

# Implementing “static” control flow in C++14

 <http://vittorioromeo.info>  
vittorio.romeo@outlook.com

**Meeting C++**

**Bloomberg**  
vromeo5@bloomberg.net

# Implementing “static” control flow in C++14

<http://github.com/SuperV1234/meetingcpp2016>

 <http://vittorioromeo.info>  
vittorio.romeo@outlook.com

**Meeting C++**

**Bloomberg**  
vromeo5@bloomberg.net

# Talk overview

- What is “**static**” control flow?
- Compile-time branching.
  - History of `static if` in C++.
  - C++17: `if constexpr`.
  - C++14: `static_if`.
- `static_if` implementation details.
- Compile-time iteration.
  - `for_each_argument`.
  - `static_for`.

# Talk overview

- What is “**static**” control flow?
- Compile-time branching.
  - History of `static if` in C++.
  - C++17: `if constexpr`.
  - C++14: `static_if`.
- `static_if` implementation details.
- Compile-time iteration.
  - `for_each_argument`.
  - `static_for`.

# Talk overview

- What is “**static**” control flow?
- Compile-time branching.
  - History of `static if` in C++.
  - C++17: `if constexpr`.
  - C++14: `static_if`.
- `static_if` implementation details.
- Compile-time iteration.
  - `for_each_argument`.
  - `static_for`.

# What is “static” control flow?

- `static` is a specifier with multiple meanings in C++.
- It's also a word commonly used by developers to refer to **compile-time control flow**.
- Existing languages, such as D, have powerful compile-time constructs like `static if`.
- Goals of this talk:
  - Understand the benefits of static control flow.
  - Look at the history of `static_if` proposals in C++, analyze `if constexpr`.
  - Implement two C++14 constructs: `static_if` and `static_for`.

*Example:* static if in D

## *Example: static if in D*

```
template INT(int i)
{
    static if (i == 32)
        alias INT = int;
    else static if (i == 16)
        alias INT = short;
    else
        static assert(0);
}
```



## *Example: static if in D*

```
template INT(int i)
{
    static if (i == 32)
        alias INT = int;
    else static if (i == 16)
        alias INT = short;
    else
        static assert(0);
}
```

```
template<int i>
struct INT;

template<>
struct INT<32>
{
    using type = int;
};

template<>
struct INT<16>
{
    using type = short;
};
```

## Example: handling variadic argument packs (traditional)

```
template <class T>
void f(T&& t)
{
    // handle one T
}
```

```
template <class T, class... Rest>
void f(T&& t, Rest&&... r)
{
    f(t);
    // handle the tail
    f(r...);
}
```

## *Example: handling variadic argument packs (static if)*

```
template <class T, class... Rest>
void f(T&& t, Rest&&... r)
{
    f(t);

    if constexpr(sizeof...(r))
    {
        // handle the tail
        f(r...);
    }
}
```

## Example: {} vs () object construction (*traditional*)

```
template <class T, class... Args>
enable_if_t<is_constructible_v<T, Args...>, unique_ptr<T>>
make_unique(Args&&... args)
{
    return unique_ptr<T>(new T(forward<Args>(args)...));
}
```

```
template <class T, class... Args>
enable_if_t<!is_constructible_v<T, Args...>, unique_ptr<T>>
make_unique(Args&&... args)
{
    return unique_ptr<T>(new T{forward<Args>(args)...});
}
```

## Example: {} vs () object construction (*static if*)

```
template <class T, class... Args>
auto make_unique(Args&&... args)
{
    if constexpr(is_constructible_v<T, Args...>)
    {
        return unique_ptr<T>(new T(forward<Args>(args)...));
    }
    else
    {
        return unique_ptr<T>(new T{forward<Args>(args)...});
    }
}
```

# History of static if proposals - 1

- The previous examples were taken from proposal [P0128R0](#):
  - "constexpr if" - *Ville Voutilainen*
- This paper was originally created as a "resurrection" of the very controversial previous "static if" [N3322](#) and [N3329](#) proposals:
  - "A Preliminary Proposal for a Static if" - *Walter E. Brown*
  - "static if declaration" - *W. Bright, H. Sutter, A. Alexandrescu*
- The two above proposals were considered harmful in [N3613](#), due to their unintuitive scope rules and inconsistency with the rest of the language:
  - "Static if considered" - *B. Stroustrup, G. Dos Reis, A. Sutton*

# History of static if proposals - 1

- The previous examples were taken from proposal [P0128R0](#):
  - "constexpr if" - *Ville Voutilainen*
- This paper was originally created as a "resurrection" of the very controversial previous "static if" [N3322](#) and [N3329](#) proposals:
  - "A Preliminary Proposal for a Static if" - *Walter E. Brown*
  - "static if declaration" - *W. Bright, H. Sutter, A. Alexandrescu*
- The two above proposals were considered harmful in [N3613](#), due to their unintuitive scope rules and inconsistency with the rest of the language:
  - "Static if considered" - *B. Stroustrup, G. Dos Reis, A. Sutton*

# History of static if proposals - 1

- The previous examples were taken from proposal [P0128R0](#):
  - "constexpr if" - *Ville Voutilainen*
- This paper was originally created as a "resurrection" of the very controversial previous "static if" [N3322](#) and [N3329](#) proposals:
  - "A Preliminary Proposal for a Static if" - *Walter E. Brown*
  - "static if declaration" - *W. Bright, H. Sutter, A. Alexandrescu*
- The two above proposals were considered harmful in [N3613](#), due to their unintuitive scope rules and inconsistency with the rest of the language:
  - "Static if considered" - *B. Stroustrup, G. Dos Reis, A. Sutton*



# History of static if proposals - 1

- The previous examples were taken from proposal [P0128R0](#):
  - "constexpr if" - *Ville Voutilainen*
- This paper was originally created as a "resurrection" of the very controversial previous "static if" [N3322](#) and [N3329](#) proposals:
  - "A Preliminary Proposal for a Static if" - *Walter E. Brown*
  - "static if declaration" - *W. Bright, H. Sutter, A. Alexandrescu*
- The two above proposals were considered harmful in [N3613](#), due to their unintuitive scope rules and inconsistency with the rest of the language:
  - "Static if considered" - *B. Stroustrup, G. Dos Reis, A. Sutton*

# History of static if proposals - 1

- The previous examples were taken from proposal [P0128R0](#):
  - "constexpr if" - *Ville Voutilainen*
- This paper was originally created as a "resurrection" of the very controversial previous "static if" [N3322](#) and [N3329](#) proposals:
  - "A Preliminary Proposal for a Static if" - *Walter E. Brown*
  - "static if declaration" - *W. Bright, H. Sutter, A. Alexandrescu*
- The two above proposals were considered harmful in [N3613](#), due to their unintuitive scope rules and inconsistency with the rest of the language:
  - "Static if considered" - *B. Stroustrup, G. Dos Reis, A. Sutton*

# History of `static if` proposals - 2

- Starting with [N4461](#) by *V. Voutilainen*, the idea of a compile-time `if` construct with familiar scope rules began to gain traction.
  - “Static if resurrected” – *Ville Voutilainen*
- Eventually, the proposed syntax and standard wording was revised multiple times in [P0128R0](#), [P0128R1](#), [P0292R0](#), and [P0292R1](#).
  - “constexpr\_if” – *Ville Voutilainen*
  - “constexpr\_if” – *Ville Voutilainen, Daveed Vandevoorde*
  - “constexpr if: A slightly different syntax” – *Jens Maurer*
  - “constexpr if: A slightly different syntax” – *Jens Maurer*

# History of `static if` proposals - 2

- Starting with [N4461](#) by *V. Voutilainen*, the idea of a compile-time `if` construct with familiar scope rules began to gain traction.
  - “Static if resurrected” – *Ville Voutilainen*
- Eventually, the proposed syntax and standard wording was revised multiple times in [P0128R0](#), [P0128R1](#), [P0292R0](#), and [P0292R1](#).
  - “constexpr\_if” – *Ville Voutilainen*
  - “constexpr\_if” – *Ville Voutilainen, Daveed Vandevoorde*
  - “constexpr if: A slightly different syntax” – *Jens Maurer*
  - “constexpr if: A slightly different syntax” – *Jens Maurer*

# History of `static if` proposals - 2

- Starting with [N4461](#) by *V. Voutilainen*, the idea of a compile-time `if` construct with familiar scope rules began to gain traction.
  - “Static if resurrected” – *Ville Voutilainen*
- Eventually, the proposed syntax and standard wording was revised multiple times in [P0128R0](#), [P0128R1](#), [P0292R0](#), and [P0292R1](#).
  - “constexpr\_if” – *Ville Voutilainen*
  - “constexpr\_if” – *Ville Voutilainen, Daveed Vandevoorde*
  - “constexpr if: A slightly different syntax” – *Jens Maurer*
  - “constexpr if: A slightly different syntax” – *Jens Maurer*

# History of `static if` proposals - 3

- The final revision, [P0292R2](#), was accepted for C++17:
  - “`constexpr if`: A slightly different syntax”  
- *Jens Maurer*



## if constexpr(...) – *valid C++17 example - 1*

```
template <class T, class... Rest>
void f(T&& t, Rest&&... r)
{
    f(t);

    if constexpr(sizeof...(r))
    {
        // handle the tail
        f(r...);
    }
}
```

## if constexpr(...) – *valid C++17 example* - 2

```
template <class T, class... Args>
auto make_unique(Args&&... args)
{
    if constexpr(is_constructible_v<T, Args...>)
    {
        return unique_ptr<T>(new T(forward<Args>(args)...));
    }
    else
    {
        return unique_ptr<T>(new T{forward<Args>(args)...});
    }
}
```



## `if constexpr(...)` – rules

- Restricted to block scopes.
- Always going to establish a new scope.
- Required that there exists values of the condition so either condition branch is well-formed.

# if constexpr(...) – rules

- Restricted to block scope
- Always going to establish
- Required that there exists at least one condition branch that is well-formed.

```
template INT(int i)
{
    static if (i == 32)
        alias INT = int;
    else static if (i == 16)
        alias INT = short;
    else
        static assert(0);
}
```

# if constexpr(...) – rules

- Restricted to block scope
- Always going to establish
- Required that there exists at least one condition branch that is well-formed.

```
template INT(int i)
{
    static if (i == 32)
        alias INT = int;
    else static if (i == 16)
        alias INT = short;
    else
        static assert(0);
}
```

## `if constexpr(...)` – rules

- Restricted to block scopes.
- Always going to establish a new scope.
- Required that there exists values of the condition so either condition branch is well-formed.

# if constexpr(...) – branch chaining

```
if constexpr (cond0)
    statement0;
else if constexpr (cond1)
    statement1;
else if constexpr (cond2)
    statement2;
else
    statement3;
```

*“Do I have to wait until C++17 is  
supported by my  
company/architecture?”*

# C++14 static\_if – *example (1)*

- Example situation:
  - Multiple food-related classes with slightly different interfaces.
- Goal:
  - Create a generic consume(x) function that will accept any kind of food instance and will print something to stdout.

```
struct banana
{
    void eat() { }
};

struct peanuts
{
    void eat() { }
};

struct water
{
    void drink() { }
};

struct juice
{
    void drink() { }
};
```

# C++14 static\_if – *example (1)*

- Example situation:
  - Multiple food-related classes with slightly different interfaces.
- Goal:
  - Create a generic consume(x) function that will accept any kind of food instance and will print something to stdout.

```
struct banana
{
    void eat() { }
};

struct peanuts
{
    void eat() { }
};

struct water
{
    void drink() { }
};

struct juice
{
    void drink() { }
};
```



# C++14 static\_if – *example (1)*

- Example situation:
  - Multiple food-related classes with slightly different interfaces.
- Goal:
  - Create a generic consume(x) function that will accept any kind of food instance and will print something to stdout.

```
struct banana
{
    void eat() { }
```

```
struct peanuts
{
    void eat() { }
```

```
struct water
{
    void drink() { }
```

```
struct juice
{
    void drink() { }
```

## C++14 static\_if – *example (2)*

- Both if constexpr and my static\_if implementation require a *constant expression* as their branching condition.
- Let's define some constexpr bool variable templates to categorize the foods depending on their interface.

```
template <typename T>  
constexpr bool is_solid{false};
```

```
template <>  
constexpr bool is_solid<banana>{true};
```

```
template <>  
constexpr bool is_solid<peanuts>{true};
```

```
template <typename T>  
constexpr bool is_liquid{false};
```

```
template <>  
constexpr bool is_liquid<water>{true};
```

```
template <>  
constexpr bool is_liquid<juice>{true};
```

```
template <typename T>
auto consume(T&& x)
{
    static_if(bool_v<is_solid<T>>)
        .then([](auto&& y)
            {
                y.eat();
                std::cout << "eating solid\n";
            })
        .else_if(bool_v<is_liquid<T>>)
        .then([](auto&& y)
            {
                y.drink();
                std::cout << "drinking liquid\n";
            })
        .else_([](auto&&)
            {
                std::cout << "cannot consume\n";
            })(FWD(x)));
}
```

```
template <typename T>
auto consume(T&& x)
{
    static_if(bool_v<is_solid<T>>)
        .then([](auto&& y)
            {
                y.eat();
                std::cout << "eating solid\n";
            })
        .else_if(bool_v<is_liquid<T>>)
        .then([](auto&& y)
            {
                y.drink();
                std::cout << "drinking liquid\n";
            })
        .else_([](auto&&)
            {
                std::cout << "cannot consume\n";
            })(FWD(x)));
}
```

The implementation requires the condition to be wrapped inside a compile-time boolean variable wrapper: that's what `bool_v` is for.

```
template <typename T>
auto consume(T&& x)
{
    static_if(bool_v<is_solid<T>>)
        .then([](auto&& y)
            {
                y.eat();
                std::cout << "eating solid\n";
            })
        .else_if(bool_v<is_liquid<T>>)
        .then([](auto&& y)
            {
                y.drink();
                std::cout << "drinking liquid\n";
            })
        .else_([](auto&&)
            {
                std::cout << "cannot consume\n";
            })(FWD(x)));
}
```

```

template <typename T>
auto consume(T&& x)
{
    static_if(bool_v<is_solid<T>>)
        .then([](auto&& y)

```

```

template <bool TX>
using bool_ = std::integral_constant<bool, TX>;

template <bool TX>
constexpr bool_<TX> bool_v{};

```

```

        .else_([](auto&&)
        {
            std::cout << "cannot consume\n";
        })(FWD(x));
    }
}

```

```
template <typename T>
auto consume(T&& x)
{
    static_if(bool_v<is_solid<T>>)
        .then([](auto&& y)
            {
                y.eat();
                std::cout << "eating solid\n";
            })
        .else_if(bool_v<is_liquid<T>>)
        .then([](auto&& y)
            {
                y.drink();
                std::cout << "drinking liquid\n";
            })
        .else_([](auto&&)
            {
                std::cout << "cannot consume\n";
            })(FWD(x)));
}
```

```
template <typename T>  
auto consume(T&& x)  
{
```

Wrapping types inside values (*and vice versa*) is what allows amazing libraries such as `boost::hana` (by *Louis Dionne*) or `fit` and `tick` (by *Paul Fultz II*) to provide extremely powerful, clean, and intuitive metaprogramming facilities.

```
    y.drink();  
    std::cout << "drinking liquid\n";  
    })  
    .else_([](auto&&  
    {  
        std::cout << "cannot consume\n";  
    }))(FWD(x));  
}
```



```
template <typename T>
auto consume(T&& x)
{
```

Wrapping types inside values (*and vice versa*) is what allows amazing libraries such as `boost::hana` (*by Louis Dionne*) or `fit` and `tick` (*by Paul Fultz II*) to provide extremely powerful, clean, and intuitive metaprogramming facilities.

More info regarding "type-value encoding"/"dependent typing":

<http://pfultz2.com/blog/2015/01/24/dependent-typing/>

<http://boostorg.github.io/hana/index.html#tutorial-type>

```
})(FWD(x));
```

```
}
```

```
template <typename T>
auto consume(T&& x)
{
    static_if(bool_v<is_solid<T>>)
        .then([](auto&& y)
            {
                y.eat();
                std::cout << "eating solid\n";
            })
        .else_if(bool_v<is_liquid<T>>)
        .then([](auto&& y)
            {
                y.drink();
                std::cout << "drinking liquid\n";
            })
        .else_([](auto&&)
            {
                std::cout << "cannot consume\n";
            })(FWD(x)));
}
```

```
template <typename T>
auto consume(T&& x)
{
    static_if(bool_v<is_solid<T>>)
        .then([](auto&& y)
            {
                y.eat();
                std::cout << "eating solid\n";
            })
        .else_if(bool_v<is_liquid<T>>)
        .then([](auto&& y)
            {
                y.drink();
                std::cout << "drinking liquid\n";
            })
        .else_([](auto&&)
            {
                std::cout << "cannot consume\n";
            })(FWD(x)));
}
```

```
template <typename T>
auto consume(T&& x)
{
    static_if(bool_v<is_solid<T>>)
        .then([](auto&& y)
            {
                y.eat();
                std::cout << "eating solid\n";
            })
        .else_if(bool_v<is_liquid<T>>)
        .then([](auto&& y)
            {
                y.drink();
                std::cout << "drinking liquid\n";
            })
        .else_([](auto&&)
            {
                std::cout << "cannot consume\n";
            })(FWD(x)));
}
```

Scope rules are what you would expect.

```
template <typename T>
auto consume(T&& x)
{
    static_if(bool_v<is_solid<T>>)
        .then([](auto&& y)
        {
            y.eat();
            std::cout << "eating solid\n";
        })
        .else_if(bool_v<is_liquid<T>>)
        .then([](auto&& y)
        {
            y.drink();
            std::cout << "drinking liquid\n";
        })
        .else_([](auto&&)
        {
            std::cout << "cannot consume\n";
        })(FWD(x)));
}
```

```
template <typename T>
auto consume(T&& x)
{
    static_if(bool_v<is_solid<T>>)
        .then([](auto&& y)
            {
                y.eat();
                std::cout << "eating solid\n";
            })
        .else_if(bool_v<is_liquid<T>>)
        .then([](auto&& y)
            {
                y.drink();
                std::cout << "drinking liquid\n";
            })
        .else_([](auto&&)
            {
                std::cout << "cannot consume\n";
            })(FWD(x)));
}
```

```

template <typename T>
auto consume(T&& x)
{
    static_if(bool_v<is_solid<T>>)
        .then([](auto&& y)
            {
                y.eat();
                std::cout << "eating solid\n";
            })
        .else_if(bool_v<is_liquid<T>>)
        .then([](auto&& y)
            {
                y.drink();
                std::cout << "drinking liquid\n";
            })
        .else_([](auto&&)
            {
                std::cout << "cannot consume\n";
            })(FWD(x)));
}

```

Think of every branch of the `static_if` as a template function that **will only be instantiated if the predicate matches**.

In this example, even if `y.eat()` does not exist, **we won't get a compilation error**, because the branch won't be instantiated.

```
template <typename T>
auto consume(T&& x)
{
    static_if(bool_v<is_solid<T>>)
        .then([](auto&& y)
            {
                y.eat();
                std::cout << "eating solid\n";
            })
        .else_if(bool_v<is_liquid<T>>)
        .then([](auto&& y)
            {
                y.drink();
                std::cout << "drinking liquid\n";
            })
        .else_([](auto&&)
            {
                std::cout << "cannot consume\n";
            })(FWD(x));
}
```



```
template <typename T>
auto consume(T&& x)
{
    static_if(bool_v<is_solid<T>>)
        .then([](auto&& y)
            {
                y.eat();
                std::cout << "eating solid\n";
            })
        .else_if(bool_v<is_liquid<T>>)
        .then([](auto&& y)
            {
                y.drink();
                std::cout << "drinking liquid\n";
            })
        .else_([](auto&&)
            {
                std::cout << "cannot consume\n";
            })(FWD(x)));
}
```



This pattern works thanks to C++14's generic lambdas.

This pattern works thanks to C++14's generic lambdas.

```
[](auto&& y)
{
    y.eat();
    std::cout << "eating solid\n";
}
```

This pattern works thanks to C++14's generic lambdas.

```
[](auto&& y)
{
    y.eat();
    std::cout << "eating solid\n";
}
```

```
struct lambda
{
    template <typename T>
    auto operator()(T&& y) const
    {
        y.eat();
        std::cout << "eating solid\n";
    }
};
```

This pattern works thanks to C++14's generic lambdas.

```
[](auto&& y)
{
    y.eat();
    std::cout << "eating solid\n";
}
```

```
struct lambda
{
    template <typename T>
    auto operator()(T&& y) const
    {
        y.eat();
        std::cout << "eating solid\n";
    }
};
```

# C++14 static\_if – instantiating matching branch

- What allows static\_if branches to only be instantiated when the condition is true?
- *"Passing the argument back to static\_if"* with the final call does that: thanks to this trick, the instantiation of the branches is *"delayed"* so that potentially invalid branches do not cause a compilation error unless chosen.

```
template <typename T>
auto consume(T&& x)
{
    static_if(bool_v<is_solid<T>>)
        .then([](auto&& y)
            {
                y.eat();
                std::cout << "eating solid\n";
            })
        .else_if(bool_v<is_liquid<T>>)
        .then([](auto&& y)
            {
                y.drink();
                std::cout << "drinking liquid\n";
            })
        .else_([](auto&&)
            {
                std::cout << "cannot consume\n";
            })(FWD(x));
}
```

# C++14 static\_if – instantiating matching branch

- What allows static\_if branches to only be instantiated when the condition is true?
- *"Passing the argument back to static\_if"* with the final call does that: thanks to this trick, the instantiation of the branches is *"delayed"* so that potentially invalid branches do not cause a compilation error unless chosen.

```
template <typename T>
auto consume(T&& x)
{
    static_if(bool_v<is_solid<T>>)
        .then([](auto&& y)
            {
                y.eat();
                std::cout << "eating solid\n";
            })
        .else_if(bool_v<is_liquid<T>>)
        .then([](auto&& y)
            {
                y.drink();
                std::cout << "drinking liquid\n";
            })
        .else_([](auto&&)
            {
                std::cout << "cannot consume\n";
            })(FWD(x));
}
```



*Let's analyze the technique,  
step-by-step.*

## Step 0: call consume

consume(juice{});

```
template <typename T>
auto consume(T&& x)
{
    static_if(bool_v<is_solid<T>>)
        .then([](auto&& y)
            {
                y.eat();
                std::cout << "eating solid\n";
            })
        .else_if(bool_v<is_liquid<T>>)
        .then([](auto&& y)
            {
                y.drink();
                std::cout << "drinking liquid\n";
            })
        .else_([](auto&&)
            {
                std::cout << "cannot consume\n";
            })(FWD(x));
}
```

## Step 0: call consume

consume(juice{});

```
template <typename T>
auto consume(T&& x)
{
    static_if(bool_v<is_solid<T>>)
        .then([](auto&& y)
            {
                y.eat();
                std::cout << "eating solid\n";
            })
        .else_if(bool_v<is_liquid<T>>)
        .then([](auto&& y)
            {
                y.drink();
                std::cout << "drinking liquid\n";
            })
        .else_([](auto&&)
            {
                std::cout << "cannot consume\n";
            })(FWD(x));
}
```

## Step 0: call consume

consume(juice{});

```
template <typename T>
auto consume(T&& x)
{
    static_if(bool_v<is_solid<T>>)
        .then([](auto&& y)
            {
                y.eat();
                std::cout << "eating solid\n";
            })
        .else_if(bool_v<is_liquid<T>>)
        .then([](auto&& y)
            {
                y.drink();
                std::cout << "drinking liquid\n";
            })
        .else_([](auto&&)
            {
                std::cout << "cannot consume\n";
            })(FWD(x));
}
```

## Step 0: call consume

consume(juice{});

```
template <typename T>
auto consume(T&& x)
{
    static_if(bool_v<is_solid<T>>)
        .then([](auto&& y)
            {
                y.eat();
                std::cout << "eating solid\n";
            })
        .else_if(bool_v<is_liquid<T>>)
        .then([](auto&& y)
            {
                y.drink();
                std::cout << "drinking liquid\n";
            })
        .else_([](auto&&)
            {
                std::cout << "cannot consume\n";
            })(FWD(x));
}
```

## Step 0: call consume

```
template <typename T>
auto consume(T&& x)
{
    static_if(bool_v<is_solid<T>>)
        .then([](auto&& y)
            {
                y.eat();
                std::cout << "eating solid\n";
            })
        .else_if(bool_v<is_liquid<T>>)
        .then([](auto&& y)
            {
                y.drink();
                std::cout << "drinking liquid\n";
            })
        .else_([](auto&&)
            {
                std::cout << "cannot consume\n";
            })(FWD(x));
}
```

## Step 0: call consume

```
template <typename T>
auto consume(T&& x)
{
    static_if(bool_v<is_solid<T>>)
        .then([](auto&& y)
            {
                y.eat();
                std::cout << "eating solid\n";
            })
        .else_if(bool_v<is_liquid<T>>)
        .then([](auto&& y)
            {
                y.drink();
                std::cout << "drinking liquid\n";
            })
        .else_([](auto&&)
            {
                std::cout << "cannot consume\n";
            })(FWD(x));
}
```

## Step 1: find matching branch

```
template <typename T>
auto consume(T&& x)
{
    static_if(bool_v<is_solid<T>>)
        .then([](auto&& y)
            {
                y.eat();
                std::cout << "eating solid\n";
            })
        .else_if(bool_v<is_liquid<T>>)
        .then([](auto&& y)
            {
                y.drink();
                std::cout << "drinking liquid\n";
            })
        .else_([](auto&&)
            {
                std::cout << "cannot consume\n";
            })(FWD(x));
}
```



## Step 1: find matching branch

```
template <typename T>
auto consume(T&& x)
{
    static_if(bool_v<is_solid<T>>)
        .then([](auto&& y)
            {
                y.eat();
                std::cout << "eating solid\n";
            })
        .else_if(bool_v<is_liquid<T>>)
        .then([](auto&& y)
            {
                y.drink();
                std::cout << "drinking liquid\n";
            })
        .else_([](auto&&)
            {
                std::cout << "cannot consume\n";
            })(FWD(x));
}
```

## Step 1: find matching branch

```
template <typename T>
auto consume(T&& x)
{
    static_if(bool_v<is_solid<T>>)
        .then([](auto&& y)
            {
                y.eat();
                std::cout << "eating solid\n";
            })
        .else_if(bool_v<is_liquid<T>>)
        .then([](auto&& y)
            {
                y.drink();
                std::cout << "drinking liquid\n";
            })
        .else_([](auto&&)
            {
                std::cout << "cannot consume\n";
            })(FWD(x));
}
```

## Step 1: find matching branch

```
template <typename T>
auto consume(T&& x)
{
    static_if(bool_v<is_solid<T>>)
        .then([](auto&& y)
            {
                y.eat();
                std::cout << "eating solid\n";
            })
        .else_if(bool_v<is_liquid<T>>)
        .then([](auto&& y)
            {
                y.drink();
                std::cout << "drinking liquid\n";
            })
        .else_([](auto&&)
            {
                std::cout << "cannot consume\n";
            })(FWD(x));
}
```

## Step 1: find matching branch

```
template <typename T>
auto consume(T&& x)
{
    static_if(bool_v<is_solid<T>>)
        .then([](auto&& y)
            {
                y.eat();
                std::cout << "eating solid\n";
            })
        .else_if(bool_v<is_liquid<T>>)
        .then([](auto&& y)
            {
                y.drink();
                std::cout << "drinking liquid\n";
            })
        .else_([](auto&&)
            {
                std::cout << "cannot consume\n";
            })(FWD(x));
}
```

## Step 1: find matching branch

```
template <typename T>
auto consume(T&& x)
{
    static_if(bool_v<is_solid<T>>)
        .then([](auto&& y)
            {
                y.eat();
                std::cout << "eating solid\n";
            })
        .else_if(bool_v<is_liquid<T>>)
        .then([](auto&& y)
            {
                y.drink();
                std::cout << "drinking liquid\n";
            })
        .else_([](auto&&)
            {
                std::cout << "cannot consume\n";
            })(FWD(x));
}
```

## Step 1: find matching branch

```
template <typename T>
auto consume(T&& x)
{
    static_if(bool_v<is_solid<T>>)
        .then([](auto&& y)
            {
                y.eat();
                std::cout << "eating solid\n";
            })
        .else_if(bool_v<is_liquid<T>>)
        .then([](auto&& y)
            {
                y.drink();
                std::cout << "drinking liquid\n";
            })
        .else_([](auto&&)
            {
                std::cout << "cannot consume\n";
            })(FWD(x));
}
```

## Step 1: find matching branch

```
template <typename T>
auto consume(T&& x)
{
    static_if(bool_v<is_solid<T>>)
        .then([](auto&& y)
            {
                y.eat();
                std::cout << "eating solid\n";
            })
        .else_if(bool_v<is_liquid<T>>)
        .then([](auto&& y)
            {
                y.drink();
                std::cout << "drinking liquid\n";
            })
        .else_([](auto&&)
            {
                std::cout << "cannot consume\n";
            })(FWD(x));
}
```

## Step 2: instantiate and call matching branch

```
template <typename T>
auto consume(T&& x)
{
    static_if(bool_v<is_solid<T>>)
        .then([](auto&& y)
            {
                y.eat();
                std::cout << "eating solid\n";
            })
        .else_if(bool_v<is_liquid<T>>)
        .then([](auto&& y)
            {
                y.drink();
                std::cout << "drinking liquid\n";
            })
        .else_([](auto&&)
            {
                std::cout << "cannot consume\n";
            })(FWD(x));
}
```



## Step 2: instantiate and call matching branch

```
template <typename T>
auto consume(T&& x)
{
    ... static_if(bool_v<is_solid<T>>)
    ...     .then([](auto&& y)
    ...         {
    ...             y.eat();
    ...             std::cout << "eating solid\n";
    ...         })
    ...     .else_if(bool_v<is_liquid<T>>)
    ...     .then([](auto&& y)
    ...         {
    ...             y.drink();
    ...             std::cout << "drinking liquid\n";
    ...         })
    ...     .else_([](auto&&)
    ...         {
    ...             std::cout << "cannot consume\n";
    ...         }) (FWD(x));
}
```

## Step 2: instantiate and call matching branch

```
template <typename T>
auto consume(T&& x)
{
    ... static_if(bool_v<is_solid<T>>)
    ...     .then([](auto&& y)
    ...         {
    ...             y.eat();
    ...             std::cout << "eating solid\n";
    ...         })
    ...     .else_if(bool_v<is_liquid<T>>)
    ...     .then([](auto&& y)
    ...         {
    ...             y.drink();
    ...             std::cout << "drinking liquid\n";
    ...         })
    ...     .else_([](auto&&)
    ...         {
    ...             std::cout << "cannot consume\n";
    ...         }) (FWD(x));
}
```

Imagine that the entire `static_if` “collapses” to the first matching branch’s body. The body can then be called with the argument `FWD(x)`.

## Step 2: instantiate and call matching branch

```
template <typename T>
auto consume(T&& x)
{
    ... static_if(bool_v<is_solid<T>>)
    ...     .then([](auto&& y)
    ...         {
    ...             y.eat();
    ...             std::cout << "eating solid\n";
    ...         })
    ...     .else_if(bool_v<is_liquid<T>>)
    ...     .then([](auto&& y)
    ...         {
    ...             y.drink();
    ...             std::cout << "drinking liquid\n";
    ...         })
    ...     .else_([](auto&&)
    ...         {
    ...             std::cout << "cannot consume\n";
    ...         }) (FWD(x));
}
```

Imagine that the entire `static_if` “collapses” to the first matching branch’s body. The body can then be called with the argument `FWD(x)`.

## Step 2: instantiate and call matching branch

```
template <typename T>
auto consume(T&& x)
{
    ... static_if(bool_v<is_solid<T>>)
    ...     .then([](auto&& y)
    ...         {
    ...             y.eat();
    ...             std::cout << "eating solid\n";
    ...         })
    ...     .else_if(bool_v<is_liquid<T>>)
    ...     .then([](auto&& y)
    ...         {
    ...             y.drink();
    ...             std::cout << "drinking liquid\n";
    ...         })
    ...     .else_([](auto&&)
    ...         {
    ...             std::cout << "cannot consume\n";
    ...         }) (FWD(x));
}
```

Imagine that the entire `static_if` “collapses” to the first matching branch’s body. The body can then be called with the argument `FWD(x)`.

## Step 2: instantiate and call matching branch

```
template <typename T>
auto consume(T&& x)
{
    ... static_if(bool_v<is_solid<T>>)
    ...     .then([](auto&& y)
    ...         {
    ...             y.eat();
    ...             std::cout << "eating solid\n";
    ...         })
    ...     .else_if(bool_v<is_liquid<T>>)
    ...     .then([](auto&& y)
    ...         {
    ...             y.drink();
    ...             std::cout << "drinking liquid\n";
    ...         })
    ...     .else_([](auto&&)
    ...         {
    ...             std::cout << "cannot consume\n";
    ...         }) (FWD(x));
}
```

Imagine that the entire `static_if` “collapses” to the first matching branch’s body. The body can then be called with the argument `FWD(x)`.

## Step 2: instantiate and call matching branch

```
template <typename T>
auto consume(T&& x)
{
    ... static_if(bool_v<is_solid<T>>)
    ...     .then([](auto&& y)
    ...         {
    ...             y.eat();
    ...             std::cout << "eating solid\n";
    ...         })
    ...     .else_if(bool_v<is_liquid<T>>)
    ...     .then([](auto&& y)
    ...         {
    ...             y.drink();
    ...             std::cout << "drinking liquid\n";
    ...         })
    ...     .else_([](auto&&)
    ...         {
    ...             std::cout << "cannot consume\n";
    ...         }) (FWD(x));
}
```

Imagine that the entire `static_if` “collapses” to the first matching branch’s body. The body can then be called with the argument `FWD(x)`.

## Step 2: instantiate and call matching branch

```
[ ](auto&& y)
{
    y.drink();
    std::cout << "drinking liquid\n";
}(FWD(x));
```

Imagine that the entire `static_if` “collapses” to the first matching branch’s body. The body can then be called with the argument `FWD(x)`.

# C++14 static\_if – capture?

- Would capturing the variable work?

consume(juice{});

```
static_if(bool_v<is_solid<T>>)
    .then([&x]
        {
            x.eat();
            std::cout << "eating solid\n";
        })
    .else_if(bool_v<is_liquid<T>>)
    .then([&x]
        {
            x.drink();
            std::cout << "drinking liquid\n";
        })
    .else_([
        {
            std::cout << "cannot consume\n";
        })());
```



# C++14 static\_if – capture?

- Would capturing the variable work?

consume(juice{});

Nope.

```
static_if(bool_v<is_solid<T>>)
    .then([&x]
        {
            x.eat();
            std::cout << "eating solid\n";
        })
    .else_([&x]
        {
            x.consume();
            std::cout << "drinking liquid\n";
        })
    .or_else_([&x]
        {
            std::cout << "cannot consume\n";
        })
    .and_then([&x]
        {
            // ...
        });
```

# C++14 static\_if – capture?

- Would capturing the variable work?

consume(jui

```
struct juice
{
    void drink() { }
};

struct lambda
{
    juice& x;

    auto operator()() const
    {
        x.eat();
        std::cout << "ate solid food\n";
    }
};
```

solid\n";

ng liquid\n";

consume\n";

# C++14 static\_if – capture?

- Would capturing the variable work?

consume(jui

```
struct juice
{
    void drink() { }
};

struct lambda
{
    juice& x;

    auto operator()() const
    {
        x.eat();
        std::cout << "ate solid food\n";
    }
};
```

solid\n";

>)

ng liquid\n";

consume\n";

# C++14 static\_if – capture?

- Would capture the variable work?

consume(juice)

```
struct juice
{
    void drink() { }
};

struct lambda
{
    juice& x;

    auto operator()() const
    {
        x.eat();
        std::cout << "ate solid food\n";
    }
};
```

solid\n";

This gets “immediately instantiated” and causes a compilation error.

consume\n";

**code();**

# Questions?

<http://vittorioromeo.info>  
<http://github.com/SuperV1234>

[vittorio.romeo@outlook.com](mailto:vittorio.romeo@outlook.com)  
[vromeo5@bloomberg.net](mailto:vromeo5@bloomberg.net)

Thank you for attending!