Abstract

We propose to make tuples of 2 elements and pairs comparable. We extend construction, comparison, and assignment between tuple and any object following the tuple protocol, and generalize $\text{tuple\_cat}$.

Tony tables

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>constexpr std::pair p {1, 3.0};</td>
<td>constexpr std::pair p {1, 3.0};</td>
</tr>
<tr>
<td>constexpr std::tuple t {1.0, 3};</td>
<td>constexpr std::tuple t {1.0, 3};</td>
</tr>
<tr>
<td>static_assert(std::tuple(p) == t);</td>
<td>static_assert(p == t);</td>
</tr>
<tr>
<td>static_assert(std::tuple(p) &lt;=&gt; t == 0);</td>
<td>static_assert(p &lt;=&gt; t == 0);</td>
</tr>
</tbody>
</table>

Revisions

R1

- The wording in R0 was non-sensical
- Add a note on deduction guide
- Modify $\text{tuple\_cat}$ to unconditionally support tuple-like entities

Motivation

Pairs are platonic tuples of 2 elements. $\text{pair}$ and $\text{tuple}$ share most of their interface.

Notably, a tuple can be constructed and assigned from a pair. However, $\text{tuple}$ and $\text{pair}$ are not comparable. This proposal fixes that.
This makes tuple more consistent (assignment and comparison usually form a pair, at least in regular-ish types), and makes the library ever so slightly less surprising.

Following that reasoning, we can extend support for these operations to any tuple-like object, aka objects following the tuple protocol.

**Design**

We introduce an exposition only concept tuple-like which can then be used in the definition of tuples construction, comparison and assignment operators.

A type satisfies tuple-like if it implements the tuple protocol (std::get, std::tuple_element, std::tuple_size)

That same concept can be used in [ranges] to simplify the specification. pair is not modified as to not introduce dependencies between <pair> and <tuple>.

In comparisons, One of the 2 objects has to be a tuple, this is done so that comparison operators can be made hidden friends, and to avoid enormous overload sets.

We also make tuple_cat support any tuple-like parameter. This is conditionally supported by implementations already (but may be restricted to pair and array, we generalize that).

**Questions For LEWG**

Should tuple-like and pair-like be named concepts (as opposition to exposition only) ?

**CTAD issues**

A previous version of this paper modified the deduction guides to using the tuple-like constructors for tuple-like objects.

But this would change the meaning of tuple {array<int, 2>{}}. The current version does not add or modify deduction guides. As such, tuple {boost::tuple<int, int>[]} is deduced as std::tuple<boost::tuple<int, int>>

This is obviously not ideal, but, it is a pre-existing problem in C++20. tuple pair<int, int> is currently deduced to std::tuple<int, int>, while other tuple-like objects T are deduced as std::tuple<T>, which may be surprising. This is the same problem that all deduction guides involving wrapper types, and may require a more comprehensive fix, for example:

tuple {pair, pair } // ok
tuple {pair} // ill-formed / deprecated
tuple {std::of_value, pair } // tuple<pair<foo, bar>>
tuple {std::of_elems, pair } // tuple<foo, bar>
Future work

Tuple comparison operators are good candidates for hidden friends.

Wording

**Header <tuple> synopsis**

[...]

// ??, tuple creation functions
inline constexpr unspecified ignore;

template<class... TTypes>
constexpr tuple<unwrap_ref_decay_t<TTypes>...> make_tuple(TTypes&&...);

template<class... TTypes>
constexpr tuple<TTypes&&...> forward_as_tuple(TTypes&&...) noexcept;

template<class... TTypes>
constexpr tuple<TTypes&&...> tie(TTypes&&...) noexcept;

template<class... Tuples>
constexpr tuple<CTypes...> tuple_cat(Tuples&&...);

// ??, calling a function with a tuple of arguments
template<class F, class Tuple>
constexpr decltype(auto) apply(F&& f, Tuple&& t);

[...]

template<class T, class... Types>
constexpr const T& get(const tuple<Types...>& t) noexcept;

template<class T, class... Types>
constexpr const T&& get(const tuple<Types...>&& t) noexcept;

template <typename T, std::size_t N> // exposition only
constexpr bool is_tuple_element = requires (T t) {
    typename tuple_element_t<N-1, remove_const_t<T>>;
    { get<N-1>(t) } -> convertible_to<tuple_element_t<N-1, T>&>;
} && is_tuple_element<T, N-1>;

template <typename T>
constexpr bool is_tuple_element<T, 0> = true;

template <typename T>
concept tuple-like // exposition only
template <typename T>
concept pair-like // exposition only
    = tuple-like<T> && std::tuple_size_v<T> == 2;

// [tuple.rel], relational operators
template<class... TTypes, class... UTypes>
    requires tuple-like<T<UTypes...>>
constexpr bool operator==(const tuple<TTypes...>&, const tuple T<UTypes...>&);

// [tuple.traits], allocator-related traits
template<class... Types, class Alloc>
struct uses_allocator<tuple<Types...>, Alloc>;

namespace std {

    template<class... Types>
    class tuple {
    public:

        // ??, tuple construction
        constexpr explicit(see below) tuple();
        constexpr explicit(see below) tuple(const Types&...);
        // only if sizeof...(Types) >= 1
        template<class... UTypes>
        constexpr explicit(see below) tuple(UTypes&&...);
        // only if sizeof...(Types) >= 1
        tuple(const tuple&) = default;
        tuple(tuple&&) = default;

        template<class... UTypes>
            requires tuple-like<T<UTypes...>>
        constexpr explicit(see below) tuple(const tuple T<UTypes...>&&);

        template<class... UTypes>
            requires tuple-like<T<UTypes...>>
        constexpr explicit(see below) tuple(tuple T<UTypes...>&&);

        template<class U1, class U2>
        constexpr explicit(see below)
        tuple(const pair<U1, U2>&); // only if sizeof...(Types) == 2
    }
}
// allocator-extended constructors

```cpp
template<class Alloc>
constexpr explicit(see below)
tuple(allocator_arg_t, const Alloc& a);

template<class Alloc>
constexpr explicit(see below)
tuple(allocator_arg_t, const Alloc& a, const Types&...);

template<class Alloc, class... UTypes>
constexpr explicit(see below)
tuple(allocator_arg_t, const Alloc& a, UTypes&&...);

template<class Alloc>
constexpr tuple(allocator_arg_t, const Alloc& a, const tuple&);

template<class Alloc>
constexpr tuple(allocator_arg_t, const Alloc& a, tuple&&);

template<class Alloc, class... UTypes, typename...>
requires tuple-like<T<UTypes...>>
constexpr explicit(see below)
tuple(allocator_arg_t, const Alloc& a, const tuple T<UTypes...>&);

template<class Alloc, class... UTypes, typename...>
requires tuple-like<T<UTypes...>>
constexpr explicit(see below)
tuple(allocator_arg_t, const Alloc& a, tuple T<UTypes...>&&);

template<class Alloc, class U1, class U2>
constexpr explicit(see below)
tuple(allocator_arg_t, const Alloc& a, const pair<U1, U2>&);

template<class Alloc, class U1, class U2>
constexpr explicit(see below)
tuple(allocator_arg_t, const Alloc& a, pair<U1, U2>&&);

// ??, tuple assignment
constexpr tuple& operator=(const tuple&);
constexpr tuple& operator=(tuple&&) noexcept(see below);

template<class... UTypes, typename...>
requires tuple-like<T<UTypes...>>
constexpr tuple& operator=(const tuple T<UTypes...>&);

template<class... UTypes, typename...>
requires tuple-like<T<UTypes...>>
constexpr tuple& operator=(tuple T<UTypes...>&&);

template<class U1, class U2>
constexpr tuple& operator=(const pair<U1, U2>&);

template<class U1, class U2>
constexpr tuple& operator=(pair<U1, U2>&&);

// only if sizeof...(Types) == 2
```
// ??, tuple swap
cconstexpr void swap(tuple&) noexcept(see below);
};

 Construction [tuple.cnstr]

 template<class... UTypes,  template<typename...> typename T>
 requires tuple-like<T<UTypes...>>
 constexpr explicit(see below) tuple(const tuple T<UTypes...>& u);

 Constraints:
 • sizeof...(Types) equals sizeof...(UTypes) and
 • is_constructible_v<T, const U_i&> is true for all i, and
 • either sizeof...(Types) is not 1, or (when Types... expands to T and UTypes... expands to U) is_convertible_v<const tuple<U>&, T>, is_constructible_v<T, const tuple<U>&>, and is_same_v<T, U> are all false.

 Effects: Initializes each element of *this with the corresponding element of u.

 Remarks: The expression inside explicit is equivalent to:

 !conjunction_v<is_convertible<const UTypes&, Types>...>

 template<class... UTypes,  template<typename...> typename T>
 requires tuple-like<T<UTypes...>>
 constexpr explicit(see below) tuple(tuple T<UTypes...>&& u);

 Constraints:
 • sizeof...(Types) equals sizeof...(UTypes), and
 • is_constructible_v<T, U_i> is true for all i, and
 • either sizeof...(Types) is not 1, or (when Types... expands to T and UTypes... expands to U) is_convertible_v<tuple<U>, T>, is_constructible_v<T, tuple<U>>, and is_same_v<T, U> are all false.

 Effects: For all i, initializes the i th element of *this with std::forward<U_i>(get<i>(u)).

 Remarks: The expression inside explicit is equivalent to:

 !conjunction_v<is_convertible<UTypes, Types>...>

 template<class U1, class U2> constexpr explicit(see below) tuple(const pair<U1, U2>& u);

 Constraints:
 • sizeof...(Types) is 2,
• `is_constructible_v<T_0, const U1&>` is true, and
• `is_constructible_v<T_1, const U2&>` is true.

**Effects:** Initializes the first element with `u.first` and the second element with `u.second`.

The expression inside `explicit` is equivalent to:

```
!is_convertible_v<const U1&, T_0> || !is_convertible_v<const U2&, T_1>
```

template<class U1, class U2> constexpr explicit(see below) tuple(pair<U1, U2>&& u);

**Constraints:**
• `sizeof...(Types)` is 2,
• `is_constructible_v<T_0, U1>` is true, and
• `is_constructible_v<T_1, U2>` is true.

**Effects:** Initializes the first element with `std::forward<U1>(u.first)` and the second element with `std::forward<U2>(u.second)`.

The expression inside `explicit` is equivalent to:

```
!is_convertible_v<U1, T_0> || !is_convertible_v<U2, T_1>
```

template<class Alloc> constexpr explicit(see below) tuple(allocator_arg_t, const Alloc& a);

template<class Alloc> constexpr explicit(see below) tuple(allocator_arg_t, const Alloc& a, const Types&...);

template<class Alloc, class... UTypes> constexpr explicit(see below) tuple(allocator_arg_t, const Alloc& a, UTypes&&...);

template<class Alloc> constexpr tuple(allocator_arg_t, const Alloc& a, const tuple&);

template<class Alloc> constexpr tuple(allocator_arg_t, const Alloc& a, tuple&&);

```template<class Alloc, class U1, class U2> constexpr explicit(see below) tuple(allocator_arg_t, const Alloc& a, const pair<U1, U2>&)
```
template<class Alloc, class U1, class U2>  
constexpr explicit(
  see below)
  tuple(allocator_arg_t, const Alloc& a, pair<U1, U2>&&);

Expects: Alloc meets the Cpp17Allocator requirements ().

Effects: Equivalent to the preceding constructors except that each element is constructed with uses-allocator construction.

Assignment

For each tuple assignment operator, an exception is thrown only if the assignment of one of the types in Types throws an exception. In the function descriptions that follow, let $i$ be in the range $[0, \text{sizeof...(Types)})$ in order, $T_i$ be the $i^{th}$ type in Types, and $U_i$ be the $i^{th}$ type in a template parameter pack named UTypes, where indexing is zero-based.

constexpr tuple& operator=(const tuple& u);

Effects: Assigns each element of $u$ to the corresponding element of *this.
Remarks: This operator is defined as deleted unless is_copy_assignable_v<$T_i>$ is true for all $i$.
Returns: *this.

constexpr tuple& operator=(tuple&& u) noexcept(
  see below);

Constraints: is_move_assignable_v<$T_i>$ is true for all $i$.
Effects: For all $i$, assigns std::forward<$T_i>$(get<$i>$(u)) to get<$i>(*this).
Remarks: The expression inside noexcept is equivalent to the logical AND of the following expressions:
   is_nothrow_move_assignable_v<$T_i>$

where $T_i$ is the $i^{th}$ type in Types.
Returns: *this.

template<class... UTypes, template<typename...> typename T>
requires tuple-like<T<UTypes...>>
constexpr tuple& operator=(const tuple<T<UTypes...>>& u);

Constraints:

* sizeof...(Types) equals sizeof...(UTypes) and
* isAssignable_v<$T_i$, const $U_i$>& is true for all $i$.

Effects: Assigns each element of $u$ to the corresponding element of *this.
Returns: *this.
template<class... UTypes, template<typename...> typename T>
requires tuple-like<T<UTypes...>>
constexpr tuple& operator=(tuple I<UTypes...>&& u);

Constraints:

- sizeof...(Types) equals sizeof...(UTypes) and
- is_assignable_v<Ti&, Uj> is true for all i.

Effects: For all i, assigns std::forward<Uj>(get<i>(u)) to get<i>(*this).
Returns: *this.

template<class U1, class U2> constexpr tuple& operator=(const pair<U1, U2>& u);

Constraints:

- sizeof...(Types) is 2 and
- is_assignable_v<T0&, const U1&> is true, and
- is_assignable_v<T1&, const U2&> is true.

Effects: Assigns u.first to the first element of *this and u.second to the second element of *this.
Returns: *this.

template<class U1, class U2> constexpr tuple& operator=(pair<U1, U2>&& u);

Constraints:

- sizeof...(Types) is 2 and
- is_assignable_v<T0&, U1> is true, and
- is_assignable_v<T1&, U2> is true.

Effects: Assigns std::forward<U1>(u.first) to the first element of *this and
         std::forward<U2>(u.second) to the second element of *this.
Returns: *this.

 Tuple creation functions
[tuple.creation]

template<class... Tuples>
requires (tuple-like<std::remove_reference_t<Tuples>>&...)
constexpr tuple<CTypes...> tuple_cat(Tuples&&... tpls);

In the following paragraphs, let Ti be the i-th type in Tuples, Ui be remove_reference_t<Ti>,
and tp_i be the i-th parameter in the function parameter pack tpls, where all indexing is
zero-based.
Expects: For all $i$, $U_i$ is the type $cv_i$ tuple<Args$_i$...>, where $cv_i$ is the (possibly empty) $i^{th}$ cv-qualifier-seq and Args$_i$ is the template parameter pack representing the element types in $U_i$. Let $A_{ik}$ be the $k^{th}$ type in Args$_i$. For all $A_{ik}$ the following requirements are met:

- If $T_i$ is deduced as an lvalue reference type, then is_constructible_v<$A_{ik}$, $cv_i$ $A_{ik}$&> == true; otherwise
- is_constructible_v<$A_{ik}$, $cv_i$ $A_{ik}$&&> == true.

Remarks: The types in CTYPES are equal to the ordered sequence of the extended types Args$_0$, Args$_1$, ..., Args$_{n-1}$, where $n$ is equal to sizeof...(Tuples). Let $e_i$... be the $i^{th}$ ordered sequence of tuple elements of the resulting tuple object corresponding to the type sequence Args$_i$.

Returns: A tuple object constructed by initializing the $k_{i}$ th type element $e_{ik}$ in $e_i$... with

get<$k_i$>(std::forward<T$_i$>(tp$_i$))

for each valid $k_i$ and each group $e_i$ in order.

[ Note: An implementation may support additional types in the template parameter pack Tuples that support the tuple-like protocol, such as pair and array. — end note ]

Relational operators

[tpl-rel]

template<class... TTypes, class... UTypes, typename T>  
requires tuple-like<T<UTypes...>>
constexpr bool operator==(const tuple<TTypes...>& t, tuple T<UTypes...>& u);

Mandates: For all $i$, where $0 \leq i < $ sizeof...(TTypes), get<i>(t) == get<i>(u) is a valid expression returning a type that is convertible to bool. sizeof...(TTypes) equals sizeof...(UTypes).

Returns: true if get<i>(t) == get<i>(u) for all $i$, otherwise false. For any two zero-length tuples $e$ and $f$, $e == f$ returns true.

Effects: The elementary comparisons are performed in order from the zeroth index upwards. No comparisons or element accesses are performed after the first equality comparison that evaluates to false.

template<class... TTypes, class... UTypes, typename T>  
requires tuple-like<T<UTypes...>>
constexpr common_comparison_category_t<synth-three-way-result<TTypes, UTypes>...> operator<=>(const tuple<TTypes...>& t, const tuple T<UTypes...>& u);

Effects: Performs a lexicographical comparison between $t$ and $u$. For any two zero-length tuples $t$ and $u$, $t <=> u$ returns strong_ordering::equal. Otherwise, equivalent to:

if (auto c = synth-three-way(get<0>(t), get<0>(u)); c != 0) return c;
return t.tail <=> u.tail;
where \( r_{\text{tail}} \) for some tuple \( r \) is a tuple containing all but the first element of \( r \).

[Note: The above definition does not require \( t_{\text{r}} \) (or \( u_{\text{r}} \)) to be constructed. It may not even be possible, as \( t \) and \( u \) are not required to be copy constructible. Also, all comparison functions are short circuited; they do not perform element accesses beyond what is required to determine the result of the comparison. — end note]

**Range utilities**

**Sub-ranges**

The `subrange` class template combines together an iterator and a sentinel into a single object that models the `view` concept. Additionally, it models the `sized_range` concept when the final template parameter is `subrange_kind::sized`.

```cpp
namespace std::ranges {
    template<class From, class To>
    concept convertible-to-non-slicing = // exposition only
    convertible_to<From, To> &&
    !(is_pointer_v<decay_t<From>> &&
    is_pointer_v<decay_t<To>> &&
    not_same_as<remove_pointer_t<decay_t<From>>, remove_pointer_t<decay_t<To>>>);
}
```

```cpp
namespace std::ranges {
    template<class T>
    concept pair-like = // exposition only
    !is_reference_v<T> && requires(T t) {
        typename tuple_size<T>::type; // ensures tuple_size<T> is complete
        requires derived_from<tuple_size<T>, integral_constant<size_t, 2>>;
        typename tuple_element_t<0, remove_const_t<T>>;
        typename tuple_element_t<1, remove_const_t<T>>;
        { get<0>(t) } -> convertible_to<const tuple_element_t<0, T>&>;
        { get<1>(t) } -> convertible_to<const tuple_element_t<1, T>&>;
    };
}
```

```cpp
namespace std::ranges {
    template<class T, class U, class V>
    concept pair-like-convertible-from = // exposition only
    !range<T> && pair-like<T> &&
    constructible_from<T, U, V> &&
    convertible-to-non-slicing<U, tuple_element_t<0, T>>, &&
    convertible-to<V, tuple_element_t<1, T>>;
}
```

**Elements view**

**Class template elements_view**

```cpp
namespace std::ranges {
    template<class T, size_t N>
    concept has-tuple-element = // exposition only
```
tuple-like\(\langle T \rangle\) && tuple_size_v\(\langle T \rangle\) < N;
requires(T t) {
    typename tuple_size<T>::type;
    requires N < tuple_size_v<T>;
    typename tuple_element_t<N, T>;
    ( get<N>(t) ) -> convertible_to<const tuple_element_t<N, T> &>;
}

Acknowledgments

References

https://wg21.link/N4861