Comparing pairs and tuples

Document #: D2165R0
Date: 2020-05-17
Project: Programming Language C++
Audience: LEWG
Reply-to: Corentin Jabot <corentin.jabot@gmail.com>

Abstract

We propose to make tuples of 2 elements and pairs comparable

Tony tables

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

Motivation

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

Notably a tuple can be constructed and assigned fom a pair. However, `tuple` and `pair` are not comparable. This proposal fixes that.

This makes tuple more consistent (assignment and comparison usually form a pair, at least in regular-ish types), and makes the library ever so slightly less surprising.

Design

Because `tuple` is already constructible from `pair`, and to avoid inter-dependencies between `<utility>` and `<tuple>`, we propose to add the comparison operator in the `<tuple>` header.

The design of these new operators for comparing a tuple and a pair is similar to the operators for comparing a pair of tuples.

Proposal

Wording

� Header <tuple> synopsis [tuple.syn]

[...]

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

template<class... TTypes, class... UTypes>
requires (sizeof..(UTyes) == 2)
constexpr bool operator==(const tuple<TTypes...>& t, const pair<UTypes...>& u);

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

template<class... TTypes, class... UTypes>
requires (sizeof..(UTyes) == 2)
constexpr common_comparison_category_t<synth-three-way-result<TTypes, UTypes>...>
operator<=>(const tuple<TTypes...>& t, const pair<UTypes...>& u);

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

� Relational operators [tuple.rel]

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

template<class... TTypes, class... UTypes>
requires (sizeof..(UTyes) == 2)
constexpr bool operator==(const tuple<TTypes...>& t, const pair<UTypes...>& u);
```

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

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

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

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

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

```
if (auto c = synth-three-way(get<0>(t), get<0>(u)); c != 0) return c;
return t_tail <=> u_tail;
```

where $\mathtt{r}_{\text{tail}}$ for some tuple `r` is a tuple containing all but the first element of `r`.

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

```
template<class... TTypes, class... UTypes>
requires (sizeof...(UTyes) == 2)
constexpr common_comparison_category_t<synth-three-way-result<TTypes, UTypes>...>
operator<=>(const tuple<TTypes...>& t, const pair<UTypes...>& u);
```

Effects: Performs a lexicographical comparison between `t` and `u`.

Equivalent to:

```
if (auto c = synth-three-way(get<0>(t), get<0>(u)); c != 0)
    return c;
return  synth-three-way(get<1>(t), get<1>(u));
```

Acknowledgments

References

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