Pragmatic approach to standard structural types

Document #:	P3861R0
Date:	2025-10-06
Programming Language C++	
Audience:	EWG, LEWG

Reply-to: Corentin Jabot <corentin.jabot@gmail.com>

Abstract

We propose to make `string_view`, `span`, and `tuple` structural types through compiler magic. This partially addresses FR-014-160.

Motivation

For objects to be used as template parameters, they need to:

- be structural
- be constant expressions and therefore not perform transient allocations.

For a type to be structural, we would like to ensure that the structural representation is always canonical. There is also a desire for that canonicalization to be enforced by the compiler.

These are not trivial problems to solve, and while it seems we might converge on a solution in the next few years, the reflection facilities added in C++26 would benefit from more standard types being structural. (Mateusz Pusz's The 10 Essential Features for the Future of C++ Libraries talk gives a great overview of use cases).

For now, structural class types are limited to types whose data members are public. Of course, there have been attempts to fix it in the library. For example P3094R6 [1] proposes to add a new string type that would satisfy the current requirements for structural types.

A previous effort to make standard types structural was P2484R0 [5]. This proposes to add a defaulted operator to some standard types, which would make some types (that would be structural if not for having private members) structural, and therefore usable as a constant template parameter.

The flaw in that proposal is that it assumes that the proposed operator would be necessary and sufficient to be generalized to arbitrary user types in a way where canonicalization can be enforced. Which is an arguably risky bet.

There is a simpler first step: designate some standard types as structural. We know they can be structural because we specify them, we know what we want the structural equivalence to be, and we know they are canonical because we trust implementers. As the changes proposed here only works private members, they will be replaceable by a more general solution in the

future without having to keep special cases around forever (of course, in implementations, their structural representation will have to remain identical for abi compatibility).

`string` and `vector`

For `string` and `vector` to be usable as constant template parameters, they first need to support non-transient allocation. This is also a hard problem to solve in the general case (as illustrated by P2670R1 [2] and P1974R0 [6]). However, we can narrow the problem to `basic_string` and `vector`s whose elements themselves are structural. In the same way that we can easily allow some standard types to be structural, we can allow some containers to have non-transient allocations. This is covered by P3554R0 [4], so it is not explored further here. Adopting P3554R0 [4] would be a prerequisite to making `vector` and `string` structural - this feature is fairly critical to bridge the gap between reflection, packs, and structured binding (enabling better index sequence features, easily allowing range algorithms to manipulate types, etc).

`string_view` and `span` representation

As alluded to in P2484R0 [5], while `string_view` has an equality operator that compares the underlying sequence of bytes (which may or may not be sensible), both `string_view` and `span` represent a range over a specific container.

Therefore, the equivalence of these types is based on the value of the pointers rather than the range, which is self-consistent.

```
const char a[] = "Hello";
const char b[] = "Hello";
template <const char* S> struct Char   {};
template <string_view S> struct String {};

static_assert(std::same_as<Char<a>, Char<b>>); // error
static_assert(same_as<String<a>, String<b>>);  // error
```

String Literals

String literals are not usable as constant template parameters. This proposal does not make `T<string_view("")>` well-formed, but it would not prevent making that work in the future, as proposed in P3380R1 [3].

`optional` and `variant`

Unlike P2484R0 [5], we are not proposing to make `optional` and `variant` structural types. Doing so is possible, but we would have to bake in compiler optional/variant specific logic to check that their values are equivalent, which felt like going a bit too far in the direction of compiler magic. We would also need to invent ad-hoc ABIs for the representation of these types, whereas `tuple`, `string_view`, and `span` can, in effect, behave like aggregates for this purpose. Going in that direction seems valuable for `string` and `vector` - whose representation can be an array - but not necessarily for other types.

What about non-standard types?

This paper only affects standard types, in effect giving them capabilities than non-standard types can't have. However, reflection already depends on these types, `std::string_view`, `span`, etc are easily intercompatible with equivalent third party types and most importantly, what is proposed here is a stop-gap solution aimed at making reflection more useful, while we find a more comprehensive, language-level design.

Implementation

I tested the validity of that idea in a fork of clang, sometimes in the past 2 years. It seems that this branch no longer exists.

Core Wording

[temp.param] Template parameters

A constant template parameter shall have one of the following (possibly cv-qualified) types:

- a structural type (see below),
- a type that contains a placeholder type[dcl.spec.auto], or
- a placeholder for a deduced class type[dcl.type.class.deduct].

The top-level *cv-qualifier*s on the *template-parameter* are ignored when determining its type.

A *structural type* is one of the following:

- a scalar type, or
- an lvalue reference type, or
- a class type designated as being a structural type in the library clauses, or
- a literal class type with the following properties:
  - all base classes and non-static data members are public and non-mutable and
  - the types of all base classes and non-static data members are structural types or (possibly multidimensional) arrays thereof.

In every specialization `basic_string_view<charT, traits>`, the type `traits` shall meet the character traits requirements[char.traits]. [*Note:* The program is ill-formed if `traits::char_type` is not the same type as `charT`. — *end note*]

Library Wording

[string.view.template] Class template `basic_string_view`

[string.view.template.general] General

For a `basic_string_view str`, any operation that invalidates a pointer in the range

```
[str.data(), str.data() + str.size())
```

invalidates pointers, iterators, and references to elements of `str`.

The complexity of `basic_string_view` member functions is $\mathcal{O}(1)$ unless otherwise specified.

`basic_string_view<charT, traits>` is a trivially copyable type.

`basic_string_view<charT, traits>` is a structural type [temp.pre].

Two values `s1` and `s2` of type `basic_string_view<charT, traits>` are template-argument-equivalent ([temp.type]) if and only if

- `s1.size()` and `s2.size()` are template-argument-equivalent, and
- `s1.data()` and `s2.data()` are template-argument-equivalent.

[*Editor's note:* [...]]

[views] Views

[views.general] General

[views.contiguous] Contiguous access

[views.span] Class template `span`

[span.overview] Overview

`span<ElementType, Extent>` is a trivially copyable type [term.trivially.copyable.type].

`span<ElementType, Extent>` is a structural type [temp.pre]. Two values `s1` and `s2` of type `span<ElementType, Extent>` are template-argument-equivalent ([temp.type]) if and only if

- `s1.size()` and `s2.size()` are template-argument-equivalent, and
- `s1.data()` and `s2.data()` are template-argument-equivalent.

`ElementType` is required to be a complete object type that is not an abstract class type.

For a `span s`, any operation that invalidates a pointer in the range `[s.data(), s.data() + s.size())` invalidates pointers, iterators, and references to elements of `s`.

[*Editor's note:* [...]]

🞐 Tuples [tuple]

🞐 Class template `tuple` [tuple.tuple]

🞐 General [tuple.tuple.general]

If a program declares an explicit or partial specialization of `tuple`, the program is ill-formed, no diagnostic required.

If every type in `Types` is a structural type [temp.pre], `tuple<Types...>` is a tructural type. Two values `t1` and `t2` of type `tuple<Types..>` are template-argument-equivalent ([temp.type]) if and only if each pair of corresponding elements from t1 and t2 are template-argument-equivalent.

🞐 Construction [tuple.cnstr]

References

[1] Mateusz Pusz. P3094R6: std::basic_fixed_string. https://wg21.link/p3094r6, 1 2025.

[2] Barry Revzin. P2670R1: Non-transient constexpr allocation. https://wg21.link/p2670r1, 2 2023.

[3] Barry Revzin. P3380R1: Extending support for class types as non-type template parameters. https://wg21.link/p3380r1, 12 2024.

[4] Barry Revzin and Peter Dimov. P3554R0: Non-transient allocation with vector and basic_string. https://wg21.link/p3554r0, 1 2025.

[5] Richard Smith. P2484R0: Extending class types as non-type template parameters. https://wg21.link/p2484r0, 11 2021.

[6] Jeff Snyder, Louis Dionne, and Daveed Vandevoorde. P1974R0: Non-transient constexpr allocation using propconst. https://wg21.link/p1974r0, 5 2020.

[N5008] Thomas Köppe *Working Draft, Standard for Programming Language C++* https://wg21.link/N5008