DESIGN PATTERNS FOR BEGINNERS (1/2)

Mustapha Tachouct

2016/10/19

a-volute

3D Sound Expert

Contents

- What are Design Patterns?
- Gang Of Four
- Usage of Design Patterns
- 3 Types of Design Patterns
- 12 Design Patterns

[What are Design Patterns?]

- Design patterns are solutions to general problems that software developers faced during software development

- Design patterns represent the best practices used by experienced object-oriented software developers

- These solutions were obtained by trial and error

[Gang Of Four (GOF)]

- In 1994, four authors Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides published a book

- "Design Patterns - Elements of Reusable Object-Oriented Software"

[3 Facts]

- A Design Pattern is an idea not an implementation

- There are 99+ referenced Design Patterns

- The majority of OO developers uses Design Patterns without to know

[Usage of Design Pattern]

Two main usages :

- Common language for developers

- Best Practices

3 Types in 23 Design Patterns

1. Creational Patterns
2. Structural Patterns
3. Behavioral Patterns

[3 Types : Creational Patterns (1/3)]

- provide a way to create objects while hiding the creation logic
- replace the using of new operator

[3 Types : Structural Patterns (2/3)]

- Manage realtionships between entities
- Define ways to add new functionalities

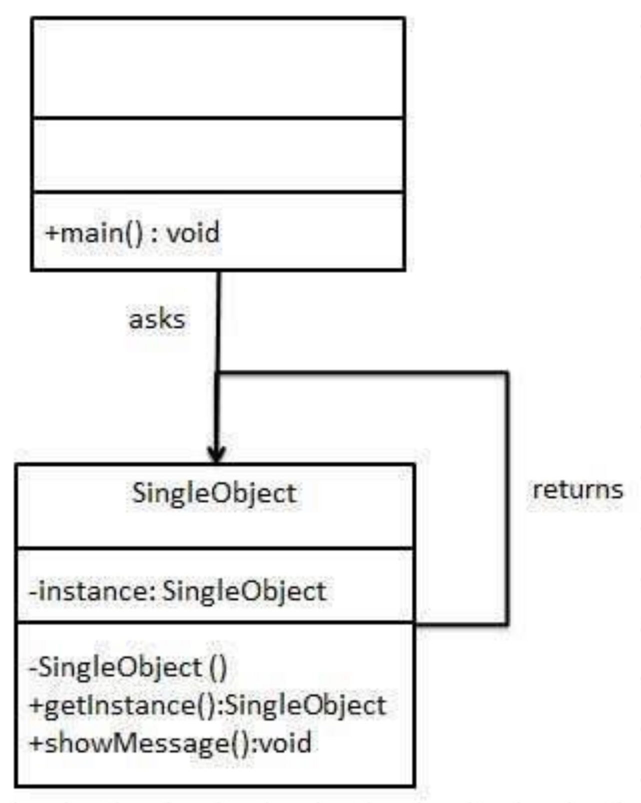
[3 Types : Behavioral Patterns (3/3)]

- concerne with communication between instances
- Increase flexibility the perform of this communication

[12 Patterns]

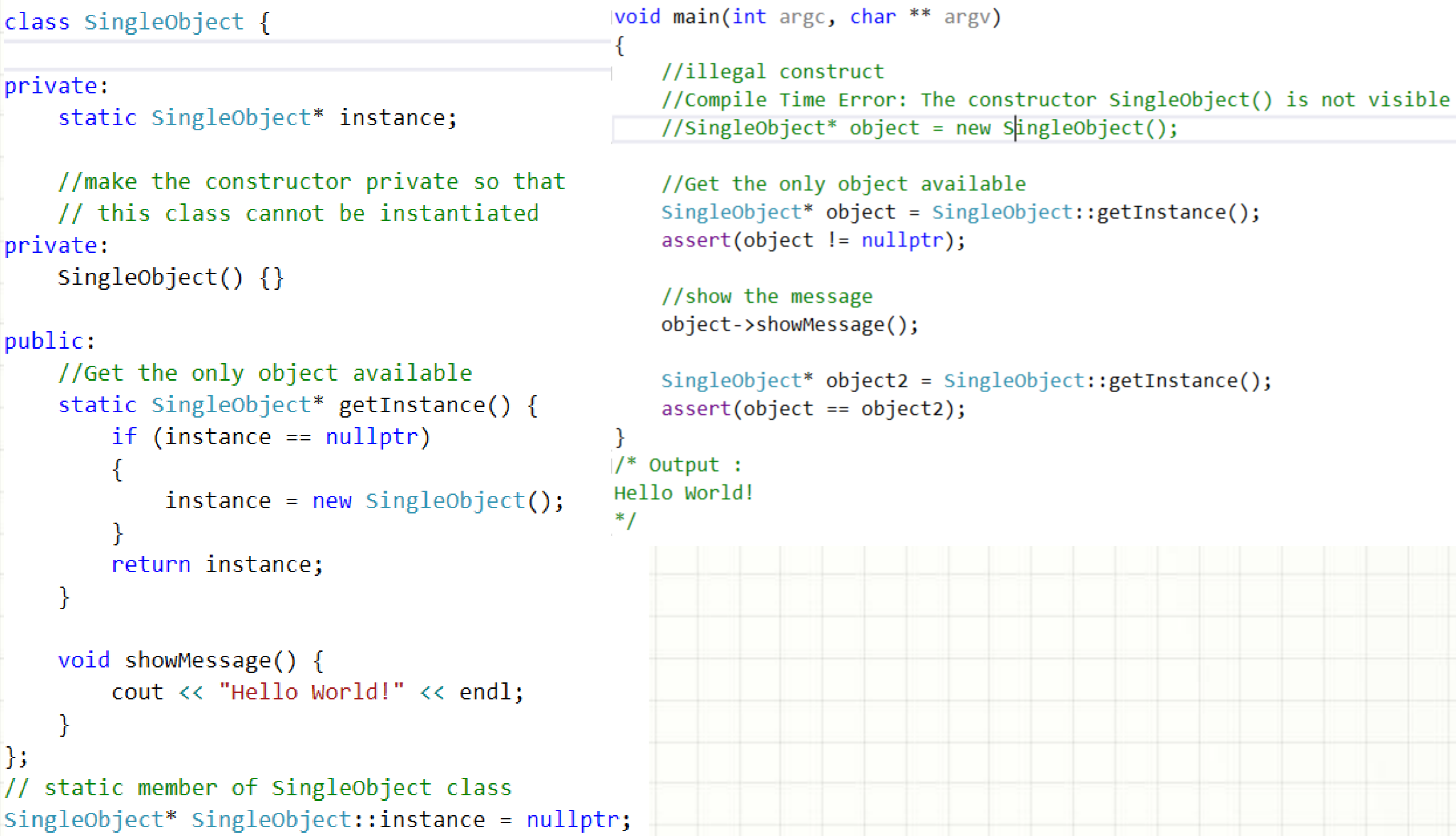
- Singleton, Factory, Iterator, Bridge, Proxy
- Strategy, Chain Of Responsibility, Prototype
- Memento, Adapter Observer, Mediator

[Singleton (1/23)]



- A unique instance for all the program
- Example : Constructor in private + a static member for the instance
- type : creational pattern

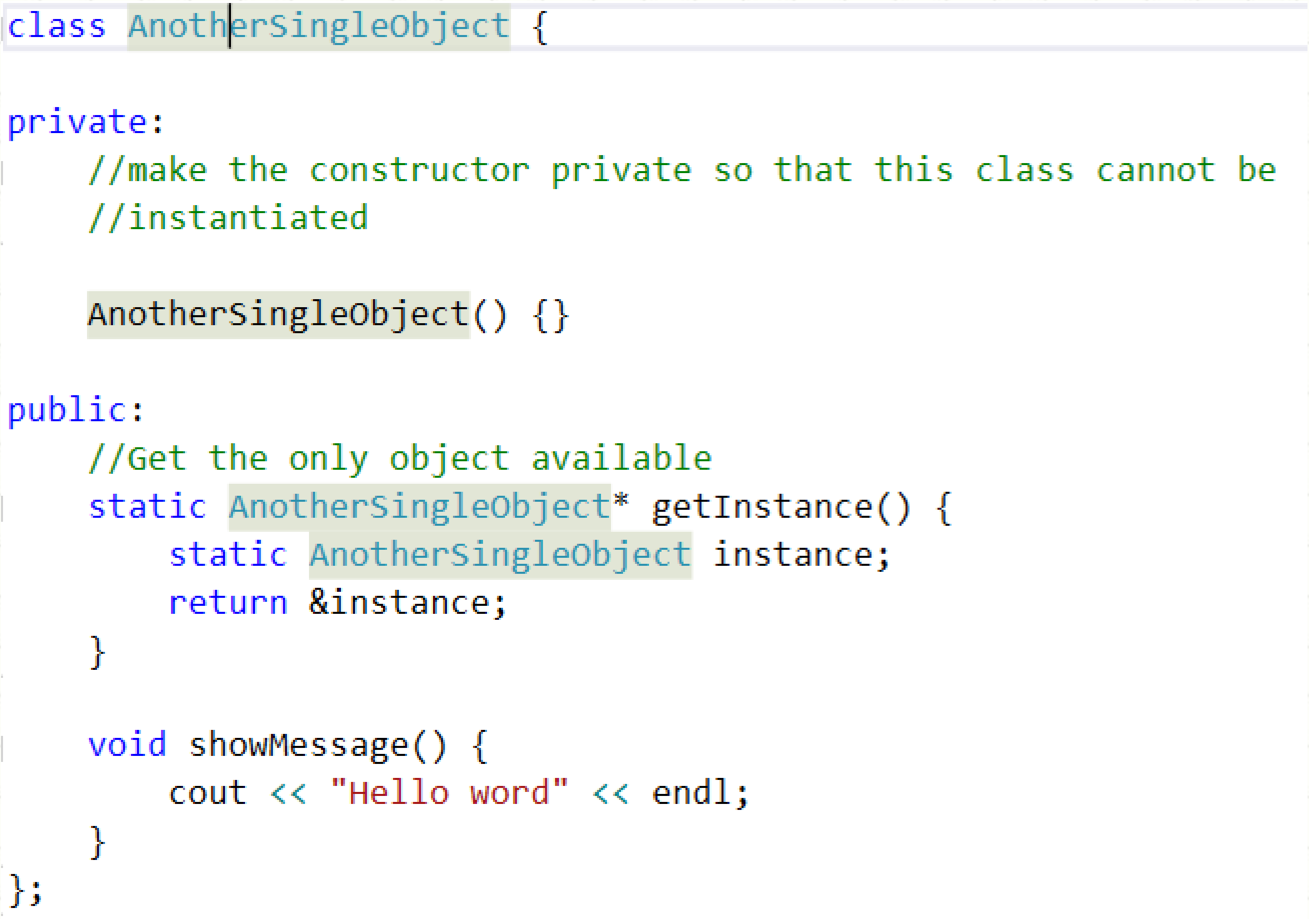
[Singleton (1/23)]



```
class SingleObject {

private:
    static SingleObject* instance;

    //make the constructor private so that
    // this class cannot be instantiated
private:
    SingleObject() {}

public:
    //Get the only object available
    static SingleObject* getInstance() {
        if (instance == nullptr)
        {
            instance = new SingleObject();
        }
        return instance;
    }

    void showMessage() {
        cout << "Hello World!" << endl;
    }
};
// static member of SingleObject class
SingleObject* SingleObject::instance = nullptr;
```

```
void main(int argc, char ** argv)
{
    //illegal construct
    //Compile Time Error: The constructor SingleObject() is not visible
    //SingleObject* object = new SingleObject();

    //Get the only object available
    SingleObject* object = SingleObject::getInstance();
    assert(object != nullptr);

    //show the message
    object->showMessage();

    SingleObject* object2 = SingleObject::getInstance();
    assert(object == object2);
}
/* Output :
Hello World!
*/
```

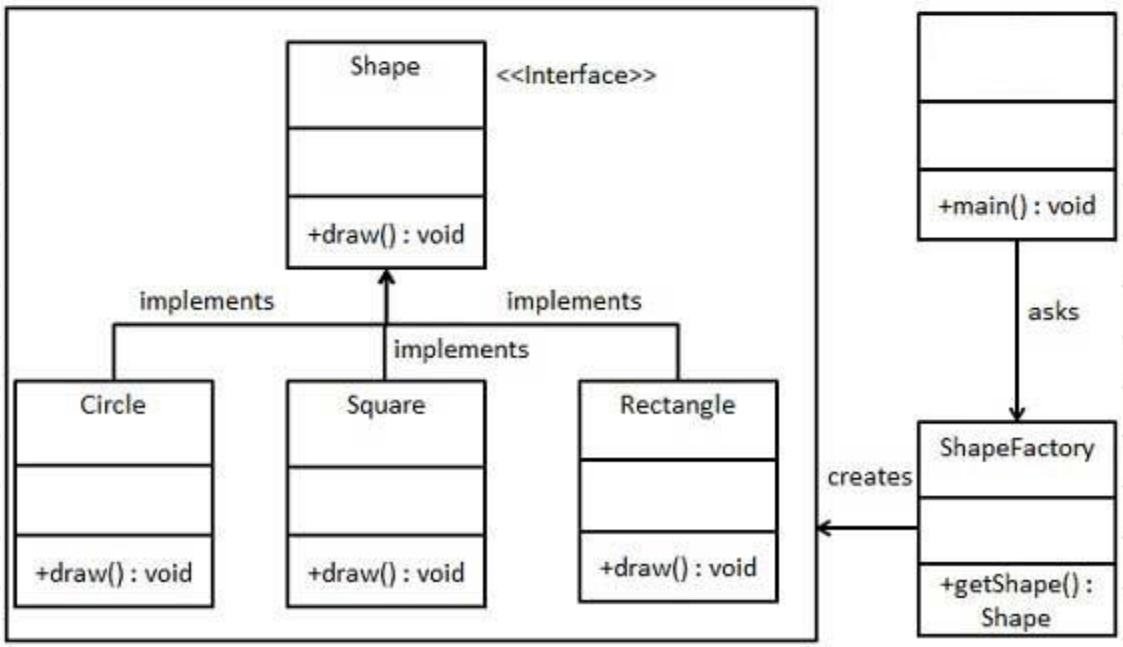
[Singleton (1/23)]



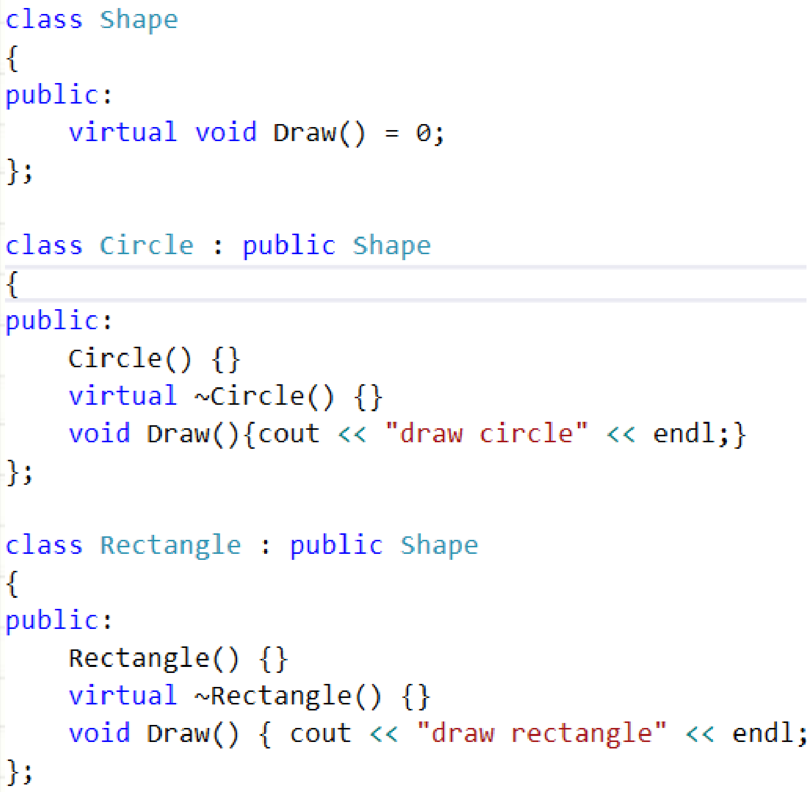
```
class AnotherSingleObject {

private:
    //make the constructor private so that this class cannot be
    //instantiated

    AnotherSingleObject() {}

public:
    //Get the only object available
    static AnotherSingleObject* getInstance() {
        static AnotherSingleObject instance;
        return &instance;
    }

    void showMessage() {
        cout << "Hello word" << endl;
    }
};
```

[Factory (2/23)]

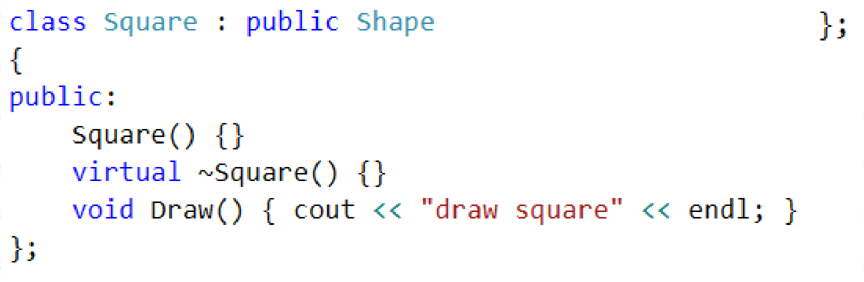
- Create instance(s) without exposing the creation logic
- type : creational pattern



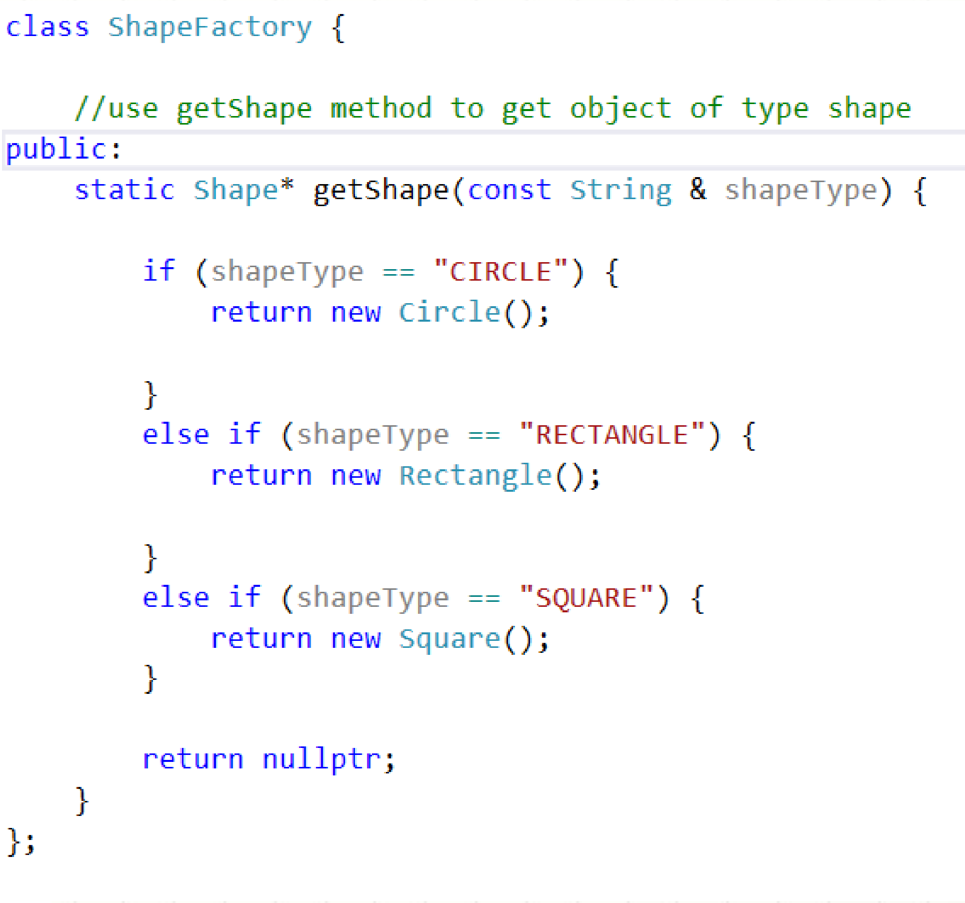
[Factory (2/23)]



```cpp
class Shape
{
public:
    virtual void Draw() = 0;
};

class Circle : public Shape
{
public:
    Circle() {}
    virtual ~Circle() {}
    void Draw(){cout << "draw circle" << endl;}
};

class Rectangle : public Shape
{
public:
    Rectangle() {}
    virtual ~Rectangle() {}
    void Draw() { cout << "draw rectangle" << endl;
};
```

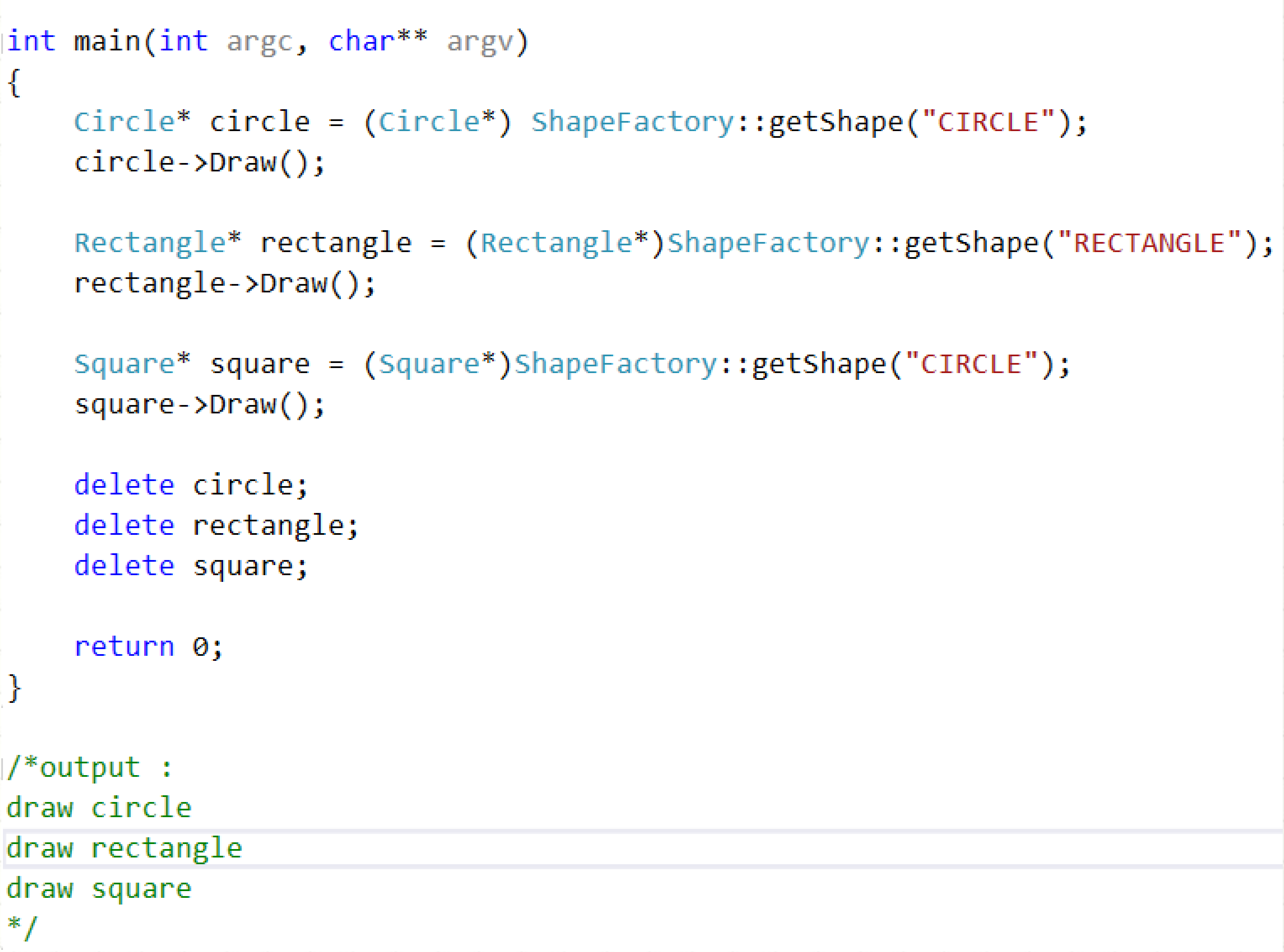


```cpp
class Square : public Shape
{
public:
    Square() {}
    virtual ~Square() {}
    void Draw() { cout << "draw square" << endl; }
};
```



```cpp
class ShapeFactory {

    //use getShape method to get object of type shape
public:
    static Shape* getShape(const String & shapeType) {

        if (shapeType == "CIRCLE") {
            return new Circle();

        }
        else if (shapeType == "RECTANGLE") {
            return new Rectangle();

        }
        else if (shapeType == "SQUARE") {
            return new Square();
        }

        return nullptr;
    }
};
```

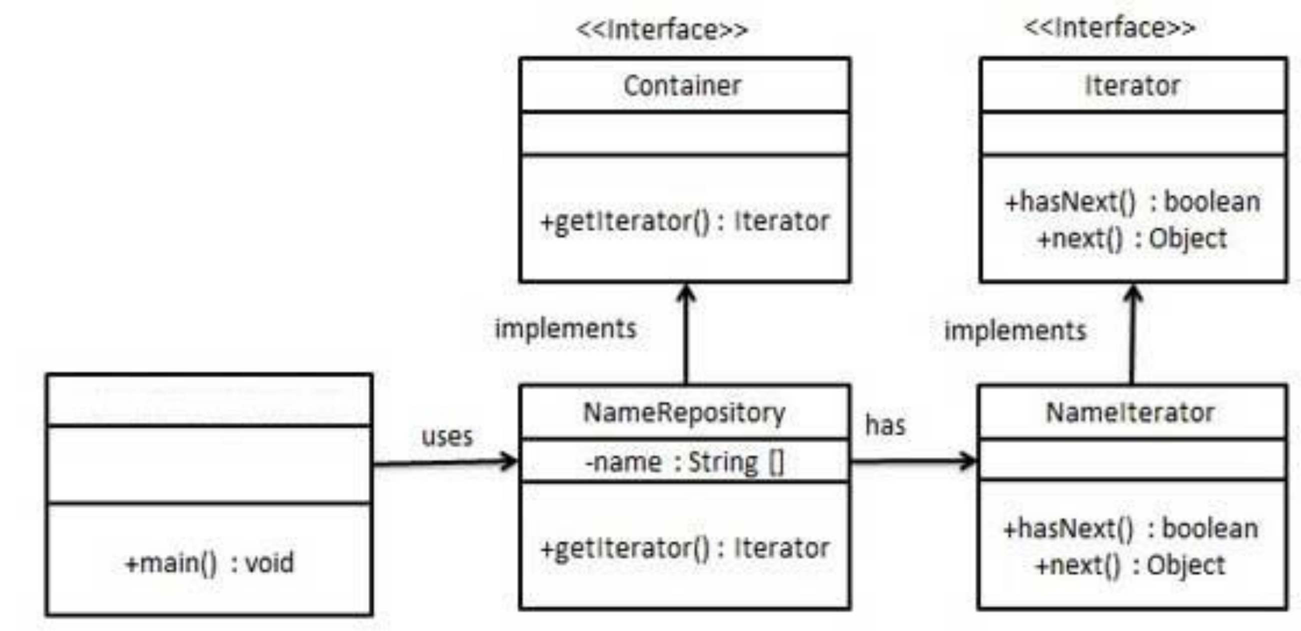
[Factory (2/23)]



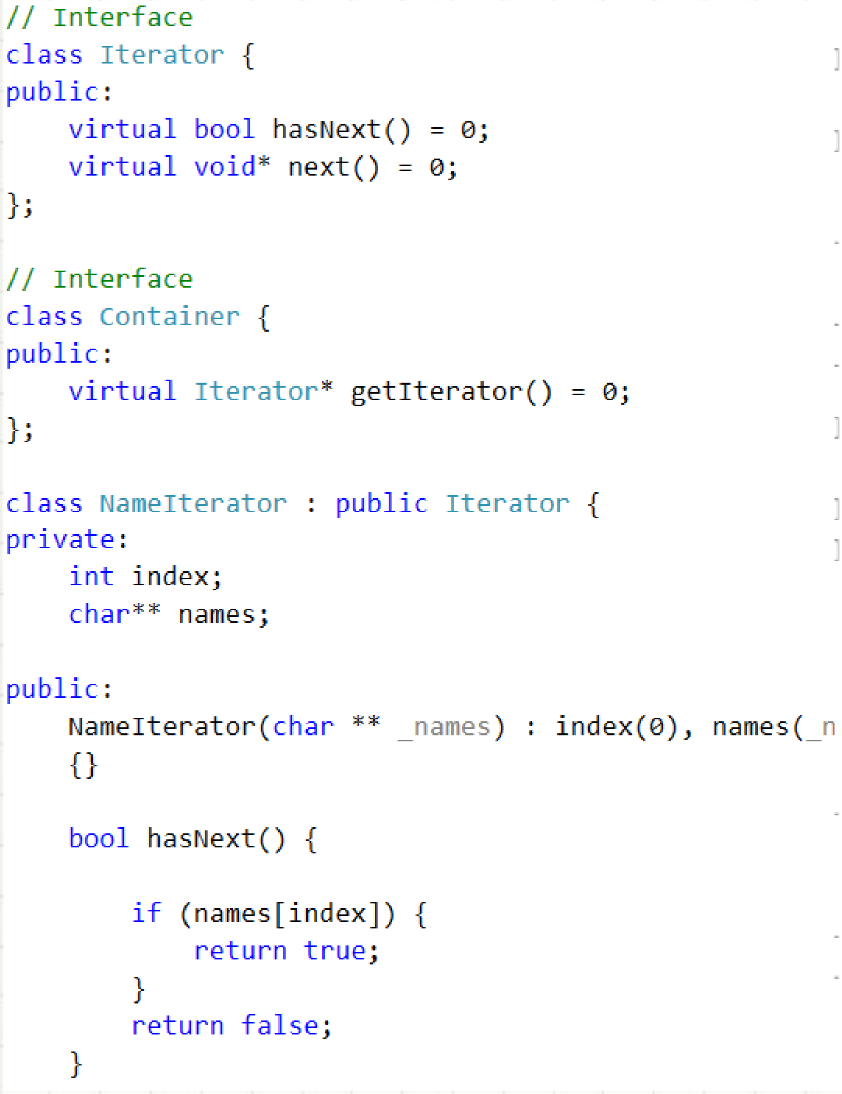
```
int main(int argc, char** argv)
{
    Circle* circle = (Circle*) ShapeFactory::getShape("CIRCLE");
    circle->Draw();

    Rectangle* rectangle = (Rectangle*)ShapeFactory::getShape("RECTANGLE");
    rectangle->Draw();

    Square* square = (Square*)ShapeFactory::getShape("CIRCLE");
    square->Draw();

    delete circle;
    delete rectangle;
    delete square;

    return 0;
}

/*output :
draw circle
draw rectangle
draw square
*/
```

[Iterator (3/23)]

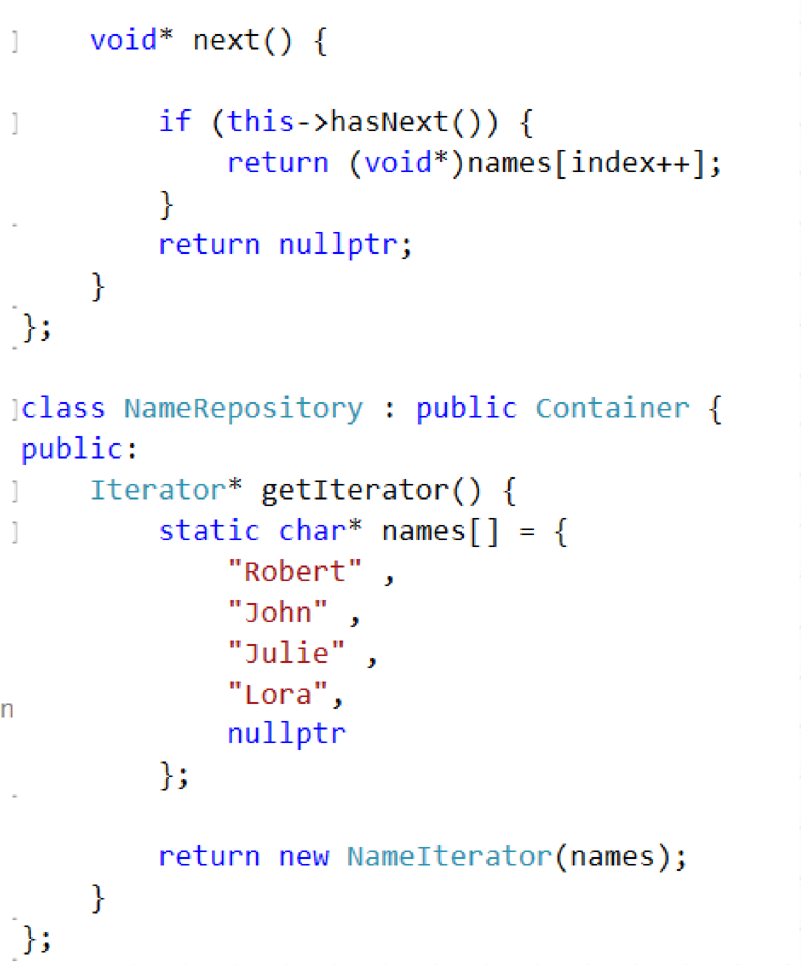
- This pattern is used to get a way to access the elements of a collection object in sequential
- The Data Structure is hidden (list, tree, array, map, stack, ...)
- type : behavioral pattern



[Iterator (3/23)]

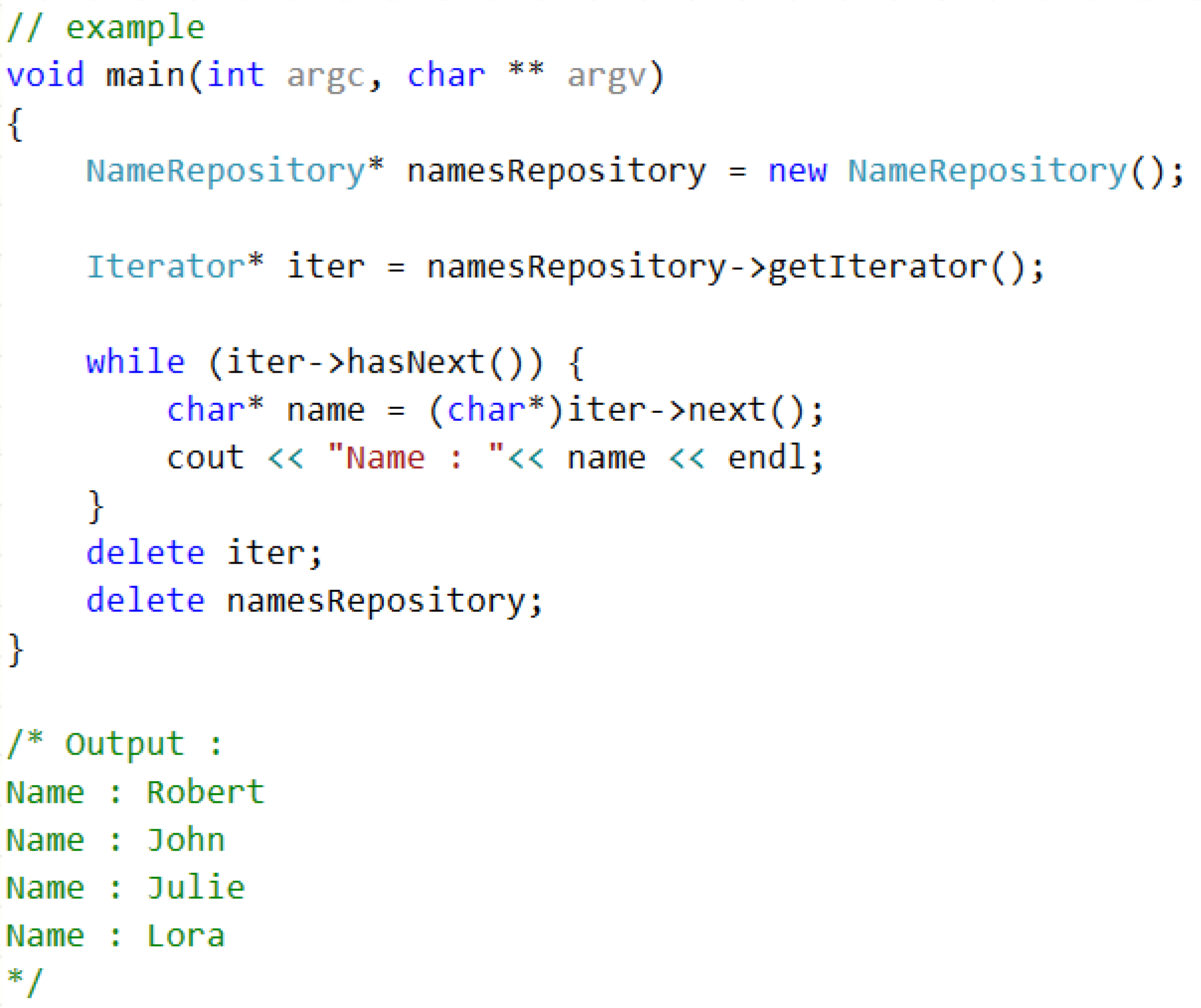


```
// Interface
class Iterator {
public:
    virtual bool hasNext() = 0;
    virtual void* next() = 0;
};

// Interface
class Container {
public:
    virtual Iterator* getIterator() = 0;
};

class NameIterator : public Iterator {
private:
    int index;
    char** names;

public:
    NameIterator(char ** _names) : index(0), names(_n
    {}

    bool hasNext() {

        if (names[index]) {
            return true;
        }
        return false;
    }
```



```
    void* next() {

        if (this->hasNext()) {
            return (void*)names[index++];
        }
        return nullptr;
    }
};

class NameRepository : public Container {
public:
    Iterator* getIterator() {
        static char* names[] = {
            "Robert" ,
            "John" ,
            "Julie" ,
            "Lora",
            nullptr
        };

        return new NameIterator(names);
    }
};
```

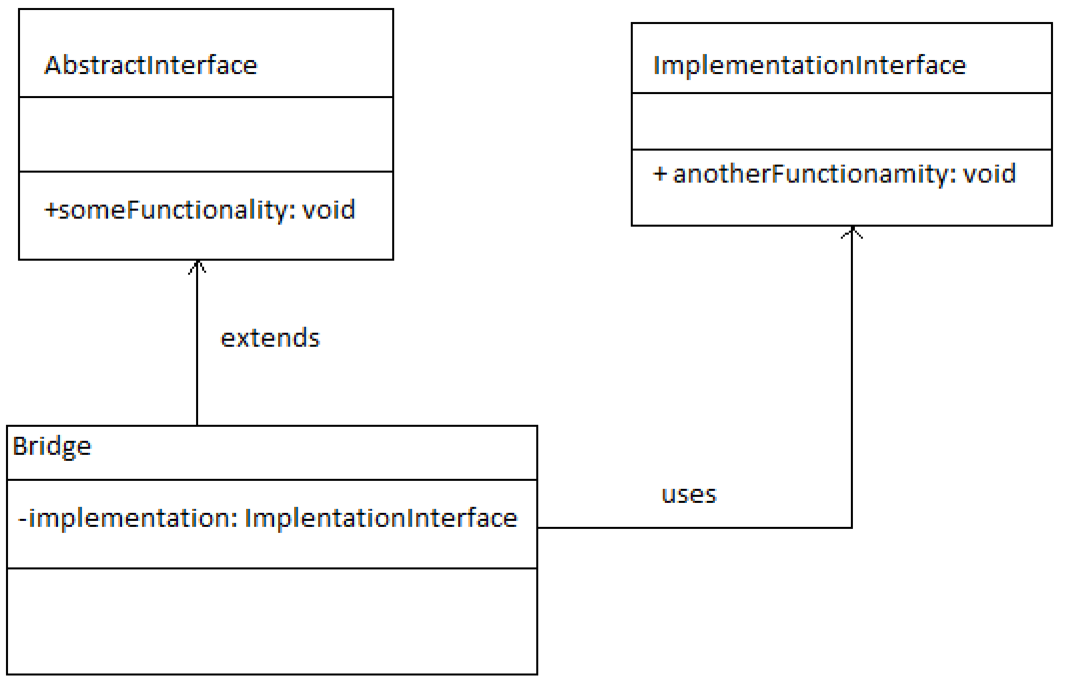
[Iterator (3/23)]



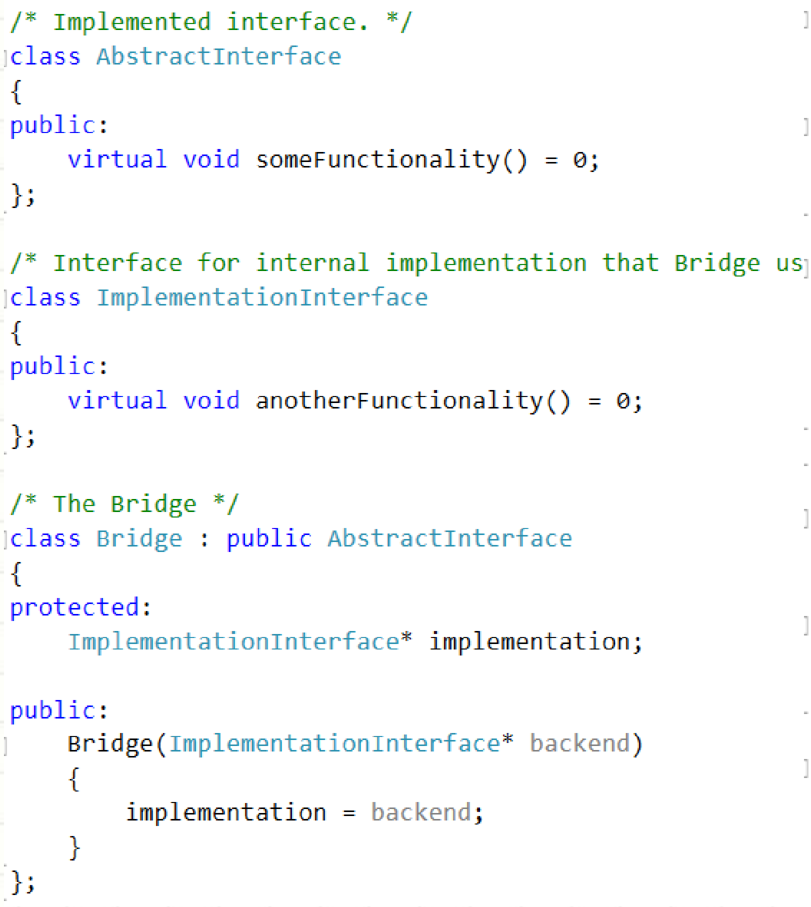
```
// example
void main(int argc, char ** argv)
{
    NameRepository* namesRepository = new NameRepository();

    Iterator* iter = namesRepository->getIterator();

    while (iter->hasNext()) {
        char* name = (char*)iter->next();
        cout << "Name : "<< name << endl;
    }
    delete iter;
    delete namesRepository;
}

/* Output :
Name : Robert
Name : John
Name : Julie
Name : Lora
*/
```

[Bridge (4/23)]

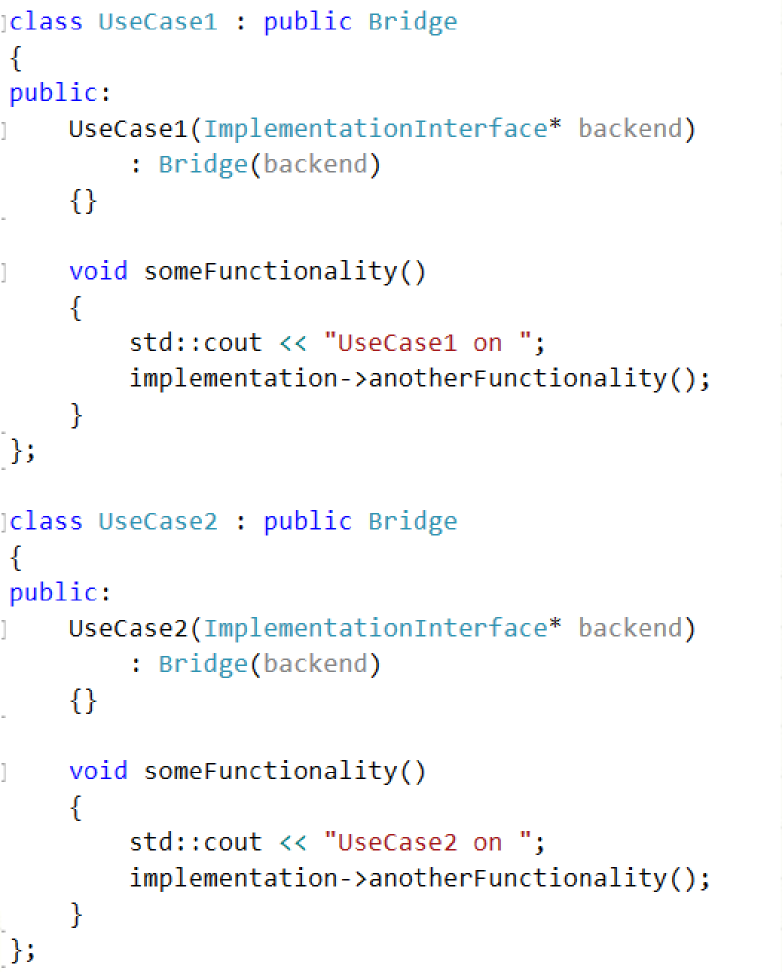
- Used when we need to decouple an abstraction from its implementation
- type : structural pattern



[Bridge (4/23)]

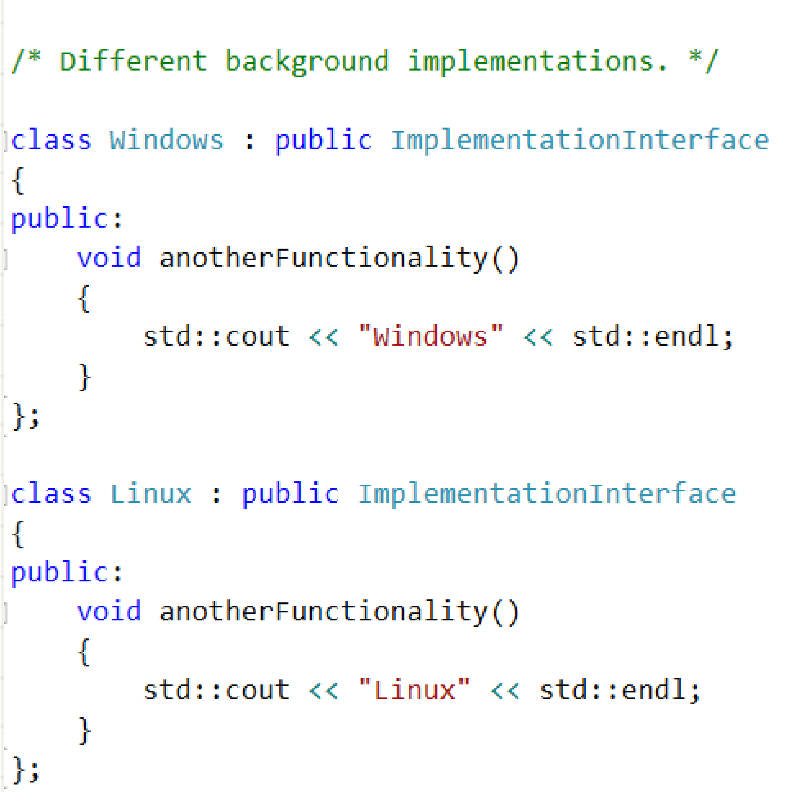


```cpp
/* Implemented interface. */
class AbstractInterface
{
public:
    virtual void someFunctionality() = 0;
};

/* Interface for internal implementation that Bridge us
class ImplementationInterface
{
public:
    virtual void anotherFunctionality() = 0;
};

/* The Bridge */
class Bridge : public AbstractInterface
{
protected:
    ImplementationInterface* implementation;

public:
    Bridge(ImplementationInterface* backend)
    {
        implementation = backend;
    }
};
```

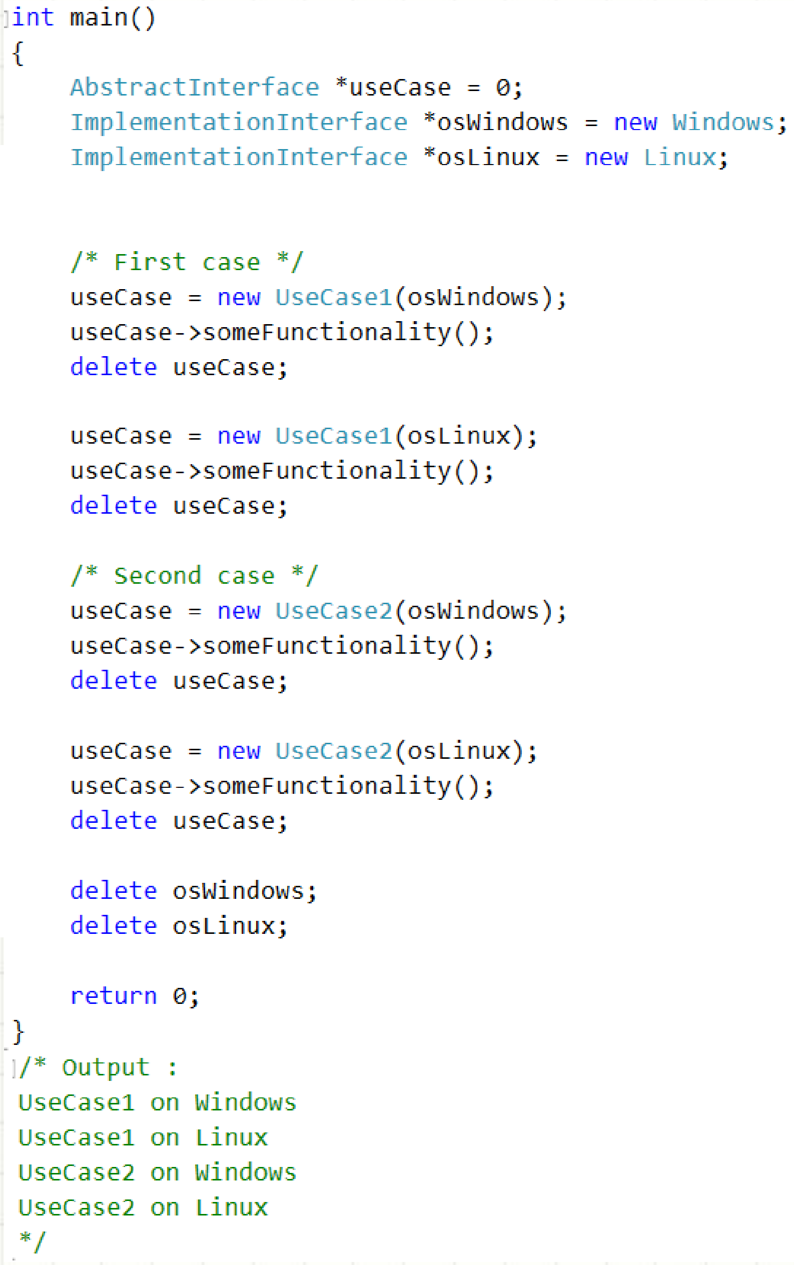


```cpp
class UseCase1 : public Bridge
{
public:
    UseCase1(ImplementationInterface* backend)
        : Bridge(backend)
    {}

    void someFunctionality()
    {
        std::cout << "UseCase1 on ";
        implementation->anotherFunctionality();
    }
};

class UseCase2 : public Bridge
{
public:
    UseCase2(ImplementationInterface* backend)
        : Bridge(backend)
    {}

    void someFunctionality()
    {
        std::cout << "UseCase2 on ";
        implementation->anotherFunctionality();
    }
};
```

[Bridge (4/23)]



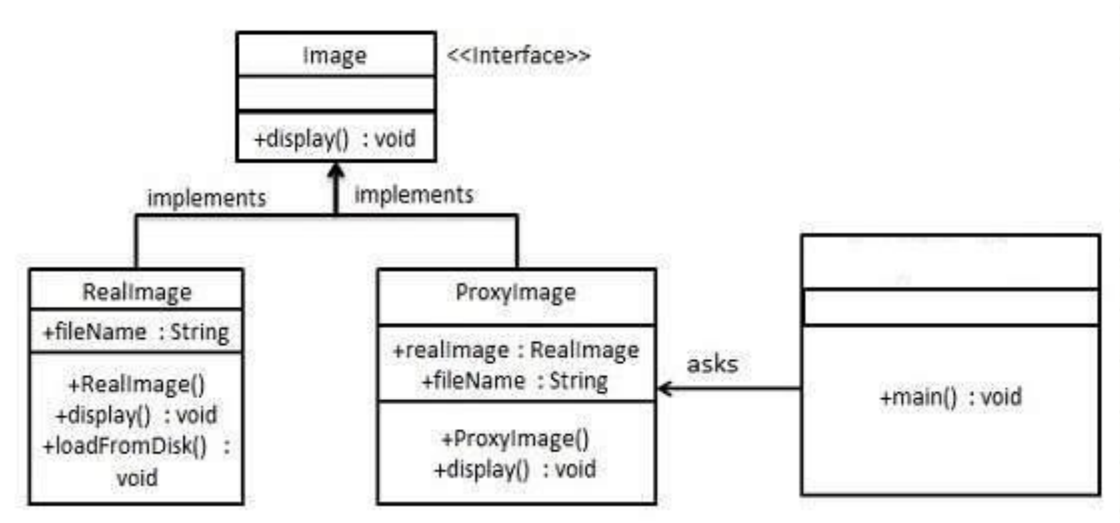
```
/* Different background implementations. */

class Windows : public ImplementationInterface
{
public:
    void anotherFunctionality()
    {
        std::cout << "Windows" << std::endl;
    }
};

class Linux : public ImplementationInterface
{
public:
    void anotherFunctionality()
    {
        std::cout << "Linux" << std::endl;
    }
};
```



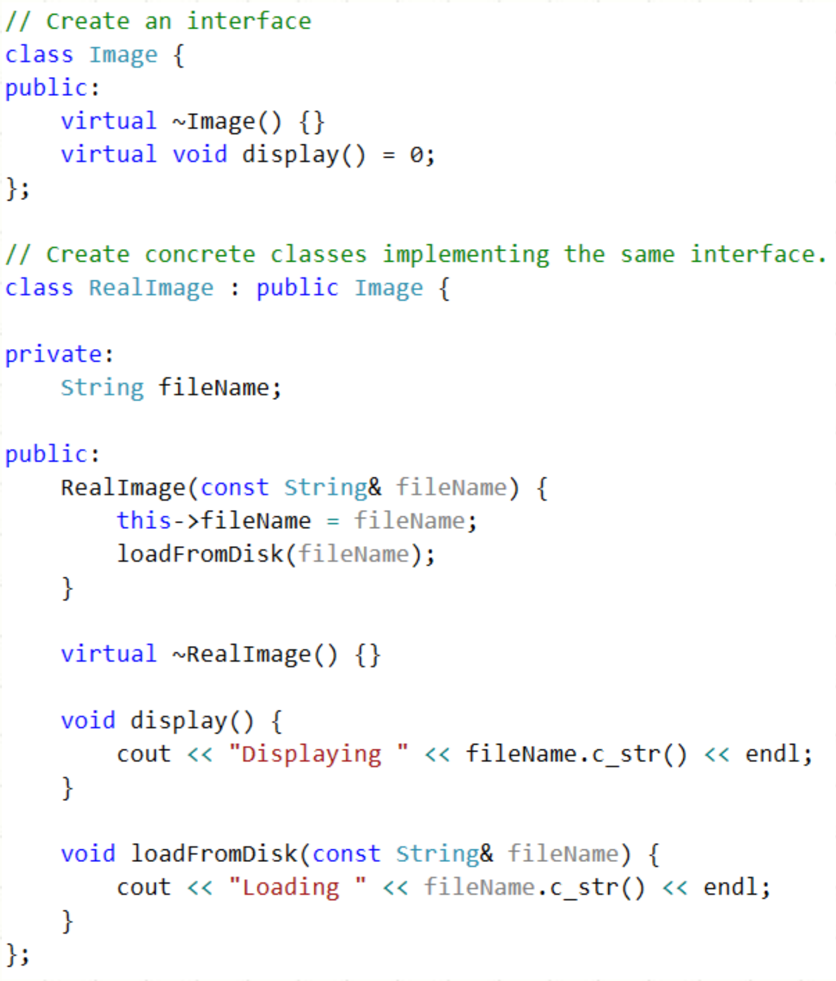
```
int main()
{
    AbstractInterface *useCase = 0;
    ImplementationInterface *osWindows = new Windows;
    ImplementationInterface *osLinux = new Linux;


    /* First case */
    useCase = new UseCase1(osWindows);
    useCase->someFunctionality();
    delete useCase;

    useCase = new UseCase1(osLinux);
    useCase->someFunctionality();
    delete useCase;

    /* Second case */
    useCase = new UseCase2(osWindows);
    useCase->someFunctionality();
    delete useCase;

    useCase = new UseCase2(osLinux);
    useCase->someFunctionality();
    delete useCase;

    delete osWindows;
    delete osLinux;

    return 0;
}
/* Output :
UseCase1 on Windows
UseCase1 on Linux
UseCase2 on Windows
UseCase2 on Linux
*/
```

[Proxy (5/23)]

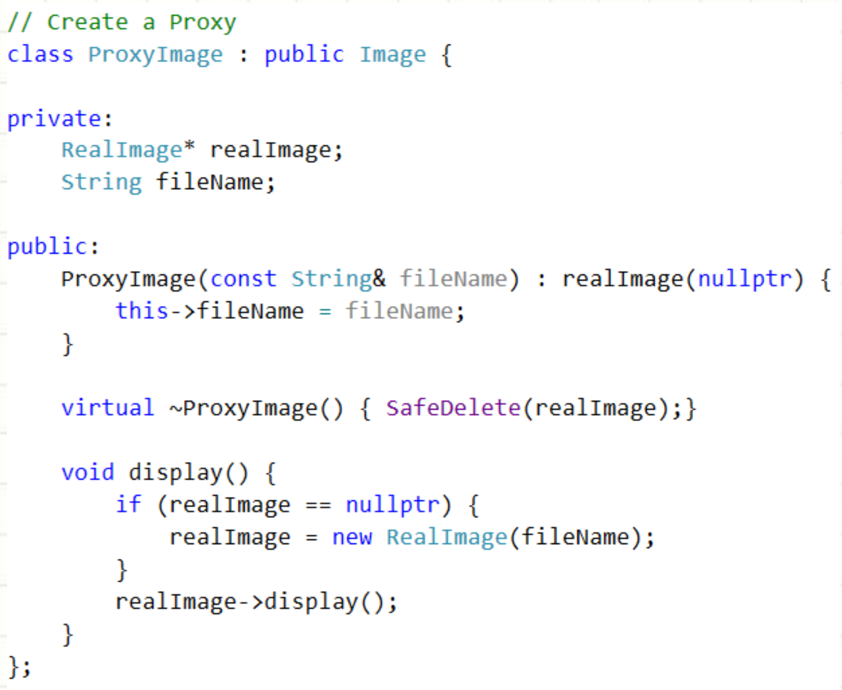
- a proxy is also named a wrapper
- can simply be forwarding to the real object or can provide additional logic
- can also delay the creation of an instance
- type : structural pattern



[Proxy (5/23)]

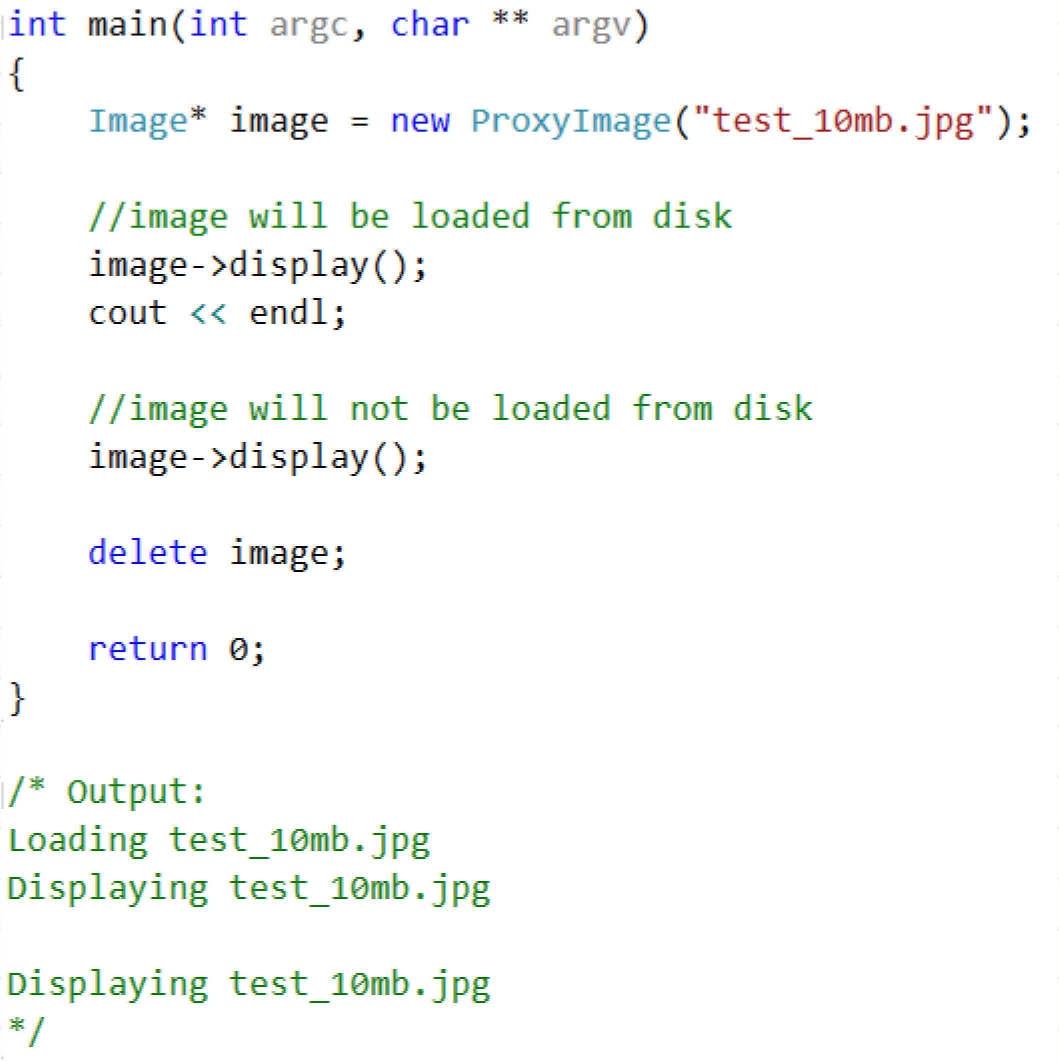


```
// Create an interface
class Image {
public:
    virtual ~Image() {}
    virtual void display() = 0;
};

// Create concrete classes implementing the same interface.
class RealImage : public Image {

private:
    String fileName;

public:
    RealImage(const String& fileName) {
        this->fileName = fileName;
        loadFromDisk(fileName);
    }

    virtual ~RealImage() {}

    void display() {
        cout << "Displaying " << fileName.c_str() << endl;
    }

    void loadFromDisk(const String& fileName) {
        cout << "Loading " << fileName.c_str() << endl;
    }
};
```



```
// Create a Proxy
class ProxyImage : public Image {

private:
    RealImage* realImage;
    String fileName;

public:
    ProxyImage(const String& fileName) : realImage(nullptr) {
        this->fileName = fileName;
    }

    virtual ~ProxyImage() { SafeDelete(realImage);}

    void display() {
        if (realImage == nullptr) {
            realImage = new RealImage(fileName);
        }
        realImage->display();
    }
};
```

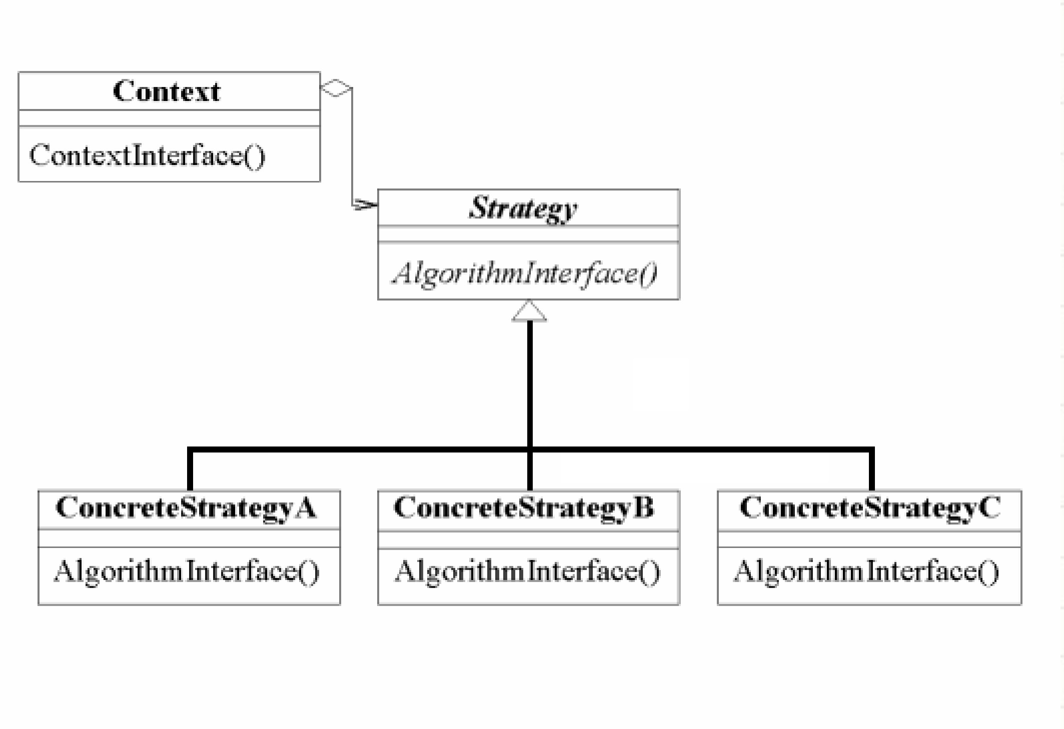
[Proxy (5/23)]



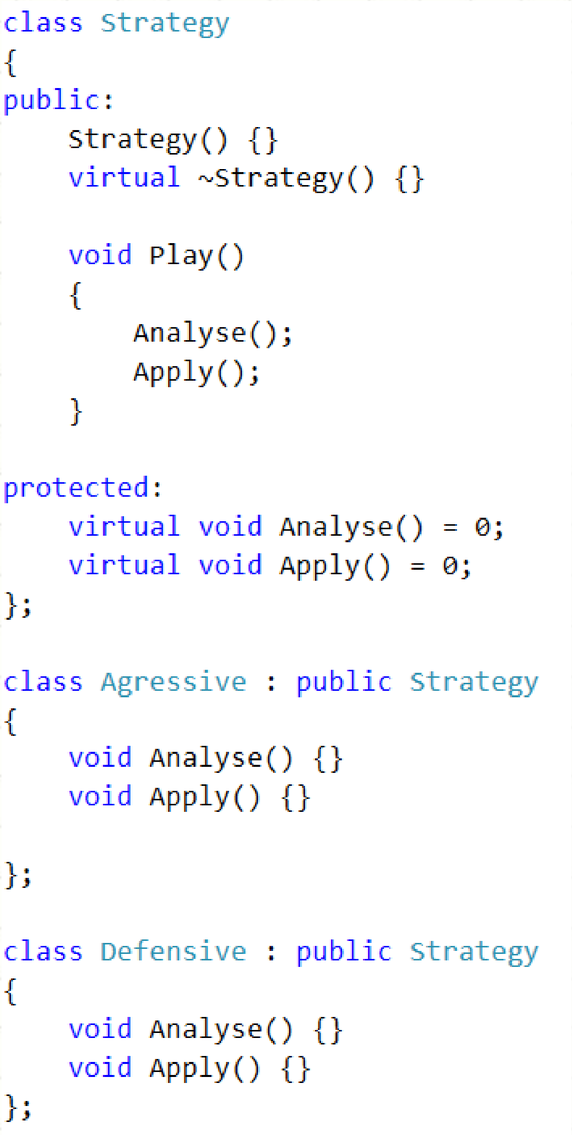
```
int main(int argc, char ** argv)
{
    Image* image = new ProxyImage("test_10mb.jpg");

    //image will be loaded from disk
    image->display();
    cout << endl;

    //image will not be loaded from disk
    image->display();

    delete image;

    return 0;
}

/* Output:
Loading test_10mb.jpg
Displaying test_10mb.jpg

Displaying test_10mb.jpg
*/
```

[Strategy (6/23)]

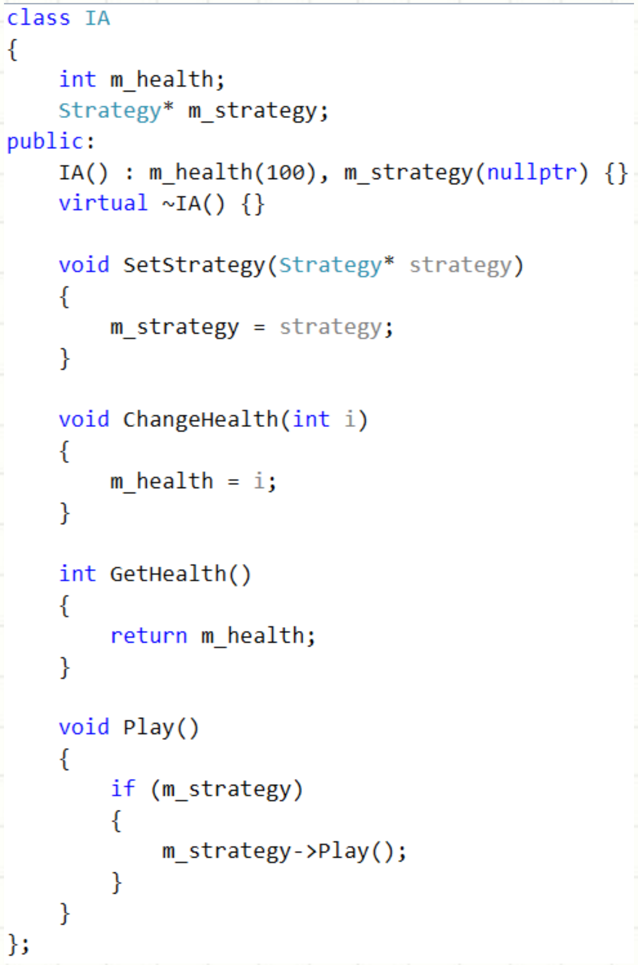
- allow to change a algorytm on fly (among a family of algorytms)
- type : behavior pattern



[Strategy (6/23)]

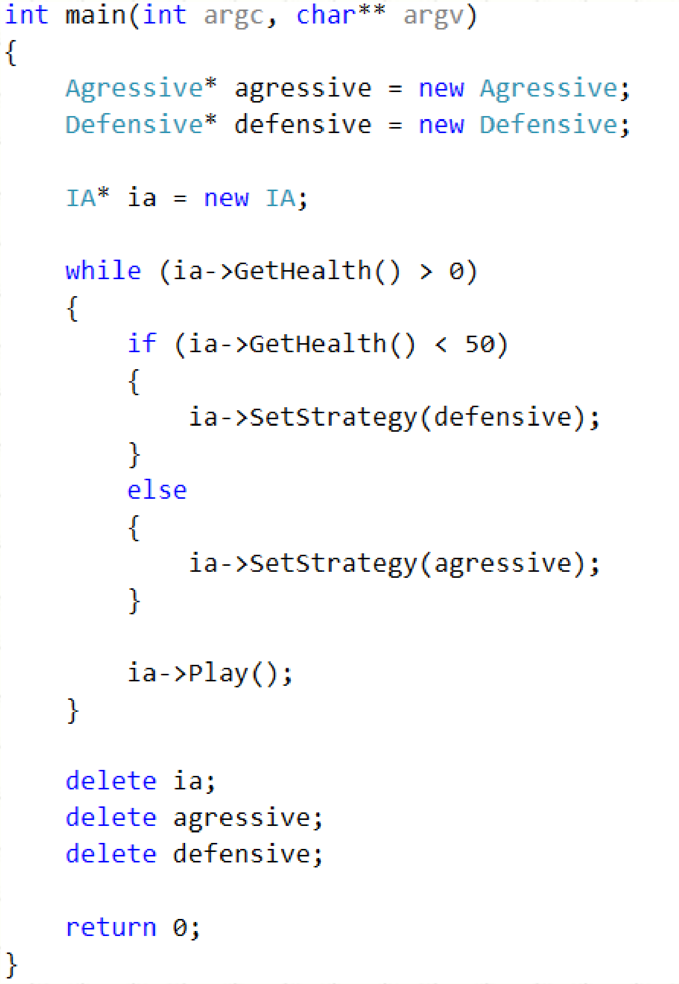


```
class Strategy
{
public:
    Strategy() {}
    virtual ~Strategy() {}

    void Play()
    {
        Analyse();
        Apply();
    }

protected:
    virtual void Analyse() = 0;
    virtual void Apply() = 0;
};

class Agressive : public Strategy
{
    void Analyse() {}
    void Apply() {}

};

class Defensive : public Strategy
{
    void Analyse() {}
    void Apply() {}
};
```



```
class IA
{
    int m_health;
    Strategy* m_strategy;
public:
    IA() : m_health(100), m_strategy(nullptr) {}
    virtual ~IA() {}

    void SetStrategy(Strategy* strategy)
    {
        m_strategy = strategy;
    }

    void ChangeHealth(int i)
    {
        m_health = i;
    }

    int GetHealth()
    {
        return m_health;
    }

    void Play()
    {
        if (m_strategy)
        {
            m_strategy->Play();
        }
    }
};
```

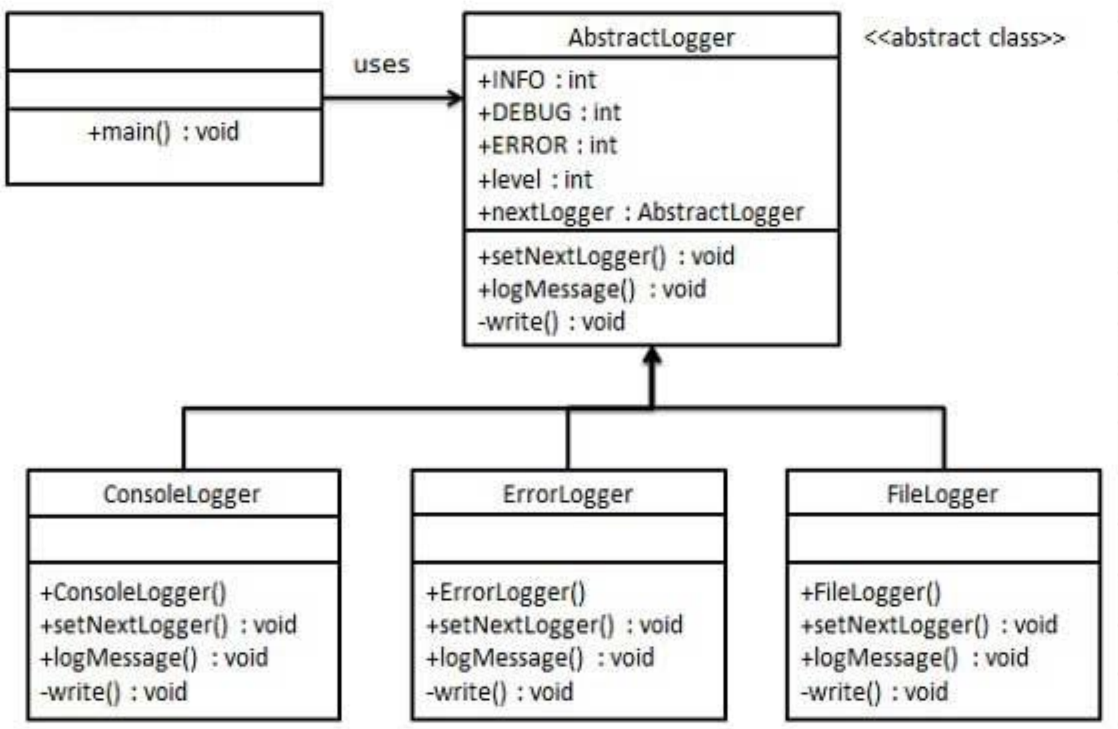
[Strategy (6/23)]



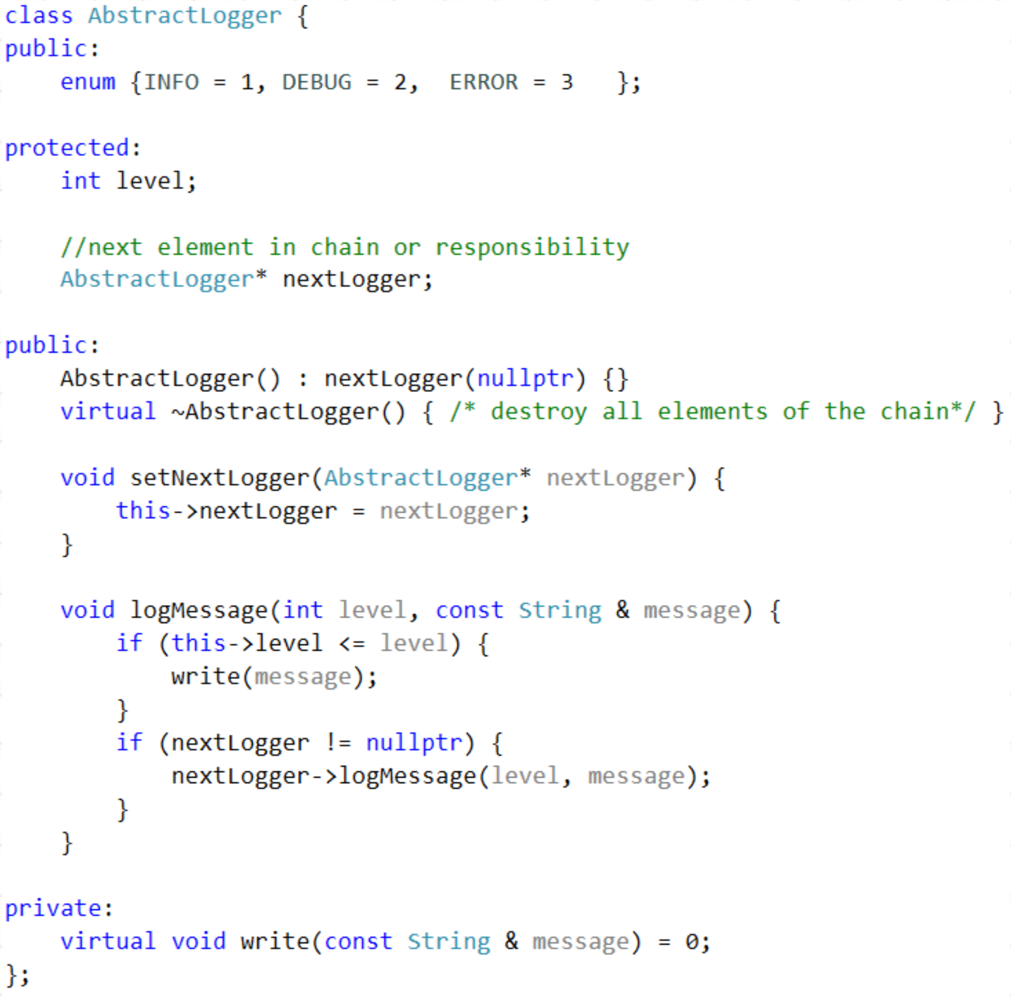
```
int main(int argc, char** argv)
{
    Agressive* agressive = new Agressive;
    Defensive* defensive = new Defensive;

    IA* ia = new IA;

    while (ia->GetHealth() > 0)
    {
        if (ia->GetHealth() < 50)
        {
            ia->SetStrategy(defensive);
        }
        else
        {
            ia->SetStrategy(agressive);
        }

        ia->Play();
    }

    delete ia;
    delete agressive;
    delete defensive;

    return 0;
}
```

[Chain Of Responsability (7/23)]

- creates a chain of receiver objects for a request
- each receiver contains reference to another receiver
- type: behavioral pattern

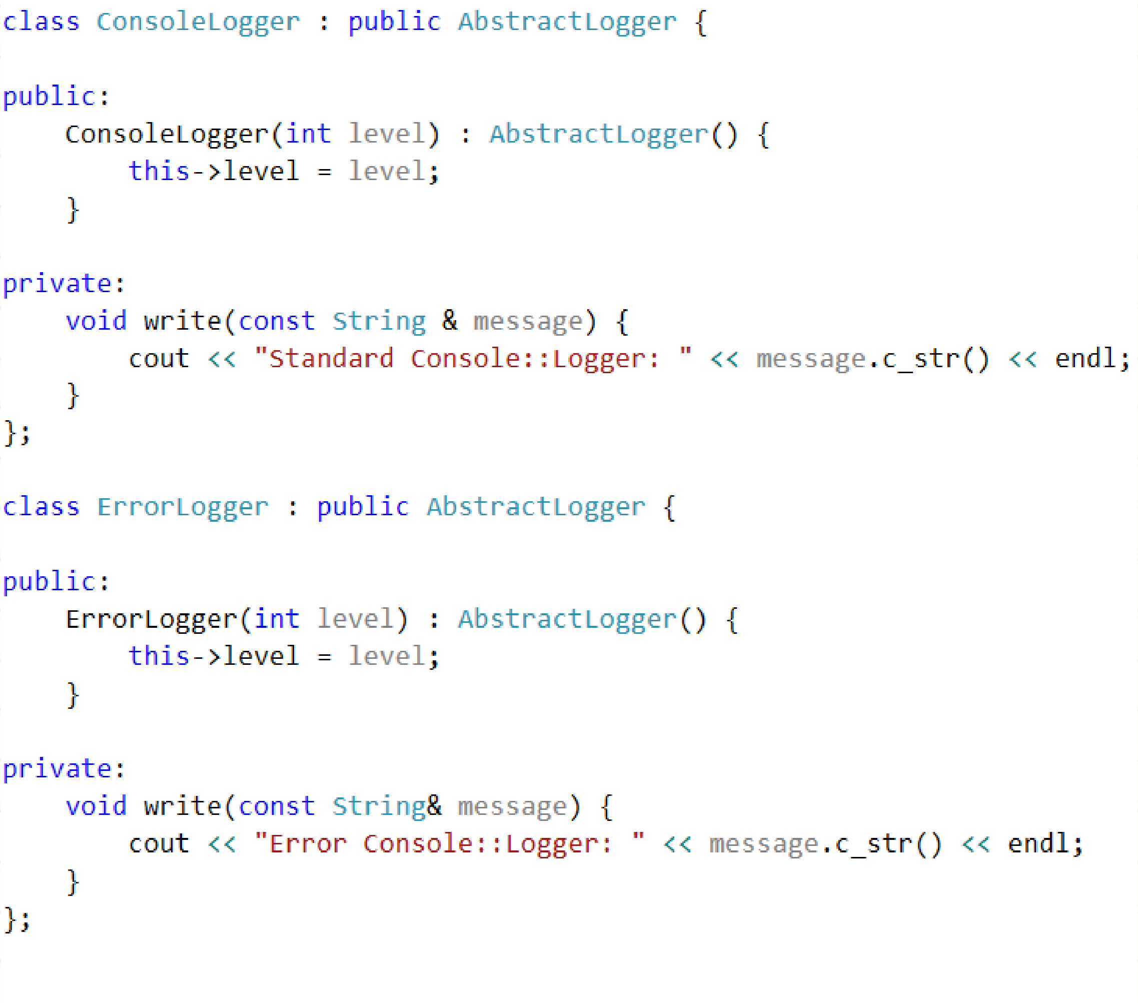


[Chain Of Responsability (7/23)]



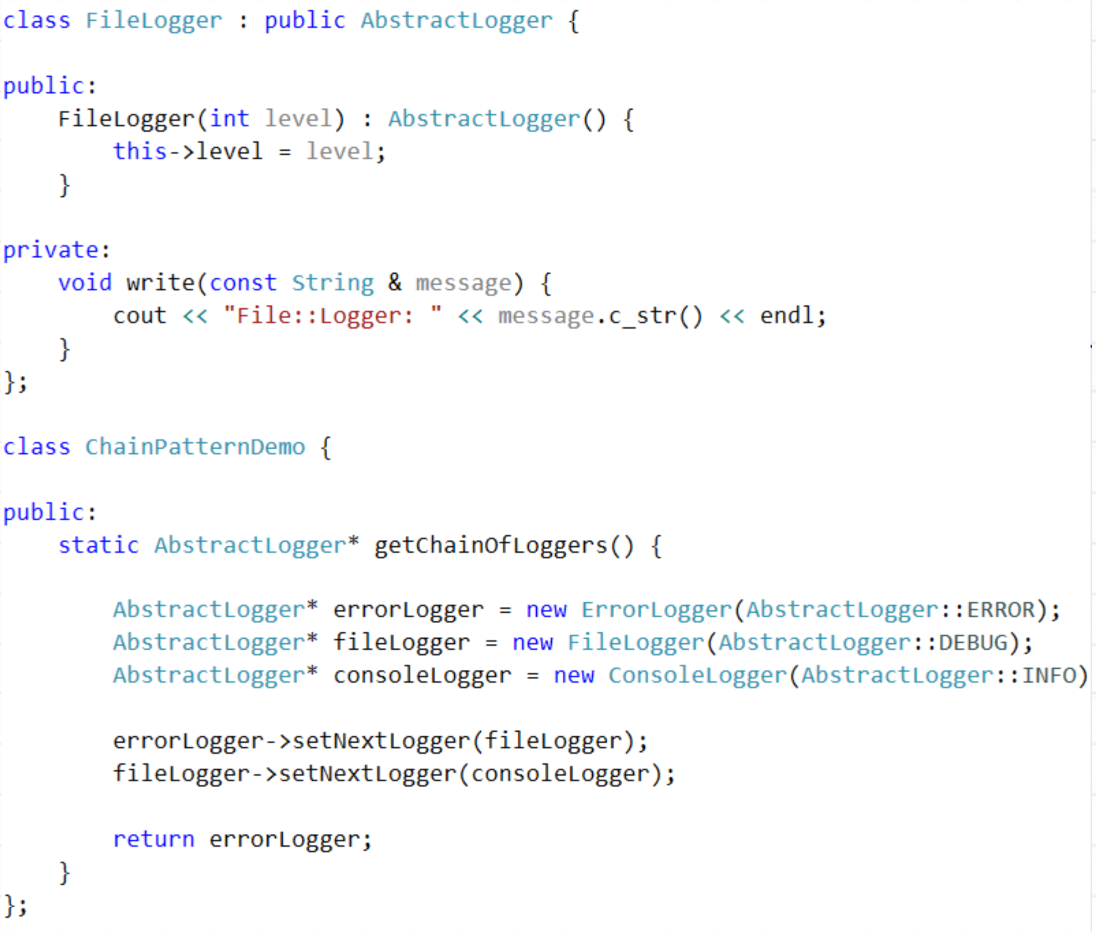
```cpp
class AbstractLogger {
public:
    enum {INFO = 1, DEBUG = 2,  ERROR = 3   };

protected:
    int level;

    //next element in chain or responsibility
    AbstractLogger* nextLogger;

public:
    AbstractLogger() : nextLogger(nullptr) {}
    virtual ~AbstractLogger() { /* destroy all elements of the chain*/ }

    void setNextLogger(AbstractLogger* nextLogger) {
        this->nextLogger = nextLogger;
    }

    void logMessage(int level, const String & message) {
        if (this->level <= level) {
            write(message);
        }
        if (nextLogger != nullptr) {
            nextLogger->logMessage(level, message);
        }
    }

private:
    virtual void write(const String & message) = 0;
};
```

[Chain Of Responsability (7/23)]



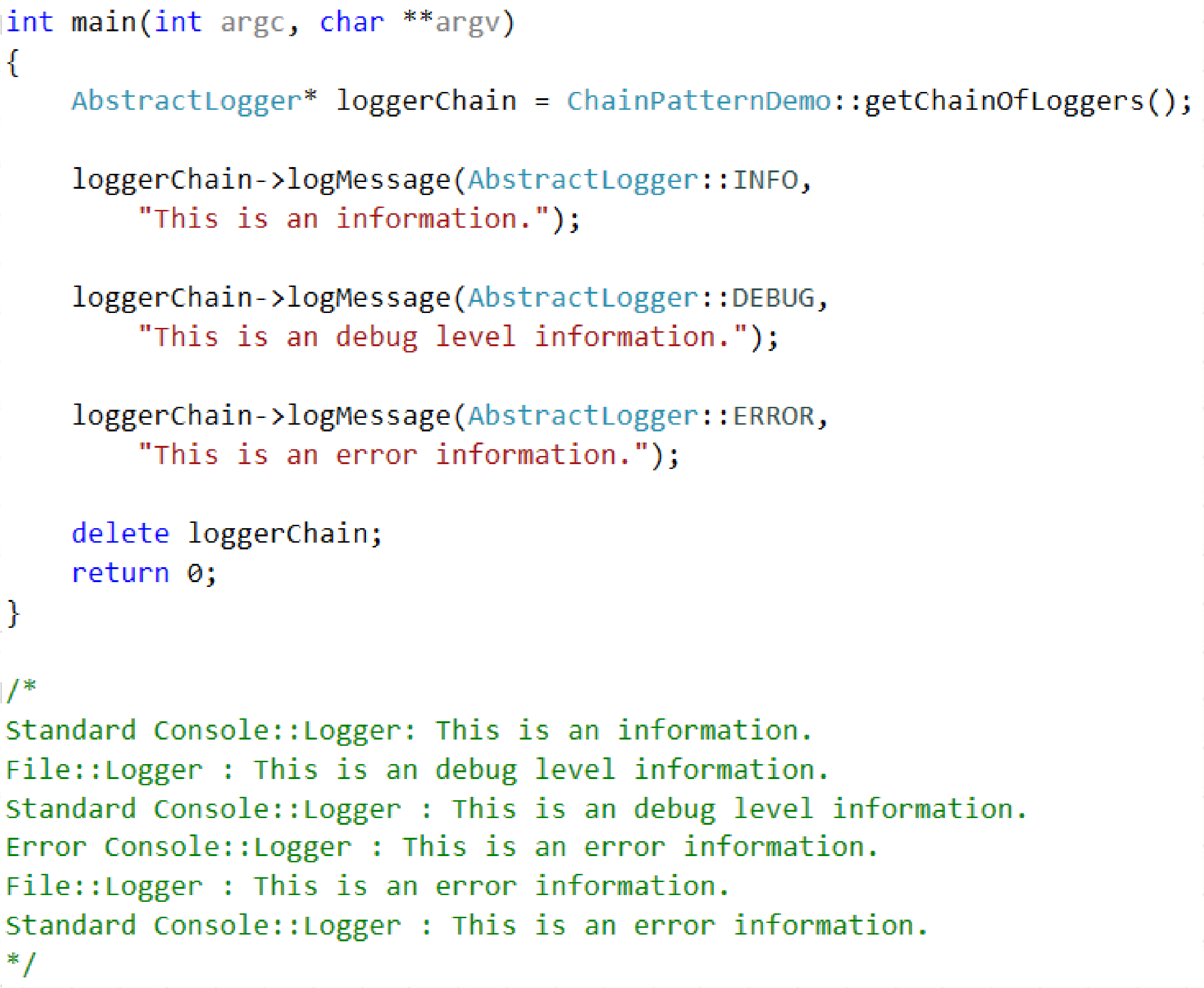
```
class ConsoleLogger : public AbstractLogger {

public:
    ConsoleLogger(int level) : AbstractLogger() {
        this->level = level;
    }

private:
    void write(const String & message) {
        cout << "Standard Console::Logger: " << message.c_str() << endl;
    }
};

class ErrorLogger : public AbstractLogger {

public:
    ErrorLogger(int level) : AbstractLogger() {
        this->level = level;
    }

private:
    void write(const String& message) {
        cout << "Error Console::Logger: " << message.c_str() << endl;
    }
};
```

[Chain Of Responsability (7/23)]



```
class FileLogger : public AbstractLogger {

public:
    FileLogger(int level) : AbstractLogger() {
        this->level = level;
    }

private:
    void write(const String & message) {
        cout << "File::Logger: " << message.c_str() << endl;
    }
};

class ChainPatternDemo {

public:
    static AbstractLogger* getChainOfLoggers() {

        AbstractLogger* errorLogger = new ErrorLogger(AbstractLogger::ERROR);
        AbstractLogger* fileLogger = new FileLogger(AbstractLogger::DEBUG);
        AbstractLogger* consoleLogger = new ConsoleLogger(AbstractLogger::INFO)

        errorLogger->setNextLogger(fileLogger);
        fileLogger->setNextLogger(consoleLogger);

        return errorLogger;
    }
};
```

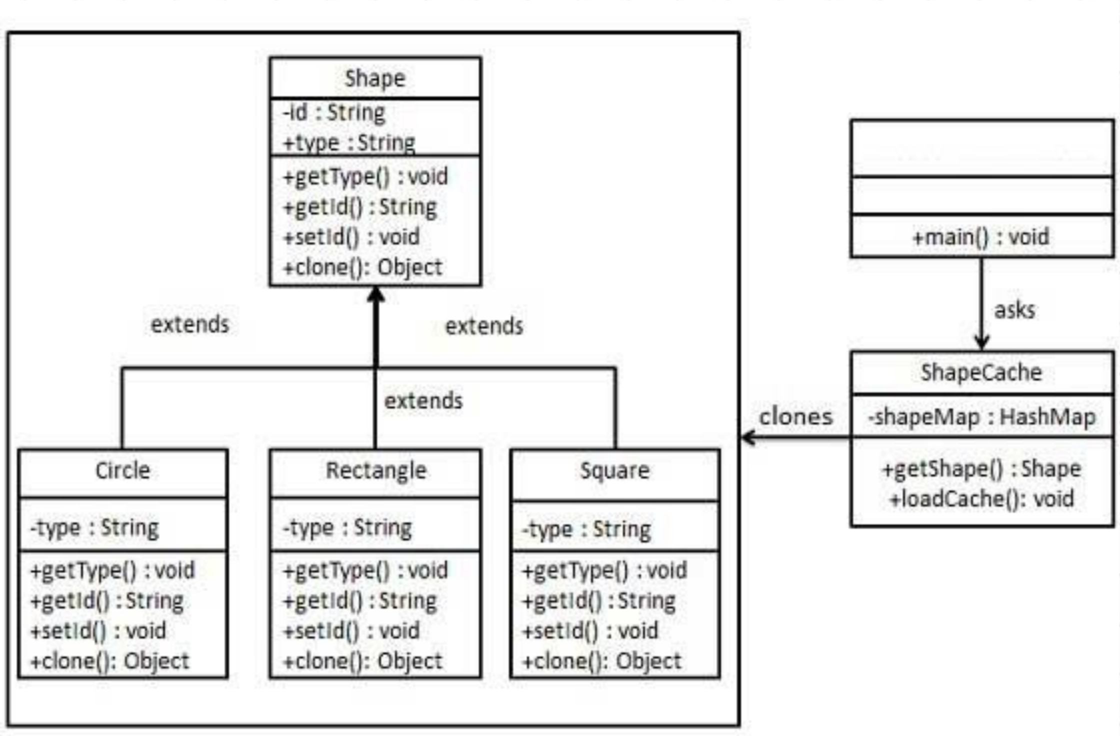
[Chain Of Responsability (7/23)]



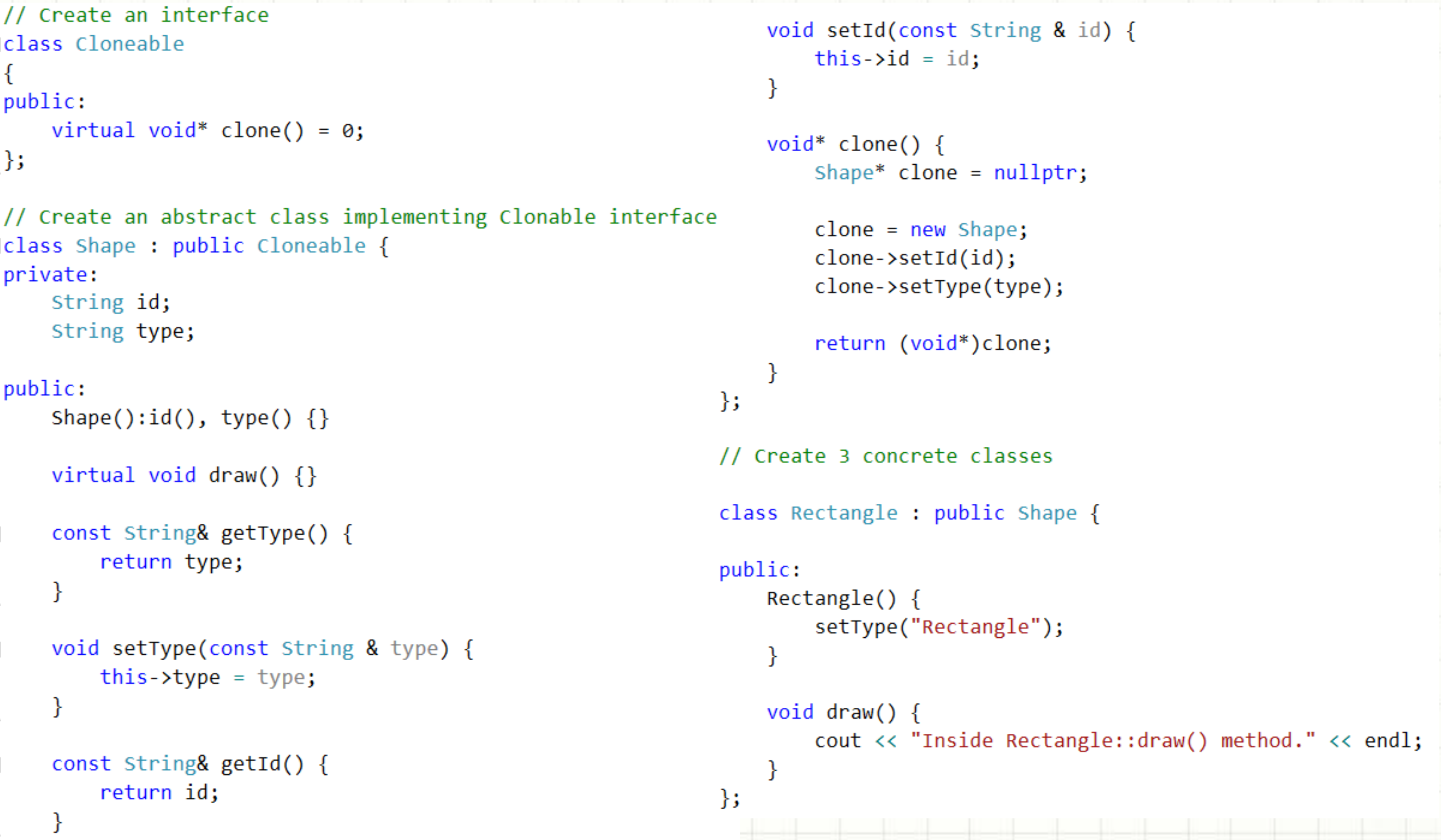
```
int main(int argc, char **argv)
{
    AbstractLogger* loggerChain = ChainPatternDemo::getChainOfLoggers();

    loggerChain->logMessage(AbstractLogger::INFO,
        "This is an information.");

    loggerChain->logMessage(AbstractLogger::DEBUG,
        "This is an debug level information.");

    loggerChain->logMessage(AbstractLogger::ERROR,
        "This is an error information.");

    delete loggerChain;
    return 0;
}

/*
Standard Console::Logger: This is an information.
File::Logger : This is an debug level information.
Standard Console::Logger : This is an debug level information.
Error Console::Logger : This is an error information.
File::Logger : This is an error information.
Standard Console::Logger : This is an error information.
*/
```

[Prototype (8/23)]

- creating a duplicated instance while keeping performance in mind (that you want to modify)
- type : creational pattern



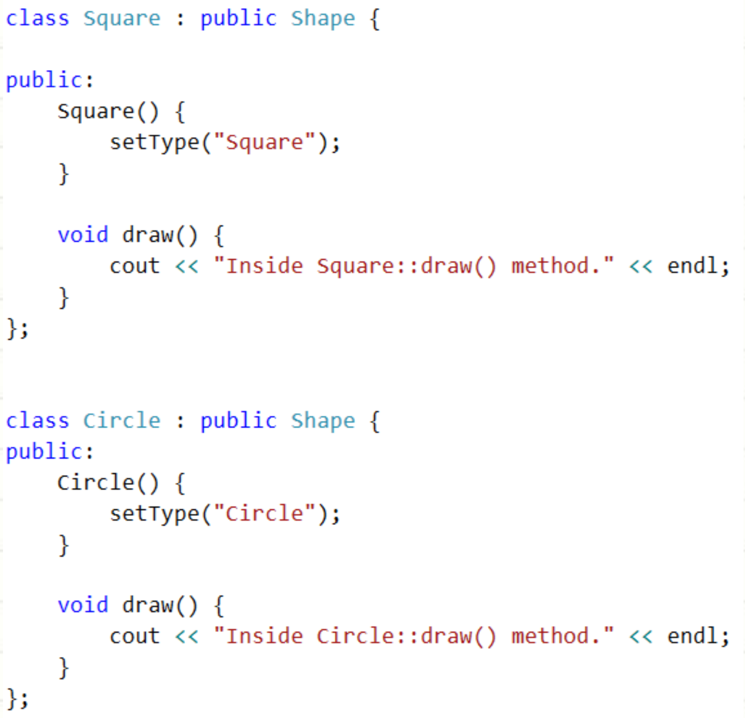
[Prototype (8/23)]



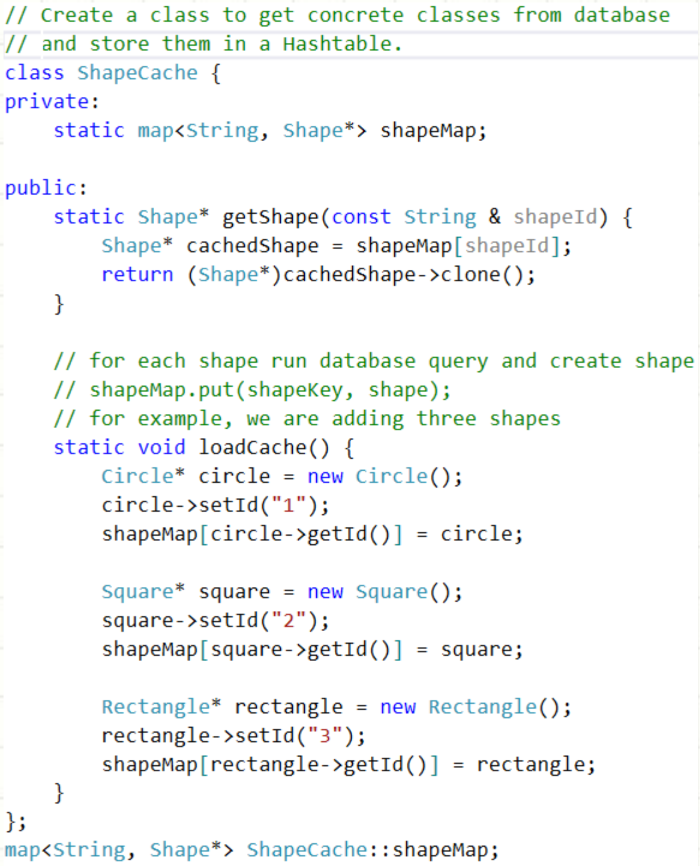
```
// Create an interface
class Cloneable
{
public:
    virtual void* clone() = 0;
};

// Create an abstract class implementing Clonable interface
class Shape : public Cloneable {
private:
    String id;
    String type;

public:
    Shape():id(), type() {}

    virtual void draw() {}

    const String& getType() {
        return type;
    }

    void setType(const String & type) {
        this->type = type;
    }

    const String& getId() {
        return id;
    }
```

```
    void setId(const String & id) {
        this->id = id;
    }

    void* clone() {
        Shape* clone = nullptr;

        clone = new Shape;
        clone->setId(id);
        clone->setType(type);

        return (void*)clone;
    }
};

// Create 3 concrete classes

class Rectangle : public Shape {

public:
    Rectangle() {
        setType("Rectangle");
    }

    void draw() {
        cout << "Inside Rectangle::draw() method." << endl;
    }
};
```

[Prototype (8/23)]

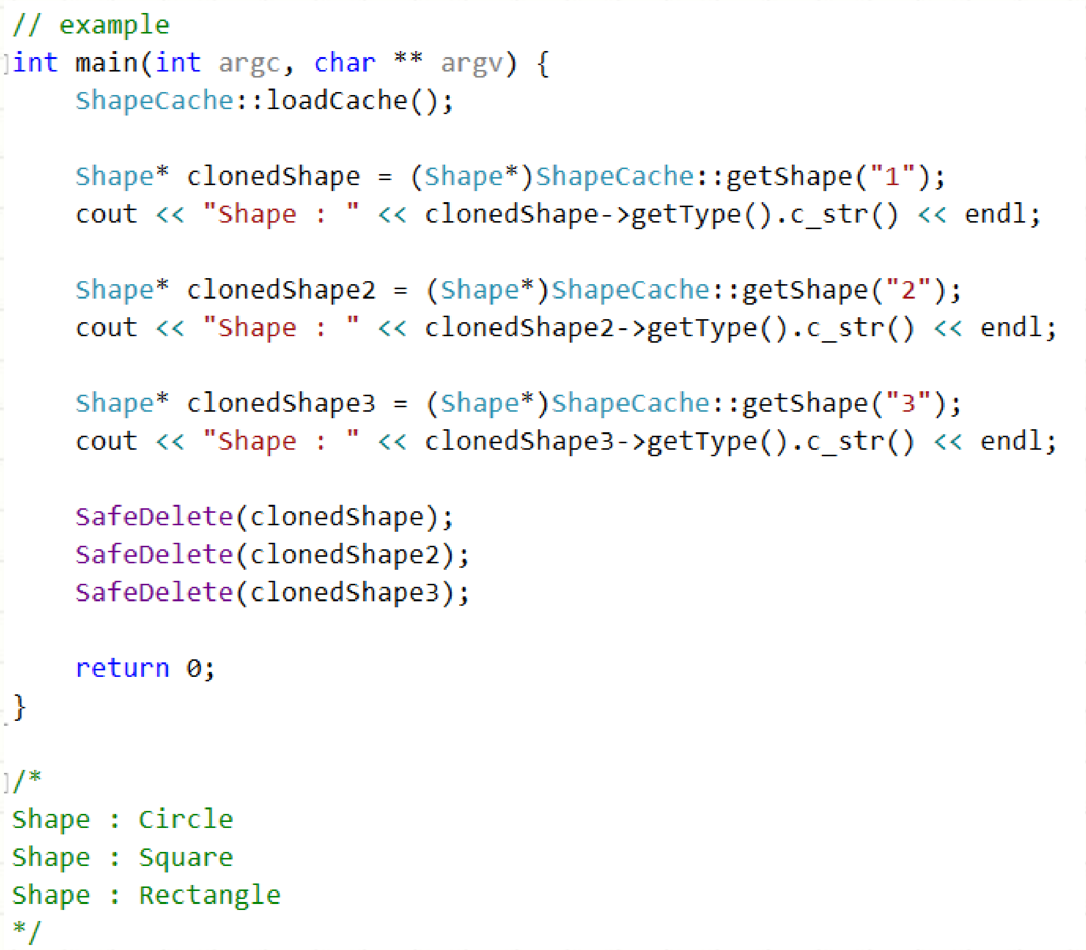


```
class Square : public Shape {

public:
    Square() {
        setType("Square");
    }

    void draw() {
        cout << "Inside Square::draw() method." << endl;
    }
};


class Circle : public Shape {
public:
    Circle() {
        setType("Circle");
    }

    void draw() {
        cout << "Inside Circle::draw() method." << endl;
    }
};
```



```
// Create a class to get concrete classes from database
// and store them in a Hashtable.
class ShapeCache {
private:
    static map<String, Shape*> shapeMap;

public:
    static Shape* getShape(const String & shapeId) {
        Shape* cachedShape = shapeMap[shapeId];
        return (Shape*)cachedShape->clone();
    }

    // for each shape run database query and create shape
    // shapeMap.put(shapeKey, shape);
    // for example, we are adding three shapes
    static void loadCache() {
        Circle* circle = new Circle();
        circle->setId("1");
        shapeMap[circle->getId()] = circle;

        Square* square = new Square();
        square->setId("2");
        shapeMap[square->getId()] = square;

        Rectangle* rectangle = new Rectangle();
        rectangle->setId("3");
        shapeMap[rectangle->getId()] = rectangle;
    }
};
map<String, Shape*> ShapeCache::shapeMap;
```

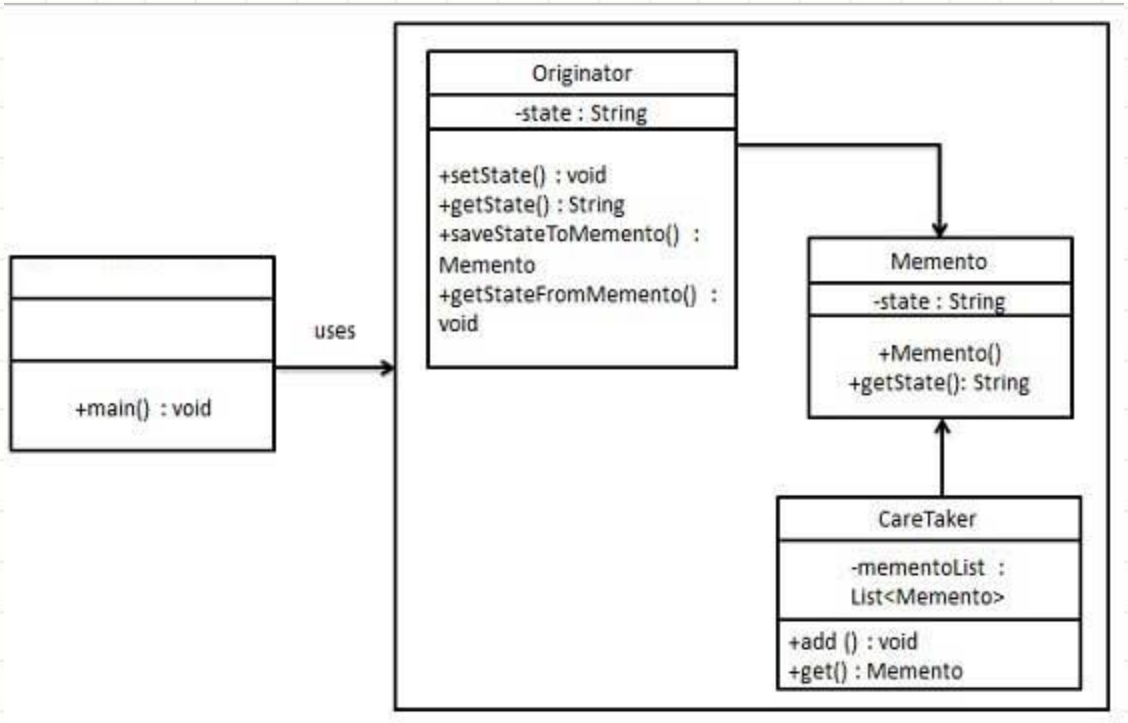
[Prototype (8/23)]



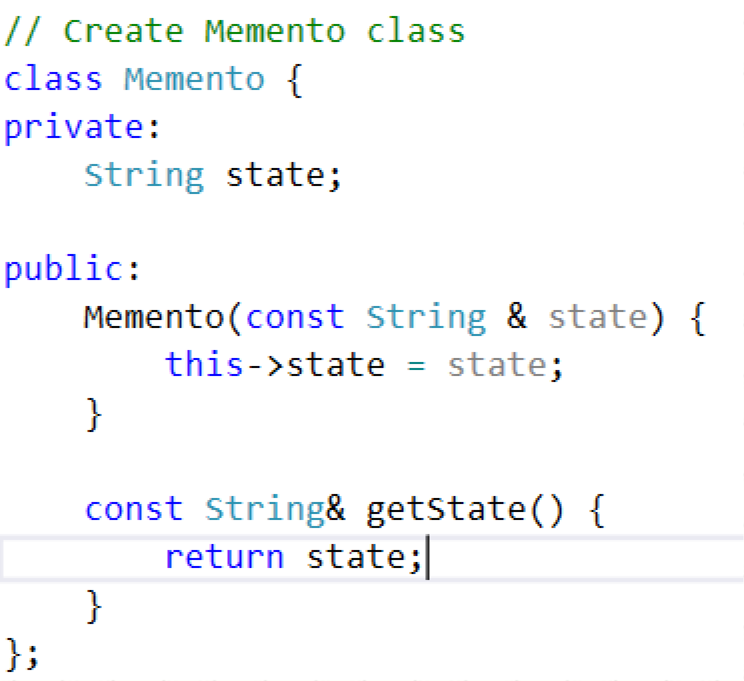
```cpp
// example
int main(int argc, char ** argv) {
    ShapeCache::loadCache();

    Shape* clonedShape = (Shape*)ShapeCache::getShape("1");
    cout << "Shape : " << clonedShape->getType().c_str() << endl;

    Shape* clonedShape2 = (Shape*)ShapeCache::getShape("2");
    cout << "Shape : " << clonedShape2->getType().c_str() << endl;

    Shape* clonedShape3 = (Shape*)ShapeCache::getShape("3");
    cout << "Shape : " << clonedShape3->getType().c_str() << endl;

    SafeDelete(clonedShape);
    SafeDelete(clonedShape2);
    SafeDelete(clonedShape3);

    return 0;
}

/*
Shape : Circle
Shape : Square
Shape : Rectangle
*/
```

[Memento (9/23)]

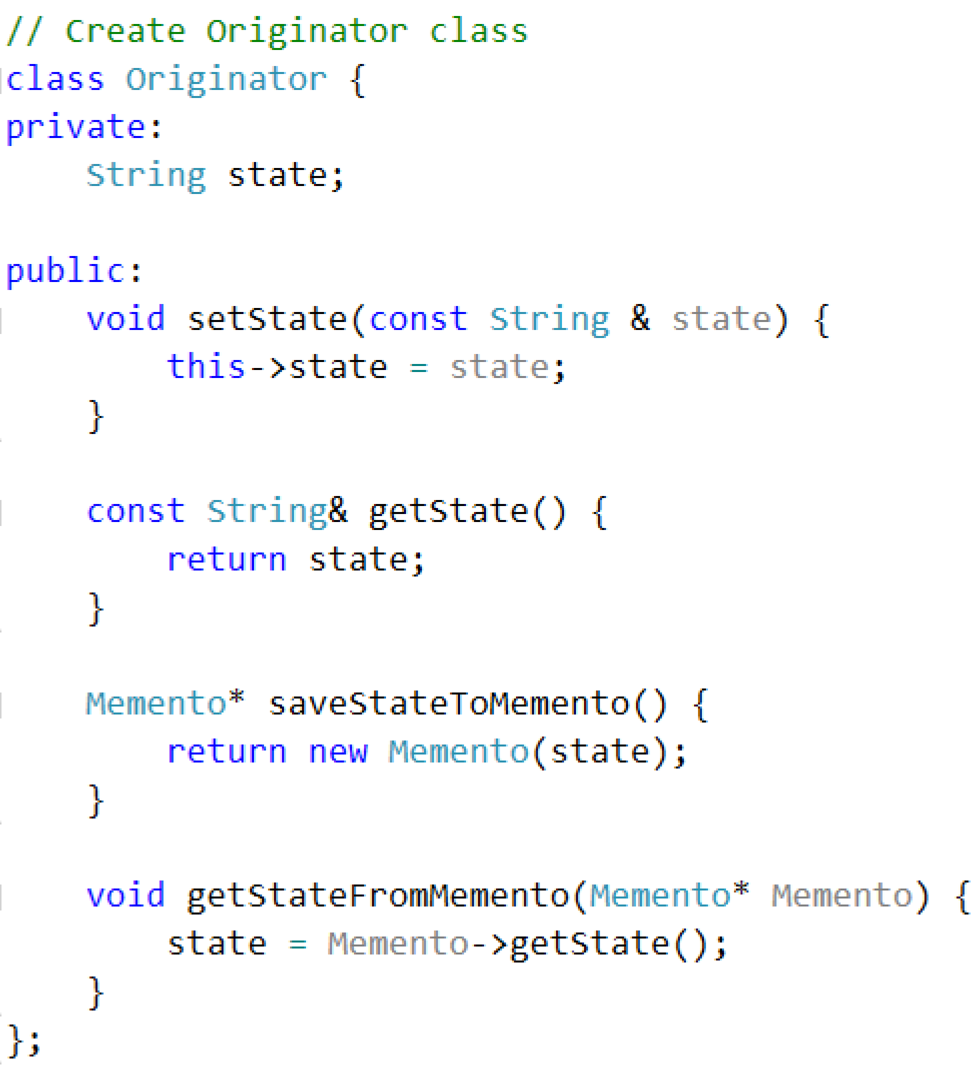
- used to restore state of an instance to a previous state (ex .undo/redo)
- type: behavioral pattern



[Memento (9/23)]

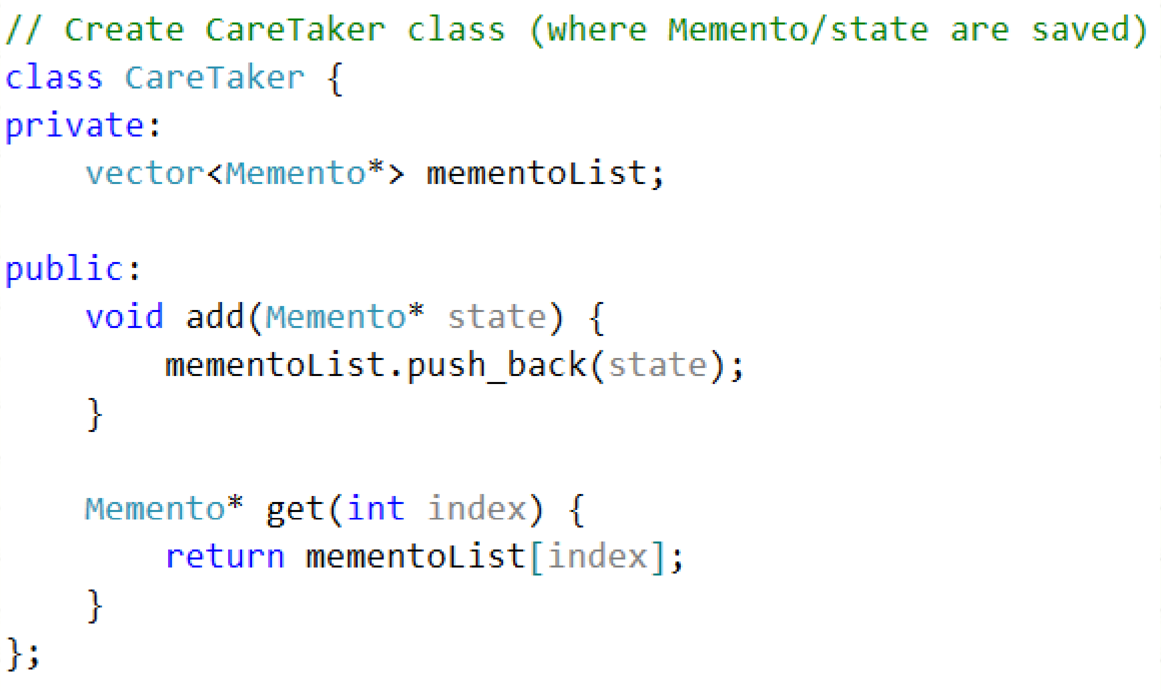


```
// Create Memento class
class Memento {
private:
    String state;

public:
    Memento(const String & state) {
        this->state = state;
    }

    const String& getState() {
        return state;
    }
};
```



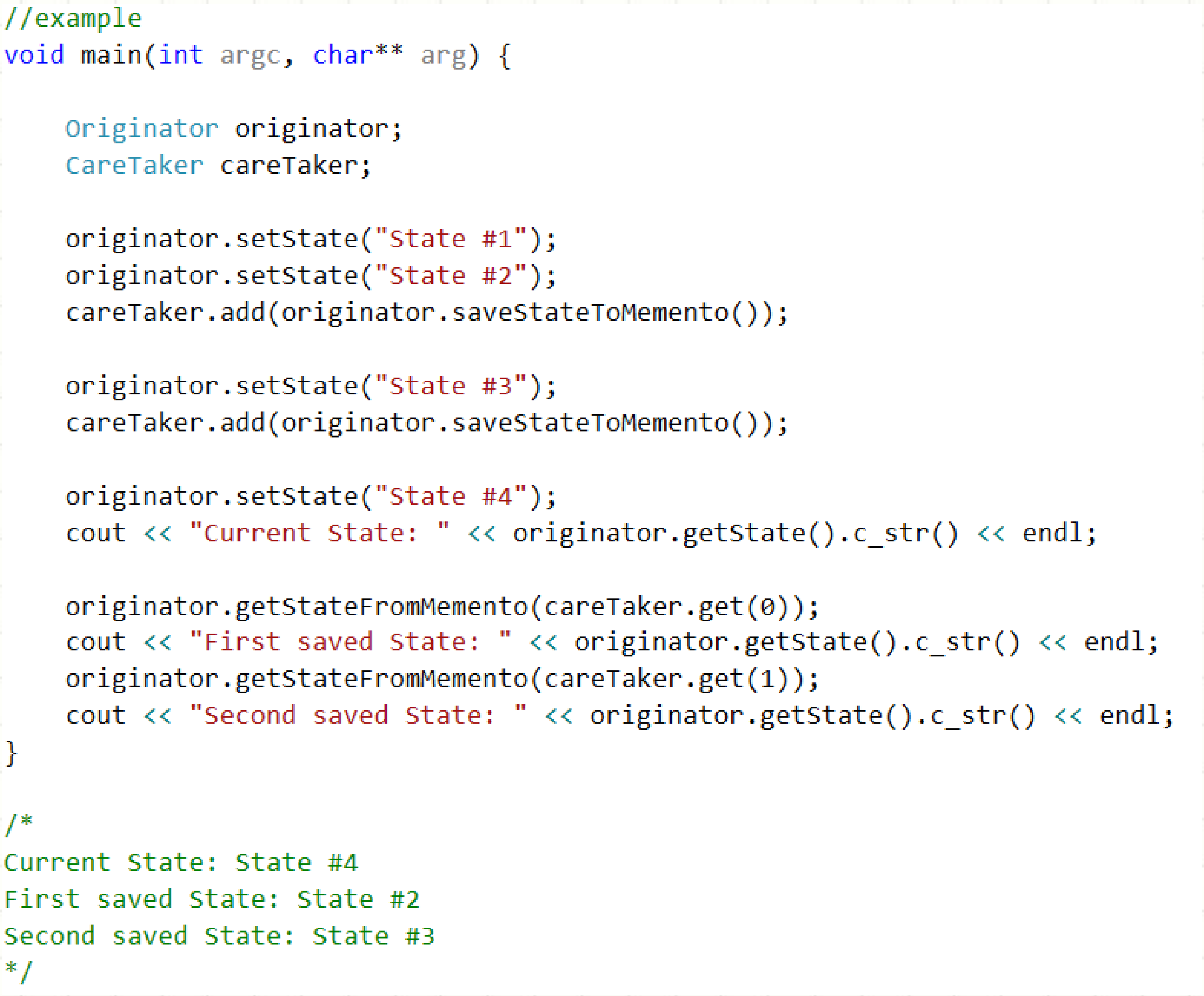
```
// Create Originator class
class Originator {
private:
    String state;

public:
    void setState(const String & state) {
        this->state = state;
    }

    const String& getState() {
        return state;
    }

    Memento* saveStateToMemento() {
        return new Memento(state);
    }

    void getStateFromMemento(Memento* Memento) {
        state = Memento->getState();
    }
};
```

[Memento (9/23)]



```
// Create CareTaker class (where Memento/state are saved)
class CareTaker {
private:
    vector<Memento*> mementoList;

public:
    void add(Memento* state) {
        mementoList.push_back(state);
    }

    Memento* get(int index) {
        return mementoList[index];
    }
};
```

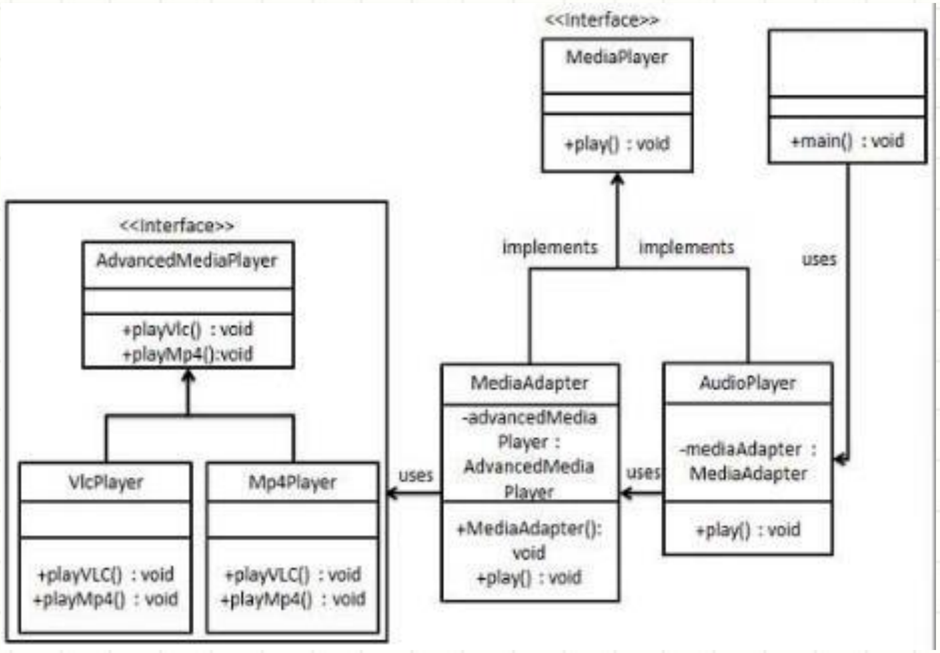
[Memento (9/23)]



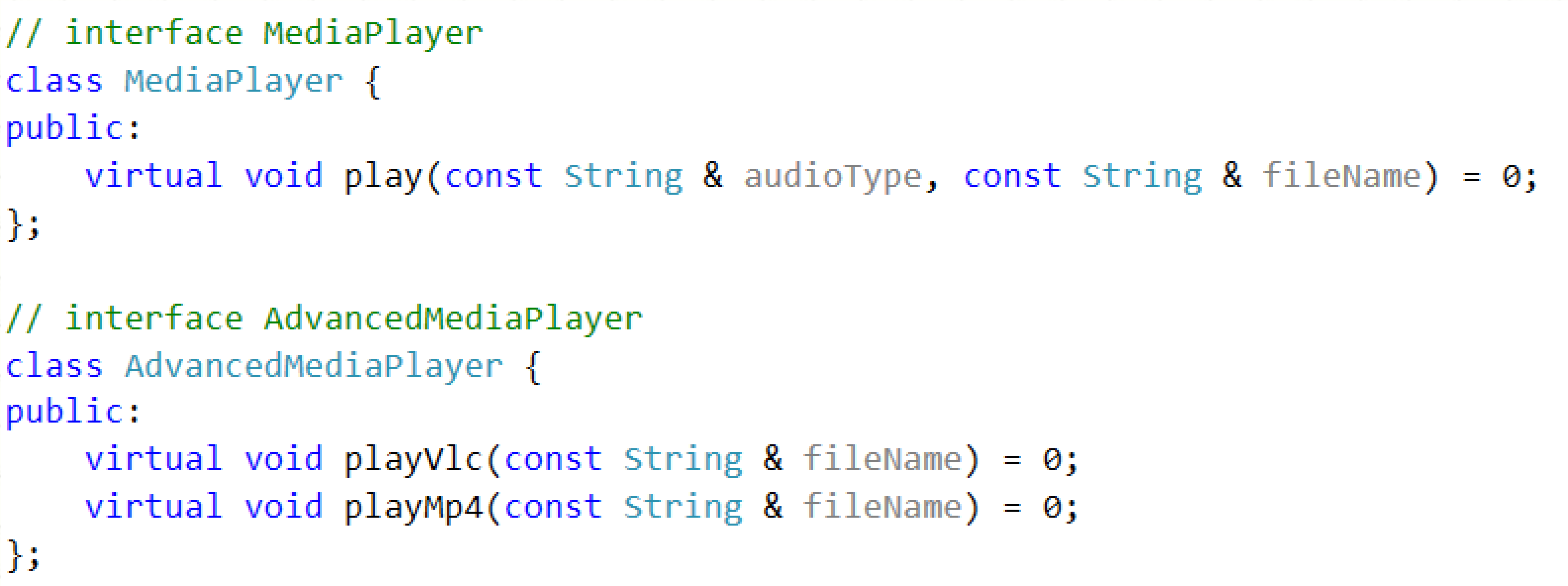
```
//example
void main(int argc, char** arg) {

    Originator originator;
    CareTaker careTaker;

    originator.setState("State #1");
    originator.setState("State #2");
    careTaker.add(originator.saveStateToMemento());

    originator.setState("State #3");
    careTaker.add(originator.saveStateToMemento());

    originator.setState("State #4");
    cout << "Current State: " << originator.getState().c_str() << endl;

    originator.getStateFromMemento(careTaker.get(0));
    cout << "First saved State: " << originator.getState().c_str() << endl;
    originator.getStateFromMemento(careTaker.get(1));
    cout << "Second saved State: " << originator.getState().c_str() << endl;
}

/*
Current State: State #4
First saved State: State #2
Second saved State: State #3
*/
```

[Adapter (10/23)]

- involves a single class which is responsible to join functionalities of incompatible interfaces
- type : structural pattern

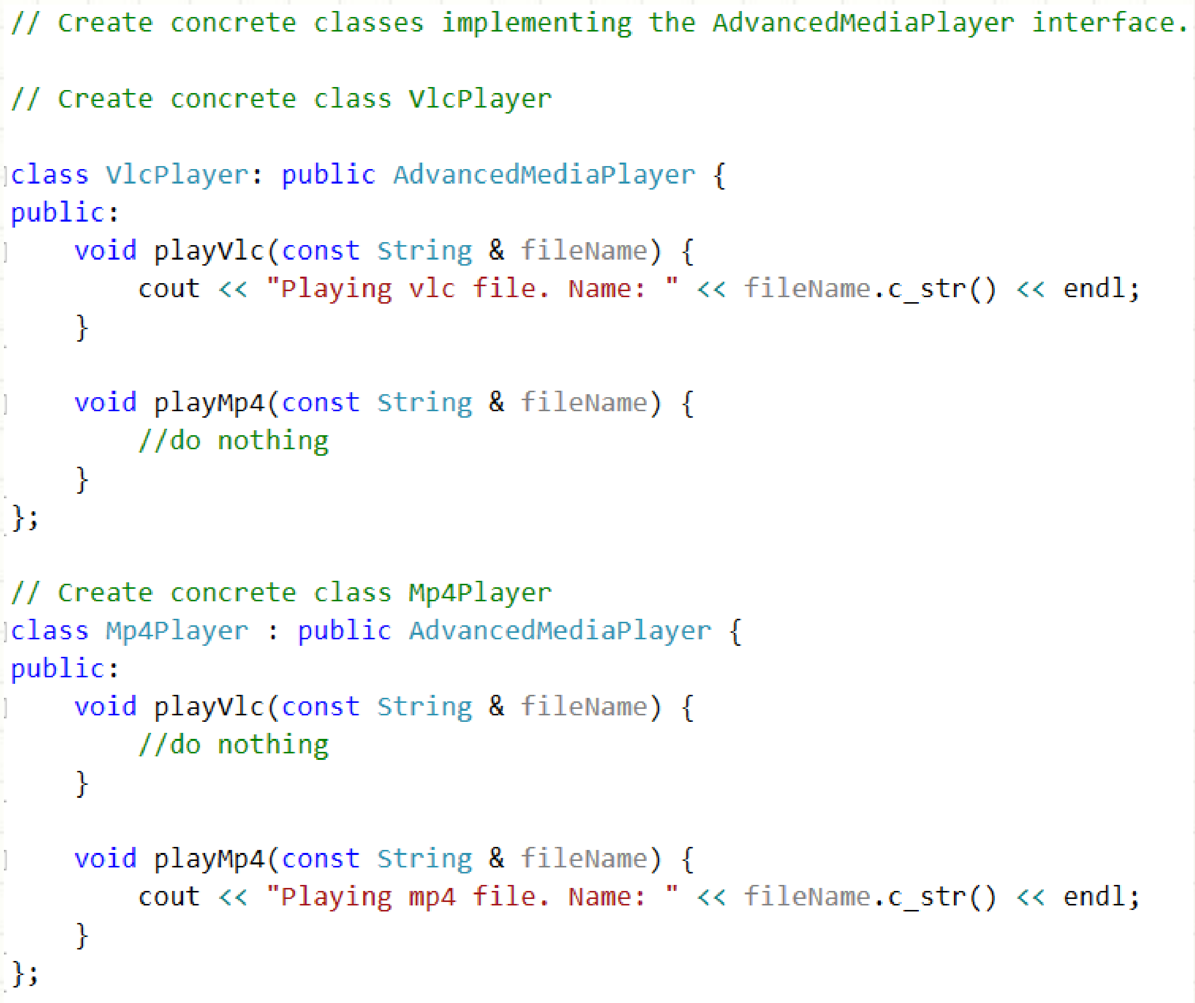


[Adapter (10/23)]



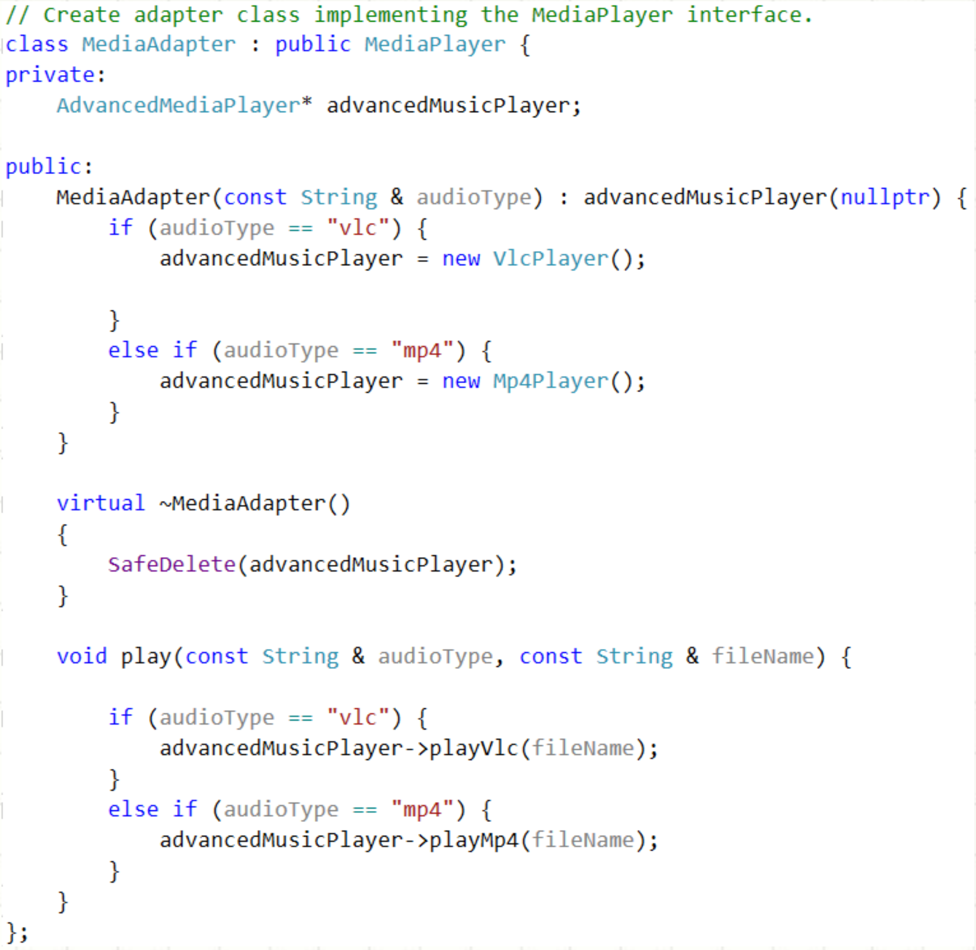
```
// interface MediaPlayer
class MediaPlayer {
public:
    virtual void play(const String & audioType, const String & fileName) = 0;
};

// interface AdvancedMediaPlayer
class AdvancedMediaPlayer {
public:
    virtual void playVlc(const String & fileName) = 0;
    virtual void playMp4(const String & fileName) = 0;
};
```

[Adapter (10/23)]



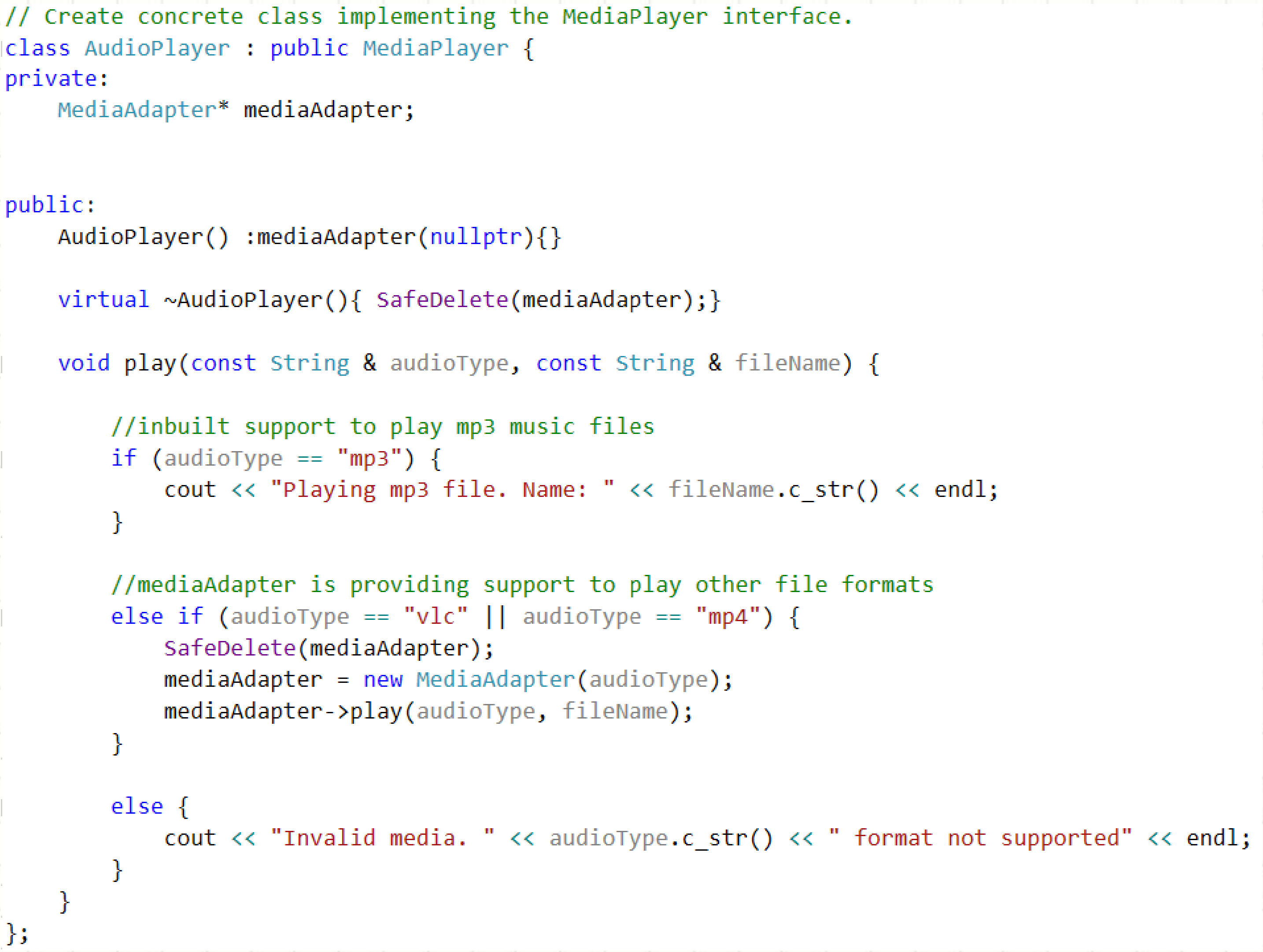
```
// Create concrete classes implementing the AdvancedMediaPlayer interface.

// Create concrete class VlcPlayer

class VlcPlayer: public AdvancedMediaPlayer {
public:
    void playVlc(const String & fileName) {
        cout << "Playing vlc file. Name: " << fileName.c_str() << endl;
    }

    void playMp4(const String & fileName) {
        //do nothing
    }
};

// Create concrete class Mp4Player
class Mp4Player : public AdvancedMediaPlayer {
public:
    void playVlc(const String & fileName) {
        //do nothing
    }

    void playMp4(const String & fileName) {
        cout << "Playing mp4 file. Name: " << fileName.c_str() << endl;
    }
};
```

[Adapter (10/23)]



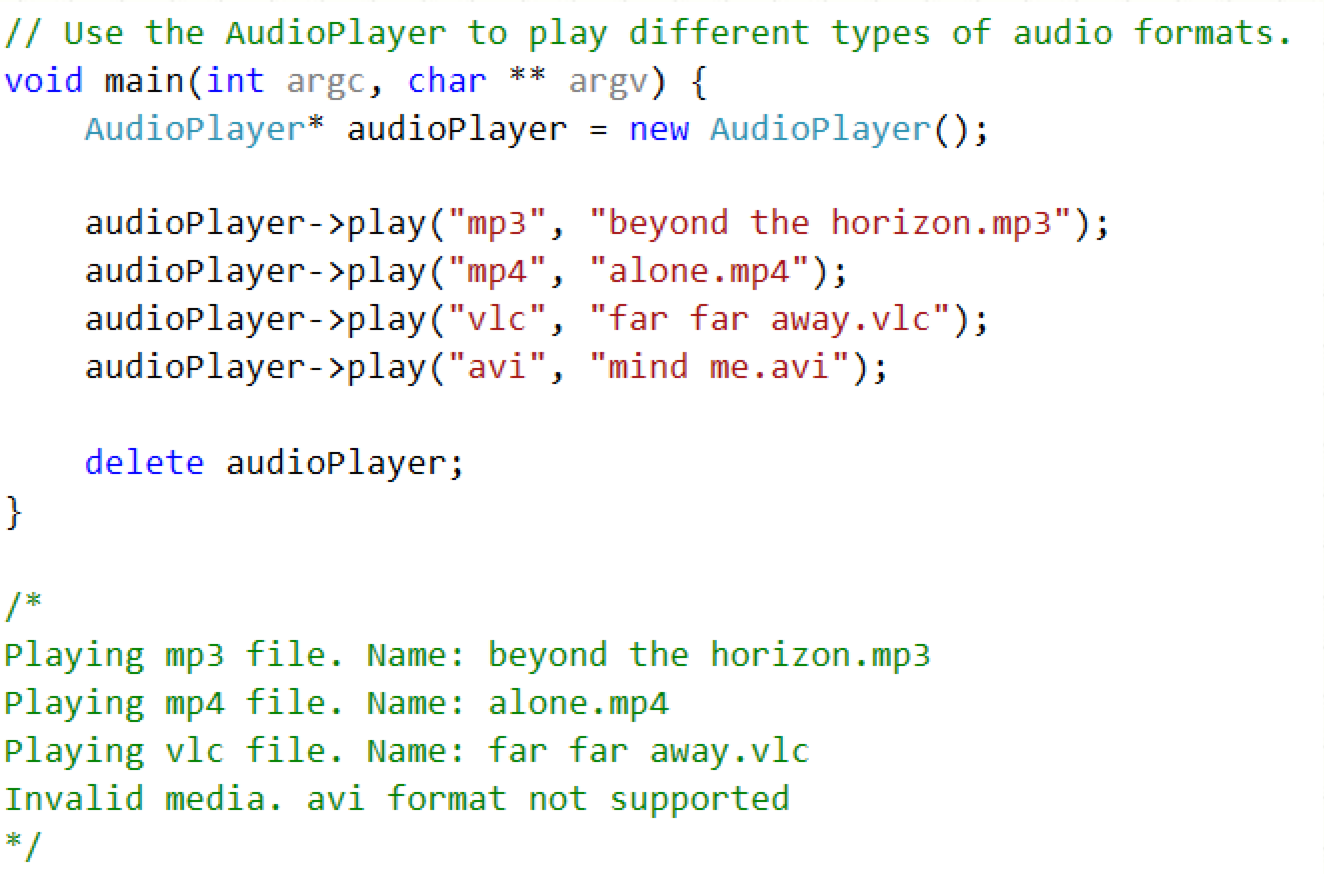
```cpp
// Create adapter class implementing the MediaPlayer interface.
class MediaAdapter : public MediaPlayer {
private:
    AdvancedMediaPlayer* advancedMusicPlayer;

public:
    MediaAdapter(const String & audioType) : advancedMusicPlayer(nullptr) {
        if (audioType == "vlc") {
            advancedMusicPlayer = new VlcPlayer();

        }
        else if (audioType == "mp4") {
            advancedMusicPlayer = new Mp4Player();
        }
    }

    virtual ~MediaAdapter()
    {
        SafeDelete(advancedMusicPlayer);
    }

    void play(const String & audioType, const String & fileName) {

        if (audioType == "vlc") {
            advancedMusicPlayer->playVlc(fileName);
        }
        else if (audioType == "mp4") {
            advancedMusicPlayer->playMp4(fileName);
        }
    }
};
```

[Adapter (10/23)]



```
// Create concrete class implementing the MediaPlayer interface.
class AudioPlayer : public MediaPlayer {
private:
    MediaAdapter* mediaAdapter;


public:
    AudioPlayer() :mediaAdapter(nullptr){}

    virtual ~AudioPlayer(){ SafeDelete(mediaAdapter);}

    void play(const String & audioType, const String & fileName) {

        //inbuilt support to play mp3 music files
        if (audioType == "mp3") {
            cout << "Playing mp3 file. Name: " << fileName.c_str() << endl;
        }

        //mediaAdapter is providing support to play other file formats
        else if (audioType == "vlc" || audioType == "mp4") {
            SafeDelete(mediaAdapter);
            mediaAdapter = new MediaAdapter(audioType);
            mediaAdapter->play(audioType, fileName);
        }

        else {
            cout << "Invalid media. " << audioType.c_str() << " format not supported" << endl;
        }
    }
};
```

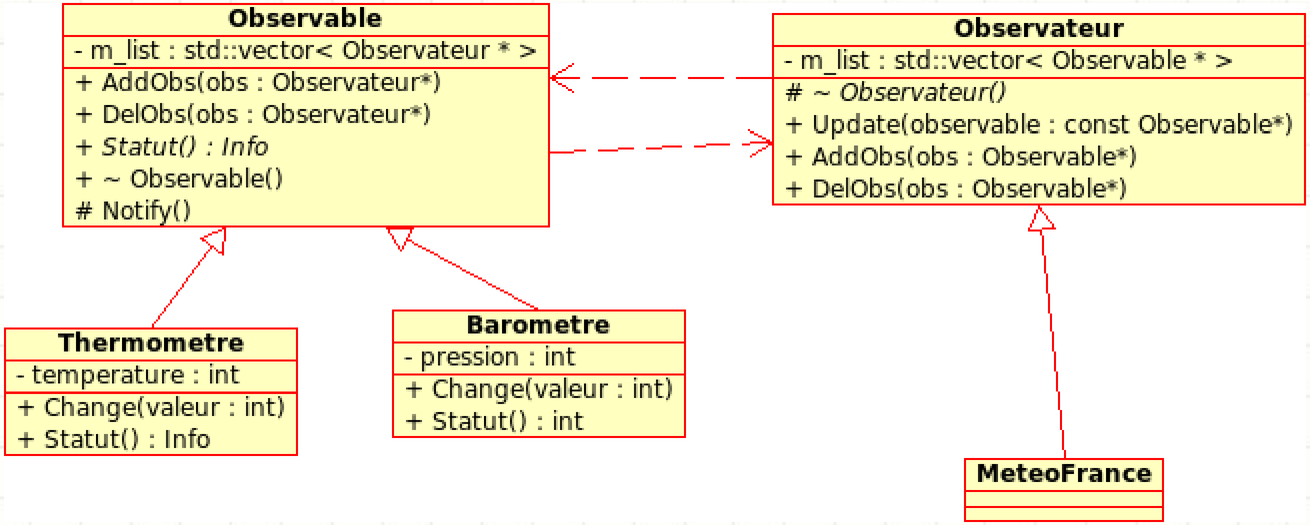
[Adapter (10/23)]



```
// Use the AudioPlayer to play different types of audio formats.
void main(int argc, char ** argv) {
    AudioPlayer* audioPlayer = new AudioPlayer();

    audioPlayer->play("mp3", "beyond the horizon.mp3");
    audioPlayer->play("mp4", "alone.mp4");
    audioPlayer->play("vlc", "far far away.vlc");
    audioPlayer->play("avi", "mind me.avi");

    delete audioPlayer;
}

/*
Playing mp3 file. Name: beyond the horizon.mp3
Playing mp4 file. Name: alone.mp4
Playing vlc file. Name: far far away.vlc
Invalid media. avi format not supported
*/
```

[Observer (11/23)]

- When a instance is modified, its observer(s) receive(s) a notification
- type: behavioral pattern



Observable

- m_list : std::vector< Observateur * >

+ AddObs(obs : Observateur*)

+ DelObs(obs : Observateur*)

+ *Statut() : Info*

+ ~ Observable()

Notify()

Observateur

- m_list : std::vector< Observable * >

~ *Observateur()*

+ Update(observable : const Observable*)

+ AddObs(obs : Observable*)

+ DelObs(obs : Observable*)

Thermometre

- temperature : int

+ Change(valeur : int)

+ Statut() : Info

Barometre

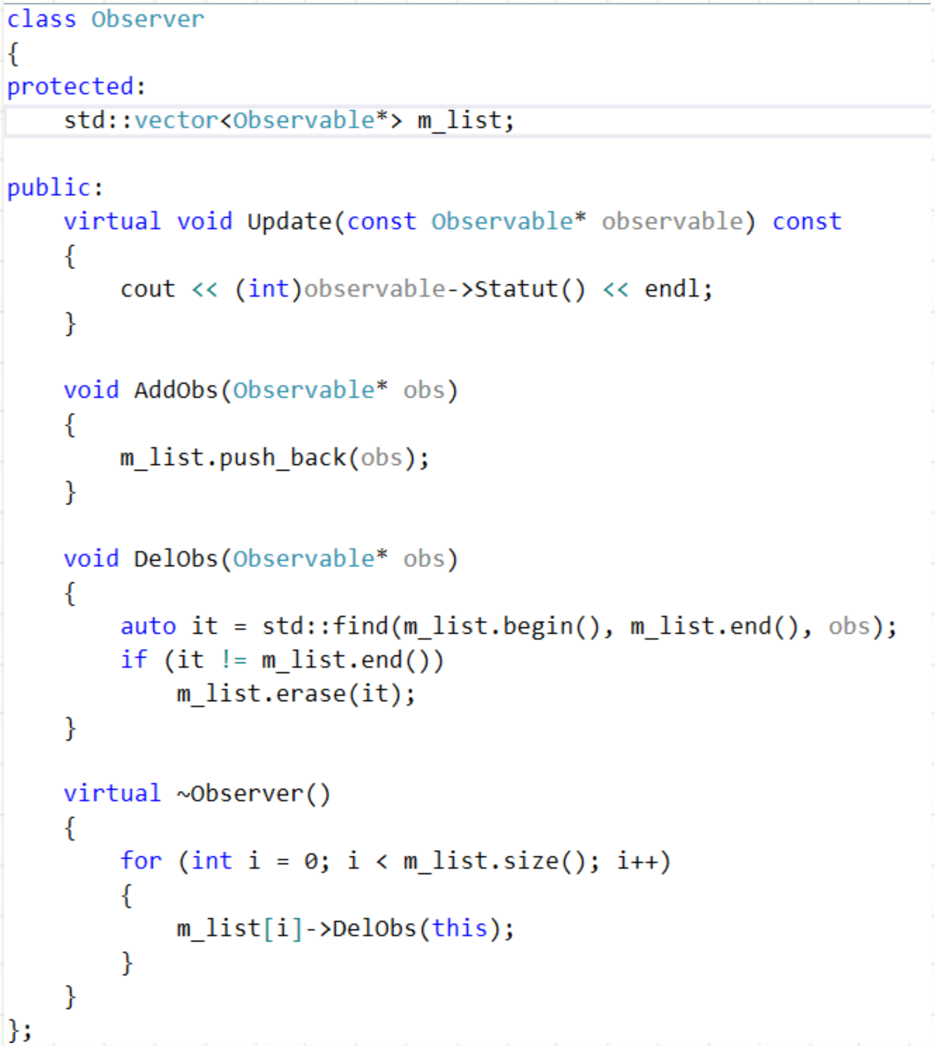
- pression : int

+ Change(valeur : int)

+ Statut() : int

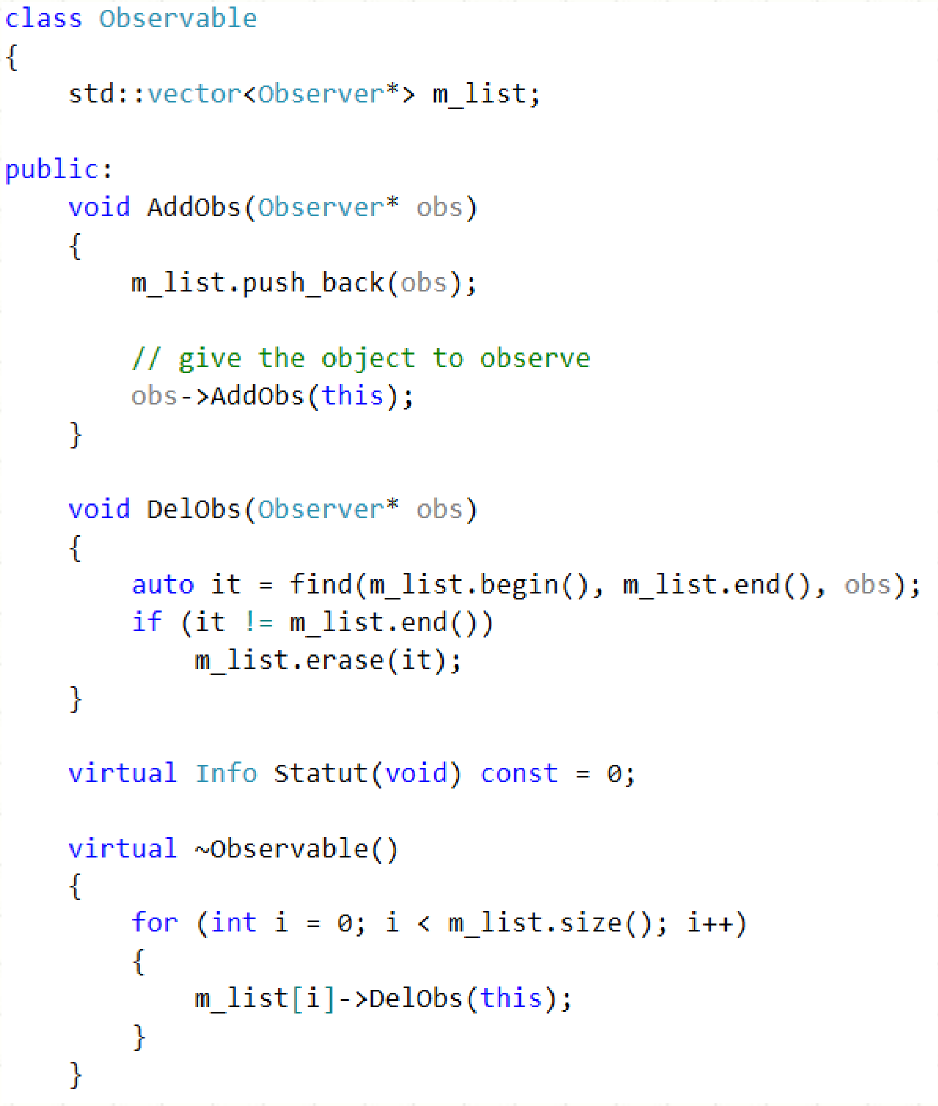
MeteoFrance

[Observer (11/23)]



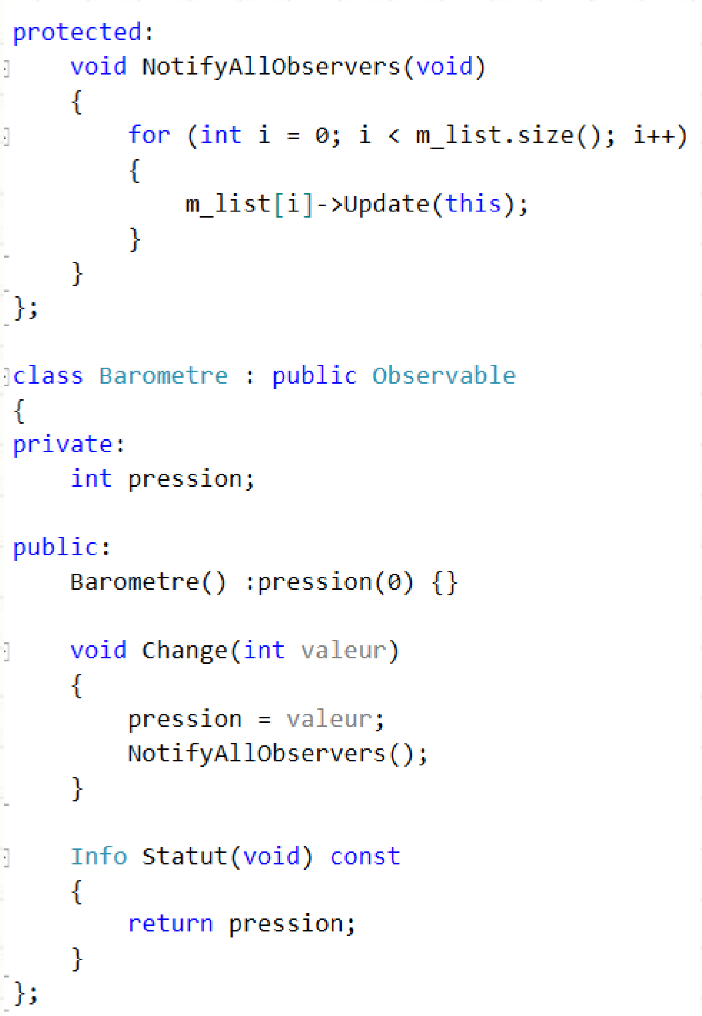
```
class Observer
{
protected:
    std::vector<Observable*> m_list;

public:
    virtual void Update(const Observable* observable) const
    {
        cout << (int)observable->Statut() << endl;
    }

    void AddObs(Observable* obs)
    {
        m_list.push_back(obs);
    }

    void DelObs(Observable* obs)
    {
        auto it = std::find(m_list.begin(), m_list.end(), obs);
        if (it != m_list.end())
            m_list.erase(it);
    }

    virtual ~Observer()
    {
        for (int i = 0; i < m_list.size(); i++)
        {
            m_list[i]->DelObs(this);
        }
    }
};
```

[Observer (11/23)]

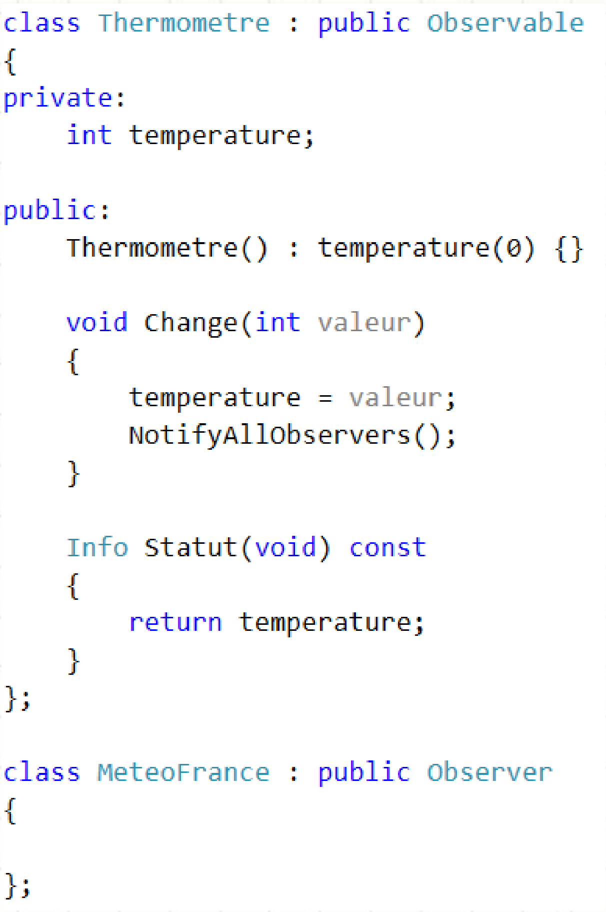


```
class Observable
{
    std::vector<Observer*> m_list;

public:
    void AddObs(Observer* obs)
    {
        m_list.push_back(obs);

        // give the object to observe
        obs->AddObs(this);
    }

    void DelObs(Observer* obs)
    {
        auto it = find(m_list.begin(), m_list.end(), obs);
        if (it != m_list.end())
            m_list.erase(it);
    }

    virtual Info Statut(void) const = 0;

    virtual ~Observable()
    {
        for (int i = 0; i < m_list.size(); i++)
        {
            m_list[i]->DelObs(this);
        }
    }
```

[Observer (11/23)]

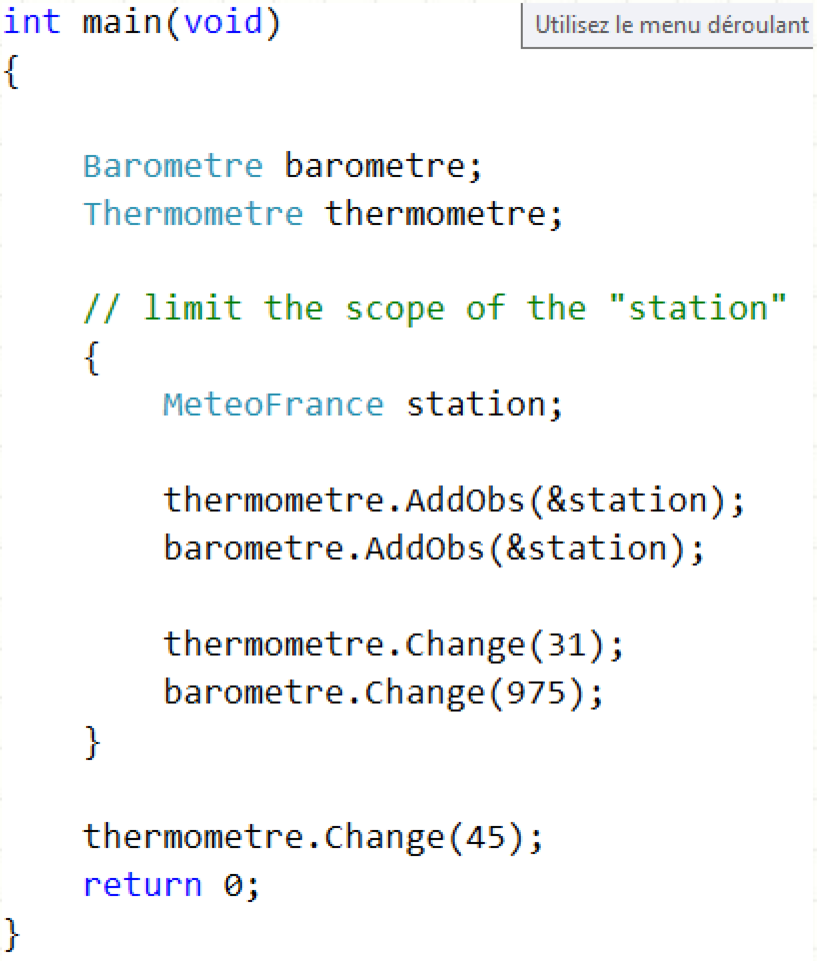


```
protected:
    void NotifyAllObservers(void)
    {
        for (int i = 0; i < m_list.size(); i++)
        {
            m_list[i]->Update(this);
        }
    }
};

class Barometre : public Observable
{
private:
    int pression;

public:
    Barometre() :pression(0) {}

    void Change(int valeur)
    {
        pression = valeur;
        NotifyAllObservers();
    }

    Info Statut(void) const
    {
        return pression;
    }
};
```



```
class Thermometre : public Observable
{
private:
    int temperature;

public:
    Thermometre() : temperature(0) {}

    void Change(int valeur)
    {
        temperature = valeur;
        NotifyAllObservers();
    }

    Info Statut(void) const
    {
        return temperature;
    }
};

class MeteoFrance : public Observer
{

};
```

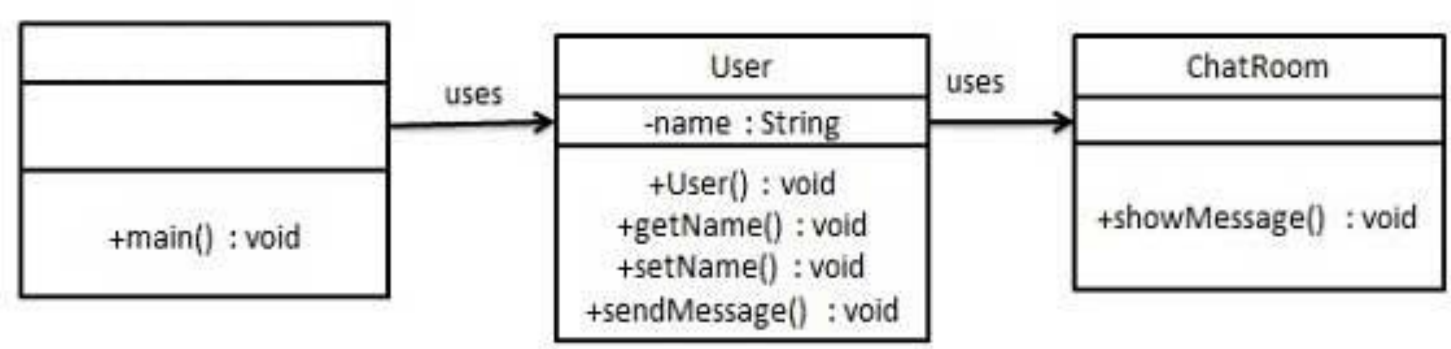
[Observer (11/23)]



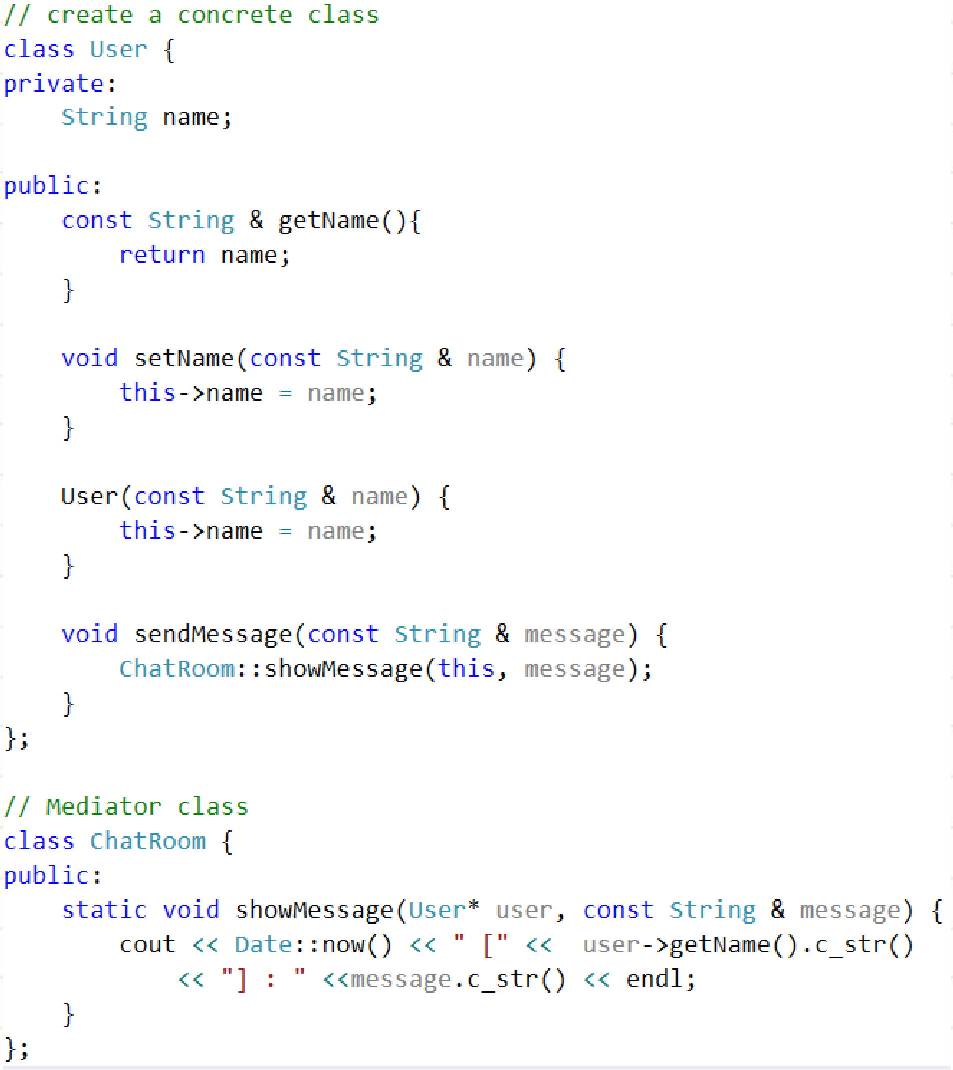
```
int main(void)
{

    Barometre barometre;
    Thermometre thermometre;

    // limit the scope of the "station"
    {
        MeteoFrance station;

        thermometre.AddObs(&station);
        barometre.AddObs(&station);

        thermometre.Change(31);
        barometre.Change(975);
    }

    thermometre.Change(45);
    return 0;
}
```

[Mediator (12/23)]

- reduce communication complexity between multiple objects or classes
- type: behavioral pattern

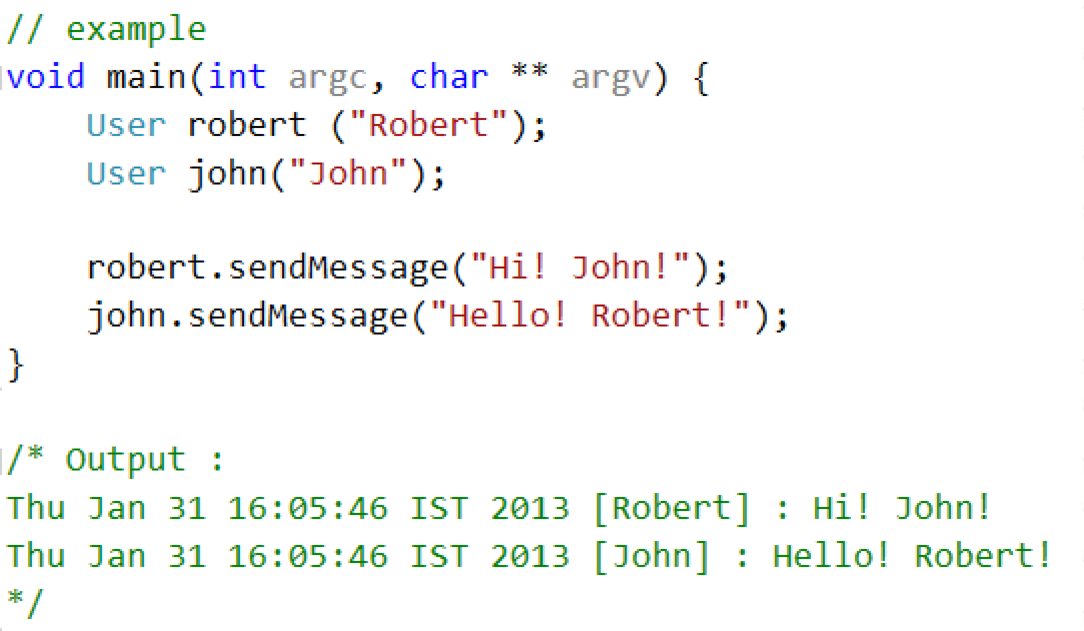


[Mediator (12/23)]



```
// create a concrete class
class User {
private:
    String name;

public:
    const String & getName(){
        return name;
    }

    void setName(const String & name) {
        this->name = name;
    }

    User(const String & name) {
        this->name = name;
    }

    void sendMessage(const String & message) {
        ChatRoom::showMessage(this, message);
    }
};

// Mediator class
class ChatRoom {
public:
    static void showMessage(User* user, const String & message) {
        cout << Date::now() << " [" <<  user->getName().c_str()
            << "] : " <<message.c_str() << endl;
    }
};
```

[Mediator (12/23)]



```
// example
void main(int argc, char ** argv) {
    User robert ("Robert");
    User john("John");

    robert.sendMessage("Hi! John!");
    john.sendMessage("Hello! Robert!");
}

/* Output :
Thu Jan 31 16:05:46 IST 2013 [Robert] : Hi! John!
Thu Jan 31 16:05:46 IST 2013 [John] : Hello! Robert!
*/
```

CONCLUSION

Ressources

- https://sourcemaking.com/design_patterns
- https://gist.github.com/pazdera/
- http://come-david.developpez.com/tutoriels/dps/
- https://www.tutorialspoint.com/