Modern Techniques for Keeping Your Code DRY

Björn Fahller

```
assert(state == IDLE || state == DISCONNECTING || state == DISCONNECTED);
```

```
assert(state == IDLE || state == DISCONNECTING || state == DISCONNECTED);
```

Modern Techniques for Keeping Your Code DRY

Björn Fahller

Modern Techniques for Keeping Your Code DRY

Björn Fahller

Modern Techniques for Keeping Your Code DRY

Björn Fahller

```
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };
```

```
assert(state == IDLE || state == DISCONNECTING || state == DISCONNECTED);
```

```
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };
```

variadic function template

```
assert(state == IDLE || state == DISCONNECTING || state == DISCONNECTED);
```

```cpp
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };

template <typename ... Ts>
bool is_any_of(state_type s, const Ts& ... ts)
{
    return ((s == ts) || ...);
}
```

```cpp
assert(state == IDLE || state == DISCONNECTING || state == DISCONNECTED);
```

```
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };

template <typename ... Ts>
bool is_any_of(state_type s, const Ts& ... ts)
{
    return ((s == ts) || ...);
}
```

```
assert(state == IDLE || state == DISCONNECTING || state == DISCONNECTED);
```

```
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };

template <typename ... Ts>
bool is_any_of(state_type s, const Ts& ... ts)
{
    return ((s == ts) || ...);
}
```

```
assert(state == IDLE || state == DISCONNECTING || state == DISCONNECTED);
```

```cpp
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };

template <typename ... Ts>
bool is_any_of(state_type s, const Ts& ... ts)
{
    return ((s == ts) || ...);
}
```

```cpp
assert(is_any_of(state, IDLE, DISCONNECTING, DISCONNECTED));
```

```
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };

template <typename ... Ts>
bool is_any_of(state_type s, const Ts& ... ts)
{
    return ((s == ts) || ...);
}
```


```
assert(is_any_of(state, IDLE, DISCONNECTING, DISCONNECTED));
```

```
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };
```

variadic non-type template parameter function template

```
assert(state == IDLE || state == DISCONNECTING || state == DISCONNECTED);
```

```cpp
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };

template <state_type ... states>
bool is_any_of(state_type t)
{
    return ((t == states) || ...);
}
```

```cpp
assert(state == IDLE || state == DISCONNECTING || state == DISCONNECTED);
```

```
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };

template <state_type ... states>
bool is_any_of(state_type t)
{
    return ((t == states) || ...);
}
```

```
assert(state == IDLE || state == DISCONNECTING || state == DISCONNECTED);
```

```cpp
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };

template <state_type ... states>
bool is_any_of(state_type t)
{
    return ((t == states) || ...);
}
```

```cpp
assert(is_any_of<IDLE, DISCONNECTING, DISCONNECTED>(state));
```

```
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };

template <auto ... alternatives, typename T>
bool is_any_of(const T& t)
{
    return ((t == alternatives) || ...);
}
```

auto for non-type template parameters from C++17

```
assert(is_any_of<IDLE, DISCONNECTING, DISCONNECTED>(state));
```

```
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };

template <auto ... alternatives, typename T>
bool is_any_of(const T& t)
{
    return ((t == alternatives) || ... );
}
```

Not all types can be used as non-type template parameters

Only `constexpr` values

Yoda speak

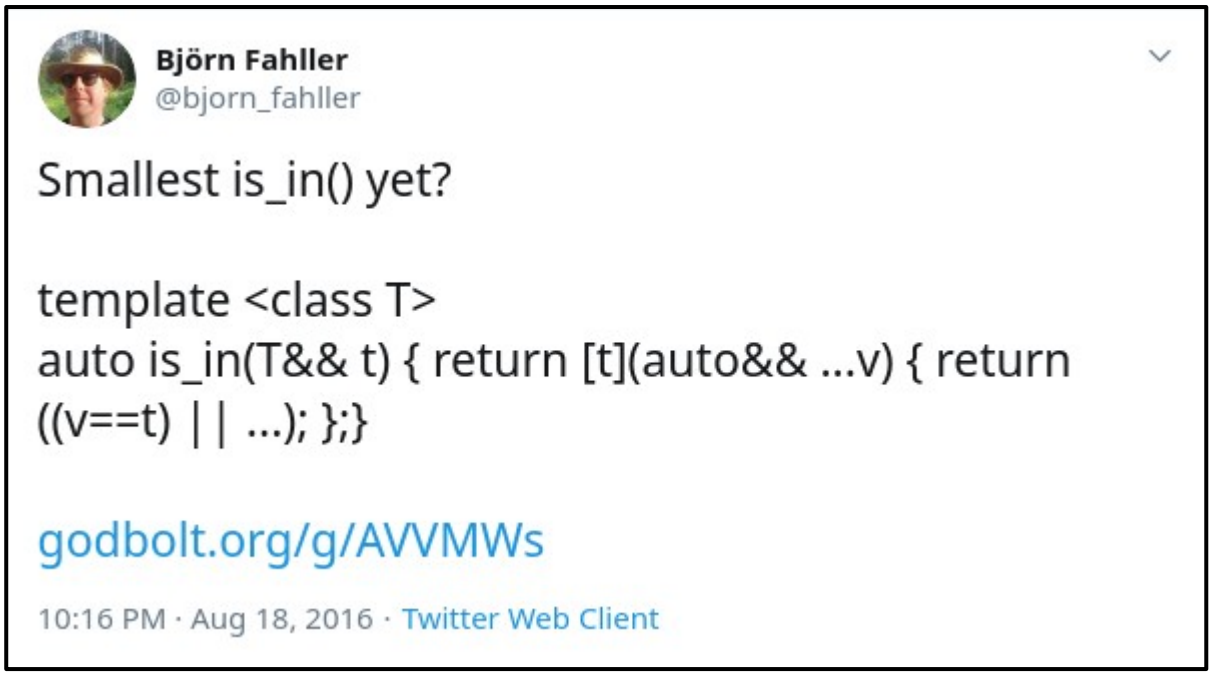
```
assert(is_any_of<IDLE, DISCONNECTING, DISCONNECTED>(state));
```

```
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };
```

construct then test

```
assert(state == IDLE || state == DISCONNECTING || state == DISCONNECTED);
```

```
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };
```




```
assert(state == IDLE || state == DISCONNECTING || state == DISCONNECTED);
```

```cpp
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };

template <typename T>
auto is_in(const T& t)
{
    return [t](const auto& ... vs) { return ((t == vs) || ...); };
}
```

```cpp
assert(state == IDLE || state == DISCONNECTING || state == DISCONNECTED);
```

```
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };

template <typename T>
auto is_in(const T& t)
{
    return [t](const auto& ... vs) { return ((t == vs) || ...); };
}
```

```
assert(is_in(state)(IDLE, DISCONNECTING, DISCONNECTED));
```

```cpp
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };

template <typename T>
auto is_in(const T& t)
{
    return [t](const auto& ... vs) { return ((t == vs) || ...); };
}
```

```cpp
assert(is_in(state)(IDLE, DISCONNECTING, DISCONNECTED));
```

```
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };

template <typename T>
auto is_in(const T& t)
{
    return [t](const auto& ... vs) { return ((t == vs) || ...); };
}
```

Works with all types

Looks horrible

Is a bit too terse

```
assert(is_in(state)(IDLE, DISCONNECTING, DISCONNECTED));
```

```
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };
```

explicit construct and compare function

```
assert(state == IDLE || state == DISCONNECTING || state == DISCONNECTED);
```

```cpp
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };

template <typename T>
struct is {
    T t;
    template <typename ... Ts>
    bool any_of(const Ts& ... ts) const { return ((t == ts) || ...); }
};

template <typename T>
is(T) -> is<T>;
```

```cpp
assert(state == IDLE || state == DISCONNECTING || state == DISCONNECTED);
```

```cpp
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };

template <typename T>
struct is {
   T t;
   template <typename ... Ts>
   bool any_of(const Ts& ... ts) const { return ((t == ts) || ...); }
};

template <typename T>
is(T) -> is<T>;
```

```cpp
assert(state == IDLE || state == DISCONNECTING || state == DISCONNECTED);
```

```cpp
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };

template <typename T>
struct is {
   T t;
   template <typename ... Ts>
   bool any_of(const Ts& ... ts) const { return ((t == ts) || ...); }
};
```

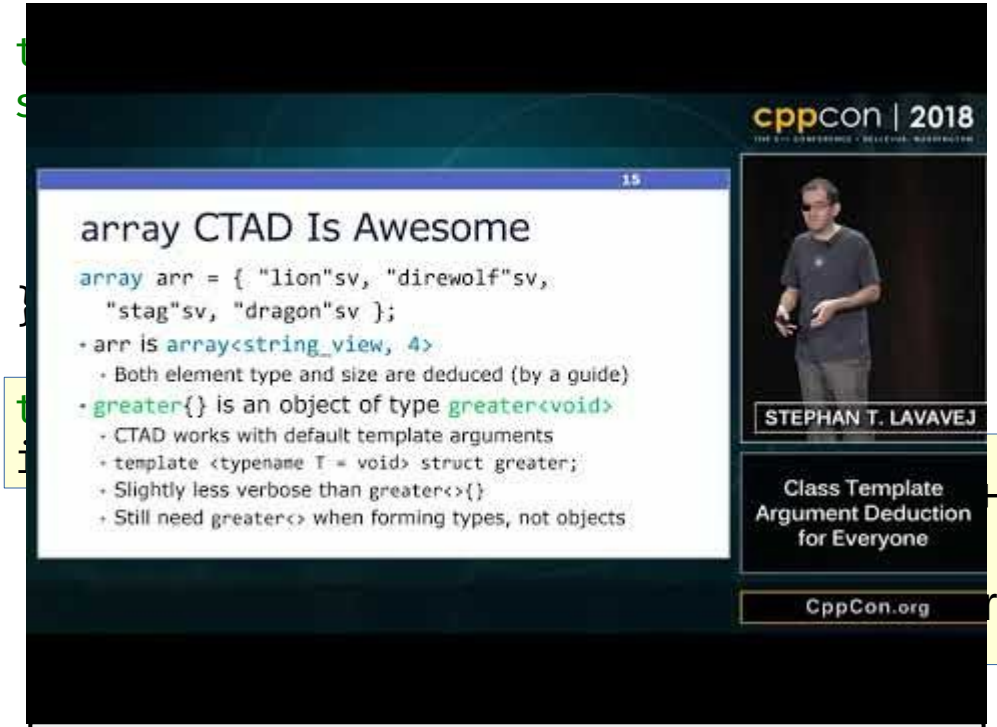
```cpp
template <typename T>
is(T) → is<T>;
```

C++17 Deduction Guide for

Constructor Template Argument Deduction

```cpp
assert(state == IDLE || state == DISCONNECTING || state == DISCONNECTED);
```

```
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };
```



https://youtu.be/-H-ut6j1BYU

https://youtu.be/UDs90b0yjjQ

```
assert(state == IDLE || state == DISCONNECTING || state == DISCONNECTED);
```

```
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };

template <typename T>
struct is {
    T t;
    template <typename ... Ts>
    bool any_of(const Ts& ... ts) const { return ((t == ts) || ...); }
};

template <typename T>
is(T) → is<T>;
```

Works with all types

Explicit

Still Yoda speak

```
assert(is{state}.any_of(IDLE, DISCONNECTING, DISCONNECTED));
```

```
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };
```

time passes...

```
assert(state == IDLE || state == DISCONNECTING || state == DISCONNECTED);
```

```
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };
```

your mildly annoyed developer stays annoyed...

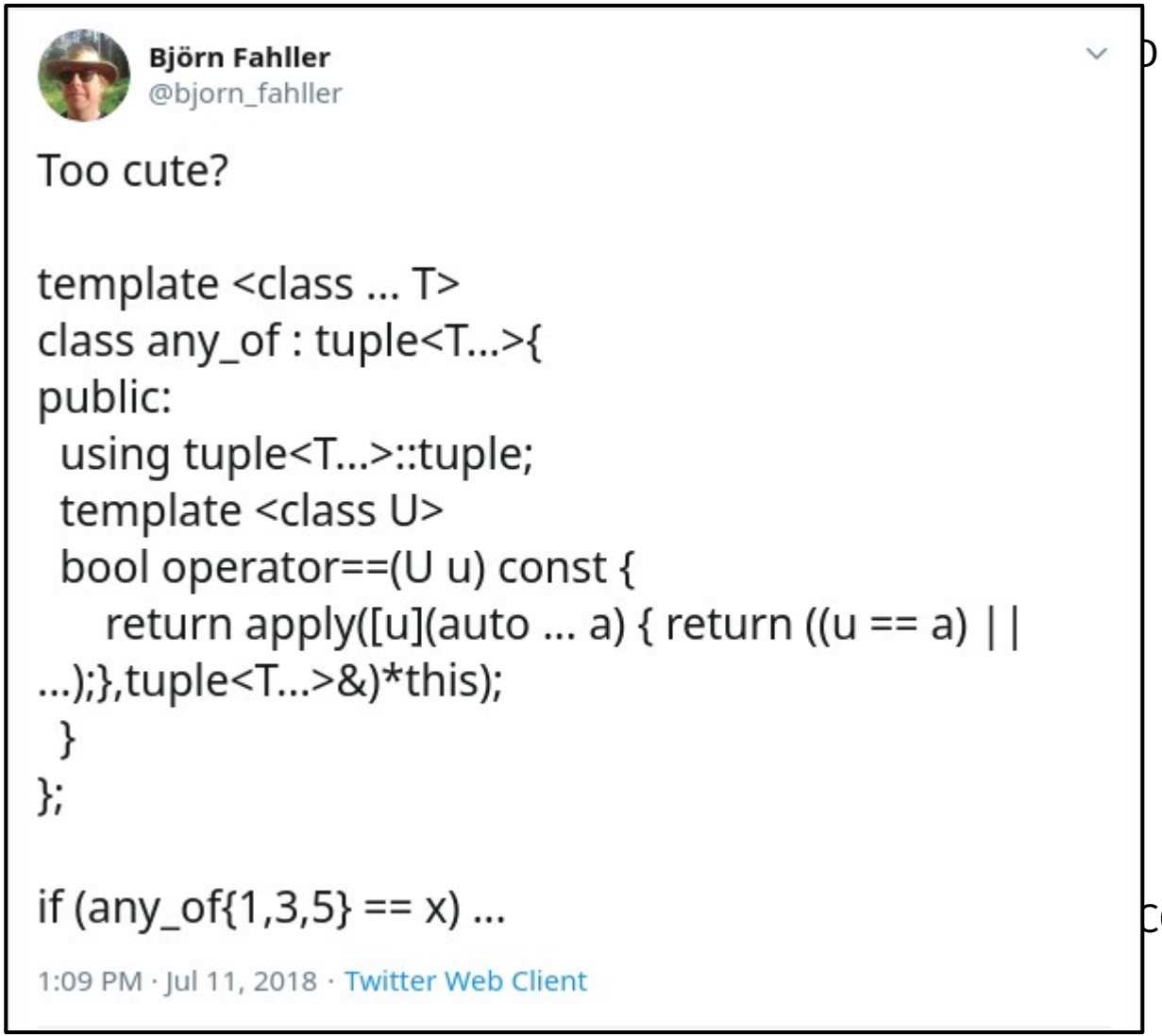
```
assert(state == IDLE || state == DISCONNECTING || state == DISCONNECTED);
```

```
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };
```

Then one day, two years later!

```
assert(state == IDLE || state == DISCONNECTING || state == DISCONNECTED);
```

```
enum state_type { ... DISCONNECTED };
```




```
assert(state == ... CONNECTED);
```

```cpp
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };

template <typename ... Ts>
struct any_of : private std::tuple<Ts...>
{
    using std::tuple<Ts...>::tuple;
    template <typename T>
    bool operator==(const T& t) const {
        return std::apply([&t](const auto&… ts){ return ((ts == t) || ...);},
                          static_cast<const std::tuple<Ts...>&>(*this));
    }


};

template <typename ... Ts>
any_of(Ts...) -> any_of<Ts...>;

assert(any_of(IDLE, DISCONNECTING, DISCONNECTED) == state);
```

```cpp
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };

template <typename ... Ts>
struct any_of : private std::tuple<Ts...>
{
    using std::tuple<Ts...>::tuple;
    template <typename T>
    bool operator==(const T& t) const {
        return std::apply([&t](const auto&... ts){ return ((ts == t) || ...);},
                          static_cast<const std::tuple<Ts...>&>(*this));
    }


};

template <typename ... Ts>
any_of(Ts...) -> any_of<Ts...>;

assert(any_of(IDLE, DISCONNECTING, DISCONNECTED) == state);
```

Pure laziness, but these give me 18 constructors

```cpp
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };

template <typename ... Ts>
struct any_of : private std::tuple<Ts...>
{
    using std::tuple<Ts...>::tuple;
    template <typename T>
    bool operator==(const T& t) const {
        return std::apply([&t](const auto& ... ts){ return ((ts == t) || ...);},
                          static_cast<const std::tuple<Ts...>&>(*this));
    }


};

template <typename ... Ts>
any_of(Ts...) -> any_of<Ts...>;

assert(state == any_of(IDLE, DISCONNECTING, DISCONNECTED));
```

Call function with each member of the tuple as a parameter.

```cpp
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };

template <typename ... Ts>
struct any_of : private std::tuple<Ts...>
{
    using std::tuple<Ts...>::tuple;
    template <typename T>
    bool operator==(const T& t) const {
        return std::apply([&t](const auto& ... ts){ return ((ts == t) || ...);},
                          static_cast<const std::tuple<Ts...>&>(*this));
    }
    template <typename T>
    friend bool operator==(const T& lh, const any_of& rh) { return rh == lh;}
};

template <typename ... Ts>
any_of(Ts...) -> any_of<Ts...>;

assert(state == any_of(IDLE, DISCONNECTING, DISCONNECTED));
```

Provide symmetric operator==

```cpp
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };

template <typename ... Ts>
struct any_of : private std::tuple<Ts...>
{
    using std::tuple<Ts...>::tuple;
    template <typename T>
    bool operator==(const T& t) const {
        return std::apply([&t](const auto&... ts){ ret      (ts == t) || ...);},
                          static_cast<const std::tup     s...>&>(*this));
    }
    template <typename T>
    friend bool operator==(const T& lh, const an     _of& rh) { return rh == lh;}
};

template <typename ... Ts>
any_of(Ts...) -> any_of<Ts...>;

assert(state == any_of(IDLE, DISCONNECTING, DISCONNECTED));
```

Maybe exaggerated cuteness, but I like this!

```
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };

template <typename ... Ts>
struct any_of : private std::tuple<Ts...>
{
    using std::tuple<Ts...>::tuple;
    template <typename T>
    bool operator==(const T& t) const {
        return std::apply([&t](const auto&... ts){ return ((ts == t) || ...);},
                                                                              ;}
};

template <typename ... Ts>
any_of(Ts...) -> any_of<Ts...>;

assert(state == any_of(IDLE, DISCONNECTING, DISCONNECTED));
```

https://gcc.godbolt.org/z/IY865r

```cpp
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };

template <typename ... Ts>
struct any_of : private std::tuple<Ts...>
{
    using std::tuple<Ts...>::tuple;
    template <typename T>
    bool operator==(const T& t) const {
        return std::apply([&t](const auto&... ts){ return ((ts == t) || ...);},
                          static_cast<const std::tuple<Ts...>&>(*this));
    }
    template <typename T>
    friend bool operator==(const T& lh, const any_of& rh) { return rh == lh;}
};

template <typename ... Ts>
any_of(Ts...) -> any_of<Ts...>;

assert(state == any_of(IDLE, DISCONNECTING, DISCONNECTED));
```

```
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };

template <typename ... Ts>
struct any_of : private std::tuple<Ts...>
{
    using std::tuple<Ts...>::tuple;
    template <typename T>
    bool operator==(const T& t) const {
        return std::apply([&t](const auto& ... ts){
                          static_cast<const std::tup
    }
    template <typename T>
    friend bool operator==(const T& lh, const any_of
};

template <typename ... Ts>
any_of(Ts...) -> any_of<Ts...>;

assert(state == any_of(IDLE, DISCONNECTING, DISCONNECTED));
```

Add relational operators!

```cpp
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };

template <typename ... Ts>
struct any_of : private std::tuple<Ts...>
{
    using std::tuple<Ts...>::tuple;
    template <typename T>
    bool operator==(const T& t) const {
        return std::apply([&t](const auto&... ts){ return ((ts == t) || ...);},
                          static_cast<const std::tuple<Ts...>&>(*this));
    }
    template <typename T>
    bool operator<(const T& t) const {
        return std::apply([&t](const auto&... ts){ return ((ts < t) || ...);},
                          static_cast<const std::tuple<Ts...>&>(*this));
    }
    ...
};

while (any_of(a, b, c) < 0) ...
```

```
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };

template <typename ... Ts>
struct any_of : private std::tuple<Ts ... >
{
    using std::tuple<Ts ... >::tuple;
    template <typename T>
    bool operator==(const T& t) const {
        return std::apply([&t](const auto& ... ts){ ret          == t) || ... );},
                          static_cast<const std::          this));
    }
    template <typename T>
    bool operator<(const T& t) const {
        return std::apply([&t](const auto& ... t          < t) || ... );},
                          static_cast<const std::          >&>(*this));
    }
     ...
};

while (any_of(a, b, c) < 0) ...
```

```
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };

template <typename ... Ts>
str
{
```

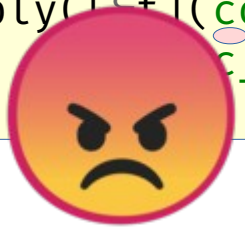
https://gcc.godbolt.org/z/YyvAtT

```
    bool operator==(const T& t) const {
        return std::apply([&t](const auto& ... ts){ ret = t) || ... );},
                          static_cast<const std:: this));
    }
    template <typename T>
    bool operator<(const T& t) const {
        return std::apply([&t](const auto& ... t < t) || ... );},
                          static_cast<const std:: >&>(*this));
    }
    ...
};

while (any_of(a, b, c) < 0) ...
```

```
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };

template <typename ... Ts>
struct any_of : private std::tuple<Ts...>
{
    using std::tuple<Ts...>::tuple;
    template <typename T>
    bool operator==(const T& t) const {
        return std::apply([&t](const auto&... ts){ return ((ts == t) || ...);},
                          static_cast<const std::tuple<Ts...>&>(*this));
    }
    template <typename T>
    bool operator<(const T& t) const {
        return std::apply([&t](const auto&... ts){ return ((ts < t) || ...);},
                          static_cast<const std::tuple<Ts...>&>(*this));
    }
    ...
};

while (any_of(a, b, c) < 0) ...
```

```
enum state_type { IDLE, CONNECTING, CONNECTED, DISCONNECTING, DISCONNECTED };

template <typename ... Ts>
struct any_of : private std::tuple<Ts ... >
{
    using std::tuple<Ts ... >::tuple;
    template <typename T>
    bool operator==(const T& t) con
        return std::apply([&t](const
                           static_cast<
    }
    template <typename T>
    bool operator<(const T& t) const {
        return std::apply([&t](const auto& ... ts){ return ((ts < t) || ... );},
                           c_cast<const std::tuple<Ts ... >&>(*this));
    }
    ...
};

while (any_of(a, b, c) < 0) ...
```

An awful lot of repetition here!

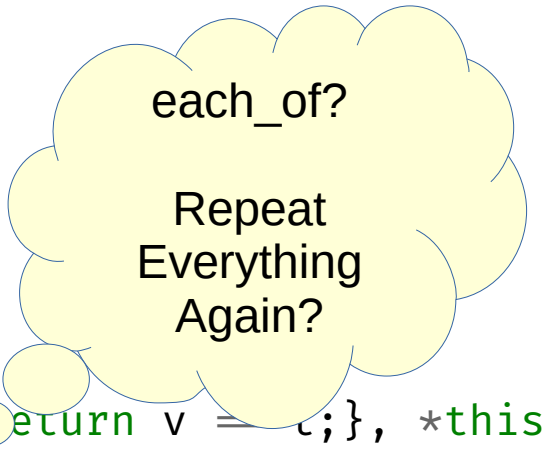


```cpp
template <typename F, typename ... Ts>
bool or_elements(const F& f, const std::tuple<Ts...>& t)
{
    return std::apply([&f](const auto& ... ts){ return (f(ts) || ...);}, t);
}
```

```cpp
template <typename F, typename ... Ts>
bool or_elements(const F& f, const std::tuple<Ts...>& t)
{
    return std::apply([&f](const auto& ... ts){ return (f(ts) || ...);}, t);
}

template <typename ... Ts>
struct any_of : private std::tuple<Ts...>
{
    using std::tuple<Ts...>::tuple;
    template <typename T>
    bool operator==(const T& t) const {
        return or_elements([&t](const auto& v){ return v == t;}, *this);
    }
    template <typename T>
    bool operator<(const T& t) const {
        return or_elements([&t](const auto& v){ return v < t;}, *this);
    }
};

while (any_of(a, b, c) < 0) ...
```

```cpp
template <typename F, typename ... Ts>
bool or_elements(const F& f, const std::tuple<Ts...>& t)
{
    return std::apply([&f](const auto& ... ts){ return (f(ts) || ...);}, t);
}

template <typename ... Ts>
struct any_of : private std::tuple<Ts...>
{
    https://godbolt.org/z/SP8POJ
    }
    template <typename T>
    bool operator<(const T& t) const {
        return or_elements([&t](const auto& v){ return v < t;}, *this);
    }
};

while (any_of(a, b, c) < 0) ...
```

```cpp
template <typename F, typename ... Ts>
bool or_elements(const F& f, const std::tuple<Ts...>& t)
{
    return std::apply([&f](const auto& ... ts){ return (f(ts) || ...);}, t);
}

template <typename ... Ts>
struct any_of : private std::tuple<Ts...>
{
    using std::tuple<Ts...>::tuple;
    template <typename T>
    bool operator==(const T& t) const {
        return or_elements([&t](const auto& v){ return v == t;}, *this);
    }
    template <typename T>
    bool operator<(const T& t) const {
        return or_elements([&t](const auto& v){ return v < t;}, *this);
    }
};

while (any_of(a, b, c) < 0) ...
```



```
struct or_elements {
    template <typename F, typename ... Ts>
    static bool apply(const F& f, const std::tuple<Ts...>& t) {
        return std::apply([&](const auto& ... ts){ return (f(ts) || ...);}, t);
    }
};

template <typename Op, typename ... Ts>
struct op_t : private std::tuple<Ts...> {
    using std::tuple<Ts...>::tuple;
    template <typename T>
    bool operator==(const T& t) const {
        return Op::apply([&t](const auto& v){ return v == t;}, *this);
    }
    ...
};

template <typename ... Ts>
struct any_of : op_t<or_elements, Ts...> {
    using op_t<or_elements, Ts...>::op_t;
};
```

```
struct or_elements {
    template <typename F, typename ... Ts>
    static bool apply(const F& f, const std::tuple<Ts...>& t) {
        return std::apply([&](const auto& ... ts){ return (f(ts) || ...);}, t);
    }
};

template <typename Op, typename ... Ts>
struct op_t : private std::tuple<Ts...> {
    using std::tuple<Ts...>::tuple;
    template <typename T>
    bool operator==(const T& t) const {
        return Op::apply([&t](const auto& v){ return v == t;}, *this);
    }
    ...
};

template <typename ... Ts>
struct any_of : op_t<or_elements, Ts...> {
    using op_t<or_elements, Ts...>::op_t;
};
```

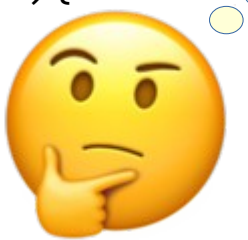
```
struct or_elements {
    template <typename F, typename ... Ts>
    static bool apply(const F& f, const std::tuple<Ts...>& t) {
        return std::apply([&](const auto& ... ts){ return (f(ts) || ...);}, t);
    }
};

template <typename Op, typename ... Ts>
struct op_t : private std::tuple<Ts...> {
    using std::tuple<Ts...>::tuple;
    template <typename T>
    bool operator==(const T& t) const {
        return Op::apply([&t](const auto& v){ return v == t;}, *this);
    }
    ...
};

template <typename ... Ts>
struct any_of : op_t<or_elements, Ts...> {
    using op_t<or_elements, Ts...>::op_t;
};
```

```
struct or_elements {
    template <typename F, typename ... Ts>
    static bool apply(const F& f, const std::tuple<Ts...>& t) {
        return std::apply([&](const auto& ... ts){ return (f(ts) || ...);}, t);
    }
};

template <typename Op, typename ... Ts>
struct op_t : private std::tuple<Ts...> {
    using std::tuple<Ts...>::tuple;
    template <typename T>
    bool operator==(const T& t) const {
        return Op::apply([&t](const auto& v){ return v == t;}, *this);
    }
    ...
};

template <typename ... Ts>
struct any_of : op_t<or_elements, Ts...> {
    using op_t<or_elements, Ts...>::op_t;
};
```

```
struct and_elements {
    template <typename F, typename ... Ts>
    static bool apply(const F& f, const std::tuple<Ts...>& t) {
        return std::apply([&](const auto& ... ts){ return (f(ts) && ...);}, t);
    }
};

template <typename Op, typename ... Ts>
struct op_t : private std::tuple<Ts...> {
    using std::tuple<Ts...>::tuple;
    template <typename T>
    bool operator==(const T& t) const {
        return Op::apply([&t](const auto& v){ return v == t;}, *this);
    }
    ...
};

template <typename ... Ts>
struct each_of : op_t<and_elements, Ts...> {
    using op_t<and_elements, Ts...>::op_t;
};
```

```
struct and_elements {
    template <typename F, typename ... Ts>
    static bool apply(const F& f, const std::tuple<Ts...>& t) {
        return std::apply([&](const auto& ... ts){ return (f(ts) && ...);}, t);
    }
};

template <typename ... Ts>
using each_of = op_t<and_elements, Ts...>;

template <typename ... Ts>
struct op_t : private std::tuple<Ts...> {
    using std::tuple<Ts...>::tuple;
    template <typename T>
    bool operator==(const T& t) const {
        return Op::apply([&t](const auto& v){ return v == t;}, *this);
    }
    ...
};
```

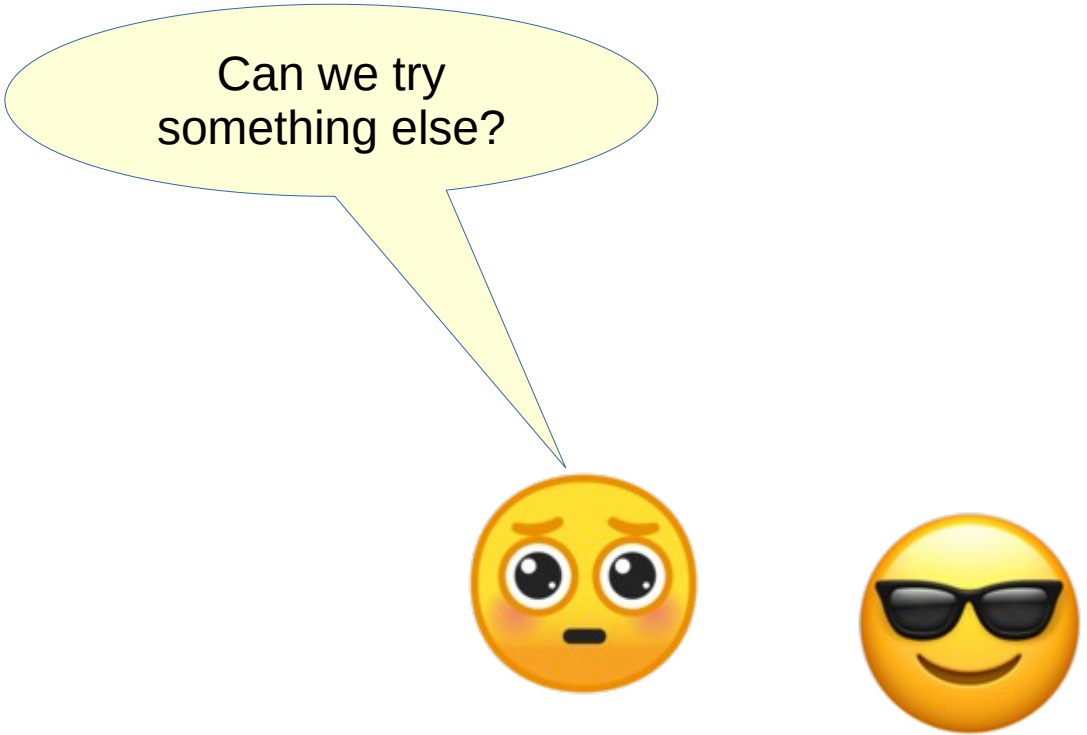
In a better world we could deduce constructor template args for aliases too

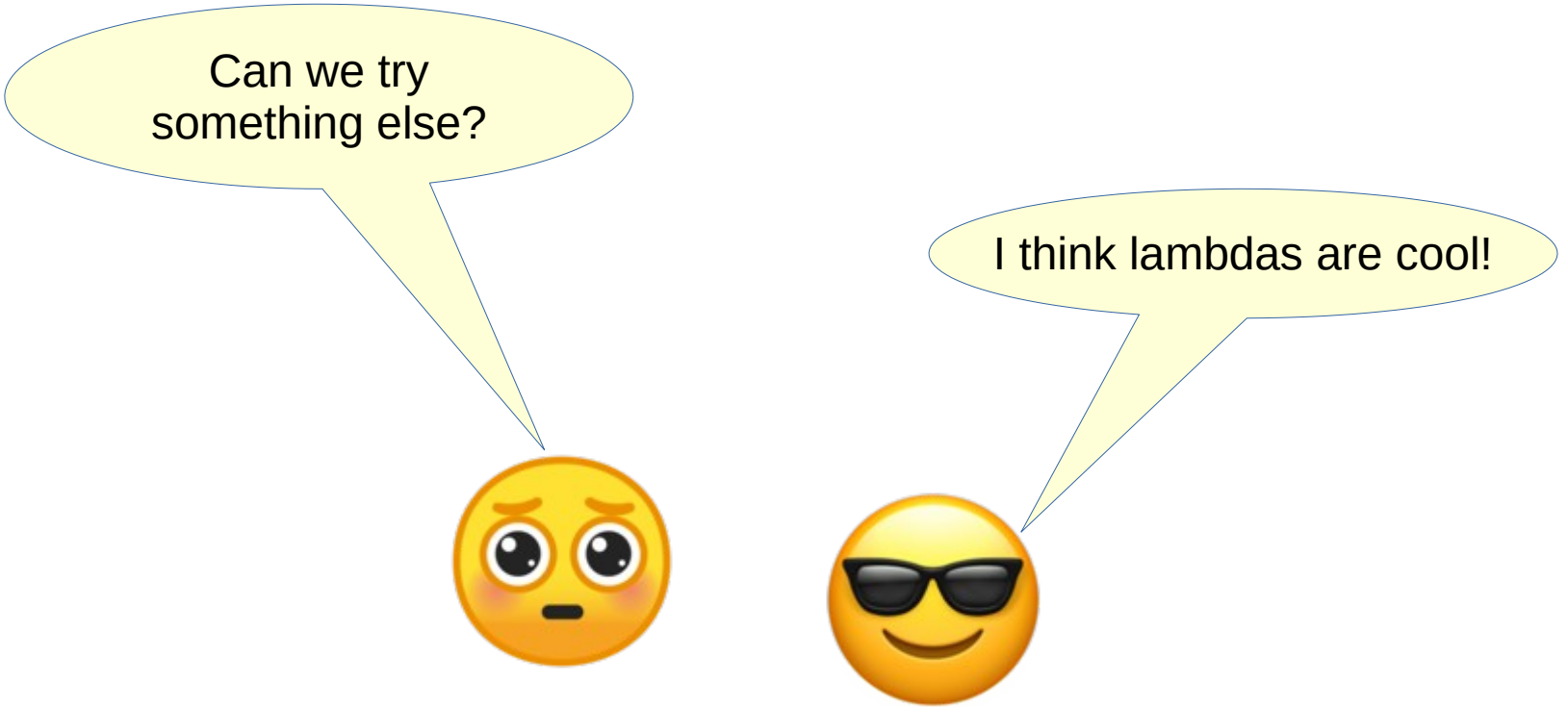


```
template <typename ... Ts>
struct each_of : op_t<and_elements, Ts...> {
    using op_t<and_elements, Ts...>::op_t;
};
```

```
struct and_elements {
    template <typename F, typename ... Ts>
    static bool apply(const F& f, const std::tuple<Ts...>& t) {
        return std::apply([&](const auto& ... ts){ return (f(ts) && ...);}, t);
    }
};

template <typename Op, typename ... Ts>
struct op_t : private std::tuple<Ts...> {
    }
    ...
};

template <typename ... Ts>
struct each_of : op_t<and_elements, Ts...> {
    using op_t<and_elements, Ts...>::op_t;
};
```

https://gcc.godbolt.org/z/Ay1TA_


Can we try
something else?


Can we try
something else?
I think lambdas are cool!

```
constexpr auto tuple = [](auto ... ts)
{
    return [=](const auto& func) { return func(ts ... ); };
};
```

```
assert(tuple(a,b,c)([](auto ... e){ return ((e > 0) && ... ); }));
```

```
constexpr auto tuple = [](auto ... ts)
{
    return [=](const auto& func) { return func(ts ... ); };
};
```

```
assert(tuple(a,b,c)([](auto ... e){ return ((e > 0) && ... ); }));
```

```
constexpr auto tuple = [](auto ... ts)
{
    return [=](const auto& func) { return func(ts ... ); };
};
```

```
assert(tuple(a,b,c)([](auto ... e){ return ((e > 0) && ... ); }));
```

```
constexpr auto tuple = [](auto ... ts)
{
    return [=](const auto& func) { return func(ts ... ); };
};

constexpr auto and_elements = [](auto func)
{
    return [=](auto ... elements) { return (func(elements) && ... ); };
};
```

```
assert(tuple(a,b,c)(and_elements([](auto e){ return e > 0;})));
```

```
constexpr auto tuple = [](auto ... ts)
{
    return [=](const auto& func) { return func(ts ... ); };
};

constexpr auto and_elements = [](auto func)
{
    return [=](auto ... elements) { return (func(elements) && ... ); };
};
```

```
assert(tuple(a,b,c)(and_elements([](auto e){ return e > 0;})));
```

```
constexpr auto tuple = [](auto ... ts)
{
    return [=](const auto& func) { return func(ts ... ); };
};

constexpr auto and_elements = [](auto func)
{
    return [=](auto ... elements) { return (func(elements) && ... ); };
};
```

```
assert(tuple(a,b,c)(and_elements([](auto e){ return e > 0;})));
```

```
constexpr auto tuple = [](auto ... ts)
{
    return [=](const auto& func) { return func(ts ... ); };
};

constexpr auto and_elements = [](auto func)
{
    return [=](auto ... elements) { return (func(elements) && ... ); };
};

constexpr auto bind_rh = [](auto func, auto rh)
{
    return [=](auto lh) { return func(lh, rh); };
};
```

```
assert(tuple(a,b,c)(and_elements([](auto e){ return e > 0;})));
```

```
constexpr auto tuple = [](auto ... ts)
{
    return [=](const auto& func) { return func(ts ... ); };
};

constexpr auto and_elements = [](auto func)
{
    return [=](auto … elements) { return (func(elements) && ... ); };
};

constexpr auto bind_rh = [](auto func, auto rh)
{
    return [=](auto lh) { return func(lh, rh); };
};

constexpr auto greater_than = [](auto rh)
{
    return bind_rh(std::greater{}, rh);
};

assert(tuple(a,b,c)(and_elements([](auto e){ return e > 0;})));
```

```
constexpr auto tuple = [](auto ... ts)
{
    return [=](const auto& func) { return func(ts ... ); };
};

constexpr auto and_elements = [](auto func)
{
    return [=](auto … elements) { return (func(elements) &&  ... ); };
};

constexpr auto bind_rh = [](auto func, auto rh)
{
    return [=](auto lh) { return func(lh, rh); };
};

constexpr auto greater_than = [](auto rh)
{
    return bind_rh(std::greater{}, rh);
};

assert(tuple(a,b,c)(and_elements(greater_than(0))));
```

```
constexpr auto tuple = [](auto ... ts)
{
    return [=](const auto& func) { return func(ts ... ); };
};

constexpr auto and_elements = [](auto func)
{
    return [=](auto … elements) { return (func(elements) &&  ... ); };
};

constexpr auto bind_rh = [](auto func, auto rh)
{
    return [=](auto lh) { return func(lh, rh); };
};

constexpr auto greater_than = [](auto rh)
{
    return bind_rh(std::greater{}, rh);
};

assert(tuple(a,b,c)(and_elements(greater_than(0))));
```

```
constexpr auto tuple = [](auto ... ts) ...
constexpr auto and_elements = [](auto func) ...
constexpr auto equal_to = [](auto rh) ...
constexpr auto greater_than = [](auto rh) ...

template <typename Func, typename Tuple>
class op_t {
    Tuple tup;
    Func func;
    template <typename F>
    auto apply(F f) const{ return tup(func(f)); }
public:
    op_t(Func f, Tuple t) : tup(t), func(f) {}
    template <typename T>
    auto operator==(const T& t) const { return apply(equal_to(t)); }
    template <typename T>
    auto operator>(const T& t) const{ return apply(greater_than(t)); }
};
constexpr auto each_of=[](auto ... ts){ return op_t(and_elements,tuple(ts ... ));};

assert(each_of(a,b,c) > 0);
```

```
constexpr auto tuple = [](auto ... ts) ...
constexpr auto and_elements = [](auto func) ...
constexpr auto equal_to = [](auto rh) ...
constexpr auto greater_than = [](auto rh) ...

template <typename Func, typename Tuple>
class op_t {
    Tuple tup;
    Func func;
    template <typename F>
    auto apply(F f) const{ return tup(func(f)); }
public:
    op_t(Func f, Tuple t) : tup(t), func(f) {}
    template <typename T>
    auto operator==(const T& t) const { return apply(equal_to(t)); }
    template <typename T>
    auto operator>(const T& t) const{ return apply(greater_than(t)); }
};
constexpr auto each_of=[](auto ... ts){ return op_t(and_elements,tuple(ts ... ));};

assert(each_of(a,b,c) > 0);
```

```
constexpr auto tuple = [](auto ... ts) ...
constexpr auto and_elements = [](auto func) ...
constexpr auto equal_to = [](auto rh) ...
constexpr auto greater_than = [](auto rh) ...

template <typename Func, typename Tuple>
class op_t {
    Tuple tup;
    Func func;
    template <typename F>
    auto apply(F f) const{ return tup(func(f)); }
public:
    op_t(Func f, Tuple t) : tup(t), func(f) {}
    template <typename T>
    auto operator==(const T& t) const { return apply(equal_to(t)); }
    template <typename T>
    auto operator>(const T& t) const{ return apply(greater_than(t)); }
};
constexpr auto each_of=[](auto ... ts){ return op_t(and_elements,tuple(ts ... ));};

assert(each_of(a,b,c) > 0);
```

```
constexpr auto tuple = [](auto ... ts) ...
constexpr auto and_elements = [](auto func) ...
constexpr auto equal_to = [](auto rh) ...
constexpr auto greater_than = [](auto rh) ...

template <typename Func, typename Tuple>
class op_t {
    Tuple tup;
    Func func;
    template <typename F>
    auto apply(F f) const{ return tup(func(f)); }
public:
    op_t(Func f, Tuple t) : tup(t), func(f) {}
    template <typename T>
    auto operator==(const T& t) const { return apply(equal_to(t)); }
    template <typename T>
    auto operator>(const T& t) const{ return apply(greater_than(t)); }
};
constexpr auto each_of=[](auto ... ts){ return op_t(and_elements,tuple(ts ... ));};

assert(each_of(a,b,c) > 0);
```

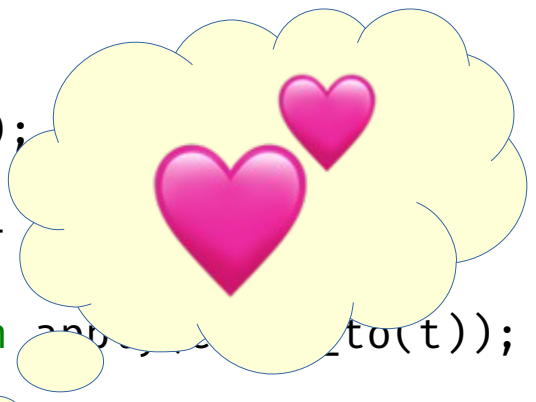
```
constexpr auto tuple = [](auto ... ts) ...
constexpr auto and_elements = [](auto func) ...
constexpr auto equal_to = [](auto rh) ...
constexpr auto greater_than = [](auto rh) ...

template <typename Func, typename Tuple>
class op_t {
    Tuple tup;
    Func func;
    template <typename F>
    auto apply(F f) const{ return tup(func(f)); }
public:
    op_t(Func f, Tuple t) : tup(t), func(f) {}
    template <typename T>
    auto operator==(const T& t) const { return apply(equal_to(t)); }
    template <typename T>
    auto operator>(const T& t) const{ return apply(greater_than(t)); }
};
constexpr auto each_of=[](auto ... ts){ return op_t(and_elements,tuple(ts ... ));};

assert(each_of(a,b,c) > 0);
```

```
constexpr auto tuple = [](auto ... ts) ...
constexpr auto and_elements = [](auto func) ...
constexpr auto equal_to = [](auto rh) ...
constexpr auto greater_than = [](auto rh) ...

template <typename Func, typename Tuple>
class op_t {
    Tuple tup;
    [[no_unique_address]] Func func;
    template <typename F>
    auto apply(F f) const{ return tup(func(f)); }
public:
    op_t(Func f, Tuple t) : tup(t), func(f) {}
    template <typename T>
    auto operator==(const T& t) const { return apply(equal_to(t)); }
    template <typename T>
    auto operator>(const T& t) const{ return apply(greater_than(t)); }
};
constexpr auto each_of=[](auto ... ts){ return op_t(and_elements,tuple(ts ... ));};

assert(each_of(a,b,c) > 0);
```

C++20 attribute to make empty member take no space.

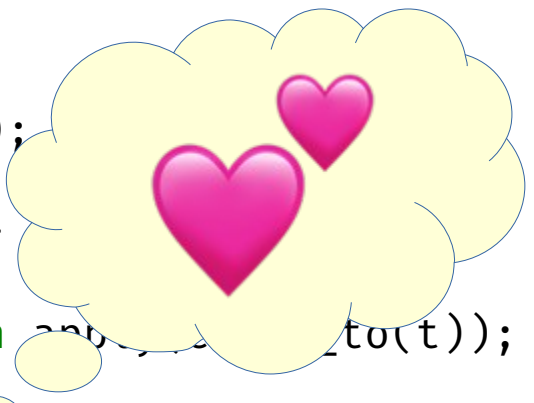
If saving this byte is important pre C++20, use private inheritance for Empty Base Class Optimization.

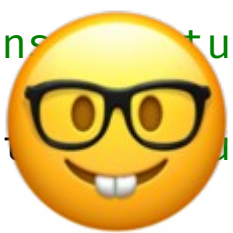
```
constexpr auto tuple = [](auto ... ts) ...
constexpr auto and_elements = [](auto func) ...
constexpr auto equal_to = [](auto rh) ...
constexpr auto greater_than = [](auto rh) ...

template <typename Func, typename Tuple>
class op_t {
    Tuple tup;
    [[no_unique_address]] Func func;
    template <typename F>
    auto apply(F f) const{ return tup(func(f));
public:
    op_t(Func f, Tuple t) : tup(t), func(f) {}
    template <typename T>
    auto operator==(const T& t) const { return ap                to(t)); }
    template <typename T>
    auto operator>(const T& t) cons        turn apply(greater_than(t)); }
};
constexpr auto each_of=[](auto ...        urn op_t(and_elements,tuple(ts ... ));};

assert(each_of(a,b,c) > 0);
```



```
constexpr auto tuple = [](auto ... ts) ...
constexpr auto and_elements = [](auto func) ...
constexpr auto equal_to = [](auto rh) ...
con

tem
cla
    Tuple tup;
    [[no_unique_address]] Func func;
    template <typename F>
    auto apply(F f) const{ return tup(func(f));
public:
    op_t(Func f, Tuple t) : tup(t), func(f) {}
    template <typename T>
    auto operator==(const T& t) const { return ap                to(t)); }
    template <typename T>
    auto operator>(const T& t) cons          turn apply(greater_than(t)); }
};
constexpr auto each_of=[](auto ...          urn op_t(and_elements,tuple(ts ... ));};

assert(each_of(a,b,c) > 0);
```

https://gcc.godbolt.org/z/uyDzC8





```
constexpr auto tuple = [](auto ... ts) ...
constexpr auto and_elements = [](auto func) ...
constexpr auto equal_to = [](auto rh) ...
constexpr auto greater_than = [](auto rh)

template <typename Func, typename Tuple>
class op_t {
    Tuple tup;
    [[no_unique_address]] Func func;
    template <typename F>
    auto apply(F f) const{ return tup(fun
public:
    op_t(Func f, Tuple t) : tup(t), func(
    template <typename T>
    auto operator==(const T& t) const
    template <typename T>
    auto operator>(const T& t) const{ return apply(greater_than(t)); }
};
constexpr auto each_of=[](auto ... ts){ return op_t(and_elements,tuple(ts ... ));};

assert(each_of(a,b,c) > 0);
```

Things to improve:

```
constexpr auto tuple = [](auto ... ts) ...
constexpr auto and_elements = [](auto func) ...
constexpr auto equal_to = [](auto rh) ...
constexpr auto greater_than = [](auto rh)

template <typename Func, typename Tuple>
class op_t {
    Tuple tup;
    [[no_unique_address]] Func func;
    template <typename F>
    auto apply(F f) const{ return tup(fun
public:
    op_t(Func f, Tuple t) : tup(t), func(
    template <typename T>
    auto operator==(const T& t) const
    template <typename T>
    auto operator>(const T& t) const{ return apply(greater_than(t)); }
};
constexpr auto each_of=[](auto ... ts){ return op_t(and_elements,tuple(ts ... ));};

assert(each_of(a,b,c) > 0);
```

Things to improve:

- constexpr

```
constexpr auto tuple = [](auto ... ts) ...
constexpr auto and_elements = [](auto func) ...
constexpr auto equal_to = [](auto rh) ...
constexpr auto greater_than = [](auto rh)

template <typename Func, typename Tuple>
class op_t {
    Tuple tup;
    [[no_unique_address]] Func func;
    template <typename F>
    constexpr auto apply(F f) const{ retu
public:
    constexpr op_t(Func f, Tuple t) : tup
    template <typename T>
    constexpr auto operator==(const T&                    (t)); }
    template <typename T>
    constexpr auto operator>(const T& t) const{ return apply(greater_than(t));}
};
constexpr auto each_of=[](auto ... ts){ return op_t(and_elements,tuple(ts ... ));};

assert(each_of(a,b,c) > 0);
```

Things to improve:

- constexpr

```
constexpr auto tuple = [](auto ... ts) ...
constexpr auto and_elements = [](auto func) ...
constexpr auto equal_to = [](auto rh) ...
constexpr auto greater_than = [](auto rh)

template <typename Func, typename Tuple>
class op_t {
    Tuple tup;
    [[no_unique_address]] Func func;
    template <typename F>
    constexpr auto apply(F f) const{ retu
public:
    constexpr op_t(Func f, Tuple t) : tup
    template <typename T>
    constexpr auto operator==(const T&                    (t)); }
    template <typename T>
    constexpr auto operator>(const T& t) const{ return apply(greater_than(t));}
};
constexpr auto each_of=[](auto ... ts){ return op_t(and_elements,tuple(ts ... ));};

assert(each_of(a,b,c) > 0);
```

Things to improve:

- ✔ constexpr
- Perfect forwarding

```
constexpr auto tuple = [](auto ... ts) ...
constexpr auto and_elements = [](auto func) ...
constexpr auto equal_to = [](auto rh) ...
constexpr auto greater_than = [](auto rh)

template <typename Func, typename Tuple>
class op_t {
    Tuple tup;
    [[no_unique_address]] Func func;
    template <typename F>
    constexpr auto apply(F f) const{ retu
public:
    constexpr op_t(Func f, Tuple t) : tup
    template <typename T>
    constexpr auto operator==(const T&                    (t)); }
    template <typename T>
    constexpr auto operator>(const T& t) const{ return apply(greater_than(t));}
};
constexpr auto each_of=[](auto ... ts){ return op_t(and_elements,tuple(ts ... ));};

assert(each_of(a,b,c) > 0);
```

Things to improve:

- ✔ constexpr
- Perfect forwarding
- Conditional noexcept

```
constexpr auto tuple = [](auto ... ts) ...
constexpr auto and_elements = [](auto func) ...
constexpr auto equal_to = [](auto rh) ...
constexpr auto greater_than = [](auto rh)

template <typename Func, typename Tuple>
class op_t {
    Tuple tup;
    [[no_unique_address]] Func func;
    template <typename F>
    constexpr auto apply(F f) const{ retu
public:
    constexpr op_t(Func f, Tuple t) : tup
    template <typename T>
    constexpr auto operator==(const T&                    (t)); }
    template <typename T>
    constexpr auto operator>(const T& t) const{ return apply(greater_than(t));}
};
constexpr auto each_of=[](auto ... ts){ return op_t(and_elements,tuple(ts ... ));};

assert(each_of(a,b,c) > 0);
```

Things to improve:

- ✔ constexpr
- Perfect forwarding
- Conditional noexcept
- Explicit return type

```
constexpr auto tuple = [](auto ... ts) ...
constexpr auto and_elements = [](auto func) ...
constexpr auto equal_to = [](auto rh) ...
constexpr auto greater_than = [](auto rh)

template <typename Func, typename Tuple>
class op_t {
    Tuple tup;
    [[no_unique_address]] Func func;
    template <typename F>
    constexpr auto apply(F f) const{ retu
public:
    constexpr op_t(Func f, Tuple t) : tup
    template <typename T>
    constexpr auto operator==(const T&                    (t)); }
    template <typename T>
    constexpr auto operator>(const T& t) const{ return apply(greater_than(t));}
};
constexpr auto each_of=[](auto ... ts){ return op_t(and_elements,tuple(ts ... ));};

assert(each_of(a,b,c) > 0);
```

Things to improve:

- ✔ constexpr
- Perfect forwarding
- Conditional noexcept
- Explicit return type

```
constexpr auto and_elements = [](auto func)
{
    return [=](auto ... elements) { return (func(elements) && ... ); };
};
```

```
constexpr auto and_elements = [](auto&& func)
{
    return [func = std::forward<decltype(func)>(func)](auto ... elements) {
                return (func(elements) && ... );
            };
};
```

```
constexpr auto and_elements = [](auto&& func)
{
    return [func = std::forward<decltype(func)>(func)](auto ... elements) {
            return (func(elements) && ...);
        };
};
```

```
constexpr auto and_elements = []<typename T>(T&& func)
{
    return [func = std::forward<T>(func)](auto ... elements) {
                return (func(elements) && ... );
            };
};
```

C++20 allows us to explicitly state template parameters to lambdas.

```
constexpr auto tuple = [](auto ... ts)
{
    return [=](const auto& func) { return func(ts ... ); };
};

constexpr auto and_elements = []<typename T>(T&& func)
{
    return [func = std::forward<T>(func)](auto ... elements) {
                return (func(elements) && ... );
            };
};
```

```
constexpr auto tuple = [](auto&& ... ts)
{
    return [=](const auto& func) { return func(ts ... ); };
};

constexpr auto and_elements = []<typename T>(T&& func)
{
    return [func = std::forward<T>(func)](auto ... elements) {
                return (func(elements) && ... );
            };
};
```

Not possible to forward parameter packs in C++14 or C++17!

```
constexpr auto tuple = [](auto&& ... ts)
{
    return [ ... ts = std::forward<decltype(ts)>(ts)](const auto& func) {
            return func(ts ... );
        };
};
constexpr auto and_elements = []<typename T>(T&& func)
{
    return [func = std::forward<T>(func)](auto ... elements) {
            return (func(elements) && ... );
        };
};
```

C++20 parameter pack capture with perfect forwarding.

```
constexpr auto tuple = []<typename ... Ts>(Ts&& ... ts)
{
    return [ ...ts = std::forward<Ts>(ts)](const auto& func) {
                return func(ts...);
            };
};
constexpr auto and_elements = []<typename T>(T&& func)
{
    return [func = std::forward<T>(func)](auto ... elements) {
                return (func(elements) && ...);
            };
};
```

```
constexpr auto tuple = []<typename ... Ts>(Ts&& ... ts) ...
constexpr auto and_elements = []<typename T>(T&& func) ...

template <typename Func, typename Tuple>
class op_t {
    Tuple tup;
    [[no_unique_address]] Func func;
    template <typename F>
    constexpr auto apply(F&& f) const { return tup(func(std::forward<F>(f))); }
public:
    constexpr op_t(Func f, Tuple t) : Func(std::move(f)), tup(std::move(t)) {}
    template <typename T>
    constexpr auto operator==(const T& t) const { return apply(equal_to(t)); }
    template <typename T>
    constexpr auto operator>(const T& t) const{ return apply(greater_than(t));}
};


constexpr auto each_of=[]<typename ... Ts>(Ts&& ... ts){
     return op_t(and_elements,tuple(std::forward<Ts>(ts) ... ));
};
```

```
constexpr auto tuple = []<typename ... Ts>(Ts&& ... ts) ...
constexpr auto and_elements = []<typename T>(T&& func) ...

template <typename Func, typename Tuple>
class op_t {
    Tuple tup;
    [[no_unique_address]] Func func;
    template <typename F>
    constexpr auto apply(F&& f) const { r                    (f))); }
public:
    constexpr op_t(Func f, Tuple t) : Fun                    (t)) {}
    template <typename T>
    constexpr auto operator==(const T& t)                    (t)); }
    template <typename T>
    constexpr auto operator>(const T&                        an(t));}
};
```

Things to improve:

- ✔ constexpr
- ✔ Perfect forwarding
- Conditional noexcept
- Explicit return type

```
constexpr auto each_of=[]<typename ... Ts>(Ts&& ... ts){
     return op_t(and_elements,tuple(std::forward<Ts>(ts)... ));
};
```

```
constexpr auto tuple = []<typename ... Ts>(Ts&& ... ts) ...
constexpr auto and_elements = []<typename T>(T&& func) ...

template <typename Func, typename Tuple>
class op_t {
    Tuple tup;
    [[no_unique_address]] Func func;
    template <typename F>
    constexpr auto apply(F&& f) const { r                    (f))); }
public:
    constexpr op_t(Func f, Tuple t) : Fun                    (t)) {}
    template <typename T>
    constexpr auto operator==(const T&                       (t)); }
    template <typename T>
    constexpr auto operator>(const T&                        an(t));}
};


constexpr auto each_of=[]<typename ... Ts>(Ts&& ... ts){
     return op_t(and_elements,tuple(std::forward<Ts>(ts)... ));
};
```

Things to improve:

- ✔ constexpr
- ✔ Perfect forwarding
- Conditional noexcept
- Explicit return type

```
constexpr auto tuple = []<typename ... Ts>(Ts&& ... ts)
{
    return [ ...ts = std::forward<Ts>(ts)](const auto& func)

            {
                return func(ts ... );
            };
};
constexpr auto and_elements = []<typename T>(T&& func)
{
    return [func = std::forward<T>(func)](auto … elements)

            {
                return (func(elements) &&  ... );
            };
};
```

```
constexpr auto tuple = []<typename ... Ts>(Ts&& ... ts)
{
    return [ ...ts = std::forward<Ts>(ts)](const auto& func)
            noexcept(noexcept(func(ts ... )))
        {
            return func(ts ... );
        };
};
constexpr auto and_elements = []<typename T>(T&& func)
{
    return [func = std::forward<T>(func)](auto ... elements)
            noexcept(noexcept((func(elements) && ... )))
        {
            return (func(elements) && ... );
        };
};
```

Yes, double parenthesis

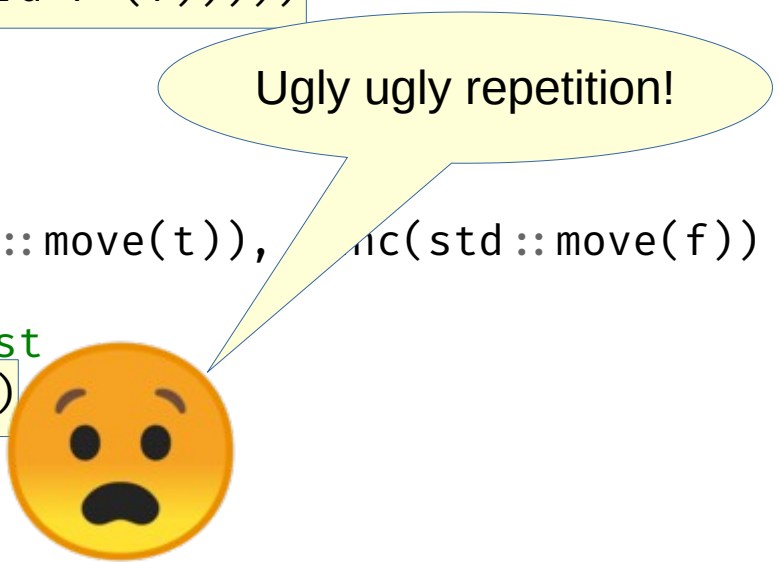
```
constexpr auto tuple = []<typename ... Ts>(Ts&& ... ts)
{
    return [ ...ts = std::forward<Ts>(ts)](const auto& func)
                noexcept(noexcept(func(ts ... )))
            {
                return func(ts ... );
            };
};
constexpr auto and_elements = []<typename T>(T&& func)
{
    return [func = std::forward<T>(func)](auto ... elements)
                noexcept(noexcept((func(elements) && ... )))
            {
                return (func(elements) && ... );
            };
};
```

There's no way around this repetition!

```
constexpr auto tuple = []<typename ... Ts>(Ts&& ... ts) ...
constexpr auto and_elements = []<typename T>(T&& func) ...

template <typename Func, typename Tuple>
class op_t {
    Tuple tup;
    [[no_unique_address]] Func func;
    template <typename F>
    constexpr auto apply(F&& f) const
        noexcept(noexcept(tup(func(std::forward<F>(f)))))
    {
        return tup(func(std::forward<F>(f)));
    }
public:
    constexpr op_t(Func f, Tuple t) : tup(std::move(t)), func(std::move(f)) {}
    template <typename T>
    constexpr auto operator==(const T& t) const
        noexcept(noexcept(apply(equal_to(t))))
    { return apply(equal_to(t)); }
    ...
};
```

```
constexpr auto tuple = []<typename ... Ts>(Ts&& ... ts) ...
constexpr auto and_elements = []<typename T>(T&& func) ...

template <typename Func, typename Tuple>
class op_t {
    Tuple tup;
    [[no_unique_address]] Func func;
    template <typename F>
    constexpr auto apply(F&& f) const
        noexcept(noexcept(tup(func(std::forward<F>(f)))))
    {
        return tup(func(std::forward<F>(f)));
    }
public:
    constexpr op_t(Func f, Tuple t) : tup(std::move(t)), func(std::move(f)) {}
    template <typename T>
    constexpr auto operator==(const T& t) const
        noexcept(noexcept(apply(equal_to(t))))
    { return apply(equal_to(t)); }
    ...
};
```

```
constexpr auto tuple = []<typename ... Ts>(Ts&& ... ts) ...
constexpr auto and_elements = []<typename T>(T&& func) ...

template <typename Func, typename Tuple>
class op_t {
    Tuple tup;
    [[no_unique_address]] Func func;
    template <typename F>
    constexpr auto apply(F&& f) const
        noexcept(noexcept(tup(func(std::forward<F>(f)))))
    {
        return tup(func(std::forward<F>(f)));
    }
public:
    constexpr op_t(Func f, Tuple t) : tup(std::move(t)), func(std::move(f)) {}
    template <typename T>
    constexpr auto operator==(const T& t) const
        noexcept(noexcept(apply(equal_to(t))))
    { return apply(equal_to(t)); }
    ...
};
```



```
constexpr auto tuple = []<typename ... Ts>(Ts&& ... ts) ...
constexpr auto and_elements = []<typename T>(T&& func) ...

template <typename Func, typename Tuple>
class op_t {
    Tuple tup;
    [[no_unique_address]] Func func;
    template <typename F>
    constexpr auto apply(F&& f) const
        noexcept(noexcept(tup(func(std::f
    {
        return tup(func(std::forward<F>(f
    }
public:
    constexpr op_t(Func f, Tuple t) :                    (f)) {}
    template <typename T>
    constexpr auto operator==(const T& t) const
        noexcept(noexcept(apply(equal_to(t))))
    { return apply(equal_to(t)); }
     ...
};
```

Things to improve:

- ✔ constexpr
- ✔ Perfect forwarding
- ✔ Conditional noexcept
- • Explicit return type

```
constexpr auto tuple = []<typename ... Ts>(Ts&& ... ts) ...
constexpr auto and_elements = []<typename T>(T&& func) ...

template <typename Func, typename Tuple>
class op_t {
    Tuple tup;
    [[no_unique_address]] Func func;
    template <typename F>
    constexpr auto apply(F&& f) const
        noexcept(noexcept(tup(func(std::f
    {
        return tup(func(std::forward<F>(f
    }
public:
    constexpr op_t(Func f, Tuple t) :                    (f)) {}
    template <typename T>
    constexpr auto operator==(const T& t) const
        noexcept(noexcept(apply(equal_to(t))))
    { return apply(equal_to(t)); }
    ...
};
```

Things to improve:

- ✔ constexpr
- ✔ Perfect forwarding
- ✔ Conditional noexcept
- Explicit return type

```
constexpr auto tuple = []<typename ... Ts>(Ts&& ... ts)
{
    return [ ...ts = std::forward<Ts>(ts)](const auto& func)
                noexcept(noexcept(func(ts…)))

            {
                return func(ts ... );
            };
};
constexpr auto and_elements = []<typename T>(T&& func)
{
    return [func = std::forward<T>(func)](auto ... elements)
                noexcept(noexcept((func(elements) && ... )))

            {
                return (func(elements) && ... );
            };
};
```

```
constexpr auto tuple = []<typename ... Ts>(Ts&& ... ts)
{
    return [ ...ts = std::forward<Ts>(ts)](const auto& func)
                noexcept(noexcept(func(ts ... )))
            → decltype(func(ts ... ))
            {
                return func(ts ... );
            };
};
constexpr auto and_elements = []<typename T>(T&& func)
{
    return [func = std::forward<T>(func)](auto ... elements)
                noexcept(noexcept((func(elements) && ... )))
            → decltype((func(elements) && ... ))
            {
                return (func(elements) && ... );
            };
};
```

Yes, double parenthesis

```
constexpr auto tuple = []<typename ... Ts>(Ts&& ... ts)
{
    return [ ...ts = std::forward<Ts>(ts)](const auto& func)
               noexcept(noexcept(func(ts ... )))
           -> decltype(func(ts ... ))
           {
               return func(ts ... );
           };
};
constexpr auto and_elements = []<typename T>(T&& func)
{
    return [func = std::forward<T>(func)](auto ... elements)
               noexcept(noexcept((func(elements) && ... )))
           -> decltype((func(elements) && ... ))
           {
               return (func(elements) && ... );
           };
};
```

https://flic.kr/p/Jp53FM

```
cons
{
};
cons
{
};
```

www.youtube.com/watch?v=I3T4lePH-yA

```
constexpr auto tuple = []<typename ... Ts>(Ts&& ... ts) ...
constexpr auto and_elements = []<typename T>(T&& func) ...

template <typename Func, typename Tuple>
class op_t {
    Tuple tup;
    [[no_unique_address]] Func func;
    template <typename F>
    constexpr auto apply(F&& f) const
        noexcept(noexcept(tup(func(std::f
    → decltype(tup(func(std::forward<F>(
    {
        return tup(func(std::forward<F>(f
    }
public:
    constexpr op_t(Func f, Tuple t) :                    (f)) {}
    template <typename T>
    constexpr auto operator==(const T& t) const
        noexcept(noexcept(apply(equal_to(t))))
    → decltype(apply(equal_to(t)))
    { return apply(equal_to(t)); }
```

Things to improve:

- ✔ constexpr
- ✔ Perfect forwarding
- ✔ Conditional noexcept
- ✔ Explicit return type

That was

That was

Modern Techniques for Keeping Your Code DRY

That was

Modern Techniques for Keeping Your Code DRY

Björn Fahller

That was

Modern Techniques for Keeping Your Code DRY

Björn Fahller

Remember

That was

Modern Techniques for Keeping Your Code DRY

Björn Fahller

Remember

- Fold expressions are awesome

Modern Techniques for Keeping Your Code DRY

Björn Fahller

Remember

- Fold expressions are awesome
- `std::tuple<>` and `std::apply()` is awsome

Modern Techniques for Keeping Your Code DRY

Björn Fahller

Remember

- Fold expressions are awesome
- `std::tuple<>` and `std::apply()` is awsome
- Higher order functions are awesome

Modern Techniques for Keeping Your Code DRY

Björn Fahller

Remember

- Fold expressions are awesome
- `std::tuple<>` and `std::apply()` is awsome
- Higher order functions are awesome
- Lambdas are awesome

Modern Techniques for Keeping Your Code DRY

Björn Fahller

Remember

- Fold expressions are awesome
- `std::tuple<>` and `std::apply()` is awsome
- Higher order functions are awesome
- Lambdas are awesome
  - C++20 lambdas are awsomer!

Modern Techniques for Keeping Your Code DRY

Björn Fahller

Remember

- Fold expressions are awesome
- `std::tuple<>` and `std::apply()` is awsome
- Higher order functions are awesome
- Lambdas are awesome
  - C++20 lambdas are awsomer!
- Compilers are awesome

Modern Techniques for Keeping Your Code DRY

Björn Fahller

Remember

- Fold expressions are awesome
- `std::tuple<>` and `std::apply()` is awsome
- Higher order functions are awesome
- Lambdas are awesome
  - C++20 lambdas are awsomer!
- Compilers are awesome

Modern Techniques for Keeping Your Code DRY

Björn Fahller

Remember

- Fold expressions are awesome
- `std::tuple<>` and `std::apply()` is awsome
- Higher order functions are awesome
- Lambdas are awesome
  - C++20 lambdas are awsomer!
- Compilers are awesome
- noexcept/SFINAE-return is aweso^H^Hful

Modern Techniques for Keeping Your Code DRY

Björn Fahller

Remember

- Fold expressions are awesome
- `std::tuple<>` and `std::apply()` is awsome
- Higher order functions are awesome
- Lambdas are awesome
  - C++20 lambdas are awsomer!
- Compilers are awesome
- noexcept/SFINAE-return is aweso^H^Hful
- Sweating the small stuff makes you annoyed

Modern Techniques for Keeping Your Code DRY

Björn Fahller

Remember

- Fold expressions are awesome
- `std::tuple<>` and `std::apply()` is awsome
- Higher order functions are awesome
- Lambdas are awesome
  - C++20 lambdas are awsomer!
- Compilers are awesome
- noexcept/SFINAE-return is aweso^H^Hful
- Sweating the small stuff makes you annoyed
  - But it can lead to neat code

Modern Techniques for Keeping Your Code DRY

Björn Fahller

Remember

- Fold expressions are awesome
- `std::tuple<>` and `std::apply()` is awsome
- Higher order functions are awesome
- Lambdas are awesome
  - C++20 lambdas are awsomer!
- Compilers are awesome
- noexcept/SFINAE-return is aweso^H^Hful
- Sweating the small stuff makes you annoyed
  - But it can lead to neat code

Modern Techniques for Keeping Your Code DRY

Björn Fahller

bjorn@fahller.se 

@bjorn_fahller 

@rollbear    

#include <C++>