DRAFT

Document number:	D0XXXR0
Date:	2016-04-25
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>

Product types: about structure binding and tuple-like access

Abstract

This paper proposes 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.

In addition, it proposes to add some product type specific operators to get the size and the n^{th} element as well as some specific traits and functions.

The wording has been modified so that both structured binding and product type access wording doesn't repeat themselves.

Motivation

Status-quo

[P0144R1] proposes the ability to bind all the members of a *extended tuple-like* type at a time via the new structure 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.

[P0144R1] 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).

[P0144R1] case 3 support structure binding to bitfields as it does for any other non-static public member.

This means that with [P0144R1] we will be able to access the members of some *extended tuple-like* types that would not have a *tuple-like* access. P0197R0 proposes the generation of the *tuple-like* access function for simple structs as the [P0144R1] does for simple structs (case 3 in [P0144R1]). However we are unable to define the `get<N>(tl)` function to access bitfields. So we cannot have a *tuple-like* access for all the types supported by P0217R1. This is unfortunately asymmetric. We want to have structure 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 [P0144R1] 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 [P0144R1].

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 Structure 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.

The authors consider that this is an acceptable situation, but would prefer to see what the committee thinks of an alternative design.

Alternative design

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

Independence on 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 instead to look for `product_type_size(T)` but this would be restricted to copyable types. We could instead to 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.

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

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

Able to manage with bitfields

To provide *extended tuple-like* access for all the types covered by [P0144R1] which support getting the size and the nth element, we need to define some kind of predefined operators `pt_size(T)` / `pt_get<N>(pt)` that could use the new *product-type-like* customization points.

Parameter packs

We shouldn't forget parameter packs, that 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 nth 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

This paper proposes to define a new *product type* access, to cover the previous *extended tuple-like* access, on which [P0144R1] would be based. The user shall be able to customize his own types to see them as *product types* ([P0144R1] case 2).

The *product type* access is based on two operators: one `pt_size(T) to get the size and the other` pt_get(pt) to get the `N` th element of a product type instance `pt` of type `T`. The definition of these operators is based on the wording of structured binding P0217R1.

The user should of course, be able to customize his *product* types, as she already is able to do it for *tuple-*

like types, but now it should define the member operations `product_type_size()` and `product_type_get<N>()`.

The name of the operators `pt_size` and `pt_get` are of course subject to bike-shedding. This paper proposes the following names for the operators and the customization operations:

- `pt_size(PT)` = `product_type_size(PT)`
- `pt_get<N>(pt)` = `product_type_get(N,pt)`

Customization points are functions with the same name as the operators:

- `PT::pt_size()` = `PT::product_type_size()`
- `pt.pt_get<N>()` = `pt.product_type_get<N>()`
- `pt.pt_get<N>()` = `product_type_get<N>(pt)`

Note that `product_type_size` and `product_type_get` must therefore be contextual keywords.

But what would be the result type of those operators? While we can consider `product_type_size` as a function and we could say that it returns an `unsigned int`, `product_type_get(N,pt)` wouldn't be a function (if we want to support bitfields), and so `decltype(product_type_get(N,pt))` wouldn't be defined if the N^{th} element is a bitfield managed on [P0144R1] case 3. In all the other cases we can define it depending on the const-rvalue nature of `pt`.

The following could be syntactic sugar for those operators but we don't proposes them yet, waiting to see what we do with parameter packs direct access and sum types.

- `pt_size(PT)` = `sizeof...(PT)`
- `pt_get<N>(pt)` = `pt.[N]`

Caveats

`sizeof(T)`, `pt_size(T)` and `pt_get<N>(pt)` are not functions, and so they cannot be used in any algorithm expecting a function. Generic algorithms working on *product* types should take the type as a template parameter and possibly an integral constant for the indices.

Library interface

However, an alternative is to define generic functions `std::product_type::size<T>()` and `std::product_type::get<I>(pt)`.

We have two possibilities for `std::product_type::get`: either it supports bitfield elements and we need a `std::bitfield_ref` type, or it doesn't supports them.

We believe that we should provide the `bitfield_ref` class in the future, but it seems it is too late to propose such a class for C++17.

However, we could already define the functions that will work well expect for bitfields.

```
namespace std {
namespace product_type {

template <class PT>
constexpr size_t size() { return product_type_size(PT); }

template <size_t N, class PT>
constexpr auto get(PT&& pt) { return product_type_get(N, forward<PT>(pt)); } // Woul

}}
```

This means that in C++17 we could work with product types using the operators, and that we wouldn't have a complete function interface until C++20. While this could be seen as a limitation, and it would be in some cases, a lot of algorithms can be defined using the operator interfaces without any limitation. Only some algorithms would need the function interface.

Users could already define their own `bitfield_ref` class in C++17 and define its customization point for bitfields members if needed. However, the default implementation of the `pt_get` operator would never return a bitfield_reference. In order to define this function in C++20 we will need a compiler type trait `product_type_element_is_bitfield<N,PT>` that would say if the N^{th} element is a bitfield or not.

```
namespace std {
namespace product_type {

template <class PT>
size_t size() { return product_type_size(PT); }

template <size_t N, class PT>
// requires product_type_element_is_bitfield<N,PT>::value
bitfield_ref_t<PT, N> get(PT&& pt) { return bitfield_ref_t<PT, N>(pt); }

template <size_t N, class PT>
// requires ! product_type_element_is_bitfield<N,PT>::value
auto get(PT&& pt) { return product_type_get(N, forward<PT>(pt)); }

}}
```

Wording

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 *product type size* `E` is defined and `i <` *product type size* `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.

If either one of the previous macros is undefined the other is undefined also.

Defines the following operators

TBC

On the Structurd binding, 7.1.6.4 [dcl.spec.auto] paragraph 8 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*

Library

Product type object

In `<product_type>`

```
namespace std {
namespace product_type {

template <class PT>
constexpr size_t size() { return product_type_size(PT); }

template <size_t N, class PT>
constexpr auto get(PT&& pt) { return product_type_get(N, forward<PT>(pt)); }

}}
```

`product_type::size`

```
template <class PT>
static constexpr size_t size();
```

Effect: As if return *product type size* `PT`.

Remark: This operation would not be defined if *product type size* `PT`. is undefined.

`product_type::get`

```
template <size_t N, class PT>
constexpr auto get(PT&& pt);
```

Requires: `N < size<PT>()`

Effect: As if return *product type Nth-element* of `pt`.

Remark: This operation would not be defined if *product type Nth-element* of `pt` is undefined.

std::tuple

Add the associated customization

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

std::array

Add the associated customization

```
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() { return get<I>(*this); };
    template <size_t I>
    constexpr auto product_type_get() const { return get<I>(*this); };
    template <size_t I>
    constexpr auto product_type_get() && { return get<I>(*this); };
    template <size_t I>
    constexpr auto product_type_get() const && { return get<I>(*this); };

};
```

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 customization points for `PRODUCT_TYPE_SIZE` to be `PT::product_type_size()` ?
- Do we want the customization points for `PRODUCT_TYPE_GET` to be `pt.product_type_get<i>()` / `product_type_size<I>(pt)` ?
- Do we want the `PRODUCT_TYPE_SIZE` / `PRODUCT_TYPE_GET` macros?
- Do we want the `product_type_size` / `product_type_get` operators?
- Do we want the `std::product_type::size` / `std::product_type::get` functions?

Future work

Allow product type function access to bitfield references

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 structure 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 Nicol Bolas, Matthew Woehlke and T.C..

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

- [P0144R1] Structured Bindings

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0144r1.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