Document number:	P0327R1
Date:	2016-10-16
Project:	ISO/IEC JTC1 SC22 WG21 Programming Language C++
Audience:	Reflection Working Group / Library Evolution Working Group
Reply-to:	Vicente J. Botet Escribá <vicente.botet@nokia.com>

Product-Type access

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 *Product Type*. The interface includes getting the number of elements, access to the n^{th} element and the type of the n^{th} element.

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

In addition, some of the algorithms that work for *tuple-like* access types are adapted to work with *Product-Types*.

Table of Contents

History

- 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*.

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 n^{th} element and the type of the n^{th} element. This interface doesn't use ADL.

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

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 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. 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 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 as well as a number of functions that can be built on top of this access functions.

Future *Product type* operator proposal (Not yet)

We don't propose yet the *product type* operators to get the size and the n^{th} 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 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 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 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.

Algorithms and function adaptation

`std::tuple_cat`

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

NOTE: This algorithm could be moved to a *product type* specific algorithms file.

Constructor from a product type with the same number of elements as the tuple

Similar to the constructor from `pair` .

This simplifies a lot the `std::tuple` interface (See N4387).

`std::apply`

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

NOTE: This algorithm could be moved to a *product type* specific algorithms file.

`std::pair`

piecewise constructor

The following constructor could also be generalized to *product types*

```
template <class... Args1, class... Args2>
    pair(piecewise_construct_t,
        tuple<Args1...> first_args, tuple<Args2...> second_args);
```

```
template <class PT1, class PT2>
    pair(piecewise_construct_t, PT1 first_args, PT2 second_args);
```

Constructor and assignment from a product type with two elements

Similar to the `tuple` constructor from `pair` .

This simplifies a lot the `std::pair` interface (See N4387).

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.

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.

Other functions for *ProductType*

There are a lot of useful function associated to product types that make use only of the product type access traits and functions.

`apply`

```
template <class F, class ProductType>
    constexpr decltype(auto) apply(F&& f, ProductType&& pt);
```

This is the equivalent of `std::apply` applicable to product types instead of tuple-like types.

`std::apply` could be defined in function of it.

`assign`

```
template <class PT1, class PT2>
    PT1& assign(PT1& pt1, PT2&& pt2);
```

Assignment from another product type with the same number of elements and convertible elements.

This function can be used while defining the `operator=` on product types. See the wording changes for `std::tuple`, `std::pair` and `std::array`.

`for_each`

```
template <class F, class ProductType>
  constexpr void for_each(F&& f, ProductType&& pt);
```

This is the equivalent of `std::for_each` applicable to product types instead of homogeneous containers or range types.

`make_from`

```
template <class T, class ProductType>
    constexpr `see below` make_from(ProductType&& pt);
```

This is the equivalent of `std::make_from_tuple` applicable to product types instead of tuple-like types.

`std::make_from_tuple` could be defined in function of it.

This function is similar to `apply` when applied with a specific `construct<T>` function.

`swap`

```
template <class PT>
   void swap(PT& x, PT& y) noexcept(`see below`);
```

Swap of two product types.

This function can be used while defining the `swap` on the namespace associated to the product type.

If we adopt were able to customize `swap` for concepts as in SWAPPABlE we could even be able to customize the swap operation for *ProductTypes*.

`to_tuple`

```
template <class ProductType>
    constexpr `see below` to_tuple(ProductType&& pt);
```

`std::tuple` is the more generic product type. Some functions could expect a specific `std::tuple` product type.

`fold_left` / `fold_right` / `accumulate`

This is the equivalent of `std::accumulate` applicable to product types instead of homogeneous containers types.

`lexicographical_compare`

This is the equivalent of `std::lexicographical_compare` applicable to product types instead of homogeneous containers types.

This function can be used while defining the comparison operators on product types when the default comparisons N4475 are not applicable. Note that default comparison is not applicable to all the *Product Types*, in particular the product types customized by the user.

This function requires that all the element of the product type are *OrderedComparable*.

`all_of`

Checks if n-unary n-predicate `p` returns `true` for all elements in the product type.

`any_of`

Checks if n-unary n-predicate `p` returns `true` for at least one elements in the product type.

`none_of`

Checks if n-n-unary predicate `p` returns `true` for no elements in the product type.

Other functions for *TypeConstructible ProductTypes*

Some algorithms need a *TypeConstructible ProductTypes* as they need to construct a new instance of a *ProductTypes*.

An alternative is to use `std::tuple` as the parameter determining the *Product Type* to construct.

We could also add a *TypeConstructible* parameter, as e.g.

```
template <template <class...> TC, class ...ProductTypes>
    constexpr `see below` cat(ProductTypes&& ...pts);
template <class TC, class ...ProductTypes>
    constexpr `see below` cat(ProductTypes&& ...pts);
```

Where `TC` is a variadic template for a *ProductType* as e.g. `std::tuple` or a TypeConstructor P0343R0.

cat

```
    template <class ...ProductTypes>
        constexpr `see below` cat(ProductTypes&& ...pts);
```

This is the equivalent of `std::tuple_cat` applicable to product types instead of tuple-like types. This function requires the first *Product Type* to be *Type Constructible*.

An alternative is to use `std::tuple` when the first *Product Type* is not *Type Constructible*.

We could also have

```
    template <template <class...> TC, class ...ProductTypes>
        constexpr `see below` cat(ProductTypes&& ...pts);
    template <class TC, class ...ProductTypes>
        constexpr `see below` cat(ProductTypes&& ...pts);
```

Where `TC` is a variadic template for a *ProductType* as e.g. `std::tuple` or a TypeConstructor [TypeConstructor].

`std::tuple_cat` could be defined in function of it one of the alternatives.

`transform`

```
  template <class F, class ProductType>
    constexpr `see below` transform(F&& f, ProductType&& pt);
```

This is the equivalent of `std::transform` applicable to product types instead of homogeneous containers types.

This needs in addition that `ProductType` is *TypeConstructible* (See P0338R0). Note that `std::pair` , `std::tuple` and `std::array` are *TypeConstructible*, but `std::pair` and `std::array` limit either in the number or in the kind of types (all the ame). A c-array is not type *TypeConstructible* as it cannot be returned by value.

Proposed Wording

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

Note that the wording for the "Product types terms" section have not been adapted to the current

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 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 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 other are not defined.

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

Add the following section in [N4564]

Product type object

Product type synopsis

```
namespace std {
    template <class PT>
        struct is_product_type;

namespace product_type {

    template <class PT>
        struct size;

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

    template <size_t I, class PT>
        struct element;

}}
```

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 <class PT>
struct element {
    using type = `see below`
};
```

Remark: if *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);
```

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

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

Remark: This operation would not be defined if *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`.

Add the following section in [N4564]

Product type algorithms

Product type algorithms synopsis

```
namespace std {

namespace product_type {

    template <class F, class ProductType>
        constexpr decltype(auto) apply(F&& f, ProductType&& pt);

    template <class PT1, class PT2>
        PT1& assign(PT1& pt1, PT2&& pt2);

    template <class ...PTs>
        constexpr `see below` cat(PTs&& ...pts);

    template <class F, class State, class ProductType
        constexpr State fold_left(ProductType&& pt, State&& state, F&& f);
    template <class F, class ProductType
        constexpr State fold_left(ProductType&& pt, F&& f);

    template <class F, class ProductType
        constexpr void for_each(ProductType&& pt, F&& f);

    template <class T, class PT>
        constexpr `see below` make_from(PT&& pt);

    template <class PT1, class PT2>
        PT1& move(PT1& pt1, PT2&& pt2);

    template <class PT>
        constexpr `see below` to_tuple(PT&& pt);

    template <class PT>
       void swap(PT& x, PT& y) noexcept(`see below`);


}}
```

Function Template `product_type::apply`

```
template <class F, class PT>
    constexpr decltype(auto) apply(F&& f, PT&& pt);
```

Effects: Given the exposition only function:

```
template <class F, class PT, size_t... I>
constexpr decltype(auto) apply_impl(F&& f, PT&& t, index_sequence<I...>) { // exposition only
    return INVOKE(std::forward<F>(f),
        product_type::get<I>(std::forward<Tuple>(
}
```

Equivalent to:

return apply<u>impl(std::forward(f), std::forward(t), product</u>type::element<u>sequence</u>for{});

Let `Ui` is `product_type::element_t<i, remove_cv_t<remove_reference_t<<PT>>>` .

Function Template `product_type::assign`

```
template <class PT1, class PT2>
    PT1& assign(PT1& pt1, PT2&& pt2);
```

In the following paragraphs, let `VPT2` be `remove_cv_t<remove_reference_t<PT2>>` , `Ti` be `product_type::element<i, PT1>` and `Ui` `product_type::element<i, VPT2>` .

Requires: both `PT1` and `VPT2` are *ProductTypes* with the same size, `product_type::size<PT1>::value==product_type::size<VPT2>::value` and `is_assignable_v<Ti&, const Ui&>` is true for all `i` .

Effects: Assigns each element of `pt2` to the corresponding element of `pt1` .

Function Template `product_type::cat`

```
template <template <class...> TC, class ...PTs>
    constexpr TC<CTypes> cat(PTs&& ...pts);
```

In the following paragraphs, let `Ti` be the `i` th type in `PTs` , `Ui` be `remove_reference_t<Ti>` , `pti` be the `i` th parameter in the function parameter pack `pts` , where all indexing is zero-based and

Requires: For all `i` , `Ui` shall be the type `cvi PTi` , where `cvi` is the (possibly empty) `i` th cv-qualifier-seq. Let `Aik` be `product_type::element<ki, PTi>` , the `ki` th type in `PTi` . For all `Aik` the following requirements shall be satisfied: If `Ti` is deduced as an lvalue reference type, then `is_constructible_v<Aik , cvi Aik &> == true` , otherwise `is_constructible_v<Aik, cviAik&&> == true` .

TODO: reword this paragraph *Remarks*: The types in `Ctypes` shall be equal to the ordered sequence of the extended types `Args0..., Args1..., ... Argsn-1...` , where `n` is equal to `sizeof...(PTs)` . Let `ei` ... be the `i` th ordered sequence of tuple elements of the resulting tuple object corresponding to the type sequence `Argsi` .

TODO: reword this paragraph *Returns*: A tuple object constructed by initializing the `ki` th type element `eik` in `ei...` with `get<ki>(std::forward<Ti>(pti))` for each valid `ki` and each group `ei` in order.

Note: An implementation may support additional types in the parameter pack `Tuples` that support the tuple-like protocol, such as pair and array.

Function Template `product_type::fold_left`

```
template <class F, class State, class ProductType>
  constexpr State fold_left(ProductType&& pt, State&& state, F&& f);

template <class F, class ProductType
  constexpr State fold_left(ProductType&& pt, F&& f);
```

Function Template `product_type::make_from`

```
template <class T, class PT>
    constexpr `see below` make_from(PT&& pt);
```

Effects: Given the exposition-only function:

```
template <class T, class PT, size_t... I>
    constexpr T make_fromimpl(PT&& t, index_sequence<I...>) { // exposition only
        return T(product_type::get<I>(std::forward<Tuple>(t))...);
    }
```

Equivalent to:

```
return make_from_impl<T>(forward<Tuple>(t),
    product_type::element_sequence_for<PT>{});
```

[Note: The type of T must be supplied as an explicit template parameter, as it cannot be deduced from the argument list. - end note]

Function Template `product_type::move`

```
template <class PT1, class PT2>
    PT1& move(PT1& pt1, PT2&& pt2);
```

In the following paragraphs, let `VPT2` be `remove_cv_t<remove_reference_t<PT2>>` , `Ti` be `product_type::element<i, PT1>` and `Ui` `product_type::element<i, VPT2>` .

Requires: both `PT1` and `VPT2` are *ProductTypes* with the same size, `product_type::size<PT1>::value==product_type::size<VPT2>::value` and `is_assignable_v<Ti&, Ui&&>` is true for all `i` .

Effects: Moves each element of `pt2` to the corresponding element of `pt1` .

Function Template `product_type::swap`

```
template <class PT>
    void swap(PT& x, PT& y) noexcept(`see below`);
```

Remark: The expression inside `noexcept` is equivalent to the logical and of the following expressions: `is_nothrow_swappable_v<Ti>` where `Ti` is `product_type::element<i, PT>` .

Requires: Each element in `x` shall be swappable with (17.6.3.2) the corresponding element in `y`.

Effects: Calls `swap` for each element in `x` and its corresponding element in `y`.

Throws: Nothing unless one of the element-wise `swap` calls throws an exception.

Function Template `product_type::to_tuple`

```
template <class PT>
    constexpr `see below` to_tuple(PT&& pt);
```

Effects: Equivalent to

Change 20.5.1p1 [tuple.general], Header synopsis as indicated.

Replace

```
template <class... Tuples>
constexpr tuple<CTypes...> tuple_cat(Tuples&&... tpls);
```

by

```
template <class... PTs>
constexpr tuple<CTypes...> tuple_cat(PTs&&... pts);
```

Change 20.5.2 [tuple.tuple], class template tuple synopsis, as indicated.

Replace

```
    // 20.4.2.1, tuple construction
    ...
    template <class... UTypes>
      EXPLICIT constexpr tuple(const tuple<UTypes...>&);
    template <class... UTypes>
      EXPLICIT constexpr tuple(tuple<UTypes...>&&);

    template <class U1, class U2>
      EXPLICIT constexpr tuple(const pair<U1, U2>&);          // only if sizeof...(Types) == 2
    template <class U1, class U2>
      EXPLICIT constexpr tuple(pair<U1, U2>&&);               // only if sizeof...(Types) == 2

    // 20.4.2.2, tuple assignment
    ...
    template <class... UTypes>
      tuple& operator=(const tuple<UTypes...>&);
    template <class... UTypes>
      tuple& operator=(tuple<UTypes...>&&);
    template <class U1, class U2>
      tuple& operator=(const pair<U1, U2>&); // only if sizeof...(Types) == 2
    template <class U1, class U2>
      tuple& operator=(pair<U1, U2>&&); // only if sizeof...(Types) == 2

    // allocator-extended constructors
    ...
    template <class Alloc, class... UTypes>
      EXPLICIT tuple(allocator_arg_t, const Alloc& a, const tuple<UTypes...>&);
    template <class Alloc, class... UTypes>
      EXPLICIT tuple(allocator_arg_t, const Alloc& a, tuple<UTypes...>&&);
    template <class Alloc, class U1, class U2>
      EXPLICIT tuple(allocator_arg_t, const Alloc& a, const pair<U1, U2>&);
    template <class Alloc, class U1, class U2>
      EXPLICIT tuple(allocator_arg_t, const Alloc& a, pair<U1, U2>&&);
```

by

```
// 20.4.2.1, tuple construction
...
template <class PT>
  EXPLICIT constexpr tuple(PT&&);

// 20.4.2.2, tuple assignment
...
template <class PT>
    tuple& operator=(PT&& u);

// allocator-extended constructors
...
template <class Alloc, class PT>
  EXPLICIT tuple(allocator_arg_t, const Alloc& a, PT&&);
```

Constructor from a product type

Suppress in 20.5.2.1p3, Construction [tuple.cnstr]

> , and `Ui` be the `i` `th` type in a template parameter pack named `UTypes`, where indexing is zero-based

Replace 20.5.2.1p15-26, Construction [tuple.cnstr] by

```
template <class PT>
  EXPLICIT constexpr tuple(PT&& u);
```

Let `Ui` is `product_type::element_t<i, remove_cv_t<remove_reference_t<<PT>>>`.

Effects: For all `i`, the constructor initializes the `i` th element of `*this` with `std::forward<Ui>(product_type::get<i>(u))`.

Remarks: This constructor shall not participate in overload resolution unless `PT` is not `tuple<Types...>,` PT `is a *product type* with the same number elements than this tuple and` is<u>constructible::value</u> `is true for all` i `. The constr`

Assignment from a product type

Suppress in 20.5.2.2p1, Assignment [tuple.assign]

> and `Ui` be the `i` `th` type in a template parameter pack named `UTypes`, where indexing is zero-based

Replace 20.5.2.2p9-20, Assignment [tuple.assign] by

```
template <class PT>
  tuple& operator=(PT&& u);
```

Let `Ui` is `product_type::element_t<i, remove_cv_t<remove_reference_t<<PT>>>`.

Effects: For all `i`, assigns `std::forward<Ui>(product_type::get<i>(u))` to `product_type::get<i>(*this)`

Returns: `*this`

Remarks: This function shall not participate in overload resolution unless `PT` is a *product type* with the same number elements than this tuple and `is_assignable<Ti&, const Ui&>::value` is true for all `i`.

[Note: - We could as well say equivalent to `product_type::copy(std::forward<PT>(u), *this); return *this`. end note]

Allocator-extended constructors from a product type

Change the signatures

```
template <class Alloc>
  tuple(allocator_arg_t, const Alloc& a, const tuple&);
template <class Alloc>
  tuple(allocator_arg_t, const Alloc& a, tuple&&);
template <class Alloc, class... UTypes>
EXPLICIT tuple(allocator_arg_t, const Alloc& a, const tuple<UTypes...>&);
template <class Alloc, class... UTypes>
EXPLICIT tuple(allocator_arg_t, const Alloc& a, tuple<UTypes...>&&);
template <class Alloc, class U1, class U2>
EXPLICIT tuple(allocator_arg_t, const Alloc& a, const pair<U1, U2>&);
template <class Alloc, class U1, class U2>
EXPLICIT tuple(allocator_arg_t, const Alloc& a, pair<U1, U2>&&);
```

by

```
    template <class Alloc, class PT>
      EXPLICIT tuple(allocator_arg_t, const Alloc& a, PT&&);
```

`std::tuple_cat`

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

Replace `Tuples` by `PTs` , `tpls` by `pts` , `tuple` by `product type` , get by `product_type::get` and `tuple_size` by `product_type::size` .

```
template <class... PTs>
constexpr tuple<CTypes...> tuple_cat(PTs&&... pts);
```

[Note: - We could as well say equivalent to `product_type::cat<tuple>(std::forward<PT>(pts)...);` . end note]

`std::apply`

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

Replace `Tuple` by `PT` , `t` by `pt` , `tuple` by `product type` , `std::get` by `product_type::get` and `std::tuple_size` by `product_type::size` .

```
template <class F, class PT>
constexpr decltype(auto) apply(F&& f, PT&& t);
```

[Note: - We could as well say equivalent to `product_type::apply(std::forward<F>(f), std::forward<PT>(t));` . end note]

`std::pair`

Change 20.3.2 [pairs.pair], class template pair synopsis, as indicated:

Replace

```
template <class... Args1, class... Args2>
    pair(piecewise_construct_t,
        tuple<Args1...> first_args, tuple<Args2...> second_args);
```

by

```
template <class PT1, class PT2>
    pair(piecewise_construct_t, PT1 first_args, PT2 second_args);
```

Add
`` ```c++ ``
template EXPLICIT constexpr pair(PT&& u); ...

```
template <class PT>
  pair& operator=(PT&& u);
```

} `` ``` ``

piecewise constructor

Replace

```
template <class... Args1, class... Args2>
    pair(piecewise_construct_t,
        tuple<Args1...> first_args, tuple<Args2...> second_args);
```

by

```
template <class PT1, class PT2>
    pair(piecewise_construct_t, PT1 first_args, PT2 second_args);
```

Constructor from a product type

Add

```
template <class PT>
  EXPLICIT constexpr pair(PT&& u);
```

Let `Ui` is `product_type::element_t<i, remove_cv_t<remove_reference_t<<PT>>>` .

Effects: For all `i` , the constructor initializes the `i` th element of `*this` with `std::forward(product_type::get*(u))*.

Remarks: This function shall not participate in overload resolution unless `PT` is not `pair<T1, T2>` , `PT` is a product type with 2 elements and `is_constructible<Ti, Ui&&>::value` is true for all `i` The constructor is explicit if and only if `is_convertible<Ui&&, Ti>::value` is false for at least one `i` .

Assignment from a product type

```
template <class PT>
  pair& operator=(PT&& u);
```

Let `Ui` is `product_type::element_t<i, remove_cv_t<remove_reference_t<<PT>>>` .

Effects: For all `i` in `0..1` , assigns `std::forward<Ui>(product_type::get<i>(u))` to `product_type::get<i>(*this)`

Returns: `*this`

Remarks: This function shall not participate in overload resolution unless `PT` is a product type with 2 elements and `is_assignable<Ti&, const Ui&>::value` is true for all `i` .

[Note: - We could as well say equivalent to `product_type::copy(std::forward<PT>(u), *this); return *this` . end note]

`std::array`

Add

Assignment from a product type

```
template <class PT>
  array& operator=(PT&& u);
```

Let `Ui` is `product_type::element_t<i, remove_cv_t<remove_reference_t<<PT>>>` .

Effects: For all `i` in `0..1` , assigns `std::forward<Ui>(product_type::get<i>(u))` to `product_type::get<i>(*this)`

Returns: `*this`

Remarks: This function shall not participate in overload resolution unless `PT` is a product type with `N` elements and `is_assignable<T&, const Ui&>::value` is true for all `i` .

[Note: - We could as well say equivalent to `product_type::copy(std::forward<PT>(u), *this); return *this` . end note]

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 an implementation for the part that doesn't depend on the core language [PT_impl]() emulating the cases 1 and 2. The standard library has not been adapted yet neither.

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 `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`*
- *Do we want the new constructors for `std::pair` and `std::tuple`*
- *Do we want the `pt_size` / `pt_get` operators in a future proposal?*

Future work

Add other algorithms on Product Types

`front`

`front: PT(T) -> T`

`back`

`back: PT(T) -> T`

`is_empty`

`is_empty : PT(T) -> bool`

Add other algorithms on TypeConstructible Product Types

The following algorithms need a `make<TC>(args...)` factory [P0338R0](P0338R0).

If the first product type argument is TypeConstructible from the `CTypes` then return an instance of it; otherwise construct a `std::tuple`.

`cat`

`cat: TCPT(T)... -> TCPT(T)`

`transform`

`transform: TCPT(T)... -> TCPT(T)`

`drop_front`

`drop_front: TCPT(T) -> TCPT(T)`

`drop_back`

`drop_back: TCPT(T) -> TCPT(T)`

`group`

`TCPT(T) -> TCPT(TCPT(T))`

`insert`

`insert: TCPT(T) x unsigned x T -> TCPT(T)`

...

Product Types views and lazy algorithms

Based on Range views for homogeneous Ranges Range-v3, views for heterogeneous sequences Boost.Fusion define Product Types views, adaptors, ...

Tagged Product Types

Based on the work N4569 for tagged tuples, associative sequences in Boost.Fusion, Struct in Boost.Hana define Tagged ProductTypes and specific algorithms for them.

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 R0.

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

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

- *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

- *N4569 Proposed Ranges TS working draft*

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/n4569.pdf

- *N4564 N4564 - Working Draft, C++ Extensions for Library Fundamentals, Version 2 PDTS*

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4564.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

- P0338R0 *C++ generic factories*

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0338r0.pdf

- P0341R0 *parameter packs outside of templates*

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

- P0343R0 *Meta-programming High-Order Functions*

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0343r0.html

- PT_impl *Product types access emulation and algorithms*

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

- SWAPPABlE *ProductTypes must be Swappable by default*

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

PT_SWAP *ProductTypes must be Swappable by default*

```
https://github.com/viboes/std-make/blob/master/include/experimental/fundamental/v3/product_type/swap.hpp
```

Range-v3 *range-v3*

```
https://github.com/ericniebler/range-v3
```

Range-v3