Пример 09.01. Выбор шаблона (специализации) подстановкой параметров (инстанцирование).

```
# include <iostream>

using namespace std;

# define PRIM_5

# ifdef PRIM_1
template <typename T, bool>
struct PrintHelper
{
    static void print(const T& t) { cout << t << endl; }
};

template <typename T>
struct PrintHelper<T, true>
{
    static void print(const T& t) { cout << *t << endl; }
};

template <typename T>
void print(const T& t)
{
    PrintHelper<T, is_pointer_v<T>>::print(t);
}

# elif defined(PRIM_2)
template <typename T>
void printHelper(false_type, const T& t) { cout << t << endl; }

template <typename T>
void printHelper(true_type, const T& t) { cout << *t << endl; }

template <typename T>
void print(const T& t)
{
    printHelper(typename is_pointer<T>::type{}, t);
}

# elif defined(PRIM_3)
template <typename T>
void print(const T& t)
{
    if constexpr (is_pointer_v<T>)
    {
        cout << *t << endl;
    }
    else
    {
        cout << t << endl;
    }
}

# elif defined(PRIM_4)
void print(auto& t)
{
    cout << t << endl;
}

void print(auto* t)
{
    cout << *t << endl;
}

# elif defined(PRIM_5)
void print(const auto& t)
{
    cout << t << endl;
}

template <typename T>
```

```
concept pointer = is_pointer_v<T>;

void print(const pointer auto& t)
{
    cout << *t << endl;
}

# endif


int main()
{
    double d = 1.5;

    print(d);
    print(&d);
}
```

Пример 09.02. Использование простого концепта.

```
# include <iostream>
# include <vector>

using namespace std;

template <typename T>
concept HasBeginEnd = requires(T a)
{
    a.begin();
    a.end();
};

# define PRIM_1

# ifdef PRIM_1
template <typename T>
requires HasBeginEnd<T>
ostream& operator <<(ostream& out, const T& v)
{
    for (const auto& elem : v)
        out << elem << endl;

    return out;
}

# elif defined(PRIM_2)
template <HasBeginEnd T>
ostream& operator <<(ostream& out, const T& v)
{
    for (const auto& elem : v)
        out << elem << endl;

    return out;
}

# elif defined(PRIM_3)
ostream& operator <<(ostream& out, const HasBeginEnd  auto& v)
{
    for (const auto& elem : v)
        out << elem << endl;

    return out;
}

# endif

int main()
{
    vector<double> v{ 1., 2., 3., 4., 5. };

    cout << v;
```

```
}
```

Пример 09.03. Варианты использования концепта.

```
# include <iostream>

using namespace std;

template <typename T>
concept Incrementable = requires(T t)
{
    {++t} noexcept;
    t++;
};

# define PRIM_1

# ifdef PRIM_1
template <typename T>
requires Incrementable<T>
auto inc(T& arg)
{
    return ++arg;
}

# elif defined(PRIM_2)
template <typename T>
auto inc(T& arg) requires Incrementable<T>
{
    return ++arg;
}

# elif defined(PRIM_3)
template <Incrementable T>
auto inc(T& arg)
{
    return ++arg;
}

# elif defined(PRIM_4)
auto inc(Incrementable auto& arg)
{
    return ++arg;
}

# elif defined(PRIM_5)
template <typename T>
requires requires(T t)
{
    {++t} noexcept;
    {t++};
}
auto inc(T& arg)
{
    return ++arg;
}

# endif

class A {};

int main()
{
    int i = 0;

    cout << "i = " << inc(i) << endl;

    A obj{};

    //    cout << "obj = " << inc(obj) << endl;
}
```

Пример 09.04. Концепты с составными ограничениями по типу выражений.

```
# include <iostream>

using namespace std;

# define PRIM_3

#ifdef PRIM_1
template <typename T, typename U, typename = void>
struct is_equal_comparable : false_type {};

template <typename T, typename U>
struct is_equal_comparable<T, U,
    void_t<decltype(declval<T>() == declval<U>())>> : true_type {};

template <typename T, typename U>
requires is_equal_comparable<T, U>::value
bool ch_equal(T&& lhs, U&& rhs)
{
    return lhs == rhs;
}

# elif defined(PRIM_2)
template <typename T, typename U>
requires requires(T t, U u) { t == u; }
bool ch_equal(T&& lhs, U&& rhs)
{
    return lhs == rhs;
}

# elif defined(PRIM_3)
template <typename T, typename U>
concept WeaklyEquialityComparable = requires(T t, U u)
{
    { t == u } -> convertible_to<bool>;
    { t != u } -> convertible_to<bool>;
};

template <typename T>
concept EquialityComparable = WeaklyEquialityComparable<T, T>;

template <typename T>
concept StrictTotallyOrdered = EquialityComparable<T> &&
requires(const remove_reference_t<T>&t1, const remove_reference_t<T>&t2)
{
    { t1 < t2 } -> convertible_to<bool>;
    { t1 > t2 } -> convertible_to<bool>;
};

template <typename T, typename U>
requires WeaklyEquialityComparable<T, U>
bool ch_equal(T&& lhs, U&& rhs)
{
    return (lhs <=> rhs) == 0;
}

#endif

int main()
{
    cout << boolalpha << ch_equal(3., 1) << endl;
    //    cout << ch_equal(" ", 1) << endl;
}
```

Пример 09.05. Использование концепта с несколькими параметрами.

```
# include <iostream>
```

```
using namespace std;

template <typename BI, typename EI>
concept Comparable = requires(BI bi, EI ei)
{
    { bi == ei } -> convertible_to<bool>;
};

# define VARIANT_3

# ifdef VARIANT_1
template <typename EI, Comparable<EI> BI>
constexpr bool my_equal(BI bi, EI ei)
{
    return bi == ei;
}

# elif defined(VARIANT_2)
template <typename BI, typename EI>
requires Comparable<BI, EI>
constexpr bool my_equal(BI bi, EI ei)
{
    return bi == ei;
}

# elif defined(VARIANT_3)
template <typename BI, typename EI>
constexpr bool my_equal(BI bi, EI ei) requires Comparable<BI, EI>
{
    return bi == ei;
}

# endif

class A
{
public:
    bool operator ==(const A&) { return true; }
};

class B
{
public:
    operator A() { return A{}; }
};

int main()
{
    cout << boolalpha << my_equal(1., 1) << endl;

    A objA{};
    B objB{};

    cout << boolalpha << my_equal(objA, objB) << endl;
}
```

Пример 09.06. Концепт с вариативным количеством параметров.

```
# include <iostream>
# include <concepts>

using namespace std;

template <typename Type, typename... Types>
constexpr inline bool are_same_v = conjunction_v<is_same<Type, Types>...>;

# define PRIM_3

# ifdef PRIM_1
template <typename Type, typename... Types>
requires (is_same_v<Type, Types> && ... && true)
```

```
auto sum(Type&& value, Types&&... params)
{
    return forward<Type>(value) + (... + forward<Types>(params));
}

# elif defined(PRIM_2)

template <typename... Types>
requires are_same_v<Types...>
auto sum(Types&&... params)
{
    return (... + forward<Types>(params));
}

# elif defined(PRIM_3)
template <typename Type, typename... Types>
struct first_arg { using type = Type; };

template <typename... Types>
using first_arg_t = typename first_arg<Types...>::type;

template <typename... Types>
concept Addable = requires(Types&&... args)
{
    {(... + forward<Types>(args))} -> same_as<first_arg_t<Types...>>;

    requires are_same_v<Types...>;
    requires sizeof...(Types) > 1;
};

template <typename... Types>
requires Addable<Types...>
auto sum(Types&&... args)
{
    return (... + forward<Types>(args));
}

# endif

int main()
{
    cout << sum(1., 2., 3., 4., 5.) << endl;
}
```

Пример 09.07. Концепты и инстанцирование.

```
# include <iostream>
# include <vector>

template <typename T> constexpr bool is_vector = false;
template <typename T> constexpr bool is_vector<std::vector<T>> = true;
template <typename T>
concept Vec = is_vector<T>;

template <typename T> constexpr bool is_pointer = false;
template <typename T> constexpr bool is_pointer<T*> = true;
template <typename T>
concept Ptr = is_pointer<T>;

template <typename T>
void f(T)
{
    std::cout << "def" << std::endl;
}

template <Ptr T>
void f(T t)
{
    std::cout << "ptr" << std::endl;
    f(*t);
}
```

```
template <Vec T>
void f(T t)
{
    std::cout << "vec" << std::endl;
    f(t[0]);
}

int main()
{
    std::vector v{ 1 };

    auto pv = &v;
    auto ppv = &pv;

    std::vector vv{ { v } };
    std::vector vvv{ { vv } };

    f(ppv);
    std::cout << std::endl;

    f(v);
    std::cout << std::endl;

    f(vvv);
}
```

Пример 09.08. Инстанцирование шаблонов методов класса.

```
# include <iostream>

using namespace std;

class A
{
public:
    template <typename Type>
    requires is_floating_point<Type>::value
    A(Type t)
    {
        cout << "Creating float object" << endl;
    }
    template <typename Type>
    requires is_integral<Type>::value
    A(Type t)
    {
        cout << "Creating integer object" << endl;
    }
};

int main()
{
    A obj(1.);
}
```

Пример 09.09. Ограничения для шаблонов классов, использование дизъюнкции.

```
# include <iostream>

using namespace std;

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

template <typename T>
concept Void = is_same_v<T, void>;

template <typename T = void>
requires Ord<T> || Void<T>
struct Less;
```

```
template <Ord T>
struct Less<T>
{
    bool operator ()(T a, T b) const { return a < b; }
};

template <>
struct Less<void>
{
    template <Ord T>
    bool operator ()(T& a, T& b) const { return &a < &b; }
};

int main()
{
    Less<double> d1;

    cout << boolalpha << d1(2., 3.) << endl;

    Less d2;

    int a = 0, b = 1;
    cout << boolalpha << d2(a, b) << endl;
}
```

Пример 09.10. Шаблон Holder.

```
# include <iostream>

using namespace std;

template <typename Type>
class Holder
{
private:
    Type* ptr{ nullptr };

public:
    Holder() = default;
    explicit Holder(Type* p) : ptr(p) {}
    Holder(Holder&& other) noexcept
    {
        ptr = other.ptr;
        other.ptr = nullptr;
    }
    ~Holder() { delete ptr; }

    Type* operator ->() noexcept { return ptr; }
    Type& operator *() noexcept { return *ptr; }
    operator bool() noexcept { return ptr != nullptr; }

    Type* release() noexcept
    {
        Type* work = ptr;
        ptr = nullptr;

        return work;
    }

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

class A
{
public:
    void f() { cout << "Function f of class A is called" << endl; }
};

int main()
```

```
{
    Holder<A> obj(new A{});

    obj->f();
}
```

Пример 09.11. Применение unique_ptr.

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

using namespace std;

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

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

int main()
{
    unique_ptr<A> obj1(new A{});
    unique_ptr<A> obj2 = make_unique<A>();
    unique_ptr<A> obj3(obj1.release()); // move(obj1)

    obj1 = move(obj3);

    if (!obj3)
    {
        A* p = obj1.release();

        obj2.reset(p);
        obj2->f();
    }
}
```

Пример 09.12. Установка deleter для unique_ptr на примере закрытия файла.

```
# include <iostream>
# include <memory>
# include <stdio.h>

using namespace std;

# define V_1

# ifdef V_1
class Deleter
{
public:
      void operator ()(FILE* stream) noexcept
      {
            fclose(stream);
            cout << "file is closed" << endl;
      }
};

# elif defined(V_2)
using Deleter = decltype([](FILE* stream)
      {
            fclose(stream);
            cout << "file is closed" << endl;
      });

# endif

using FilePtr_t = unique_ptr< FILE, Deleter >;
```

```
FilePtr_t make_file(const char* filename, const char* mode)
{
	FILE* stream = fopen(filename, mode);

	if (!stream) throw runtime_error("File not found!");

	cout << "file is open" << endl;

	return FilePtr_t{ stream };
}

int main()
{
	try
	{
		FilePtr_t stream = make_file("test.txt", "r");
	}
	catch (const runtime_error& ex)
	{
		cout << ex.what() << endl;
	}
}
```

Пример 09.13. Утечка памяти при использовании shared_ptr.

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

using namespace std;

class Base
{
protected:
	string name;

public:
	Base(const string& nm) : name(nm) {}

	void print(const string& nm)
	{
		cout << name << " now points to " << nm << endl;
	}
};

class BadWidget : public Base
{
private:
	shared_ptr<BadWidget> otherWidget;

public:
	BadWidget(const string& n) : Base(n)
	{
		cout << "BadWidget " << name << endl;
	}
	~BadWidget() { cout << "~BadWidget " << name << endl; }

	void setOther(const shared_ptr<BadWidget>& x)
	{
		otherWidget = x;
		print(x->name);
	}
};

class GoodWidget : public Base
{
private:
	weak_ptr<GoodWidget> otherWidget;

public:
```

```
        GoodWidget(const string& n) : Base(n)
        {
                cout << "GoodWidget " << name << endl;
        }
        ~GoodWidget() { cout << "~GoodWidget " << name << endl; }

        void setOther(const shared_ptr<GoodWidget>& x)
        {
                otherWidget = x;
                print(x->name);
        }
};

int main()
{
        { // В этом примере происходит утечка памяти
                cout << "Example 1" << endl;
                shared_ptr<BadWidget> w1 = make_shared<BadWidget>("1 First");
                shared_ptr<BadWidget> w2 = make_shared<BadWidget>("1 Second");
                w1->setOther(w2);
                w2->setOther(w1);
        }
        { // А в этом примере использован weak_ptr и утечки памяти не происходит
                cout << "Example 2" << endl;
                shared_ptr<GoodWidget> w1 = make_shared<GoodWidget>("2 First");
                shared_ptr<GoodWidget> w2 = make_shared<GoodWidget>("2 Second");
                w1->setOther(w2);
                w2->setOther(w1);
        }
        return 0;
}
```

Пример 09.14. Возврат shared_ptr на себя.

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

using namespace std;

class A : public enable_shared_from_this<A>
{
public:
    A() { cout << "Constructor" << endl; }
    ~A() { cout << "Destructor" << endl; }
    shared_ptr<A> getptr()
    {
        return shared_from_this();
    }
};

int main()
{
    try
    {
        shared_ptr<A> obj1 = make_shared<A>();
        shared_ptr<A> obj2 = obj1->getptr();
        cout << "good1.use_count() = " << obj1.use_count() << endl;

        A obj;
        shared_ptr<A> gp = obj.getptr();
    }
    catch (const bad_weak_ptr& e)
    {
        cout << e.what() << endl;
    }
}
```

Пример 09.15. Возврат shared_ptr на член данное объекта.

```
# include <iostream>
```

```
# include <memory>

using namespace std;

template <typename Type>
class Node : public enable_shared_from_this<Node<Type>>
{
private:
    shared_ptr<Node> nt;
    Type data;

    Node(shared_ptr<Node> nxt, Type d) : nt(nxt), data(d) {}

public:
    Node(const Node&) = delete;
    Node(Node&&) = delete;

    template <typename... Args>
    static shared_ptr<Node> create(Args&&... params);
    shared_ptr<Node> next();
    shared_ptr<Type> get();
};

# pragma region Method
template <typename Type>
template <typename... Args>
shared_ptr<Node<Type>> Node<Type>::create(Args&&... params)
{
    struct Enable_make_shared : public Node<Type>
    {
        Enable_make_shared(Args&&... params) : Node<Type>(forward<Args>(params)...) {}
    };

    return make_shared<Enable_make_shared>(forward<Args>(params)...);
}

template <typename Type>
shared_ptr<Node<Type>> Node<Type>::next()
{
    return nt;
}

template <typename Type>
shared_ptr<Type> Node<Type>::get()
{
    shared_ptr<Node> work = this->shared_from_this();

    return { work, &work->data };
}

# pragma endregion

int main()
{
    shared_ptr<double> value;

    {
        auto nd = Node<double>::create(nullptr, 10.);

        value = nd->get();
    }

    if (value.use_count() == 0)
        cout << "empty" << endl;
    else
        cout << "value = " << *value << endl;
}
```

Пример 09.16. Реализация хранителя unique_ptr.

```
# include <iostream>
```

```cpp
using namespace std;

template <typename Type>
struct DefaultDelete
{
	DefaultDelete() = default;
	DefaultDelete(const DefaultDelete&) = default;

	void operator()(Type* ptr) const { delete ptr; }
};

template <typename Type, typename Deleter = DefaultDelete<Type>>
class UniquePtr
{
public:
	UniquePtr() = default;
	constexpr UniquePtr(nullptr_t) {}
	explicit UniquePtr(Type* p) noexcept : ptr(p) {}
	UniquePtr(UniquePtr&& vright) noexcept;
	~UniquePtr() { Deleter{}(ptr); }

	UniquePtr& operator =(nullptr_t) noexcept;
	UniquePtr& operator =(UniquePtr&& vright) noexcept;

	Type& operator*() const noexcept { return *ptr; }
	Type* const operator->() const noexcept { return ptr; }
	Type* get() const noexcept { return ptr; }
	explicit operator bool() const noexcept { return ptr != nullptr; }

	Type* release() noexcept;
	void reset(Type* p = nullptr) noexcept;
	void swap(UniquePtr& other) noexcept;

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

private:
	Type* ptr{ nullptr };
};

# pragma region Method UniquePtr
template <typename Type, typename Deleter>
UniquePtr<Type, Deleter>::UniquePtr(UniquePtr&& vright) noexcept : ptr(vright.release()) {}

template <typename Type, typename Deleter>
UniquePtr<Type, Deleter>& UniquePtr<Type, Deleter>::operator =(nullptr_t) noexcept
{
	reset();

	return *this;
}

template <typename Type, typename Deleter>
UniquePtr<Type, Deleter>& UniquePtr<Type, Deleter>::operator =(UniquePtr&& vright) noexcept
{
	swap(vright);

	return *this;
}

template <typename Type, typename Deleter>
Type* UniquePtr<Type, Deleter>::release() noexcept
{
	Type* p = ptr;
	ptr = nullptr;

	return p;
}

template <typename Type, typename Deleter>
void UniquePtr<Type, Deleter>::reset(Type* p) noexcept
```

```
{
	Deleter{}(ptr);
	ptr = p;
}

template <typename Type, typename Deleter>
void UniquePtr<Type, Deleter>::swap(UniquePtr& other) noexcept
{
	::swap(ptr, other.ptr);
}

template <typename Type, typename... Args>
UniquePtr<Type> makeUnique(Args&&... params)
{
	return UniquePtr<Type>(new Type(forward<Args>(params)...));
}

namespace Unique
{
	template <typename Type>
	UniquePtr<Type> move(const UniquePtr<Type>& unique)
	{
		return UniquePtr<Type>(const_cast<UniquePtr<Type>&>(unique).release());
	}
}

# pragma endregion

class A
{
	int key;
public:
	A(int k) : key(k)
	{
		cout << "Calling the constructor of class A (obj" << key << ");" << endl;
	}
	~A()
	{
		cout << "Calling a class A destructor (obj" << key << ");" << endl;
	}

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

int main()
{
	UniquePtr<A> obj1(new A(1));
	UniquePtr<A> obj2 = makeUnique<A>(2);
	UniquePtr<A> obj3(obj1.release());

	obj2->f();
	(*obj2).f();

	obj1 = Unique::move(obj3);

	if (!obj3)
	{
		obj2.reset(obj1.release());
		obj2->f();
	}

	obj1.swap(obj2);
}
```

Пример 09.17. Реализация shared_ptr и weak_ptr.

```
# include <iostream>

using namespace std;

template <typename Type>
```

```cpp
class UniquePtr;

template <typename Type>
class SharedPtr;

template <typename Type>
class WeakPtr;

struct Count
{
	long countS{ 0 };
	long countW{ 0 };

	Count(long cS = 1, long cW = 0) noexcept : countS(cS), countW(cW) {}
};

template <typename Type>
class Pointers
{
public:
	long use_count() const noexcept { return rep ? rep->countS : 0; }

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

protected:
	Pointers() = default;

	Type* get() const noexcept { return ptr; }
	void set(Type* p, Count* r) noexcept { ptr = p; rep = r; }

	void delShared() noexcept;
	void delWeak() noexcept;
	void delCount() noexcept;

	bool _compare(const Pointers& right) const noexcept { return ptr == right.ptr; }
	void _swap(Pointers& right) noexcept
	{
		std::swap(ptr, right.ptr);
		std::swap(rep, right.rep);
	}
	void _copyShared(const Pointers& right) noexcept;
	void _copyWeak(const Pointers& right) noexcept;
	void _move(Pointers& right) noexcept;

private:
	Type* ptr{ nullptr };
	Count* rep{ nullptr };
};

#pragma region Method Pointers

template <typename Type>
void Pointers<Type>::delShared() noexcept
{
	if (!ptr) return;

	(rep->countS)--;

	if (!rep->countS)
	{
		delete ptr;
		ptr = nullptr;
		delCount();
	}
}

template <typename Type>
void Pointers<Type>::delWeak() noexcept
{
	if (rep)
	{
```

```cpp
		(rep->countW)--;
		delCount();
	}
}

template <typename Type>
void Pointers<Type>::delCount() noexcept
{
	if (!rep->countS && !rep->countW)
	{
		delete rep;
		rep = nullptr;
	}
}

template <typename Type>
void Pointers<Type>::_copyShared(const Pointers<Type>& right) noexcept
{
	if (right.ptr)
		(right.rep->countS)++;

	ptr = right.ptr;
	rep = right.rep;
}

template <typename Type>
void Pointers<Type>::_copyWeak(const Pointers<Type>& right) noexcept
{
	if (right.rep)
		(right.rep->countW)++;

	ptr = right.ptr;
	rep = right.rep;
}

template <typename Type>
void Pointers<Type>::_move(Pointers<Type>& right) noexcept
{
	ptr = right.ptr;
	rep = right.rep;

	right.ptr = nullptr;
	right.rep = nullptr;
}

#pragma endregion


template <typename Type>
class SharedPtr : public Pointers<Type>
{
public:
	SharedPtr() = default;
	explicit SharedPtr(Type* p);
	constexpr SharedPtr(nullptr_t) noexcept {}
	SharedPtr(const SharedPtr& other) noexcept;
	explicit SharedPtr(const WeakPtr<Type>& other) noexcept;
	SharedPtr(SharedPtr&& right) noexcept;
	SharedPtr(UniquePtr<Type>&& right);
	~SharedPtr();

	SharedPtr& operator =(const SharedPtr& vright) noexcept;
	SharedPtr& operator =(SharedPtr&& vright) noexcept;
	SharedPtr& operator =(UniquePtr<Type>&& vright);

	Type& operator *() const noexcept { return *this->get(); }
	Type* operator ->() const noexcept { return this->get(); }
	explicit operator bool() const noexcept { return this->get() != nullptr; }
	bool unique() const noexcept { return this->use_count() == 1; }

	void swap(SharedPtr<Type>& right) noexcept { this->_swap(right); }
	void reset(Type* p = nullptr) noexcept { (p ? SharedPtr(p) : SharedPtr()).swap(*this); }
```

```cpp
};

# pragma region Methods SharedPtr

template <typename Type>
SharedPtr<Type>::SharedPtr(Type* p)
{
	this->set(p, new Count());
}

template <typename Type>
SharedPtr<Type>::SharedPtr(const SharedPtr<Type>& other) noexcept
{
	this->_copyShared(other);
}

template <typename Type>
SharedPtr<Type>::SharedPtr(const WeakPtr<Type>& other) noexcept
{
	this->_copyShared(other);
}

template <typename Type>
SharedPtr<Type>::SharedPtr(SharedPtr<Type>&& right) noexcept
{
	this->_move(right);
}

template <typename Type>
SharedPtr<Type>::SharedPtr(UniquePtr<Type>&& vright)
{
	Type* p = vright.release();

	if (p)
		this->set(p, new Count());
}

template <typename Type>
SharedPtr<Type>::~SharedPtr()
{
	this->delShared();
}

template <typename Type>
SharedPtr<Type>& SharedPtr<Type>::operator =(const SharedPtr<Type>& vright) noexcept
{
	if (this->_compare(vright)) return *this;

	this->delShared();

	this->_copyShared(vright);

	return *this;
}

template <typename Type>
SharedPtr<Type>& SharedPtr<Type>::operator =(SharedPtr<Type>&& vright) noexcept
{
	if (this->_compare(vright)) return *this;

	this->delShared();

	this->_move(vright);

	return *this;
}

template <typename Type>
SharedPtr<Type>& SharedPtr<Type>::operator =(UniquePtr<Type>&& vright)
{
	this->delShared();
```

```
	Type* p = vright.release();

	this->set(p, p ? new Count() : nullptr);

	return *this;
}

#pragma endregion

template <typename Type>
class WeakPtr : public Pointers<Type>
{
public:
	WeakPtr() = default;
	WeakPtr(const WeakPtr& other) noexcept;
	WeakPtr(const SharedPtr<Type>& other) noexcept;
	WeakPtr(WeakPtr&& other) noexcept;
	~WeakPtr();

	WeakPtr& operator =(const WeakPtr& vright) noexcept;
	WeakPtr& operator =(const SharedPtr<Type>& vright) noexcept;
	WeakPtr& operator =(WeakPtr&& vright) noexcept;

	void reset() noexcept { WeakPtr().swap(*this); }
	void swap(WeakPtr& other) noexcept { this->_swap(other); }
	bool expired() const noexcept { return this->use_count() == 0; }

	SharedPtr<Type> lock()const noexcept { return SharedPtr<Type>(*this); }
};

# pragma region Methods WeakPtr

template <typename Type>
WeakPtr<Type>::WeakPtr(const WeakPtr<Type>& other) noexcept
{
	this->_copyWeak(other);
}

template <typename Type>
WeakPtr<Type>::WeakPtr(const SharedPtr<Type>& other) noexcept
{
	this->_copyWeak(other);
}

template <typename Type>
WeakPtr<Type>::WeakPtr(WeakPtr<Type>&& other) noexcept
{
	this->_move(other);
}

template <typename Type>
WeakPtr<Type>::~WeakPtr()
{
	this->delWeak();
}

template <typename Type>
WeakPtr<Type>& WeakPtr<Type>::operator =(const WeakPtr<Type>& vright) noexcept
{
	if (this->_compare(vright)) return *this;

	this->delWeak();
	this->_copyWeak(vright);

	return *this;
}

template <typename Type>
WeakPtr<Type>& WeakPtr<Type>::operator =(const SharedPtr<Type>& vright) noexcept
{

	if (this->_compare(vright)) return *this;
```

```cpp
	this->delWeak();
	this->_copyWeak(vright);

	return *this;
}

template <typename Type>
WeakPtr<Type>& WeakPtr<Type>::operator =(WeakPtr<Type>&& vright) noexcept
{
	if (this->_compare(vright)) return *this;

	this->delWeak();
	this->_move(vright);

	return *this;
}

#pragma endregion

class A
{
	int key;
public:
	A(int k) : key(k)
	{
		cout << "Calling the constructor of class A (obj" << key << ");" << endl;
	}
	~A() { cout << "Calling a class A destructor (obj" << key << ");" << endl; }

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

void main()
{
	SharedPtr<A> obj1(new A(1));

	obj1->f();

	SharedPtr<A> s1, s2(obj1), s3;

	s2->f();

	cout << s2.use_count() << endl;

	WeakPtr<A> w1 = s2;

	s1 = w1.lock();

	SharedPtr<A> s4(w1);

	cout << s2.use_count() << endl;

	WeakPtr<A> w2;
	{
		SharedPtr<A> obj2(new A(2));
		w2 = obj2;

		if (!w2.expired())
			(w2.lock())->f();
	}
	if (!w2.expired())
		(w2.lock())->f();

	s2.reset();
	s3 = s1;
}
```