Document number:	P0326R0
Date:	2016-05-28
Project:	ISO/IEC JTC1 SC22 WG21 Programming Language C++
Audience:	Evolution Working Group / Library Evolution Working Group
Reply-to:	Vicente J. Botet Escribá <vicente.botet@nokia.com>

Structured binding: customization points issues

Abstract

The Structured binding paper P0217R1 has some problems:

- it makes the core language depend on the `<utility>` header,
- it excludes usage of structs with private bitfield members, and

This paper is an attempt to solve these problems. The first approach is to add additional wording to Structured binding P0217R1 to cover with the core language dependency on the library file `<utility>`. The second one consists in changing the customization points of Structured binding to something more specific and related to product types: `product_type_size` and `product_type_get`. These functions would be either members or non-members found as `begin` / `end` so that the dependency to the library file is removed.

Concerning the bitfield member access problem, this paper proposes to remove the dependency on `tuple_element` to deduce the type of the variables or to postpone the structured binding to bitfields members until we have a proposal that allows the user to customize bitfields members.

Motivation

There are two issues.

Dependent on library

P0217R1 makes the language dependent on the customization point `std::tuple_size<T>`, which is defined in the library file `<utility>`. This file is not part of the freestanding implementation. We should avoid this kind of dependency as much as possible.

Suppose we want to customize the structured binding behavior of the following class

```
class S {
    int x, y;
public:
    // ...
};
```

P0217R1 says you must specialize `std::tuple_size<S>` and `std::tuple_element<i,E>` to do so.

Fine so far, except that you can only specialize a template if you've seen the declaration of the primary template. Users can't declare the primary template themselves (it's in namespace `std`), so they need to include the correct header, which happens to be `<utility>`.

However, `<utility>` is not required to be available in a freestanding implementation (see 17.6.1.3 table 16).

> "A freestanding implementation is one in which execution may take place without the benefit of an operating system, and has an implementation-defined set of libraries that includes certain language-support libraries"

Net result: You can't (portably) use structured bindings with customized `get()` in a freestanding implementation.

The obvious solution is to add '' to list of includes required for freestanding implementations. While the authors feel that this may be a tenable direction (and provide wording to fix this), we would strongly prefer that the committee considers our proposed alternate designs based on the which would address these issues more cleanly.

What does *tuple-like* access mean?

The standard is ambiguous on the meaning of *tuple-like* access. For the purpose of this paper we presume it to consists in requiring the following traits to be well defined `std::tuple_size<TPL>`, `std::tuple_element<I, TPL>` and the following function `std::get<I>(tpl)`. Examples of the

use of this *tuple-like* access are the functions `std::tuple_cat` and `std::apply`. The wording is not always explicit.

Note that `get<I>(tpl)` cannot be found by ADL, and so the client must either qualify it `std::get<I>(tpl)` or introduce it `using std::get`

`std::apply` makes explicit use of `std::get<I>` as in

```
return std::invoke(std::forward<F>(f), std::get<I>(std::forward<Tuple>(t))...);
```

`std::tuple_cat` is less explicit, but the authors consider that this is an editorial issue as `get<ki>` intent is not find `get` it by ADL and only the `std` namespace is associated.

Returns: A tuple object constructed by initializing the kith type element `eik` in `ei...` with `get<ki>(std::forward<Ti>(tpi))` for each valid `ki` and each group `ei` in order.

It is not clear for the authors if this *tuple-like* access is currently a customization point and if a user can specialize these traits and overload the `std::get` function inside the std namespace.

Note that the proposed structure binding customization point is not aligned to the current *tuple-like* access, as the user can define the `e.get<I>()` function as a member or as a free function `get<I>(e)` associated to the namespace of the type of the expression `e`. This mean that functions like `std::apply` and `std::tuple_cat` will not work for types supporting structured binding, even for those customized by the user. As consequence we will need to update the definition of these functions. This paper doesn't attempt to solve this issue, just identifies it to reinforce the fact that we will need to change the definition of these functions anyway and replace the *tuple-like* access interface by a *product type* access interface (see Product Types P0327R0).

Alternative customization approach

Note that changing the customization point would mean that types that conform to the not so explicit *tuple-like* access will not support structured binding.

Independence from library

In order to overcome the library dependency we need to find a way to avoid the use of `std::tuple_size<T>` and `std::tuple_element<I, T>`.

For the size we have 2 possibilities:

- A member function `product_type_size` or non-member function `product_type_size` found in the namespace of the tuple expression.
- Deduce the size from the customization point for the access, that is, deduce the tuple size as `N` for which `product_type_get<I>(tpl)` is well defined for any `I` in `0..N(-1)` and

`product_type_get<N>(tpl)` is not defined.

member function `product_type_size` and non-member function `product_type_size`

This follows the the same design as the customization to get the element and the same design to customize `begin` / `end` for range-based for loops.

This seems much simpler.

Base it on the `product_type_get` customization point

This reduce the work done by the user, but determining the size could be expensive at compile time.

If `pt.product_type_get<I>()` or `product_type_get<I>(pt)` is well defined for all `I` in `0..(N-1)` and `pt.product_type_get<N>()` or `product_type_get<N>(pt)` is not well defined then the size is `N`.

This is not proposed by the paper, but added for completion.

A combination of all the previous

Let the user define member or non-member functions of `product_type_size` and `product_type_get`.

We consider that the user has customized his class when

- either `product_type_size` is customized and the size is the result of the customized expression, say `N` and `pt.product_type_get<I>()` or `product_type_get<I>(pt)` is well defined for all `I` in `0..(N-1)`,
- either `product_type_size` is not customized and If `pt.product_type_get<I>()` or `product_type_get<I>(pt)` is well defined for all `I` in `0..(N-1)` and `pt.product_type_get<N>()` or `product_type_get<N>(pt)` is not well defined then the size is `N`..

This is not proposed by the paper, but added for completion.

Ability to work with bitfields only partially

P0217R1 supports bitfields in case 3, where the compiler is able to identify the data members. However the wording for the customization case, makes use of `std::tuple_element` to define the type of the elements of the lvalues.

> Given the type `Ti` designated by `std::tuple_element<i-1,E>::type`, each `vi` is a variable of type "reference to `Ti`" initialized with the initializer, where the reference is an lvalue

> reference if the initializer is an lvalue and an rvalue reference otherwise; the referenced type is `Ti`.

This works well as far as the type returned by `get<I,e>` is a reference to `std::tuple_element_t<I,E>`, but it doesn't work at all for bitfields.

However, when a user wants to customize a class with bitfields members would need to define `std::tuple_element` for this bitfield member. However, we don't have a good candidate for the real type of the member. The function `get` could return an instance of a class `bitfield_ref`, but the `bitfield_ref` returned by `get` cannot be reference to a `std::tuple_element<i,E>` as it would be an rvalue.

What if it is identical to the type returned by `get<I>(pt)` or `pt.get<I>()`?

Nevertheless, while returning a `bitfield_ref` seem to work, we are unable to define `tuple_element<i,E>` for a bitfield member and the way we could make use of a bitfield member and non-bitfield member wouldn't be homogeneous. For example:

```
template <size_t I, class X>
class bitfield_ref;

struct X3 {
  unsigned i:2;
  int j:5;
  int k;
public:

};

template <>
class bitfield_ref<0,X3> {
  X3& x;
public:
  bitfield_ref(X3& x) : x(x) {}
  operator unsigned() const { return x.i; }
  bitfield_ref& operator=(int v)
  {
    x.i = v;
    return *this;
  }
};
template <>
class bitfield_ref<1,X3> {
  X3& x;
public:
  bitfield_ref(X3& x) : x(x) {}
  operator int() const { return x.j; }
  bitfield_ref& operator=(int v)
```

```
  {
    x.j = v;
    return *this;
  }
};
// Something similar for const& and &&, but without assignment
// ...
namespace std {
  template <>
  class tuple_size<X3> : integral_constant<size_t, 3> {};
  template <>
  class tuple_element<0,X3> { public: using type = unsigned; };
  template <>
  class tuple_element<1,X3> { public: using type = int; };
  template <>
  class tuple_element<2,X3> { public: using type = int; };
}
bitfield_ref<0, X3> get_element(std::integral_constant<size_t, 0>, X3 & x) {
  return bitfield_ref<0, X3>(x);
}
bitfield_ref<1, X3> get_element(std::integral_constant<size_t, 1>, X3 & x) {
  return bitfield_ref<1, X3>(x);
}
int& get_element(std::integral_constant<size_t, 2>, X3 & x) {
  return x.k;
}
template <size_t I>
auto get(X3 & x) {
  return get_element(std::integral_constant<size_t, I>{} , x);
}

// Something similar for const& and &&
// ...
```

Given

```
    X3 x {0,1,2};
```

the following couldn't compile for a bitfield member as the result is a rvalue, not a real reference

```
      auto &xi = get<0>(x);
```

while the following compiles for a non bitfield member

```
    auto &xk = get<2>(x);
```

The proposed P0327R0 access interface for *product types* doesn't handle this case either.

Proposal

What follows are our proposed solutions to the aforementioned problems.

We name *product type* the types covered by Structured binding.

P0327R0 is an extension paper to this proposal that includes *product type* access library interface.

Alternative proposal 1.1

Let the core language depend on an additional library file and add this file to the freestanding implementation.

Currently the traits `std::tuple_size` and `std::tuple_element` are defined in the `<utility>` file. In order to reduce the freestanding implementation constraints, we proposes to move these traits to a `<tuple_like>` file.

Alternative proposal 1.2

Change the customization points of Structured binding to something more specific and related to the types supporting structured binding. Let me call them *product types*: `product_type_size` and `product_type_get` either as members or non-members functions found by ADL so that we remove the dependency from the library file.

The authors believe that as we need to replace the customization point for `std::tuple_size` it is worth changing the `get<I>` customization point also to make the design more coherent.

Alternative proposal 2.0

Status-quo: Be able to manage with bitfield members in case 3 even if the user is unable to customize them and if we cannot take the reference to a bitfield members.

Alternative proposal 2.1

Adapt the structured binding wording to make it possible to customize the bitfield members even if we cannot take the reference to a bitfield members.

Alternative proposal 2.2

Postpone the support bitfield members in any case until we have a bitfield members references.

Design Rationale

Why the language core shouldn't depend on the library files?

The current C++ standard depends already on the library files at least for `<initializer_list>`. Adding more dependencies will open the door to more dependencies. This makes the freestanding implementations more library dependent.

The file `<utility>` contains a lot of things and could contain even more. Adding anything to this file in the future would need to check a freestanding implementation shall provide it or not.

What do we loss by changing the current customization point?

There are not many classes providing a *tuple-like* access on the standard and they can be adapted easily. However we don't know on the user side.

What do we gain by changing the current customization point?

We don't increase the dependency of the core language on the library.

The current *tuple-like* access `tuple_size` / `tuple_element` / `get` has a customization point `get` that is used also for other types that don't provide a *tuple-like* access. There is no real problem as the other customization points are more specific.

Adopting the *product type* customization points `product_type_size` / `product_type_get` are more explicit and in line with *product type* access P0327R0 .

What do we loss if Structured binding is not able to bind to bitfield members?

There is not a lot we loss. The user can always nest the bitfields on a nested structure and use the bitfield access.

Instead of

```
class X3 {
  unsigned i:2;
  int j:5;
  int k;
  // friend declarations as needed
public:
  ...
};
```

the user can define

```
class X3 {
public:
  struct IJ {
    unsigned i:2;
    int j:5;
  };
private:
  IJ ij;
  int k;
  // friend declarations as needed
public:
  ...
};
```

What do we gain if Structured binding don't support bitfield members?

There is much we still do not know about bitfields. If we handle them now, we may be painting ourselves in a corner later.

Excluding their support makes it possible to have a uniform interface. We would be able to name the exact type for any element and have access to the element via a reference to the member data.

If a uniform solution is found later on that support bitfields references we could always update the proposal in a backward compatible way.

Wording

Alternative 1

In 7.1.6.4 [dcl.spec.auto] paragraph 8 of the Structured Binding proposal, replace

In either case, `e` is an lvalue if the type of the entity `e` is an lvalue reference and an xvalue otherwise. Each `v` `i` is a variable of type `decltype(pt_size)` initialized with the initializer.

Alternative 1.1

Add a new `<utility>` file in 17.6.2.2 Headers [using.headers] Table 16

Add the following to `[utility]` Header synopsis

```
namespace std {
    template <class T> class tuple_size<const T>;
    template <class T> class tuple_size<volatile T>;
    template <class T> class tuple_size<const volatile T>;

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

Alternative 1.2

Add a new `<tuple_like>` file in 17.6.1.2 Headers [headers] Table 14

Add a section Tuple like Objects in 20

** 20.X Tuple like Objects**

Header synopsis

The header defines the tuple-like traits.

```
namespace std {
    template <class T> class tuple_size; // undefined
    template <class T> class tuple_size<const T>;
    template <class T> class tuple_size<volatile T>;
    template <class T> class tuple_size<const volatile T>;

    template <size_t I, class T> class tuple_element; // undefined
    template <size_t I, class T> class tuple_element<I, const T>;
    template <size_t I, class T> class tuple_element<I, volatile T>;
    template <size_t I, class T> class tuple_element<I, const volatile T>;
}
```

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

Remarks: All specializations of `tuple_size<T>` shall meet the *UnaryTypeTrait* requirements (20.10.1) with a *BaseCharacteristic* of `integral_constant<size_t, N>` for some `N`.

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

Let `TS` denote `tuple_size<T>` of the cv-unqualified type `T`. Then each of the three templates shall meet the `UnaryTypeTrait` requirements (20.10.1) with a `BaseCharacteristic` of `integral_constant<size_t, TS::value>`

In addition to being available via inclusion of the `<tuple_like>` header, the three templates are available when either of the headers `<array>` or `<utility>` or `tuple` are included.

```
    template <size_t I, class T> class tuple_element; // undefined
```

Remarks: `std::tuple_element<I,T>::type` shall be defined for all the `I` in `0..(std::tuple_size<T>::value-1)`.

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

Let `TE` denote `tuple_element<I, T>` of the cv-unqualified type `T`. Then each of the three templates shall meet the `TransformationTrait` requirements (20.10.1) with a member typedef type that names the following type:

- for the first specialization, `add_const_t<TE::type>`,
- for the second specialization, `add_volatile_t<TE::type>`, and
- for the third specialization, `add_cv_t<TE::type>`.

In addition to being available via inclusion of the header, the three templates are available when either of the headers or or are included.

Extract the following from `[utility]` Header synopsis

```
    template <class T> class tuple_size;
    template <size_t I, class T> class tuple_element;
```

Add the following to `[utility]` Header synopsis

```
#include <tuple_like>
```

Extract the following from `[tuple.general]` Header synopsis

```
    template <class T> class tuple_size; // undefined
    template <class T> class tuple_size<const T>;
    template <class T> class tuple_size<volatile T>;
    template <class T> class tuple_size<const volatile T>;

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

Add the following to `[tuple.general]` Header synopsis

```
#include <tuple_like>
```

Rename 20.4.2.5 Tuple helper classes as Tuple Tuple-like configuration.

Remove from 20.4.2.5 the definition for tuple*size and tuple*element 0 3,4, 5 and 6

Add a new `<tuple_like>` file in 17.6.2.2 Headers [using.headers] Table 16

Wording Alternative 2

In 7.1.6.4 [dcl.spec.auto] paragraph 8 of the Structured Binding proposal, replace

Otherwise, let *pt_size* be defined as follows. The unqualified-id `product_type_size` is looked up in the scope of `E` by class member access lookup (3.4.5 [basic.lookup.classref]), and if that finds at least one declaration, then *pt_size* is `e.product_type_size()`. Otherwise, then *pt_size* is `product_type_size(e)`, where `product_type_size` is looked up in the associated namespaces (3.4.2 [basic.lookup.argdep]). [Note: Ordinary unqualified lookup (3.4.1 [basic.lookup.unqual]) is not performed. -- end note].

If the expression *pt_size* is a well-formed integral constant expression, the number of elements in the identifier-list shall be equal to the value of that expression. The unqualified-id `product_type_get` is looked up in the scope of `E` by class member access lookup (3.4.5 [basic.lookup.classref]), and if that finds at least one declaration, the initializer is `e.product_type_get<i-1>()`. Otherwise, the initializer is `product_type_get<i-1>(e)`, where `product_type_get` is looked up in the associated namespaces (3.4.2 [basic.lookup.argdep]). [Note: Ordinary unqualified lookup (3.4.1 [basic.lookup.unqual]) 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. Each `v` `i` is a variable of type `decltype(pt_size)` initialized with the initializer.

Add the associated customization in `[tuple.tuple]`

Class template tuple

```
    ...
    constexpr size_t product_type_size() { return sizeof...(Ts); };
    template <size_t I>
    constexpr auto product_type_get();
    template <size_t I>
    constexpr auto product_type_get() const;
    template <size_t I>
    constexpr auto product_type_get() &&;
    template <size_t I>
    constexpr auto product_type_get() const &&;
```

std::array

```
template <class T, size_t N>
class array {
    ...
    constexpr size_t product_type_size() { return N; };
    template <size_t I>
    constexpr auto product_type_get();
    template <size_t I>
    constexpr auto product_type_get() const;
    template <size_t I>
    constexpr auto product_type_get() &&;
    template <size_t I>
    constexpr auto product_type_get() const &&;
};
```

Open Questions

The authors would like to have an answer to the following points if there is any interest at all in this proposal:

- Do we want the core language depend on the file library?
- If yes, do we prefer to move to a `<tuple_like>` file?
- If not,
- Do we want the proposed customization points?
- Do we want customization points for *product type* size to be optional?

Future work

Extend the default definition to aggregates

With P0017R1 we have now that we can consider classes with non-virtual public base classes as aggregates. P0197R0 considers the elements of the base class as elements of the *tuple-like* type. I would expect that all the aggregates can be seen as *tuple-like* types, so we need surely to consider this case in P0217R1 and P0197R0.

We should see aggregate initialization and structured binding almost as inverse operations.

This could already be the case for predefined *tuple-like* types which will have aggregate initialization. However user defined *tuple-like* types would need to define the corresponding constructor.

Acknowledgments

Thanks to Jens Maurer, Matthew Woehlke and Tony Van Eerd for their comments in private discussion about structured binding and product types.

Thanks to all those that have commented the idea of a tuple-like generation on the std-proposals ML better helping to identify the constraints, in particular to J. "Nicol Bolas" McKesson, Matthew Woehlke and Tim "T.C." Song.

Thanks to David Sankel for revising the last version carefully.

References

- Boost.Fusion Boost.Fusion 2.2 library

  http://www.boost.org/doc/libs/1*60*0/libs/fusion/doc/html/index.html

- Boost.Hana Boost.Hana library

  http://boostorg.github.io/hana/index.html

- N4527 Working Draft, Standard for Programming Language C++

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4527.pdf

- P0017R1 Extension to aggregate initialization

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0017r1.html

- P0144R2 Structured Bindings

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0144r2.pdf

- P0197R0 Default Tuple-like Access

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0197r0.pdf

- P0217R1 Proposed wording for structured bindings

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0217r1.html

- P0311R0 A Unified Vision for Manipulating Tuple-like Objects

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0311r0.html

- P0327R0 Product types access

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0327r0.pdf

- DSPM C++ Language Support for Pattern Matching and Variants

  http://davidsankel.com/uncategorized/c-language-support-for-pattern-matching-and-variants