Пример 09.1. Фабричный метод (Factory Method). Новый объект.

# include <iostream>

# include <memory>

using namespace std;

class Product;

class Creator

{

public:

virtual ~Creator() = 0;

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

};

Creator::~Creator() = default;

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() = 0;

virtual void run() = 0;

};

Product::~Product() = default;

class ConProd1 : public Product

{

public:

virtual ~ConProd1() override { cout << "Destructor;" << endl; }

virtual void run() override { cout << "Method run;" << endl; }

};

#pragma endregion

class User

{

public:

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

};

void User::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 09.2. Фабричный метод (Factory Method). Без повторного создания.

# include <iostream>

# include <memory>

using namespace std;

class Product;

class Creator

{

public:

virtual ~Creator() = 0;

shared\_ptr<Product> getProduct();

protected:

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

private:

shared\_ptr<Product> product;

};

Creator::~Creator() = default;

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() = 0;

virtual void run() = 0;

};

Product::~Product() {}

class ConProd1 : public Product

{

public:

virtual ~ConProd1() override { cout << "Destructor;" << endl; }

virtual void run() override { cout << "Method run;" << 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 09.3. Фабричный метод (Factory Method). Разделение обязанностей.

# include <iostream>

# include <memory>

# include <map>

using namespace std;

class Product;

class Creator

{

public:

virtual ~Creator() = 0;

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

};

Creator::~Creator() = default;

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() = 0;

virtual void run() = 0;

};

Product::~Product() {}

class ConProd1 : public Product

{

public:

virtual ~ConProd1() override { cout << "Destructor;" << endl; }

virtual void run() override { cout << "Method run;" << endl; }

};

#pragma endregion

unique\_ptr<Creator> createConCreator()

{

return unique\_ptr<Creator>(new ConCreator<ConProd1>());

}

class Solution

{

public:

typedef unique\_ptr<Creator> (\*CreateCreator)();

bool registration(size\_t id, CreateCreator createfun)

{

return callbacks.insert(CallBackMap::value\_type(id, createfun)).second;

}

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

unique\_ptr<Creator> create(size\_t id)

{

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

if (it == callbacks.end())

{

// throw IdError();

}

return unique\_ptr<Creator>((it->second)());

}

private:

using CallBackMap = map<size\_t, CreateCreator>;

CallBackMap callbacks;

};

int main()

{

Solution solution;

solution.registration(1, createConCreator);

shared\_ptr<Creator> cr(solution.create(1));

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

ptr->run();

}

Пример 09.31. 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() = 0;

virtual void run() = 0;

};

Product::~Product() = default;

class ConProd1 : public Product

{

public:

virtual ~ConProd1() override { cout << "Destructor;" << endl; }

virtual void run() override { cout << "Method run;" << 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 09.4. 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 << "Constructor QtGraphics;" << endl; }

virtual ~QtGraphics() override { cout << "Destructor QtGraphics;" << 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 unique\_ptr<BaseGraphics>(new QtGraphics(im)); }

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

{ return unique\_ptr<BasePen>(new QtPen()); }

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

{ return unique\_ptr<BaseBrush>(new QtBrush()); }

};

int main()

{

shared\_ptr<AbstractGraphFactory> grfactory(new QtGraphFactory());

shared\_ptr<BaseGraphics> graphics1 = grfactory->createGraphics(shared\_ptr<Image>(new Image()));

shared\_ptr<BaseGraphics> graphics2 = grfactory->createGraphics(shared\_ptr<Image>(new Image()));

}

Пример 09.5. Строитель (Builder).

# include <iostream>

# include <memory>

using namespace std;

class Product

{

public:

Product() { cout << "Default constructor;" << endl; }

~Product() { cout << "Destructor;" << endl; }

void run() { cout << "Method run;" << 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:

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 = shared\_ptr<Product>(new Product()); }

return product;

}

#pragma endregion

int main()

{

shared\_ptr<Builder> builder(new ConBuilder());

shared\_ptr<Director> director(new Director());

shared\_ptr<Product> prod = director->create(builder);

if (prod)

prod->run();

}

Пример 09.6. 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 << "Default constructor;" << endl; }

Object1(const Object1& obj) { cout << "Copy constructor;" << endl; }

~Object1() { cout << "Destructor;" << endl; }

virtual unique\_ptr<BaseObject> clone() override

{

return unique\_ptr<BaseObject>(new Object1(\*this));

}

};

int main()

{

unique\_ptr<BaseObject> ptr1(new Object1());

auto ptr2 = ptr1->clone();

}

Пример 09.7. 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 << "Destructor;" << endl; }

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

Product(const Product&) = delete;

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

private:

Product() { cout << "Default constructor;" << endl; }

};

int main()

{

shared\_ptr<Product> ptr(Product::instance());

ptr->f();

}

Пример 09.8. Шаблон одиночка (Singleton).

# include <iostream>

# include <memory>

using namespace std;

template <typename Type>

class Singleton

{

public:

static Type& instance()

{

static unique\_ptr<Type> myInstance(new Type());

return \*myInstance;

}

Singleton() = delete;

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

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

};

class Product

{

public:

Product() { cout << "Default constructor;" << endl; }

~Product() { cout << "Destructor;" << endl; }

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

};

int main()

{

Product& d = Singleton<Product>::instance();

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, shared\_ptr<Type>(new 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;

}