Пример 1. Шаблоны функций.

```
template <typename Type>
Type* initArray(int count);
template <typename Type>
void freeArray(Type* arr);
template <typename Type>
Type* inputArray(Type* arr, int q);
template <typename Type>
void outputArray(const Type* arr, int q);

template <typename Type> using Tfunc = int(*)(const Type&, const Type&);

template <typename Type>
void sort(Type* arr, int q, Tfunc<Type> cmp);

int compare(const double& d1, const double& d2) { return d1 - d2; }

int main()
{
	const int N = 10;
	double* arr = initArray<double>(N);

	cout << "Enter array: ";
	inputArray(arr, N);

	sort(arr, N, compare);

	cout << "Resulting array: ";
	outputArray(arr, N);

	freeArray(arr);

	return 0;
}

template <typename Type>
Type* initArray(int count) { return new Type[count]; }

template <typename Type>
void freeArray(Type* arr) { delete[]arr; }

template <typename Type>
Type* inputArray(Type* arr, int q)
{
	for (int i = 0; i < q; i++)
		cin >> arr[i];

	return arr;
}

template <typename Type>
void outputArray(const Type* arr, int q)
{
	for (int i = 0; i < q; i++)
		cout << arr[i] << " ";
	cout << endl;
}

template <typename Type>
void sort(Type* arr, int q, Tfunc<Type> cmp)
{
	for (int i = 0; i < q - 1; i++)
		for (int j = i + 1; j < q; j++)
			if (cmp(arr[i], arr[j]) > 0)
				swap(arr[i], arr[j]);
}
```

Пример 11. Правило вызова функций.

```
template <typename Type>
void swap(Type& val1, Type& val2)
{
	Type temp = val1; val1 = val2; val2 = temp;
}

template<>
void swap<float>(float& val1, float& val2)
{
	float temp = val1; val1 = val2; val2 = temp;
}

void swap(float& val1, float& val2)
{
	float temp = val1; val1 = val2; val2 = temp;
}

void swap(int& val1, int& val2)
{
	int temp = val1; val1 = val2; val2 = temp;
}

void main()
{
	const int N = 2;
	int a1[N];
	float a2[N];
	double a3[N];

	swap(a1[0], a1[1]);         // swap(int&, int&)
	swap<int>(a1[0], a1[1]);    // swap<int>(int&, int&)
	swap(a2[0], a2[1]);         // swap(float&, float&)
	swap<float>(a2[0], a2[1]);  // swap<>(float&, float&)
	swap(a3[0], a3[1]);         // swap<double>(double&, double&)
}
```

Пример 12. Определение типа возвращаемого значения для шаблона функции.

```
template <typename T, typename U>
auto sum(const T& elem1, const U& elem2) -> decltype(elem1 + elem2)
{
	return elem1 + elem2;
}

int main()
{
	auto s = sum(1, 1.2);

	cout << "Result: " << s << endl;
}
```

Пример 2. Шаблон класса, шаблоны методов.

```
template <typename Type, size_t N>
class Array
{
private:
	Type arr[N];

public:
	Array() = default;
	Array(initializer_list<Type> lt);
```

```
	Type& operator[](int ind);
	const Type& operator[](int ind) const;

	bool operator ==(const Array<Type, N>& a) const;

	template <typename Type, size_t N>
	friend Array<Type, N> operator+(const Array<Type, N>& a1, const Array<Type, N>& a2);
};

template <typename Type, size_t N>
Array<Type, N>::Array(initializer_list<Type> lt)
{
	int n = N <= lt.size() ? N : lt.size();
	const Type* iter = lt.begin();
	int i;
	for (i = 0; i < n; i++, iter++)
		arr[i] = *iter;

	for (; i < N; i++)
		arr[i] = 0.;
}

template <typename Type, size_t N>
Type& Array<Type, N>::operator[](int ind) { return arr[ind]; }

template <typename Type, size_t N>
const Type& Array<Type, N>::operator[](int ind) const { return arr[ind]; }

template <typename Type, size_t N>
bool Array<Type, N>::operator ==(const Array<Type, N>& a) const
{
	if (this == &a) return true;

	bool Key = true;
	for (int i = 0; Key && i < N; i++)
		Key = this->arr[i] == a.arr[i];

	return Key;
}

template <typename Type, size_t N>
Array<Type, N> operator+(const Array<Type, N>& a1, const Array<Type, N>& a2)
{
	Array<Type, N> res;

	for (int i = 0; i < N; i++)
		res.arr[i] = a1.arr[i] + a2.arr[i];

	return res;
}

template <typename Type, size_t N>
ostream& operator<<(ostream& os, const Array<Type, N>& a)
{
	for (int i = 0; i < N; i++)
		os << a[i] << " ";

	return os;
}

int main()
{
	Array<double, 3> a1{ 1, 2, 3 }, a2{ 1, 2, 3 }, a3{4, 2};

	if (a1 == a2)
		a1 = a2 + a3;
```

```
        cout << a1 << endl;

        return 0;
}
```

Пример 3. Полная специализация шаблона класса и метода шаблона класса.

```
template <typename Type>
class A
{
public:
        A() { cout << "constructor of template A;" << endl; }
        void f() { cout << "metod f of template A;" << endl; }
};

template<>
void A<int>::f() { cout << "specialization of metod f of template A;" << endl;}

template <>
class A<float>
{
public:
        A() { cout << "specialization constructor template A;" << endl; }
        void f() { cout << "metod f specialization template A;" << endl; }
        void g() { cout << "metod g specialization template A;" << endl; }
};

int main()
{
        A<double> obj1;
        obj1.f();

        A<float> obj2;
        obj2.f();
        obj2.g();

        A<int> obj3;
        obj3.f();

        return 0;
}
```

Пример 4. Частичная специализация шаблона класса, параметры шаблона класса по умолчанию.

```
template <typename T1, typename T2 = double>
class A
{
public:
        A() { cout << "constructor of template A<T1, T2>;" << endl; }
};

template <typename T>
class A<T, T>
{
public:
        A() { cout << "constructor of template A<T, T>;" << endl; }
};

template <typename T>
class A<T, int>
{
public:
        A() { cout << "constructor of template A<T, int>;" << endl; }
};

template <typename T1, typename T2>
```

```
class A<T1*, T2*>
{
public:
	A() { cout << "constructor of template A<T1*, T2*>;" << endl; }
};


int main()
{
	A<int> a0;
	A<int, float> a1;
	A<float, float> a2;
	A<float, int> a3;
	A<int*, float*> a4;

//	A<int, int> a5;			// Error!!!
//	A<int*, int*> a6;	// Error!!!
}
```

Пример 5. Шаблон функции с переменным числом параметров.

```
template <typename Type>
Type sum(Type value)
{
	return value;
}

template <typename Type, typename ...Args>
Type sum(Type value, Args... args)
{
	return value + sum(args...);
}

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

	return 0;
}
```

Пример 6. Шаблон с переменным числом параметров значений.

```
template<size_t...>
struct Sum {};

template<>
struct Sum<>
{
	enum { value = 0 };
};

template<size_t val, size_t... args>
struct Sum<val, args...>
{
	enum { value = val + Sum<args...>::value };
};

int main()
{
	cout << Sum<1, 2, 3, 4>::value << endl;

	return 0;
}
```

Пример 7. Шаблон класса с переменным числом параметров. Рекурсивная реализация кортежа.

```
template <typename... Types>
class Tuple;

template <typename Head, typename... Tail>
class Tuple<Head, Tail...>
{
private:
	Head value;
	Tuple<Tail...> tail;
public:
	Tuple() = default;
	Tuple(const Head& v, const Tuple<Tail...>& t) : value(v), tail(t) {}
	Tuple(const Head& v, const Tail&... tail) : value(v), tail(tail...) {}

	Head& getHead() { return value; }
	const Head& getHead() const { return value; }

	Tuple<Tail...>& getTail() { return tail; }
	const Tuple<Tail...>& getTail() const { return tail; }
};

template <>
class Tuple<>
{
};

template <size_t N>
struct Get
{
	template <typename Head, typename... Tail>
	static auto apply(const Tuple<Head, Tail...>& t)
	{
		return Get<N - 1>::apply(t.getTail());
	}
};

template <>
struct Get<0>
{
	template <typename Head, typename... Tail>
	static const Head& apply(const Tuple<Head, Tail...>& t)
	{
		return t.getHead();
	}
};

template <size_t N, typename... Types>
auto get(const Tuple<Types...>& t)
{
	return Get<N>::apply(t);
}

size_t count(const Tuple<>&)
{
	return 0;
}

template <typename Head, typename... Tail>
size_t count(const Tuple<Head, Tail...>& t)
{
	return 1 + count(t.getTail());
}

ostream& writeTuple(ostream& os, const Tuple<>&)
{
	return os;
```

```
}

template <typename Head, typename... Tail>
ostream& writeTuple(ostream& os, const Tuple<Head, Tail...>& t)
{
	os << t.getHead() << " ";
	return writeTuple(os, t.getTail());
}

template <typename... Types>
ostream& operator<<(ostream& os, const Tuple<Types...>& t)
{
	return writeTuple(os, t);
}

int main()
{
	Tuple<const char*, double, int, char> obj("Pi: ", 3.14, 15, '!');

	cout << get<0>(obj) << get<1>(obj) << get<2>(obj) << get<3>(obj) << endl;

	cout << obj << endl;

	cout << "Count = " << count(obj) << endl;
}
```

Пример 13. Приведение типов в C++.

```
class A
{
	int a = 0;
public:
	virtual ~A() = 0;

	void f() { cout << "method f class A:"<< a << endl; }
};

A::~A() {}

class B : public A
{
	int b = 1;
public:
	void f() { cout << "method f class B;" << b << endl; }

	void g1() { cout << "method g1 class B;" << endl; }
};

class C : public B
{
	int c = 2;
public:
	void f() { cout << "method f class C;" << c << endl; }

	void g2() { cout << "method g2 class B;" << endl; }
};

class D : public A
{
	int d = 3;
public:
	void f() { cout << "method f class D;" << d << endl; }

};

int main()
{
```

```
	A* pa = new B;

	B* pb = static_cast<B*>(pa);

	pb->f();

	C* pc = static_cast<C*>(pa);

	pc->f();

	D* pd = static_cast<D*>(pa);

	pd->f();

	pb = dynamic_cast<B*>(pa);
	if (!pb)
	{
		cout << "Error bad cast!" << endl;
	}
	else
	{
		pb->f();
		pb->g1();
	}

	pc = dynamic_cast<C*>(pa);
	if (!pc)
	{
		cout << "Error bad cast!" << endl;
	}
	else
	{
		pc->f();
		pc->g2();
	}

	const B obj;
	const B* p = &obj;

	const_cast<B*>(p)->f();
}
```

Пример 8. Реализация хранителя unique_ptr.

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

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

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

	Type* get() const noexcept { return ptr; }
	Type* release() noexcept;
	void reset(Type* p = nullptr) noexcept;

	UniquePtr(const UniquePtr<Type>&) = delete;
	UniquePtr& operator=(const UniquePtr<Type>&) = delete;
```

```cpp
private:
	Type* ptr{ nullptr };
};

# pragma region Method UniquePtr
template <typename Type>
UniquePtr<Type>::UniquePtr(UniquePtr<Type>&& vright) noexcept
{
	ptr = vright.ptr;
	vright.ptr = nullptr;
}

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

	return *this;
}

template <typename Type>
UniquePtr<Type>& UniquePtr<Type>::operator=(UniquePtr<Type>&& vright) noexcept
{
	ptr = vright.ptr;
	vright.ptr = nullptr;

	return *this;
}

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

	return p;
}

template <typename Type>
void UniquePtr<Type>::reset(Type* p) noexcept
{
	delete ptr;
	ptr = p;
}

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

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

int main()
```

```
{
	UniquePtr<A> obj1(new A);

	obj1->f();
	(*obj1).f();

	UniquePtr<A> obj2;

//	obj2 = obj1; Error!!!
	obj2 = Unique::move(obj1);
}
```

Пример 9. Реализация shared_ptr и weak_ptr.

```
# include "UniquePtr.h"

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<Type>&) = delete;
	Pointers<Type>& operator=(const Pointers<Type>&) = 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<Type>& right) const noexcept { return this->get() ==
right.get(); }
	void _swap(Pointers<Type>& right) noexcept
	{
		std::swap(ptr, right.ptr);
		std::swap(rep, right.rep);
	}
	void _copyShared(const Pointers<Type>& right) noexcept;
	void _copyWeak(const Pointers<Type>& right) noexcept;
	void _move(Pointers<Type>& 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)
		{
			(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;
		constexpr SharedPtr(nullptr_t) noexcept {}
```

```cpp
	explicit SharedPtr(Type* p);
	SharedPtr(const SharedPtr<Type>& other) noexcept;
	explicit SharedPtr(const WeakPtr<Type>& other) noexcept;
	SharedPtr(SharedPtr<Type>&& right) noexcept;
	SharedPtr(UniquePtr<Type>&& right);
	~SharedPtr();

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

# 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<Type>& other) noexcept;
        WeakPtr(const SharedPtr<Type>& other) noexcept;
        WeakPtr(WeakPtr<Type>&& other) noexcept;
        ~WeakPtr();

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

        void reset() noexcept { WeakPtr().swap(*this); }
        void swap(WeakPtr<Type>& 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;

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

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

int main()
{
	SharedPtr<A> obj1(new A);

	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);
                w2 = obj2;

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

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

Пример 10. Создание итератора (без проверок и обработки исключительных ситуация).

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

using namespace std;

template <typename Type>
class Iterator;

class BaseArray
{
public:
        BaseArray(size_t sz = 0) { count = shared_ptr<size_t>( new size_t(sz) ); }
        virtual ~BaseArray() = default;

        size_t size() { return bool(count) ? *count : 0; }
        operator bool() { return size(); }

protected:
        shared_ptr<size_t> count;
};

template <typename Type>
class Array final : public BaseArray
{
public:
        Array(initializer_list<Type> lt);
        virtual ~Array() {}

        Iterator<Type> begin() const { return Iterator<Type>(arr, count); }
        Iterator<Type> end() const { return Iterator<Type>(arr, count, *count);     }

private:
        shared_ptr<Type[]> arr{ nullptr };
};

template <typename Type>
class Iterator : public std::iterator<std::input_iterator_tag, Type>
{
        friend class Array<Type>;

private:
        Iterator(const shared_ptr<Type[]>& a, const shared_ptr<size_t>& c, size_t ind = 0) : arr(a),
count(c), index(ind) {}
public:
        Iterator(const Iterator &it) = default;
```

```
	bool operator!=(Iterator const& other) const;
	bool operator==(Iterator const& other) const;

	Type& operator*();
	const Type& operator*() const;
	Type* operator->();
	const Type* operator->() const;
	Iterator<Type>& operator++();
	Iterator<Type> operator++(int);

private:
	weak_ptr<Type[]> arr;
	weak_ptr<size_t> count;
	size_t index = 0;
};

#pragma region Method Array

template <typename Type>
Array<Type>::Array(initializer_list<Type> lt)
{
	if (!(*count = lt.size())) return;

	arr = shared_ptr<Type[]>(new Type[*count]);

	size_t i = 0;
	for (Type elem : lt)
		arr[i++] = elem;
}

#pragma endregion

#pragma region Methods Iterator

template <typename Type>
bool Iterator<Type>::operator!=(Iterator const& other) const { return index != other.index; }

template <typename Type>
Type& Iterator<Type>::operator*()
{
	shared_ptr<Type[]> a(arr);

	return a[index];
}

template <typename Type>
Iterator<Type>& Iterator<Type>::operator++()
{
	shared_ptr<size_t> n(count);
	if (index < *n)
		index++;

	return *this;
}

template <typename Type>
Iterator<Type> Iterator<Type>::operator++(int)
{
	Iterator<Type> it(*this);

	++(*this);

	return it;
}

#pragma endregion
```

```cpp
template <typename Type>
ostream& operator<<(ostream& os, const Array<Type>& arr)
{
	for (auto elem : arr)
		cout << elem << " ";

	return os;
}

int main()
{
	Array<int> arr{ 1, 2, 3, 4, 5 };

	cout << " Array: " << arr << endl;
}
```