Пример 13.01. Стратегия (Strategy).

# include <iostream>

# include <memory>

# include <vector>

using namespace std;

class Strategy

{

public:

virtual ~Strategy() = default;

virtual void algorithm() = 0;

};

class ConStrategy1 : public Strategy

{

public:

virtual void algorithm() override { cout << "Algorithm 1;" << endl; }

};

class ConStrategy2 : public Strategy

{

public:

virtual void algorithm() override { cout << "Algorithm 2;" << endl; }

};

class Context

{

protected:

unique\_ptr<Strategy> strategy;

public:

explicit Context(unique\_ptr<Strategy> ptr = make\_unique<ConStrategy1>())

: strategy(move(ptr)) {}

virtual ~Context() = default;

virtual void algorithmStrategy() = 0;

};

class Client1 : public Context

{

public:

using Context::Context;

virtual void algorithmStrategy() override { strategy->algorithm(); }

};

void main()

{

// shared\_ptr<Context> obj = make\_shared<Client1>();

shared\_ptr<Context> obj = make\_shared<Client1>(make\_unique<ConStrategy2>());

obj->algorithmStrategy();

}

Пример 13.02. Стратегия (Strategy).

# include <iostream>

# include <memory>

# include <vector>

using namespace std;

class Strategy

{

public:

virtual ~Strategy() = default;

virtual void algorithm() = 0;

};

class ConStrategy1 : public Strategy

{

public:

virtual void algorithm() override { cout << "Algorithm 1;" << endl; }

};

class ConStrategy2 : public Strategy

{

public:

virtual void algorithm() override { cout << "Algorithm 2;" << endl; }

};

class Context

{

public:

virtual void algorithmStrategy(shared\_ptr<Strategy> strategy) = 0;

};

class Client1 : public Context

{

public:

virtual void algorithmStrategy(shared\_ptr<Strategy> strategy = make\_shared<ConStrategy1>()) override

{ strategy->algorithm(); }

};

void main()

{

shared\_ptr<Context> obj = make\_shared<Client1>();

shared\_ptr<Strategy> strategy = make\_shared<ConStrategy2>();

obj->algorithmStrategy(strategy);

}

Пример 13.03. Стратегия (Strategy). Стратегия на шаблоне.

# include <iostream>

# include <memory>

# include <vector>

using namespace std;

class Strategy

{

public:

virtual ~Strategy() = default;

virtual void algorithm() = 0;

};

class ConStrategy1 : public Strategy

{

public:

virtual void algorithm() override { cout << "Algorithm 1;" << endl; }

};

class ConStrategy2 : public Strategy

{

public:

virtual void algorithm() override { cout << "Algorithm 2;" << endl; }

};

template <typename TStrategy = ConStrategy1>

class Context

{

private:

unique\_ptr<TStrategy> strategy;

public:

Context() : strategy(make\_unique<TStrategy>()) {}

void algorithmStrategy() { strategy->algorithm(); }

};

int main()

{

// shared\_ptr<Context<>> obj = make\_shared<Context<>>();

using Client = Context<ConStrategy2>;

shared\_ptr<Client> obj = make\_shared<Client>();

obj->algorithmStrategy();

}

Пример 13.04. Стратегия (Strategy) на примере сортировки массива.

# include <iostream>

# include <memory>

# include <initializer\_list>

using namespace std;

class Strategy;

class Array

{

public:

Array(initializer\_list<double> list);

void sort(shared\_ptr<Strategy> algorithm);

const double& operator [](int index) const { return this->arr[index]; }

unsigned size() const { return count; }

private:

shared\_ptr<double[]> arr;

unsigned count;

};

class Strategy

{

public:

virtual void algorithmSort(shared\_ptr<double[]> ar, unsigned cnt) = 0;

};

#pragma region Array methods

Array::Array(initializer\_list<double> list)

{

this->count = list.size();

this->arr = shared\_ptr<double[]>(new double[this->count]);

unsigned i = 0;

for (auto elem : list)

arr[i++] = elem;

}

void Array::sort(shared\_ptr<Strategy> algorithm)

{

algorithm->algorithmSort(this->arr, this->count);

}

#pragma endregion

template <typename TComparison>

class BustStrategy : public Strategy

{

public:

virtual void algorithmSort(shared\_ptr<double[]> ar, unsigned cnt) override

{

for (int i = 0; i < cnt - 1; i++)

for (int j = i + 1; j < cnt; j++)

{

if (TComparison::compare(ar[i], ar[j]) > 0)

swap(ar[i], ar[j]);

}

}

};

template <typename Type>

class Comparison

{

public:

static int compare(const Type& elem1, const Type& elem2) { return elem1 - elem2; }

};

ostream& operator <<(ostream& os, const Array& ar)

{

for (int i = 0; i < ar.size(); i++)

os << " " << ar[i];

return os;

}

void main()

{

using TStrategy = BustStrategy<Comparison<double>>;

shared\_ptr<Strategy> strategy = make\_shared<TStrategy>();

Array ar{ 8., 6., 4., 3., 2., 7., 1. };

ar.sort(strategy);

cout << ar << endl;

}

Пример 13.13. “Статический полиморфизм”. Паттерн CRTP (Curiously Recurring Template Pattern).

# include <iostream>

# include <memory>

using namespace std;

template<typename Implementation>

class Product

{

public:

virtual ~Product() { cout << "Destructor Product;" << endl; }

void run() { impl()->method(); }

private:

Implementation\* impl()

{

return static\_cast<Implementation\*>(this);

}

void method() { cout << "Method Product;" << endl; }

};

class ConProd1 : public Product<ConProd1>

{

public:

virtual ~ConProd1() override { cout << "Destructor Conprod1;" << endl; }

private:

friend class Product<ConProd1>;

void method() { cout << "Method ConProd1;" << endl; }

};

class ConProd2 : public Product<ConProd2>

{

public:

virtual ~ConProd2() override { cout << "Destructor Conprod2;" << endl; }

};

int main()

{

unique\_ptr<Product<ConProd1>> prod1 = make\_unique<ConProd1>();

prod1->run();

unique\_ptr<Product<ConProd2>> prod2 = make\_unique<ConProd2>();

prod2->run();

}

Пример 13.17. “Статический полиморфизм” на примере паттерна “Прототип”.

# include <iostream>

# include <memory>

using namespace std;

class Base {

public:

virtual ~Base() = default;

virtual unique\_ptr<Base> clone() const = 0;

};

template <typename Derived>

class Cloner : public Base

{

public:

virtual unique\_ptr<Base> clone() const override

{

return unique\_ptr<Base>(new Derived(\*static\_cast<const Derived\*>(this)));

}

};

class Derived : public Cloner<Derived>

{

public:

Derived() { cout << "Default constructor of the Deriver class;" << endl; }

Derived(const Derived& dr) { cout << "Copy constructor of the Deriver class;" << endl; }

~Derived() { cout << "Destructor of the Deriver class;" << endl; }

};

void main()

{

unique\_ptr<Base> b0 = make\_unique<Derived>();

unique\_ptr<Base> b1 = b0->clone();

}

Пример 13.05. Команда (Command). Объект известен.

# include <iostream>

# include <memory>

# include <vector>

# include <initializer\_list>

using namespace std;

class Command

{

public:

virtual ~Command() = default;

virtual void execute() = 0;

};

template <typename Reseiver>

class SimpleCommand : public Command

{

using Action = void(Reseiver::\*)();

using Pair = pair<shared\_ptr<Reseiver>, Action>;

private:

Pair call;

public:

SimpleCommand(shared\_ptr<Reseiver> r, Action a) : call(r, a) {}

virtual void execute() override { ((\*call.first).\*call.second)(); }

};

class CompoundCommand : public Command

{

using VectorCommand = vector<shared\_ptr<Command>>;

private:

VectorCommand vec;

public:

CompoundCommand(initializer\_list<shared\_ptr<Command>> lt);

virtual void execute() override;

};

# pragma region Methods

CompoundCommand::CompoundCommand(initializer\_list<shared\_ptr<Command>> lt)

{

for (auto elem : lt)

vec.push\_back(elem);

}

void CompoundCommand::execute()

{

for (auto com : vec)

com->execute();

}

# pragma endregion

class Object

{

public:

void run() { cout << "Run method;" << endl; }

};

int main()

{

shared\_ptr<Object> obj = make\_shared<Object>();

shared\_ptr<Command> command = make\_shared<SimpleCommand<Object>>(obj, &Object::run);

command->execute();

shared\_ptr<Command> complex(new CompoundCommand

{

make\_shared<SimpleCommand<Object>>(obj, &Object::run),

make\_shared<SimpleCommand<Object>>(obj, &Object::run)

});

complex->execute();

}

Пример 13.06. Команда (Command). Объект неизвестен.

# include <iostream>

# include <memory>

using namespace std;

template <typename Reseiver>

class Command

{

public:

virtual ~Command() = default;

virtual void execute(shared\_ptr<Reseiver>) = 0;

};

template <typename Reseiver>

class SimpleCommand : public Command<Reseiver>

{

using Action = void(Reseiver::\*)();

private:

Action act;

public:

SimpleCommand(Action a) : act(a) {}

virtual void execute(shared\_ptr<Reseiver> r) override { ((\*r).\*act)(); }

};

class Object

{

public:

virtual void run() = 0;

};

class ConObject : public Object

{

public:

virtual void run() override { cout << "Run method;" << endl; }

};

int main()

{

shared\_ptr<Command<Object>> command = make\_shared<SimpleCommand<Object>>(&Object::run);

shared\_ptr<Object> obj = make\_shared<ConObject>();

command->execute(obj);

}

Пример 13.07. Цепочка обязанностей (Chain of Responsibility).

# include <iostream>

# include <initializer\_list>

# include <memory>

using namespace std;

class AbstractHandler

{

using PtrAbstractHandler = shared\_ptr<AbstractHandler>;

protected:

PtrAbstractHandler next;

virtual bool run() = 0;

public:

using Default = shared\_ptr<AbstractHandler>;

virtual ~AbstractHandler() = default;

virtual bool handle() = 0;

void add(PtrAbstractHandler node);

void add(PtrAbstractHandler node1, PtrAbstractHandler node2, ...);

void add(initializer\_list<PtrAbstractHandler> list);

};

class ConHandler : public AbstractHandler

{

private:

bool condition{ false };

protected:

virtual bool run() override { cout << "Method run;" << endl; return true; }

public:

ConHandler() : ConHandler(false) {}

ConHandler(bool c) : condition(c) { cout << "Constructor;" << endl; }

virtual ~ConHandler() override { cout << "Destructor;" << endl; }

virtual bool handle() override

{

if (!condition) return next ? next->handle() : false;

return run();

}

};

#pragma region Methods

void AbstractHandler::add(PtrAbstractHandler node)

{

if (next)

next->add(node);

else

next = node;

}

void AbstractHandler::add(PtrAbstractHandler node1, PtrAbstractHandler node2, ...)

{

for (Default\* ptr = &node1; \*ptr; ++ptr)

add(\*ptr);

}

void AbstractHandler::add(initializer\_list<PtrAbstractHandler> list)

{

for (auto elem : list)

add(elem);

}

#pragma endregion

int main()

{

shared\_ptr<AbstractHandler> chain = make\_shared<ConHandler>();

chain->add(

{

make\_shared<ConHandler>(false),

make\_shared<ConHandler>(true),

make\_shared<ConHandler>(true),

AbstractHandler::Default()

}

);

cout << "Result = " << chain->handle() << ";" << endl;

}

Пример 13.08. Подписчик-издатель (Publish-Subscribe).

# include <iostream>

# include <memory>

# include <vector>

using namespace std;

class Subscriber;

using Reseiver = Subscriber;

class Publisher

{

using Action = void(Reseiver::\*)();

using Pair = pair<shared\_ptr<Reseiver>, Action>;

private:

vector<Pair> callback;

int indexOf(shared\_ptr<Reseiver> r);

public:

bool subscribe(shared\_ptr<Reseiver> r, Action a);

bool unsubscribe(shared\_ptr<Reseiver> r);

void run();

};

class Subscriber

{

public:

virtual ~Subscriber() = default;

virtual void method() = 0;

};

class ConSubscriber : public Subscriber

{

public:

virtual void method() override { cout << "method;" << endl; }

};

#pragma region Methods Publisher

bool Publisher::subscribe(shared\_ptr<Reseiver> r, Action a)

{

if (indexOf(r) != -1) return false;

Pair pr(r, a);

callback.push\_back(pr);

return true;

}

bool Publisher::unsubscribe(shared\_ptr<Reseiver> r)

{

int pos = indexOf(r);

if (pos != -1)

callback.erase(callback.begin() + pos);

return pos != -1;

}

void Publisher::run()

{

cout << "Run:" << endl;

for (auto elem : callback)

((\*elem.first).\*(elem.second))();

}

int Publisher::indexOf(shared\_ptr<Reseiver> r)

{

int i = 0;

for (auto it = callback.begin(); it != callback.end() && r != (\*it).first; i++, ++it);

return i < callback.size() ? i : -1;

}

#pragma endregion

int main()

{

shared\_ptr<Subscriber> subscriber1 = make\_shared<ConSubscriber>();

shared\_ptr<Subscriber> subscriber2 = make\_shared<ConSubscriber>();

shared\_ptr<Publisher> publisher = make\_shared<Publisher>();

publisher->subscribe(subscriber1, &Subscriber::method);

if (publisher->subscribe(subscriber2, &Subscriber::method))

publisher->unsubscribe(subscriber1);

publisher->run();

}

Пример 13.09. Посредник (Mediator).

# include <iostream>

# include <memory>

# include <list>

# include <vector>

using namespace std;

class Message {}; // Request

class Mediator;

class Colleague

{

private:

weak\_ptr<Mediator> mediator;

public:

virtual ~Colleague() = default;

void setMediator(shared\_ptr<Mediator> mdr) { mediator = mdr; }

virtual bool send(shared\_ptr<Message> msg);

virtual void receive(shared\_ptr<Message> msg) = 0;

};

class ColleagueLeft : public Colleague

{

public:

virtual void receive(shared\_ptr<Message> msg) override { cout << "Right - > Left;" << endl; }

};

class ColleagueRight : public Colleague

{

public:

virtual void receive(shared\_ptr<Message> msg) override { cout << "Left - > Right;" << endl; }

};

class Mediator

{

protected:

list<shared\_ptr<Colleague>> colleagues;

public:

virtual ~Mediator() = default;

virtual bool send(const Colleague\* coleague, shared\_ptr<Message> msg) = 0;

static bool add(shared\_ptr<Mediator> mediator, initializer\_list<shared\_ptr<Colleague>> list);

};

class ConMediator : public Mediator

{

public:

virtual bool send(const Colleague\* coleague, shared\_ptr<Message> msg) override;

};

#pragma region Methods Colleague

bool Colleague::send(shared\_ptr<Message> msg)

{

shared\_ptr<Mediator> mdr = mediator.lock();

return mdr ? mdr->send(this, msg) : false;

}

#pragma endregion

#pragma region Methods Mediator

bool Mediator::add(shared\_ptr<Mediator> mediator, initializer\_list<shared\_ptr<Colleague>> list)

{

if (!mediator || list.size() == 0) return false;

for (auto elem : list)

{

mediator->colleagues.push\_back(elem);

elem->setMediator(mediator);

}

return true;

}

bool ConMediator::send(const Colleague\* colleague, shared\_ptr<Message> msg)

{

bool flag = false;

for (auto& elem : colleagues)

{

if (dynamic\_cast<const ColleagueLeft\*>(colleague) && dynamic\_cast<ColleagueRight\*>(elem.get()))

{

elem->receive(msg);

flag = true;

}

else if (dynamic\_cast<const ColleagueRight\*>(colleague) && dynamic\_cast<ColleagueLeft\*>(elem.get()))

{

elem->receive(msg);

flag = true;

}

}

return flag;

}

#pragma endregion

int main()

{

shared\_ptr<Mediator> mediator =make\_shared<ConMediator>();

shared\_ptr<Colleague> col1 = make\_shared<ColleagueLeft>();

shared\_ptr<Colleague> col2 = make\_shared<ColleagueRight>();

shared\_ptr<Colleague> col3 = make\_shared<ColleagueLeft>();

shared\_ptr<Colleague> col4 = make\_shared<ColleagueLeft>();

Mediator::add(mediator, { col1, col2, col3, col4 });

shared\_ptr<Message> msg = make\_shared<Message>();

col1->send(msg);

col2->send(msg);

}

Пример 13.10. Посетитель (Visitor).

# include <iostream>

# include <memory>

# include <vector>

using namespace std;

class Circle;

class Rectangle;

class Visitor

{

public:

virtual ~Visitor() = default;

virtual void visit(Circle& ref) = 0;

virtual void visit(Rectangle& ref) = 0;

};

class Shape

{

public:

virtual ~Shape() = default;

virtual void accept(shared\_ptr<Visitor> visitor) = 0;

};

class Circle : public Shape

{

public:

virtual void accept(shared\_ptr<Visitor> visitor) override { visitor->visit(\*this); }

};

class Rectangle : public Shape

{

public:

virtual void accept(shared\_ptr<Visitor> visitor) override { visitor->visit(\*this); }

};

class ConVisitor : public Visitor

{

public:

virtual void visit(Circle& ref) override { cout << "Circle;" << endl; }

virtual void visit(Rectangle& ref) override { cout << "Rectangle;" << endl; }

};

class Formation

{

public:

static vector<shared\_ptr<Shape>> initialization(initializer\_list<shared\_ptr<Shape>> list)

{

vector<shared\_ptr<Shape>> vec;

for (auto elem : list)

vec.push\_back(elem);

return vec;

}

};

int main()

{

vector<shared\_ptr<Shape>> figure = Formation::initialization(

{ make\_shared<Circle>(), make\_shared<Rectangle>(), make\_shared<Circle>() }

);

shared\_ptr<Visitor> visitor = make\_shared<ConVisitor>();

for (auto& elem : figure)

elem->accept(visitor);

}

Пример 13.21 Шаблонный посетитель (Template Visitor) с использованием паттерна CRTP.

# include <iostream>

# include <memory>

# include <initializer\_list>

# include <vector>

using namespace std;

template <typename ...Types>

class Visitor;

template <typename Type>

class Visitor<Type>

{

public:

virtual void visit(Type& t) = 0;

};

template <typename Type, typename ...Types>

class Visitor<Type, Types...> : public Visitor<Types...>

{

public:

using Visitor<Types...>::visit;

virtual void visit(Type& t) = 0;

};

using ShapeVisitor = Visitor<class Figure, class Camera>;

class Point {};

class Shape

{

public:

virtual ~Shape() = default;

Shape(const Point& pnt) : point(pnt) {}

const Point& getPoint() const { return point; }

void setPoint(const Point& pnt) { point = pnt; }

virtual void accept(shared\_ptr<ShapeVisitor> v) = 0;

private:

Point point;

};

template <typename Derived>

class Visitable : public Shape

{

using Shape::Shape;

public:

virtual void accept(shared\_ptr<ShapeVisitor> v) override

{

v->visit(\*static\_cast<Derived\*>(this));

}

};

class Figure : public Visitable<Figure>

{

using Visitable<Figure>::Visitable;

};

class Camera : public Visitable<Camera>

{

using Visitable<Camera>::Visitable;

};

class DrawVisitor : public ShapeVisitor

{

public:

virtual void visit(Figure& fig) override { cout << "Draws a figure;" << endl; }

virtual void visit(Camera& fig) override { cout << "Draws a camera;" << endl; }

};

class Formation

{

public:

static vector<shared\_ptr<Shape>> initialization(initializer\_list<shared\_ptr<Shape>> list)

{

vector<shared\_ptr<Shape>> vec;

for (auto elem : list)

vec.push\_back(elem);

return vec;

}

};

int main()

{

Point p;

vector<shared\_ptr<Shape>> figure = Formation::initialization(

{ make\_shared<Figure>(p), make\_shared<Camera>(p), make\_shared<Figure>(p) }

);

shared\_ptr<ShapeVisitor> visitor = make\_shared<DrawVisitor>();

for (auto& elem : figure)

elem->accept(visitor);

}

Пример 13.11. Опекун (Memento).

# include <iostream>

# include <memory>

# include <list>

using namespace std;

class Memento;

class Caretaker

{

public:

unique\_ptr<Memento> getMemento();

void setMemento(unique\_ptr<Memento> memento);

private:

list<unique\_ptr<Memento>> mementos;

};

class Originator

{

public:

Originator(int s) : state(s) {}

const int getState() const { return state; }

void setState(int s) { state = s; }

std::unique\_ptr<Memento> createMemento() { return make\_unique<Memento>(\*this); }

void restoreMemento(std::unique\_ptr<Memento> memento);

private:

int state;

};

class Memento

{

friend class Originator;

public:

Memento(Originator o) : originator(o) {}

private:

void setOriginator(Originator o) { originator = o; }

Originator getOriginator() { return originator; }

private:

Originator originator;

};

#pragma region Methods Caretaker

void Caretaker::setMemento(unique\_ptr<Memento> memento)

{

mementos.push\_back(move(memento));

}

unique\_ptr<Memento> Caretaker::getMemento() {

unique\_ptr<Memento> last = move(mementos.back());

mementos.pop\_back();

return last;

}

#pragma endregion

#pragma region Method Originator

void Originator::restoreMemento(std::unique\_ptr<Memento> memento)

{

\*this = memento->getOriginator();

}

#pragma endregion

int main()

{

auto originator = make\_unique<Originator>(1);

auto caretaker = make\_unique<Caretaker>();

cout << "State = " << originator->getState() << endl;

caretaker->setMemento(originator->createMemento());

originator->setState(2);

cout << "State = " << originator->getState() << endl;

caretaker->setMemento(originator->createMemento());

originator->setState(3);

cout << "State = " << originator->getState() << endl;

caretaker->setMemento(originator->createMemento());

originator->restoreMemento(caretaker->getMemento());

cout << "State = " << originator->getState() << endl;

originator->restoreMemento(caretaker->getMemento());

cout << "State = " << originator->getState() << std::endl;

originator->restoreMemento(caretaker->getMemento());

cout << "State = " << originator->getState() << std::endl;

}

Пример 13.14. Шаблонный метод (Template Method).

# include <iostream>

using namespace std;

class AbstractClass

{

public:

void templateMethod()

{

primitiveOperation();

concreteOperation();

hook();

}

protected:

virtual void primitiveOperation() = 0;

void concreteOperation() { cout << "concreteOperation;" << endl; }

virtual void hook() { cout << "hook Base;" << endl; }

};

class ConClassA : public AbstractClass

{

protected:

virtual void primitiveOperation() override { cout << "primitiveOperation A;" << endl; }

};

class ConClassB : public AbstractClass

{

protected:

virtual void primitiveOperation() override { cout << "primitiveOperation B;" << endl; }

void hook() { cout << "hook B;" << endl; }

};

int main()

{

ConClassA ca;

ConClassB cb;

ca.templateMethod();

cb.templateMethod();

}

Пример 13.12. Свойство (Property).

# include <iostream>

# include <memory>

using namespace std;

template <typename Owner, typename Type>

class Property

{

private:

using Getter = Type (Owner::\*)() const;

using Setter = void (Owner::\*)(const Type&);

Owner\* owner;

Getter methodGet;

Setter methodSet;

public:

Property() = default;

Property(Owner\* owr, Getter getmethod, Setter setmethod) : owner(owr), methodGet(getmethod), methodSet(setmethod) {}

void init(Owner\* owr, Getter getmethod, Setter setmethod)

{

owner = owr;

methodGet = getmethod;

methodSet = setmethod;

}

operator Type() { return (owner->\*methodGet)(); } // Getter

void operator=(const Type& data) { (owner->\*methodSet)(data); } // Setter

};

class Object

{

private:

double value;

public:

Object(double v) : value(v) { Value.init(this, &Object::getValue, &Object::setValue); }

double getValue() const { return value; }

void setValue(const double& v) { value = v; }

Property<Object, double> Value;

};

int main()

{

Object obj(5.);

cout << "value = " << obj.Value << endl;

obj.Value = 10.;

cout << "value = " << obj.Value << endl;

unique\_ptr<Object> ptr(new Object(15.));

cout << "value =" << ptr->Value << endl;

obj = \*ptr;

obj.Value = ptr->Value;

}

Пример 13.15. Пример определения наличия метода в классе.

# include <iostream>

using namespace std;

struct A

{

void f() {}

};

struct B {};

template <typename Type>

class Detecting

{

using P = void (Type::\*)();

template<typename U, P = &U::f>

using True = short;

using False = char;

private:

template<typename U>

static True<U> detect(U\*);

static False detect(...);

public:

static const bool exists = (sizeof(False) != sizeof(detect(static\_cast<Type\*>(0))));

};

void main()

{

cout << boolalpha << Detecting<A>::exists << endl; // true

cout << boolalpha << Detecting<B>::exists << endl; // false

}

Пример 13.16. Шаблон nullptr.

# include <iostream>

using namespace std;

const class nullPtr\_t

{

public:

// Может быть приведен к любому типу нулевого указателя (не на член класса)

template<class T>

inline operator T\* () const { return 0; }

// или любому типу нулевого указателя на член

template<class C, class T>

inline operator T C::\* () const { return 0; }

private:

void operator &() const = delete; // мы не можем взять адрес nullptr

} nullPtr = {};

void main()

{

int\* i = nullPtr;

if (i == nullPtr)

cout << "null ptr;" << endl;

}

Пример 13.18. Шаблон any (“безопасный” void\*) на основе идиомы Type erasure.

# include <iostream>

# include <memory>

using namespace std;

class Any

{

template <typename Type>

friend Type any\_Cast(const Any& he);

template <typename Type>

friend const Type\* any\_Cast(const Any\* he);

private:

class BaseValue;

unique\_ptr<BaseValue> object;

public:

Any() = default;

Any(const Any& other);

Any(Any&& other) noexcept;

template <typename Type>

Any(const Type& val);

Any& operator =(const Any& other);

Any& operator =(Any&& other) noexcept;

template <typename Type>

Any& operator =(const Type& val);

template <typename Type>

Any& emplace(const Type& val);

bool has\_value() const noexcept { return static\_cast<bool>(object); }

const std::type\_info& type() const noexcept

{

return object ? object->type() : typeid(void);

}

void reset() { object.reset(); }

template <typename Type>

operator Type() const;

private:

class BaseValue

{

public:

virtual ~BaseValue() = default;

virtual const std::type\_info& type() const noexcept = 0;

virtual unique\_ptr<BaseValue> clone() const = 0;

};

template <typename Type>

class Value : public BaseValue

{

private:

Type object;

public:

Value(const Type& t) : object(t) {}

virtual const std::type\_info& type() const noexcept override

{

return typeid(object);

}

virtual unique\_ptr<BaseValue> clone() const override

{

return make\_unique<Value<Type>>(object);

}

Type get() const { return object; }

const Type\* getPtr() const { return &object; }

};

};

template <typename Type>

Type any\_Cast(const Any& he)

{

if (typeid(Type) != he.type())

{

throw std::runtime\_error("Bad any\_cast!");

}

return Type(he);

}

template <typename Type>

const Type\* any\_Cast(const Any\* he)

{

Any::Value<Type>\* type = dynamic\_cast<Any::Value<Type>\*>(he->object.get());

return type ? type->getPtr() : nullptr;

}

# pragma region Any methods

Any::Any(const Any& other)

{

if (other.object)

{

this->object = other.object->clone();

}

}

Any::Any(Any&& other) noexcept

{

this->object = move(other.object);

}

template <typename Type>

Any::Any(const Type& val)

{

this->object = make\_unique<Value<Type>>(val);

}

Any& Any::operator =(const Any& other)

{

if (other.object)

{

this->object = other.object->clone();

}

else

{

this->object.reset();

}

return \*this;

}

Any& Any::operator =(Any&& other) noexcept

{

this->object = move(other.object);

return \*this;

}

template <typename Type>

Any& Any::operator =(const Type& val)

{

return emplace(val);

}

template <typename Type>

Any& Any::emplace(const Type& val)

{

this->object = make\_unique<Value<Type>>(val);

return \*this;

}

template <typename Type>

Any::operator Type() const

{

Value<Type>& type = dynamic\_cast<Value<Type>&>(\*object);

return type.get();

}

#pragma endregion

Any f()

{

Any temp = 7.5;

return temp;

}

void main()

{

try

{

Any v1 = 2, v2 = v1, v3 = f(), v4;

if (v3.has\_value())

{

cout << v3.type().name() << endl;

if (v3.type() == typeid(double))

cout << "v3 = " << double(v3) << endl;

}

v4 = f();

v1.reset();

int j = 7;

int& aj = j;

v1 = j;

cout << "v1 = " << any\_Cast<int>(v1) << endl;

cout << "v2 = " << any\_Cast<int>(v2) << endl;

v2.emplace(5.0f);

cout << "v2 = " << any\_Cast<float>(v2) << endl;

int i = v1;

float d = v2;

cout << "i = " << i << " f = " << d << endl;

}

catch (const std::exception& err)

{

cout << err.what() << endl;

}

}

Пример 13.19. Шаблон operation. Используется идиома Type erasure.

# include <iostream>

# include <memory>

using namespace std;

template <typename TypeUnused>

class Function;

template <typename TypeReturn, typename ...Args>

class Function<TypeReturn(Args ...)>

{

class Function\_holder\_base;

using invoker\_t = unique\_ptr<Function\_holder\_base>;

private:

invoker\_t mInvoker;

public:

Function() = default;

Function(const Function& other) : mInvoker(other.mInvoker->clone()) {}

template <typename TFunction>

Function(TFunction func)

: mInvoker(make\_unique<Function\_holder<TFunction>>(func)) {}

template <typename TypeFunction, typename TypeClass>

Function(TypeFunction TypeClass::\* method)

: mInvoker(make\_unique<Method\_holder<TypeFunction, Args ...>>(method)) {}

Function& operator = (const Function& other)

{

mInvoker = other.mInvoker->clone();

return \*this;

}

TypeReturn operator ()(Args ...args) { return mInvoker->invoke(args ...); }

private:

class Function\_holder\_base

{

public:

virtual ~Function\_holder\_base() = default;

virtual TypeReturn invoke(Args ...args) = 0;

virtual invoker\_t clone() const = 0;

};

template <typename TFunction>

class Function\_holder : public Function\_holder\_base

{

using self\_t = Function\_holder<TFunction>;

private:

TFunction mFunction;

public:

Function\_holder(TFunction func) : mFunction(func) {}

virtual TypeReturn invoke(Args ... args) override { return mFunction(args ...); }

virtual invoker\_t clone() const override

{

return invoker\_t(make\_unique<self\_t>(mFunction));

}

};

template <typename TypeFunction, typename TypeClass, typename ...RestArgs>

class Method\_holder : public Function\_holder\_base

{

using TMethod = TypeFunction TypeClass::\*;

private:

TMethod mFunction;

public:

Method\_holder(TMethod method) : mFunction(method) {}

virtual TypeReturn invoke(TypeClass obj, RestArgs ...restArgs) override

{

return (obj.\*mFunction)(restArgs...);

}

virtual invoker\_t clone() const override

{

return invoker\_t(new Method\_holder(mFunction));

}

};

};

struct Foo1

{

double smth(int x) { return x / 2.; }

};

struct Foo2

{

double smth(int x) { return x / 3.; }

};

class Test

{

int elem = 5;

public:

template <typename Tobj>

double result(Tobj& obj, Function<double(Tobj, int)> func)

{

return func(obj, this->elem);

}

};

void main()

{

Function<double(Foo1, int)> f1 = &Foo1::smth, f2;

Foo1 foo;

f2 = f1;

cout << "calling member function: " << f2(foo, 5) << endl;

Test ts;

cout << "calling member function: " << ts.result(foo, f2) << endl;

}