Пример 11.01. Фабричный метод (Factory Method). Новый объект.

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

using namespace std;

# pragma region Product
class Product
{
public:
	virtual ~Product() = default;

	virtual void run() = 0;
};

class ConProd1 : public Product
{
public:
	ConProd1() { cout << "Calling the ConProd1 constructor;" << endl; }
	~ConProd1() override { cout << "Calling the ConProd1 destructor;" << endl; }

	void run() override { cout << "Calling the run method;" << endl; }
};

# pragma endregion

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

	virtual unique_ptr<Product> createProduct() = 0;
};

template <typename Derived, typename Base>
concept Derivative = is_abstract_v<Base> && is_base_of_v<Base, Derived>;

template <Derivative<Product> Tprod>
class ConCreator : public Creator
{
public:
	unique_ptr<Product> createProduct() override
	{
		return unique_ptr<Product>(new Tprod());
	}
};

class User
{
public:
	void use(shared_ptr<Creator>& cr)
	{
		shared_ptr<Product> ptr = cr->createProduct();

		ptr->run();
	}
};

int main()
{
	shared_ptr<Creator> cr = make_shared<ConCreator<ConProd1>>();

	unique_ptr<User> us = make_unique<User>();

	us->use(cr);
}
```

Пример 11.02. Фабричный метод (Factory Method). Шаблонный creator.

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

using namespace std;

class Product;

template <typename Type>
concept NotAbstract = !is_abstract_v<Type>;

template <NotAbstract Tprod>
requires derived_from<Tprod, Product>
class Creator
{
public:
	unique_ptr<Product> createProduct()
	{
		return make_unique<Tprod>();
	}
};

# pragma region Product
class Product
{
public:
	virtual ~Product() = default;

	virtual void run() = 0;
};

class ConProd1 : public Product
{
public:
	ConProd1() { cout << "Calling the ConProd1 constructor;" << endl; }
	~ConProd1() override { cout << "Calling the ConProd1 destructor;" << endl; }

	void run() override { cout << "Calling the run method;" << endl; }
};

# pragma endregion

class User
{
public:
	template<NotAbstract Tprod>
	void use(shared_ptr<Creator<Tprod>> cr);
};

template<NotAbstract Tprod>
void User::use(shared_ptr<Creator<Tprod>> cr)
{
	shared_ptr<Product> ptr = cr->createProduct();

	ptr->run();
}

int main()
{
	shared_ptr<Creator<ConProd1>> cr(new Creator<ConProd1>());

	unique_ptr<User> us = make_unique<User>();

	us->use(cr);
}
```

Пример 11.03. Фабричный метод (Factory Method). Шаблонный базовый класс creator.

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

using namespace std;

# pragma region Product
class Product
{
public:
	virtual ~Product() = default;

	virtual void run() = 0;
};

class ConProd1 : public Product
{
private:
	int count;
	double price;

public:
	ConProd1(int c, double p) : count(c), price(p)
	{
		cout << "Calling the ConProd1 constructor;" << endl;
	}
	~ConProd1() override { cout << "Calling the ConProd1 destructor;" << endl; }

	void run() override { cout << "Count = " << count << "; Price = " << price << endl; }
};

class ConProd2// : public Product
{
public:
	ConProd2(int c, double p)
	{
		cout << "Calling the ConProd2 constructor;" << endl;
	}
	virtual ~ConProd2() { cout << "Calling the ConProd2 destructor;" << endl; }
	virtual void run() { cout << "Calling the run method ConProd2;" << endl; }
};

# pragma endregion

template <typename Type>
concept Abstract = is_abstract_v<Type>;

template <typename Type>
concept NotAbstract = !is_abstract_v<Type>;

template <typename Derived, typename Base>
concept Derivative = is_abstract_v<Base> && is_base_of_v<Base, Derived>;

# pragma region Variants of the concept Constructible
# define V_1

# ifdef V_1
template<typename Type, typename... Args>
concept Constructible = requires(Args... args)
{
	Type{ args... };
};

# elif defined(V_2)
template<typename Type, typename... Args>
concept Constructible = requires
{
	Type{ declval<Args>()... };
};
```

```
# elif defined(V_3)
template<typename Type, typename... Args>
concept Constructible = is_constructible_v<Type, Args...>;

# endif

# pragma endregion

template <Abstract Tbase, typename... Args>
class BaseCreator
{
public:
	virtual ~BaseCreator() = default;

	virtual unique_ptr<Tbase> create(Args&& ...args) = 0;
};

template <typename Tbase, typename Tprod, typename... Args>
requires NotAbstract<Tprod>&& Derivative<Tprod, Tbase>&& Constructible<Tprod, Args...>
class Creator : public BaseCreator<Tbase, Args...>
{
public:
	unique_ptr<Tbase> create(Args&& ...args) override
	{
		return make_unique<Tprod>(forward<Args>(args)...);
	}
};

using BaseCreator_t = BaseCreator<Product, int, double>;

class User
{
public:
	void use(shared_ptr<BaseCreator_t>& cr)
	{
		shared_ptr<Product> ptr = cr->create(1, 100.);

		ptr->run();
	}
};

int main()
{
	shared_ptr<BaseCreator_t> cr = make_shared<Creator<Product, ConProd1, int, double>>();

	unique_ptr<User> us = make_unique<User>();

	us->use(cr);
}
```

Пример 11.04. Фабричный метод (Factory Method). Без повторного создания.

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

using namespace std;

class Product;

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

	shared_ptr<Product> getProduct();

protected:
	virtual shared_ptr<Product> createProduct() = 0;
private:
	shared_ptr<Product> product;
};

template <derived_from<Product> Tprod>
class ConCreator : public Creator
{
protected:
	shared_ptr<Product> createProduct() override
	{
		return  make_shared<Tprod>();
	}
};

# pragma region Method Creator
shared_ptr<Product> Creator::getProduct()
{
	if (!product)
	{
		product = createProduct();
	}

	return product;
}

# pragma endregion


# pragma region Product
class Product
{
public:
	virtual ~Product() = default;

	virtual void run() = 0;
};

class ConProd1 : public Product
{
public:
	ConProd1() { cout << "Calling the ConProd1 constructor;" << endl; }
	~ConProd1() override { cout << "Calling the ConProd1 destructor;" << endl; }

	void run() override { cout << "Calling the run method;" << endl; }
};

# pragma endregion

int main()
{
	shared_ptr<Creator> cr = make_shared<ConCreator<ConProd1>>();
	shared_ptr<Product> ptr1 = cr->getProduct();
	shared_ptr<Product> ptr2 = cr->getProduct();

	cout << "use count = " << ptr1.use_count() << endl;
	ptr1->run();
}
```

Пример 11.05. Фабричный метод (Factory Method). Разделение обязанностей.

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

using namespace std;
```

```
class Product;

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

    virtual unique_ptr<Product> createProduct() = 0;
};

template <derived_from<Product> Tprod>
class ConCreator : public Creator
{
public:
    unique_ptr<Product> createProduct() override
    {
        return make_unique<Tprod>();
    }
};

# pragma region Product
class Product
{
public:
    virtual ~Product() = default;

    virtual void run() = 0;
};

class ConProd1 : public Product
{
public:
    ConProd1() { cout << "Calling the ConProd1 constructor;" << endl; }
    ~ConProd1() override { cout << "Calling the ConProd1 destructor;" << endl; }

    void run() override { cout << "Calling the run method ConProd1;" << endl; }
};

class ConProd2 : public Product
{
public:
    ConProd2() { cout << "Calling the ConProd2 constructor;" << endl; }
    ~ConProd2() override { cout << "Calling the ConProd2 destructor;" << endl; }

    void run() override { cout << "Calling the run method ConProd2;" << endl; }
};
# pragma endregion

class CrCreator
{
public:
    template <typename Tprod>
    static unique_ptr<Creator> createConCreator()
    {
        return make_unique<ConCreator<Tprod>>();
    }

};

class Solution
{
    using CreateCreator = unique_ptr<Creator>(&)();
    using CallBackMap = map<size_t, CreateCreator>;

public:
    Solution() = default;
    Solution(initializer_list<pair<size_t, CreateCreator>> list);

    bool registration(size_t id, CreateCreator createfun);
```

```
	bool check(size_t id) { return callbacks.erase(id) == 1; }

	unique_ptr<Creator> create(size_t id);

private:
	CallBackMap callbacks;
};

# pragma region Solution
Solution::Solution(initializer_list<pair<size_t, CreateCreator>> list)
{
	for (auto&& elem : list)
		this->registration(elem.first, elem.second);
}

bool Solution::registration(size_t id, CreateCreator createfun)
{
	return callbacks.insert(CallBackMap::value_type(id, createfun)).second;
}

unique_ptr<Creator> Solution::create(size_t id)
{
	CallBackMap::const_iterator it = callbacks.find(id);

	if (it == callbacks.end())
	{
		//			throw IdError();
	}

	return unique_ptr<Creator>(it->second());
}

# pragma endregion

int main()
{
	shared_ptr<Solution> solution(new Solution({ {1, CrCreator::createConCreator<ConProd1>} }));

	if (!solution->registration(2, CrCreator::createConCreator<ConProd2>))
	{
		cout << "Error registration!" << endl;
		// throw ...
	}
	else
	{
		solution->registration(2, CrCreator::createConCreator<ConProd2>);

		shared_ptr<Creator> cr(solution->create(2));
		shared_ptr<Product> ptr = cr->createProduct();

		ptr->run();
	}
}
```

Пример 11.06. Фабричный метод (Factory Method). «Статический полиморфизм» (CRTP).

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

using namespace std;

# pragma region Product
class Product
{
public:
	virtual ~Product() = default;

	virtual void run() = 0;
};

class ConProd1 : public Product
{
public:
	ConProd1() { cout << "Calling the ConProd1 constructor;" << endl; }
	~ConProd1() override { cout << "Calling the ConProd1 destructor;" << endl; }

	void run() override { cout << "Calling the run method;" << endl; }
};

# pragma endregion

template <typename Tcrt>
class Creator
{
public:
	auto create() const
	{
		return static_cast<const Tcrt*>(this)->create_impl();
	}
};

template <typename Tprod>
class ProductCreator : public Creator<ProductCreator<Tprod>>
{
public:
	unique_ptr<Product> create_impl() const
	{
		return make_unique<Tprod>();

//		return unique_ptr<Product>(new Tprod());
	}
};

template <typename Type>
concept Creatable = requires(Type t)
{
	t.create();
};

class Work
{
public:
	template <Creatable Type>
	auto create(const Type& crt)
	{
		return crt.create();
	}
};

int main()
{
	Creator<ProductCreator<ConProd1>> cr;

	auto product = Work{}.create(cr);

	product->run();
}
```

Пример 11.07. Использование паттерна «фабричный метод» для паттерна Command.

```
# include <iostream>
# include <functional>

using namespace std;

class Command;
class BaseCommandCreator
{
public:
    ~BaseCommandCreator() = default;

    virtual shared_ptr<Command> create_command() const = 0;
};

template <typename Tder, typename Tbase = Command>
concept Derived = is_base_of_v<Tbase, Tder>;

template <Derived<Command> Type>
class CommandCreator : public BaseCommandCreator
{
public:
    template <typename... Args>
    CommandCreator(Args ...args)
    {
        create_func = [args...]() { return make_shared<Type>(args...); };
    }
    ~CommandCreator() = default;

    shared_ptr<Command> create_command() const override
    {
        return create_func();
    }

private:
    function<shared_ptr<Command>()> create_func;
};

# pragma region Member_Function_Pointer
namespace MFP
{
    template <typename T>
    struct is_member_function_pointer_helper : std::false_type {};

    template <typename T, typename U>
    struct is_member_function_pointer_helper<T U::*> : std::is_function<T> {};

    template <typename T>
    struct is_member_function_pointer
        : is_member_function_pointer_helper< typename std::remove_cv<T>::type > {};

    template <typename T>
    inline constexpr bool is_member_function_pointer_v = is_member_function_pointer<T>::value;
}

# pragma endregion

# pragma region Command
class Command
{
public:
    virtual ~Command() = default;

    virtual void execute() = 0;
};

template <typename Reseiver>
requires is_class_v<Reseiver> && MFP::is_member_function_pointer_v<void (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)(); }
};

# pragma endregion

class Object
{
public:
    void operation() { cout << "Run method;" << endl; }
};

class Invoker
{
public:
    void run(shared_ptr<Command> com) { com->execute(); }
};

template <typename Type>
using SimpleComCreator = CommandCreator<SimpleCommand<Type>>;

int main()
{
    shared_ptr<Invoker> inv = make_shared<Invoker>();
    shared_ptr<Object> obj = make_shared<Object>();

    shared_ptr<BaseCommandCreator> cr
        = make_shared<SimpleComCreator<Object>>(obj, &Object::operation);

    shared_ptr<Command> com = cr->create_command();

    inv->run(com);
}
```

Пример 11.08. Абстрактная фабрика (Abstract Factory).

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

using namespace std;

class Image {};
class Color {};

class BaseGraphics
{
public:       virtual ~BaseGraphics() = 0;
};
BaseGraphics::~BaseGraphics() {}

class BasePen {};
class BaseBrush {};

class QtGraphics : public BaseGraphics
{
public:
       QtGraphics(shared_ptr<Image> im) { cout << "Calling the QtGraphics constructor;" << endl; }
       ~QtGraphics() override { cout << "Calling the QtGraphics destructor;" << endl; }
};

class QtPen : public BasePen {};
class QtBrush : public BaseBrush {};

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

       virtual unique_ptr<BaseGraphics> createGraphics(shared_ptr<Image> im) = 0;
       virtual unique_ptr<BasePen> createPen(shared_ptr<Color> cl) = 0;
       virtual unique_ptr<BaseBrush> createBrush(shared_ptr<Color> cl) = 0;
};

class QtGraphFactory : public AbstractGraphFactory
{
public:
       unique_ptr<BaseGraphics> createGraphics(shared_ptr<Image> im) override
       {
               return make_unique<QtGraphics>(im);
       }

       unique_ptr<BasePen> createPen(shared_ptr<Color> cl) override
       {
               return make_unique<QtPen>();
       }

       unique_ptr<BaseBrush> createBrush(shared_ptr<Color> cl) override
       {
               return make_unique<QtBrush>();
       }
};

int main()
{
       shared_ptr<AbstractGraphFactory> grfactory = make_shared<QtGraphFactory>();

       shared_ptr<Image> image = make_shared<Image>();

       shared_ptr<BaseGraphics> graphics1 = grfactory->createGraphics(image);
}
```

Пример 11.09. Строитель (Builder).

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

using namespace std;

class Product
{
public:
        Product() { cout << "Calling the ConProd1 constructor;" << endl; }
        ~Product() { cout << "Calling the ConProd1 destructor;" << endl; }

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

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

        virtual bool buildPart1() = 0;
        virtual bool buildPart2() = 0;

        shared_ptr<Product> getProduct();

protected:
        virtual shared_ptr<Product> createProduct() = 0;

        shared_ptr<Product> product;
};

class ConBuilder : public Builder
```

```
{
public:
        bool buildPart1() override
        {
                cout << "Completed part: " << ++part << ";" << endl;
                return true;
        }
        bool buildPart2() override
        {
                cout << "Completed part: " << ++part << ";" << endl;
                return true;
        }

protected:
        virtual shared_ptr<Product> createProduct() override;

private:
        size_t part{ 0 };
};

class Director
{
public:
        shared_ptr<Product> create(shared_ptr<Builder> builder)
        {
                if (builder->buildPart1() && builder->buildPart2()) return builder->getProduct();

                return shared_ptr<Product>();
        }
};

# pragma region Methods
shared_ptr<Product> Builder::getProduct()
{
        if (!product) { product = createProduct(); }

        return product;
}

shared_ptr<Product> ConBuilder::createProduct()
{
        if (part == 2) { product = make_shared<Product>(); }

        return product;
}
# pragma endregion

int main()
{
        shared_ptr<Builder> builder = make_shared<ConBuilder>();
        shared_ptr<Director> director = make_shared<Director>();

        shared_ptr<Product> prod = director->create(builder);

        if (prod)
                prod->run();
}
```

Пример 11.10. Прототип (Prototype).

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

using namespace std;

class BaseObject
{
public:
```

```cpp
    virtual ~BaseObject() = default;

    virtual unique_ptr<BaseObject> clone() = 0;
};

class Object1 : public BaseObject
{
public:
    Object1() { cout << "Calling the default constructor;" << endl; }
    Object1(const Object1& obj) { cout << "Calling the Copy constructor;" << endl; }
    ~Object1() override { cout << "Calling the destructor;" << endl; }

    unique_ptr<BaseObject> clone() override
    {
        return make_unique<Object1>(*this);
    }
};

int main()
{
    shared_ptr<BaseObject> ptr1 = make_shared<Object1>();

    auto ptr2 = ptr1->clone();
}
```

Пример 11.14. Прототип (Prototype). «Статический полиморфизм» (CRTP).

```cpp
# include <iostream>
# include <memory>
# include <concepts>

using namespace std;

struct Base_Obj
{
    virtual ~Base_Obj() = default;

    virtual unique_ptr<Base_Obj> clone() const = 0;
    virtual ostream& print(ostream& os) const = 0;
};

template <typename Type>
concept Abstract = is_abstract_v<Type>;

template <Abstract Base, typename Derived>
struct Clonable : public Base
{
    unique_ptr<Base> clone() const override
    {
        return make_unique<Derived>(static_cast<const Derived&>(*this));
    }
};

class Descendant : public Clonable<Base_Obj, Descendant>
{
private:
    int data;

public:
    Descendant(int d) : data(d) { cout << "Calling the constructor;" << endl; }
    Descendant(const Descendant& obj) : data(obj.data)
    { cout << "Calling the Copy constructor;" << endl; }
    ~Descendant() override { cout << "Calling the destructor;" << endl; }

    ostream& print(ostream& os) const override
    {
        return os << "data = " << data;
    }
};

// C++23
/*
template <typename Base>
struct Clonable : public Base
{
    template <typename Self>
    unique_ptr<Base> clone(this Selt&& self) const override
    {
        return unique_ptr<Base>(new Self(self));
    }
};

class Descendant : public Clonable<Base_Obj>
{
private:
    int data;

public:
    Descendant(int d) : data(d) {}

    ostream& print(ostream& os) const override
    {
        return os << "data = " << data;

    }
};
*/

ostream& operator <<(ostream& os, const unique_ptr<Base_Obj>& obj)
{
    return obj->print(os);
}

int main()
{
    unique_ptr<Base_Obj> v1 = make_unique<Descendant>(10);
    auto v2 = v1->clone();

    cout << v2 << endl;
}
```

Пример 11.11. Одиночка (Singleton).

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

using namespace std;

class Product
{
public:
        static shared_ptr<Product> instance()
        {
                class Proxy : public Product {};

                static shared_ptr<Product> myInstance = make_shared<Proxy>();

                return myInstance;
        }
        ~Product() { cout << "Calling the destructor;" << endl; }

        void f() { cout << "Method f;" << endl; }

        Product(const Product&) = delete;
        Product& operator =(const Product&) = delete;
private:
	Product() { cout << "Calling the default constructor;" << endl; }
};

int main()
{
	shared_ptr<Product> ptr(Product::instance());

	ptr->f();
}
```

Пример 11.12. Шаблон одиночка (Singleton).

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

using namespace std;

template <typename T>
concept NotAbstractClass = is_class_v<T> && !is_abstract_v<T>;

template <typename T>
concept CopyConstructible = requires(T t)
{
	T(t);
};

template <typename T>
concept Assignable = requires(T t1, T t2)
{
	t1 = t2;
};

template <typename T>
concept OnlyObject = NotAbstractClass<T> && !CopyConstructible<T> && !Assignable<T>;

template <OnlyObject Type>
class Singleton
{
private:
	static unique_ptr<Type> inst;

public:
	template <typename ...Args>
	static Type& instance(Args ...args)
	{
		struct Proxy : public Type
		{
			Proxy(Args&& ...args) : Type(forward<Args>(args)...) {}
		};

		if (!inst)
			inst = make_unique<Proxy>(forward<Args>(args)...);

		return *inst;
	}

	Singleton() = delete;
	Singleton(const Singleton&) = delete;
	Singleton& operator =(const Singleton&) = delete;
};

template <OnlyObject Type>
unique_ptr<Type> Singleton<Type>::inst{};

class Product
{
```

```
private:
	int num;
	double data;

protected:
	Product() = default;
	Product(int n, double d) : num(n), data(d)
	{
		cout << "Calling the constructor;" << endl;
	}

public:
	~Product() { cout << "Calling the destructor;" << endl; }

	void f() { cout << "num = " << num << "; data = " << data << endl; }

	Product(const Product&) = delete;
	Product& operator =(const Product&) = delete;
};

int main()
{
	decltype(auto) d1 = Singleton<Product>::instance(1, 2.);
	decltype(auto) d2 = Singleton<Product>::instance();

	d2.f();
}
```

Пример 11.13. Пул объектов (Object Pool).

```
# include <iostream>
# include <memory>
# include <iterator>
# include <vector>

using namespace std;

template <typename T>
concept PoolObject = requires(T t)
{
	t.clear();
};

class Product
{
private:
	static size_t count;

public:
	Product() { cout << "Constructor(" << ++count << ");" << endl; }
	~Product() { cout << "Destructor(" << count-- << ");" << endl; }

	void clear() { cout << "Method clear: 0x" << this << endl; }
};

size_t Product::count = 0;

template <PoolObject Type>
class Pool
{
public:
	static shared_ptr<Pool<Type>> instance();

	shared_ptr<Type> getObject();
	bool releaseObject(shared_ptr<Type>& obj);
	size_t count() const { return pool.size(); }

	Pool(const Pool&) = delete;
```

```
    Pool& operator =(const Pool&) = delete;

private:
    vector<pair<bool, shared_ptr<Type>>> pool;

    Pool() {}

    pair<bool, shared_ptr<Type>> create();

    template <typename Type>
    friend ostream& operator << (ostream& os, const Pool<Type>& pl);
};

# pragma region ObjectPool class Methods
template <PoolObject Type>
shared_ptr<Pool<Type>> Pool<Type>::instance()
{
    static shared_ptr<Pool<Type>> myInstance(new Pool<Type>());

    return myInstance;
}

template <PoolObject Type>
shared_ptr<Type> Pool<Type>::getObject()
{
    size_t i;
    for (i = 0; i < pool.size() && pool[i].first; ++i);

    if (i < pool.size())
    {
        pool[i].first = true;
    }
    else
    {
        pool.push_back(create());
    }

    return pool[i].second;
}

template <PoolObject Type>
bool Pool<Type>::releaseObject(shared_ptr<Type>& obj)
{
    size_t i;
    for (i = 0; pool[i].second != obj && i < pool.size(); ++i);

    if (i == pool.size()) return false;

    obj.reset();
    pool[i].first = false;
    pool[i].second->clear();

    return true;
}

template <PoolObject Type>
pair<bool, shared_ptr<Type>> Pool<Type>::create()
{
    return { true, make_shared<Type>() };
}

# pragma endregion

template <typename Type>
ostream& operator << (ostream& os, const Pool<Type>& pl)
{
    for (auto elem : pl.pool)
        os << "{" << elem.first << ", 0x" << elem.second << "} ";
    return os;
}

int main()
{
    shared_ptr<Pool<Product>> pool = Pool<Product>::instance();

    vector<shared_ptr<Product>> vec(4);

    for (auto& elem : vec)
        elem = pool->getObject();

    pool->releaseObject(vec[1]);

    cout << *pool << endl;

    shared_ptr<Product> ptr = pool->getObject();
    vec[1] = pool->getObject();

    cout << *pool << endl;
}
```