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.

```cpp
# 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>
```

```cpp
# 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;
}
```