1 Recursion

Implement a lambda expression `f(int x)` that calculates $x!$ (factorial of `x`) by recursion upon calling! Use the following skeleton:

```
int main() {
    auto f = /* TODO */
        return x < 2 ? x : x * f( /* TODO */
    });

    return f(3);
}
```

2 Function traits

Finish the implementation of `function_traits` and `lambda_traits`, such that the program compiles successfully! Note, that it is sufficient for `function_traits` to only accept instances of `std::function`. Think about how you can use your implementation of `function_traits` with lambda expressions and implement this logic in `lambda_traits`. Note further, that invocations of `std::function` are not critical in terms of possible performance penalties in this scenario, as long as the traits are available at compile time (as required in `main`).

```
#include <functional>
#include <tuple>
#include <type_traits>
#include <utility>

template <typename T> struct function_traits;
template <typename R, typename... Args>
struct function_traits<std::function<R(Args...)>> {
    /* TODO */
};

template <typename F>
using lambda_traits = function_traits< /* TODO */ >;

int main() {
    int x = 5;
    auto g = [x](int a, bool invert_a) { return invert_a ? x - a : x + a; };

    using T = lambda_traits<decltype(g)>;
    static_assert(std::is_same_v<T::result_type, int>);
    static_assert(T::nargs == 2);
    static_assert(std::is_same_v<T::arg_type<0>, int>);
    static_assert(std::is_same_v<T::arg_type<1>, bool>);
}
```

`godbolt.org/z/HitKk7`

Hint: You may want to use `std::tuple_element` and `std::declval`.

3 Passing lambdas as argument

Implement `Filters` as an logical *or* filter gathering, such that the program below compiles and runs successfully.

```
#include <cassert>

template <typename T>
struct Filters {
    template <typename F, typename... Fs>
    Filters(F f, Fs... fs) { /* TODO */ }

    [[nodiscard]] auto operator()(T x) const { /* TODO */ }

    template <typename F>
    void add_filter(F f) { /* TODO */ }
};

template <typename... Fs> Filters(Fs... fs) -> Filters< /* TODO */ >;

int main() {
    auto f1 = [](int x) { return x % 2 == 0; };
    auto f2 = [](int x) { return x % 3 == 0; };
    Filters filters{f1, f2};

    assert(filters(1) == false);
    assert(filters(2) == true);
    assert(filters(3) == true);
    assert(filters(4) == true);
    assert(filters(5) == false);
    assert(filters(6) == true);
    assert(filters(7) == false);

    int d = 5;
    auto f3 = [d](int x) { return x % d == 0; };
    filters.add_filter(f3);

    assert(filters(1) == false);
    assert(filters(2) == true);
    assert(filters(3) == true);
    assert(filters(4) == true);
    assert(filters(5) == true);
    assert(filters(6) == true);
    assert(filters(7) == false);
}
```

`godbolt.org/z/kr8_YE`

The constructor of `Filters` should take one or more filters,

```
template <typename F, typename... Fs> Filters(F f, Fs... fs)
```

where each filter in `fs` (and `f` itself) is a lambda (or `std::function`) that accepts an argument of type `T` and returns a boolean. `T` should be generic and is allow to be different amongst the filters. The call operator

```
bool Filters::operator()(T x) const
```

should return the logical *or* of all filters for applied `x` as shown in the example. Further, implement a function

```
template <typename F> void Filters::add_filter(F)
```

that adds a filter to the gathering. Note, that in the example there is no common type in terms of `std::common_type_t` for `f1`, `f2` and `f3`:

```
using T12 = std::common_type_t<decltype(f1), decltype(f2)>; // OK
using T123 = std::common_type_t<T12, decltype(f3)>;         // Compile-time error!
```

If in trouble, follow the step-by-step instructions printed below!

3.1 Step I

Start by finding an implementation for `Filters` that suffices the relaxed requirement

```
template <typename F1, typename F2>
struct Filters {
    Filters(F1 f1, F2 f2) {
        /* TODO */
    }

    [[nodiscard]] auto operator()(int x) const {
        /* TODO */
    }
};

int main() {
    auto f1 = [](int x) { return x % 2 == 0; };
    auto f2 = [](int x) { return x % 3 == 0; };
    Filters filters(f1, f2);
    return filters(5) ? 1 : 0;
}
```

...and answer the question why we need two separate template types `F1` and `F2`!

3.2 Step II

Make the type of the passed argument of a filter (`int`) generic and pass it as a template parameter. You will now need to instantiate `Filters` akin to

```
auto f1 = [](int x) { return x % 2 == 0; };
auto f2 = [](int x) { return x % 3 == 0; };
Filters<decltype(f1), decltype(f2), int> filters(f1, f2);
```

This is unpleasent and we will address this issue in the next step.

3.3 Step III

Use a deduction guide to get rid of the explicit type naming of the filters. This is a delicate task, since we have to find the type of the argument of a passed filter. You may want to have a look at Sec. 2 to find a solution for this problem.

```
template <typename F1, typename F2>
Filters(F1 f1, F2 f2) -> Filters<F1, F2, std::common_type_t< /* TODO */ >>
```

3.4 Step IV

We now generalize our solution for an arbitrary number of filters. Use variadic templates for the constructor and the deduction guide, and store the filters in a `std::vector`! Use `std::function<bool(T)>` as the value type of the vector.

```
template <typename T>
struct Filters {
    std::vector<std::function<bool(T)>> filters;

    template <typename F, typename... Fs>
    Filters(F f, Fs... fs): filters( /* TODO */ ) {}

    /* TODO */
};

template <typename... Fs>
Filters(Fs... fs)
-> Filters<std::common_type_t< /* TODO */ >>;
```

Add an sufficient implementation of

```
template <typename F> void Filters::add_filter(F)
```

and don't forget to adopt your implementation of `Filters.operator()(T)`! (use `std::accumulate` if possible.)

3.5 Step V

Can you think of an alternative to `std::function`? Why can we not use `std::common_type` or `std::variant`?
Hint: Compare the sizes of lambdas with different captures:

```
int capture_me = 1;
std::cout << sizeof([](int x) { return x; }) << '\n';
std::cout << sizeof([y=capture_me](int x) { return x + y; }) << '\n';
std::cout << sizeof([y=&capture_me](int x) { return x + y; }) << '\n';
```

godbolt.org/z/ZBUfgY

3.6 Optional

In case you are wondering what `std::common_type` does; its rules are based on the rules for the ternary operator which can be confusing, e.g.

```
struct S {};

template<class T, int> struct CT {
    operator T() const;
};

int main() {
    auto a = false ? CT<int, 1>{} : CT<int, 2>{};   // OK
    auto b = false ? CT<int*, 1>{} : CT<int*, 2>{}; // OK
    auto c = false ? CT<S*, 1>{} : CT<S*, 2>{};     // OK
    auto d = false ? CT<S, 1>{} : CT<S, 2>{};       // Compile-time error
```

```
}
```

`godbolt.org/z/4TGmK6`

Depending on what compiler you are using, the error message for `d` varies. Take a look at:

- `[over.build] §27`
- `[expr.cond] §6`

if you want to learn more.

4 Solutions

4.1 Recursion

```
#include <utility>

template <typename Callable>
[[nodiscard]] auto make_recursion(Callable f) noexcept {
    return [f]<typename... Args>(Args&&... args) {
        return f(f, std::forward<Args>(args)...);
    };
}

int main() {
    auto f = make_recursion([](auto f, auto x) noexcept -> decltype(x) {
        return x < 2 ? x : x * f(f, x - 1);
    });

    return f(3);
}
```

godbolt.org/z/kf73qG

4.2 Function traits

```
#include <functional>
#include <tuple>
#include <type_traits>
#include <utility>

template <typename T> struct function_traits;
template <typename R, typename... Args>
struct function_traits<std::function<R(Args...)>> {
    using result_type = R;
    static constexpr auto nargs = sizeof...(Args);

    template <std::size_t i>
    using arg_type = std::tuple_element_t<i, std::tuple<Args...>>;
};

template <typename F>
using lambda_traits = function_traits<decltype(std::function{std::declval<F>()})>;

int main() {
    int x = 5;
    auto g = [x](int a, bool invert_a) { return invert_a ? x - a : x + a; };

    using T = lambda_traits<decltype(g)>;
    static_assert(std::is_same_v<T::result_type, int>);
    static_assert(T::nargs == 2);
    static_assert(std::is_same_v<T::arg_type<0>, int>);
    static_assert(std::is_same_v<T::arg_type<1>, bool>);
}
```

godbolt.org/z/c6j8YY

4.3 Passing lambdas as argument

```cpp
#include <cassert>
#include <functional>
#include <numeric>
#include <vector>

template <typename T>
struct Filters {
    std::vector<std::function<bool(T)>> filters;

    template <typename F, typename... Fs>
    Filters(F f, Fs... fs): filters({std::function<bool(T)>{std::move(f)},
                                     std::function<bool(T)>{std::move(fs)}...}) {}

    [[nodiscard]] auto operator()(T x) const {
        auto logical_or = [x](bool acc, auto f) { return acc || f(x); };
        return std::accumulate(filters.begin(), filters.end(), false, logical_or);
    }

    template <typename F>
    void add_filter(F f) {
        filters.emplace_back(std::move(f));
    }
};

template <typename T> struct arg;
template <typename R, typename T> struct arg<std::function<R(T)>> { using type = T; };
template <typename T> using arg_t = typename arg<T>::type;

template <typename... Fs>
Filters(Fs... fs)
-> Filters<std::common_type_t<arg_t<decltype(std::function(fs))>...>>;

int main() {
    auto f1 = [](int x) { return x % 2 == 0; };
    auto f2 = [](int x) { return x % 3 == 0; };
    Filters filters{f1, f2};

    assert(filters(1) == false);
    assert(filters(2) == true);
    assert(filters(3) == true);
    assert(filters(4) == true);
    assert(filters(5) == false);
    assert(filters(6) == true);
    assert(filters(7) == false);

    int d = 5;
    auto f3 = [d](int x) { return x % d == 0; };
    filters.add_filter(f3);

    assert(filters(1) == false);
    assert(filters(2) == true);
    assert(filters(3) == true);
    assert(filters(4) == true);
    assert(filters(5) == true);
    assert(filters(6) == true);
    assert(filters(7) == false);
}
```

godbolt.org/z/wLCZRS