Can’t we just synthesize std::tuple_element from get?

Jonathan Müller — `@foonathan`

Adding structured bindings support to a custom tuple-like type

```
template <typename ... T>
struct my_tuple { … };
```

Adding structured bindings support to a custom tuple-like type

```
template <typename ... T>
struct my_tuple { … };
```

```
template <std::size_t I, typename Tuple>
    requires instance_of<my_tuple, std::remove_cvref_t<Tuple>>
auto get(Tuple&& t) -> decltype(auto) { … }
```

Adding structured bindings support to a custom tuple-like type

```
template <typename ... T>
struct my_tuple { … };
```

```
template <std::size_t I, typename Tuple>
    requires instance_of<my_tuple, std::remove_cvref_t<Tuple>>
auto get(Tuple&& t) -> decltype(auto) { … }
```

```
template <typename ... T>
struct std::tuple_size<my_tuple<T...>> {
    static constexpr std::size_t value = sizeof...(T);
};
```

Adding structured bindings support to a custom tuple-like type

```
template <typename ... T>
struct my_tuple { … };
```

```
template <std::size_t I, typename Tuple>
    requires instance_of<my_tuple, std::remove_cvref_t<Tuple>>
auto get(Tuple&& t) -> decltype(auto) { … }
```

```
template <typename ... T>
struct std::tuple_size<my_tuple<T...>> {
    static constexpr std::size_t value = sizeof...(T);
};
```

```
template <std::size_t I, typename ... T>
struct std::tuple_element<I, my_tuple<T...>> {
    using type = /* Ith type of T... */;
};
```

Why do we need to specialize `std::tuple_element`?

Can't we just synthesize `std::tuple_element` from `get`?

Attempt 1: Just `decltype`

```
template <std::size_t I, typename Tuple>
struct my_tuple_element {
    using type = decltype(get<I>(std::declval<Tuple>()));
};
```

think-cell

Problem 1: `std::tuple_element` does not respect value category

```
std::tuple<int> tpl;
```

Problem 1: `std::tuple_element` does not respect value category

```
std::tuple<int> tpl;
```

```
static_assert(std::same_as<decltype(get<0>(tpl)), int&>);
static_assert(std::same_as<decltype(get<0>(std::move(tpl))), int&&>);
```

think-cell

Problem 1: `std::tuple_element` does not respect value category

```
std::tuple<int> tpl;
```

```
static_assert(std::same_as<decltype(get<0>(tpl)), int&>);
static_assert(std::same_as<decltype(get<0>(std::move(tpl))), int&&>);
```

```
static_assert(std::same_as<
  std::tuple_element_t<0, decltype(tpl)>,
  int
>);
```

think-cell

Attempt 2: Remove the reference-ness

```
template <std::size_t I, typename Tuple>
struct my_tuple_element {
    using type = std::remove_reference_t<decltype(
        get<I>(std::declval<Tuple>())
    )>;
};
```

think-cell

Problem 2: Tuple of references

```
std::tuple<int&> tpl;
```

think-cell

Problem 2: Tuple of references

```
std::tuple<int&> tpl;
```

```
static_assert(std::same_as<decltype(get<0>(tpl)), int&>);
```

think-cell

Problem 2: Tuple of references

```
std::tuple<int&> tpl;
```

```
static_assert(std::same_as<decltype(get<0>(tpl)), int&>);
```

```
static_assert(std::same_as<std::tuple_element_t<0, decltype(tpl)>, int&>);
```

Attempt 3: Detect references

```
template <std::size_t I, typename Tuple>
struct my_tuple_element {
    using type_lvalue = decltype(get<I>(std::declval<Tuple&>()));
    using type_rvalue = decltype(get<I>(std::declval<Tuple&&>()));

    static constexpr auto stores_reference
      = std::same_as<type_lvalue, type_rvalue>;

    using type = std::conditional_t<
        stores_reference,
            type_lvalue,
            std::remove_reference_t<type_lvalue>
    >;
};
```

Problem 3: Tuple of rvalue references

```
std::tuple<int&&> tpl;
```

Problem 3: Tuple of rvalue references

```
std::tuple<int&&> tpl;
```

```
static_assert(std::same_as<decltype(get<0>(tpl)), int&>); // !!!
static_assert(std::same_as<decltype(get<0>(std::move(tpl))), int&&>);
```

Problem 3: Tuple of rvalue references

```
std::tuple<int&&> tpl;
```

```
static_assert(std::same_as<decltype(get<0>(tpl)), int&>); // !!!
static_assert(std::same_as<decltype(get<0>(std::move(tpl))), int&&>);
```

```
static_assert(std::same_as<
    std::tuple_element_t<0, decltype(tpl)>,
    int&&
>);
```

think-cell

Attempt 4: Detect references, differently

```
template <std::size_t I, typename Tuple>
struct my_tuple_element {
    using type_mut = decltype(get<I>(std::declval<Tuple&&>()));
    using type_const = decltype(get<I>(std::declval<const Tuple&&>()));

    static constexpr auto stores_reference
        = std::same_as<type_mut, type_const>;

    using type = std::conditional_t<
        stores_reference,
            type_mut,
            std::remove_reference_t<type_mut>
    >;
};
```

Problem 4: Tuple transform

```
tc::transform(tpl, f)
```

- `my_tuple_element` instantiates `f` twice
- `f` might not work on `const`
- `f` might react to rvalue-ness and do an in-place transformation

think-cell

Why do we need to specialize `std::tuple_element`?

Why do we actually need `std::tuple_element`?

Why do we actually need `std::tuple_element`?

"What is the Ith type of a tuple?" has two answers:

- one that respects the value category and const-ness of the tuple object
- one that ignores the value category and const-ness of the tuple object

std::tuple_element and const

```
static_assert(std::same_as<
    std::tuple_element_t<0, std::tuple<int>>,
    int
>);

static_assert(std::same_as<
    std::tuple_element_t<0, const std::tuple<int>>,
    const int
>);
```

Why do we need to specialize `std::tuple_element`?

You should never use `std::tuple_element`!

- You always care about value category and const-ness of the result.
- In general, there is no satisfying answer that ignores the value category and const-ness of the tuple object.

Why do we need to specialize `std::tuple_element`?

You should never use `std::tuple_element`!

- You always care about value category and const-ness of the result.
- In general, there is no satisfying answer that ignores the value category and const-ness of the tuple object.

Even structured bindings don't really want `std::tuple_element`:

cppreference:

> *For each identifier, a variable whose type is "reference to std::tuple_element<i, E>::type" is introduced: lvalue reference if its corresponding initializer is an lvalue, rvalue reference otherwise.*

think-cell

Why do we need to specialize `std::tuple_element`?

You should never use `std::tuple_element`!

- You always care about value category and const-ness of the result.
- In general, there is no satisfying answer that ignores the value category and const-ness of the tuple object.

Even structured bindings don't really want `std::tuple_element`:

cppreference:

> *For each identifier, a variable whose type is “reference to std::tuple_element<i, E>::type” is introduced: lvalue reference if its corresponding initializer is an lvalue, rvalue reference otherwise.*

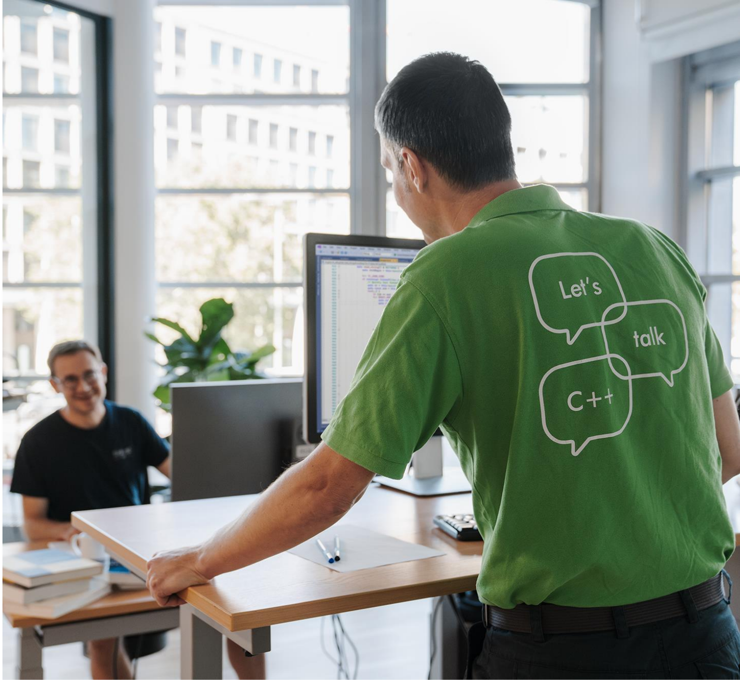
That is `decltype(get<i>(e))`!

think-cell

think-cell

Join our team as a C++ Developer or Intern

think-cell.com/career



What did I do?

What did I do?

```cpp
template <std::size_t I, typename Tuple>
struct my_tuple_element {
    using type_temporary = decltype(
        get<I>(std::declval<tc::temporary<Tuple>>())
    );
    static constexpr auto stores_reference
      = !tc::is_temporary<type_temporary>;
    using type = std::conditional_t<
        stores_reference,
            type_temporary,
            tc::remove_temporary_t<type_lvalue>
    >;
};
```

think-cell

What did I do?

```
template <std::size_t I, typename Tuple>
struct my_tuple_element {
    using type_temporary = decltype(
        get<I>(std::declval<tc::temporary<Tuple>>())
    );
    static constexpr auto stores_reference
      = !tc::is_temporary<type_temporary>;
    using type = std::conditional_t<
        stores_reference,
            type_temporary,
            tc::remove_temporary_t<type_lvalue>
    >;
};
```

Overengineering max(a, b): Mixed comparison functions, common references, and Rust's lifetime annotations

think-cell

Join our team as a
C++ Developer
or Intern

think-cell.com/career