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.

# 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.

# 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:

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).

# 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;

}