DRAFT

Document number:	D0XXXR0
Date:	2016-05-01
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: alternative design for customization points

Abstract

This paper proposes to either add additional wording to Structured binding P0217R1 to cover with the core language dependency on the library file `<utility>` or to change the customization points of Structured binding to something more specific and related to product types: `product_type_size` and `product_type_get` either as members or non-members found by ADL so that we remove the dependency from the library file.

Introduction

This paper proposes to either add additional wording to Structured binding P0217R1 to cover with the core language dependency on the library file `<utility>` or to change the customization points of Structured

binding to something more specific and related to product types: `product_type_size` and `product_type_get` either as members or non-members found by ADL so that we remove the dependency from the library file.

The wording has been modified so that both structured binding and the possible product type access PT wording doesn't repeat themselves. An alternative could be to preserve the structure of the current wording P0217R1 and let the refactoring of the wording to a future product type access proposal PT.

Motivation

Almost Status-quo

P0144R2 proposes the ability to bind all the members of a *extended tuple-like* type at a time via the new structured binding statement which has some primitive matching for some specific cases and let the user customize their types using the case 2 with the current ad-hoc *tuple-like* interface.

P0144R2 case 1 takes care of c-array (wondering if we cannot define already a *tuple-like* access for them that can be found by ADL).

P0144R2 case 3 support structured binding to bitfields as it does for any other non-static public member.

This means that with P0144R2 we will be able to access the members of some *extended tuple-like* types that would not have *tuple-like* access. P0197R0 proposes the generation of the *tuple-like* access function for simple structs as P0144R2 does for simple structs (case 3 in P0144R2). However we are unable to define the `get<N>(tl)` function to access bitfields. So we cannot have *tuple-like* access for all the types supported by P0217R1. This is unfortunately asymmetric. We want to have structured binding, pattern matching and *tuple-like* access for the same types.

This means that the *extended tuple-like* access cannot be limited to the current *tuple-like* customization points. We need something different.

In addition, 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.

Adopting P0144R2 as such would mean that

- we accept a dependency on the library file `<utility>`, and
- we would be unable to have *tuple-like* access for all the types covered by P0144R2.

Accepting the previous points would at least mean that we need to change the freestanding implementation requirements and that the *extended tuple-like* access is based on the structured binding statement instead of the ad-hoc *tuple-like* access, and that we cannot access the size or the n^{th} element directly and

independently. While the authors feel that this may be a tenable direction, we would strongly prefer that the committee consider our proposed alternate design which would address these issues more cleanly.

Alternative design

Not accepting any of the previous points would mean a change on the definition of *tuple-like* access.

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>`. We have 3 possibilities:

- A non-member function `product_type_size` that must be found by ADL.
- A member function `product_type_size`.
- Use `product_type_get<N>(tpl)` as customization point and 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.

non-member function `product_type_size`

We could think of a non-member function `product_type_size` that must be found by ADL. However `product_type_size<T>()` couldn't be found by ADL. We need to add the type on as a parameter.

We could look for `product_type_size(T)` but this would be restricted to copyable types. We could look for `product_type_size(T const&)` but this would accept types inheriting from `T`. We could use a nullary function that returns a pointer to `T` and look for `product_type_size(T*(*)())`. This has the advantage that it doesn't accept derived types and works for any type.

member function `product_type_size`

This seems much simpler. However it prevents the customization of 3PP classes.

non-member function `product_type_get<I>(tpl)`

This also seems much simpler, but determining the size could be expensive at compile time.

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 `PT::product_type_size()` or

`product_type_size([]()->PT* {return nullptr})` and the result is `N` and `pt.product_type_get<I>()` or `product_type_get<I>(pt)` is well defined for all `I` in `0..(N-1)`,
- either the member or non-member functions for `product_type_size` and the `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,

Ability to work with bitfields

To provide *extended tuple-like* access for all the types covered by P0144R2 which support getting the size and the n^{th} element, we need to define some kind of predefined operators `pt_size(T)` / `pt_get(N, pt)` that could use the new *product type* customization points. The use of operators, as opposed to pure library functions, is particularly required to support bitfield members.

A function interface could also be provided as soon as we have a `bitfield_ref` class.

Parameter packs

We shouldn't forget parameter packs, which could be seen as being similar to product types. Parameter packs already have the `sizeof...(T)` operator. Some (see e.g. P0311R0 and references therein) are proposing to have a way to explicitly access the n^{th} element of a pack (a variety of possible syntaxes have been suggested). The authors believe that the same operators should apply to parameter packs and product types.

Proposal

Taking into consideration these points, this paper proposes two alternative proposals for the *product type* customization points for Structured binding.

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

Alternative proposal 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 file.

Alternative proposal 2

Change the customization points of Structured binding to something more specific and related to 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.

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.

What do we loss by changing the current customization point?

There are not too much 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?

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 PT.

Wording

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

Add the following Product type section

Product types terms

A type `E` is a *product type* if the following terms are well defined.

product type size

- If E is an array type with element type T, equal to the number of elements of E.
- Else, if the expression `e.product_type_size()` is a well-formed integral constant expression, equal to `e.product_type_size()`.
- Else, if all of E's non-static data members and bit-fields shall be public direct members of E or of the same unambiguous public base class of E, E shall not have an anonymous union member, equal to the number of non-static data members of E.
- Else it is undefined.

product type i th-element

- If the *product type size* of `E` is defined and `i` is less than the *product type size* of `E`.
    - If `E` is an array type with element type `T`, equal to `e[i]`.
    - Else, if the expression `e.product_type_size()` is a well-formed integral constant

expression, equal to the following: 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 value is `e.product_type_get<i-1>()`. Otherwise, the value 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].
    - else, if all of `E`'s non-static data members and bit-fields shall be public direct members of `E` or of the same unambiguous public base class of `E`, `E` shall not have an anonymous union member, equal to `e.mi` where `i`-th non-static data member of `E` in declaration order is designated by `mi`.
    - Else it is undefined.

- Else it is undefined.

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

If `E` is an array type with element type `T`, the number of elements in the identifier-list shall be equal to the number of elements of `E`. Each v_i is the name of an lvalue that refers to the element `i-1` of the array and whose type is `T`; the referenced type is `T`. [Note: The top-level cv-qualifiers of `T` are cv. -- end note]

Otherwise, if the expression `std::tuple_size<E>::value` 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 `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.get<i-1>()`. Otherwise, the initializer is `get<i-1>(e)`, where `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. Given the type T_i designated by `std::tuple_element<i-1,E>::type`, each v_i is a variable of type "reference to T_i" 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 T_i.

Otherwise, all of `E`'s non-static data members and bit-fields shall be public direct members of `E` or of the same unambiguous public base class of `E`, `E` shall not have an anonymous union member, and the number of elements in the identifier-list shall be equal to the number of non-static data members of `E`. The `i`-th non-static data member of `E` in declaration order is designated by m_i. Each v_i is the name of an lvalue that refers to the member m_i of `e` and whose type is cv T_i, where T_i is the declared type of that member; the referenced type is cv T_i. The lvalue is a bit-field if that member is a bit-field.

with

The number of elements in the identifier-list shall be equal to *product type size of* `E`.

Each `v` `i` is the name of an lvalue that refers to *product type* `i` *th-element of* `e`.

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 &&;
};
```

Implementability

There is no implementation as of the date of this paper.

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.

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

- N4381 Suggested Design for Customization Points

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4381.html

- N4387 Improving pair and tuple, revision 3

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4387.html

- N4428 Type Property Queries (rev 4)

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4428.pdf

- N4451 Static reflection

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4451.pdf

- N4475 Default comparisons (R2)

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4475.pdf

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

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

- N4532 Proposed wording for default comparisons

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4532.html

- P0017R1 Extension to aggregate initialization

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

- P0091R1 - Template argument deduction for class templates (Rev. 4)

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0091r1.html

- P0144R2 Structured Bindings

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

- P0151R0 Proposal of Multi-Declarators

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0151r0.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.pdf

- P0311R0 A Unified Vision for Manipulating Tuple-like Objects

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

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

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

- PT Product types

  https://github.com/viboes/std-make/blob/master/doc/proposal/reflection/ProductTypes.md