C++ Design Pattern

Qing Huang

Topics

- Memento
  - Static memento
  - Dynamic memento
- Mediator
  - Chat room
  - Event Broker
- Observer
  - Notifier and Listener
  - Multithread and mutex

```cpp
class Memento
{
  int balance;
public:
  Memento(int balance)
    : balance(balance)
  {
  }
  friend class BankAccount;
  friend class BankAccount2;
};

class BankAccount
{
  int balance = 0;
public:
  explicit BankAccount(const int balance)
    : balance(balance)
  {
  }

  Memento deposit(int amount)
  {
    balance += amount;
    return { balance };
  }

  void restore(const Memento& m)
  {
    balance = m.balance;
  }

  friend ostream& operator<<(ostream& os, const BankAccour
  {
    return os << "balance: " << obj.balance;
  }
};
```

Use friend class to set valiue

```cpp
class BankAccount2
{
  int balance = 0;
  vector<shared_ptr<Memento>> changes;
  int current;
public:
  explicit BankAccount2(const int balance)
  : balance(balance)
  {
    changes.emplace_back(make_shared<Memento>(balance));
    current = 0;
  }

  shared_ptr<Memento> deposit(int amount)
  {
    balance += amount;
    auto m = make_shared<Memento>(balance);
    changes.push_back(m);
    ++current;
    return m;
  }
  shared_ptr<Memento> undo()
  {
    if (current > 0)
    {
      --current;
      auto m = changes[current];
      balance = m->balance;
      return m;
    }
    return{};
  }

  shared_ptr<Memento> redo()
  {
    if (current + 1 < changes.size())
    {
      ++current;
      auto m = changes[current];
      balance = m->balance;
      return m;
    }
    return{};
  }
```

Use vector to store value

```cpp
#include<iostream>

using namespace std;

struct Buffer
{
    bool isIndent = false;
    struct Indent{
        Buffer& buffer;
        Indent(Buffer & buff) :buffer(buff){
            buffer.isIndent = true;
        }
        ~Indent(){
            buffer.isIndent = false;
        }
    };
    friend ostream& operator<<(ostream& os, const Buffer& obj)
    {
        if(obj.isIndent)
            return os <<"        " << "hello world";
        else
            return os << "hello world";
    }
};
int main()
{
    Buffer test;
    {
        Buffer::Indent indent(test);
        cout << test << endl;
    }
    cout <<  test << endl;
}
```

```
~/Documents/workspace/design_pattern/design-patterns-in
$ g++ automatic.cpp -o test.o -std=c++11

~/Documents/workspace/design_pattern/design-patterns-in
$ ./test.o
        hello world
hello world

~/Documents/workspace/design_pattern/design-patterns-in
$
```

Memento Exercise

```cpp
#include <vector>
#include <memory>
using namespace std;

struct Token
{
  int value;

  Token(int value) : value(value) {}
};

struct Memento
{
  vector<shared_ptr<Token>> tokens;
  int currIndex = 0;
};

struct TokenMachine
{
  vector<shared_ptr<Token>> tokens;


  Memento add_token(int value)
  {
    return add_token(make_shared<Token>(value));
  }

  // adds the token to the set of tokens and returns the
  // snapshot of the entire system
Memento add_token(const shared_ptr<Token>& token)
  {
    tokens.push_back(token);
    Memento m;
    for (auto t : tokens)
      m.tokens.emplace_back(make_shared<Token>(t->value));
    return m;
  }

  void revert(const Memento& m)
  {
    tokens.clear();
    for (auto t : m.tokens)
      tokens.emplace_back(make_shared<Token>(t->value));
  }
```

Mediator

```cpp
#pragma once

struct ChatRoom
{
  vector<Person*> people; // assume append-only

  void join(Person* p);
  void broadcast(const string& origin,
                 const string& message);
  void message(const string& origin,
               const string& who,
               const string& message);
};
```

```cpp
struct ChatRoom;

struct Person
{
  string name;
  ChatRoom* room = nullptr;

  Person(const string& name);
  void receive(const string& origin, const string& message);

  void say(const string& message) const;
  vector<string> chat_log;

  void pm(const string& who, const string& message) const;

  // generated in IDE
  friend bool operator==(const Person& lhs, const Person& rhs)
  {
    return lhs.name == rhs.name;
  }
```

```cpp
#include "person.h"
#include "chatroom.h"
#include <algorithm>

void ChatRoom::broadcast(const string& origin,
                         const string& message)
{
  for (auto p : people)
    if (p->name != origin)
      p->receive(origin, message);
}

void ChatRoom::join(Person* p)
{
  string join_msg = p->name + " joins the chat";
  broadcast("room", join_msg);

  p->room = this;
  people.push_back(p);
}
```

```cpp
void ChatRoom::message(const string& origin,
       const string& who, const string& message)
{
  auto target = find_if(begin(people),
                        end(people),
                        [&](const Person* p){
                            return p->name == who;
                            });
  if (target != end(people))
  {
    (*target)->receive(origin, message);
  }
}
```

```
include <iostream>
include <string>
include <vector>
truct Game;
sing namespace std;

include <boost/signals2.hpp>
sing namespace boost::signals2;

truct EventData

 virtual ~EventData() = default;
 virtual void print() const = 0;
;

truct Player;
truct PlayerScoredData : EventData

 string player_name;
int goals_scored_so_far;

PlayerScoredData(const string& player_name,
                 const int goals_scored_so_far)
  : player_name(player_name),
    goals_scored_so_far(goals_scored_so_far)
{
}

void print() const override
{
  cout << player_name << " has scored! (their "
    << goals_scored_so_far << " goal)" << "\n";
}
;
```

```
struct Game
{
  signal<void(EventData*)> events; // observer
};

struct Player
{
  string name;
  int goals_scored = 0;
  Game& game;


  Player(const string& name, Game& game)
    : name(name),
      game(game)
  {
  }

  void score()
  {
    goals_scored++;
    PlayerScoredData ps{name, goals_scored};
    game.events(&ps);
  }
};

struct Coach
{
  Game& game;

  explicit Coach(Game& game)
    : game(game)
  {
    // celebrate if player has scored <3 goals
    game.events.connect([](EventData* e)
    {
      PlayerScoredData* ps = dynamic_cast<PlayerScoredData*>(e);
      if (ps && ps->goals_scored_so_far < 3)
      {
        cout << "coach says: well done, " << ps->player_name << "\n"
      }
    });
  }
};
```

Mediator exercise

```cpp
#include<vector>
using namespace std;
struct IParticipant
{};
struct Participant;

struct Mediator{
    vector<Participant*> participants;
    inline void broadcast(int add_value, Participant* initiator);
};
struct Participant : IParticipant
{
    int value{0};
    Mediator& mediator;

    Participant(Mediator &mediator) : mediator(mediator)
    {
      mediator.participants.push_back(this);
    }

    void say(int value)
    {
        mediator.broadcast(value, this);
    }
};
inline void Mediator::broadcast(int add_value, Participant* initiator)
{
    for(auto participant:participants)
        if(participant!=initiator)
            participant->value += add_value;
}
```

Observer

Interface example:

```
class INotifier
{
public:
    class IListener{
        virtual void onCall(string& buffer) = 0;
    };
    virtual subscribe(IListener* IListener) = 0;
}
```

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

class Notifier
{
public:
    class Listener;
    Notifier() {};
    virtual void subscribe(Listener *listener)
    {
        listeners.push_back(listener);
    }
    void publishMessage(string message){
        cout << "publsihing  message " << endl;
        for(auto listener:listeners)
            listener->onCall(message);
    }
    int getNewId()
    {
        return listeners.size();
    }
    class Listener
    {
    public:
        Listener(Notifier& notifier) {
            id = notifier.getNewId();
            notifier.subscribe(this);
            }
        virtual void onCall(string message){
            cout <<"Id="<< id << "|message=" << message << endl;
        }
    private:
        int id;
    };
private:
    vector<Listener*> listeners;
};
int main()
{
    Notifier notifier;
    Notifier::Listener l1(notifier), l2(notifier);
    notifier.publishMessage("hello world" );
}
```

Dependency

Try two callbacks

```
class Person
{
public:
    class PersonListener{
    public:
        virtual void onAgeUpdated(int age);
        virtual void onCanVoteUpdated(int age);
    };
    void change_age(int new_age){
        for(auto listener:listeners)
        {
            listener->onAgeUpdated(new_age);
            listener->onCanVoteUpdated(new_age);
        }
    };
private:
    vector<PersonListener*> listeners;
};
```

```
#include "Headers.hpp"
#include "Observer.hpp"
#include "Observable.hpp"
#include "SaferObservable.hpp"

class Person : public SaferObservable<Person>
{
  int age;
public:
  Person(int age) : age(age) {}

  int get_age() const
  {
    return age;
  }

  void set_age(int age)
  {
    if (this->age == age) return;

    auto old_can_vote = get_can_vote();
    this->age = age;
    notify(*this, "age");

    // determine if voting status changed
    if (old_can_vote != get_can_vote())
      notify(*this, "can_vote");
  }

  bool get_can_vote() const
  {
    return age >= 16;
  }
};

// we could define a Person-specific listener
struct PersonListener
{
  virtual void person_changed(Person& p,
    const string property_name) = 0;
}; // changes can occur on other objects, so


struct ConsolePersonObserver
  : public Observer<Person> // , Observer<Creature>
{
  void field_changed(Person &source, const string &field_name) override
  {
    cout << "Person's " << field_name << " has changed to ";
    if (field_name == "age") cout << source.get_age();
    if (field_name == "can_vote") cout << boolalpha << source.get_can_vote();
    cout << ".\n";
  }
};
```

Boost

```cpp
template <typename T>
struct Observable
{
  virtual ~Observable() = default;
  signal<void(T&, const string&)> property_changed;
};

struct Person : Observable<Person>
{
  explicit Person(int age)
    : age(age)
  {
  }

  int get_age() const
  {
    return age;
  }

  void set_age(const int age)
  {
    if (this->age == age) return;
    this->age = age;
    property_changed(*this, "age");
  }

private:
  int age;
};

int main_()
{
  Person p{123};
  p.property_changed.connect([](Person&, const string& prop_name)
  {
    cout << prop_name << " has been changed" << endl;
  });
  p.set_age(20);

  getchar();
  return 0;
}
```

Multi-thread example

```cpp
#include <thread>
#include <map>
#include <iostream>
#include <mutex>
#include <unistd.h>

using namespace std;

struct SharedObject
{
    std::mutex mapMu;
    SharedObject(){};
    std::map<int, int> sharedMap;
    void add_item()
    {
        while(true)
        {
            std::lock_guard<std::mutex> guard(mapMu);
            for(int i = 0; i < 10; ++i)
                sharedMap[i] = i;
        }
    }
    void print_item()
    {
        while(true)
        {
            cout << "reading maps" << endl;
            {
                std::lock_guard<std::mutex> guard(mapMu);
                for(auto item:sharedMap)
                    cout << item.first  << ":" << item.second << endl;
                sharedMap.clear();
            }
            cout << "finished reading map" << endl;
            sleep(1);
        }
        return;
    }
};

void add_obj(SharedObject& obj)
{
    obj.add_item();
    return;
}
void read_obj(SharedObject& obj)
{
    obj.print_item();
    return;
}
```

Observer exercise

```cpp
#include <iostream>
#include <vector>

using namespace std;

struct IRat{
    virtual void change_attack(int value_change) =0;
};
struct Game
{
    vector<IRat*> listeners;
    void add_rat(IRat* rat)
    {
        listeners.emplace_back(rat);
        for(auto listener:listeners)
            listener->change_attack(1);
    }
    void reduce_rat(IRat* rat)
    {
        for(auto listener:listeners)
            listener->change_attack(-1);
        listeners.erase(remove(listeners.begin(), listeners.end(), rat
        listeners.end());
    }
};

struct Rat : IRat
{
    Game& game;
    int attack{1};

    Rat(Game &game) : game(game)
    {
        attack = game.listeners.size();
        game.add_rat(this);
    }
    virtual void change_attack(int value_change)
    {
        attack += value_change;
    }

    ~Rat()
    {
        game.reduce_rat(this);
    }
};
```