Manual DI:

- **Simplicity**: It is easy to understand and requires no additional frameworks or tools.
- **Control**: Full control over how and when dependencies are instantiated.

Disadvantages of Manual DI:

- **Scalability**: As the number of dependencies grows, managing them manually becomes error-prone and complex.
- **Boilerplate Code**: Repeated code is required to manually inject dependencies into multiple parts of the application.

Summary:

In the **manual DI** approach:

- Dependencies are explicitly created and injected into the `Controller` class during initialization.
- This solution is adequate for small-scale applications but lacks scalability as the number of dependencies increases.

Boost Dependency Injection Framework

Dependency Injection with Boost.DI

In the file `DI-boostFramework.cpp`, the **Boost.DI** framework is utilized to handle the dependency injection automatically. **Boost.DI** is a powerful, header-only C++ library that simplifies and automates dependency management.

Instead of manually creating and injecting dependencies, **Boost.DI** manages this process for you by resolving the dependencies at runtime, based on the types required by each class. This method is more scalable and maintainable than manual dependency injection, especially for larger applications with complex dependency graphs.

1. Main Entry Point (DI-Framework_Boost.cpp)

This file demonstrates the use of the **Boost.DI** framework to automatically inject dependencies into the **Controller**.

Code Breakdown

```
#include "MVC.h"
#include "BoostDI.hpp"

// DI Main with Boost DI framework
int main() {
    auto injector = boost::di::make_injector();

    auto config = injector.create<Configuration>();
    auto logger = injector.create<Logger>();
    auto model = injector.create<Model>();
    auto view = injector.create<View>();
    auto controller = injector.create<Controller>();

    logger.log("App Name: " + config.app_name + " - BoostDI");
    controller.run();
```

```
    return 0;
}
```

Key Steps in Boost.DI Dependency Injection:

1. **Boost.DI Injector Creation**:

- The **Boost.DI** injector is created using the `boost::di::make_injector()` function. This injector is responsible for resolving and injecting dependencies automatically.

```
auto injector = boost::di::make_injector();
```

2. **Dependency Resolution**:

- With the `injector` in place, the dependencies `Configuration`, `Logger`, `Model`, and `View` are automatically created using the injector's `create` function. **Boost.DI** ensures that the correct constructors are called with the appropriate dependencies.

```
auto config = injector.create<Configuration>();
auto logger = injector.create<Logger>();
auto model = injector.create<Model>();
auto view = injector.create<View>();
auto controller = injector.create<Controller>();
```

3. **Logging and Application Execution**:

- The `Logger` logs the application name along with a note that **Boost.DI** is being used.

```
logger.log("App Name: " + config.app_name + " - BoostDI");
```

- The `Controller::run()` function is called to execute the application logic, which interacts with the user to ask for their name and then displays a greeting message.

```
controller.run();
```

Explanation of Boost.DI

Boost.DI automates the process of injecting dependencies. It uses **type-based resolution** to determine what dependencies are required by each class and creates the appropriate instances. This approach eliminates the need for manually managing dependencies, which leads to cleaner and more scalable code.

Advantages of Boost.DI:

- **Automation**: Dependencies are automatically resolved, reducing manual code and potential human error.
- **Scalability**: Easily scales to larger applications where many components have complex dependencies.
- **Decoupling**: Components are loosely coupled, meaning that classes do not need to be aware of how their dependencies are created.

Disadvantages of Boost.DI:

- **Complexity**: While Boost.DI reduces manual dependency management, it adds complexity in terms of setup and understanding for developers unfamiliar with dependency injection frameworks.
- **Overhead**: For small applications, using a DI framework might feel like over-engineering.

Summary:

The **Boost.DI** approach greatly simplifies dependency injection in larger and more complex applications. By automating the resolution of dependencies, **Boost.DI** reduces boilerplate code and makes the application easier to scale and maintain.

In this example, we see how the **Boost.DI** framework resolves dependencies for the **Controller** class without manual intervention, improving code clarity and maintainability.

This approach is a significant improvement over manual dependency injection, particularly for applications that require many interdependent components. However, for smaller applications, this might add unnecessary complexity.

Custom Dependency Injection Framework

Custom Dependency Injection Framework

In the file `DI-customFramework.cpp`, we implement a simple custom-built **Dependency Injection (DI) Framework** to manage the dependencies of the **Model-View-Controller (MVC)** architecture. This approach sits between manual dependency injection and a full-fledged DI framework like **Boost.DI**. It allows for greater flexibility than manual DI, without the overhead of introducing a large external library.

1. Custom DI Framework (DIContainer)

The core of this file is the `DIContainer` class, which is a minimal DI container used to manage the lifecycle of dependencies and inject them where required. The container supports both simple object instantiation and instantiation of objects with dependencies.

Custom DI Framework Code Breakdown

```
#include "MVC.h"
#include <iostream>
#include <map>
#include <functional>
#include <memory>

// Simple DI container
class DIContainer {
public:
    // Register a type that does not have dependencies
    template<typename T>
    void registerType() {
        creators_[typeid(T).name()] = [this]() -> std::shared_ptr<void>
{
            return std::make_shared<T>();
        };
```

```
    }

    // Register a type that has dependencies (e.g., Controller)
    template<typename T, typename... Args>
    void registerTypeWithDependencies() {
        creators_[typeid(T).name()] = [this]() -> std::shared_ptr<void>
{
            return std::make_shared<T>(resolve<Args>()...);
        };
    }

    // Resolve a type from the container
    template<typename T>
    std::shared_ptr<T> resolve() {
        auto it = creators_.find(typeid(T).name());
        if (it != creators_.end()) {
            return std::static_pointer_cast<T>(it->second());
        }
        throw std::runtime_error("Type not registered in DI
container.");
    }

private:
    // A map of creators that holds lambdas for creating each type
    std::map<std::string, std::function<std::shared_ptr<void>()>>
creators_;
};
```

Key Features of the DIContainer:

1. **registerType()**:

- This method registers types that **do not have dependencies**. It stores a lambda function in a map (`creators_`), which can later be invoked to create a new instance of the type using `std::make_shared`.

```
template<typename T>
void registerType() {
    creators_[typeid(T).name()] = [this]() -> std::shared_ptr<void> {
        return std::make_shared<T>();
    };
}
```

2. **registerTypeWithDependencies()**:

- This method registers types that **do have dependencies**. The lambda function stored in the `creators_` map resolves the required dependencies first and then constructs the type using `std::make_shared`.

```
template<typename T, typename... Args>
void registerTypeWithDependencies() {
    creators_[typeid(T).name()] = [this]() -> std::shared_ptr<void> {
        return std::make_shared<T>(resolve<Args>()...);
    };
}
```

3. **resolve()**:

- This method retrieves the registered lambda function for the requested type and calls it to create and return an instance of the type as a `std::shared_ptr`. If the type is not found in the `creators_` map, an error is thrown.

```
template<typename T>
std::shared_ptr<T> resolve() {
    auto it = creators_.find(typeid(T).name());
    if (it != creators_.end()) {
        return std::static_pointer_cast<T>(it->second());
    }
    throw std::runtime_error("Type not registered in DI container.");
}
```

2. Main Entry Point (DI-customFramework.cpp)

The `main()` function demonstrates how to use the custom DI container to resolve dependencies and inject them into the `Controller`.

** Main Entry Code Breakdown**

```
int main() {
    DIContainer container;

    // Register types without dependencies
    container.registerType<Logger>();
    container.registerType<Model>();
    container.registerType<View>();
    container.registerType<Configuration>();

    // Register Controller with dependencies (Model, View, Logger)
    container.registerTypeWithDependencies<Controller, Model, View,
Logger>();

    // Resolve and use the objects
    auto config = container.resolve<Configuration>();
    auto logger = container.resolve<Logger>();
    auto model = container.resolve<Model>();
    auto view = container.resolve<View>();
    auto controller = container.resolve<Controller>();

    logger->log("App Name: " + config->app_name + " - Custom DI
Framework");
    controller->run();

    return 0;
}
```

Key Steps in Custom Dependency Injection:

1. **Register Types**:

- The `registerType` method is called for types that do not have dependencies (`Logger`, `Model`, `View`, `Configuration`).

- The `registerTypeWithDependencies` method is called for the `Controller`, which has dependencies on `Model`, `View`, and `Logger`.

```
container.registerType<Logger>();
container.registerType<Model>();
container.registerType<View>();
container.registerType<Configuration>();

container.registerTypeWithDependencies<Controller, Model, View,
Logger>();
```

2. **Resolve Dependencies**:

- The `resolve` method is used to instantiate each type. **The DIContainer automatically injects the dependencies** for the `Controller`, ensuring that it receives the correct instances of `Model`, `View`, and `Logger`.

```
auto config = container.resolve<Configuration>();
auto logger = container.resolve<Logger>();
auto model = container.resolve<Model>();
auto view = container.resolve<View>();
auto controller = container.resolve<Controller>();
```

3. **Logging and Application Execution**:

- The `Logger` logs the application name along with a note that the custom DI framework is being used.
- The `Controller::run()` function is called to execute the application logic.

```
logger->log("App Name: " + config->app_name + " - Custom DI
Framework");
controller->run();
```

Explanation of Custom DI Framework

This custom-built DI container allows for a more structured and flexible way of injecting dependencies compared to manual DI. By registering types with or without

dependencies, the DI container dynamically resolves and injects dependencies where necessary.

Advantages of Custom DI Framework:

- **Flexibility**: Dependencies are managed dynamically, allowing for a scalable and maintainable solution.
- **Control**: Developers retain control over the creation and registration of dependencies while avoiding the overhead of manual injection.
- **Simplicity**: This framework is lightweight and easy to understand, making it suitable for small to medium-sized applications.

Disadvantages of Custom DI Framework:

- **Limited Features**: Compared to industry-standard DI frameworks like **Boost.DI**, this custom solution is limited in functionality (e.g., no scope management, complex dependency graphs).
- **Manual Registration**: Developers still need to manually register each type in the container, which can become tedious for large applications.

Summary:

In this solution, a **Custom DI Framework** is used to manage dependencies between the **Model**, **View**, and **Controller**. The framework allows for flexible, dynamic dependency injection while remaining lightweight and easy to understand.

This approach strikes a balance between **manual DI** and fully-fledged frameworks like **Boost.DI**, offering a solution that is more scalable than manual DI but simpler than advanced frameworks.