

# Compatibility between tuple, pair and *tuple-like* objects

Document #: P2165R2  
Date: 2021-06-14  
Project: Programming Language C++  
Audience: SG9, LEWG  
Reply-to: Corentin Jabot <[corentin.jabot@gmail.com](mailto:corentin.jabot@gmail.com)>

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

## Abstract

We propose to make pair, tuple, tuple\_cat, and associative containers more compatible with objects satisfying the tuple-protocol.

## Revisions

### 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

Before	After
<pre>constexpr std::pair p {1, 3.0}; constexpr std::tuple t {p}; // OK  std::pair&lt;int, double&gt; pp     (get&lt;0&gt;(t), get&lt;1&gt;(t));  static_assert(std::tuple(p) == t);  static_assert(p == t); static_assert(p &lt;=&gt; t == 0);  std::tuple&lt;int,int&gt; t = std::array {1, 2};  std::map m{     std::pair{get&lt;0&gt;(t), get&lt;1&gt;(t)},     std::pair{get&lt;0&gt;(u), get&lt;1&gt;(u)} };  static_assert(same_as&lt;std::tuple&lt;int&gt;,     range_value_t&lt;decltype(vIEWS::zip(v))&gt;&gt;&gt;);  static_assert(same_as&lt;std::pair&lt;int,int&gt;,     range_value_t&lt;decltype(vIEWS::zip(v, v))&gt;&gt;&gt;);  // x is std::tuple&lt;int, int&gt; // because tuple is convertible from pair auto x = true ? tuple{0,0} : pair{0,0};</pre>	<pre>constexpr std::pair p {1, 3.0}; constexpr std::tuple t {p}; // OK  std::pair&lt;int, double&gt; pp{t};  static_assert(std::tuple(p) == t);  static_assert(p == t); static_assert(p &lt;=&gt; t == 0);  std::tuple&lt;int,int&gt; t = std::array {1, 2};  // not the same size: ill-formed std::tuple&lt;int&gt; t = std::array {1, 2};  std::map m{t, u};  static_assert(same_as&lt;std::tuple&lt;int&gt;,     range_value_t&lt;decltype(vIEWS::zip(v))&gt;&gt;&gt;);  static_assert(same_as&lt;std::tuple&lt;int,int&gt;,     range_value_t&lt;decltype(vIEWS::zip(v, v))&gt;&gt;&gt;);  // Both types are interconvertible, // The expression is ambiguous an this is ill-formed auto x = true ? tuple{0,0} : pair{0,0};</pre>

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.

Having both types in the standard library is somewhat redundant - as noted in [N2270](#) [?] - a problem that [N2533](#) [?] tried to address before C++, alas unsuccessfully.

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](#) [?], 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-like objects, 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](#) [?], cartesian\_product: [P2374R0](#) [?]) 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.

Lastly, while there is support for `enumerate` to have a reference type with named members, there is also a desire that `enumerate(container) | to<map>` should work. In general, it is ridiculously hard and costly to make simple structs that can be used as reference types of ranges. With the proposed changes, any named type that implements the tuple protocol shares a common reference with the corresponding tuple if the members themselves have a common reference.

As such, only

- `tuple_element`
- `tuple_size`
- `get`

need to be provided for a range's reference's type to be any type, as long as the value type is a tuple.

This paper takes care of providing a `basic_common_reference` and a `basic_common_type` between tuple and tuple-like entities. [P1858R2](#) [?] and [P1096R0](#) [?] explores ways to simplify further the tuple protocol.

Standard types supporting the tuple protocol include

- `pair`
- `tuple`
- `array`
- `subrange`
- the proposed `enumerate`'s reference type.

- span of static extent- **prior to P2116R0 [?] which removed that support**

## C arrays and aggregate

C arrays and aggregate are supported by structured bindings, but this mechanism does not use the tuple protocol. As such, these types will not be convertible to `tuple` with this paper.

## 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 `tuple-like` object **of the same size**.
- Allow a `pair` to be constructed, assigned and compared with any `tuple-like` object of size 2.
- With `pair` constructible from any `tuple-like` object, we allow associative containers (like `map`) construction and insertion from any `tuple-like` object. Note that these containers already support insertion/ emplacement from types that their value types is constructible from, so only construction from `std::initializer_list`, deduction guides and maybe iterator constructors need change.
- Can use `tuple` in `zip` and similar views consistently, in the 2 views case.
- Can design `enumerate` as proposed by P2164R5 [?].

**The intent of this paper is not to modify the behavior of `tuple`'s and `pair`'s members. In particular, it does not intend to change the behavior of well-formed existing code in regard to the constraints, `noexcept` and `explicit` specifications placed on existing methods, but only to allow these same methods to work with more types that expose the same semantics as `tuple` and `pair` do.**

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 (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>
```

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

### Breaking API changes

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.

## ABI

This paper removes the existing `pair` overloads in `tuple`, and modify other overloads. They can be kept by an implementation for mangling purposes if necessary.

## Implementation

This proposal has been implemented in `libstdc++` [\[Github\]](#).

## Future work

Tuple comparison operators are good candidates for hidden friends.

## Wording

### ◆ Header <tuple> synopsis

[tuple.syn]

[Editor's note: The wording directly below is relative to P2321R1]

```
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 basic_common_type<TTuple, UTuple, TQual, UQual>;

[...]
```

```
// ??, 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>
concept is_tuple_element = requires (T t) { // exposition only
    typename std::tuple_element_t<N, std::remove_const_t<T>>;
    { get<N>(t) } -> std::convertible_to<std::tuple_element_t<N, T>&>;
};
```

```
template <typename T>
concept tuple_like // exposition only
    = !is_reference_v<T> && requires {
    typename tuple_size_v<T>::type;
    same_as<decltype(tuple_size_v<T>), size_t>;
} && [<std::size_t... I>(std::index_sequence<I...>)
    { return (is_tuple_element<T, I> && ..); }(std::make_index_sequence<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 tuple_like UTuple>
constexpr bool operator==(const tuple<TTypes...>&, const tuple<UTypes...> UTuple&);
```

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

```
// [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 tuple-like UTuple>
constexpr explicit(see below) tuple(const tuple<UTypes... UTuple&);
template<class... UTypes tuple-like UTuple>
constexpr explicit(see below) tuple(tuple<UTypes... UTuple&&);

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

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

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

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);
};

```

## ❖ Construction

[tuple.cnstr]

In the descriptions that follow, let  $i$  be in the range  $[0, \text{sizeof} \dots (\text{Types}))$  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 the type denoted by `tuple_element_t<i, UTuple>` of a template parameter named UTuple satisfying *tuple-like*. UTypes denotes a pack formed of the sequence of  $U_i$ .

For each tuple constructor, an exception is thrown only if the construction of one of the types in Types throws an exception.

[...]

```

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

```

*Constraints:*

- `sizeof...(Types)` equals ~~`sizeof...(UTypes)`~~ `tuple_size_v<UTuple>` 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 tuple-like UTuple>
constexpr explicit(see below) tuple(tuple<UTypes...> UTuple&& u);

```

*Constraints:*

- `sizeof...(Types)` equals `sizeof...(UTypes) tuple_size_v<UTuple>`, and
- `is_constructible_v<Ti, Ui>` 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> UTuple, T>`, `is_constructible_v<T, tuple<U> UTuple>`, and `is_same_v<T, U>` are all false.

*Effects:* For all  $i$ , initializes the  $i^{\text{th}}$  element of `*this` with `std::forward<Ui>(get<math>i</math>(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<T0, const U1&>` is true, and
- `is_constructible_v<T1, 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 tuple-like UTuple>
constexpr explicit(see below)
tuple(allocator_arg_t, const Alloc& a, const tuple<UTypes...> UTuple&);
template<class Alloc, class... UTypes tuple-like UTuple>
constexpr explicit(see below)
tuple(allocator_arg_t, const Alloc& a, tuple<UTypes...> UTuple&&);

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  $[0, \text{sizeof} \dots (\text{Types}))$  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~~ *the type denoted by `tuple_element_t<i, UTuple>` of a template parameter named UTuple satisfying *tuple-like*. `UTypes` denotes a pack formed of the sequence of  $U_i$ .*

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

*Effects:* Assigns each element of  $u$  to the corresponding element of  $*\text{this}$ .

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

*Returns:*  $*\text{this}$ .

```
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<Ti>`

where T<sub>i</sub> is the *i*<sup>th</sup> type in Types.

*Returns:* \*this.

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

*Constraints:*

- sizeof...(Types) equals ~~sizeof...(UTypes)~~ tuple\_size\_v<UTuple> and
- is\_assignable\_v<T<sub>i</sub>&, const U<sub>i</sub>&> is true for all *i*.

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

*Returns:* \*this.

```
template<class... UTypes tuple-like UTuple>
constexpr tuple& operator=(tuple<UTypes...> UTuple&& u);
```

*Constraints:*

- sizeof...(Types) equals ~~sizeof...(UTypes)~~ tuple\_size\_v<UTuple> and
- is\_assignable\_v<T<sub>i</sub>&, U<sub>i</sub>> is true for all *i*.

*Effects:* For all *i*, assigns std::forward<U<sub>i</sub>>(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<sub>0</sub>&, const U1&> is true, and
- is\_assignable\_v<T<sub>1</sub>&, 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  $T_i$  be the  $i^{\text{th}}$  type in `Tuples`,  $U_i$  be `remove_reference_t<Ti>`, and  $tp_i$  be the  $i^{\text{th}}$  parameter in the function parameter pack `tpls`, where all indexing is zero-based.

*Expects:* For all  $i$ ,  $U_i$  is the type `cvi tuple<Argsi...>`, where `cvi` is the (possibly empty)  $i^{\text{th}}$  *cv-qualifier-seq* and `Argsi` is the template parameter pack representing the element types in  $U_i$ . Let  $A_{ik}$  be the  $k^{\text{th}}$  type in `Argsi`. 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<Aik, cvi Aik&> == true`, otherwise
- `is_constructible_v<Aik, cvi Aik&&> == true`.

*Remarks:* The types in `CTypes` are equal to the ordered sequence of the extended types `Args0...`, `Args1...`, ..., `Argsn-1...`, where  $n$  is equal to `sizeof...(Tuples)`. Let  $e_i...$  be the  $i^{\text{th}}$  ordered sequence of tuple elements of the resulting tuple object corresponding to the type sequence `Argsi`.

*Returns:* A tuple object constructed by initializing the  $k_i^{\text{th}}$  type element  $e_{ik}$  in  $e_i...$  with `get<ki>(std::forward<Ti>(tpi))`

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

[tuple.rel]

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

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

**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`.

**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`.

In the description that follow, let  $i$  be in the range  $[0, \text{tuple\_size\_v}<\text{UTuple}>)$  in order,  $U_i$  the type denoted by `tuple_element_t<i, UTuple>` of a template parameter named `UTuple` satisfying *tuple-like*. `UTypes` denotes a pack formed of the sequence of  $U_i$ .

```
template<class... TTypes, class... UTypes tuple-like UTuple>
constexpr common_comparison_category_t<synth-three-way-result<TTypes, UTypes>...>
operator<=>(const tuple<TTypes...>& t, const tuple<UTypes...> UTuple& 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  $t_{\text{tail}}$  (or  $u_{\text{tail}}$ ) 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*]

[...]

## common\_reference specialization

[[tuple.common\\_ref](#)]

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

Let  $T_i$  be the type denoted by `tuple_element_t<i, TTuple>` of a template parameter named `TTuple` satisfying *tuple-like*. `TTypes` denotes a pack formed of the sequence of  $T_i$ .

Let  $U_i$  be the type denoted by `tuple_element_t<i, UTuple>` of of a template parameter named `UTuple` satisfying *tuple-like*. `UTypes` denotes a pack formed of 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:**

- `TTuple` is a specialization of `tuple` or `UTuple` is a specialization of `tuple`,
- `tuple_size_v<TTuple> == tuple_size_v<UTuple>` is true and,
- `tuple<common_reference_t<TQual<TTypes>, UQual<UTypes>>...>` denotes a type.

type denotes the type `tuple<common_reference_t<TQual<TTypes>, UQual<UTypes>>...>`.

```
template<tuple-like TTuple, tuple-like UTuple>
struct basic_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_size_v<TTuple> == tuple_size_v<UTuple>` is true and,
- `tuple<common_type_t<TTypes, UTypes>...>` denotes a type.

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



## Pairs

[pairs]



### In general

[pairs.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

[pairs.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(see below) pair();
        constexpr explicit(see below) pair(const T1& x, const T2& y);
        template<class U1, class U2>
        constexpr explicit(see below) pair(U1&& x, U2&& y);
        template<class U1, class U2 pair-like U>
        constexpr explicit(see below) pair(const pair<U1, U2> U& p);
        template<class U1, class U2 pair-like U>
```

```

constexpr explicit(see below) pair(pair<U1, U2> U&& 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);
template<class U1, class U2 pair-like U>
constexpr pair& operator=(const pair<U1, U2> U& p);
constexpr pair& operator=(pair&& p) noexcept(see below);
template<class U1, class U2 pair-like U>
constexpr pair& operator=(pair<U1, U2> U&& p);

constexpr void swap(pair& p) noexcept(see below);
};

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

```

[....]

```

template<class U1, class U2 pair-like U>
constexpr explicit(see below) pair(const pair<U1, U2> U& p);

```

**Constraints:**

- is\_constructible\_v<first\_type, const U1 tuple\_element\_t<0, remove\_cvref\_t<U>>&> is true and
- is\_constructible\_v<second\_type, const U2 tuple\_element\_t<1, remove\_cvref\_t<U>>&> is true.

**Effects:** Initializes members from the corresponding members of the argument.

Initializes first with get<0>(p) and second with get<1>(p).

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

```

!is_convertible_v<const U1 tuple_element_t<0, remove_cvref_t<U>>&, first_type>
|| !is_convertible_v<const U2 tuple_element_t<1, remove_cvref_t<U>>&, second_type>

```

```

template<class U1, class U2 pair-like U>
constexpr explicit(see below) pair(pair<U1, U2> U&& p);

```

**Constraints:**

- is\_constructible\_v<first\_type, U1 tuple\_element\_t<0, remove\_cvref\_t<U>>> is true and
- is\_constructible\_v<second\_type, U2 tuple\_element\_t<1, remove\_cvref\_t<U>>> is true.

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



Initializes first with `get<0>(std::forward<U>(p))`  
and second with `get<1>(std::forward<U>(p))`.

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

```
!is_convertible_v<U1 tuple_element_t<0, remove_cvref_t<U>>, first_type>  
|| !is_convertible_v<U2 tuple_element_t<1, remove_cvref_t<U>>, second_type>
```

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

**Mandates:**

- `is_constructible_v<first_type, Args1...>` is true and
- `is_constructible_v<second_type, 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*.

```
constexpr pair& operator=(const pair& p);
```

**Effects:** Assigns `p.first` to `first` and `p.second` to `second`.

**Returns:** `*this`.

**Remarks:** This operator is defined as deleted unless `is_copy_assignable_v<first_type>` is true and `is_copy_assignable_v<second_type>` is true.

```
template<<class U1, class U2 pair-like U>  
constexpr pair& operator=(const pair<U1, U2> U& p);
```

**Constraints:**

- `is_assignable_v<first_type&, const U1 tuple_element_t<0, remove_cvref_t<U>>&>` is true and
- `is_assignable_v<second_type&, const U2 tuple_element_t<1, remove_cvref_t<U>>&>` is true.

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

**Returns:** `*this`.

```
constexpr pair& operator=(pair&& p) noexcept(see below);
```

**Constraints:**

- `is_move_assignable_v<first_type>` is true and
- `is_move_assignable_v<second_type>` is true.

*Effects:* Assigns to `first` with `std::forward<first_type>(p.first)` and to `second` with `std::forward<second_type>(p.second)`.

*Returns:* `*this`.

*Remarks:* The exception specification is equivalent to:

```
is_nothrow_move_assignable_v<T1> && is_nothrow_move_assignable_v<T2>
```

```
template<<class U1, class U2 pair-like U>
constexpr pair& operator=(pair<U1, U2> U&& p);
```

*Constraints:*

- `is_assignable_v<first_type&, U1 tuple_element_t<0, remove_cvref_t<U>>>` is true and
- `is_assignable_v<second_type&, U2 tuple_element_t<1, remove_cvref_t<U>>>` is true.

*Effects:* Assigns to `first` with `std::forward<U1>(p.first)` `get<0>(std::forward<U>(p))` and to `second` with `std::forward<U2>(p.second)` `get<1>(std::forward<U>(p))`.

*Returns:* `*this`.

```
constexpr void swap(pair& p) noexcept(see below);
```

*Expects:* `first` is swappable with `p.first` and `second` is swappable with `p.second`.

*Effects:* Swaps `first` with `p.first` and `second` with `p.second`.

*Remarks:* The exception specification is equivalent to:

```
is_nothrow_swappable_v<first_type> && is_nothrow_swappable_v<second_type>
```



## Specialized algorithms

[pairs.spec]

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

*Returns:* `x.first == y.first && x.second == y.second`.

```
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<T1, T2> Pair& y);
```

*Effects:* Equivalent to:

```
if (auto c = synth-three-way(x.first, y-first get<0>(y)); c != 0) return c;
return synth-three-way(x.second, y-second get<1>(y));
```



## 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>&>;
-};

```

◆ **Containers** [containers]

◆ **Associative containers** [associative]

[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** [associative.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:

```

template<class InputIterator>
using iter_value_type =
  typename iterator_traits<InputIterator>::value_type;           // exposition only
template<class InputIterator>
using iter_key_type = remove_const_t<
  typename iterator_traits<InputIterator>::value_type::first_type
tuple_element_t<0, iterator_traits<InputIterator>::value_type>>; // exposition only
template<class InputIterator>
using iter_mapped_type =
  typename iterator_traits<InputIterator>::value_type::second_type
tuple_element_t<1, iterator_traits<InputIterator>::value_type>>; // exposition only
template<class InputIterator>
using iter_to_alloc_type = pair<
  add_const_t<typename iterator_traits<InputIterator>::value_type::first_type
tuple_element_t<0, iterator_traits<InputIterator>::value_type>>,
  typename iterator_traits<InputIterator>::value_type::second_type
tuple_element_t<1, iterator_traits<InputIterator>::value_type>>; // exposition only

```

## ❖ Class template map

[map]

### ❖ Overview

[map.overview]

A map is an associative container that supports unique keys (contains at most one of each key-value) and provides for fast retrieval of values of another type T based on the keys. The map class supports bidirectional iterators.

A map meets all of the requirements of a container, of a reversible container, of an associative container, and of an allocator-aware container (). A map also provides most operations described in ?? for unique keys. This means that a map supports the a\_uniq operations in ?? but not the a\_eq operations. For a map<Key,T> the key\_type is Key and the value\_type is pair<const Key,T>. Descriptions are provided here only for operations on map that are not described in one of those tables or for operations where there is additional semantic information.

```
namespace std {
    template<class Key, class T, class Compare = less<Key>,
            class Allocator = allocator<pair<const Key, T>>>
    class map {
    public:
        // types
        using key_type          = Key;
        using mapped_type       = T;
        using value_type        = pair<const Key, T>;
        using key_compare       = Compare;
        using allocator_type    = Allocator;
        using pointer           = typename allocator_traits<Allocator>::pointer;
        using const_pointer     = typename allocator_traits<Allocator>::const_pointer;
        using reference         = value_type&;
        using const_reference   = const value_type&;
        using size_type         = implementation-defined; // see ??
        using difference_type   = implementation-defined; // see ??
        using iterator          = implementation-defined; // see ??
        using const_iterator    = implementation-defined; // see ??
        using reverse_iterator  = std::reverse_iterator<iterator>;
        using const_reverse_iterator = std::reverse_iterator<const_iterator>;
        using node_type         = unspecified;
        using insert_return_type = insert-return-type<iterator, node_type>;

        class value_compare {
        friend class map;
        protected:
            Compare comp;
            value_compare(Compare c) : comp(c) {}
        public:
            bool operator()(const value_type& x, const value_type& y) const {
                return comp(x.first, y.first);
            }
        };

        // ??, construct/copy/destroy
```

```

map() : map(Compare()) { }
explicit map(const Compare& comp, const Allocator& = Allocator());
template<class InputIterator>
map(InputIterator first, InputIterator last,
const Compare& comp = Compare(), const Allocator& = Allocator());
map(const map& x);
map(map&& x);
explicit map(const Allocator&);
map(const map&, const Allocator&);
map(map&&, const Allocator&);
template<pair-like P>
requires is_constructible_v<value_type, P>
map(initializer_list<value_type P>, const Compare& = Compare(), const Allocator& = Allocator());
template<class InputIterator>
map(InputIterator first, InputIterator last, const Allocator& a)
: map(first, last, Compare(), a) { }
template<pair-like P>
requires is_constructible_v<value_type, P>
map(initializer_list<value_type P> il, const Allocator& a)
: map(il, Compare(), a) { }
~map();
map& operator=(const map& x);
map& operator=(map&& x)
noexcept(allocator_traits<Allocator>::is_always_equal::value &&
is_nothrow_move_assignable_v<Compare>);

template<pair-like P>
requires is_constructible_v<value_type, P>
map& operator=(initializer_list<value_type P>);
allocator_type get_allocator() const noexcept;

// iterators
iterator          begin() noexcept;
const_iterator    begin() const noexcept;
iterator          end() noexcept;
const_iterator    end() const noexcept;

reverse_iterator  rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
reverse_iterator  rend() noexcept;
const_reverse_iterator rend() const noexcept;

const_iterator    cbegin() const noexcept;
const_iterator    cend() const noexcept;
const_reverse_iterator crbegin() const noexcept;
const_reverse_iterator crend() const noexcept;

// capacity
[[nodiscard]] bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

```

```

// ??, element access
mapped_type& operator[](const key_type& x);
mapped_type& operator[](key_type&& x);
mapped_type& at(const key_type& x);
const mapped_type& at(const key_type& x) const;

// ??, modifiers
template<class... Args> pair<iterator, bool> emplace(Args&&... args);
template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
pair<iterator, bool> insert(const value_type& x);
pair<iterator, bool> insert(value_type&& x);
template<class P> pair<iterator, bool> insert(P&& x);
iterator insert(const_iterator position, const value_type& x);
iterator insert(const_iterator position, value_type&& x);
template<class P>
iterator insert(const_iterator position, P&&);
template<class InputIterator>
void insert(InputIterator first, InputIterator last);
void insert(initializer_list<value_type>);

node_type extract(const_iterator position);
node_type extract(const key_type& x);
insert_return_type insert(node_type&& nh);
iterator insert(const_iterator hint, node_type&& nh);

template<class... Args>
pair<iterator, bool> try_emplace(const key_type& k, Args&&... args);
template<class... Args>
pair<iterator, bool> try_emplace(key_type&& k, Args&&... args);
template<class... Args>
iterator try_emplace(const_iterator hint, const key_type& k, Args&&... args);
template<class... Args>
iterator try_emplace(const_iterator hint, key_type&& k, Args&&... args);
template<class M>
pair<iterator, bool> insert_or_assign(const key_type& k, M&& obj);
template<class M>
pair<iterator, bool> insert_or_assign(key_type&& k, M&& obj);
template<class M>
iterator insert_or_assign(const_iterator hint, const key_type& k, M&& obj);
template<class M>
iterator insert_or_assign(const_iterator hint, key_type&& k, M&& obj);

iterator erase(iterator position);
iterator erase(const_iterator position);
size_type erase(const key_type& x);
iterator erase(const_iterator first, const_iterator last);
void swap(map&)
noexcept(allocator_traits<Allocator>::is_always_equal::value &&
is_nothrow_swappable_v<Compare>);
void clear() noexcept;

```

```

template<class C2>
void merge(map<Key, T, C2, Allocator>& source);
template<class C2>
void merge(map<Key, T, C2, Allocator>&& source);
template<class C2>
void merge(multimap<Key, T, C2, Allocator>& source);
template<class C2>
void merge(multimap<Key, T, C2, Allocator>&& source);

// observers
key_compare key_comp() const;
value_compare value_comp() const;

// map operations
iterator      find(const key_type& x);
const_iterator find(const key_type& x) const;
template<class K> iterator      find(const K& x);
template<class K> const_iterator find(const K& x) const;

size_type      count(const key_type& x) const;
template<class K> size_type count(const K& x) const;

bool           contains(const key_type& x) const;
template<class K> bool contains(const K& x) const;

iterator      lower_bound(const key_type& x);
const_iterator lower_bound(const key_type& x) const;
template<class K> iterator      lower_bound(const K& x);
template<class K> const_iterator lower_bound(const K& x) const;

iterator      upper_bound(const key_type& x);
const_iterator upper_bound(const key_type& x) const;
template<class K> iterator      upper_bound(const K& x);
template<class K> const_iterator upper_bound(const K& x) const;

pair<iterator, iterator>      equal_range(const key_type& x);
pair<const_iterator, const_iterator> equal_range(const key_type& x) const;
template<class K>
pair<iterator, iterator>      equal_range(const K& x);
template<class K>
pair<const_iterator, const_iterator> equal_range(const K& x) const;
};

template<class InputIterator, class Compare = less<iter-key-type<InputIterator>>,
class Allocator = allocator<iter-to-alloc-type<InputIterator>>>
map(InputIterator, InputIterator, Compare = Compare(), Allocator = Allocator())
-> map<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>, Compare, Allocator>;

template<pair-like T class Key, class T, class Compare = less<Key tuple_element_t<0, T>>,
class Allocator = allocator<pair<const Key tuple_element_t<0, T>, T>>>

```



```

map(initializer_list<pair<Key, T> T>, Compare = Compare(), Allocator = Allocator())
-> map<Key tuple_element_t<0, T>, T tuple_element_t<1, T>, Compare, Allocator>;

template<class InputIterator, class Allocator>
map(InputIterator, InputIterator, Allocator)
-> map<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>,
less<iter-key-type<InputIterator>>, Allocator>;

template<class Key, class T pair-like T, class Allocator>
map(initializer_list<<pair<Key, T> T>, Allocator)
-> map<Key tuple_element_t<0, T>, T tuple_element_t<1, T>, less<Key tuple_element_t<0, T>>, Allocator>;
}

```

[Editor's note: TODO: multimap, unordered\_map, unordered\_multimap]

## Impact on zip & cartesian\_product wordings

In [P2374R0](#) [?] and [P2321R1](#) [?]

[Editor's note: TODO: Write actual wording when these papers get merged]

- Remove the exposition-only tuple-or-pair declaration
- Replace all use of tuple-or-pair by tuple

## Acknowledgments

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

## References

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