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

Document #:	P2165R4
Date:	2022-07-15
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

R4

The following changes where made following LWG small group review (June 14th 2022)

- Remove the comparison between pair and tuple-like as they were insufficient to satisfay equality_comparible as a common_reference isn't specialized for a pair and a tuple-like.
- Remove the changes to support use_allocator construction which the group agreed were not needed.
- Modify the definition of `tuple-like` to support `std::get` implementations returning prvalues, such as `subrange`.
- Simplify the wording of the added tuple constructor.
- Constrain the added `tuple` constructor, and the `tuple` assignement operators from tuple-like to types that are not exactly the same `tuple`.
- Constrain the `pair` assignement operators from tuple-like to types that are not exactly the same `pair`.
- Rewrite the `tuple_cat` wording to be clearer (Thanks Tomasz!)
- Fix a number of typos, errors and formatting issues.

The following changes where made following LWG review (June 24th 2022)

- Rewrite `tuple-like` to explicity state only `pair`, `tuple`, `array`.
- Mark the return type of the new `operator<=>` "see below" in the synopsis.
- The description of the new `operator<=>` in `<tuple>` was referring to a non-existing pack. Add wording to introduce that pack.
- `tuple_element_t` and `tuple_size_v` were sometimes used on reference types. Add `remove_cvref_t`.
- `common_type` still had extraneous template parameters.
- Constrain `common_type` and `basic_common_reference` of `tuple-like` to require non cvref qualified types and `tuple-like` of the same size.
- The new `tuple` constructor still had some unecessary wording referencing a non existing Ui. Remove that wording.
- Fix the effects of `tuple::operator=(tuple-like)`.
- `std::get` was inconsistently qualified. As the existing wording uses it unqualied, remove the qualifications.
- In the description of `tuple_cat`, `remove_cvref_t` instead of `remove_reference_t`, for consistency.

The following changes where made following LWG small group review (June 28th 2022)

- Fix notation in the description of `tuple_cat`.
- Replace "Preconditions" by "Mandates" in the description of `tuple_cat`.
- Reorder constraints in `basic_common_reference` and `common_type`
- Remove `FWD` in the tuple constructor (only used once)
- Fix a couple of typos and formatting issue
- Remove an extraneous > in [associative.general]
- Add references to *different-from*
- In `operator==(tuple, tuple)`, the statements that two empty tuples compare equal is now a note.

The following changes where made following LWG review (July 8th 2022)

- The `tuple-like` concept now models **and satisfies** its requirements.
- Reword the introduction of TTypes/Utypes in [tuple.common_ref].
- Reorder the constraints in [tuple.common_ref].
- Use the wording "The member *typedef-name* `type`..." in [tuple.common_ref].
- Missing period in `tuple_cat`
- In `tuple_cat`, introduce n first and don't use range notation.

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 foud 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

Before

```cpp
constexpr std::pair  p {1, 3.0};
constexpr std::tuple t {p}; // OK

std::pair<int, double> pp
             (get<0>(t), get<1>(t));

static_assert(std::tuple(p) == t);

static_assert(p == t);
static_assert(p <=> t == 0);

std::tuple<int,int> t = std::array {1, 2};


static_assert(same_as<std::tuple<int>,
    range_value_t<decltype(views::zip(v))>>);

static_assert(same_as<std::pair<int,int>,
    range_value_t<decltype(views::zip(v, v))>>);


// x is std::tuple<int, int>
// because tuple is convertible from pair
auto x = true ? tuple{0,0} : pair{0,0};
```

After

```cpp
constexpr std::pair  p {1, 3.0};
constexpr std::tuple t {p}; // OK

std::pair<int, double> pp{t};


static_assert(std::tuple(p) == t);

static_assert(p == t);
static_assert(p <=> t == 0);

std::tuple<int,int> t = std::array {1, 2};

// not the same size: ill-formed
std::tuple<int> t = std::array {1, 2};


static_assert(<std::tuple<int>,
    range_value_t<decltype(views::zip(v))>>);

static_assert(same_as<std::tuple<int,int>,
    range_value_t<decltype(views::zip(v, v))>>);


// Both types are interconvertible,
// The expression is ambiguous an this is ill-formed
auto x = true ? tuple{0,0} : pair{0,0};
```

Red text is ill-formed

Motivation

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

Notably, a tuple can be constructed and assigned from a pair, 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 [3] - a problem that N2533 [4] 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 [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], `carthesian_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 addopt 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-exising 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

Open Questions

pair const assignement

`zip` added const assignement 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.

```
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

```
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 [tuple.syn]

```
#include <compare>              // see ??

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


template <typename T>
concept tuple-like =  see below; // exposition only

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

template<class... TTypes, class... UTypes, template<class> class TQual>
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>
struct common_type<TTuple, UTuple>;


// ??, 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<CTypes...> 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 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;
}
```

� Concept *tuple-like* [tuple.like]

```
template <typename T>
concept tuple-like =  see below; // exposition only
```

A type `T` models and satisfies the exposition-only concept *tuple-like* if `std::remove_cvref_t<T>` is a specialization of `array`, `pair`, `tuple` or `ranges::subrange`.

� Class template `tuple` [tuple.tuple]

```
namespace std {
template<class... Types>
class tuple {
    public:
    // ??, tuple construction
    constexpr explicit(see below) tuple();
    constexpr explicit(see below) tuple(const Types&...); // only if sizeof...(Types) >= 1
    template<class... UTypes>
    constexpr explicit(see below) tuple(UTypes&&...); // only if sizeof...(Types) >= 1

    tuple(const tuple&) = default;
    tuple(tuple&&) = default;
```

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

template<tuple-like UTuple>
constexpr explicit(see below) tuple(UTuple&&);

// allocator-extended constructors
template<class Alloc>
constexpr explicit(see below)
tuple(allocator_arg_t, const Alloc& a);
template<class Alloc>
constexpr explicit(see below)
tuple(allocator_arg_t, const Alloc& a, const Types&...);
template<class Alloc, class... UTypes>
constexpr explicit(see below)
tuple(allocator_arg_t, const Alloc& a, UTypes&&...);
template<class Alloc>
constexpr tuple(allocator_arg_t, const Alloc& a, const tuple&);
template<class Alloc>
constexpr tuple(allocator_arg_t, const Alloc& a, tuple&&);
template<class Alloc, class... UTypes>
constexpr explicit(see below)
tuple(allocator_arg_t, const Alloc& a, 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, const tuple<UTypes...>&&);
template<class Alloc, class U1, class U2>
constexpr explicit(see below)
tuple(allocator_arg_t, const Alloc& a, pair<U1, U2>&);
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>&&);
    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, tuple-like UTuple>
    constexpr explicit(see below) tuple(allocator_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(see below);
};

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, sizeof...(Types))` in order, T_i be the i^{th} type in `Types`, and U_i be the i^{th} type in a template parameter pack named `UTypes`, where indexing is zero-based.

[...]

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

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

Constraints:

- `sizeof...(Types)` is 2,
- `is_constructible_v<`T_0`, decltype(get<0>(`*FWD*`(u)))>` is `true`, and
- `is_constructible_v<`T_1`, decltype(get<1>(`*FWD*`(u)))>` is `true`.

Effects: Initializes the first element with `get<0>(`*FWD*`(u))` and the second element with `get<1>(`*FWD*`(u))`.

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

```
!is_convertible_v<decltype(get<0>(FWD(u))), T0> ||
!is_convertible_v<decltype(get<1>(FWD(u))), T1>
```

```
template<tuple-like UTuple> constexpr explicit(see below) tuple(UTuple&& u);
```

Let `I` be the pack `0, 1, ..., (sizeof...(Types) - 1)`.

Constraints:

- *`different-from`*`<UTuple, tuple>` [range.utility.helpers] is `true`,
- `remove_cvref_t<UTuple>` is not a specialization of `ranges::subrange`,
- `sizeof...(Types)` equals `tuple_size_v<remove_cvref_t<UTuple>>`,
- `(is_constructible_v<Types, decltype(get<I>(std::forward<UTuple>(u)))> && ...)` is `true`, and

- either `sizeof...(Types)` is not 1, or (when `Types...` expands to `T`) `is_convertible_v<UTuple, T>`, `is_constructible_v<T, UTuple>` are both `false`.

Effects: For all i, initializes the i^{th} element of `*this` with `get<`i`>(std::forward<UTuple>(u))`.

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

```
!(is_convertible_v<decltype(get<I>(std::forward<UTuple>(u))), Types> && ...)
```

```
template<class Alloc>
constexpr explicit(see below)
tuple(allocator_arg_t, const Alloc& a);
template<class Alloc>
constexpr explicit(see below)
tuple(allocator_arg_t, const Alloc& a, const Types&...);
template<class Alloc, class... UTypes>
constexpr explicit(see below)
tuple(allocator_arg_t, const Alloc& a, UTypes&&...);
template<class Alloc>
constexpr tuple(allocator_arg_t, const Alloc& a, const tuple&);
template<class Alloc>
constexpr tuple(allocator_arg_t, const Alloc& a, tuple&&);
template<class Alloc, class... UTypes>
constexpr explicit(see below)
tuple(allocator_arg_t, const Alloc& a, 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, const tuple<UTypes...>&&);
template<class Alloc, class U1, class U2>
constexpr explicit(see below)
tuple(allocator_arg_t, const Alloc& a, pair<U1, U2>&);
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>&&);
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, tuple-like UTuple>
constexpr explicit(see below) tuple(allocator_arg_t, const Alloc& a, UTuple&&);
```

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

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

� Assignment [tuple.assign]

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

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

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

Returns: `*this`.

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

[...]

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

Constraints:

- `sizeof...(Types)` is 2,
- `is_assignable_v<const T_0&, U1>` is `true`, and
- `is_assignable_v<const T_1&, 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:

- `different-from<UTuple, tuple>` [range.utility.helpers] is `true`,
- `remove_cvref_t<UTuple>` is not a specialization of `ranges::subrange`,
- `sizeof...(Types)` equals `tuple_size_v<remove_cvref_t<UTuple>>` and,
- `is_assignable_v<T_i&, decltype(get<i>(std::forward<UTuple>(u)))>` is `true` for all i.

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

Returns: `*this`.

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

Constraints:

- `different-from<UTuple, tuple>` [range.utility.helpers] is `true`,

- `remove_cvref_t<UTuple>` is not a specialization of `ranges::subrange`,
- `sizeof...(Types)` equals `tuple_size_v<remove_cvref_t<UTuple>>` and,
- `is_assignable_v<const` T_i`&, decltype(get<`i`>(std::forward<UTuple>(u)))>` is `true` for all i.

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

Returns: `*this`.

� Tuple creation functions [tuple.creation]

[Editor's note: Modify the definition of `tuple_cat` as follow in 22.4.4]

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

Let n be `sizeof...(Tuples)`.

For every integer $0 \leq i < n$:

- Let T_i be the i^{th} type in `Tuples`.
- Let U_i be `remove_cvref_t<`T_i`>`
- Let tp_i be the i^{th} element in the function parameter pack `tpls`.
- Let S_i be `tuple_size_v<`U_i`>`.
- Let E_i^k be `tuple_element_t<`k`,` U_i`>`.
- Let e_i^k be `get<`k`>(std::forward<`T_i`>(`tp_i`))`.
- Let $Elems_i$ be a pack of the types $E_i^0, \cdots, E_i^{S_i-1}$.
- Let $elems_i$ be a pack of the expressions $e_i^0, \cdots, e_i^{S_i-1}$.

The types in `CTypes` are equal to the ordered sequence of the expanded packs of types $Elems_0 \ldots, Elems_1 \ldots, \cdots, Elems_{n-1} \ldots$.

Let `celems` be the ordered sequence of the expanded packs of expressions $elems_0 ..., \cdots, elems_{n-1} ...$.

Mandates: `(is_constructible_v<CTypes, decltype(celems)> && ...)` is `true`.

Returns: `tuple<CTypes...>(celems...)`.

~~In the following paragraphs, let T_i be the i^{th} type in `Tuples`, U_i be `remove_reference_t<`T_i`>`, and tp_i 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:~~

- If $\texttt{T}_i$ is deduced as an lvalue reference type, then $\texttt{is\_constructible\_v<A}_{ik}\texttt{,}\ \mathit{cv}_i\ \texttt{A}_{ik}\texttt{\&> == true}$, otherwise
- $\texttt{is\_constructible\_v<A}_{ik}\texttt{,}\ \mathit{cv}_i\ \texttt{A}_{ik}\texttt{\&\&> == true}$.

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

Returns: A `tuple` object constructed by initializing the $k_i{}^{\text{th}}$ type element $\texttt{e}_{ik}$ in $\texttt{e}_i\texttt{...}$ with

$$\texttt{get<}k_i\texttt{>(std::forward<T}_i\texttt{>(tp}_i\texttt{))}$$

for each valid k_i and each group $\texttt{e}_i$ 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:

```
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:

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

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

Effects: Given the exposition-only function:

```
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:

```
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*]

� Relational operators [tuple.rel]

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

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

For the first overload let `UTuple` be `tuple<UTypes...>`.

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

Returns: `true` if `get<i>(t) == get<i>(u)` for all `i`, otherwise `false`. [Note: ~~For any two zero-length tuples `e` and `f`, `e == f`~~ If `sizeof...(TTypes)` equals zero, 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, Elems>...>
operator<=>(const tuple<TTypes...>& t, const UTuple& u);
```

For the second overload, `Elems` denotes the pack of types `tuple_element_t<0, UTuple>, tuple_element_t<1, UTuple>, ..., tuple_element_t<tuple_size_v<UTuple> -1, UTuple>`.

Effects: Performs a lexicographical comparison between `t` and `u`. ~~For any two zero-length tuples `t` and `u`, `t <=> u`~~ If `sizeof...(TTypes)` equals zero, 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 $\texttt{r}_{\text{tail}}$ 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 $\texttt{t}_{\text{tail}}$ (or $\texttt{u}_{\text{tail}}$) 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 descriptions that follow:

- Let `TTypes` be a pack formed by the sequence of `tuple_element_t<`i`, TTuple>` for every integer $0 \le i <$ `tuple_size_v<TTuple>`.
- Let `UTypes` be a pack formed by the sequence of `tuple_element_t<`i`, UTuple>` for every integer $0 \le i <$ `tuple_size_v<UTuple>`.

```
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`.
- `is_same_v<TTuple, decay_t<TTuple>>` is `true`.
- `is_same_v<UTuple, decay_t<UTuple>>` is `true`.
- `tuple_size_v<TTuple>` equals `tuple_size_v<UTuple>`.
- `tuple<common_reference_t<TQual<TTypes>, UQual<UTypes>>...>` denotes a type.

The member *typedef-name* `type` denotes the type `tuple<common_reference_t<TQual<TTypes>, UQual<UTypes>>...>`.

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

Constraints:

- `TTuple` is a specialization of `tuple` or `UTuple` is a specialization of `tuple`.
- `is_same_v<TTuple, decay_t<TTuple>>` is `true`.
- `is_same_v<UTuple, decay_t<UTuple>>` is `true`.
- `tuple_size_v<TTuple>` equals `tuple_size_v<UTuple>`.
- `tuple<common_type_t<TTypes, UTypes>...>` denotes a type.

The member *typedef-name* `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 = T1, class U2 = T2>
        constexpr explicit(see below) pair(U1&& x, U2&& y);
        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);

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


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

  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*]

```
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:

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

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

Constraints:

- `is_constructible_v<T1, U1>` is `true` and
- `is_constructible_v<T2, U2>` 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:

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

```
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);

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`.

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

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

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

```
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` and
- `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 `get(x)`.) This form of construction, whereby constructor arguments for `first` and `second` are each provided in a separate `tuple` object, is called *piecewise construction*.

[...]

Constraints:

- `is_assignable_v<const T1&, U1>` is `true`, and
- `is_assignable_v<const T2&, U2>` is `true`.

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

Returns: `*this`.

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

Constraints:

- `different-from<P, pair>` [range.utility.helpers] is `true`,
- `remove_cvref_t<P>` is not a specialization of `ranges::subrange`,
- `is_assignable_v<T1&, decltype(get<0>(std::forward<P>(p)))>` is `true`, and
- `is_assignable_v<T2&, decltype(get<1>(std::forward<P>(p)))>` is `true`.

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

Returns: `*this`.

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

Constraints:

- `different-from<P, pair>` is `true`,
- `remove_cvref_t<P>` is not a specialization of `ranges::subrange`,
- `is_assignable_v<const T1&, decltype(get<0>(std::forward<P>(p)))>` is `true`, and
- `is_assignable_v<const T2&, decltype(get<1>(std::forward<P>(p)))>` is `true`.

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

Returns: `*this`.

� 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 requiere `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, iter-value-type<InputIterator>>>;    // exposition only
    template<class InputIterator>
    using iter-mapped-type =
    typename iterator_traits<InputIterator>::value_type::second_type
tuple_element_t<1, iter-value-type<InputIterator>>;    // 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, iter-value-type<InputIterator>>>,
    typename iterator_traits<InputIterator>::value_type::second_type
tuple_element_t<1, iter-value-type<InputIterator>>>;   // exposition only
```

� 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> && !is_reference_v<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> && N < tuple_size_v<T>;
     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]

Class template `zip_view` [range.zip.view]

```
namespace std::ranges {
    template<class... Rs>
    concept zip-is-common =                             // exposition only
    (sizeof...(Rs) == 1 && (common_range<Rs> && ...)) ||
    (!(bidirectional_range<Rs> && ...) && (common_range<Rs> && ...)) ||
    ((random_access_range<Rs> && ...) && (sized_range<Rs> && ...));

    template<class... Ts>
```

```
~~using tuple-or-pair = see below; // exposition only~~

template<class F, class Tuple>
constexpr auto tuple-transform(F&& f, Tuple&& tuple) { // exposition only
    return apply([&]<class... Ts>(Ts&&... elements) {
        return tuple-or-pair<invoke_result_t<F&, Ts>...>(
        invoke(f, std::forward<Ts>(elements))...
        );
    }, std::forward<Tuple>(tuple));
}
```

[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

[1] Sy Brand. P2374R0: views::cartesian_product. https://wg21.link/p2374r0, 5 2021.

[2] Corentin Jabot. P2164R5: views::enumerate. https://wg21.link/p2164r5, 6 2021.

[3] B. Kosnik and M. Austern. N2270: Incompatible changes in c++0x. https://wg21.link/n2270, 4 2007.

[4] Alisdair Meredith. N2533: Tuples and pairs. https://wg21.link/n2533, 2 2008.

[5] Tim Song. P2116R0: Remove tuple-like protocol support from fixed-extent span. https://wg21.link/p2116r0, 2 2020.

[6] Tim Song. P2321R1: zip. https://wg21.link/p2321r1, 4 2021.

[N4901] Thomas Köppe *Working Draft, Standard for Programming Language C++*
https://wg21.link/N4901