5.21. Реализуйте метафункцию has_method, позволяющую проверить, присутствует ли у класса T метод с заранее заданным названием от данных типов аргументов. Объясните, как работает ваша реализация. Для чего в ней нужна функция declval и что эта функция из себя представляет? Для чего и в каких местах STL используется метафункция has_method?

Рассмотрим функцию has_method на примере метода construct

```
// T - класс в котором проверяем метод, Args - аргументы метода
template <typename T, typename... Args>
struct has_method_construct {
private:
    template<typename TT, typename... AArgs>
    static auto f(int) -> decltype(declval<TT>().construct(declval<AArgs>()
    ...), int());
    // возвращаемый тип - int, так как оператор запятая возвращает последний операнд

    template<typename...>
    static char f(...);
public:
    static const bool value = std::is_same_v< decltype(f<T, Args...>(0)),
    int >;
    // будет истина только когда выбралась первая версия функции
};
```

Объяснение: Как и все подобные функции, has_method работает благодаря SFINAE. Когда мы вызываемся от 0, компилятор пытается выбрать первое объявление f как более частное (возвращаемый тип int), но если искомого метода у класса нет, то сработает SFINAE и он перейдет ко второму варианту (возвращаемый тип char).

Зачем делать функции f шаблонными? Если не сделать и просто воспользоваться T и Args, то шаблонные параметры зафиксируются в момент инстанцирования класса и SFINAE не сработает.

Функция declval: Для того чтобы все заработало нам понадобилось получить выражения типа T и Args. Возникла проблема: не у всех классов есть конструкторы по умолчанию. Чтобы избавится от этой проблемы мы воспользовались функцией declval. Что же она делает?

```
template<typename T>
T&& declval() noexcept;
```

Двойной амперсанд нужен, чтобы функция работала для incomplete types (тех у которых нет определения. Объекты такого типа нельзя создать, а вот ссылку на них - можно) + тип value не испортится (если T было lvalue/rvalue оно таким и останется).

Заметим, что у функции нет тела, нам оно и не нужно, так как она используется только для проверок в compile-time. Таким образом, можно сказать, что declval - это противоположность decltype.

$$\text{type T} \underset{\text{declval}}{\overset{\text{decltype}}{\longleftrightarrow}} \text{expression of type T}$$

Применение: В STL функция has_method используется, например, в allocator_traits, когда пытаемся определить, есть ли пользовательский метод construct/destroy/etc. или нужно взять дефолтный.

5.22. Реализуйте метафункции is_constructible, is_copy_constru is_move_constructible. Объясните, как работают ваши реализации. Для чего в них нужна функция declval и что она из себя представляет?

```cpp
// T - класс в котором проверяем метод, Args - аргументы метода
template <typename T, typename... Args>
struct is_constructible {
private:
    template<typename TT, typename... AArgs>
    static auto f(int) -> decltype(TT(declval<AArgs>()...), int());
    // возвращаемый тип - int, так как оператор запятая возвращает последний операнд

    template<typename...>
    static char f(...);
public:
    static const bool value = std::is_same_v< decltype(f<T, Args...>(0)),
    int >;
    // будет истина только когда выбралась первая версия функции
};

template<typename T>
using is_copy_constructible = is_constructible<T, const T&>

template<typename T>
using is_move_constructible = is_constructible<T, T&&>
```

Объяснение: Реализация практически ничем не отличается от has_method, объяснение аналогичное. В первой версии пытаемся вызвать конструктор от нужных аргументов, если его нет, то сработает SFINAE. (Про declval см. билет 5.21.)

По определению copy и move конструкторы - это конструкторы от const T& и T&& соответственно, поэтому просто выражаем метафункции для них через is_constructible.