DRAFT

Document number:	D0XXXR0
Date:	2016-05-01
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 types access

Abstract

This paper proposes a library interface to access the same types covered by Structured binding SBR. This proposal name them *product types* The interface includes getting the number of elements, access to the n^{th} element and the type of the n^{th} element.

The wording depends on the wording of SBR.

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.

This paper proposes a library interface to access the same types covered by Structured binding SBR. This proposal name them *product types* The interface includes getting the number of elements, access to the n^{th} element and the type of the n^{th} element.

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

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.

Algorithms such as `std::tuple_cat` and `std::experimental::apply` that work well with *tuple-like* types, should work also for *extended tuple-like*. 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 `fold`, `accumulate`, `for_each` `any_of`, `all_of`, `none_of`, `find`, `count`, `filter`, `transform`, `replace`, `join`, `zip`, `flatten`.

P0144R2/P0217R1/SBR 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 P0217R1/SBR, allows to do structure binding for arrays and allow bitfields as members in case 3. 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 P0217R1.

The wording in P0217R1/SBR support C-arrays, but we are unable to find a `get<I>(arr)` overload on arrays that is found by ADL.

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

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

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 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 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 a *product type* access library interface.

Future Product type operator proposal (Not yet)

We don't propose yet the *product type* operators to get the size and thenth element as we don't have a good proposal for the names.

The *product type* access would 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 P0217R1.

The name of the operators `pt_size` and `pt_get` would be 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 N^{th} 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 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

1. As `sizeof(T)`, `pt_size(T)` and `pt_get(N, pt)` wouldn't be 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.

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

Product type library proposal

An alternative is to define generic functions `std::product_type::size<T>()` and `std::product_type::get<I>(pt)` using either some wording similar to the one in P0217R1.

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 could 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;

}}
```

If the committee considers that this interface has value for the C++17, this could means that in C++17 we could work with product types using these operations that would work for all the product types (except for bitfields), 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, we can already start to define a lot of algorithms.

Users could already define their own `bitfield_ref` class in C++17 and define its customization point for bitfields members if needed.

In order to define this function in C++20 we will need an additional compiler type trait `product_type_element_is_bitfield<N,PT>` that would say if the N^{th} element is a bitfield or not.

Design Rationale

What do we loss if we don't add this *product type* access in C++17?

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

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 generic algorithms based on the *tuple-like* access and we can not have a *tuple-like* access for c-arrays and for the types covered by Structured binding case 3 SBR.

Traits versus functions

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

Locating the interface on a specific namespace

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

We can also prefix them by `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 that we can use using directives and using declarations.

Using a `struct` would make the interface closed to nesting, but open by derivation.

Wording

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

Add the following section

Product type object

Product type synopsis

```
namespace std {
namespace product_type {

    template <class PT>
    struct size;

    template <size_t I, class PT>
    struct element;

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

}}
```

Template Function `product_type::size`

```
template <class PT>
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.

Template Class `product_type::size`

```
template <class PT>
struct size : integral_constant<size_t, *product type size* `PT`> {};
```

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

Template Function `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.

Template Class `product_type::element`

```
template <size_t N, class PT>
struct element { using type = decltype(product_type::get<N>(declval<PT>()))};
```

Remark: This trait would not be defined if `product_type::get<N>(declval<PT>())` is undefined.

`std::tuple_cat`

Adapt the definition of `std::tuple_cat` in[tuple.creation] to take care of product type

`std::apply`

Adapt the definition of `std::apply` in [xxx] to take care of product type

Implementability

This is not just a library proposal as the behavior depends on Structured binding SBR. There is no implementation as of the date of the whole proposal paper, however there is an implementation for the part that doesn't depend on the core language PT_impl.

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 `pt_size` / `pt_get` operators?
- 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 to adapt `std::tuple_cat`
- Do we want to adapt `std::apply`

Future work

Add `bitfield_ref` class and allow product type function access to bitfield fields

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

- SBR Structured binding: alternative design for customization points

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

- PT_impl Product types

https://github.com/viboes/std-make/blob/master/doc/proposal/reflection/product_type.cpp