Document number:	P0327R3
Date:	2017-10-15
Project:	ISO/IEC JTC1 SC22 WG21 Programming Language C++
Audience:	Library Evolution Working Group
Reply-to:	Vicente J. Botet Escribá <vicente.botet@nokia.com>

Product-Type access (Revision 3)

Abstract

This paper proposes a library mechanism for deconstructing types that parallels the language mechanism described in Structured binding P0144R2. This proposal name a type concerned by structured binding a *ProductType*. The interface includes getting the number of elements, access to the nth element and the type of the nth element.

The main benefits of this are cheap reflection, allow automatic serialization support, automated interfaces, etc.

Table of Contents

Introduction

Defining *tuple-like* access `tuple_size` , `tuple_element` and `get<I>/get<T>` for simple classes is -- as for comparison operators (N4475) -- tedious, repetitive, slightly error-prone, and easily automated.

P0144R2/P0217R3 proposes the ability to bind all the members of some type, at a time via the new structured binding statement. This proposal names those types *product types*.

P0197R0 proposed the generation of the *tuple-like* access function for simple structs as the P0144R2 does for simple structs (case 3).

This paper proposes a library interface to access the same types covered by Structured binding P0144R2, *product types*. The interface includes getting the number of elements, access to the nth element and the type of the nth element. This interface doesn't use ADL.

Motivation

Status-quo

Besides `std::pair` , `std::tuple` and `std::array` , aggregates in particular are good candidates to be considered as *tuple-like* types. However defining the *tuple-like* access functions is tedious, repetitive, slightly error-prone, and easily automated.

Some libraries, in particular Boost.Fusion and Boost.Hana provide some macros to generate the needed reflection instantiations. Once this reflection is available for a type, the user can use the struct in algorithms working with heterogeneous sequences. Very often, when macros are used for something, it is hiding a language feature.

P0144R2/P0217R3 proposes the ability to bind all the members of a *tuple-like* type at a time via the new structured binding statement. P0197R0 proposes the generation of the *tuple-like* access function for simple structs as the P0144R2 does for simple structs (case 3 in P0144R2).

The wording in P0217R3, allows to do structure binding for C-arrays and allow bitfields as members in case 3 (built-in). But

- bitfields cannot be managed by the current *tuple-like* access function `get<I>(t)` without returning a bitfields reference wrapper, so P0197R0 doesn't provides a *tuple-like* access for all the types supported by P0217R3.

- we are unable to find a `get<I>(arr)` overload on C-arrays using ADL.

This is unfortunately asymmetric. We want to have structure binding, pattern matching and *product types* access for the same types.

This means that the *extended tuple-like* access cannot be limited to *tuple-like* access.

Algorithms such as `std::tuple_cat` and `std::experimental::apply` that work well with *tuple-like* types, should work also for *product* types. There are many more of them; a lot of the homogeneous container algorithm are applicable to heterogeneous containers and functions, see Boost.Fusion and Boost.Hana. Some examples of such algorithms are `swap`, `lexicographical_compare`, `for_each`, `filter`, `find`, `fold`, `any_of`, `all_of`, `none_of`, `accumulate`, `count`, ...

Other algorithms that need in addition that the *ProductType* to be also *TypeConstructible* are e.g. `transform`, `replace`, `join`, `zip`, `flatten`, ...

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 nth element, we would 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.

The authors don't know how to define a function interface that could manage with bitfield references. See P0326R0 "Ability to work with bitfields only partially" for a description of the customization issues.

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 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

Taking into consideration these points, this paper proposes a *product type* access library interface. See P0648R0 and P0649R0 for specific *ProdutTypes* algorithms and how the standard library can by generalizing *tuple-likke* type to *productType* types.

Future *Product type* operator proposal (Not yet)

We don't propose yet the *product type* operators to get the size and the nth element as we don't have a good proposal for the operators's name. We prefer to wait until we have some concrete proposal for parameter packs direct access.

The *product type* access could be based on two operators: one `pt_size(T)` to get the size and the other `pt_get(N, pt)` to get the `N` th element of a *product type* instance `pt` of type `T`. The definition of these operators would be based on the wording of structured binding P0217R3.

The name of the operators `pt_size` and `pt_get` are of course subject to bike-shedding.

But what would be the result type of those operators? While we can consider `pt_size` as a function and we could say that it returns an `unsigned int`, `pt_get(N,pt)` wouldn't be a function (if we want to support bitfields), and so `decltype(pt_get(N,pt))` wouldn't be defined if the Nth element is a bitfield managed on P0144R2 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 propose them yet. We wait 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

1. `pt_size(T)`, `pt_element(T)` and `pt_get(N, pt)` aren't functions nor traits, and so they cannot be used in any algorithm expecting a function or a traits as parameter.

2. We need to find the name for those operators.

Product type access library proposal

An alternative is to define generic function `std::product_type::get<I>(pt)` and traits `std::product_type::size<PT>::value` `std::product_type::element_t<PT>` using wording similar to that in P0217R3.

The interface tries to follow in someway the guidelines presented in N4381.

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 a `bitfield_ref` class in the future, but this is out of the scope of this paper.

However, we can already define the functions that will work well with all the *product types* expect for bitfields.

```
namespace std {
namespace product_type {

    template <class PT>
    struct size;

    // Wouldn't work for bitfields
    template <size_t N, class PT>
    constexpr auto get(PT&& pt)

    template <size_t N, class PT>
    struct element;

}}
```

While this could be seen as a limitation, and it would be in some cases, we can already start to define a lot of algorithms.

Users could already define their own `bitfield_ref` class and define its customization point for bitfields members if needed when structured binding will be updated to allow bitfield customization.

Waiting for that, the user will need to wrap the bitfields in a specific structure and do bit manipulation outside independently of the product type access.

Design Rationale

What do we loss if we don't add this *product type* access?

We will be unable to define algorithms working on the same kind of types supported by Structured binding P0144R2.

While Structured binding is a good tool for the user, it is not adapted to the library authors, as we need to know the number of elements of a product type to do Structured binding.

This means that the user would continue to write generic algorithms based on the *tuple-like* access and we don't have a *tuple-like* access for c-arrays (which could be added) and for the types covered by Structured binding case 3 P0217R3.

Can the *ProductType* interface be implemented using Reflection

Even if we can generate the implementation for some of the types Reflection, we are unable to generate the whole interface (or at least the authors don't know how todo it).

Reflection can help for arrays (case 1) and structs (case 3). However, the case 2 is more subtle. How Reflection could help to "lookup in the associated namespaces (3.4.2)"? Would this mean that the reflection interface would provide the different kind of lookup.

Traits versus functions

Should the *product type* `size` access be a constexpr function or a trait?

We have chosen a traits to be inline with *tuple-like* access. Note that the trait defines the function call

```
    auto s = product_type::size<PT>{}();
```

Note also that having a function to get the element type is not natural and its use is not friendly.

Locating the interface on a specific namespace

The name of *product type* interface, `size` , `get` , `element` , are quite common. Nesting them on a specific namespace makes the intent explicit.

We can also preface them with `product_type_` , but the role of namespaces was to be able to avoid this kind of prefixes.

Namespace versus struct

We can also place the interface nested on a struct. Using a namespace has the advantage is open for addition. It can also be used with using directives and using declarations.

Using a `struct` would make the interface closed to adding new nested functions, but it would be open by derivation.

What we surely need is an *explicit namespace* that is open for additions and that request explicit qualification. N1691 "Explicit Namespaces" suggest something like that, but goes too far.

Should we add a more specific customization point

This paper has the same customization point for product types that the structured binding supporting types.

In Toronto it was suggested that we should have a more specific way to customize the product types.

This paper doesn't proposes yet a specific customization point as it will invalidate the purpose to map product types and structured binding supporting types. If the LEWG believes this is a good idea we will need to adapt the structured binding wording.

Anyway, let me present now how we could have a customization point more specific and how it would manage with the current *tuple-like* customization point in a backward compatible way. This design is the one adopted by the library emulation in PT_impl.

Following [CUSTOM] Traits: An Alternative Design for Customization Points, we will define in namespce `product_type` a traits class.

```
namespace product_type
{

template <class PT, class Enabler = void>
struct traits;

// Default failing specialization
template <class PT, bool condition>
struct traits<PT, meta::when<condition>>
{
    template <class T>
    static constexpr auto get(T &&x) = delete;
};

// Forward to customized class using tuple-like access
template <class PT>
struct traits<PT, meta::when<has_tuple_like_access<PT>::value>>
{
    using size = tuple_size<PT>;

    template <size_t I>
    using element = tuple_element<I, PT>;

    template <size_t I, class PT2,
              class = std::enable_if_t<I<size::value>> static constexpr decltype(
                      auto) get(PT2 &&pt) noexcept
    {
        return product_type_detail::get_adl::xget<I>(forward<PT2>(pt));
    }
};

template <class T, size_t N>
struct traits<T[N]>
{
    using size = integral_constant<size_t, N>;
    template <size_t I>
    struct element
    {
        using type = T;
    };

    template <size_t I, class U, size_t M,
              class = std::enable_if_t<I<N>> static constexpr U &get(
                  U (&arr)[M]) noexcept
    {
        return arr[I];
    }
};

}
```

Where `has_tuple_like_access<PT>::value` states if the type `PT` is a *tuple-like* type.

The wording of structured binding should be adapted but the author suspect that a breaking change will be introduced respect to the way `get` is found.

Proposed Wording

The proposed changes are expressed as edits to N5131 Working Draft, Standard for Programming Language C++.

Note that the wording for the structured binding has not been changed even it could profit from the definition of the"Product types terms". This will in some sense have a duplicated wording, but the authors expect that this can be solved later on.

Add the following section

Product types terms

If `E` is an array type with element type `T`,

- the *product type size of E* is equal to the number of elements of E,
- the *product type i* th*-element of E* is `e[i-1]`,
- the *product type i* th*-element type of E* 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 *product type size of E* is equal to `std::tuple_size<E>::value` ,

If the expression `std::tuple_element<E>::type` is a well-formed type

- the *product type i* th*-element type of E* is this type.

The unqualified-id `get` is looked up in the scope of `E` by class member access lookup (3.4.5), 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). In either case, `get<i - 1>` is interpreted as a template-id. [Note: Ordinary unqualified lookup (3.4.1) is not performed. — end note]

- the *product type i* th*-element of E* is this initializer

Otherwise, all of `E` 's non-static data members 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. The `i` th non-static data member of `E` in declaration order is designated by `mi` .

- the *product type size of E* is equal to the number of non-static data members of E.
- the *product type i* th*-element of E* is this `e.mi` ,
- the *product type i* th*-element type of E* is the declared type of that `E::mi` .

Otherwise the terms are undefined.

If any of the previous terms is not defined the others are not defined.

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

**Add the following section in N4617 **

Product type object

Product type synopsis

```
namespace std {
    template <class PT>
        struct is_product_type;
    template <class T>
        constexpr bool is_product_type_v = is_product_type<T>::value;
namespace product_type {

    template <class PT>
        struct size;

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

    template <size_t N, class PT>
        struct element;

}}
```

Template Class `is_product_type`

This trait is `true_type` if the type `T` is a product type.

Template Class `product_type::size`

```
template <class PT>
struct size : integral_constant<size_t, `see below`> {};
```

Remark: if *product type size* `PT` is defined, the value of the integral constant is *product type size* `PT`. Otherwise the trait is undefined.

Note: In order to implement this trait library it would be required that the compiler provides some builtin as e.g. `__builtin_pt_size(PT)` that implements *product type size* `PT`.

Template Class `product_type::element`

```
template <size_t N, class PT>
struct element {
    using type = `see below`
};
```

Remark: if `0 <= N` and `N < product_type::size<PT>::value` and *product type N*th*-element type of PT* is defined the nested alias `type` is *product type N*th*-element type of PT*. Otherwise it is undefined.

Note: In order to implement this trait library it would be required that the compiler provides some builtin as e.g. `__builtin_pt_element_type(N, PT)` that implements *product type element type* `N`, `PT`.

Function Template `product_type::get`

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

Returns: the *product type `N` th-element* of `pt`.

Remark: This operation would not be defined if `0 > N` and `N >= product_type::size<PT>::value` or *product type Nth-element* of `pt` is undefined.

Note: In order to implement this function library it would be required that the compiler provides some builtin as e.g. `__builtin_pt_get(N, pt)` that implements *product type Nth-element* of `pt`.

Implementability

This is not just a library proposal as the behavior depends on Structured binding P0217R3. There is no implementation as of the date of the whole proposal paper, however there is a non conforming implementation PT_impl for the parts that don't depend on the core language emulating the cases 1 and 2. The emulation doesn't conforms completely to the case 2 as it is using ordinary unqualified lookup (3.4.1) and should use lookup in the associated namespaces (3.4.2).

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 this for the IS or a TS?
- Do we want the interface inside a namespace `product_type`?
- Do we want the `std::product_type::size` / `std::product_type::get` functions?
- Do we want the `std::product_type::size` / `std::product_type::element` traits?
- Do we want the `pt_size` / `pt_get` operators in a future proposal?
- Do we want the `product_type` customization point in addition to the tuple-like one in a future revision?

Acknowledgments

Thanks to the LEWG Toronto participants on the review of the previous revision and for agreeing to move this paper to the LEWG.

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 R0 and presenting R1.

Special thanks and recognition goes to Technical Center of Nokia - Lannion for supporting in part the production of this proposal.

History

R3

Take in account the feedback from Toronto meeting. Next follows the direction of the committee:

- Don't even suggest any change to structured binding wording.
- Consider the possibility to have `product_type` customization points using traits in the rationale. No wording provided.

R2

Take in account the feedback from Kona meeting. Next follows the direction of the committee:

- Split the document into 3 documents
    - Product Type Access
    - Adaptation of current tuple-like algorithms to *ProductType*
    - More *ProductType* algorithms
- See if *ProductType* implementation for the types supporting structured binding can be generated using the Reflection TS P0194R3 interface.

This document describes the Product Type Access interface and shows that even if we can generate the implementation for some of the types, we are unable to generate the whole interface (or at least the authors don't know how todo it).

R1

- Adaptation to the adopted structured binding paper P0217R3.
- Addition of algorithms working on *Product-Types*.
- Adaptation of `<tuple>` , `<utility>` and `<array>` to *Product-Types*.

References

- Boost.Fusion Boost.Fusion 2.2 library

  http://www.boost.org/doc/libs/1600/libs/fusion/doc/html/index.html

- Boost.Hana Boost.Hana library

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

- N1691 Explicit Namespaces

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1691.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

- N4475 Default comparisons (R2)

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

- N4617 N4617 - Working Draft, C++ Extensions for Library Fundamentals, Version 2 DTS

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/n4617.pdf

- N5131 Working Draft, Standard for Programming Language C++

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n5131.pdf

- 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

- P0095R1 Pattern Matching and Language Variants http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0095r1.pdf

- 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

- P0217R3 Proposed wording for structured bindings

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0217r3.html

- P0221R2 Proposed wording for default comparisons

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/P0221R1.html

- P0311R0 A Unified Vision for Manipulating Tuple-like Objects

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

- P0326R0 Structured binding: alternative design for customization points

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0326r0.pdf

- P0327R1 Product Type Access (Revision 1)

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0327r1.pdf

- P0327R2 Product Type Access (Revision 2)

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0327r2.pdf

- P0341R0 parameter packs outside of templates

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0341r0.html

- PT_impl Product types access emulation and algorithms

  https://github.com/viboes/std-make/tree/master/include/experimental/fundamental/v3/product_type

- P0194R3 Static reflection

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0194r3.html

- P0385R2 Static reflection: Rationale, design and evolution

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0385r2.pdf

- P0578R0 Static Reflection in a Nutshell

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0578r0.html

- P0648R0 Extending Tuple-like algorithms to Product-Types

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0648r0.html

- P0649R0 Other Product-Type algorithms

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0649r0.html

- Reflection Product Type access reflection implementation for case 3

  https://gist.github.com/jacquelinekay/6bcacee7a3bce7d82b9d6387b6afee96