Compatibility between tuple, pair and tuple-like objects

Document #: P2165R3
Date: 2022-01-19
Programming Language C++
Audience: LEWG
Reply-to: Corentin Jabot <corentin.jabot@gmail.com>

A tuple by any other name would unpack just as well - Shakespair

Abstract

We propose to make pair constructible from tuple and std::array. We mandate tuple_cat and friends to be compatible with these types, and associative containers more compatible with them. The changes proposed in this paper make the use of std::pair unnecessary in new code.

Revisions

R3

• Reduce the scope to types that have a get method in the std:: namespace. In effect only tuple, pair, array and ranges::subrange are tuple-like with that definition. The intent is to extend to user-provided types later once std::get or equivalent is defined as a customization-point. Limiting to std::get allows to unconditionally use std::tuple as the reference type of zip, cartesian_product in C++23.

• Rebase the wording onto the latest draft, which contains significant changes (const assignment operators).

• Add the wording modification to zip.

• Remove the modification to associative containers, with the intent to add these constructors in a later version of C++.

• Conserve the existing tuple and pair constructors, assignment, and comparison operators. The proposed changes to tuple and pair are now additions exclusively. This is both because implementers cannot remove them because of ABI, and because the proposed constructors are not found for classes that inherit publicly from pair or tuple. This removes some concern for breaking changes.

• Add an overload to uses_allocator_construction_args.

• Add a feature test macro.
• Fix the tuple-like concept to support reference types.

R2
The scope and design have changed quite a bit since R1. First, R1 failed to account for most tuple-like things like `array`. Second, R2 also modifies associative containers to accept tuple-like objects.

R1
• The wording in R0 was non-sensical
• Add a note on deduction guide
• Modify `tuple_cat` to unconditionally support tuple-like entities
## Tony tables

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>constexpr std::pair p {1, 3.0}; constexpr std::tuple t {p}; // OK</td>
<td>constexpr std::pair p {1, 3.0}; constexpr std::tuple t {p}; // OK</td>
</tr>
<tr>
<td>std::pair&lt;int, double&gt; pp (get&lt;0&gt;(t), get&lt;1&gt;(t));</td>
<td>std::pair&lt;int, double&gt; pp(t);</td>
</tr>
<tr>
<td>static_assert(std::tuple(p) == t);</td>
<td>static_assert(std::tuple(p) == t);</td>
</tr>
<tr>
<td>static_assert(p == t); static_assert(p &lt;=&gt; t == 0);</td>
<td>static_assert(p == t); static_assert(p &lt;=&gt; t == 0);</td>
</tr>
<tr>
<td>static_assert(same_as&lt;std::tuple&lt;int&gt;, range_value_t&lt;decltype(views::zip(v))&gt;&gt;);</td>
<td>// not the same size: ill-formed</td>
</tr>
<tr>
<td>static_assert(same_as&lt;std::tuple&lt;int,int&gt;, range_value_t&lt;decltype(views::zip(v, v))&gt;&gt;);</td>
<td>static_assert(same_as&lt;std::tuple&lt;int&gt;, range_value_t&lt;decltype(views::zip(v))&gt;&gt;);</td>
</tr>
<tr>
<td>static_assert(same_as&lt;std::tuple&lt;int, int&gt;, range_value_t&lt;decltype(views::zip(v, v))&gt;&gt;);</td>
<td>static_assert(same_as&lt;std::tuple&lt;int, int&gt;, range_value_t&lt;decltype(views::zip(v, v))&gt;&gt;);</td>
</tr>
<tr>
<td>// x is std::tuple&lt;int, int&gt; // because tuple is convertible from pair auto x = true ? tuple{0,0} : pair{0,0};</td>
<td>// Both types are interconvertible, // The expression is ambiguous an this is ill-formed auto x = true ? tuple{0,0} : pair{0,0};</td>
</tr>
</tbody>
</table>

Red text is ill-formed

## Motivation

Pairs 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, but the reverse is not true. Tuple and pairs cannot be compared.


We are not proposing to get rid of `pair`. However, we are suggesting that maybe new facilities
should use tuple, or when appropriate, a structure with named members. The authors of N2270 [3], circa 2007, observed:

> There is very little reason, other than history, for the library to contain both `pair<T, U>` and `tuple<T, U>`. If we do deprecate `pair`, then we should change all interfaces in the library that use it, including the associative containers, to use `tuple` instead. This will be a source-incompatible change, but it need not be ABI-breaking.

As `pair` will continue to exist, it should still be possible for users of the standard library to ignore its existence, which can be achieved by making sure pairs are constructible from `tuple`, and types that are currently constructible from `pair` can be constructed from another kind of `tuple`.

For example, associative containers deal in pairs, and they do not allow construction from sequences of tuples. This has forced ranges (zip: P2321R1 [6], cartesian_product: P2374R0 [1]) to deal in `pair` when dealing with tuples of 2 elements.

`view_of_tuples | to<map>` currently doesn’t work, and we think it should.

Standard types supporting the tuple protocol include

- `pair`
- `tuple`
- `array`
- `subrange`
- the proposed `enumerate`’s reference type.
- `span` of static extent- prior to P2116R0 [5] which removed that support

### Design

We introduce an exposition only concept `tuple-like` which can then be used in the definition of `tuple` and `pair` construction, comparison and assignment operators. A type satisfies `tuple-like` if it implements the tuple protocol (`std::get`, `std::tuple_element`, `std::tuple_size`). The concept is a generalization of the `pair-like` exposition-only concept used by `subrange` and `views::values/views::keys`.

With that concept, we

- Allow a `tuple` to be constructed, assigned and compared with any standard tuple-like object (of the same size).
- Allow a `pair` to be constructed, assigned and compared with any standard tuple-like object of size 2.
- Can use `tuple` in `zip` and similar views consistently, in the 2 views case.
• Define a common_reference and a common_type between std::tuple and any tuple-like object. This simplifies using a custom tuple-like type in a zip-like view. **This change is only necessary in C++23** if we want to adopt the design of enumerate as proposed by P2164R5 [2].

In comparisons, one of the 2 objects has to be a tuple, or a pair. This is done so that comparison operators can be made hidden friends in order to avoid enormous overload sets. We also make tuple_cat support any tuple-like parameter. This is conditionally supported by implementations already.

std::apply and std::make_from_tuple are similarly constrained. There is currently non stated constraints on these functions but types that do not satisfy tuple-like do not satisfy the implicit requirements of implementations. Constraining them improves diagnostic quality.

Associative containers are already specified to be usable with elements convertible to their value_type, so all constructors and methods can be used with tuple-like, except those taking an initializer_list as parameter. Changes to initializer_list are not proposed in this revision of this paper as it is not strictly necessary to be done in 23. It could be considered as a future extension.

**std::get**

I was initially under the misguided impression that get was designed to be found by unqualified lookup, which it is not. We do not have time to research and specify a CPO forget - which would probably require additional changes to the language (structured binding) and the library. As such, this paper offers the minimal changes necessary to make the standard types inter-compatible, with the express purpose to be able to use tuple in zip and cartesian_product.

[Note: Because implementations should always called std::get qualified, it is not a potentially-breaking change to constrain the previously unconstrained apply, tuple_cat and make_from_tuple functions. — end note]

**CTAD issues**

A previous version of this paper modified the deduction guides to use 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

5
While we could add a non-ambiguous guide for pair, we think it's better for pair and tuple to remain consistent.

**We do not propose modifications to CTAD constructors**

**Open Questions**

**pair const assignment**

zip added const assignment operators and to both tuple and pair for the benefits of views, which use tuple and pair as proxy types. With the present proposal, pair is no longer used by ranges. Do we want to keep its added assignment operators?

**Breaking API changes**

**Ternary operator ambiguities**

Before this paper, tuple was constructible from pair, but the opposite was not true.
As such `expr ? apair : atuple` would resolve unambiguously to a tuple.
Because this changes makes both pair and tuple constructible from each other, the expression is now ambiguous.
This proposal is, therefore, a breaking change. However, it is unlikely that this pattern exists in practice. It can be resolved by casting either expression to the type of the other.
Similar expressions such as `true ? std::tuple{0.} : std::tuple{0}` are ill-formed in C++20 because they are ambiguous.

**The lying tuple also converting to std::tuple**

Consider the following example, courtesy of Tomasz Kamiński and Barry Revzin.

```cpp
struct M {
    operator tuple<int, int>() const { return {1, 1}; }
};

namespace std {
    template <> struct tuple_size<M> : integral_constant<size_t, 2> { }; 
    template <int I> struct tuple_element<I, M> { using type = int; }; 
    template <int I> auto get(M) { return 2; }
}
```

In C++20, `std::tuple<int, int>{M}` would be equal to `std::tuple<int, int>{1, 1};`
With the current proposal, the tuple-like constructor is a better match than the conversion operator, and `std::tuple<int, int>{M}` would be equal to `std::tuple<int, int>{2, 2};`
As the conversion is not called, side effects that this operator might have (ex: logging) - are not executed. Another scenario may be that the conversion operator would return a tuple with different element types than get/tuple_element_t.

And while there exists types that are both convertible to std::tuple and tuple-like - like ranges-v3's compressed_pair, We do not think the case of types that would do different things when the conversion operator is called rather than the proposed tuple-like std::tuple constructor is worth considering

- Checking the presence of an operator tuple<tuple_element_t<Index>...> is costly.
- Checking the presence of an arbitrary conversion operator a tuple can be constructed from is not possible
- We make no promise not to add overloads.
- It is UB for users to add std::get overloads

But, if this was a problem, we could provide an opt-out mechanism such as

```cpp
template<class>
inline constexpr bool disable_tuple-like = false; // user specializable
```

There is precedence for that (ranges::disable_sized_range), but this is not proposed in this paper, as we think the potential for breakage is theoretical.

**Implementation**

This proposal has been fully implemented in libstdc++ [Github], such that all existing tests pass, and new changes have been tested by the author. (The proposed changes to zip have not been made)

[Note: And implementation can and probably should short-circuit the tuple-like concept-checking for known standard tuple-like types. — end note]

**Future work**

This proposal does not explore a way to make std::get a customization point, our main goal being to get rid of tuple-or-pair for 23. This requires future exploration. Tuple comparison operators are good candidates for hidden friends.

**Possible associative containers modifications**

A previous version of this paper proposed to modify {unordered_}{multi}map so that they could be constructed from an initializer_list of tuple-like. These changes were, however, broken and are not necessary for C++23. I, therefore, elected to remove them from the paper. However, associative containers are already specified to be usable with elements convertible to their value_type, so all constructors and methods can be used with tuple-like, except those taking an initializer_list as parameter.
Wording

Feature test macros

[Editor's note: Add a new macro in `<version>_cpp_lib_tuple_like` set to the date of adoption. The macro `__cpp_lib_tuple_like` is also present in `<utility>`, `<tuple>`, `<map>`, `<unordered_map>`]

⚠️ Header `<tuple>` synopsis  

#include <compare> // see ??

namespace std {
    // ??, class template tuple
    template<class... Types>
    class tuple;

    template<class... TTypes, class... UTypes, template<class> class TQual, template<class> class UQual>
    requires requires {
        typename tuple<common_reference_t<TQual<TTypes>, UQual<UTypes>>...>;
    } struct basic_common_reference<tuple<TTypes...>, tuple<UTypes...>, TQual, UQual> {
        using type = tuple<common_reference_t<TQual<TTypes>, UQual<UTypes>>...>;
    };

    template<class... TTypes, class... UTypes>
    requires requires {
        typename tuple<common_type_t<TTypes, UTypes>...>;
    } struct common_type<tuple<TTypes...>, tuple<UTypes...>> {
        using type = tuple<common_type_t<TTypes, UTypes>...>;
    };

    template<tuple-like TTuple, tuple-like UTuple, template<class> class TQual, template<class> class UQual>
    struct basic_common_reference<TTuple, UTuple, TQual, UQual>;

    template<tuple-like TTuple, tuple-like UTuple, template<class> class TQual, template<class> class UQual>
    struct common_type<TTuple, UTuple, TQual, UQual>;

    template <typename T, size_t N>
    concept is-tuple-element = requires (T t) { // exposition only
        typename tuple_element_t<N, T>;
        { std::get<N>(t) } -> convertible_to<tuple_element_t<N, T>&>;
    };

    template <typename T>
    concept tuple-like-impl // exposition only
    = requires {
        typename tuple_size<T>::type;
        requires same_as<remove_cvref_t<decltype(tuple_size_v<T>>), size_t>;
    } && []<size_t... I>(index_sequence<I...>)
    { return (is-tuple-element<T, I> && ...); }(make_index_sequence<tuple_size_v<T>>{});

}
template <typename T>
concepts tuple-like // exposition only
= tuple-like-impl<remove_cvref_t<T>>;

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

// ??, 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... tuple-like... Tuples>
constexpr tuple<Tuples&&...> tuple_cat(Tuples&&...);

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

template<class T, class tuple-like Tuple>
constexpr T make_from_tuple(Tuple&& t);

// ??, tuple helper classes
template<class T> struct tuple_size; // not defined

template<class T>
struct tuple_size<const T>;

template<class... Types>
struct tuple_size<tuple<Types...>>;

template<size_t I, class T>
struct tuple_element; // not defined

template<size_t I, class T>
struct tuple_element<I, const T>;

template<size_t I, class... Types>
struct tuple_element<I, tuple<Types...>>;

template<size_t I, class T>
using tuple_element_t = typename tuple_element<I, T>::type;

// ??, element access

template<size_t I, class... Types>
constexpr tuple_element_t<I, tuple<Types...>>& get(tuple<Types...>&) noexcept;

template<size_t I, class... Types>
constexpr tuple_element_t<I, tuple<Types...>>& get(tuple<Types...>&&) noexcept;

template<size_t I, class... Types>

constexpr const tuple_element_t<I, tuple<Types...>>& get(const tuple<Types...>&) noexcept;
template<size_t I, class... Types>
constexpr const tuple_element_t<I, tuple<Types...>>&& get(const tuple<Types...>&&) noexcept;

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

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;

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

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

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

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

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

// ??, specialized algorithms
template<class... Types>
constexpr void swap(tuple<Types...>& x, tuple<Types...>& y) noexcept(see below);

template<class... Types>
constexpr void swap(const tuple<Types...>& x, const tuple<Types...>& y) noexcept(see below);

// ??, tuple helper classes
template<class T>
inline constexpr size_t tuple_size_v = tuple_size<T>::value;

}
constexpr explicit(see below) tuple();
constexpr explicit(see below) tuple(const Types&...); // only if sizeof...(Types) >= 1
tuple(const tuple&) = default;
tuple(tuple&) = default;

template<class... UTypes>
constexpr explicit(see below) tuple(tuple<UTypes...>&);

template<class... UTypes>
constexpr explicit(see below) tuple(tuple<UTypes...>&&);

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

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

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

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

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

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

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

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

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

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

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

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

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

// allocator-extended constructors

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

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

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

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

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

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

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

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

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

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

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

template<class Alloc, tuple-like UTuple>
constexpr explicit(see below) tuple(allocation_arg_t, const Alloc& a, UTuple&&);

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

template<class... UTypes>
constexpr tuple& operator=(const tuple<UTypes...>&);

template<class... UTypes>
constexpr const tuple& operator=(const tuple<UTypes...>&) const;

template<class... UTypes>
constexpr tuple& operator=(tuple<UTypes...>&&);

template<class... UTypes>
constexpr const tuple& operator=(tuple<UTypes...>&&) const;

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

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

// only if sizeof...(Types) == 2

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

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

template<tuple-like UTuple>
constexpr tuple& operator=(UTuple&&);

template<tuple-like UTuple>
constexpr const tuple& operator=(UTuple&&) const;

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

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

◆ Construction [tuple.cnstr]

In the descriptions that follow, let \( i \) be in the range \([0, \text{sizeof...}(\text{Types}))\) in order, \( T_i \) be the \( i \)th type in \( \text{Types} \), and \( U_i \) be the \( i \)th type in a template parameter pack named \( \text{UTypes} \), where indexing is zero-based.

For each \( \text{tuple} \) constructor, an exception is thrown only if the construction of one of the types in \( \text{Types} \) throws an exception.

The defaulted move and copy constructor, respectively, of \( \text{tuple} \) is a constexpr function if and only if all required element-wise initializations for move and copy, respectively, would satisfy the requirements for a constexpr function. The defaulted move and copy constructor of \( \text{tuple}<> \) are constexpr functions.

If \( \text{is_trivially_destructible_v}<T_i> \) is true for all \( T_i \), then the destructor of \( \text{tuple} \) is trivial.

The default constructor of \( \text{tuple}<> \) is trivial.

constexpr explicit(see below) tuple();

**Constraints:** \( \text{is_default_constructible_v}<T_i> \) is true for all \( i \).

**Effects:** Value-initializes each element.

**Remarks:** The expression inside explicit evaluates to true if and only if \( T_i \) is not copy-list-initializable from an empty list for at least one \( i \). [Note: This behavior can be implemented with a trait that checks whether a const \( T_i \&) \) can be initialized with \( \{} \). — end note]

constexpr explicit(see below) tuple(const Types&...);

**Constraints:** \( \text{sizeof...}(\text{Types}) \geq 1 \) and \( \text{is_copy_constructible_v}<T_i> \) is true for all \( i \).

**Effects:** Initializes each element with the value of the corresponding parameter.

**Remarks:** The expression inside explicit is equivalent to:

\[ \text{!conjunction}_v<\text{is_convertible}<\text{const Types}&, \text{Types}>...> \]
template<class... UTypes> constexpr explicit(see below) tuple(UTypes&&... u);

Let disambiguating-constraint be:

- negation<is_same<remove_cvref_t<U0>, tuple>> if sizeof...(Types) is 1;
- otherwise, bool_constant<is_same_v<remove_cvref_t<U0>, allocator_arg_t> || is_same_v<remove_cvref_t<T0>, allocator_arg_t>> if sizeof...(Types) is 2 or 3;
- otherwise, true_type.

Constraints:

- sizeof...(Types) equals sizeof...(UTypes),
- sizeof...(Types) ≥ 1, and
- conjunction_v<disambiguating-constraint, is_constructible<Types, UTypes>...> is true.

Effects: Initializes the elements in the tuple with the corresponding value in std::forward<UTypes>(u).

Remarks: The expression inside explicit is equivalent to:

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

tuple(const tuple& u) = default;

Mandates: is_copy_constructible_v<T_i> is true for all i.

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

tuple(tuple&& u) = default;

Constraints: is_move_constructible_v<T_i> is true for all i.

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

template<class... UTypes> constexpr explicit(see below) tuple(tuple<UTypes...>& u);

Let I be the pack 0, 1, ..., (sizeof...(Types) - 1).
Let FWD(u) be static_cast<decltype(u)>(u).

Constraints:

- sizeof...(Types) equals sizeof...(UTypes), and
- (is_constructible_v<TTypes, decltype(get<I>(FWD(u)))> && ...) is true, and
- either sizeof...(Types) is not 1, or (when Types... expands to T and UTypes... expands to U) is_convertible_v<decltype(u), T>, is_constructible_v<T, decltype(u)>, and is_same_v<T, U> are all false.
Effects: For all \( i \), initializes the \( i \)th element of \( *\text{this} \) with \( \text{get}<i>(FWD(u)) \).

Remarks: The expression inside \texttt{explicit} is equivalent to:

\[
!(\text{is\_convertible\_v<decltype(get<0>(FWD(u)))}, \text{Types} && \ldots)
\]

\[
!(\text{is\_convertible\_v<decltype(get<1>(FWD(u)))}, \text{Types} && \ldots)
\]

Constraints:

- \( \text{sizeof...(Types)} \) is 2,
- \( \text{is\_constructible\_v<T_0, decltype(get<0>(FWD(u)))> is true, and} \)
- \( \text{is\_constructible\_v<T_1, decltype(get<1>(FWD(u)))> is true.} \)

Effects: Initializes the first element with \( \text{get}<0>(FWD(u)) \) and the second element with \( \text{get}<1>(FWD(u)) \).

Remarks: The expression inside \texttt{explicit} is equivalent to:

\[
!(\text{is\_convertible\_v<decltype(get<0>(FWD(u)))}, \text{Types} ||
!(\text{is\_convertible\_v<decltype(get<1>(FWD(u)))}, \text{Types} && \ldots)
\]

Constraints:

- \( \text{sizeof...(Types)} \) equals \( \text{tuple\_size\_v<UTuple>} \), and
- \( \text{is\_constructible\_v<Types, decltype(get<0>(FWD(u)))> && \ldots} \) is true, and
- either \( \text{sizeof...(Types)} \) is not 1, or (when Types... expands to T and UTypes... expands to U) \( \text{is\_convertible\_v<decltype(u), T>, is\_constructible\_v<T, decltype(u)>, and is\_same\_v<T, U> are all false.} \)

Effects: For all \( i \), initializes the \( i \)th element of \( *\text{this} \) with \( \text{get}<i>(FWD(u)) \).

Remarks: The expression inside \texttt{explicit} is equivalent to:

\[
!(\text{is\_convertible\_v<decltype(get<0>(FWD(u)))}, \text{Types} && \ldots)
\]

\[
!(\text{is\_convertible\_v<decltype(get<1>(FWD(u)))}, \text{Types} && \ldots)
\]


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

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

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

Preconditions: 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 \([0, \text{sizeof...}(\text{Types}))\) in order, \(T_i\) be the \(i^{\text{th}}\) type in Types, and \(U_i\) be the \(i^{\text{th}}\) 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.

**Returns:** *this.

**Remarks:** This operator is defined as deleted unless is_copyAssignable_v<T_i> is true for all i.

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

**Constraints:** (is_copyAssignable_v<const Types> && ...) is true.

**Effects:** Assigns each element of u to the corresponding element of *this.

**Returns:** *this.

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

**Constraints:** isMoveAssignable_v<T_i> is true for all i.

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

**Returns:** *this.

**Remarks:** The exception specification is equivalent to the logical AND of the following expressions:

isnothrowMoveAssignable_v<T_i>

where T_i is the i-th type in Types.

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

**Constraints:** (isAssignable_v<const Types&, Types> && ...) is true.

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

**Returns:** *this.

template<class... UTypes> castexpr tuple& operator=(const tuple<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> castexpr const tuple& operator=(const tuple<UTypes...>& u) const;

**Constraints:**

17
• sizeof...(Types) equals sizeof...(UTypes) and
  • (is_assignable_v<const Types&, const UTypes& && ...) is true.

**Effects:** Assigns each element of u to the corresponding element of *this.

**Returns:** *this.

template<class... UTypes> constexpr tuple& operator=(tuple<UTypes...>&& u);

**Constraints:**
• sizeof...(Types) equals sizeof...(UTypes) and
• is_assignable_v<Tᵢ&, Uᵢ> is true for all i.

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

**Returns:** *this.

template<class... UTypes> constexpr const tuple& operator=(tuple<UTypes...>&& u) const;

**Constraints:**
• sizeof...(Types) equals sizeof...(UTypes) and
• (is_assignable_v<const Types&, UTypes> && ...) is true.

**Effects:** For all i, assigns std::forward<Uᵢ>(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 const tuple& operator=(const pair<U1, U2>& u) const;

**Constraints:**
• sizeof...(Types) is 2,
• is_assignable_v<const T₀&, const U1&> is true, and
• is_assignable_v<const T₁&, const U2&> is true.
**Effects:** Assigns `u.first` to the first element and `u.second` to the second element.

**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`.

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

**Constraints:**

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

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

**Returns:** `*this`.

template<tuple-like UTuple> constexpr tuple& operator=(UTuple&& u);

**Constraints:**

- `sizeof...(Types)` equals `tuple_size_v<UTuple>` and
- `is_assignable_v<T_i&, element_t<i, UTuple>>` is true for all `i`.

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

**Returns:** `*this`.

template<tuple-like UTuple> constexpr const tuple& operator=(UTuple&& u) const;

**Constraints:**

- `sizeof...(Types)` equals `tuple_size_v<UTuple>` and
- `is_assignable_v<const T_i&, element_t<i, UTuple>>&` is true for all `i`.

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

◆ swap

constexpr void swap(tuple& rhs) noexcept(see below);
constexpr void swap(const tuple& rhs) const noexcept(see below);

Mandates:
• For the first overload, (is_swappable_v<Types> && ...) is true.
• For the second overload, (is_swappable_v<const Types> && ...) is true.

Preconditions: Each element in *this is swappable with the corresponding element in rhs.

Effects: Calls swap for each element in *this and its corresponding element in rhs.

Throws: Nothing unless one of the element-wise swap calls throws an exception.

Remarks: The exception specification is equivalent to
• (is_nothrow_swappable_v<Types> && ...) for the first overload and
• (is_nothrow_swappable_v<const Types> && ...) for the second overload.

◆ Tuple creation functions

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

Returns: tuple<unwrap_ref_decay_t<TTypes>...>(std::forward<TTypes>(t)...).

[Example:
  int i; float j;
  make_tuple(1, ref(i), cref(j))
]

creates a tuple of type tuple<int, int&, const float&>. — end example]

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

Effects: Constructs a tuple of references to the arguments in t suitable for forwarding as arguments to a function. Because the result may contain references to temporary objects, a program shall ensure that the return value of this function does not outlive any of its arguments (e.g., the program should typically not store the result in a named variable).

Returns: tuple<TTypes&&...>(std::forward<TTypes>(t)...).
template<class... TTypes>
constexpr tuple<TTypes&...> tie(TTypes&... t) noexcept;

Returns: tuple<TTypes&...>(t...). When an argument in t is ignore, assigning any value to the corresponding tuple element has no effect.

Example: tie functions allow one to create tuples that unpack tuples into variables. ignore can be used for elements that are not needed:

```cpp
int i; std::string s;
tie(i, ignore, s) = make_tuple(42, 3.14, "C++");
// i == 42, s == "C++"
```

— end example

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

In the following paragraphs, let $T_i$ be the $i$th type in Tuples, $U_i$ be remove_reference_t<$T_i$>, and $t_{pi}$ be the $i$th parameter in the function parameter pack $tpls$, where all indexing is zero-based.

Preconditions: 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:

[Editor’s note: Is “the template parameter pack representing the element types in $U_i$” clear enough?]

- 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

```cpp
get<$k_i$>(std::forward<$T_i$>($t_{pi}$))
```

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

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

Calling a function with a tuple of arguments [tuple.apply]
template<class F, class tuple-like Tuple>
constexpr decltype(auto) apply(F&& f, Tuple&& t);

**Effects:** Given the exposition-only function:

```cpp
namespace std {
    template<class F, class tuple-like Tuple, size_t... I>
    constexpr decltype(auto) apply-impl(F&& f, Tuple&& t, index_sequence<I...>) {
        // exposition only
        return INVOKE(std::forward<F>(f), get<I>(std::forward<Tuple>(t))...); // see ??
    }
}
```

Equivalent to:

```cpp
return apply-impl(std::forward<F>(f), std::forward<Tuple>(t), make_index_sequence<tuple_size_v<remove_reference_t<Tuple>>>{});
```

```cpp
template<class T, class tuple-like Tuple>
constexpr T make_from_tuple(class tuple-like&& t);
```

**Effects:** Given the exposition-only function:

```cpp
namespace std {
    template<class T, class tuple-like Tuple, size_t... I>
    requires is_constructible_v<T, decltype(get<I>(declval<Tuple>()))...>
    constexpr T make-from-tuple-impl(Tuple&& t, index_sequence<I...>) { // exposition only
        return T(get<I>(std::forward<Tuple>(t))...);
    }
}
```

Equivalent to:

```cpp
return make-from-tuple-impl<T>(
    std::forward<Tuple>(t),
    make_index_sequence<tuple_size_v<remove_reference_t<Tuple>>>{});
```

[Note: The type of `T` must be supplied as an explicit template parameter, as it cannot be deduced from the argument list. — end note]
Returns: true if \( \text{get}<i>(t) == \text{get}<i>(u) \) for all \( i \), otherwise false. For any two zero-length tuples \( e \) and \( f \), \( e == f \) returns true.

Remarks: 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.

Remarks: The second overload is to be found via argument-dependent lookup only.

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

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

Effects: Performs a lexicographical comparison between \( t \) and \( u \). For any two zero-length tuples \( t \) and \( u \), \( t <=> u \) returns \strong{equal}. Otherwise, equivalent to:

\[
\text{if (auto c = synth-three-way(get<0>(t), get<0>(u)); c != 0) return c; return } t_{\text{tail}} <=> u_{\text{tail}};
\]

where \( r_{\text{tail}} \) for some tuple \( r \) is a tuple containing all but the first element of \( r \).

Remarks: The second overload is to be found via argument-dependent lookup only.

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

common_reference specialization [tuple.common_ref]

In the description that follow, let \( i \) be in the range \([0, \ \text{tuple.size}_v<TTuple>)\) in order.

Let \( T_i \) be the type denoted by \( \text{tuple.element}_t<i, \ TTuple> \) of a template parameter named \( TTuple \) satisfying \tuple-like. \( \text{TTypes} \) denotes a pack formed by the sequence of \( T_i \).

Let \( U_i \) be the type denoted by \( \text{tuple.element}_t<i, \ UTuple> \) of a template parameter named \( UTuple \) satisfying \tuple-like. \( \text{UTypes} \) denotes a pack formed by the sequence of \( U_i \).

template<
tuple-like TTuple, tuple-like UTuple, template<class> class TQual, template<class> class UQual>
struct basic_common_reference<TTuple, UTuple, TQual, UQual> {
    using type = see below;
};

Constraints:

• \( \text{TTuple} \) is a specialization of \text{tuple} or \( \text{UTuple} \) is a specialization of \text{tuple},

• \( \text{tuple<common_reference_t<TQual<TTypes>, UQual<UTypes>>...>\text{\ldots}} \) denotes a type.
type denotes the type tuple<common_reference_t<TQual<TTypes>, UQual<UTypes>>...>.

template<tuple-like TTuple, tuple-like UTuple>
struct common_type<TTuple, UTuple, TQual, UQual> {
  using type = see below;
};

Constraints:
  • TTuple is a specialization of tuple or UTuple is a specialization of tuple,
  • tuple<common_type_t<TTypes, UTypes>...> denotes a type.

type denotes the type tuple<common_type_t<TTypes, UTypes>...>.

Pairs

In general

The library provides a template for heterogeneous pairs of values. The library also provides a matching function template to simplify their construction and several templates that provide access to pair objects as if they were tuple objects (see ?? and ??).

Class template pair

namespace std {
  template<class T1, class T2>
  struct pair {
    using first_type = T1;
    using second_type = T2;
    T1 first;
    T2 second;

    pair(const pair&) = default;
    pair(pair&&) = default;
    constexpr explicit(pair());
    constexpr explicit(pair(const T1& x, const T2& y));
    template<class U1 = T1, class U2 = T2>
    constexpr explicit(pair(U1&& x, U2&& y));
    template<class U1, class U2>
    constexpr explicit(pair<T1, T2>& p);
    template<class U1, class U2>
    constexpr explicit(pair<T1, T2>&& p);
    template<class U1, class U2>
    constexpr explicit(pair<T1, T2> p);
    template<class U1, class U2>
    constexpr explicit(pair<T1, T2>&& p);
  
  template<pair-like P>
constexpr explicit(see below) pair(P&& p);

template<class... Args1, class... Args2>
constexpr pair(piecewise_construct_t,
tuple<Args1...> first_args, tuple<Args2...> second_args);

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

template<pair-like P>
constexpr pair& operator= pair(P&& p);
template<pair-like P>
constexpr const pair& operator= pair(P&& p) const;

c Jasexpr void swap(pair& p) noexcept;
c Jasexpr void swap(const pair& p) const noexcept;

};
template<class T1, class T2>
pair(T1, T2) -> pair<T1, T2>;
}

Constructors and member functions of pair do not throw exceptions unless one of the element-wise operations specified to be called for that operation throws an exception.
The defaulted move and copy constructor, respectively, of pair is a constexpr function if and only if all required element-wise initializations for move and copy, respectively, would satisfy the requirements for a constexpr function.
If (is_trivially_destructible_v<T1> && is_trivially_destructible_v<T2>) is true, then the destructor of pair is trivial.
pair<T, U> is a structural type if T and U are both structural types. Two values p1 and p2 of type pair<T, U> are template-argument-equivalent if and only if p1.first and p2.first are template-argument-equivalent and p1.second and p2.second are template-argument-equivalent.

constexpr explicit(see below) pair();

Constraints:
• `is_default_constructible_v<T1>` is true and
• `is_default_constructible_v<T2>` is true.

**Effects:** Value-initializes first and second.

**Remarks:** The expression inside `explicit` evaluates to `true` if and only if either `T1` or `T2` is not implicitly default-constructible. [Note: This behavior can be implemented with a trait that checks whether a `const T1&` or a `const T2&` can be initialized with `{}`. — end note]

```cpp
constexpr explicit(see below) pair(const T1& x, const T2& y);
```

**Constraints:**

• `is_copy_constructible_v<T1>` is true and
• `is_copy_constructible_v<T2>` is true.

**Effects:** Initializes first with `x` and second with `y`.

**Remarks:** The expression inside `explicit` is equivalent to:

```cpp
!is_convertible_v<const T1&, T1> || !is_convertible_v<const T2&, T2>
```

```cpp
template<class U1 = T1, class U2 = T2> constexpr explicit(see below) pair(U1&& x, U2&& y);
```

**Constraints:**

• `is_constructible_v<T1, decltype(get<0>(FWD(p)))>` is true and
• `is_constructible_v<T2, decltype(get<1>(FWD(p)))>` is true.

**Effects:** Initializes first with `std::forward<U1>(x)` and second with `std::forward<U2>(y)`.

**Remarks:** The expression inside `explicit` is equivalent to:

```cpp
!is_convertible_v<U1, T1> || !is_convertible_v<U2, T2>
```

```cpp
template<class U1, class U2> constexpr explicit(see below) pair(pair<U1, U2>& p);
template<class U1, class U2> constexpr explicit(see below) pair(const pair<U1, U2>& p);
template<class U1, class U2> constexpr explicit(see below) pair(pair<U1, U2>&& p);
template<class U1, class U2> constexpr explicit(see below) pair(const pair<U1, U2>&& p);
```

```cpp
template<pair-like P> constexpr explicit(see below) pair(P&& p);
```

Let `FWD(u)` be `static_cast<decltype(u)>(u)`.

**Constraints:**

• `is_constructible_v<T1, decltype(get<0>(FWD(p)))> is true` and
• `is_constructible_v<T2, decltype(get<1>(FWD(p)))> is true`. 

26
**Effects:** Initializes first with `get<0>(FWD(p))` and second with `get<1>(FWD(p))`.

**Remarks:** The expression inside `explicit` is equivalent to:

```cpp
!is_convertible_v<decltype(get<0>(FWD(p))), T1> ||
!is_convertible_v<decltype(get<1>(FWD(p))), T2>
```

```cpp
template<class... Args1, class... Args2>
constexpr pair(piecewise_construct_t,
    tuple<Args1...> first_args, tuple<Args2...> second_args);
```

**Mandates:**

- `is_constructible_v<T1, Args1...>` is true
- `is_constructible_v<T2, Args2...>` is true.

**Effects:** Initializes first with arguments of types `Args1...` obtained by forwarding the elements of `first_args` and initializes second with arguments of types `Args2...` obtained by forwarding the elements of `second_args`. (Here, forwarding an element `x` of type `U` within a tuple object means calling `std::forward<U>(x).`) This form of construction, whereby constructor arguments for first and second are each provided in a separate tuple object, is called **piecewise construction**.

```cpp
template<pair-like P> constexpr pair& operator=(P&& p);
```

**Constraints:**

- `is_assignable_v<const T1&, decltype(std::get<0>(std::forward<P>(p)))>` is true, and
- `is_assignable_v<const T2&, decltype(std::get<1>(std::forward<P>(p)))>` is true.

**Effects:** Assigns `std::forward<U1>(p.first)` to first and `std::forward<U2>(u.second)` to second.

**Returns:** `*this`.

```cpp
template<pair-like P> constexpr const pair& operator=(P&& p) const;
```

**Constraints:**

- `is_assignable_v<const T1&, decltype(std::get<0>(std::forward<decltype(p)>(p)))>` is true, and
- `is_assignable_v<const T2&, decltype(std::get<1>(std::forward<decltype(p)>(p)))>` is true.

**Effects:** Assigns `std::get<0>(std::forward<decltype(p)>(p))` to first and `std::get<1>(std::forward<decltype(p)>(p))` to second.

**Returns:** `*this`.

```cpp
template<pair-like P> constexpr const pair& operator=(P&& p) const;
```

**Constraints:**
• \(\text{is assignable v<const T1&, decltype(std::get<0>(std::forward<P>(p)))}\) is true, and

• \(\text{is assignable v<const T2&, decltype(std::get<1>(std::forward<P>(p)))}\) is true.

**Effects:** Assigns \(\text{std::get<0>(std::forward<decltype(p)>(p))}\) to first and \(\text{std::get<1>(std::forward<decltype(p)>(p))}\) to second.

**Returns:** *this.

---

**Specialized algorithms**

```cpp
template<class T1, class T2>
constexpr bool operator==(const pair<T1, T2>& x, const pair<T1, T2>& y);
```

**Specialized algorithms**

```cpp
template<class T1, class T2, pair-like Pair>
requires same_as<T1, tuple_element_t<0, Pair>> && same_as<T2, tuple_element_t<1, Pair>>
constexpr bool operator==(const pair<T1, T2>& x, const Pair& y);
```

**Returns:** \(x\).first == \(y\).first \(\text{std::get<0>(y)}\) && \(x\).second == \(y\).second \(\text{std::get<1>(y)}\).

```cpp
template<class T1, class T2>
constexpr common_comparison_category_t<
synth-three-way-result<T1>,
synth-three-way-result<T2>>
operator<=>(const pair<T1, T2>& x, const pair<T1, T2>& y);
```

```cpp
template<class T1, class T2, pair-like Pair>
requires same_as<T1, tuple_element_t<0, Pair>> && same_as<T2, tuple_element_t<1, Pair>>
constexpr common_comparison_category_t<
synth-three-way-result<T1>,
synth-three-way-result<T2>>
operator<=>(const pair<T1, T2>& x, const Pair& y);
```

**Effects:** Equivalent to:

```cpp
if (auto c = synth-three-way(x.first, y.first \text{std::get<0>(y)}); c \neq 0) return c;
return synth-three-way(x.second, y.second \text{std::get<1>(y)});
```

---

**Memory**

**In general**

Subclause ?? describes the contents of the header

**Header <memory> synopsis**

```cpp
namespace std {

// ??, uses-allocator construction
template<class T, class Alloc, class... Args>
```

28
constexpr auto uses_allocator_construction_args(const Alloc& alloc, Args&&... args) noexcept;

template<class T, class Alloc, class Tuple1, class Tuple2>
constexpr auto uses_allocator_construction_args(const Alloc& alloc, piecewise_construct_t, Tuple1&& x, Tuple2&& y) noexcept;

template<class T, class Alloc>
constexpr auto uses_allocator_construction_args(const Alloc& alloc) noexcept;

template<class T, class Alloc, class U, class V>
constexpr auto uses_allocator_construction_args(const Alloc& alloc, U&& u, V&& v) noexcept;

template<class T, class Alloc, class U, class V>
constexpr auto uses_allocator_construction_args(const Alloc& alloc, pair<U, V>& pr) noexcept;

template<class T, class Alloc, class U, class V>
constexpr auto uses_allocator_construction_args(const Alloc& alloc, const pair<U, V>& pr) noexcept;

template<class T, class Alloc, class U, class V>
constexpr auto uses_allocator_construction_args(const Alloc& alloc, pair<U, V>&& pr) noexcept;

template<class T, class Alloc, class U, class V>
constexpr auto uses_allocator_construction_args(const Alloc& alloc, const pair<U, V>&& pr) noexcept;

template<class T, class Alloc, class U, class V>
constexpr auto uses_allocator_construction_args(const Alloc& alloc, pair-like Pair) noexcept;

// Uses-allocator construction [allocator.uses.construction]

Constraints: T is a specialization of pair.

Effects: Equivalent to:

return uses_allocator_construction_args<T>(alloc, piecewise_construct, tuple<{}>, tuple<{}>);

Constraints: T is a specialization of pair.
**Effects:** Equivalent to:

```
return uses_allocator_construction_args<T>(alloc, piecewise_construct,
    forward_as_tuple(std::forward<U>(u)),
    forward_as_tuple(std::forward<V>(v)));
```

template<class T, class Alloc, class U, class V>
constexpr auto uses_allocator_construction_args(const Alloc& alloc, pair<U, V>& pr) noexcept;
template<class T, class Alloc, class U, class V>
constexpr auto uses_allocator_construction_args(const Alloc& alloc, const pair<U, V>&& pr) noexcept;

**Constraints:** T is a specialization of pair.

**Effects:** Equivalent to:

```
return uses_allocator_construction_args<T>(alloc, piecewise_construct,
    forward_as_tuple(get<0>(std::move(pr)),
    forward_as_tuple(get<1>(std::move(pr))));
```

template<class T, class Alloc, class U, class V>
constexpr auto uses_allocator_construction_args(const Alloc& alloc, pair<U, V>&& pr) noexcept;
template<class T, class Alloc, class U, class V>
constexpr auto uses_allocator_construction_args(const Alloc& alloc, const pair<U, V>&& pr) noexcept;

**Constraints:** T is a specialization of pair.

**Effects:** Equivalent to:

```
return uses_allocator_construction_args<T>(alloc, piecewise_construct,
    forward_as_tuple(get<0>(std::move(pr)),
    forward_as_tuple(get<1>(std::move(pr))));
```

template<class T, class Alloc, class U, class V>
constexpr auto uses_allocator_construction_args(const Alloc& alloc, piecewise_construct_args(const Alloc& alloc, pair<U, V>& pr) noexcept;
template<class T, class Alloc, class U, class V>
constexpr auto uses_allocator_construction_args(const Alloc& alloc, const pair<U, V>& pr) noexcept;

**Constraints:** T is a specialization of pair.

**Effects:** Equivalent to:

```
return uses_allocator_construction_args<T>(alloc, piecewise_construct,
    forward_as_tuple(std::forward<U>(u)),
    forward_as_tuple(std::forward<V>(v)));
```

template<class T, class Alloc, class U, class V>
constexpr auto uses_allocator_construction_args(const Alloc& alloc, piecewise_construct_args(const Alloc& alloc, const pair<U, V>& pr) noexcept;

**Constraints:** T is a specialization of pair.

**Effects:** Equivalent to:

```
return uses_allocator_construction_args<T>(alloc, piecewise_construct,
    forward_as_tuple(std::forward<U>(u)),
    forward_as_tuple(std::forward<V>(v)));
```

[Editor's note: The above wording changes intend to merge all the pair-related uses_allocator_construction_args overloads specification by always calling forward, for each of the 4 existing overload, and the new one.]
Containers

Associative containers

[Editor's note: We probably need to modify the requirements table, which I have found challenging as requirements apply equally to sets and maps. In particular, we probably want to require is_constructible<value_type, T> where T is either the type passed to insert, or the InputIterator’s value_type. Currently, we only seem to require convertible_to, which may not be sufficient?. An alternative is to add explicit insert overloads for pair-like objects].

In general

The header map defines the class templates map and multimap; the header set defines the class templates set and multiset.

The following exposition-only alias templates may appear in deduction guides for associative containers:

```cpp
template<class InputIterator>
using iter-value-type =
 typename iterator_traits<InputIterator>::value_type; // exposition only

tuple_element_t<0, iterator_traits<InputIterator>::value_type_t>; // exposition only

template<class InputIterator>
using iter-key-type = remove_const_t<
 typename iterator_traits<InputIterator>::value_type::first
 tuple_element_t<0, iterator_traits<InputIterator>::value_type_t>; // exposition only

template<class InputIterator>
using iter-mapped-type =
 typename iterator_traits<InputIterator>::value_type::second
 tuple_element_t<1, iterator_traits<InputIterator>::value_type_t>; // exposition only

template<class InputIterator>
using iter-to-alloc-type =
 pair<
 add_const_t<
 typename iterator_traits<InputIterator>::value_type::first
 tuple_element_t<0, iterator_traits<InputIterator>::value_type_t>,
 typename iterator_traits<InputIterator>::value_type::second
 tuple_element_t<1, iterator_traits<InputIterator>::value_type_t>>; // exposition only
```

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 =
        convertible_to<From, To> &&
        // exposition only
```
!(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> && !is_reference_v<T> && pair-like &&
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>>;
        { std::get<N>(t) } -> convertible_to<const tuple_element_t<N, T>>&;
    };

Class template zip_view [range.zip.view]

[Editor's note: Remove [range.zip.view]p1]

Given some pack of types Ts, the alias template tuple-or-pair is defined as follows:

• If sizeof...(Ts) is 2, tuple-or-pair<Ts...> denotes pair<Ts...>.
• Otherwise, tuple-or-pair<Ts...> denotes tuple<Ts...>.

[Editor's note: Replace all usages of tuple-or-pair by tuple in the range clause. This includes [range.zip], [range.adjacent.iterator] as well as [range.cartesian] if P2374 is adopted]
Feature test macros

Insert into [version.syn]
#define __cpp_lib_tuple_like <DATE OF ADOPTION> // also in <utility>, <tuple>

Acknowledgments

Thanks to Casey Carter, Alisdair Meredith, Christopher Di Bella and Tim Song for their invaluable feedbacks!

References