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; }

virtual ~ConProd1() override { cout << "Calling the ConProd1 destructor;" << endl; }

virtual 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 Tprod>

class ConCreator : public Creator

{

public:

virtual 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(new ConCreator<ConProd1>());

unique\_ptr<User> us = make\_unique<User>();

us->use(cr);

}

Пример 11.11. Фабричный метод (Factory Method). Шаблонный базовый класс.

# 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;

}

virtual ~ConProd1() override { cout << "Calling the ConProd1 destructor;" << endl; }

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

};

#pragma endregion

template <typename Tbase, typename ...Args>

class BaseCreator

{

public:

virtual ~BaseCreator() = default;

virtual unique\_ptr<Tbase> create(Args ...args) = 0;

};

template <typename Tbase, typename Tprod, typename ...Args>

class Creator : public BaseCreator<Tbase, Args...>

{

public:

virtual unique\_ptr<Tbase> create(Args ...args) override

{

return unique\_ptr<Tbase>(new Tprod(args...));

}

};

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

class User

{

public:

void use(shared\_ptr<TbaseCreator>& cr)

{

shared\_ptr<Product> ptr = cr->create(1, 100.);

ptr->run();

}

};

int main()

{

shared\_ptr<TbaseCreator> cr(new Creator<Product, ConProd1, int, double>());

unique\_ptr<User> us = make\_unique<User>();

us->use(cr);

}

Пример 11.02. Фабричный метод (Factory Method). Без повторного создания.

# include <iostream>

# include <memory>

using namespace std;

class Product;

class Creator

{

public:

shared\_ptr<Product> getProduct();

protected:

virtual shared\_ptr<Product> createProduct() = 0;

private:

shared\_ptr<Product> product;

};

template <typename Tprod>

class ConCreator : public Creator

{

protected:

virtual shared\_ptr<Product> createProduct() override

{

return shared\_ptr<Product>(new 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; }

virtual ~ConProd1() override { cout << "Calling the ConProd1 destructor;" << endl; }

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

};

#pragma endregion

int main()

{

shared\_ptr<Creator> cr(new ConCreator<ConProd1>());

shared\_ptr<Product> ptr1 = cr->getProduct();

shared\_ptr<Product> ptr2 = cr->getProduct();

cout << ptr1.use\_count() << endl;

ptr1->run();

}

Пример 11.03. Фабричный метод (Factory Method). Разделение обязанностей.

# include <iostream>

# include <initializer\_list>

# include <memory>

# include <map>

using namespace std;

class Product;

class Creator

{

public:

virtual unique\_ptr<Product> createProduct() = 0;

};

template <typename Tprod>

class ConCreator : public Creator

{

public:

virtual unique\_ptr<Product> createProduct() override

{

return unique\_ptr<Product>(new 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; }

virtual ~ConProd1() override { cout << "Calling the ConProd1 destructor;" << endl; }

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

};

class ConProd2 : public Product

{

public:

ConProd2() { cout << "Calling the ConProd2 constructor;" << endl; }

virtual ~ConProd2() override { cout << "Calling the ConProd2 destructor;" << endl; }

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

};

#pragma endregion

class CrCreator

{

public:

template <typename Tprod>

static unique\_ptr<Creator> createConCreator()

{

return unique\_ptr<Creator>(new 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>} }));

solution->registration(2, &CrCreator::createConCreator<ConProd2>);

shared\_ptr<Creator> cr(solution->create(2));

shared\_ptr<Product> ptr = cr->createProduct();

ptr->run();

}

Пример 11.13. Фабричный метод (Factory Method). Разделение обязанностей.

# include <iostream>

# include <memory>

# include <map>

using namespace std;

class Product;

class AbstractCreator

{

public:

virtual ~AbstractCreator() = default;

virtual unique\_ptr<Product> createProduct() = 0;

};

template <typename Tprod>

class Creator : public AbstractCreator

{

public:

Creator() { cout << "Calling the Creator constructor;" << endl; }

virtual ~Creator() override { cout << "Calling the Creator destructor;" << endl; }

virtual 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; }

virtual ~ConProd1() override { cout << "Calling the ConProd1 destructor;" << endl; }

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

};

class ConProd2 : public Product

{

public:

ConProd2() { cout << "Calling the ConProd2 constructor;" << endl; }

virtual ~ConProd2() override { cout << "Calling the ConProd2 destructor;" << endl; }

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

};

#pragma endregion

class Solution

{

using CallBackMap = map<size\_t, shared\_ptr<AbstractCreator>> ;

public:

Solution() = default;

template <typename Tprod>

bool registration(size\_t id);

bool check(size\_t id) { return callbacks.erase(id) == 1; }

shared\_ptr<AbstractCreator> create(size\_t id);

private:

CallBackMap callbacks{};

};

#pragma region Solution

template <typename Tprod>

bool Solution::registration(size\_t id)

{

return callbacks.emplace(id, make\_shared<Creator<Tprod>>()).second;

}

shared\_ptr<AbstractCreator> Solution::create(size\_t id)

{

CallBackMap::const\_iterator it = callbacks.find(id);

if (it == callbacks.end())

{

// throw IdError();

}

return it->second;

}

#pragma endregion

int main()

{

unique\_ptr<Solution> solution = make\_unique<Solution>();

if (solution->registration<ConProd1>(1))

solution->registration<ConProd2>(2);

shared\_ptr<AbstractCreator> cr(solution->create(2));

shared\_ptr<Product> ptr = cr->createProduct();

ptr->run();

}

Пример 11.10. Шаблонный фабричный метод. Подмена с перекомпиляцией.

# include <iostream>

# include <memory>

using namespace std;

class Product;

template <typename Tprod>

class Creator

{

public:

unique\_ptr<Product> createProduct()

{

return unique\_ptr<Product>(new 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; }

virtual ~ConProd1() override { cout << "Calling the ConProd1 destructor;" << endl; }

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

};

#pragma endregion

class User

{

public:

template<typename Tprod>

void use(shared\_ptr<Creator<Tprod>> cr);

};

template<typename 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 11.12. 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; }

virtual ~ConProd1() override { cout << "Calling the ConProd1 destructor;" << endl; }

virtual 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 unique\_ptr<Product>(new Tprod());

}

};

class Work

{

public:

template <typename Type>

static auto create(const Type& crt)

{

return crt.create();

}

};

void main()

{

Creator<ProductCreator<ConProd1>> cr;

auto product = Work::create(cr);

product->run();

}

Пример 11.04. 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; }

virtual ~QtGraphics() override { cout << "Calling the QtGraphics destructor;" << endl; }

};

class QtPen : public BasePen {};

class QtBrush : public BaseBrush {};

class AbstractGraphFactory

{

public:

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

{

virtual unique\_ptr<BaseGraphics> createGraphics(shared\_ptr<Image> im)

{

return make\_unique<QtGraphics>(im);

}

virtual unique\_ptr<BasePen> createPen(shared\_ptr<Color> cl)

{

return make\_unique<QtPen>();

}

virtual unique\_ptr<BaseBrush> createBrush(shared\_ptr<Color> cl)

{

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 11.05. Строитель (Builder).

# include <iostream>

# include <memory>

using namespace std;

class Product

{

public:

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 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:

virtual bool buildPart1() override

{

cout << "Completed part: " << ++part << ";" << endl;

return true;

}

virtual 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 11.06. 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() { cout << "Calling the destructor;" << endl; }

virtual unique\_ptr<BaseObject> clone() override

{

return make\_unique<Object1>(\*this);

}

};

int main()

{

shared\_ptr<BaseObject> ptr1 = make\_shared<Object1>();

auto ptr2 = ptr1->clone();

}

Пример 11.07. Одиночка (Singleton).

# include <iostream>

# include <memory>

using namespace std;

class Product

{

public:

static shared\_ptr<Product> instance()

{

static shared\_ptr<Product> myInstance(new Product());

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 11.08. Шаблон одиночка (Singleton).

# include <iostream>

# include <memory>

using namespace std;

template <typename Type>

class Singleton

{

public:

template <typename ...Args>

static Type& instance(Args ...args)

{

static unique\_ptr<Type> myInstance = make\_unique<Type>(args...);

return \*myInstance;

}

Singleton() = delete;

Singleton(const Singleton<Type>&) = delete;

Singleton<Type>& operator=(const Singleton<Type>&) = delete;

};

class Product

{

private:

int num;

double data;

public:

Product(int n, double d) : num(n), data(d)

{

cout << "Calling the constructor;" << endl;

}

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

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

Product(const Product&) = delete;

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

};

int main()

{

decltype(auto) d = Singleton<Product>::instance(1, 2.);

d.f();

}

Пример 09.9. Пул объектов (Object Pool).

# include <iostream>

# include <memory>

# include <iterator>

# include <vector>

using namespace std;

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 <typename Type>

class ObjectPool

{

public:

static shared\_ptr<ObjectPool<Type>> instance();

shared\_ptr<Type> getObject();

bool releaseObject(shared\_ptr<Type>& obj);

size\_t count() const { return pool.size(); }

iterator<output\_iterator\_tag, const pair<bool, shared\_ptr<Type>>> begin() const;

iterator<output\_iterator\_tag, const pair<bool, shared\_ptr<Type>>> end() const;

ObjectPool(const ObjectPool<Type>&) = delete;

ObjectPool<Type>& operator=(const ObjectPool<Type>&) = delete;

private:

vector<pair<bool, shared\_ptr<Type>>> pool;

ObjectPool() {}

pair<bool, shared\_ptr<Type>> create();

template <typename Type>

friend ostream& operator << (ostream& os, const ObjectPool<Type>& pl);

};

#pragma region ObjectPool class Methods

template <typename Type>

shared\_ptr<ObjectPool<Type>> ObjectPool<Type>::instance()

{

static shared\_ptr<ObjectPool<Type>> myInstance(new ObjectPool<Type>());

return myInstance;

}

template <typename Type>

shared\_ptr<Type> ObjectPool<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 <typename Type>

bool ObjectPool<Type>::releaseObject(shared\_ptr<Type>& obj)

{

size\_t i;

for (i = 0; i < pool.size() && pool[i].second != obj; ++i);

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

obj.reset();

pool[i].first = false;

pool[i].second->clear();

return true;

}

template <typename Type>

pair<bool, shared\_ptr<Type>> ObjectPool<Type>::create()

{

return pair<bool, shared\_ptr<Type>>(true, make\_shared<Type>());

}

#pragma endregion

template <typename Type>

ostream& operator << (ostream& os, const ObjectPool<Type>& pl)

{

for (auto elem : pl.pool)

os << "{" << elem.first << ", 0x" << elem.second << "} ";

return os;

}

int main()

{

shared\_ptr<ObjectPool<Product>> pool = ObjectPool<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;

}