Symantec by Broadcom

operator T& on indirect<T>

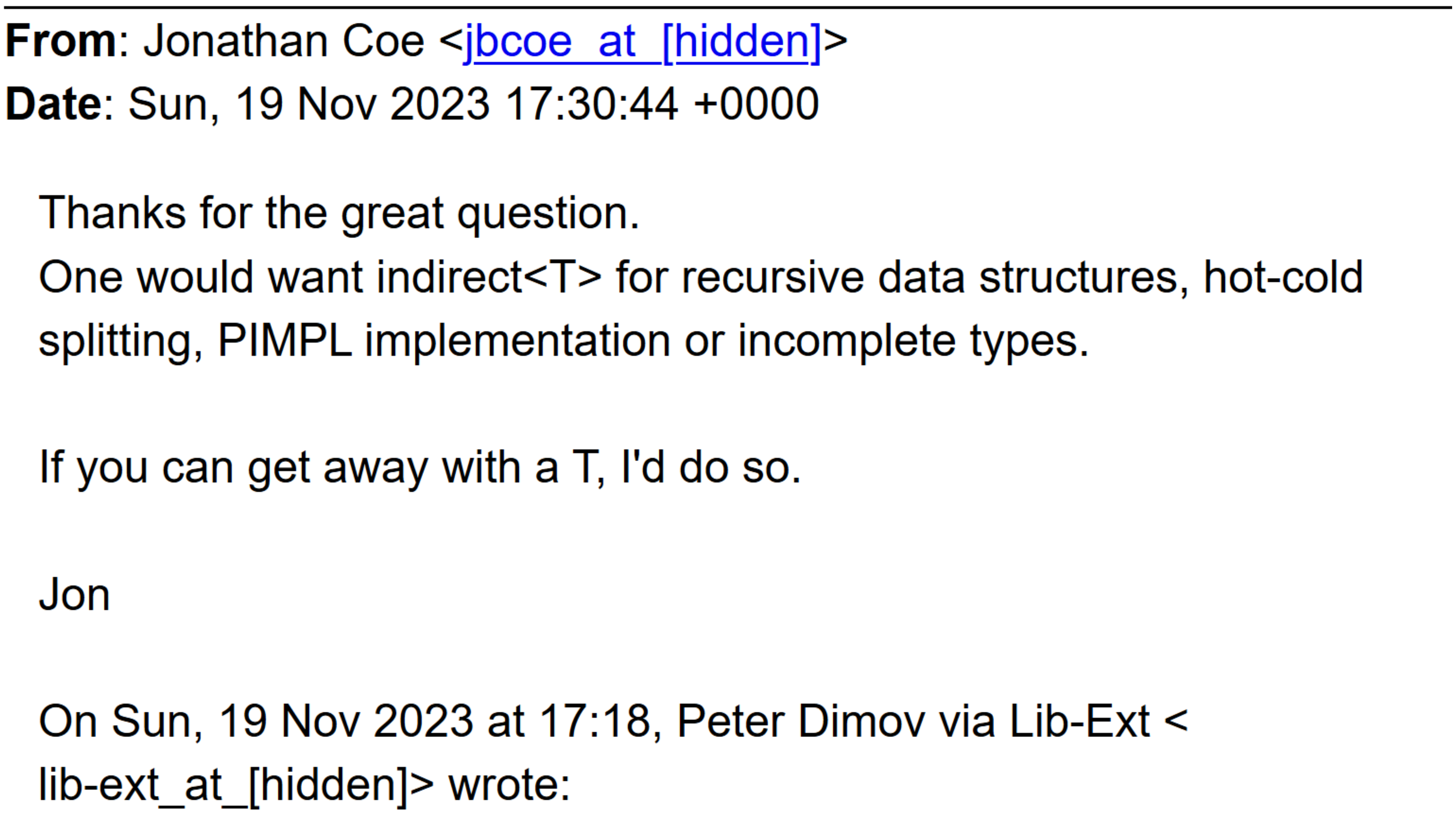
Zhihao Yuan, 2025-11

Content

- Why?
- What is it?
- Is it safe?

Why?

Q: In what scenarios would one use `indirect<T>` instead of just `T`?



From: Jonathan Coe <jbcoe_at_[hidden]>
Date: Sun, 19 Nov 2023 17:30:44 +0000

Thanks for the great question.
One would want indirect<T> for recursive data structures, hot-cold splitting, PIMPL implementation or incomplete types.

If you can get away with a T, I'd do so.

Jon

On Sun, 19 Nov 2023 at 17:18, Peter Dimov via Lib-Ext <
lib-ext_at_[hidden]> wrote:

Author’s example 1

```
struct Number {
};


struct BinOp;


struct Expression {
  std::variant<Number, std::indirect<BinOp>> info;
};


struct BinOp {
};
```

.m

->m

Author’s example 2

```
struct Number {
};


struct BinOp;


struct Expression {
  std::variant<Number, ext::deref<std::indirect<BinOp>>> info;
};


struct BinOp {
};
```

ext::deref<...>(...) …?

What can be done

```
struct Number {
};

struct BinOp;

struct Expression {
  my::variant<Number, my::rec(BinOp)> info;
};

struct BinOp {
};
```

Difference

- not exposing indirect<T> interface in the data structure
- both alternatives accessing T& without noticing a difference

```
template <typename E>
constexpr auto get() & -> E&
{
    constexpr int i = detail::find_alternative_v<E, variant>;
    if (i != rep_.index)
        throw bad_variant_access{};


    return rep_.data.rget(detail::index_c<i>);
}
```

When indirect<T> converts to T&

- Initialization is unaware of `indirect<T>`
- Access is unaware of `indirect<T>`
- Injecting `indirect<T>` into type selection, done

What is it?

Omg, an implicit conversion operator!

And it converts to references!

Background

- In C++, the type of an expression is never a reference type (see **[expr.type]**)
- Expression has a type `T` and a value category
- `decltype(`*`expr`*`)` adjusts `T` based on value category

putting materialization aside,

Every expression has an `operator T&|&&()` and it is called for evaluation every time

- `U::operator T&()` reroutes overload resolution when it is necessary to evaluate the object of type `T`
- `std::reference_wrapper<T>::operator T&()` is an example

Is it safe?

Symantec. by Broadcom

Quote from P3902R2

- `reference_wrapper` is non-owning and has no null or valueless state.

So that its `operator T&()` has no precondition,

But `indirect` has a valueless state, so it must not have an `operator T&()` with a precondition?

Valueless state?

- Start with a container of `vector<unique_ptr<int>>`
- Applied an algorithm to shrink its range
  - everything outside this range is potentially moved-out
- Can a user access the original range “by accident” and tell whether they were wrong?
- What about `vector<optional<int>>` ?

Cont.

- What about `vector<indirect<T>>` ?

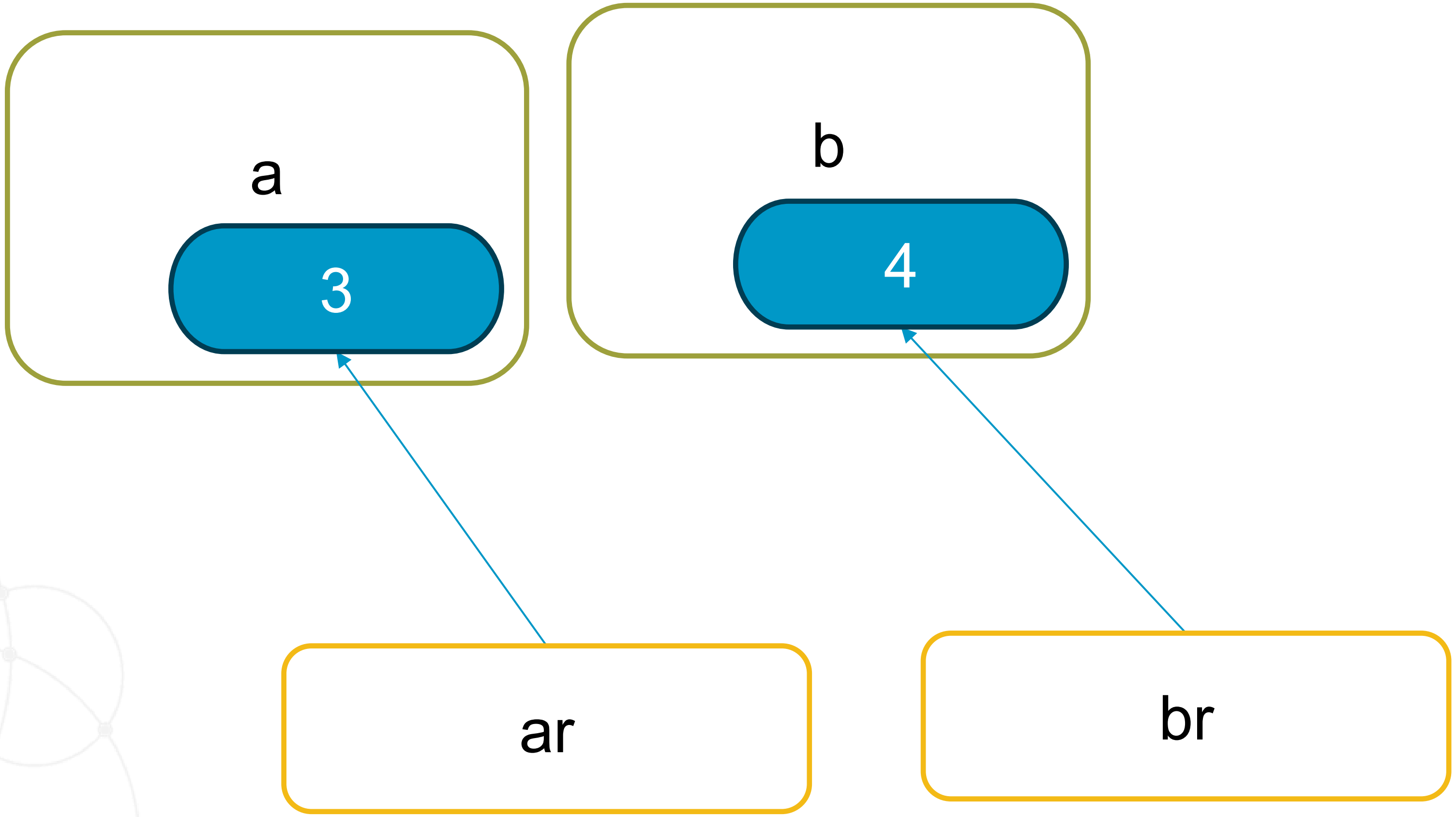
- There is another `vector<reference_wrapper<T>>` tracking the container above
- Each `reference_wrapper<T>` bound to the corresponding element before the algorithm started. When the algorithm permutes, it also moves the tracker `reference_wrapper<T>` around.

Is reference_wrapper<T>::operator T& safe?

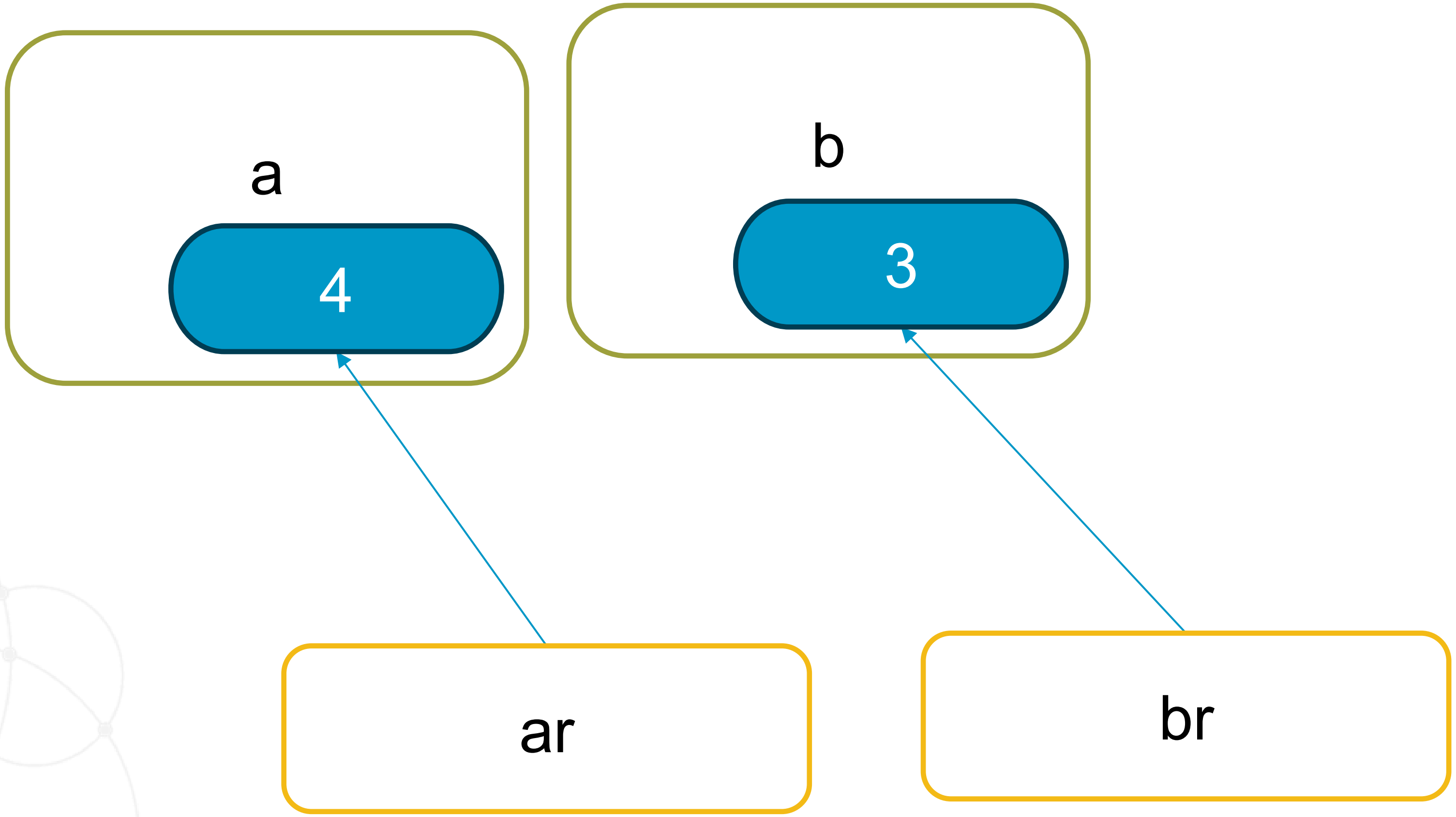
```
auto a = std::indirect(3);
auto b = std::indirect(4);

auto ar = std::ref(*a);
auto br = std::ref(*b);
```

```
assert(ar == 3);
assert(br == 4);

swap(a, b);
swap(ar, br);

assert(ar == 4);
assert(br == 3);

a = std::move(b);
ar = std::move(br);

assert(ar == 3);
// br == ?
```

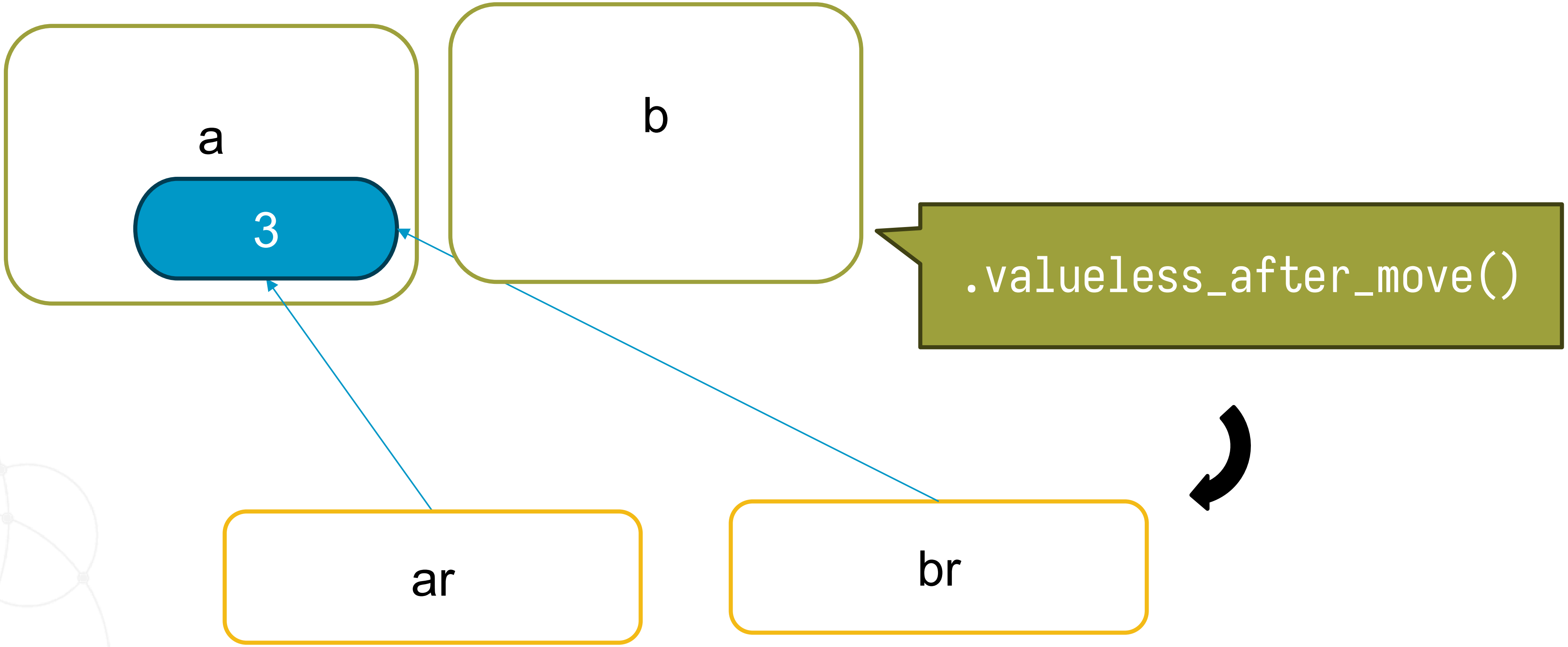
Initial



After swap x 2



After move x 2



- `reference_wrapper` has a valueless state, just not represented
- The fact that `reference_wrapper<T>::operator T&()` has no precondition does not make it safe

Imagine this world

- The `int` variable *x* that holds `42` has a narrow contract on evaluation
- The contract is violated if you “move out” the value `42` and then bind it to `int&`

Isn’t that just an `operator int&()`, with a precondition, on `int`?

- But wait, what if all I want to do is to assign a new value with *x* `= 7`?
- It won’t a problem if that *x* is in fact `indirect<int>`
  - because that expression calls `indirect<int>::operator=(int&&)`

Summary

- The motivation for `indirect<T>::operator T&()` is to support incomplete type `T` as a drop-in replacement for `T`
- `indirect<T>` additionally represents a deterministic valueless state for `T`
- The operator attributes the evaluation for lvalue or rvalue of type `T` a precondition that observes this state

It’s the unambiguously represented states that make a type safe

Symantec by Broadcom

Thank You