Пример 12.01. Адаптер (Adapter).

```
# include <iostream>
# include <memory>

using namespace std;

class BaseAdaptee
{
public:
	virtual ~BaseAdaptee() = default;

	virtual void specificRequest() = 0;
};

class ConAdaptee : public BaseAdaptee
{
public:
	virtual void specificRequest() override { cout << "Method ConAdaptee;" << endl; }
};

class Adapter
{
public:
	virtual ~Adapter() = default;

	virtual void request() = 0;
};

class ConAdapter : public Adapter
{
private:
	shared_ptr<BaseAdaptee>  adaptee;

public:
	ConAdapter(shared_ptr<BaseAdaptee> ad) : adaptee(ad) {}

	virtual void request() override;
};

# pragma region Methods
void ConAdapter::request()
{
	cout << "Adapter: ";

	if (adaptee)
	{
		adaptee->specificRequest();
	}
	else
	{
		cout << "Empty!" << endl;
	}
}

# pragma endregion

int main()
{
	shared_ptr<BaseAdaptee> adaptee = make_shared<ConAdaptee>();
	shared_ptr<Adapter> adapter = make_shared<ConAdapter>(adaptee);

	adapter->request();
}
```

Пример 12.02. Шаблон адаптер (Adapter).

```
# include <iostream>
```

```
# include <memory>
# include <vector>

using namespace std;

class Interface
{
public:
    virtual ~Interface() = default;

    virtual void request() = 0;
};

template <typename Type>
class Adapter : public Interface
{
public:
    using MethodPtr = void (Type::*)();

    Adapter(shared_ptr<Type> o, MethodPtr m) : object(o), method(m) {}

    void request() override { ((*object).*method)(); }

private:
    shared_ptr<Type> object;
    MethodPtr method;
};

class AdapteeA
{
public:
    ~AdapteeA() { cout << "Destructor class AdapteeA;" << endl; }

    void specRequestA() { cout << "Method AdapteeA::specRequestA;" << endl; }
};

class AdapteeB
{
public:
    ~AdapteeB() { cout << "Destructor class AdapteeB;" << endl; }

    void specRequestB() { cout << "Method AdapteeB::specRequestB;" << endl; }
};

auto initialize()
{
    using InterPtr = shared_ptr<Interface>;

    vector<InterPtr> vec{
        make_shared<Adapter<AdapteeA>>(make_shared<AdapteeA>(), &AdapteeA::specRequestA),
        make_shared<Adapter<AdapteeB>>(make_shared<AdapteeB>(), &AdapteeB::specRequestB)
    };

    return vec;
}

int main()
{
    auto v = initialize();

    for (const auto& elem : v)
        elem->request();
}
```

Пример 12.03. Декоратор (Decorator).

```
# include <iostream>
```

```
# include <memory>

using namespace std;

class Component
{
public:
	virtual ~Component() = default;

	virtual void operation() = 0;
};

class ConComponent : public Component
{
public:
	void operation() override { cout << "ConComponent; "; }
};

class Decorator : public Component
{
protected:
	shared_ptr<Component> component;

public:
	Decorator(shared_ptr<Component> comp) : component(comp) {}
};

class ConDecorator : public Decorator
{
public:
	using Decorator::Decorator;

	void operation() override;
};

# pragma region Method
void ConDecorator::operation()
{
	if (component)
	{
		component->operation();

		cout << "ConDecorator; ";
	}

}

# pragma endregion

int main()
{
	shared_ptr<Component> component = make_shared<ConComponent>();
	shared_ptr<Component> decorator1 = make_shared<ConDecorator>(component);

	decorator1->operation();
	cout << endl;

	shared_ptr<Component> decorator2 = make_shared<ConDecorator>(decorator1);

	decorator2->operation();
	cout << endl;
}
```

Пример 12.04. Компоновщик (Composite).

```
# include <iostream>
# include <initializer_list>
# include <memory>
```

```cpp
# include <vector>

using namespace std;

class Component;

using PtrComponent = shared_ptr<Component>;
using VectorComponent = vector<PtrComponent>;

class Component
{
public:
	using value_type = Component;
	using size_type = size_t;
	using iterator = VectorComponent::const_iterator;
	using const_iterator = VectorComponent::const_iterator;

	virtual ~Component() = default;

	virtual void operation() = 0;

	virtual bool isComposite() const { return false; }
	virtual bool add(initializer_list<PtrComponent> comp) { return false; }
	virtual bool remove(const iterator& it) { return false; }
	virtual iterator begin() const { return iterator(); }
	virtual iterator end() const { return iterator(); }
};

class Figure : public Component
{
public:
	virtual void operation() override { cout << "Figure method;" << endl; }
};

class Camera : public Component
{
public:
	virtual void operation() override { cout << "Camera method;" << endl; }
};

class Composite : public Component
{
private:
	VectorComponent vec;

public:
	Composite() = default;
	Composite(PtrComponent first, ...);

	void operation() override;

	bool isComposite() const override { return true; }
	bool add(initializer_list<PtrComponent> list) override;
	bool remove(const iterator& it) override { vec.erase(it); return true; }
	iterator begin() const override { return vec.begin(); }
	iterator end() const override { return vec.end(); }
};

# pragma region Methods
Composite::Composite(PtrComponent first, ...)
{
	for (shared_ptr<Component>* ptr = &first; *ptr; ++ptr)
		vec.push_back(*ptr);
}

void Composite::operation()
{
	cout << "Composite method:" << endl;
	for (auto elem : vec)
```

```
            elem->operation();
}

bool Composite::add(initializer_list<PtrComponent> list)
{
	for (auto elem : list)
		vec.push_back(elem);

	return true;
}

# pragma endregion

int main()
{
	using Default = shared_ptr<Component>;
	PtrComponent fig = make_shared<Figure>(), cam = make_shared<Camera>();
	auto composite1 = make_shared<Composite>(fig, cam, Default{});

	composite1->add({ make_shared<Figure>(), make_shared<Camera>() });
	composite1->operation();
	cout << endl;

	auto it = composite1->begin();

	composite1->remove(++it);
	composite1->operation();
	cout << endl;

	auto composite2 = make_shared<Composite>(make_shared<Figure>(), composite1, Default());

	composite2->operation();
}
```

Пример 12.06. Мост (Bridge).

```
# include <iostream>
# include <memory>

using namespace std;

class Implementor
{
public:
	virtual ~Implementor() = default;

	virtual void operationImp() = 0;
};

class Abstraction
{
protected:
	shared_ptr<Implementor> implementor;

public:
	Abstraction(shared_ptr<Implementor> imp) : implementor(imp) {}
	virtual ~Abstraction() = default;

	virtual void operation() = 0;
};

class ConImplementor : public Implementor
{
public:
	virtual void operationImp() override { cout << "Implementor;" << endl; }
};

class Entity : public Abstraction
```

```
{
public:
	using Abstraction::Abstraction;

	virtual void operation() override { cout << "Entity: "; implementor->operationImp(); }
};

int main()
{
	shared_ptr<Implementor> implementor = make_shared<ConImplementor>();
	shared_ptr<Abstraction> abstraction = make_shared<Entity>(implementor);

	abstraction->operation();
}
```

Пример 12.05. Заместитель (Proxy).

```
# include <iostream>
# include <memory>
# include <map>
# include <random>

using namespace std;

class Subject
{
public:
	virtual ~Subject() = default;

	virtual pair<bool, double> request(size_t index) = 0;
	virtual bool changed() { return true; }
};

class RealSubject : public Subject
{
private:
	bool flag{ false };
	size_t counter{ 0 };

public:
	virtual pair<bool, double> request(size_t index) override;
	virtual bool changed() override;
};

class Proxy : public Subject
{
protected:
	shared_ptr<RealSubject> realsubject;

public:
	Proxy(shared_ptr<RealSubject> real) : realsubject(real) {}
};

class ConProxy : public Proxy
{
private:
	map<size_t, double> cache;

public:
	using Proxy::Proxy;

	virtual pair<bool, double> request(size_t index) override;
};

#pragma region Methods
bool RealSubject::changed()
{
	if (counter == 0)
```

```
	{
		flag = true;
	}
	if (++counter == 7)
	{
		counter = 0;
		flag = false;
	}
	return flag;
}


pair<bool, double> RealSubject::request(size_t index)
{
	random_device rd;
	mt19937 gen(rd());

	return pair<bool, double>(true, generate_canonical<double, 10>(gen));
}

pair<bool, double> ConProxy::request(size_t index)
{
	pair<bool, double> result;

	if (!realsubject)
	{
		cache.clear();

		result = pair<bool, double>(false, 0.);
	}
	else if (!realsubject->changed())
	{
		cache.clear();

		result = realsubject->request(index);

		cache.insert(map<size_t, double>::value_type(index, result.second));
	}
	else
	{
		map<size_t, double>::const_iterator it = cache.find(index);

		if (it != cache.end())
		{
			result = pair<bool, double>(true, it->second);
		}
		else
		{
			result = realsubject->request(index);

			cache.insert(map<size_t, double>::value_type(index, result.second));
		}
	}

	return result;
}
#pragma endregion

int main()
{
	shared_ptr<RealSubject> subject = make_shared<RealSubject>();
	shared_ptr<Subject> proxy = make_shared<ConProxy>(subject);

	for (size_t i = 0; i < 21; ++i)
	{
		cout << "( " << i + 1 << ", " << proxy->request(i % 3).second << " )" << endl;

		if ((i + 1) % 3 == 0)
			cout << endl;
```

```
        }
}
```