CppIndia

QICKSTEP

SET YOUR CAREER IN MOTION

Solid Principles

Harishankar Singh
Director & Mentor QICKSTEP

What goes wrong with Software ?

- How many of slowed down because of ***BAD CODE*** ?
- Why do we write code that slow us down ?

You don’t go fast by writing crap, you don’t go fast by rushing, you don’t go fast by making release by doing hacks because you want to go fast.

- You go fast deliberately not rapidly, take time analyse the problem and solve in proper way.

What is ***BAD CODE?***

- BAD code is confusing.
- Good code is self explaining.
- What happens when you modify BAD code ?

Symptoms of BAD CODE?

- First Symptom of BAD code is ***Rigidity***
  Code is rigid that has dependency that is spread across modules. Code that cant handle isolated changes, and dependencies, Coupling.

- Second symptom of BAD is ***Fragility***
  Make changes that are not dependent but break other parts.

- Third Symptom of the BAD code is ***Non Reusability***
  when you want to resume the code and the desirable part of the code is tightly coupled with the undesirable code that you can not use it.

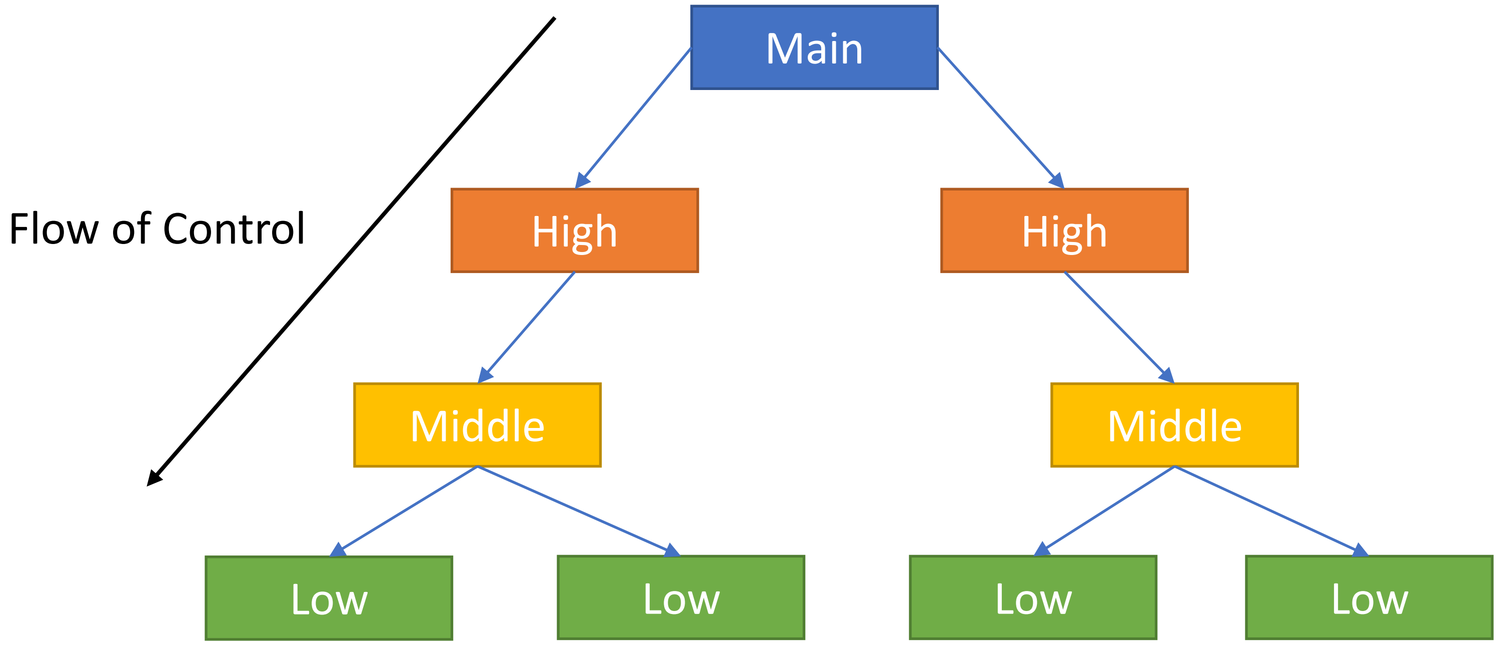
☺ will you go to that same mechanic again ?

That’s the failure of the Programmers. The reason for all problem is ***Coupling***.

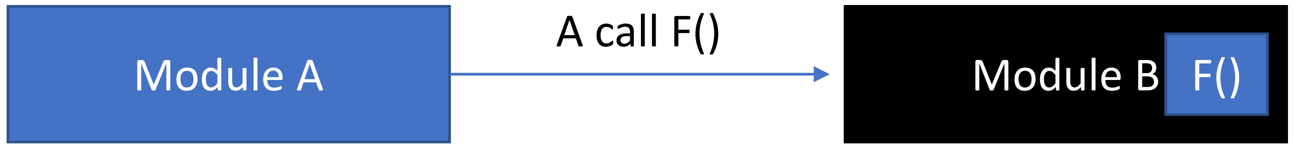
Software design is managing the dependency, Figuring out where to put the code and cutting down the Dependencies.

So how do we do that ?

Procedure function call tree



Procedure function call tree



Source code dependency on N

How do we know A knows about B? if B F() changes A has to change?
Compiler knows ? Because in A source code you have include B.h

In the procedure function call tree how that Source Code dependency flows? i.e. from High to low.
Means High level module knows about the low level modules. *Think about it ? What rule its violated ? Do you Want your high level policy violated because of low level modules ? So if we change low level who compiles ? This follows from low to high. It need to recompile and redeploy again.*

What is OOP?

What are objects? Why so much fuzz now a days about OOP?

Three Magic word

- ***Encapsulation***
- ***Inheritance***
- ***Polymorphism***

What is OOP?

- Do we have Encapsulation in C?
- What about Inheritance in C? can be achieved using structures pointers and in last have common. But it was hard. C++ Made is littlie convenient.
- Why java don't have multiple inheritance ?
- Do we have Polymorphism in C?
- Polymorphism allowed us to deal with Module A & Module B. Polymorphism give you the ability to go against the control flow
- If you have absolute control in the dependency structure you can write, Non rigid, Non Fragile and non reusable module. You can get around What goes wrong with software. Carefully deciding which direction errors between module should point. That's what OO is.

OO is about managing dependencies by selecting and inverting the key dependencies in your architecture so that you can prevent rigidity, fragility And non reusability.

SOLID

In object-oriented computer programming, SOLID is an acronym for five design principles intended to make software designs more **Understandable**, **Flexible** and **Maintainable**. The principles are a subset of many principles promoted by Robert C. Martin. Though they apply to any object-oriented design, the SOLID principles can also form a core philosophy for methodologies such as agile development.

https://en.wikipedia.org/wiki/SOLID

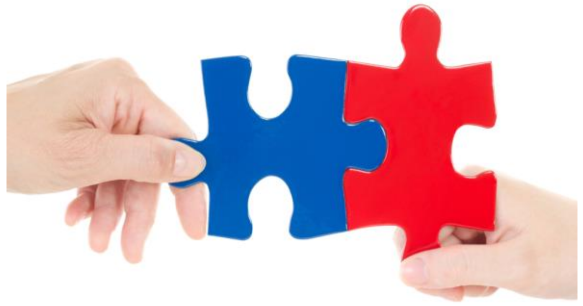
- Single Responsibility Principle
- Open/Close Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

QICKSTEP
SET YOUR CAREER IN MOTION

Solid Principles and C++
Single Responsibility

High Cohesion

- Only **Single** responsibility.
- The module should only have **One** "reason" to change.
- ***This principle is about people.***



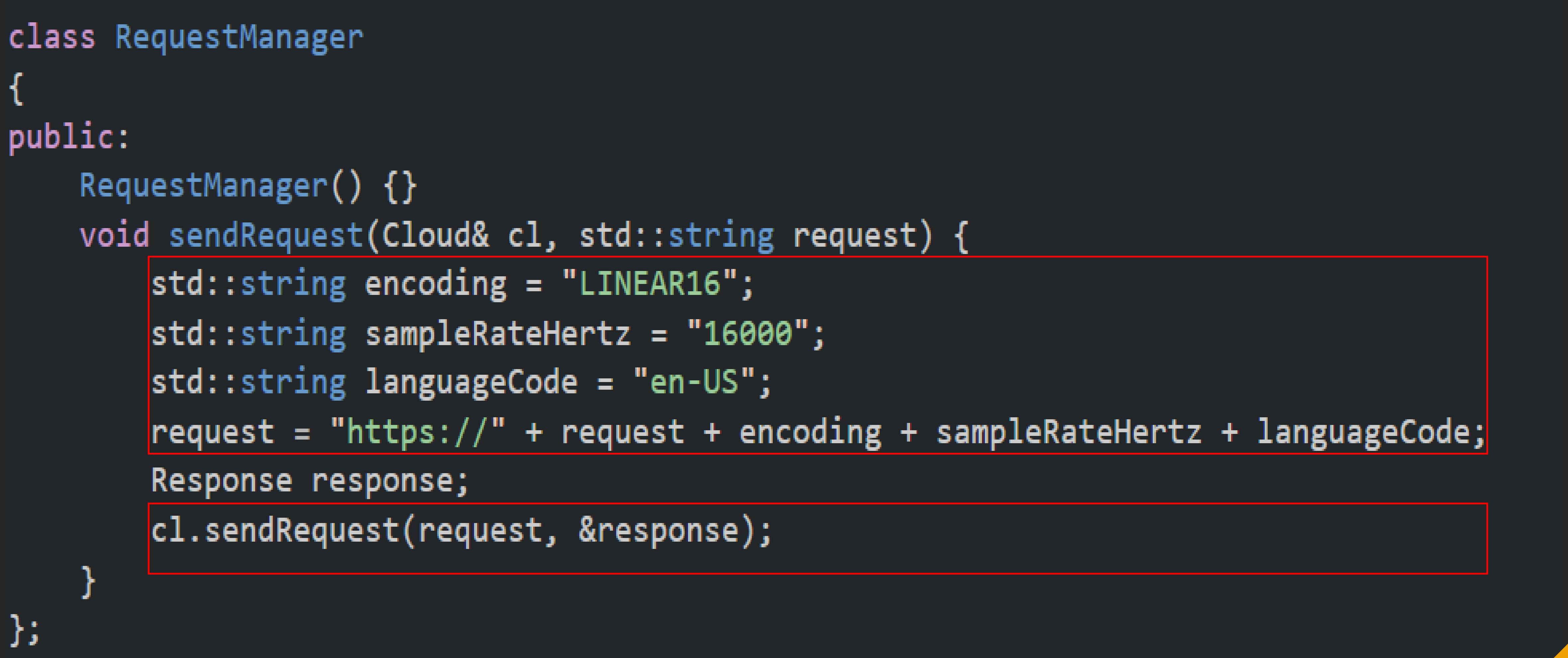
Single Responsibility Principle

Open/Close Principle

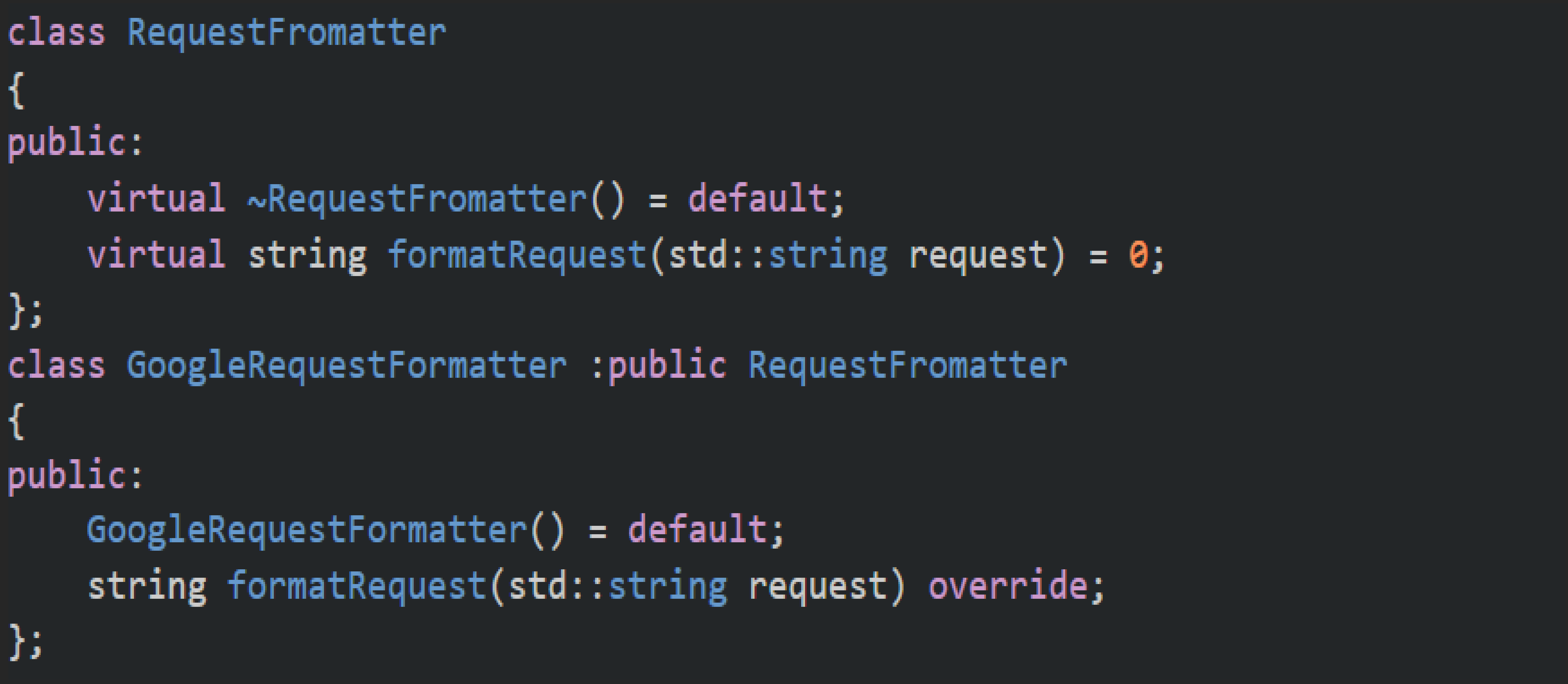
Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle



```cpp
class RequestManager
{
public:
    RequestManager() {}
    void sendRequest(Cloud& cl, std::string request) {
        std::string encoding = "LINEAR16";
        std::string sampleRateHertz = "16000";
        std::string languageCode = "en-US";
        request = "https://" + request + encoding + sampleRateHertz + languageCode;
        Response response;
        cl.sendRequest(request, &response);
    }
};
```



```cpp
class RequestFromatter
{
public:
    virtual ~RequestFromatter() = default;
    virtual string formatRequest(std::string request) = 0;
};
class GoogleRequestFormatter :public RequestFromatter
{
public:
    GoogleRequestFormatter() = default;
    string formatRequest(std::string request) override;
};
```

Solid Principles and C++
Open/Close

All systems change during their life cycles. This must be borne in mind when developing systems expected to last longer than the first version.

—Ivar Jacobson, Swedish computer scientist, 1992

Inheritance/Abstract Interfaces

- Open for **Extension,** Closed for **Modification**.
- Add new requirement without changing the existing code.
- ***Testability.***

Single Responsibility Principle

Open/Close Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

```
class MyCloud
{
    public:
        void sendRequest(std::string request, std::string& response);
}
class RequestManager
{
    private:
        RequestFromatter& m_requestFormatter;
        MyCloud& m_cloud;
    public:
        RequestManager(RequestFromatter& requestFormatter, MyCloud& mcl);
        void sendRequest(std::string request);
};
```

```cpp
class CloudInterface{
    public:
        virtual ~CloudInterface() = default;
        virtual void sendRequest(std::string request, std::string& response) = 0;
};
class MyCloud :public CloudInterface{
    public:
        MyCloud() = default;
        void sendRequest(std::string request, std::string& response) override;
};
class GoogleCloud: public CloudInterface{
    public:
        GoogleCloud() = default;
        void sendRequest(std::string request, std::string& response) override;
};
class RequestManager{
    private:
        RequestFromatter& m_requestFormatter;
        CloudInterface& m_cloud;
    public:
        RequestManager(RequestFromatter& requestFormatter, CloudInterface& mcl);
        void sendRequest(std::string request);
};
```

QICKSTEP
SET YOUR CAREER IN MOTION

Solid Principles and C++
Liskov Substitution

Subtypes must be substitutable for their base types.

—Uncle Bob, Founder of the influential Agile Manifesto

Substitutability

- Extends the **Open/Closed principle** and enables to replace objects of a parent class with objects of a subclass without breaking the application.
- Don’t implement any stricter validation rules on input parameters than implemented by the parent class.
- Apply at the least the same rules to all output parameters as applied by the parent class.

Single Responsibility Principle

Open/Close Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

```cpp
class RecognitionResult {
    public:
        virtual ~RecognitionResult() = default;
        virtual double getConfidenceScore() = 0;
};

class MyCloudRecognitionResult : public RecognitionResult {
    public:
        /**
        * Returns Confidence Score value [10.0-100.0]
        */
        double getConfidenceScore() override;
};
class GoogleCloudRecognitionResult : public RecognitionResult {
    public:
        /**
        * Returns Confidence Score value [0.0-10.0]
        */
        double getConfidenceScore() override;
};
```

```cpp
class RecognitionRequest {
    public:
        virtual ~RecognitionRequest() = default;
        /**
        * Sets Audio Sample Rate
        * @ Param: Sample Rate in Hz
        * @ Return whether Sample rate is valid
        */
        virtual bool setSampleRate(int hrtz) = 0;
};

class MyCloudRecognitionRequest : public RecognitionRequest {
    public:
        /**
        * Sets Audio Sample Rate [8000kHz-16000kHz]
        */
        bool setSampleRate(int hrtz) override;
};
class GoogleCloudRecognitionRequest : public RecognitionRequest {
    public:
        /**
        * Sets Audio Sample Rate [8000kHz-48000kHz]
        */
        bool setSampleRate(int hrtz) override;
};
```

Solid Principles and C++
Interface Segregation

Clients should not be forced to depend upon the interface that they do not use.

Low Coupling

- No one should be forced to depend upon the interface that they do not use.
- Break down the fat interface in to the smaller interfaces.

Single Responsibility Principle

Open/Close Principle

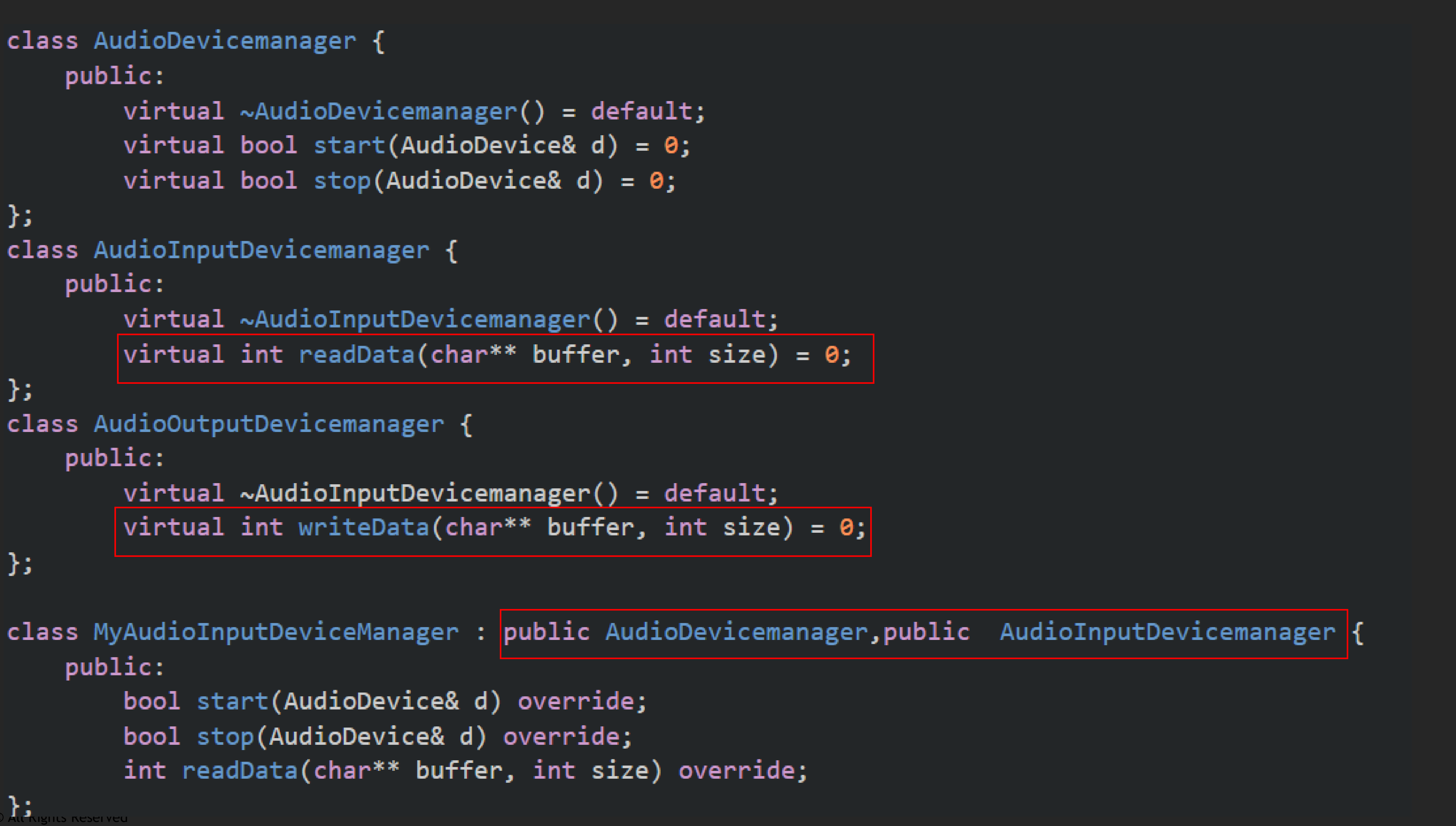
Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

```cpp
class AudioDevicemanager {
    public:
        virtual ~AudioDevicemanager() = default;
        virtual bool start(AudioDevice &d) = 0;
        virtual bool stop(AudioDevice& d) = 0;
        virtual int readData(char** buffer, int size) = 0;
        virtual int writeData(char** buffer, int size) = 0;
};

class MyAudioInputDeviceManager : public AudioDevicemanager {
    public:
        bool start(AudioDevice & d) override;
        bool stop(AudioDevice & d) override;
        int readData(char** buffer, int size) override;
        int writeData(char** buffer, int size) override;
};
```



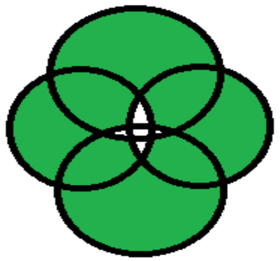
```cpp
class AudioDevicemanager {
    public:
        virtual ~AudioDevicemanager() = default;
        virtual bool start(AudioDevice& d) = 0;
        virtual bool stop(AudioDevice& d) = 0;
};
class AudioInputDevicemanager {
    public:
        virtual ~AudioInputDevicemanager() = default;
        virtual int readData(char** buffer, int size) = 0;
};
class AudioOutputDevicemanager {
    public:
        virtual ~AudioInputDevicemanager() = default;
        virtual int writeData(char** buffer, int size) = 0;
};

class MyAudioInputDeviceManager : public AudioDevicemanager,public  AudioInputDevicemanager {
    public:
        bool start(AudioDevice& d) override;
        bool stop(AudioDevice& d) override;
        int readData(char** buffer, int size) override;
};
```

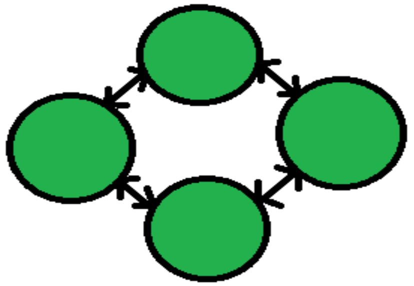
Solid Principles and C++
Dependency Inversion

Loose Coupling

- High level module should not be dependent upon low level modules. Both should be dependent upon the abstraction
- Abstractions should not depend on details. Details should depend on abstractions



Tight coupling:
1. More Interdependency
2. More coordination
3. More information flow



Loose coupling:
1. Less Interdependency
2. Less coordination
3. Less information flow

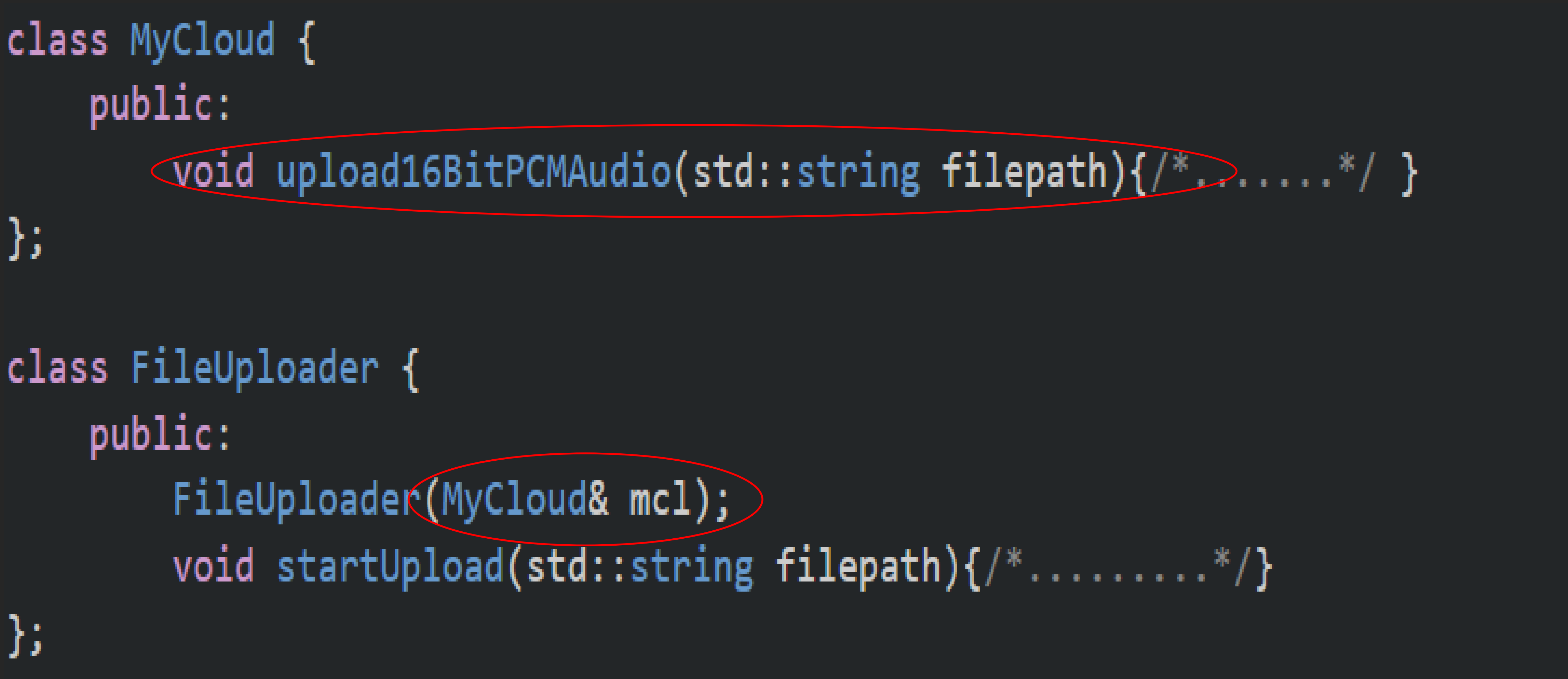
Single Responsibility Principle

Open/Close Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

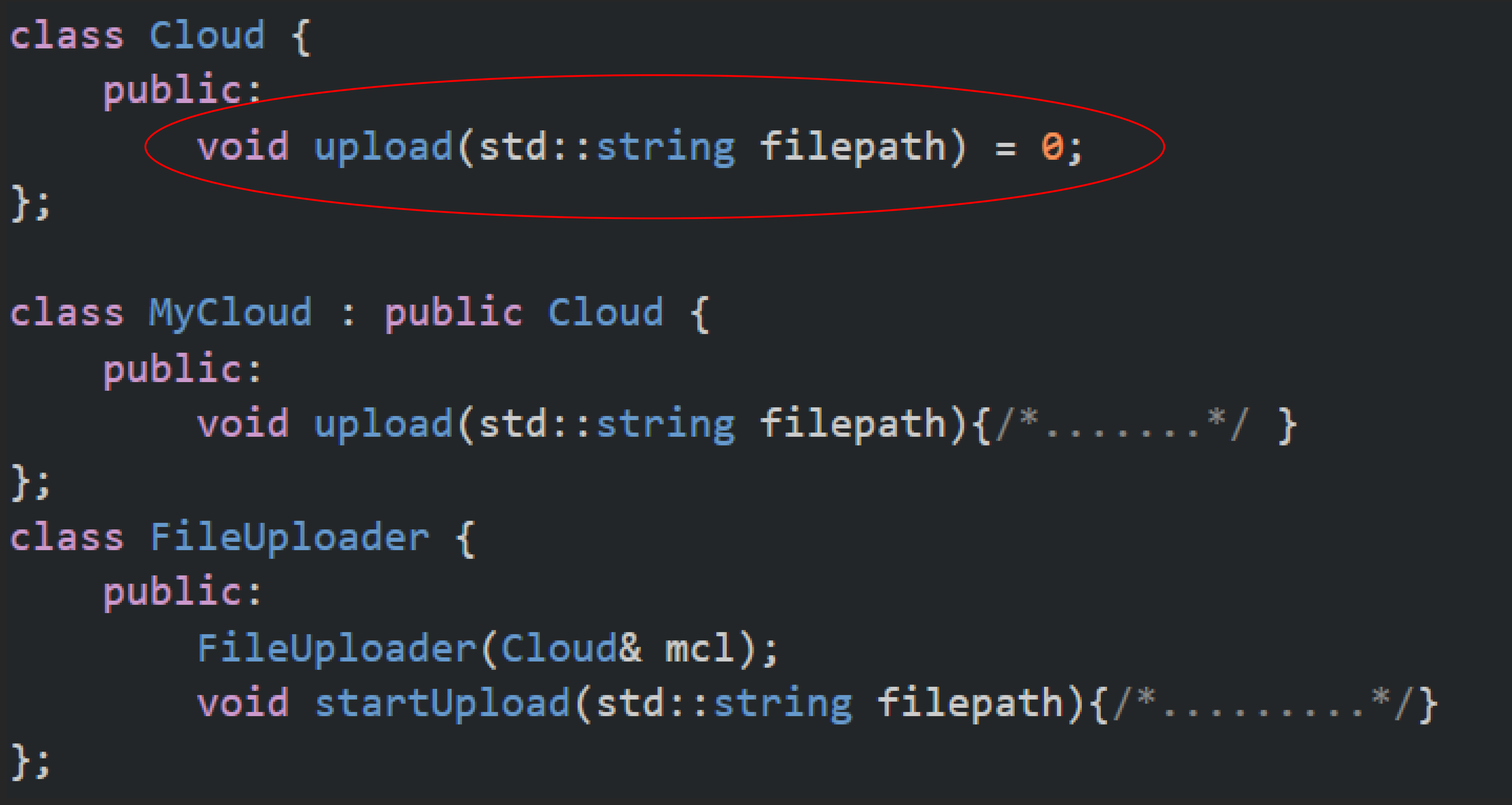


```
class MyCloud {
    public:
        void upload16BitPCMAudio(std::string filepath){/*.......*/ }
};

class FileUploader {
    public:
        FileUploader(MyCloud& mcl);
        void startUpload(std::string filepath){/*.........*/}
};
```

```
class Cloud {
    public:
        void upload16BitPCMAudio(std::string filepath) = 0;
};

class MyCloud : public Cloud {
    public:
        void upload16BitPCMAudio(std::string filepath){/*.......*/ }
};
class FileUploader {
    public:
        FileUploader(Cloud& mcl);
        void startUpload(std::string filepath){/*.........*/}
};
```



```cpp
class Cloud {
    public:
        void upload(std::string filepath) = 0;
};

class MyCloud : public Cloud {
    public:
        void upload(std::string filepath){/*.......*/ }
};
class FileUploader {
    public:
        FileUploader(Cloud& mcl);
        void startUpload(std::string filepath){/*.........*/}
};
```

Key Points

- SOLID enables agility
- SOLID is indirectly promoted by C++ core guidelines
  - C++120,C++121 etc.
- SOLID results in testable code.

Single Responsibility Principle

Open/Close Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

QICKSTEP
SET YOUR CAREER IN MOTION

Thank You

+91-9886-7727-42

www.linkedin.com/in/harishankarsinghyadav

www.qickstep.in