Пример 13.01. Стратегия (Strategy).

# include <iostream>

# include <memory>

using namespace std;

class Strategy

{

public:

virtual ~Strategy() = default;

virtual void algorithm() = 0;

};

class ConStrategy1 : public Strategy

{

public:

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

};

class ConStrategy2 : public Strategy

{

public:

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;

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

};

int main()

{

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:

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

};

class ConStrategy2 : public Strategy

{

public:

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

};

class Context

{

public:

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

};

class Client1 : public Context

{

public:

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

{

strategy->algorithm();

}

};

int 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 Strategy1

{

public:

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

};

class Strategy2

{

public:

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

};

template <typename TStrategy = Strategy1>

class Context

{

private:

unique\_ptr<TStrategy> strategy;

public:

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

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

};

int main()

{

using Client = Context<Strategy2>;

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 final

{

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:

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 13.05. 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) {}

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:

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(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; }

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

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(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)

}

);

cout << boolalpha << "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:

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:

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

};

class ColleagueRight : public Colleague

{

public:

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:

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:

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

};

class Rectangle : public Shape

{

public:

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

};

class ConVisitor : public Visitor

{

public:

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

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

};

class Figure : public Shape

{

using Shapes = vector<shared\_ptr<Shape>>;

private:

Shapes shapes;

public:

Figure(initializer\_list<shared\_ptr<Shape>> list)

{

for (auto&& elem : list)

shapes.emplace\_back(elem);

}

void accept(shared\_ptr<Visitor> visitor) override

{

for (auto& elem : shapes)

elem->accept(visitor);

}

};

int main()

{

shared\_ptr<Shape> figure = make\_shared<Figure>(

initializer\_list<shared\_ptr<Shape>>(

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

)

);

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

figure->accept(visitor);

}

Пример 13.11. Посетитель (Visitor). Приведение типа между базовыми классами.

# include <iostream>

# include <vector>

# include <memory>

using namespace std;

class AbstractVisitor

{

public:

virtual ~AbstractVisitor() = default;

};

template <typename T>

class Visitor

{

public:

virtual ~Visitor() = default;

virtual void visit(const T&) const = 0;

};

class Shape

{

public:

Shape() = default;

virtual ~Shape() = default;

virtual void accept(const AbstractVisitor&) const = 0;

};

class Circle : public Shape

{

private:

double radius;

public:

Circle(double radius) : radius(radius) {}

void accept(const AbstractVisitor& v) const override

{

auto cv = dynamic\_cast<const Visitor<Circle>\*>(&v);

if (cv)

{

cv->visit(\*this);

}

}

};

class Square : public Shape

{

private:

double side;

public:

Square(double side) : side(side) {}

void accept(const AbstractVisitor& v) const override

{

auto cv = dynamic\_cast<const Visitor<Square>\*>(&v);

if (cv)

{

cv->visit(\*this);

}

}

};

class DrawCircle : public Visitor<Circle>

{

void visit(const Circle& circle) const override

{

cout << "Circle" << endl;

}

};

class DrawSquare : public Visitor<Square>

{

void visit(const Square& circle) const override

{

cout << "Square" << endl;

}

};

class Figure : public Shape

{

using Shapes = vector<shared\_ptr<Shape>>;

private:

Shapes shapes;

public:

Figure(initializer\_list<shared\_ptr<Shape>> list)

{

for (auto&& elem : list)

shapes.emplace\_back(elem);

}

void accept(const AbstractVisitor& visitor) const override

{

for (auto& elem : shapes)

elem->accept(visitor);

}

};

class Draw : public AbstractVisitor, public DrawCircle, public DrawSquare {};

int main()

{

shared\_ptr<Shape> figure = make\_shared<Figure>(

initializer\_list<shared\_ptr<Shape>>({ make\_shared<Circle>(1), make\_shared<Square>(2) })

);

figure->accept(Draw{});

}

Пример 13.12. Посетитель (Visitor) с использованием шаблона variant (“безопасный” union).

# include <iostream>

# include <vector>

# include <variant>

using namespace std;

class Circle {};

class Square {};

using Shape = std::variant<Circle, Square>;

class Formation

{

public:

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

{

vector<Shape> vec;

for (auto&& elem : list)

vec.emplace\_back(elem);

return vec;

}

};

class Draw

{

public:

void operator ()(const Circle&) const { cout << "Circle" << endl; }

void operator ()(const Square&) const { cout << "Square" << endl; }

};

int main()

{

using Shapes = vector<Shape>;

Shapes fiqure = Formation::initialization({ Circle{}, Square{} });

for (const auto& elem : fiqure)

std::visit(Draw{}, elem);

}

Пример 13.13. 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:

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

virtual ~Shape() = default;

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

{

public:

using Shape::Shape;

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 Composite : public Shape

{

using Shapes = vector<shared\_ptr<Shape>>;

private:

Shapes shapes{};

public:

Composite(initializer\_list<shared\_ptr<Shape>> list) : Shape(Point{})

{

for (auto&& elem : list)

shapes.emplace\_back(elem);

}

void accept(shared\_ptr<ShapeVisitor> visitor) override

{

for (auto& elem : shapes)

elem->accept(visitor);

}

};

class DrawVisitor : public ShapeVisitor

{

public:

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

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

};

int main()

{

Point p;

shared\_ptr<Composite> figure = make\_shared<Composite>(

initializer\_list<shared\_ptr<Shape>>(

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

)

);

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

figure->accept(visitor);

}

Пример 13.14. 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 13.15. Шаблонный метод (Template Method).

# include <iostream>

using namespace std;

class AbstractClass

{

public:

void templateMethod()

{

primitiveOperation();

concreteOperation();

hook();

}

virtual ~AbstractClass() = default;

protected:

virtual void primitiveOperation() = 0;

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

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

};

class ConClassA : public AbstractClass

{

protected:

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

};

class ConClassB : public AbstractClass

{

protected:

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

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

};

int main()

{

ConClassA ca;

ConClassB cb;

ca.templateMethod();

cb.templateMethod();

}

Пример 13.16. Свойство (Property).

# include <iostream>

# include <memory>

using namespace std;

template <typename Owner, typename Type>

class Property

{

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

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

private:

Owner\* owner;

Getter methodGet;

Setter methodSet;

public:

Property() = default;

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

void init(Owner\* const 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

// Property(const Property&) = delete;

// Property& operator=(const Property&) = delete;

};

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 = make\_unique<Object>(15.);

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

obj = \*ptr;

obj.Value = ptr->Value;

}

Пример 13.17. Свойство (Property). Специализация для ReadOnly и WriteOnly.

# include <iostream>

using namespace std;

struct ReadOnly\_tag {};

struct WriteOnly\_tag {};

struct ReadWrite\_tag {};

template <typename Owner, typename Type, typename Access = ReadWrite\_tag>

class Property

{

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

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

private:

Owner\* owner;

Getter methodGet;

Setter methodSet;

public:

Property() = default;

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

void init(Owner\* const 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

};

template<typename Owner, typename Type>

class Property<typename Owner, typename Type, ReadOnly\_tag>

{

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

private:

Owner\* owner;

Getter methodGet;

public:

Property() = default;

Property(Owner\* const owr, Getter getmethod) : owner(owr), methodGet(getmethod) {}

void init(Owner\* const owr, Getter getmethod)

{

owner = owr;

methodGet = getmethod;

}

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

};

template<typename Owner, typename Type>

class Property<typename Owner, typename Type, WriteOnly\_tag>

{

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

private:

Owner\* owner;

Setter methodSet;

public:

Property() = default;

Property(Owner\* const owr, Setter setmethod) : owner(owr), methodSet(setmethod) {}

void init(Owner\* const owr, Setter setmethod)

{

owner = owr;

methodSet = setmethod;

}

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

};

class Object

{

public:

Object(double vRW = 0., double vRO = 0., double vWO = 0.)

: valueRW(vRW), valueRO(vRO), valueWO(vWO)

{

ValueRW.init(this, &Object::getValueRW, &Object::setValueRW);

ValueRO.init(this, &Object::getValueRO);

ValueWO.init(this, &Object::setValueWO);

}

private:

double valueRW;

public:

Property<Object, double> ValueRW;

double getValueRW() const { return valueRW; }

void setValueRW(const double& v) { valueRW = v; }

private:

double valueRO;

public:

Property<Object, double, ReadOnly\_tag> ValueRO;

double getValueRO() const { return valueRO; }

private:

double valueWO;

public:

Property<Object, double, WriteOnly\_tag> ValueWO;

void setValueWO(const double& v) { valueWO = v; }

};

void main()

{

Object obj(5., 15., 25.);

obj.ValueRW = 10.;

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

// obj.ValueRO = 10.; // Error! (ReadOnly)

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

obj.ValueWO = 10.;

// cout << "value = " << obj.ValueWO << endl; // Error! (WriteOnly)

}

Пример 13.18. “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:

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

private:

friend class Product<ConProd1>;

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

};

class ConProd2 : public Product<ConProd2>

{

public:

~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 13.19. “Статический полиморфизм”. Идиома MixIn.

# include <iostream>

using namespace std;

template <typename Derived>

struct Increment

{

Derived& operator ++()

{

auto& self = static\_cast<Derived&>(\*this);

self.setValue(self.getValue() + 1);

return self;

}

Derived operator ++(int)

{

auto& self = static\_cast<Derived&>(\*this);

Derived temp = self;

self.setValue(self.getValue() + 1);

return temp;

}

};

// C++23

/\*

struct Increment

{

auto& operator ++(this auto& self)

{

self.setValue(self.getValue() + 1);

return self;

}

auto operator ++(this auto& self, int)

{

auto tmp = self;

self.setValue(self.getValue() + 1);

return tmp;

}

};

\*/

class Age : public Increment<Age>

{

private:

unsigned short age;

public:

Age(unsigned short value) : age(value) {}

unsigned short getValue() const { return age; }

void setValue(unsigned short value) { age = value; }

};

int main()

{

Age a{ 18 };

a++;

cout << "age = " << a.getValue() << endl;

}

Пример 13.20. “Статический полиморфизм”. MixIn в виде свободного оператора.

# include <iostream>

using namespace std;

# pragma region Comparisons

template <typename Derived> struct Comparisons {};

template <typename Derived>

bool operator ==(const Comparisons<Derived>& c1, const Comparisons<Derived>& c2)

{

const Derived& d1 = static\_cast<const Derived&>(c1);

const Derived& d2 = static\_cast<const Derived&>(c2);

return !(d1 < d2) && !(d2 < d1);

}

template <typename Derived>

bool operator !=(const Comparisons<Derived>& c1, const Comparisons<Derived>& c2)

{

return !(c1 == c2);

}

# pragma endregion

# pragma region Object\_t

template <typename Derived>

struct Object\_t

{

public:

virtual ~Object\_t() = default;

bool less(const Object\_t<Derived>& rhs) const

{

const Derived& rs = static\_cast<const Derived&>(rhs);

return static\_cast<const Derived\*>(this)->less(rs);

}

protected:

Object\_t() = default;

};

template <typename Derived>

bool operator <(const Object\_t<Derived>& lhs, const Object\_t<Derived>& rhs)

{

return lhs.less(rhs);

}

# pragma endregion

class Int\_t : public Object\_t<Int\_t>, public Comparisons<Int\_t>

{

public:

Int\_t() : Int\_t(0) {}

Int\_t(int d) : data(d) {}

bool less(const Int\_t& rhs) const { return data < rhs.data; }

private:

int data;

};

int main()

{

Int\_t i{ 10 }, j{ 10 }, k;

if (i == j)

cout << "i == j" << endl;

else

cout << "i != j" << endl;

Object\_t<Int\_t>& ref = k;

cout << boolalpha << ref.less(j) << endl;

}

Пример 13.21. 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 <typename T>

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

// или любому типу нулевого указателя на член

template<typename C, typename T>

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

private:

void operator &() const = delete;

} nullPtr = {};

void main()

{

int\* i = nullPtr;

if (i == nullPtr)

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

}

Пример 13.22. .

# include <iostream>

# include <type\_traits>

# include <tuple>

using namespace std;

namespace my

{

const struct p\_1\_ { static const unsigned index = 0; } \_1\_;

const struct p\_2\_ { static const unsigned index = 1; } \_2\_;

const struct p\_3\_ { static const unsigned index = 2; } \_3\_;

template <typename T>

concept PlaceHolder = is\_same\_v<T, p\_1\_> || is\_same\_v<T, p\_2\_> || is\_same\_v<T, p\_3\_>;

template <typename T>

concept NotPlaceHolder = !PlaceHolder<T>;

template <PlaceHolder BindArg, typename CallArgTuple>

auto get\_arg(BindArg, CallArgTuple&& call\_args)

{

return std::get<BindArg::index>(call\_args);

}

template <NotPlaceHolder BindArg, typename CallArgTuple>

auto get\_arg(BindArg arg, CallArgTuple&&)

{

return arg;

}

template <typename F, typename... BindArgs>

struct binder

{

F f;

tuple<BindArgs...> bind\_args;

template <typename CallArgTuple, size\_t... Indexes>

auto call(std::index\_sequence<Indexes...>, CallArgTuple&& call\_args)

{

return f(get\_arg(std::get<Indexes>(bind\_args), call\_args)...);

}

template <typename... CallArgs>

auto operator ()(CallArgs... call\_args)

{

return call(std::make\_index\_sequence<sizeof...(BindArgs)>(), std::make\_tuple(call\_args...));

}

};

template <typename F, typename... BindArgs>

binder<F, BindArgs...> bind(F f, BindArgs... bind\_args)

{

return { f, { bind\_args... } };

}

}

void foo(int a, int b)

{

std::cout << a << " " << b << std::endl;

}

int main()

{

auto f1 = my::bind(foo, 5, my::\_1\_);

f1(8);

auto f2 = my::bind(foo, my::\_2\_, my::\_1\_);

f2(5, 8);

auto f3 = my::bind(foo, 5, 8);

f3();

}

Пример 13.23. Шаблон any (“безопасный” void) на основе идиомы Type erasure.

# include <iostream>

using namespace std;

namespace my

{

# pragma region Concepts

template <typename T>

struct is\_in\_place\_type : std::false\_type {};

template <typename T>

struct is\_in\_place\_type<std::in\_place\_type\_t<T>> : std::true\_type {};

class any;

template <typename Type, typename... Args>

concept Constructible = is\_constructible\_v<Type, Args...>;

template <typename Type>

concept CopyConstructible = is\_copy\_constructible\_v<decay\_t<Type>>;

template <typename Type>

concept NotAnyCopyConstuctible = CopyConstructible<Type> && !is\_same\_v<decay\_t<Type>, any>;

template <typename Type>

concept TypeAnyAble = NotAnyCopyConstuctible<Type>

&& !is\_in\_place\_type<std::decay\_t<Type>>::value;

# pragma endregion

class any

{

template <typename Type>

friend const Type\* any\_cast(const any\*) noexcept;

template <typename Type>

friend Type\* any\_cast(any\*) noexcept;

public:

any() = default;

any(const any& other);

any(any&& other) noexcept;

template <TypeAnyAble Type>

any(Type&& value);

template <CopyConstructible Type, typename... Args>

explicit any(in\_place\_type\_t<Type>, Args&&... args) requires Constructible<Type, Args...>;

any& operator =(const any& other);

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

template <NotAnyCopyConstuctible Type>

any& operator =(Type&& value);

template <CopyConstructible Type, typename... Args>

decay\_t<Type>& emplace(Args&&... args) requires Constructible<Type, Args...>;

bool has\_value() const noexcept { return bool(ptr); }

const type\_info& type() const noexcept

{

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

}

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

void swap(any& other) noexcept { ::swap(ptr, other.ptr); }

template <typename Type>

operator Type() const;

private:

class storage\_base;

unique\_ptr<storage\_base> ptr;

# pragma region Type erasure

class storage\_base

{

public:

virtual ~storage\_base() = default;

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

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

};

template <typename Type>

class storage\_impl final : public storage\_base

{

public:

template <typename... Args>

storage\_impl(Args&&... args) : value(forward<Args>(args)...) {}

const type\_info& type() const noexcept override { return typeid(Type); }

unique\_ptr<storage\_base> clone() const override

{

return make\_unique<storage\_impl<Type>>(value);

}

Type get() const { return value; }

const Type\* getptr() const { return &value; }

private:

Type value;

};

# pragma endregion

};

# pragma region Method

any::any(const any& other)

{

if (other.ptr)

{

ptr = other.ptr->clone();

}

}

any::any(any&& other) noexcept : ptr(move(other.ptr)) {}

template <TypeAnyAble Type>

any::any(Type&& value)

{

emplace<decay\_t<Type>>(forward<Type>(value));

}

template <CopyConstructible Type, typename... Args>

any::any(in\_place\_type\_t<Type>, Args&&... args) requires Constructible<Type, Args...>

{

emplace<decay\_t<Type>>(forward<Args>(args)...);

}

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

{

any(other).swap(\*this);

return \*this;

}

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

{

any(move(other)).swap(\*this);

return \*this;

}

template <NotAnyCopyConstuctible Type>

any& any::operator =(Type&& value)

{

any(forward<Type>(value)).swap(\*this);

return \*this;

}

template <CopyConstructible Type, typename... Args>

decay\_t<Type>& any::emplace(Args&&... args) requires Constructible<Type, Args...>

{

auto temp = make\_unique<storage\_impl<Type>>(forward<Args>(args)...);

auto vl = temp->get();

ptr = move(temp);

return vl;

}

template <typename Type>

any::operator Type() const

{

storage\_impl<Type>& type = dynamic\_cast<storage\_impl<Type>&>(\*ptr);

return type.get();

}

# pragma endregion

# pragma region Template functions

template <CopyConstructible Type, typename... Args>

any make\_any(Args&&... args) requires Constructible<Type, Args...>

{

return any(in\_place\_type<Type>, forward<Args>(args)...);

}

template <typename Type>

Type any\_cast(const any& thing)

{

auto\* value = any\_cast<Type>(&thing);

if (!value) throw runtime\_error("Bad any\_cast"); // bad\_any\_cast();

return static\_cast<Type>(\*value);

}

template <typename Type>

const Type\* any\_cast(const any\* other) noexcept

{

if (!other) return nullptr;

auto\* storage = dynamic\_cast<any::storage\_impl<Type>\*>(other->ptr.get());

return storage ? storage->getptr() : nullptr;

}

template <typename Type>

Type\* any\_cast(any\* other) noexcept

{

return const\_cast<Type\*>(any\_cast<Type>(const\_cast<const any\*>(other)));

}

# pragma endregion

}

my::any f()

{

my::any temp = 7.5;

return temp;

}

int main()

{

try

{

my::any v1 = 2, v2 = v1, v3 = f(), v4;

auto v5 = my::make\_any<float>(5.5);

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 = " << my::any\_cast<int>(v1) << endl;

cout << "v2 = " << my::any\_cast<int>(v2) << endl;

v2.emplace<float>(5.5f);

cout << "v2 = " << my::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 13.24. Шаблон variant (“безопасный” union).

# include <iostream>

# include <exception>

using namespace std;

class bad\_variant\_access : public exception

{

public:

bad\_variant\_access() : exception("Bad variant access!") {}

};

template <typename... Types>

class Variant

{

private:

template <typename... Ts>

union UnionStorage {};

template <typename Head>

union UnionStorage<Head>

{

private:

Head head;

public:

UnionStorage() {}

~UnionStorage() {}

void destroy(int index)

{

if (index != 0) throw bad\_variant\_access();

head.Head::~Head();

}

template <typename Type>

int put(const Type& value, size\_t index)

{

if (!std::is\_same\_v<Head, Type>) throw bad\_variant\_access();

new(&head) Type(value);

return index;

}

template <typename Type>

Type get(int index) const

{

if (index != 0 || !std::is\_same\_v<Head, Type>) throw bad\_variant\_access();

return \*reinterpret\_cast<const Type\*>(&head);

}

int copy(const UnionStorage<Head>& stg, size\_t index)

{

if (index != 0) throw bad\_variant\_access();

new(&head) Head(stg.head);

return index;

}

};

template <typename Head, typename... Tail>

union UnionStorage<Head, Tail...>

{

private:

Head head;

UnionStorage<Tail...> tail;

public:

UnionStorage() {}

~UnionStorage() {}

void destroy(int index)

{

if (index == 0)

head.Head::~Head();

else

tail.destroy(index - 1);

}

template <typename Type>

int put(const Type& value, size\_t index = 0)

{

if (!std::is\_same\_v<Head, Type>)

return tail.put(value, index + 1);

new(&head) Type(value);

return index;

}

template <typename Type>

Type get(int index) const

{

if (index == 0 && is\_same\_v<Head, Type>)

return \*reinterpret\_cast<const Type\*>(&head);

return tail.get<Type>(index - 1);

}

int copy(const UnionStorage<Head, Tail...>& stg, size\_t index)

{

if (index != 0)

return tail.copy(stg.tail, index - 1);

new(&head) Head(stg.head);

return index;

}

};

public:

Variant() = default;

Variant(Variant<Types...>& const vr);

Variant(Variant<Types...>&& vr) noexcept;

template <typename Type>

explicit Variant(Type&& value) { which = storage.put(value); }

~Variant() { destroy(); }

Variant& operator =(Variant<Types...>& const vr);

Variant& operator =(Variant<Types...>&& vr) noexcept;

template <typename Type>

Variant& operator =(Type&& value);

int index() const noexcept { return which; }

bool valueless\_by\_exception() const noexcept { return which == -1; }

template <typename Type>

Type get() const { return storage.get<Type>(which); }

private:

int which{ -1 };

UnionStorage<Types...> storage;

void destroy()

{

if (which != -1)

storage.destroy(which);

}

};

# pragma region Variant methods

template<typename... Types>

Variant<Types...>::Variant(Variant<Types...>& const vr)

{

which = vr.which;

storage.copy(vr.storage, vr.which);

}

template<typename... Types>

Variant<Types...>::Variant(Variant&& vr) noexcept

{

which = vr.which;

storage = vr.storege;

vr.which = -1;

}

template <typename... Types>

Variant<Types...>& Variant<Types...>::operator =(Variant<Types...>& const vr)

{

destroy();

which = vr.which;

storage.copy(vr.storage, vr.which);

return \*this;

}

template <typename ...Types>

Variant<Types...>& Variant<Types...>::operator =(Variant&& vr) noexcept

{

destroy();

which = vr.which;

storage = vr.storege;

vr.which = -1;

return \*this;

}

template <typename... Types>

template <typename Type>

Variant<Types...>& Variant<Types...>::operator =(Type&& value)

{

destroy();

which = storage.put(value);

return \*this;

}

# pragma endregion

class Object

{

private:

int num = 10;

public:

Object() { cout << "Calling the default constructor!" << endl; }

Object(const Object& obj) { cout << "Calling the copy constructor!" << endl; }

~Object() { cout << "Calling the destructor!" << endl; }

int getNum() { return num; }

};

void main()

{

try

{

Variant<double, Object, int> var(5);

cout << var.get<int>() << endl;

var = 7.1;

cout << var.get<double>() << endl;

Object obj;

var = obj;

cout << var.get<Object>().getNum() << endl;

Variant<double, Object, int> var2(var);

var2 = var;

}

catch (bad\_variant\_access& err)

{

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

}

}

Пример 13.25. Шаблон function.

# 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) {}

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

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) {}

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

{

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

}

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;

}