



# Overengineering $\max(a, b)$

Mixed comparison functions, Common references, and Rust's Lifetime Annotations

**Jonathan Müller**

**2024**

# What this talk is about

- We are going to implement `max(a, b)`.

# What this talk is about

- We are going to implement `max(a, b)`.
- We are embarking on a journey of accidental and essential complexity.

# What this talk is about

- We are going to implement `max(a, b)`.
- We are embarking on a journey of accidental and essential complexity.
- We are going to look at the worst feature C++ has ever added.

# The maximum function

## Definition

The maximum function `max(a, b)` returns the greater of the two arguments.

**How hard can it be?**

# The maximum function

```
int max(int a, int b)
{
    if (a < b)
        return b;
    else
        return a;
}
```

# The maximum function

```
int max(int a, int b)
{
    return a < b ? b : a;
}
```



# The maximum function

```
[[nodiscard]] constexpr int max(int a, int b) noexcept
{
    return a < b ? b : a;
}
```

## Level 0: Let's make it generic

## Level 0: Let's make it generic

```
template <typename T>  
const T& max(const T& a, const T& b)  
{  
    return a < b ? b : a;  
}
```

## Level 0: Let's make it generic

```
template <typename T>  
const T& max(const T& a, const T& b)  
{  
    return a < b ? b : a;  
}
```

Are we done?

## What are the type requirements?

# Type requirements

## Syntactic requirements

T must provide an operator `<` that returns something contextually convertible to `bool`.

# Type requirements

## Syntactic requirements

T must provide an operator`<` that returns something contextually convertible to `bool`.

## Semantic requirements

The operator`<` must implement a strict total order.

## Definition

A strict total order of a set  $S$  is a binary relation  $< \subseteq S \times S$  that has three properties:



## Definition

A strict total order of a set  $S$  is a binary relation  $< \subseteq S \times S$  that has three properties:

1 **Irreflexivity:**  $a \not< a$  for all  $a \in S$ .

## Definition

A strict total order of a set  $S$  is a binary relation  $< \subseteq S \times S$  that has three properties:

- 1 **Irreflexivity:**  $a \not< a$  for all  $a \in S$ .
- 2 **Transitivity:** If  $a < b$  and  $b < c$ , then  $a < c$  for all  $a, b, c \in S$ .

## Definition

A strict total order of a set  $S$  is a binary relation  $< \subseteq S \times S$  that has three properties:

- 1 **Irreflexivity:**  $a \not< a$  for all  $a \in S$ .
- 2 **Transitivity:** If  $a < b$  and  $b < c$ , then  $a < c$  for all  $a, b, c \in S$ .
- 3 **Trichotomy:** For all  $a, b \in S$ , exactly one of  $a < b$ ,  $a = b$ , or  $b < a$  holds.

## Level 0: Let's make it generic

```
template <typename T>
    // Requires: T has an operator< that implements a strict total order.
    const T& max(const T& a, const T& b)
{
    return a < b ? b : a;
}
```

## Level 0: Let's make it generic

```
template <typename T>
    // Requires: T has an operator< that implements a strict total order.
    const T& max(const T& a, const T& b)
{
    return a < b ? b : a;
}
```

Are we done?

## Level 0: Let's make it generic

```
struct person {
    int id;
    std::string name;
};

struct person_by_name {
    person* p;
    auto operator<=>(const person_by_name& other) const {
        return p->name <=> other.p->name;
    }
};

auto result = max(person_by_name{&p1}, person_by_name{&p2});
```

## Level 0: Let's make it generic

```
auto p1 = person{1, "Jonathan"};  
auto p2 = person{2, "Jonathan"};  
auto result = max(person_by_name{&p1}, person_by_name{&p2});
```

## Level 0: Let's make it generic

```
auto p1 = person{1, "Jonathan"};  
auto p2 = person{2, "Jonathan"};  
auto result = max(person_by_name{&p1}, person_by_name{&p2});
```

This violates trichotomy.



## Definition

A strict weak order of a set  $S$  is a binary relation  $< \subseteq S \times S$  that has four properties:

## Definition

A strict weak order of a set  $S$  is a binary relation  $< \subseteq S \times S$  that has four properties:

- 1 **Irreflexivity:**  $a \not< a$  for all  $a \in S$ .

## Definition

A strict weak order of a set  $S$  is a binary relation  $< \subseteq S \times S$  that has four properties:

- 1 **Irreflexivity:**  $a \not< a$  for all  $a \in S$ .
- 2 **Transitivity:** If  $a < b$  and  $b < c$ , then  $a < c$  for all  $a, b, c \in S$ .

## Definition

A strict weak order of a set  $S$  is a binary relation  $< \subseteq S \times S$  that has four properties:

- 1 **Irreflexivity:**  $a \not< a$  for all  $a \in S$ .
- 2 **Transitivity:** If  $a < b$  and  $b < c$ , then  $a < c$  for all  $a, b, c \in S$ .
- 3 **Asymmetry:** If  $a < b$ , then  $b \not< a$  for all  $a, b \in S$

## Definition

A strict weak order of a set  $S$  is a binary relation  $< \subseteq S \times S$  that has four properties:

- 1 **Irreflexivity:**  $a \not< a$  for all  $a \in S$ .
- 2 **Transitivity:** If  $a < b$  and  $b < c$ , then  $a < c$  for all  $a, b, c \in S$ .
- 3 **Asymmetry:** If  $a < b$ , then  $b \not< a$  for all  $a, b \in S$
- 4 **Transitivity of incomparability:** Define  $a \sim b$  if neither  $a < b$  nor  $b < a$ . If  $a \sim b$  and  $b \sim c$ , then  $a \sim c$  for all  $a, b, c \in S$ .

## Definition

A strict weak order of a set  $S$  is a binary relation  $< \subseteq S \times S$  that has four properties:

- 1 **Irreflexivity:**  $a \not< a$  for all  $a \in S$ .
- 2 **Transitivity:** If  $a < b$  and  $b < c$ , then  $a < c$  for all  $a, b, c \in S$ .
- 3 **Asymmetry:** If  $a < b$ , then  $b \not< a$  for all  $a, b \in S$
- 4 **Transitivity of incomparability:** Define  $a \sim b$  if neither  $a < b$  nor  $b < a$ . If  $a \sim b$  and  $b \sim c$ , then  $a \sim c$  for all  $a, b, c \in S$ .

## Definition

A strict weak order of a set  $S$  is a binary relation  $< \subseteq S \times S$  that has four properties:

- 1 **Irreflexivity:**  $a \not< a$  for all  $a \in S$ .
- 2 **Transitivity:** If  $a < b$  and  $b < c$ , then  $a < c$  for all  $a, b, c \in S$ .
- 3 **Asymmetry:** If  $a < b$ , then  $b \not< a$  for all  $a, b \in S$
- 4 **Transitivity of incomparability:** Define  $a \sim b$  if neither  $a < b$  nor  $b < a$ . If  $a \sim b$  and  $b \sim c$ , then  $a \sim c$  for all  $a, b, c \in S$ .

A strict weak order of a set  $S$  defines a strict total order of the equivalence classes of  $\sim$ .

## Level 0: Let's make it generic

```
template <typename T>
    // Requires: T has an operator< that implements a strict weak order.
    const T& max(const T& a, const T& b)
{
    return a < b ? b : a;
}
```



## Level 0: Let's make it generic

```
template <typename T>
    // Requires: T has an operator< that implements a strict weak order.
    const T& max(const T& a, const T& b)
{
    return a < b ? b : a;
}
```

```
auto p1 = person{1, "Jonathan"};
auto p2 = person{2, "Jonathan"};
auto result = max(person_by_name{&p1}, person_by_name{&p2});
```

Should this be p1 or p2?

## Level 0: Let's make it generic

```
template <typename T>
    // Requires: T has an operator< that implements a strict weak order.
    const T& max(const T& a, const T& b)
{
    return a < b ? b : a;
}
```

## Level 0: Let's make it generic

```
template <typename T>
    // Requires: T has an operator< that implements a strict weak order.
    const T& max(const T& a, const T& b)
{
    return a < b ? b : a;
}
```

```
template <typename T>
    // Requires: T has an operator< that implements a strict weak order.
    const T& max(const T& a, const T& b)
{
    return b < a ? a : b;
}
```

## Level 1: Let's support mixed comparisons

## Level 1: Let's support mixed comparisons

```
std::string a = "hello";  
std::string_view b = "world";  
std::print("{}\n", max(a, b)); // error
```

## Level 1: Let's support mixed comparisons

```
template <typename T, typename U>
    // Requires: T and U have a common operator< that
    // implements a strict weak order.
auto max(const T& a, const U& b)
{
    return a < b ? b : a;
}
```

## Level 1: Let's support mixed comparisons

```
template <typename T, typename U>
    // Requires: T and U have a common operator< that
    // implements a strict weak order.
auto max(const T& a, const U& b) -> std::common_type_t<T, U>
{
    return a < b ? b : a;
}
```

## Aside: `std::common_type` is weird

`std::common_type_t<T, U>` is essentially:

```
std::decay_t<decltype(  
    false  
    ? std::declval<std::decay_t<T>>()  
    : std::declval<std::decay_t<U>>()  
)>;
```



## Aside: `std::common_type` is weird

`std::common_type_t<T, U>` is essentially:

```
std::decay_t<decltype(  
    false  
    ? std::decay_t<T>() :  
    : std::decay_t<U>())  
>;
```

Plus some weird exception.

## Aside: `std::common_type` is user-customizable

You can add specialize `std::common_type` for your own types!

## Aside: `std::common_type` is user-customizable

You can add specialize `std::common_type` for your own types!

This does not affect `? :`.

## Aside: `std::common_type` is not associative

```
template <typename ... T>  
using std::common_type_t = ...;
```

## Aside: `std::common_type` is not associative

```
template <typename ... T>  
using std::common_type_t = ...;
```

```
std::common_type_t<A, B, C> != std::common_type_t<B, C, A>.
```

## Aside: `std::common_type` is not associative

```
template <typename ... T>  
using std::common_type_t = ...;
```

`std::common_type_t<A, B, C> != std::common_type_t<B, C, A>.`

```
struct A {};  
struct B : A {};  
struct C : A {};
```

## Level 1: Let's support mixed comparisons

```
std::string a = "hello";  
std::string_view b = "world";  
std::print("{}\n", max(a, b));
```

world

## Level 1: Let's support mixed comparisons

```
int      a = -1;  
unsigned b = 1;  
std::print("{}\n", max(a, b));
```



## Level 1: Let's support mixed comparisons

```
int      a = -1;  
unsigned b = 1;  
std::print("{}\n", max(a, b));
```

4294967295

# Problem #1: Signed/unsigned comparison

`int() < unsigned()` can be implemented in three ways:

# Problem #1: Signed/unsigned comparison

`int() < unsigned()` can be implemented in three ways:

- 1 Compare the mathematical value they represent (correct).

# Problem #1: Signed/unsigned comparison

`int() < unsigned()` can be implemented in three ways:

- 1 Compare the mathematical value they represent (correct).
- 2 Convert both to `int`, then compare two `ints`.

# Problem #1: Signed/unsigned comparison

`int() < unsigned()` can be implemented in three ways:

- 1 Compare the mathematical value they represent (correct).
- 2 Convert both to `int`, then compare two `ints`.
- 3 Convert both to `unsigned`, then compare two `unsigneds` (C++).

# Problem #1: Signed/unsigned comparison

`int() < unsigned()` can be implemented in three ways:

- 1 Compare the mathematical value they represent (correct).
- 2 Convert both to `int`, then compare two `ints`.
- 3 Convert both to `unsigned`, then compare two `unsigneds` (C++).

# Problem #1: Signed/unsigned comparison

`int() < unsigned()` can be implemented in three ways:

- 1 Compare the mathematical value they represent (correct).
- 2 Convert both to `int`, then compare two `ints`.
- 3 Convert both to `unsigned`, then compare two `unsigneds` (C++).

`std::cmp_less` does the correct thing.

## Problem #2: Signed/unsigned conversion

`std::common_type_t<int, unsigned>` can be one of:



## Problem #2: Signed/unsigned conversion

`std::common_type_t<int, unsigned>` can be one of:

- 1 The next larger signed integer type that can fit both types (correct).

## Problem #2: Signed/unsigned conversion

`std::common_type_t<int, unsigned>` can be one of:

- 1 The next larger signed integer type that can fit both types (correct).
- 2 `int`.

## Problem #2: Signed/unsigned conversion

`std::common_type_t<int, unsigned>` can be one of:

- 1 The next larger signed integer type that can fit both types (correct).
- 2 `int`.
- 3 `unsigned` (C++).

## Problem #2: Signed/unsigned conversion

`std::common_type_t<int, unsigned>` can be one of:

- 1 The next larger signed integer type that can fit both types (correct).
- 2 `int`.
- 3 `unsigned` (C++).
- 4 invalid (think-cell).

```
template <typename Source, typename Target>  
concept tc::safely_convertible_to = ...;
```

- `std::convertible_to<Source, Target>`, and
- that conversion is *safe*

```
template <typename Source, typename Target>  
concept tc::safely_convertible_to = ...;
```

- `std::convertible_to<Source, Target>`, and
- that conversion is *safe*

Conversions which are not safe:

- Derived to Base (slicing)

```
template <typename Source, typename Target>  
concept tc::safely_convertible_to = ...;
```

- `std::convertible_to<Source, Target>`, and
- that conversion is *safe*

Conversions which are not safe:

- Derived to Base (slicing)
- `std::string` to `std::string_view` (potentially dangling)

```
template <typename Source, typename Target>  
concept tc::safely_convertible_to = ...;
```

- `std::convertible_to<Source, Target>`, and
- that conversion is *safe*

Conversions which are not safe:

- Derived to Base (slicing)
- `std::string` to `std::string_view` (potentially dangling)
- `int` to `unsigned`



```
template <typename ... Ts>  
using tc::common_type_t = ...;
```

```
std::common_type_t<Ts...> provided tc::safely_convertible_to<Ts,  
std::common_type_t<Ts...>> && ....
```

## Level 1: Let's support mixed comparisons

```
template <typename T, typename U>
    // Requires: T and U have a common operator< that
    // implements a strict weak order.
auto max(const T& a, const U& b) -> tc::common_type_t<T, U>
{
    return a < b ? b : a;
}
```

## Level 1: Let's support mixed comparisons

```
template <typename T, typename U>
    // Requires: T and U have a common operator< that
    // implements a strict weak order.
auto max(const T& a, const U& b) -> tc::common_type_t<T, U>
{
    return a < b ? b : a;
}
```

```
int    a = -1;
unsigned b = 1;
std::print("{}\n", max(a, b)); // error
```

## Level 2: Let's support perfect forwarding

## Level 2: Let's support perfect forwarding

```
template <typename T, typename U>  
    // Requires: T and U have a common operator< that  
    // implements a strict weak order.  
auto max(const T& a, const U& b) -> tc::common_type_t<T, U>  
{  
    return a < b ? b : a;  
}
```

## Level 2: Let's support perfect forwarding

```
template <typename T, typename U>
    // Requires: T and U have a common operator< that
    // implements a strict weak order.
auto max(const T& a, const U& b) -> tc::common_type_t<T, U>
{
    return a < b ? b : a;
}
```

```
std::string a = "hello";
std::string b = "world";
std::print("{}\n", max(a, b));
```

## Level 2: Let's support perfect forwarding

```
template <typename T, typename U>  
    // Requires: T and U have a common operator< that  
    // implements a strict weak order.  
auto max(const T& a, const U& b) -> tc::common_type_t<T, U>  
{  
    return a < b ? b : a;  
}
```

```
std::string a = "hello";  
std::string b = "world";  
std::print("{}\n", max(a, b));
```

Unnecessary copy of the result.

## Level 2: Let's support perfect forwarding

```
template <typename T, typename U>
    // Requires: T and U have a common operator< that
    // implements a strict weak order.
auto max(const T& a, const U& b)
    -> std::conditional_t<
        std::same_as<T, U>,
        const T&, tc::common_type_t<T, U>
    >
{
    return a < b ? b : a;
}
```



## Level 2: Let's support perfect forwarding

```
template <typename T, typename U>
    // Requires: T and U have a common operator< that
    // implements a strict weak order.
auto max(const T& a, const U& b)
    -> std::conditional_t<
        std::same_as<T, U>,
        const T&, tc::common_type_t<T, U>
    >
{
    return a < b ? b : a;
}
```

```
auto a = std::string_view("hello");
std::print("{}\n", max(a, std::string("world")));
```

## Level 2: Let's support perfect forwarding

```
template <typename T, typename U>
    // Requires: T and U have a common operator< that
    // implements a strict weak order.
auto max(const T& a, const U& b)
    -> std::conditional_t<
        std::same_as<T, U>,
        const T&, tc::common_type_t<T, U>
    >
{
    return a < b ? b : a;
}
```

```
auto a = std::string_view("hello");
std::print("{}\n", max(a, std::string("world")));
```

Unnecessary copy of `std::string("world")`.

## Level 2: Let's support perfect forwarding

```
template <typename T, typename U>
    // Requires: T and U have a common operator< that
    // implements a strict weak order.
    requires requires { typename tc::common_type_t<T, U>; }
decltype(auto) max(T&& a, U&& b)
{
    return a < b ? std::forward<U>(b) : std::forward<T>(a);
}
```

## Level 2: Let's support perfect forwarding

```
template <typename T, typename U>
    // Requires: T and U have a common operator< that
    // implements a strict weak order.
    requires requires { typename tc::common_type_t<T, U>; }
decltype(auto) max(T&& a, U&& b)
{
    return a < b ? std::forward<U>(b) : std::forward<T>(a);
}
```

What if `std::remove_cvref_t<T>` and `std::remove_cvref_t<U>` are the same type?

# What does the ternary operator do?

<code>?:</code>	<code>T&amp;</code>	<code>const T&amp;</code>	<code>T&amp;&amp;</code>	<code>const T&amp;&amp;</code>
<code>T&amp;</code>	<code>T&amp;</code>	<code>const T&amp;</code>	<code>T</code>	<code>T</code>
<code>const T&amp;</code>	<code>const T&amp;</code>	<code>const T&amp;</code>	<code>T</code>	<code>T</code>
<code>T&amp;&amp;</code>	<code>T</code>	<code>T</code>	<code>T&amp;&amp;</code>	<code>const T&amp;&amp;</code>
<code>const T&amp;&amp;</code>	<code>T</code>	<code>T</code>	<code>const T&amp;&amp;</code>	<code>const T&amp;&amp;</code>

# What does the ternary operator do?

<code>?:</code>	<code>T&amp;</code>	<code>const T&amp;</code>	<code>T&amp;&amp;</code>	<code>const T&amp;&amp;</code>
<code>T&amp;</code>	<code>T&amp;</code>	<code>const T&amp;</code>	<code>T</code>	<code>T</code>
<code>const T&amp;</code>	<code>const T&amp;</code>	<code>const T&amp;</code>	<code>T</code>	<code>T</code>
<code>T&amp;&amp;</code>	<code>T</code>	<code>T</code>	<code>T&amp;&amp;</code>	<code>const T&amp;&amp;</code>
<code>const T&amp;&amp;</code>	<code>T</code>	<code>T</code>	<code>const T&amp;&amp;</code>	<code>const T&amp;&amp;</code>

`?:` is a prvalue when mixing an lvalue and an rvalue reference.

## Level 2: Let's support perfect forwarding

```
template <typename T, typename U>
    // Requires: T and U have a common operator< that
    // implements a strict weak order.
    requires requires { typename tc::common_type_t<T, U>; }
decltype(auto) max(T&& a, U&& b)
{
    return a < b ? std::forward<U>(b) : std::forward<T>(a);
}
```

## Level 2: Let's support perfect forwarding

```
template <typename T, typename U>  
    // Requires: T and U have a common operator< that  
    // implements a strict weak order.  
    requires requires { typename tc::common_type_t<T, U>; }  
decltype(auto) max(T&& a, U&& b)  
{  
    return a < b ? std::forward<U>(b) : std::forward<T>(a);  
}
```

```
auto a = std::string("world");  
std::print("{}\n", max(a, std::string("hello")));
```



## Level 2: Let's support perfect forwarding

```
template <typename T, typename U>
    // Requires: T and U have a common operator< that
    // implements a strict weak order.
    requires requires { typename tc::common_type_t<T, U>; }
decltype(auto) max(T&& a, U&& b)
{
    return a < b ? std::forward<U>(b) : std::forward<T>(a);
}
```

```
auto a = std::string("world");
std::print("{}\n", max(a, std::string("hello")));
```

Unnecessary copy of a.

# What does `std::common_reference` do?

```
template <typename ... Ts>  
using std::common_reference_t = ...;
```

- If `std::remove_cvref_t<T>` and `std::remove_cvref_t<U>` are the same type, figure out the correct cv-ref qualification.
- Otherwise, use `std::common_type_t<T, U>`

# What does `std::common_reference` do?

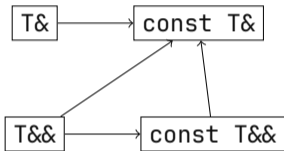
<code>std::common_reference</code>	<code>T&amp;</code>	<code>const T&amp;</code>	<code>T&amp;&amp;</code>	<code>const T&amp;&amp;</code>
<code>T&amp;</code>	<code>T&amp;</code>	<code>const T&amp;</code>	<code>const T&amp;</code>	<code>const T&amp;</code>
<code>const T&amp;</code>	<code>const T&amp;</code>	<code>const T&amp;</code>	<code>const T&amp;</code>	<code>const T&amp;</code>
<code>T&amp;&amp;</code>	<code>const T&amp;</code>	<code>const T&amp;</code>	<code>T&amp;&amp;</code>	<code>const T&amp;&amp;</code>
<code>const T&amp;&amp;</code>	<code>const T&amp;</code>	<code>const T&amp;</code>	<code>const T&amp;&amp;</code>	<code>const T&amp;&amp;</code>

# What does `std::common_reference` do?

<code>std::common_reference</code>	<code>T&amp;</code>	<code>const T&amp;</code>	<code>T&amp;&amp;</code>	<code>const T&amp;&amp;</code>
<code>T&amp;</code>	<code>T&amp;</code>	<code>const T&amp;</code>	<code>const T&amp;</code>	<code>const T&amp;</code>
<code>const T&amp;</code>	<code>const T&amp;</code>	<code>const T&amp;</code>	<code>const T&amp;</code>	<code>const T&amp;</code>
<code>T&amp;&amp;</code>	<code>const T&amp;</code>	<code>const T&amp;</code>	<code>T&amp;&amp;</code>	<code>const T&amp;&amp;</code>
<code>const T&amp;&amp;</code>	<code>const T&amp;</code>	<code>const T&amp;</code>	<code>const T&amp;&amp;</code>	<code>const T&amp;&amp;</code>

`std::common_reference` is a `const lvalue reference` when mixing an `lvalue` and an `rvalue reference`.

## Aside: Implicit conversions of references



Ergo: the common reference is `const T&` as that is the universal receiver.

**Is that ideal?**

### Is that ideal?

binds to	Temporary	const
T&	no	no
const T&	yes	yes
T&&	yes	no
const T&&	yes	yes

Functions that don't care about lifetime or mutation, use `const T&`.

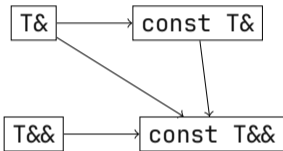
Binds to	Temporary	const
T&	no	no
const T&	no	yes
T&&	yes	no
const T&&	yes	yes

Functions that don't care about lifetime or mutation, use `const T&&`.



# Aside: Implicit conversions of references

Better



Ergo: the common reference is `const T&&` as that is the universal receiver.

```
template <typename ... Ts>  
using tc::common_reference_t = ...;
```

- `tc::common_reference_t<int&, int&&>` is `const int&&`, not `const int&`.
- Uses `tc::common_type_t` not `std::common_type_t`.

# think-cell: Safe common reference

---

<code>tc::common_reference</code>	<code>T&amp;</code>	<code>const T&amp;</code>	<code>T&amp;&amp;</code>	<code>const T&amp;&amp;</code>
<code>T&amp;</code>	<code>T&amp;</code>	<code>const T&amp;</code>	<code>const T&amp;&amp;</code>	<code>const T&amp;&amp;</code>
<code>const T&amp;</code>	<code>const T&amp;</code>	<code>const T&amp;</code>	<code>const T&amp;&amp;</code>	<code>const T&amp;&amp;</code>
<code>T&amp;&amp;</code>	<code>const T&amp;&amp;</code>	<code>const T&amp;&amp;</code>	<code>T&amp;&amp;</code>	<code>const T&amp;&amp;</code>
<code>const T&amp;&amp;</code>	<code>const T&amp;&amp;</code>	<code>const T&amp;&amp;</code>	<code>const T&amp;&amp;</code>	<code>const T&amp;&amp;</code>

---

## Level 2: Let's support perfect forwarding

```
template <typename T, typename U>
    // Requires: T and U have a common operator< that
    // implements a strict weak order.
auto max(T&& a, U&& b)
    -> tc::common_reference_t<T, U>
{
    return a < b ? std::forward<U>(b) : std::forward<T>(a);
}
```

## Level 2: Let's support perfect forwarding

```
template <typename T, typename U>
    // Requires: T and U have a common operator< that
    // implements a strict weak order.
auto max(T&& a, U&& b)
    -> tc::common_reference_t<T, U>
{
    return a < b ? std::forward<U>(b) : std::forward<T>(a);
}
```

? : is a prvalue when mixing lvalues and rvalues!

## Level 2: Let's support perfect forwarding

```
template <typename T, typename U>
    // Requires: T and U have a common operator< that
    // implements a strict weak order.
auto max(T&& a, U&& b)
    -> tc::common_reference_t<T, U>
{
    if (a < b)
        return std::forward<U>(b);
    else
        return std::forward<T>(a);
}
```

## Level 2: Let's support perfect forwarding

```
template <typename T, typename U>
    // Requires: T and U have a common operator< that
    // implements a strict weak order.
auto max(T&& a, U&& b)
    -> tc::common_reference_t<T, U>
{
    if (a < b)
        return static_cast<tc::common_reference_t<T, U>>(
            std::forward<U>(b)
        );
    else
        return static_cast<tc::common_reference_t<T, U>>(
            std::forward<T>(a)
        );
}
```



## Level 2: Let's support perfect forwarding

```
template <typename T, typename U>
    // Requires: T and U have a common operator< that
    // implements a strict weak order.
    auto max(T&& a, U&& b)
        -> tc::common_reference_t<T, U>
{
    return tc_conditional_rvalue_as_ref(
        a < b,
        std::forward<U>(b),
        std::forward<T>(a)
    );
}
```



## Aside: tc\_conditional\_rvalue\_as\_ref

```
#define tc_conditional_rvalue_as_ref(cond, lhs, rhs) \  
    ((cond) \  
     ? (tc::common_reference_t<decltype((lhs)), decltype((rhs))>)(lhs) \  
     : (tc::common_reference_t<decltype((lhs)), decltype((rhs))>)(rhs))
```

## Aside: tc\_conditional\_rvalue\_as\_ref

```
#define tc_conditional_rvalue_as_ref(cond, lhs, rhs) \  
    ((cond) \  
     ? (tc::common_reference_t<decltype((lhs)), decltype((rhs))>)(lhs) \  
     : (tc::common_reference_t<decltype((lhs)), decltype((rhs))>)(rhs))
```

```
struct A {};
```

```
struct B : A { using base = A; }
```

```
struct C : A { using base = A; }
```

```
struct D : C { using base = C; }
```

```
B b;
```

```
D d;
```

```
A& ref = tc_conditional_rvalue_as_ref(cond, b, d);
```

Level 3: `clamp(v, lo, hi)`

## Level 3: clamp(v, lo, hi)

```
template <typename T, typename U>
auto max(T&& a, U&& b) -> tc::common_reference_t<T, U> {
    return tc_conditional_rvalue_as_ref(
        a < b,
        std::forward<U>(b), std::forward<T>(a)
    );
}
```

## Level 3: clamp(v, lo, hi)

```
template <typename T, typename U>
auto max(T&& a, U&& b) -> tc::common_reference_t<T, U> {
    return tc_conditional_rvalue_as_ref(
        a < b,
        std::forward<U>(b), std::forward<T>(a)
    );
}
```

```
template <typename T, typename U>
auto min(T&& a, U&& b) -> tc::common_reference_t<T, U> {
    return tc_conditional_rvalue_as_ref(
        a > b,
        std::forward<U>(b), std::forward<T>(a)
    );
}
```



## Level 3: clamp(v, lo, hi)

```
template <typename T, typename U, typename V>
decltype(auto) clamp(T&& v, U&& lo, V&& hi)
{
    return min(
        max(std::forward<T>(v), std::forward<U>(lo)),
        std::forward<V>(hi)
    );
}
```

## Level 3: clamp(v, lo, hi)

```
template <typename T, typename U, typename V>
decltype(auto) clamp(T&& v, U&& lo, V&& hi)
{
    return min(
        max(std::forward<T>(v), std::forward<U>(lo)),
        std::forward<V>(hi)
    );
}
```

```
std::string lo = "a";
std::string hi = "z";
auto clamped = clamp(std::string("h"), lo, hi);
```

## Level 3: clamp(v, lo, hi)

```
template <typename T, typename U, typename V>
decltype(auto) clamp(T&& v, U&& lo, V&& hi)
{
    return min(
        max(std::forward<T>(v), std::forward<U>(lo)),
        std::forward<V>(hi)
    );
}
```

```
std::string_view lo = "a";
std::string hi = "z";
auto clamped = clamp(std::string("h"), lo, hi);
```



## Level 3: clamp(v, lo, hi)

```
template <typename T, typename U, typename V>
decltype(auto) clamp(
    T&& v, // std::string&&
    U&& lo, // std::string_view&
    V&& hi // std::string&
)
{
    return min(
        max(std::forward<T>(v), std::forward<U>(lo)),
        std::forward<V>(hi)
    );
}
```

## Level 3: clamp(v, lo, hi)

```
template <typename T, typename U, typename V>
decltype(auto) clamp(
    T&& v, // std::string&&
    U&& lo, // std::string_view&
    V&& hi // std::string&
)
{
    return min(
        max(std::forward<T>(v), std::forward<U>(lo)), // std::string
        std::forward<V>(hi)
    );
}
```

## Level 3: clamp(v, lo, hi)

```
template <typename T, typename U, typename V>
decltype(auto) clamp(
    T&& v, // std::string&&
    U&& lo, // std::string_view&
    V&& hi // std::string&
)
{
    return min(
        max(std::forward<T>(v), std::forward<U>(lo)), // std::string
        std::forward<V>(hi) // std::string&
    );
}
```

## Level 3: clamp(v, lo, hi)

```
template <typename T, typename U, typename V>
decltype(auto) clamp(
    T&& v, // std::string&&
    U&& lo, // std::string_view&
    V&& hi // std::string&
)
{
    return min(
        max(std::forward<T>(v), std::forward<U>(lo)), // std::string
        std::forward<V>(hi) // std::string&
    ); // const std::string&&
}
```

**Whose fault is it?**

### Whose fault is it?

**Projection** A function that returns a reference whose lifetime is tied to one of the arguments.

**Composed Projection** A projection built by composing multiple projections.

### Whose fault is it?

**Projection** A function that returns a reference whose lifetime is tied to one of the arguments.

**Composed Projection** A projection built by composing multiple projections.

**Problem:** If projections return rvalue references to prvalue arguments, composing them can lead to returning dangling references.

## Level 3: `clamp(v, lo, hi)`

**Solution 1:** A projection never returns an rvalue reference.



## Level 3: clamp(v, lo, hi)

**Solution 1:** A projection never returns an rvalue reference.

```
template <typename T>
using decay_rvalue_t = std::conditional_t<
    std::is_rvalue_reference_v<T>, std::decay_t<T>, T
>;
```

```
template <typename T, typename U>
auto max(T&& a, U&& b)
    -> decay_rvalue_t<tc::common_reference_t<T, U>>
{
    return tc_conditional_rvalue_as_ref(
        a < b, std::forward<U>(b), std::forward<T>(a)
    );
}
```

## Level 3: clamp(v, lo, hi)

**Solution 1:** A projection never returns an rvalue reference.

```
template <typename T>
using decay_rvalue_t = std::conditional_t<
    std::is_rvalue_reference_v<T>, std::decay_t<T>, T
>;
```

```
template <typename T, typename U>
auto max(T&& a, U&& b)
    -> decay_rvalue_t<tc::common_reference_t<T, U>>
{
    return tc_conditional_rvalue_as_ref(
        a < b, std::forward<U>(b), std::forward<T>(a)
    );
}
```

**Problem:** Too aggressive.

## Level 3: `clamp(v, lo, hi)`

**Solution 2:** A projection never returns an rvalue reference to a prvalue argument.

## Level 3: clamp(v, lo, hi)

**Solution 2:** A projection never returns an rvalue reference to a prvalue argument.

```
template <typename T, typename ... Args>  
using decay_if_prvalue_t = ???;
```

```
template <typename T, typename U>  
auto max(T&& a, U&& b)  
    -> decay_if_prvalue_t<tc::common_reference_t<T, U>, T, U>  
{  
    return tc_conditional_rvalue_as_ref(  
        a < b, std::forward<U>(b), std::forward<T>(a)  
    );  
}
```

## Level 3: clamp(v, lo, hi)

**Solution 2:** A projection never returns an rvalue reference to a prvalue argument.

```
template <typename T, typename ... Args>
using decay_if_prvalue_t = ???;
```

```
template <typename T, typename U>
auto max(T&& a, U&& b)
    -> decay_if_prvalue_t<tc::common_reference_t<T, U>, T, U>
{
    return tc_conditional_rvalue_as_ref(
        a < b, std::forward<U>(b), std::forward<T>(a)
    );
}
```

**Problem:** Not actually implementable.

## Aside: Forwarding references

```
template <typename T>  
void f(T&& t);
```

Category	T	T&&
lvalue	U&	U&
xvalue	U	U&&
prvalue	U	U&&

## Aside: Forwarding references

```
template <typename T>  
void f(T&& t);
```

Category	T	T&&
lvalue	U&	U&
xvalue	U	U&&
prvalue	U	U&&

A function cannot distinguish between prvalues and xvalue arguments.

```
template <typename T>  
void f(T&& t);
```

Category	T	T&&
lvalue	U&	U&
xvalue	U&&	U&&
prvalue	U	U&&



## Level 3: clamp(v, lo, hi)

```
return min(  
    max(std::forward<T>(v), std::forward<U>(lo)),  
    std::forward<V>(hi)  
); // dangling
```

## Level 3: clamp(v, lo, hi)

```
return min(  
    max(std::forward<T>(v), std::forward<U>(lo)),  
    std::forward<V>(hi)  
); // dangling
```

```
some_function(min(  
    max(std::forward<T>(v), std::forward<U>(lo)),  
    std::forward<V>(hi)  
)); // fine
```

## Level 3: clamp(v, lo, hi)

```
return min(  
    max(std::forward<T>(v), std::forward<U>(lo)),  
    std::forward<V>(hi)  
); // dangling
```

```
some_function(min(  
    max(std::forward<T>(v), std::forward<U>(lo)),  
    std::forward<V>(hi)  
)); // fine
```

**The user of a composed projection should be responsible for preventing dangling references!**

## Level 3: `clamp(v, lo, hi)`

**Solution 3:** A composed projection never returns a reference to an intermediate prvalue.

## Level 3: clamp(v, lo, hi)

**Solution 3:** A composed projection never returns a reference to an intermediate prvalue.

```
template <typename T>
auto decay_rvalue(T&& v) -> decay_rvalue_t<T&&>
{
    return std::forward<T>(v);
}
```

```
template <typename T, typename U, typename V>
decltype(auto) clamp(T&& v, U&& lo, V&& hi)
{
    return decay_rvalue(min(
        max(std::forward<T>(v), std::forward<U>(lo)),
        std::forward<V>(hi)
    ));
}
```



## Level 3: clamp(v, lo, hi)

**Solution 3:** A composed projection never returns a reference to an intermediate prvalue.

```
template <bool Cond, typename T>
auto decay_if(T&& v)
    -> std::conditional_t<Cond, std::decay_t<T>, T&&>
{
    return std::forward<T>(v);
}
```

```
template <typename T, typename U, typename V>
decltype(auto) clamp(T&& v, U&& lo, V&& hi) {
    using max_t = decltype(max(std::forward<T>(v), std::forward<U>(lo)));
    return decay_if<!std::is_reference_v<max_t>>(min(
        max(std::forward<T>(v), std::forward<U>(lo)),
        std::forward<V>(hi)
    ));
}
```

## Level 4: Generic compositions

## Level 4: Generic compositions

```
template <typename F, typename G, typename H>
struct dovekie
{
    F f;
    G g;
    H h;

    template <typename T, typename U>
    decltype(auto) operator()(T&& t, U&& u) const
    {
        return f(g(std::forward<T>(t)), h(std::forward<U>(u)));
    }
};
```



## Level 4: Generic compositions

```
template <typename F, typename G, typename H>
struct dovekie
{
    F f;
    G g;
    H h;

    template <typename T, typename U>
    decltype(auto) operator()(T&& t, U&& u) const
    {
        return decay_if<g_or_h_returns_prvalue>(
            f(g(std::forward<T>(t)), h(std::forward<U>(u)))
        );
    }
};
```

## Level 4: Generic compositions

```
template <typename F, typename G, typename H>
struct dovekie
{
    F f;
    G g;
    H h;

    template <typename T, typename U>
    decltype(auto) operator()(T&& t, U&& u) const
    {
        return decay_if<f_returns_an_rvalue_to_prvalue_of_g_or_h>(
            f(g(std::forward<T>(t)), h(std::forward<U>(u)))
        );
    }
};
```

**C++ isn't a particularly great language, but all problems in C++ can be fixed with more C++.**

# tc::temporary - A new reference type

**Idea:** Introduce a special “reference to temporary” type.

## tc::temporary - A new reference type

```
template <typename T> requires std::is_object_v<T>
class temporary {
    T&& m_ref;
public:
    template <std::same_as<T&&> U>
    explicit temporary(U ref) noexcept : m_ref(ref) {}
    template <std::same_as<T&&> U>
    temporary(U ref) noexcept : m_ref(std::move(ref)) {}

    operator T&() const& noexcept {
        return m_ref;
    }
    operator T&&() const&& noexcept {
        return std::move(m_ref);
    }
};
```

# Creating a temporary

```
template <typename T>
using prvalue_as_temporary_t = std::conditional_t<
    std::is_reference<T>::value || tc::is_temporary<T>,
    T,
    tc::temporary<T>
>;

#define tc_prvalue_as_temporary(...) \
    static_cast<tc::prvalue_as_temporary_t<decltype((__VA_ARGS__))>> \
    (__VA_ARGS__)
```

# Creating a temporary

```
int f();  
int object = 0;
```

```
tc_prvalue_as_temporary(object)           // int&
```

# Creating a temporary

```
int f();  
int object = 0;
```

```
tc_prvalue_as_temporary(object)           // int&
```

```
tc_prvalue_as_temporary(std::move(object)) // int&&
```



# Creating a temporary

```
int f();  
int object = 0;
```

```
tc_prvalue_as_temporary(object)           // int&
```

```
tc_prvalue_as_temporary(std::move(object)) // int&&
```

```
tc_prvalue_as_temporary(42)              // tc::temporary<int>
```

# Creating a temporary

```
int f();
```

```
int object = 0;
```

```
tc_prvalue_as_temporary(object)           // int&
```

```
tc_prvalue_as_temporary(std::move(object)) // int&&
```

```
tc_prvalue_as_temporary(42)               // tc::temporary<int>
```

```
tc_prvalue_as_temporary(f())              // tc::temporary<int>
```

```
static_cast<decltype((expr))>(expr)
```

- If `expr` is an lvalue or xvalue, casts it to an lvalue or rvalue reference.
- If `expr` is a prvalue, copy-elision keeps it as a prvalue.

```
static_cast<decltype((expr))>(expr)
```

- If expr is an lvalue or xvalue, casts it to an lvalue or rvalue reference.
- If expr is a prvalue, copy-elision keeps it as a prvalue.

→ no-op

```
static_cast<decltype((expr))>(expr)
```

- If expr is an lvalue or xvalue, casts it to an lvalue or rvalue reference.
- If expr is a prvalue, copy-elision keeps it as a prvalue.

→ no-op

```
static_cast<decltype(expr)>(expr)
```

- If expr is a non-identifier, as above.
- If expr is an identifier referring to an lvalue reference or object, casts it to an lvalue reference.
- If expr is an identifier referring to an rvalue reference, casts it to an rvalue reference.

→ `std::forward`

# Handling a temporary

**Rule:** A `tc::temporary` must never be returned from a function.

# Handling a temporary

**Rule:** A `tc::temporary` must never be returned from a function.

```
template <typename T>
using return_temporary_t = std::conditional_t<
    tc::is_temporary<T>,
    tc::decay_t<T>,
    T
>;

#define tc_return_temporary(...) \
    return static_cast<tc::return_temporary_t<decltype((__VA_ARGS__))>>( \
        __VA_ARGS__ \
    )
```

# Handling a temporary

```
int object;  
tc_return_temporary(object);
```

Equivalent to: `return static_cast<int&>(object);`



# Handling a temporary

```
int object;  
tc_return_temporary(object);
```

Equivalent to: `return static_cast<int&>(object);`

```
int object;  
tc_return_temporary(std::move(object));
```

Equivalent to: `return static_cast<int&&>(std::move(object));`

# Handling a temporary

```
int object;  
tc_return_temporary(object);
```

Equivalent to: `return static_cast<int&>(object);`

```
int object;  
tc_return_temporary(std::move(object));
```

Equivalent to: `return static_cast<int&&>(std::move(object));`

```
tc_return_temporary(42);
```

Equivalent to: `return static_cast<int>(42);` (copy-elision!)

# Handling a temporary

```
int object;  
tc_return_temporary(object);
```

Equivalent to: `return static_cast<int&>(object);`

```
int object;  
tc_return_temporary(std::move(object));
```

Equivalent to: `return static_cast<int&&>(std::move(object));`

```
tc_return_temporary(42);
```

Equivalent to: `return static_cast<int>(42);` (copy-elision!)

```
tc_return_temporary(tc::temporary(42));
```

Equivalent to: `return static_cast<int>(tc::temporary(42));`

## Level 4: Generic compositions

```
template <typename F, typename G, typename H>
struct dovekie
{
    F f;
    G g;
    H h;

    template <typename T, typename U>
    decltype(auto) operator()(T&& t, U&& u) const
    {
        tc_return_temporary(f(
            tc_prvalue_as_temporary(g(std::forward<T>(t))),
            tc_prvalue_as_temporary(h(std::forward<U>(u)))
        ));
    }
};
```



## Level 4: Generic compositions

```
template <typename T, typename U>  
tc::common_reference_t<T, U> f(T&& t, U&& u);
```

Returns a `tc::temporary` if at least one argument is a temporary.

## Level 4: Generic compositions

```
template <typename T, typename U>  
tc::common_reference_t<T, U> f(T&& t, U&& u);
```

Returns a `tc::temporary` if at least one argument is a temporary.

```
template <typename T, typename U>  
T&& f(T&& t, U&& u);
```

Returns a `tc::temporary` if and only if the first argument is a temporary.

## Level 4: Generic compositions

```
template <typename T, typename U>  
tc::common_reference_t<T, U> f(T&& t, U&& u);
```

Returns a `tc::temporary` if at least one argument is a temporary.

```
template <typename T, typename U>  
T&& f(T&& t, U&& u);
```

Returns a `tc::temporary` if and only if the first argument is a temporary.

```
template <typename T, typename U>  
tc::common_type_t<T, U> f(T&& t, U&& u);
```

Never returns a `tc::temporary`.

## Level 4: Generic compositions

```
template <typename F, typename G, typename H>
struct dovekie
{
    F f;
    G g;
    H h;

    template <typename T, typename U>
    decltype(auto) operator()(T&& t, U&& u) const
    {
        tc_return_temporary(f(
            tc_prvalue_as_temporary(g(std::forward<T>(t))),
            tc_prvalue_as_temporary(h(std::forward<U>(u)))
        ));
    }
};
```





# Better invoke

Let's teach it to invoke:

```
#define tc_invoke(f, ...) \  
    tc::invoke_impl(f, /* wrap arguments in tc_prvalue_as_temporary */)
```

# Better invoke

Let's teach it to invoke:

```
#define tc_invoke(f, ...) \  
    tc::invoke_impl(f, /* wrap arguments in tc_prvalue_as_temporary */)\  
  
template <typename T, typename U>  
decltype(auto) operator()(T&& t, U&& u) const  
{  
    tc_return_temporary(  
        tc_invoke(f,  
            tc_invoke(g, std::forward<T>(t)),  
            tc_invoke(h, std::forward<U>(u))  
        )  
    );  
}
```

## Level 4: Generic compositions

**Consequence:** Projections need to be aware of `tc:::temporary`.

## Level 4: Generic compositions

**Consequence:** Projections need to be aware of `tc::temporary`.

```
template <typename T, typename U>
auto max(T&& a, U&& b)
    -> tc::common_reference_t<T, U>
{
    // Implicit conversion of tc::temporary -> T& if necessary.
    return tc_conditional_rvalue_as_ref(
        a < b, std::forward<U>(b), std::forward<T>(a)
    );
}
```

## Level 4: Generic compositions

**Consequence:** Projections need to be aware of `tc::temporary`.

```
template <typename T, typename U>
auto max(T&& a, U&& b)
    -> tc::common_reference_t<T, U>
{
    // Implicit conversion of tc::temporary -> T& if necessary.
    return tc_conditional_rvalue_as_ref(
        a < b, std::forward<U>(b), std::forward<T>(a)
    );
}
```

```
template <typename T>
decltype(auto) get_foo(T&& obj)
{
    return std::forward<T>(obj).foo; // no implicit conversion!
}
```

# Unwrapping temporaries

```
template <typename T>
using unwrap_temporary_t = std::conditional_t<
    tc::is_temporary<T>,
    tc::decay_t<T>&&, T
>;

#define tc_unwrap_temporary(...) \
    static_cast<tc::unwrap_temporary_t<decltype((__VA_ARGS__))>>>( \
        __VA_ARGS__ \
    )
```

# Unwrapping temporaries

```
template <typename T>
decltype(auto) get_foo(T&& obj)
{
    return tc_unwrap_temporary(std::forward<T>(obj)).foo;
}
```

# Unwrapping temporaries

```
template <typename T>
decltype(auto) get_foo(T&& obj)
{
    return tc_unwrap_temporary(std::forward<T>(obj)).foo;
}
```

No `tc::temporary` annotation on the result.



# Rewrapping temporaries

```
template <typename Result, typename ... Args>
using rewrap_temporary_t = std::conditional_t<
    std::is_rvalue_reference_v<Result> && (tc::is_temporary<Args> || ...),
    tc::temporary<std::remove_reference_t<Result>>,
    Result
>;

#define tc_rewrap_temporary(Args, ...) \
    static_cast<tc::rewrap_temporary_t<decltype((__VA_ARGS__)), Args>( \
        __VA_ARGS__ \
    )
```

If the result is an rvalue reference and any of the arguments is a `tc::temporary`, wrap it in a `tc::temporary`.

# Rewrapping temporaries

```
template <typename Result, typename ... Args>
using rewrap_temporary_t = std::conditional_t<
    std::is_rvalue_reference_v<Result> && (tc::is_temporary<Args> || ...),
    tc::temporary<std::remove_reference_t<Result>>,
    Result
>;

#define tc_rewrap_temporary(Args, ...) \
    static_cast<tc::rewrap_temporary_t<decltype((__VA_ARGS__)), Args>( \
        __VA_ARGS__ \
    )
```

If the result is an rvalue reference and any of the arguments is a `tc::temporary`, wrap it in a `tc::temporary`.

This is a heuristic.

# Unwrapping and rewrapping temporaries

```
template <typename T>
decltype(auto) get_foo(T&& obj)
{
    return tc_rewrap_temporary(T&&,
                               tc_unwrap_temporary(std::forward<T>(obj))).foo
};
}
```

**This approach works.**

### This approach works.

- Composed projections use `tc_invoke` and `tc_return_temporary` → never returns a dangling reference.
- Tracking of `tc::temporary` is accurate → no unnecessary decay.

### This approach works.

- Composed projections use `tc_invoke` and `tc_return_temporary` → never returns a dangling reference.
- Tracking of `tc::temporary` is accurate → no unnecessary decay.

**But: all templated projections need to be aware of `tc::temporary`.**

## Level 4: Generic compositions

**Insight:** `tc::temporary` is only relevant for projections that can return an rvalue reference.

## Level 4: Generic compositions

**Insight:** `tc::temporary` is only relevant for projections that can return an rvalue reference.

```
template <typename F, typename ... Args>
decltype(auto) invoke_impl(F&& f, Args&&... args) {
    if constexpr (std::is_rvalue_reference_v<decltype(
        std::forward<F>(f)(
            tc_unwrap_temporary(std::forward<Args>(args))...
        )
    )>)
        return std::forward<F>(f)(std::forward<Args>(args)...);
    else
        return std::forward<F>(f)(
            tc_unwrap_temporary(std::forward<Args>(args))...
        );
}
```





## Level 4: Generic compositions

```
template <typename T, typename U>
decltype(auto) operator()(T&& t, U&& u) const
{
    tc_return_temporary(
        tc_invoke(f,
            tc_invoke(g, std::forward<T>(t)),
            tc_invoke(h, std::forward<U>(u))
        )
    );
}
```

## Level 4: Generic compositions

```
template <typename T, typename U>
decltype(auto) operator()(T&& t, U&& u) const
{
    tc_return_temporary(
        tc_invoke(f,
            tc_invoke(g, std::forward<T>(t)),
            tc_invoke(h, std::forward<U>(u))
        )
    );
}
```

What if t or u is already a tc::temporary?

# Temporary with lifetime

```
template <typename T, unsigned Lifetime>  
class temporary { ... };
```

- Lifetime == 0: true temporary, need to decay in tc\_return\_temporary
- Lifetime > 0: only a temporary in some parent scope
- tc::common\_reference\_t, tc\_rewrap\_temporary: minimum lifetime of all temporaries

# Temporary with lifetime

```
template <typename T, unsigned Lifetime>  
class temporary { ... };
```

- Lifetime == 0: true temporary, need to decay in tc\_return\_temporary
- Lifetime > 0: only a temporary in some parent scope
- tc::common\_reference\_t, tc\_rewrap\_temporary: minimum lifetime of all temporaries

```
#define tc_increment_lifetime(...) ...  
#define tc_decrement_lifetime(...) ...
```

# Temporary with lifetime

```
template <typename F, typename ... Args>
decltype(auto) invoke_impl(F&& f, Args&&... args) {
    if constexpr (std::is_rvalue_reference_v<decltype(
        std::forward<F>(f)(
            tc_unwrap_temporary(std::forward<Args>(args))...
        )
    )>)
        return tc_decrement_lifetime(std::forward<F>(f)(
            tc_increment_lifetime(std::forward<Args>(args))...
        ));
    else
        return std::forward<F>(f)(
            tc_unwrap_temporary(std::forward<Args>(args))...
        );
}
```

- `tc::temporary<const T, Lifetime>`

# Other temporary complexities

- `tc::temporary<const T, Lifetime>`
- Reference collapsing rules:

# Other temporary complexities

- `tc::temporary<const T, Lifetime>`
- Reference collapsing rules:
  - `tc::temporary<T, Lifetime>& → T&`



# Other temporary complexities

- `tc::temporary<const T, Lifetime>`
- Reference collapsing rules:
  - `tc::temporary<T, Lifetime>& → T&`
  - `tc::temporary<T, Lifetime>&& → tc::temporary<T, Lifetime>`

## Level 4: Generic compositions

`tc::temporary` is being used in production:

- It is only necessary for function objects in generic code.

`tc::temporary` is being used in production:

- It is only necessary for function objects in generic code.
- Composed projections handle temporaries appropriately.

`tc::temporary` is being used in production:

- It is only necessary for function objects in generic code.
- Composed projections handle temporaries appropriately.
- Generic library projections handle temporaries appropriately.

`tc::temporary` is being used in production:

- It is only necessary for function objects in generic code.
- Composed projections handle temporaries appropriately.
- Generic library projections handle temporaries appropriately.
- There isn't a single projection in user code that needs to handle temporary

## Level 4: Generic compositions

Most common projection:

```
#define tc_member(...) \  
    [<typename T>(T&& obj) -> decltype(auto) { \  
        return tc_rewrap_temporary(T&&, \  
            tc_unwrap_temporary(std::forward<T>(obj)) __VA_ARGS__ \  
        ); \  
    }  
}
```

```
tc::transform(rng, tc_member(.foo));
```

## Level 4: Generic compositions

Most common projection:

```
#define tc_member(...) \  
    [<typename T>(T&& obj) -> decltype(auto) { \  
        return tc_rewrap_temporary(T&&, \  
            tc_unwrap_temporary(std::forward<T>(obj)) __VA_ARGS__ \  
        ); \  
    }  
}
```

```
tc::transform(rng, tc_member(.foo));
```

Similar: `tc_mem_fn(.size)`.

## Level 5: New language design



## Projections:

- No unnecessary copies.
- No unnecessary moves.

## Composed Projections:

- Don't return reference to intermediate temporary.

## Projections:

- No unnecessary copies.
- No unnecessary moves.

## Composed Projections:

- Don't return reference to intermediate temporary.

**Problem:** How to track whether a resulting reference is tied to an argument?

# Rust approach: Built-in lifetime annotations

Mark the lifetime of the output reference (if it is one).

```
template <typename T, typename U>
auto max(T&&/l a, U&&/l b)
    -> tc::common_reference_t<T, U>/l
{
    return a < b ? std::forward<U>(b) : std::forward<T>(a);
}
```

# Rust approach: Built-in lifetime annotations

Mark the lifetime of the output reference (if it is one).

```
template <typename T, typename U>
auto max(T&&/l a, U&&/l b)
    -> tc::common_reference_t<T, U>/l
{
    return a < b ? std::forward<U>(b) : std::forward<T>(a);
}
```

Automatically decay references to dangling temporary in return.

```
template <typename T, typename U>
decltype(auto) operator()(T&&/l a, U&&/l b) const
{
    return f(g(std::forward<T>(t)), h(std::forward<U>(u)));
}
```



**This is still unnecessary complexity!**

**Why does Rust require lifetime annotations in the signature?**

### Why does Rust require lifetime annotations in the signature?

To ensure that you don't change the signature accidentally during internal refactorings.

## Why does Rust require lifetime annotations in the signature?

To ensure that you don't change the signature accidentally during internal refactorings.

**But:** For a projection, the lifetime follows from the behavior!



**Why do C++/Rust require explicit opt-in to call by reference?**

### Why do C++/Rust require explicit opt-in to call by reference?

Because it has consequences for lifetime tracking.

## Why do C++/Rust require explicit opt-in to call by reference?

Because it has consequences for lifetime tracking.

**But:** For a projection, there is an obvious choice.

# Better approach: Just write the projection

Specify that it is a projection, and the compiler does the right thing.

```
template <typename T, typename U>  
projection auto max(T a, U b)  
    -> tc::common_type_t<T, U>  
{  
    return a < b ? b : a;  
}
```

# Better approach: Just write the projection

Specify that it is a projection, and the compiler does the right thing.

```
template <typename T, typename U>
projection auto max(T a, U b)
    -> tc::common_type_t<T, U>
{
    return a < b ? b : a;
}
```

```
template <typename T, typename U>
projection auto operator()(T a, U b) const
{
    return f(g(a), h(b));
}
```

# First-class references are the worst feature C++ has ever added.

- Complexity in function arguments.

# First-class references are the worst feature C++ has ever added.

- Complexity in function arguments.
- T& vs. T&& vs. auto&&.

# First-class references are the worst feature C++ has ever added.

- Complexity in function arguments.
- T& vs. T&& vs. auto&&.
- Temporary lifetime extension.



# First-class references are the worst feature C++ has ever added.

- Complexity in function arguments.
- T& vs. T&& vs. auto&&.
- Temporary lifetime extension.
- Dangling references.

# First-class references are the worst feature C++ has ever added.

- Complexity in function arguments.
- T& vs. T&& vs. auto&&.
- Temporary lifetime extension.
- Dangling references.
- So much complexity for every language feature that interacts with them.

### Parameter passing modes are much better.

- in vs. out vs. inout vs. regular parameters.
- Compiler figures out how to pass stuff.

**Hylø:** References aren't a thing.

- Value semantics only.
- Pass by reference as parameter passing optimization.
- Subscripts implement “projections”.

**Hylø:** References aren't a thing.

- Value semantics only.
- Pass by reference as parameter passing optimization.
- Subscripts implement “projections”.

**This is a much cleaner design!**

## Bonus: Bells and whistles

## Bonus: Bells and whistles

```
template <typename T, typename U>
auto max(T&& a, U&& b) -> tc::common_reference_t<T, U> {
    return tc_conditional_rvalue_as_ref(
        a < b,
        std::forward<U>(b), std::forward<T>(a)
    );
}
```

# Custom predicate

```
template <typename Better, typename T, typename U>
    // Requires: Better implements a strict weak ordering.
    auto best(Better&& better, T&& a, U&& b) -> tc::common_reference_t<T, U> {
        return tc_conditional_rvalue_as_ref(
            better(tc::as_const(b), tc::as_const(a)),
            std::forward<U>(b), std::forward<T>(a)
        );
    }
}
```



# Custom predicate

```
template <typename Better, typename T, typename U>
    // Requires: Better implements a strict weak ordering.
    auto best(Better&& better, T&& a, U&& b) -> tc::common_reference_t<T, U> {
        return tc_conditional_rvalue_as_ref(
            better(tc::as_const(b), tc::as_const(a)),
            std::forward<U>(b), std::forward<T>(a)
        );
    }
```

```
template <typename T, typename U>
    auto max(T&& a, U&& b) -> tc::common_reference_t<T, U> {
        return best(std::greater<>{}, std::forward<T>(a), std::forward<U>(b));
    }
```

# Custom predicate

```
template <typename Better, typename T, typename U>
    // Requires: Better implements a strict weak ordering.
    auto best(Better&& better, T&& a, U&& b) -> tc::common_reference_t<T, U> {
        return tc_conditional_rvalue_as_ref(
            better(tc::as_const(b), tc::as_const(a)),
            std::forward<U>(b), std::forward<T>(a)
        );
    }
```

```
template <typename T, typename U>
    auto max(T&& a, U&& b) -> tc::common_reference_t<T, U> {
        return best(std::greater<>{}, std::forward<T>(a), std::forward<U>(b));
    }
```

```
template <typename T, typename U>
    auto min(T&& a, U&& b) -> tc::common_reference_t<T, U> {
        return best(std::less<>{}, std::forward<T>(a), std::forward<U>(b));
    }
```

# Compile-time predicate

```
template <typename Better, typename T, typename U>
    // Requires: Better implements a strict weak ordering.
decltype(auto) best(Better&& better, T&& a, U&& b) {
    auto result = better(tc::as_const(b), tc::as_const(a));
    if constexpr (std::same_as<decltype(result), std::true_type>) {
        return std::forward<U>(b);
    } else if constexpr (std::same_as<decltype(result), std::false_type>) {
        return std::forward<T>(a);
    } else {
        return tc_conditional_rvalue_as_ref(
            result, std::forward<U>(b), std::forward<T>(a)
        );
    }
}
```

# Compile-time predicate

```
#define tc_conditional_rvalue_as_ref(b, lhs, rhs) \  
  [&]() -> decltype(auto) {  
    if constexpr (std::same_as<decltype(b), std::true_type>)  
      return lhs;  
    else if constexpr (std::same_as<decltype(b), std::false_type>)  
      return rhs;  
    else  
      return ...;  
  }()
```

# Compile-time predicate

```
#define tc_conditional_rvalue_as_ref(b, lhs, rhs) \  
    [&]() -> decltype(auto) {  
        if constexpr (std::same_as<decltype(b), std::true_type>)  
            return lhs;  
        else if constexpr (std::same_as<decltype(b), std::false_type>)  
            return rhs;  
        else  
            return ...;  
    }()
```

But: intermediate stack frame.

# Compile-time predicate

```
#define tc_conditional_rvalue_as_ref(b, lhs, rhs) \  
  [&]() -> decltype(auto) {  
    if constexpr (std::same_as<decltype(b), std::true_type>)  
      return lhs;  
    else if constexpr (std::same_as<decltype(b), std::false_type>)  
      return rhs;  
    else  
      return ...;  
  }()
```

But: intermediate stack frame.

**Wish list:** constexpr ?: or do expressions

# Compile-time predicate

```
auto biggest_size = best(  
    [](const auto& lhs, const auto& rhs) {  
        return std::bool_constant<sizeof(lhs) > sizeof(rhs)>{};  
    },  
    3.14, "hello world"  
);
```

```
template <typename Better, typename T, typename U, typename ... Tail>
    // Requires: Better implements a strict weak ordering.
decltype(auto) best(Better&& better, T&& a, U&& b, Tail&&... tail) {
    auto result = better(tc::as_const(b), tc::as_const(a));
    if constexpr (std::same_as<decltype(result), std::true_type>) {
        return best(better, std::forward<U>(b), std::forward<Tail>(tail)...);
    } else if constexpr (std::same_as<decltype(result), std::false_type>) {
        return best(better, std::forward<T>(a), std::forward<Tail>(tail)...);
    } else {
        return tc_conditional_rvalue_as_ref(result,
            best(better, std::forward<U>(b), std::forward<Tail>(tail)...),
            best(better, std::forward<T>(a), std::forward<Tail>(tail)...));
    }
}
```





# Conclusion

# Is this really necessary?

If you want to avoid unnecessary moves yes, you need `tc::temporary` like tracking

If you are fine with unnecessary moves no, always decay rvalues

# Is this really necessary?

If you want to avoid unnecessary moves yes, you need `tc::temporary` like tracking

If you are fine with unnecessary moves no, always decay rvalues

But really: We need different language design.

[github.com/think-cell/think-cell-library](https://github.com/think-cell/think-cell-library)

**We're hiring:** [think-cell.com/en/career/dev](https://think-cell.com/en/career/dev)

**Developer blog:** [think-cell.com/en/career/devblog/overview](https://think-cell.com/en/career/devblog/overview)

[jonathanmueller.dev/talk/overengineering-max](https://jonathanmueller.dev/talk/overengineering-max)

[@foonathan@fosstodon.org](mailto:@foonathan@fosstodon.org)

[youtube.com/@foonathan](https://youtube.com/@foonathan)