5.23. Реализуйте метафункцию is_nothrow_move_constructible и с помощью нее реализуйте функцию move_if_noexcept. Объясните, как работает ваша реализация. Почему нельзя наивно выразить is_nothrow_move_constructible как is_move_constructible_v<...> && noexcept(...)?

Нам понадобится вспомогательный класс `integral_constant`, отвечающий за compile-time константы

```
template<typename T, T v>
struct integral_constant {
    static const T value = v;
};
```

Через него можно выразить две важных константы:

- `true_type = integral_constant<bool, true>`
- `false_type = integral_constant<bool, false>`

```
template<typename T>
struct is_nothrow_move_constructible {
private:
    template<typename TT>
    static auto f(int) -> integral_constant<bool,
                                            noexcept(TT(declval<TT>()))>;
    // более понятный вариант:
    // -> std::conditional_t<noexcept(TT(declval<TT>()), true_type, false_type>

    template<typename...>
    static auto f(...) -> false_type;
public:
    static const bool value = decltype(f<T>(0))::value;
};

template<typename T>
const bool is_nothrow_move_constructible_v =
            is_nothrow_move_constructible<T>::value;

template<typename T>
auto move_if_noexcept(T& x) -> std::conditional_t<
        is_nothrow_move_constructible_v<T>, T&&, const T&> {
    return std::move(x);
}
```

Объяснение:

1. У нас нет move конструктора. Тогда по SFINAE выберется вторая версия f и мы получим false
2. Есть move конструктор, но он не noexcept. Тогда возвращаемый тип первой версии f станет `false_type` и мы получим ответ false
3. Move конструктор есть и он noexcept. Аналогично пункту 2 получим ответ true.

Принцип работы move_if_noexcept очевиден (муваем если есть noexcept move конструктор, иначе - возвращаем константную ссылку)

Неправильная реализация: Почему нельзя просто сделать так?

```
template <typename T>
auto move_if_noexcept(T& x) -> std::conditional_t<
        is_move_constructible_v<T> && noexcept(T(declval<T>())),
        T&&, const T&> {
    return std::move(x);
}
```

Мы привыкли к тому, что вторая часть конъюнкции не вычисляется, но забыли о том, что она всегда компилируется. Если у нас не будет move конструктора, то часть с `noexcept(...)` попросту не скомпилируется, и даже `is_move_constructible_v<T>` нас не спасет.

5.24. Реализуйте метафункцию is_base_of<T, U>, позволяющую проверить, является ли класс U наследником класса T (в том числе приватным). Объясните, как работает ваша реализация.

```
namespace details {
    template <typename B>
    auto f(B*) -> true_type;

    template <typename...>
    auto f(...) -> false_type;

    template <typename B, typename D>
    auto test(int) -> decltype(f<B>(declval<D*>()));

    template <typename...> // сюда попадем только если наследование было приватным
    auto test(...) -> true_type;
}

template<typename B, typename D>
struct is_base_of: integral_constant <bool,
        std::is_class_v<B> && std::is_class_v<D> && // проверяем что это классы
        // вариант не работающий с приватным наследованием:
        // decltype(details::f<B>(std::declval<D*>()))::value> {};
        decltype(details::test<B, D>(0))::value> {};
```

Объяснение: Мы пытаемся подставить предполагаемого наследника вместо родителя. Если у нас не получается, то должно сработать SFINAE и перекинуть нас в функцию с возвращаемым значением `false_type`. Если у нас наследование приватное, то одних f нам недостаточно (проверка приватности произойдет после выбора версии для перегрузки, то есть мы уже выберем первую версию, проверим приватность и получим CE).

Добавляем еще один уровень SFINAE с test. В случае, когда наследования нет/оно не приватное test будет действовать так же, как наш первый вариант с f. Если наследование приватное, f не сможет скомпилироваться, а значит по SFINAE нам приется перейти во вторую версию перегрузки test, которая имеет возвращаемый тип `true_type`