Compatibility between `tuple` and tuple-like objects

Document #:	P2165R0
Date:	2020-05-15
Project:	Programming Language C++
Audience:	LEWG
Reply-to:	Corentin Jabot <corentin.jabot@gmail.com>

Abstract

We propose to make tuples of 2 elements and pairs comparable. We extend comparison and assignment between tuple and any object following the tuple protocol.

Tony tables

Before	After

`constexpr std::pair  p {1, 3.0};`
`constexpr std::tuple t {1.0, 3};`
`static_assert(std::tuple(p) == t);`
`static_assert(std::tuple(p) <=> t == 0);`
	`constexpr std::pair  p {1, 3.0};`
	`constexpr std::tuple t {1.0, 3};`
	`static_assert(p == t);`
	`static_assert(p <=> t == 0);`

Motivation

`pair`s are platonic tuples of 2 elements. `pair` and `tuple` share most of their interface.

Notably, a tuple can be constructed and assigned from a pair. However, `tuple` and `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 object following the tuple protocol.

Design

We introduce an exposition only concept `tuple-like` which can then be used in the definition of tuples 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>

Questions For LEWG

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

Future work

Tuple comparison operators are good candidates for hidden friends.

Wording

� Header <tuple> synopsis [tuple.syn]

[...]

```
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
    = !is_reference_v<T> && requires  {
    typename tuple_size<T>::type;
    same_as<decltype(tuple_size_v<T>), size_t>;
} && is_tuple_element<T, tuple_size_v<T>>;

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

constexpr common_comparison_category_t<synth-three-way-result<TTypes, UTypes>...>
operator<=>(const tuple<TTypes...>&, const tuple tuple-like<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>
    constexpr explicit(see below) tuple(const tuple tuple-like<UTypes...>&);
    template<class... UTypes>
    constexpr explicit(see below) tuple(tuple tuple-like<UTypes...>&&);

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


    // allocator-extended constructors
    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>
    constexpr explicit(see below)
    tuple(allocator_arg_t, const Alloc& a, const tuple tuple-like<UTypes...>&);
    template<class Alloc, class... UTypes>
    constexpr explicit(see below)
    tuple(allocator_arg_t, const Alloc& a, tuple tuple-like<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>
    constexpr tuple& operator=(const tuple tuple-like<<UTypes...>&);
    template<class... UTypes>
    constexpr tuple& operator=(tuple tuple-like<<UTypes...>&&);

    template<class U1, class U2>
    constexpr tuple& operator=(const pair<U1, U2>&);
     // only if sizeof...(Types) == 2
    template<class U1, class U2>
    constexpr tuple& operator=(pair<U1, U2>&&);
    // only if sizeof...(Types) == 2


    // ??, tuple swap
    constexpr void swap(tuple&) noexcept(see below);
};

template<class... UTypes>
tuple(UTypes...) -> tuple<UTypes...>;
template<class T1, class T2 class... UTypes>
tuple(pair<T1, T2> tuple-like<UTypes..>) -> tuple<T1, T2 UTypes...>;
template<class Alloc, class... UTypes>
tuple(allocator_arg_t, Alloc, UTypes...) -> tuple<UTypes...>;
template<class Alloc, class T1, class T2>
tuple(allocator_arg_t, Alloc, pair<T1, T2>) -> tuple<T1, T2>;
template<class Alloc, class... UTypes>
tuple(allocator_arg_t, Alloc, tuple tuple-like<UTypes...>) -> tuple<UTypes...>;
}
```

� Construction [tuple.cnstr]

[...]

```
template<class... UTypes>
constexpr explicit(see below) tuple(const ~~tuple~~ tuple-like<UTypes...>& u);
```

Constraints:

- `sizeof...(Types)` equals `sizeof...(UTypes)` and
- `is_constructible_v<`T_i`, 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>
constexpr explicit(see below) tuple(~~tuple~~ tuple-like<UTypes...>&& u);
```

Constraints:

- `sizeof...(Types)` equals `sizeof...(UTypes)`, and
- `is_constructible_v<`T_i`, `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&, T0> || !is_convertible_v<const U2&, T1>
```

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

Constraints:

- `sizeof...(Types)` is 2,
- `is_constructible_v<T0, U1>` is `true`, and
- `is_constructible_v<T1, 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, T0> || !is_convertible_v<U2, T1>
```

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

Expects: `Alloc` meets the *Cpp17Allocator* requirements ().

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

� Assignment [tuple.assign]

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, 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> constexpr tuple& operator=(const` ~~`tuple`~~ *`tuple-like`*`<UTypes...>& u);`

Constraints:

- `sizeof...(Types)` equals `sizeof...(UTypes)` and
- `is_assignable_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> constexpr tuple& operator=(`~~`tuple`~~ *`tuple-like`*`<UTypes...>&& u);`

Constraints:

- `sizeof...(Types)` equals `sizeof...(UTypes)` and
- `is_assignable_v<T`$_i$`&, U`$_i$`>` is `true` for all i.

Effects: For all i, assigns `std::forward<U`$_i$`>(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<`T_0`&, const U1&>` is `true`, and
- `is_assignable_v<`T_1`&, 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<`T_0`&, U1>` is `true`, and
- `is_assignable_v<`T_1`&, 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`.

� Relational operators [tuple.rel]

template<class... TTypes, class... UTypes>
constexpr bool operator==(const tuple<TTypes...>& t, ~~tuple~~ *tuple-like* tuple<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>
constexpr common_comparison_category_t<*synth-three-way-result*<TTypes, UTypes>...>
operator<=>(const tuple<TTypes...>& t, const ~~tuple~~ *tuple-like*<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 $\mathtt{r}_{\text{tail}}$ for some tuple `r` is a tuple containing all but the first element of `r`.

[*Note:* The above definition does not require $\mathtt{t}_{\text{tail}}$ (or $\mathtt{u}_{\text{tail}}$) 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 [range.utility]

� Sub-ranges [range.subrange]

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`.

```
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>>>);

    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>&>;
    };

    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 [range.elements]

� Class template `elements_view` [range.elements.view]

```
namespace std::ranges {
    template<class T, size_t N>
    concept has-tuple-element =                       // exposition only
    tuple-like<T> && tuple_size_v<T> < 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

[N4861] Richard Smith *Working Draft, Standard for Programming Language C++*
https://wg21.link/N4861