Document number: P3786R1
Date: 2025-10-05
Project: Programming Language C++
Audience: LEWG
Reply-to: Michael Florian Hava[1] <mfh.cpp@gmail.com>

Tuple protocol for fixed-size span

Abstract
This paper proposes amending fixed-size `span`s with the tuple protocol, enabling structured binding, integration with `views::elements` and pattern matching once it is approved.

Tony Table

Before	Proposed

`span<int, 3> s{…};`
 	
`auto & x{s[0]};`
`auto & y{s[1]};`
`auto & z{s[2]};`
	`span<int, 3> s{…};`
	
	`auto & [x, y, z]{s};`

`vector<span<int, 3>> ss{…};`
 	
`auto firsts{ss | views::transform(auto s) {`
`  return s[0];`
`  })`
`  | ranges::to<vector>()};`
 	
`//NOTE: views::transform returns a copy 😬`
	`vector<span<int, 3>> ss{…};`
	
	`auto firsts{ss | views::elements<0>`
	`  | ranges::to<vector>()};`
	
	`//NOTE: views::elements returns by reference 😊`

❌ `span is not compatible with pattern matching`	`//interaction with pattern matching proposal P2688R5`
	
	`span<double, 2> p{…};`
	
	`p match {`
	`  [0, 0]  => std::println("at origin");`
	`  [let x, 0] => std::println("on x-axis at {}", x);`
	`  [0, let y] => std::println("on y-axis at {}", y);`
	`  let [x, y] => std::println("at {}, {}", x, y);`
	`};`

Revisions
R0: Initial version

R1: Fixed design issue kindly pointed out by Tomasz Kamiński.

Motivation
The *tuple-protocol* has been introduced in C++11 and has been supported for `array`, `tuple` and `pair` ever since. With structured bindings (C++17) this protocol was made an integral customisation point for users to tap into a language feature - something that is bound to become even more important with the introduction of pattern matching in a future standard.

Support for the *tuple-protocol* has been applied to `ranges::subrange` and `complex` in C++20 and C++26 respectively. At the time of writing the only fixed-size library types that do not support structured binding are: `bitset`, `integer_sequence` and `span`.

[1] RISC Software GmbH, Softwarepark 32a, 4232 Hagenberg, Austria, michael.hava@risc-software.at

We can come up with rationales for why the first two in this list do not support it: the former would have to provide proxy-references, something currently not supported by structured binding, the latter is a meta-programming tool primarily used for deduction of its values. For `span` we lack such a clear rationale.

In fact P1024 already proposed this feature - together with several other useful additions and got accepted during the C++20 cycle. After its approval LWG3212 was filled, as the approved design would have resulted in `tuple_element_t<const span<T, 3>>` yielding `const T`. Per P2116 the feature was dropped from C++20.

Design Space

Given the established design of the *tuple-protocol* there is little to discuss, apart from revisiting the issue that previously lead to the removal of this feature: Our design is based on the fact that conceptually `span<T, 1>` is equivalent to `tuple<T &>`, therefore the *tuple-protocol* should be equivalent for these types as well. Which leads us to the following semantics:

- `tuple_size_v<top-level-cv span<cv T, N>>` == `N`
- `tuple_element_t<I, top-level-cv span<cv T, N>>` == `cv T &`
- `decltype(get(span<cv T, N>))` == `cv T &`

All of which is only valid if `N != dynamic_extent`.

Note that `top-level-cv` is always ignored, staying consistent with the existing language rules for applying `cv`-qualifiers to references. In addition to the above, we adjust the exposition-only *`tuple-like`* concept to include fixed-size `spans`, enabling support for adaptors like `views::elements`.

Extending the *tuple-protocol* has one unfortunate side effect: It renders the following previously valid code ambiguous.

```
void f(span<int>) { … }
void f(tuple<int, int>) { … }



span<int, 2> s{…}:
f(s); // ! ambiguous as either user-defined conversion is valid
```

The same is already true for other *`tuple-like`* types (see: https://godbolt.org/z/r65rnPY81).

```
void f(pair<int, int>) { … }
void f(tuple<int, int>) { … }



pair<const int, int> s{…};
f(s); //❌ ambiguous as either user-defined conversion is valid

array<int, 2> a;
f(s); //❌ ambiguous as either user-defined conversion is valid
```

```
void g(complex<double>) { … }
void g(pair<float, float>) { … }

complex<float> c;
g(c); //❌ ambiguous as either user-defined conversion is valid
```

This type of breakage was already explicitly pointed out in P2165 which introduced the *`tuple-like`* concept. Therefore we don't consider this issue novel or in any way blocking to this paper.

Impact on the Standard

This proposal is a library extension changing the meaning of *`tuple-like`*`<span<T, E>>`, that may result in a breaking change for existing code.

Implementation Experience

The proposed design has been implemented on Godbolt (https://godbolt.org/z/vrKffnMWr) and by Tomasz Kamiński at https://gcc.gnu.org/pipermail/libstdc++/2025-October/063762.html.

Proposed Wording

Wording is relative to [N5014]. Additions are presented like this, removals like ~~this~~ and drafting notes like **this**.

[version.syn]

```
#define __cpp_lib_tuple_like ~~202311L~~YYYYMML //also in <utility>, <tuple>, <map>, <unordered_map>, <span>
```

[DRAFTING NOTE: Adjust the placeholder value as needed to denote the proposal's date of adoption.]

[tuple.like]

??.??.? Concept *tuple-like* **[tuple.like]**

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

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

(1.1) — `array`, `complex`, `pair`, `tuple`, `ranges::subrange`, or

(1.2) — `span` and `remove_cvref_t<T>::extent` is not equal to `dynamic_extent`.

[views.contiguous]

??.?.?.? Header synopsis **[span.syn]**

```
// mostly freestanding
namespace std {
…
  // [views.span], class template span
…
  template<class ElementType, size_t Extent>
    constexpr bool ranges::enable_borrowed_range<span<ElementType, Extent>> = true;

  // [span.tuple], tuple interface
  template<class T> struct tuple_size;
  template<size_t I, class T> struct tuple_element;
  template<class ElementType, size_t Extents>
    struct tuple_size<span<ElementType, Extents>>;
  template<size_t I, class ElementType, size_t Extents>
    struct tuple_element<I, span<ElementType, Extents>>;
  template<size_t I, class ElementType, size_t Extents>
    constexpr ElementType& get(span<ElementType, Extents>) noexcept;

  // [span.objectrep], views of object representation
…
}
```

??.?.?.? Class template span **[views.span]**

…

??.?.?.?.? Iterator support **[span.iterators]**

…

```
constexpr reverse_iterator rend() const noexcept;
```

6 *Effects*: Equivalent to: `return reverse_iterator(begin());`

??.?.?.? Tuple interface **[span.tuple]**

```
template<class ElementType, size_t Extents>
  struct tuple_size<span<ElementType, Extents>> : integral_constant<size_t, Extents> {};
```

1 *Constraints*: `Extents != dynamic_extents` is true.

```
template<size_t I, class ElementType, size_t Extents>
  struct tuple_element<I, span<ElementType, Extents>> {
    using type = ElementType&;
  };
```

2 *Mandates*:

(2.1) — `Extents != dynamic_extents` is `true`, and

(2.2) — `I < Extents` is `true`.

```
template<size_t I, class ElementType, size_t Extents>
  constexpr ElementType& get(span<ElementType, Extents> s) noexcept;
```

3 *Mandates*:

(3.1) — `Extents != dynamic_extents` is `true`, and

(3.2) — `I < Extents` is `true`.

4 *Effects*: Equivalent to: `return s[I];`

??.?.?.? Views of object representation **[span.objectrep]**

[tuple.helper]

??.?.? Tuple helper classes **[tuple.helper]**

...

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

...

6 In addition to being available via inclusion of the <tuple> header, the two templates are available when any of the headers <array> (**[array.syn]**), <ranges> (**[ranges.syn]**), (**[span.syn]**), or <utility> (**[utility.syn]**) are included.

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

...

8 In addition to being available via inclusion of the <tuple> header, the two templates are available when any of the headers <array> (**[array.syn]**), <ranges> (**[ranges.syn]**), (**[span.syn]**), or <utility> (**[utility.syn]**) are included.

[depr.tuple]

D.?? Tuple **[depr.tuple]**

...

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

...

4 In addition to being available via inclusion of the <tuple> header, the two templates are available when any of the headers <array> (**[array.syn]**), <ranges> (**[ranges.syn]**), (**[span.syn]**), or <utility> (**[utility.syn]**) are included.

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

...

6 In addition to being available via inclusion of the <tuple> header, the two templates are available when any of the headers <array> (**[array.syn]**), <ranges> (**[ranges.syn]**), (**[span.syn]**), or <utility> (**[utility.syn]**) are included.

Acknowledgements

Thanks to RISC Software GmbH for supporting this work. Thanks to Tomasz Kamiński for initially pointing us to P1024 and providing ample support regarding wording and design.