

ОБЪЕКТНО-
ОРИЕНТИРОВАННОЕ
ПРОГРАММИРОВАНИЕ
2024



ДВА ВИДА МНОГОКРАТНОГО ИСПОЛЬЗОВАНИЯ КОДА

Наследование

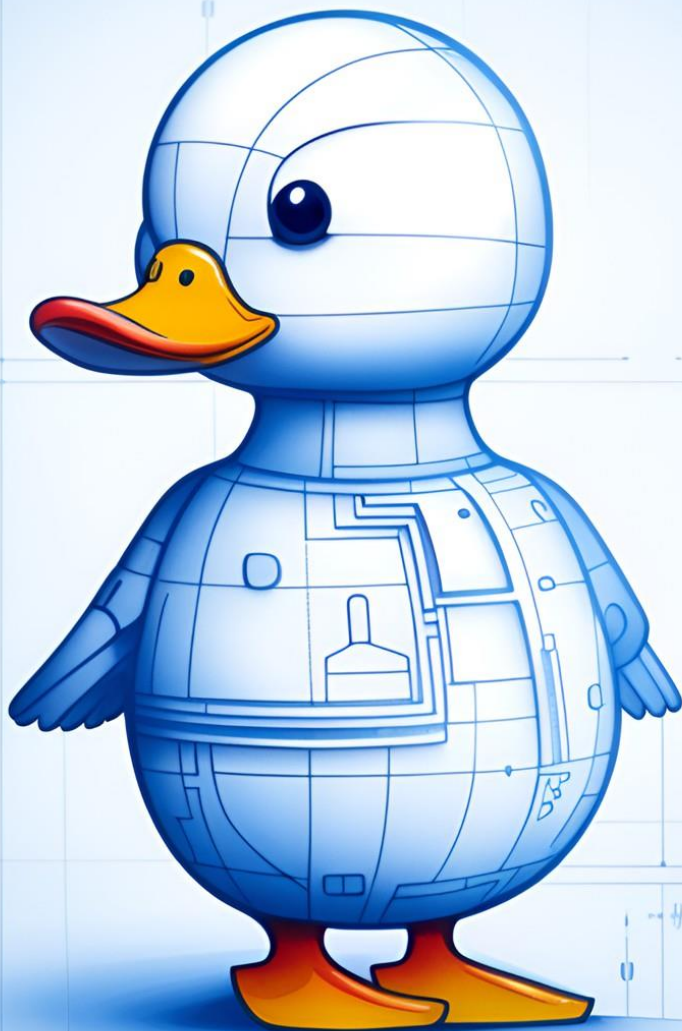
- Создаем структуру для работы с «базовым классом»
- Создаем классы-наследники на каждый случай.

Шаблоны (Template)

- Описываем «стратегию работы» с «неопределенным» классом.
- Компилятор в момент создание класса по шаблону, сам создает нужный «код» для конкретного класса.

TEMPLATE ЭТО ...

- Инструкции для генерации функции или класса в процессе компиляции программы.
- Параметром шаблона может являться как значение переменной (как в обычных функциях) так и тип данных.
- Параметры подставляются на этапе компиляции программы (должны быть вычислимы на этапе компиляции).





C++ CORE GUIDELINES

- **T.120: Use template metaprogramming only when you really need to**
- <https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#Rt-metameta>

ПРОСТОЙ TEMPLATE

```
template <class T>
struct Container{
    T payload;
    Container(const T& value) : payload(value){};
};
```

```
template <class T>
void print(T value){
    std::cout << "Value:" << value << std::endl;
}
```

```
template<int V>
struct foo{
    static const int value = V;
};
```


TEMPLATE

Перед описанием класса ставим ключевое слово `template <class T>` или `template <typename T>`

T – используем вместо имени класса, который будет заменяться при создании конкретного экземпляра класса.

```
template <class T> struct print {};
```

```
// print– это шаблон
```

```
// print<int> - это класс, сконструированный по шаблону
```

TEMPLATE VS CLASS

```
struct foo {  
    static const int value = 10;  
};
```

```
template<int V>  
struct foo {  
    static const int value = V;  
};
```

```
foo::value; // есть всегда  
foo<10>::value; // появляется при использовании
```

TWO ARGUMENTS

```
// два параметра, через запятую
template <class A, class B>
class Sum {
private:
    A a;
    B b;
public:
    Sum(A a_value, B b_value) : a(a_value), b(b_value) {
    }
    // параметры C и D (а не A,B) поскольку это ссылка на другой шаблон
    // описанный ниже в этом файле
    template <class C,class D> friend
        std::ostream& operator<<(std::ostream & os, Sum<C,D> &sum);
};
```


COMPLEX PARAMETERS

```
template <class TYPE, TYPE def_value, size_t SIZE = 10 > class Array {
protected:
    TYPE _array[SIZE];
public:
    Array() {
        for (int i = 0; i < SIZE; i++) {
            _array[i] = def_value;
        }
    }
    const size_t size() {
        return SIZE;
    }
    TYPE* begin() {
        return &_array[0];
    }
    TYPE* end() {
        return &_array[SIZE]; // element beyond the array
    }
    TYPE& operator[](size_t index) {
        if ((index >= 0) && (index < SIZE)) return _array[index];
        else throw BadIndexException(index, SIZE);
    }
};
```

SPECIALIZATION

```
template <class T> class MyContainer {  
    T element;  
    // тело класса  
};  
  
// Специализация для типа char  
template <> class MyContainer <char> {  
    char element;  
    // тело класса  
  
}
```

PARTIAL SPECIALIZATION

```
template <class A,class B,class C>
struct Foo{
    A add(B b, C c){
        return static_cast<A>(b+c);
    }
};
```

```
template <class A>
struct Foo<A,char,char>{
    A add(char b,char c){
        return static_cast<A>(b+c-'0'-'0');
    }
};
```

Частичная
специализация для
функций

не разрешена



METAFUNCTION

```
namespace example{  
    // метафункция  
    template <int V>  
    struct abs{  
        static const int value = V<0 ? -V : V;  
    };  
}
```

FACTORIAL

```
template<uint64_t n>
struct fact{
    static const uint64_t value = fact<n-1>::value * n;
};
```

```
template<>
struct fact<1>{
    static const uint64_t value = 1;
};
```


FACTORIAL 3

```
struct fact<3>{  
    static const uint64_t value = fact<2>*3;  
};
```

```
struct fact<2>{  
    static const uint64_t value = fact<1>*2;  
};
```

```
struct fact<1>{  
    static const uint64_t value = 1;  
};
```

TYPES

```
template<class T>  
struct remove_const {  
    using type = T;  
};
```

```
template<class T>  
struct remove_const<const T> {  
    using type = T;  
};
```

```
remove_const<const int>::type a=5; // int
```

Является ли тип указателем?

Специализация шаблонов

```
template <class T> struct is_pointer{  
    enum {Value = false};  
};
```

```
template <class T> struct is_pointer<T*>{  
    enum {Value = true};  
};
```

```
std::cout << is_pointer<int*>::Value?"Pointer":"Not pointer";
```



SFINAE SUBSTITUTION FAILURE IS NOT AN ERROR

<https://en.cppreference.com/w/cpp/language/sfinae>

Техника при которой компилятор пытается вывести тип для параметра шаблона встречая ошибку в конкретной специализации, не выдает ошибку пользователю, а анализирует все возможные варианты.

ENABLE_IF

```
template <bool condition, class T>  
struct enable_if{  
};
```

```
template <class T>  
struct enable_if<true, T>{  
    using value = T; // value only in specialization  
};
```

ENABLE_IF

```
#include <iostream>
#include <type_traits>
// Функция, которая работает только с целочисленными типами
template<typename T, typename std::enable_if<std::is_integral<T>::value, int>::type = 0>
void print_integer(T value) {
    std::cout << "Integer value: " << value << std::endl;
}

int main() {
    int a = 42;
    double b = 3.14;
    print_integer(a); // Работает, так как int — целочисленный тип
    // print_integer(b); // Ошибка компиляции, так как double — не целочисленный тип
    return 0;
}
```


#INCLUDE<TYPE_TRAITS>

[HTTP://WWW.CPLUSPLUS.COM/REFERENCE/TYPE_TRAITS/](http://www.cplusplus.com/reference/type_traits/)

● Type traits

Primary type categories

is_array	Is array (class template)
is_class	Is non-union class (class template)
is_enum	Is enum (class template)
is_floating_point	Is floating point (class template)
is_function	Is function (class template)
is_integral	Is integral (class template)
is_lvalue_reference	Is lvalue reference (class template)
is_member_function_pointer	Is member function pointer (class template)
is_member_object_pointer	Is member object pointer (class template)
is_pointer	Is pointer (class template)
is_rvalue_reference	Is rvalue reference (class template)
is_union	Is union (class template)
is_void	Is void (class template)

Composite type categories

is_arithmetic	Is arithmetic type (class template)
is_compound	Is compound type (class template)
is_fundamental	Is fundamental type (class template)

Много категорий
метафункций
для построения
собственных
шаблонов.

TYPE TRATES

```
template <class T>
typename std::enable_if<std::is_array<T>::value,void>::type print(T& value){
    for(auto a: value)
        std::cout << a << " ";
    std::cout << std::endl;
}
```

```
template <class T>
typename std::enable_if<std::is_pointer<T>::value,void>::type print(T& value){
    std::cout << "pointer:" << value << std::endl;
}
```



CONCEPTS

CONCEPT

— это новая возможность в C++, представленная в стандарте C++20, которая позволяет явно определять требования к типам, используемым в шаблонах. Концепты делают код более читаемым, безопасным и упрощают отладку, так как они позволяют компилятору проверять соответствие типов требованиям на этапе компиляции.

ОПРЕДЕЛЕНИЕ CONCEPT

- Концепты определяются с помощью ключевого слова concept.
- Они представляют собой логические условия, которые должны выполняться для типов, используемых в шаблонах.

ОПРЕДЕЛЕНИЕ КОНЦЕПТА

```
template<class T>  
concept Integral = std::is_integral_v<T>;
```


ИСПОЛЬЗОВАНИЕ В ШАБЛОНЕ

```
template<Integral T>
void print_number(T num) {
    std::cout << num << std::endl;
}
```

ВЫРАЖЕНИЕ ТРЕБОВАНИЙ ВНУТРИ REQUIRES

```
template<class T>  
concept Addable = requires(T a, T b) {  
    { a + b } -> std::same_as<T>;  
};
```

ТРЕБОВАНИЯ К ФУНКЦИЯМ И МЕТОДАМ

```
template<class T>  
concept HasToString = requires(T a) {  
    { a.to_string() } -> std::convertible_to<std::string>;  
};
```

КОМБИНИРОВАНИЕ ТРЕБОВАНИЙ

```
template<class T>  
concept Numeric = std::integral<T> || std::floating_point<T>;
```

ИСПОЛЬЗОВАНИЕ REQUIRES В ТЕЛЕ ФУНКЦИИ

```
template<class T>  
void foo(T x) {  
    static_assert(requires { x.bar(); }, "T must have a member function bar()");  
}
```

ПРИМЕР

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

```
// Определение концепта для целочисленных типов
template<typename T>
concept Integral = std::is_integral_v<T>;
```

```
// Функция, которая работает только с типами, удовлетворяющими концепту Integral
template<Integral T>
void print_integer(T value) { std::cout << "Integer value: " << value << std::endl; }
```


ПРИМЕР

```
template<typename T> struct is_integral : public false_type {};
```

/ Специализации для всех целочисленных типов

```
template<> struct is_integral<bool> : public true_type {};
```

```
template<> struct is_integral<char> : public true_type {};
```

```
// ...
```

```
template<class T, T v>
```

```
struct integral_constant {
```

```
    static constexpr T value = v;
```

```
    using value_type = T;
```

```
    using type = integral_constant;
```

```
    constexpr operator value_type() const noexcept { return value; }
```

```
    constexpr value_type operator()() const noexcept { return value; } // since c++14
```

```
};
```

```
using false_type = integral_constant<bool, false>;
```

```
using true_type = integral_constant<bool, true>;
```



VARIADIC TEMPLATE

Variadic template

variadic_1.cpp

```
template <class T> void print(const T& t) {  
    std::cout << t << std::endl;  
}  
  
template <class First, class... Rest>  
void print(const First& first, const Rest&... rest) {  
    std::cout << first << ", ";  
    print(rest...); // рекурсия на стадии компиляции!  
}
```

basic_tuple.cpp

```
template <class... Ts> class tuple {  
};
```

```
template <class T, class... Ts>  
class tuple<T, Ts...> : public tuple<Ts...> {  
    public:  
        tuple(T t, Ts... ts) : tuple<Ts...>(ts...), value(t) {}  
        tuple<Ts...> &next = static_cast<tuple<Ts...>&>(*this);  
        T value;  
};
```

Что внутри?

```
class tuple {  
}
```

```
class tuple<const char*> : public tuple {  
    const char* value;  
}
```

```
class tuple<uint64_t, const char*> : public tuple<const char*>{  
    uint64_t value;  
}
```

```
class tuple<double, uint64_t, const char*> : public tuple<uint64_t, const char*>{  
    double value;  
}
```

CONSTEXPR_IF.CPP //C++17

```
template <class T> std::string to_string(T x)
{
    if constexpr (std::is_same<T, std::string>::value)
    {
        return x;
        // ERROR, if no conversion to string
    }
    else if constexpr (std::is_integral<T>::value)
    {
        return std::to_string(x); // ERROR, if x is not numeric
    }
    else
    {
        return std::string(x);
        // ERROR, if no conversion to string
    }
}
```

CRTP.CPP

```
template <class T> class base{  
};  
  
class derived : public base<derived> {  
};
```

Такая конструкция
делает возможным
обращение к
производному классу
из базового!

Мы можем делать
обобщенные
алгоритмы для
методов которые
появляются в
наследнике!



УМНЫЕ УКАЗАТЕЛИ



CORE GUIDELINE

- R.20: Use `unique_ptr` or `shared_ptr` to represent ownership
- <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>

RAW POINTERS

```
int main()
{
    int a;
    a = 10; // Не знаем адрес, да и ладно

    int *ptr = &a;
    *ptr += 5; // Зачем-то знаем адрес, но не используем

    int *ptr2 = &a;
    *(ptr + 42) += 5; // Знаем адрес, но используем как-то неправильно

    int &ref = a;
    ref += 3; // Не знаем адрес, но ссылаемся
    return 1;
}
```

RAW POINTERS

```
int main()
{
    {
        int *ptr = new int{42};
    } // выход за scope - утечка
    {
        int value = 0;
        int *ptr = new int{50};
        // опять потеряли адрес - утечка
        ptr = &value;
    }
    {
        int *ptr = new int{79};
        // Утечка, если функция бросит исключение
        someFunctionHere();
        delete ptr;
    }
}
```



RAW POINTERS

- нет контроля создания / удаления
- может указывать в неизвестность, `nullptr`
- может указывать в известность, но чужую



RAW POINTERS ПРОБЛЕМЫ

- не инициализации указателя
- не удаление указателя
- копирование указателей
- -повторное удаление

RAII RESOURCE ACQUISITION IS INITIALIZATION

- **Получение ресурса есть инициализация (RAII)** — программная идиома объектно-ориентированного программирования, смысл которой заключается в том, что с помощью тех или иных программных механизмов получение некоторого ресурса неразрывно совмещается с инициализацией, а освобождение — с уничтожением объекта.
- Типичным (хотя и не единственным) способом реализации является организация получения доступа к ресурсу в **конструкторе**, а освобождения — в **деструкторе** соответствующего класса.
- Поскольку деструктор автоматической переменной вызывается при выходе её из области видимости, то ресурс гарантированно освобождается при уничтожении переменной. Это справедливо и в ситуациях, в которых возникают **исключения**.





ИДЕЯ

```
{
```

```
МойОбъект объект;
```

```
// вызывается конструктор
```

```
// выделяется память в куче
```

```
...
```

```
}
```

```
// Вызывается деструктор
```

```
// Освобождается память в куче
```


15_UNIQUEPOINTERS

UNIQUE_PTR.CPP

```
std::unique_ptr<int> ptr{new int{10}};

assert(ptr);
assert(*ptr == 10);
assert(*ptr.get() == 10);
std::cout << "sizeof(ptr) = " << sizeof(ptr) << std::endl;
```

ЧУТЬ УМНЕЕ УКАЗАТЕЛЬ

16_CUSTOMUNIQUEPOINTER

```
auto main() -> int {
    smart_ptr<SomeClass> ptr1;

    std::cout << "start" << std::endl;
    {
        smart_ptr<SomeClass> ptr2{new SomeClass()};
        ptr1 = ptr2; //дублируем
        ptr2 = smart_ptr<SomeClass>{new SomeClass()}; // затираем
    }
    std::cout << "end" << std::endl;
    return 0;
}
```

```
STD::UNIQUE_PTR  
#INCLUDE<MEMORY.H>
```

нераздельное владение объектом

нельзя копировать (только
перемещение)

размер зависит от
пользовательского deleter-a

без особой логики удаления
издержки чаще отсутствуют

std::make_unique – только в качестве
«синтаксического сахара»

РАЗДЕЛЯЕМЫЙ УКАЗАТЕЛЬ

18_CUSTOMSHARED

```
template<class T> struct smart_ptr {
    smart_ptr(T* ptr) : m_counter{new std::size_t{1}}, m_ptr{ptr} {
    }
    smart_ptr(const smart_ptr& other)
    : m_counter{ other.m_counter }, m_ptr{ other.m_ptr } {
    ++*m_counter;
    }
    ~smart_ptr() {
    if (--*m_counter == 0) {
        delete(m_ptr);
        delete(m_counter);
    }
    }
private:
    T* m_ptr;
    std::size_t* m_counter;
};
```

ШАБЛОН
STD::SHARED_PTR<T>

// #INCLUDE<MEMORY>

1. Предоставляет возможности по обеспечению автоматического удаления объекта, за счет подсчета ссылок указатели на объект;
2. Хранит ссылку на один объект;
3. При создании std::shared_ptr<T> счетчик ссылок на объект увеличивается;
4. При удалении std::shared_ptr<T> счетчик ссылок на объект уменьшается;
5. При достижении счетчиком значения 0 – объект автоматически удаляется;

STD::SHARED_PTR SHARED_PTR.CPP

можно копировать с разделением
владения

но дешевле перемещать

всегда внутри два указателя

`std::make_shared` – выделяет память
сразу под объект и счетчик за один раз!

потокобезопасный (и хорошо, и плохо)

можно создать из `unique_ptr`

ДВОЙНОЕ УДАЛЕНИЕ

```
int * ptr = new int{42};  
  
{  
  
    std::shared_ptr<int> smartPtr1{ptr};  
    std::shared_ptr<int> smartPtr2{ptr};  
  
} // двойное удаление
```

STD::DYNAMIC_POINTER_CAST<T> 19_DYNAMIC_POINTER_CAST

```
std::shared_ptr<B> b(new B());
```

```
std::shared_ptr<A> ptr = b;
```

```
if(std::shared_ptr<B> ptr_b = std::dynamic_pointer_cast<B>(ptr)){  
    ptr_b->Do();  
}
```

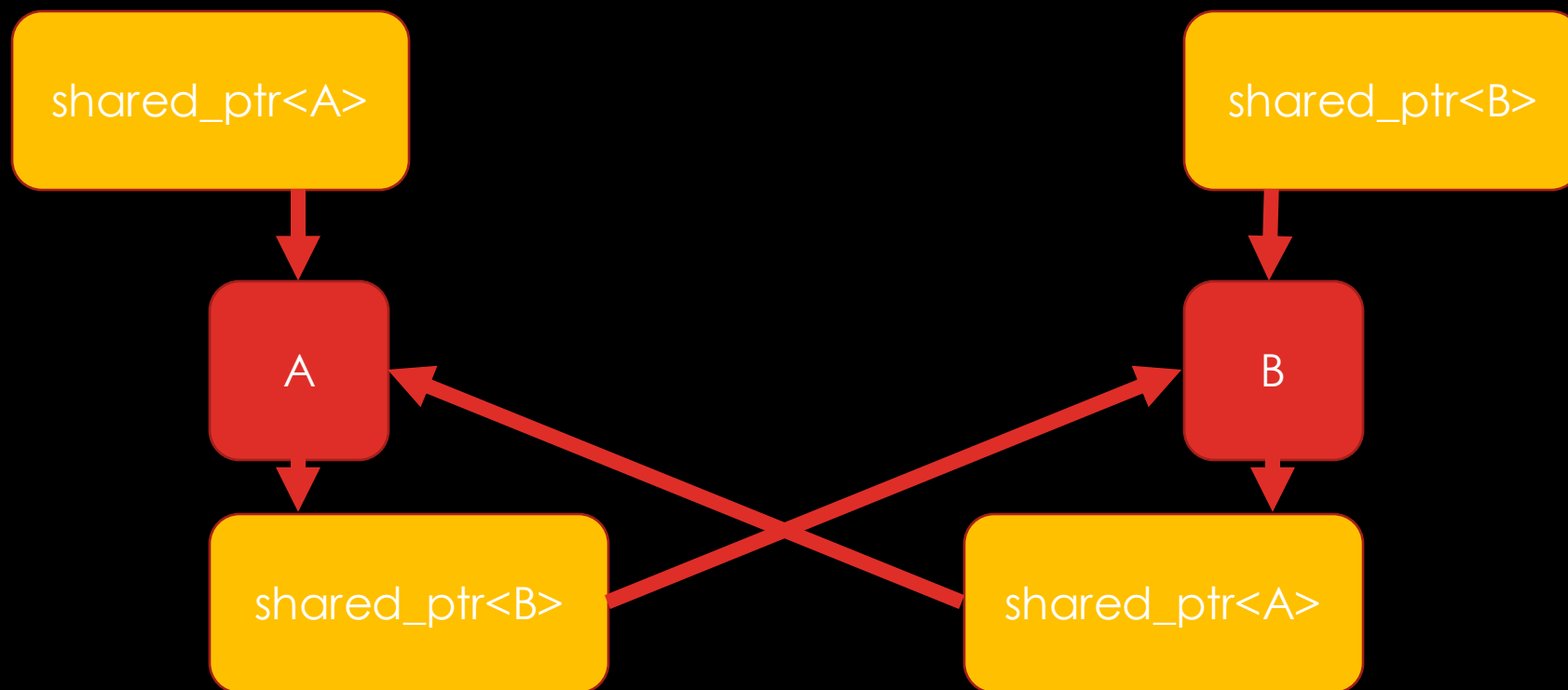

20_ENABLE_SHARED_FROM_THIS

```
struct SomeStruct : std::enable_shared_from_this<SomeStruct> {
    SomeStruct() {
        std::cout << "ctor" << std::endl;
    }
    ~SomeStruct() {
        std::cout << "dtor" << std::endl;
    }

    std::shared_ptr<SomeStruct> getPtr() {
        return shared_from_this();
    }
};
```

ПЕРЕКРЕСТНЫЕ ССЫЛКИ И STD:SHARED_PTR 21_DEAD_LOCK

Если зациклить объекты друг на друга, то появится «цикл» и объект ни когда не удалится! Т.к. деструктор не запустится!



СЛАБЫЙ УКАЗАТЕЛЬ STD::WEAK_PTR

shared_ptr представляет разделяемое владение, но с моей точки зрения разделяемое владение не является идеальным вариантом: значительно лучше, когда у объекта есть конкретный владелец и его время жизни точно определено.

std::weak_ptr

1. Обеспечивает доступ к объекту, только когда он существует;
2. Может быть удален кем-то другим;
3. Содержит деструктор, вызываемый после его последнего использования (обычно для удаления анонимного участка памяти).

22_WEAK_PTR

```
struct Observable {  
    void registerObserver(const std::shared_ptr<Observer>& observer) {  
        m_observers.emplace_back(observer);  
    }  
  
    void notify() {  
        for (auto& obs : m_observers) {  
            auto ptr = obs.lock();  
            if (ptr)  
                ptr->notify();  
        }  
    }  
private:  
    std::vector<std::weak_ptr<Observer>> m_observers;  
};
```

ТЕПЕРЬ БЕЗ DEAD LOCK 23_WEAK_PTR_DEADLOCK

```
class A {  
private:  
    std::weak_ptr<B> b;  
public:  
    void LetsLock(std::shared_ptr<B> value) {  
        b = value;  
    }  
    ~A(){  
        std::cout << "A killed!" << std::endl;  
    }  
};
```



ПРИМЕР ЛАБОРАТОРНОЙ №4



СПАСИБО!

На сегодня все