Simplify the customization point for structured bindings

Timur Doumler (papers@timur.audio)

Document #:	P1096R0
Date:	2018-10-08
Project:	Programming Language C++
Audience:	Evolution Working Group

Abstract

In C++17, enabling structured bindings for a user-defined type requires implementing all of `std::tuple_element`, `std::tuple_size`, and `get`. This is unnecessarily verbose, error-prone, and hard to teach. In this paper, we show that `std::tuple_element` can safely be made optional, therefore significantly simplifying the customization point for structured bindings.

1 Motivation

Structured bindings were introduced in C++17 as a convenient syntax to name the elements of a "tuple-like" object instead of naming the whole object in an initializing declaration. Since then, structured bindings became one of the most popular features of C++17.

Several extensions to this syntax have already been proposed, such as letting them introduce a pack [P1061R0] and being able to mark them as `static` and `constexpr` [P1091R0]. Various other extensions are possible. We expect usage of structured bindings to increase in the future.

Ideally, any type that is conceptually just a set of elements could be used with the structured bindings syntax. Many types would greatly benefit from adding structured bindings support, for example `std::complex` and `std::chrono::year_month_day`. Many more such types exist in user code outside of the standard library.

Unfortunately, enabling structured bindings for such a type today requires implementing all of `std::tuple_element`, `std::tuple_size`, and `get`. This customization point is unnecessarily verbose, error-prone, and hard to teach.

In this paper, we show that `std::tuple_element` can safely be made optional. The type of the tuple elements can easily be deduced from the `decltype` of `get` instead. This significantly reduces the boilerplate that a user is required to write in order to enable structured bindings for a user-defined type.

2 Proposed solution

We propose to make the presence of `std::tuple_element` optional when constructing structured bindings from a type for which `std::tuple_size` is defined. Whenever `std::tuple_element` is present for such a type, it will be used to determine the types of the bindings, exactly as today. When it is missing, instead of making the program ill-formed (situation today) the types of the bindings will now be determined from the `decltype` of the `get` expression that is used as the initializer for the binding in question (either `e.get<i>` or `get<i>(e)`, depending on which declaration of `get` is found by name lookup).

For the vast majority of use cases, this will result in the exact same types being deduced. There is one exception where a subtle difference arises: tuple-like types containing elements of reference type. Consider:

```
int n = 0;
std::tuple<int, int&> t{n, n};
auto& tref{t};
auto [i, iref] = tref;
```

When using `std::tuple_element` for determining the types of `i` and `iref`, they will be `int` and `int&` respectively, preserving the distinction between reference-type tuple elements and elements of a tuple decomposed by reference. However, when using `get` instead, this information would get lost: `decltype(std::get<0>(tref))` and `decltype(std::get<1>(tref))` are both `int&`. The ability to make this distinction was the reason that the `std::tuple_element` requirement was not removed from the original design for C++17 [P0144R2].

Now that there is more experience with using structured bindings in practice, we argue that the ability to make this distinction is only relevant in extremely rare cases. In fact, we are not aware of any such case. It is certainly not necessary for perfect forwarding, since that always forwards a reference anyway. We therefore believe that making `std::tuple_element` opt-in (and therefore modifying the current rules) is a cheap price to pay considering that it will significantly simplify structured bindings for everyone.

At the same time, we do not propose any changes to the existing `std::tuple_element` specializations for `std::tuple`, `std::pair`, and `std::array`. This ensures that our proposed change can never break, or change the behaviour of, existing C++ code.

If this proposal gets adapted, we will recommend library writers to always use the simpler customization point by just defining `std::tuple_size` and `get`, unless they have a very good reason to also define `std::tuple_element`.

An additional benefit of the change proposed here is that it makes the customization point of structured bindings more consistent with that of `std::apply`, which already today requires only `std::tuple_size` and `get` to be present. Our proposal would automatically enable structured bindings for all types that support `std::apply`, even if they do not define `std::tuple_element`.

3 Further considerations

We also considered the possibility to make `std::tuple_size` optional, to simplify the customization point even further. In theory, all the required information can be deduced from only a suitably defined `get` and the number of *declarator*s in the structured binding's *declarator-list*. In practice however, implementing this approach creates some complications for which we did not yet find satisfactory solutions. Therefore, at this time we do not propose any changes to the way `std::tuple_size` works.

4 Proposed wording

The proposed changes are relative to the C++ working paper [Smith2018].

Modify **[dcl.struct.bind]** paragraph 3 as follows:

> Otherwise, if the *qualified-id* `std::tuple_size<E>` names a complete type, the expression `std::tuple_size<E>::value` shall be a well-formed integral constant expression and the number of elements in the *identifier-list* shall be equal to the value of that expression. The *unqualified-id* `get` is looked up in the scope of `E` by class member access lookup (6.4.5), and if that finds at least one declaration that is a function template whose first template parameter is a non-type parameter, the initializer is `e.get<i>()`. Otherwise, the initializer is `get<i>(e)`, where `get` is looked up in the associated namespaces (6.4.2). In either case, `get<i>` is interpreted as a *template-id*. [*Note:* Ordinary unqualified lookup (6.4.1) is not performed. — *end note*] In either case, `e` is an lvalue if the type of the entity `e` is an lvalue reference and an xvalue otherwise. Given the type $\mathtt{T}_i$ designated by `std::tuple_element<i, E>::type` if it names a type, otherwise by the `decltype` of the initializer, variables are introduced with unique names $\mathtt{r}_i$ of type "reference to $\mathtt{T}_i$" initialized with the initializer (11.6.3), where the reference is an lvalue reference if the initializer is an lvalue and an rvalue reference otherwise. Each $\mathtt{v}_i$ is the name of an lvalue of type $\mathtt{T}_i$ that refers to the object bound to $\mathtt{r}_i$; the referenced type is $\mathtt{T}_i$.

Acknowledgements

Many thanks to Herb Sutter, Ville Voutilainen, Peter Dimov, Tony Van Eerd, Tomasz Kamiński, Arthur O'Dwyer, Jens Maurer, Barry Revzin, Gabriel Dos Reis, and Mathias Stearn for their very helpful comments.

References

[P0144R2] Herb Sutter, Bjarne Stroustrup, and Gabriel Dos Reis. Structured bindings. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0144r2.pdf, 2016.

[P1061R0] Barry Revzin and Jonathan Wakely. Structured Bindings can introduce a Pack. http://open-std.org/JTC1/SC22/WG21/docs/papers/2018/p1061r0.html, 2018.

[P1091R0] Nicolas Lesser. Extending structured bindings to be more like variable declarations. http://open-std.org/JTC1/SC22/WG21/docs/papers/2018/p1091r0.html, 2018.

[Smith2018] Richard Smith. Working Draft, Standard for Programming Language C++. https://github.com/cplusplus/draft, 2018.