Document number:	P0648R0
Date:	2017-06-15
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>

Extending Tuple-like algorithms to Product-Types

Abstract

This paper proposes to adapt the algorithms and interfaces working today with *tuple-like* types to *Product-Types* based on P0327R2 proposal.

Table of Contents

- History
- Introduction
- Motivation
- Proposal
- Design Rationale
- Proposed Wording
- Implementability
- Open points
- Future work
- Acknowledgements
- References

History

R0

Take in account the feedback from Kona meeting concerning P0327R1. Next follows the direction of the committee:

- Split the proposal into 3 documents
  - P0327R2 Product Type Access
  - [P0648R0] Extension of current tuple-like algorithms to *ProductType*
  - [P0649R0] Other *ProductType* algorithms

In this document, we describe the extension of the current tuple-like algorithms to the proposed *ProductType* requirements.

Introduction

There are some algorithms working almost on *tuple-like* types that can be extended to *ProductTypes*. Some examples of such algorithms are `apply` , `swap` , `lexicographical_compare` , `cat` , `assign` , `move` , ...

Motivation

Adaptation

Algorithms such as `std::tuple_cat` and `std::apply` that work well with *tuple-like* types, should work also for any *ProductType* types.

Reuse

The definition of some the existing functions and assignment operators will surely be implemented using the same algorithm generalized for any *ProductType*. This paper proposes only the algorithms that could be needed to implement (or define) the current *tuple-like* interface extended to *ProductTypes*.

Deprecation

Some of the current algorithms could be deprecated in favor of the new ones. E.g. we could deprecate `std::apply` in favor of `std::product_type::apply`.

Extension

There are many more of them. [P0649R0] takes in account some of the algorithms that work well with *ProductTypes*.

Proposal

tuple-like algorithms and function adaptation

`std::tuple_cat`

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

Note: The single difference between `std::tuple_cat` and `std::product_type::cat` is `std::tuple_cat` forces `std::tuple<>` as result type.

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`

Adaptation of the definition of `std::apply` to take care of *ProductType*.

NOTE: This algorithm could just forward to `product_type::apply`.

`std::pair`

piecewise constructor

The following constructor could also be generalized to *ProductTypes*.

Instead of

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

we could have

```
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).

Functions for *ProductType*

The definition of the existing function will surely be implemented using the same algorithm generalized for any *ProductType*.

`product_type::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.

`product_type::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`.

`product_type::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.

`product_type::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 adopted the possibility to customize customization points for concepts we could even be able to customize the swap operation for *ProductTypes*.

`product_type::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.

`product_type::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*.

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.

`product_type::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* but this creates a cycle.

We could also have an additional parameter stating which will be the result

```
    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](P0343R0).

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

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 same).

A c-array is not *TypeConstructible* as it cannot be returned by value.

Design Rationale

Locating the interface on a specific namespace

The name of *product type* algorithms, `cat`, `apply`, `swap`, `assign` or `move` 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.

Proposed Wording

The proposed changes are expressed as edits to N4564 Working Draft, C++ Extensions for Library Fundamentals, Version 2.

If [P0550R0] is accepted, any use of `remove_cv_t<remove_reference_t<T>>` should be replaced by `uncvref_t

Add the following section in N4564

Product type algorithms

Product type algorithms synopsis

```
namespace std::experimental {
inline namespace fundamental_v3 {
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 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_impl(std::forward<F>(f), std::forward<PT>(t),
                    product_type::element_sequence_for<PT>{});
```

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_t<i, PT1>` and `Ui` be `product_type::element_t<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_t<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::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_t<i, PT1>` and `Ui` `product_type::element_t<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_t<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 ......

```
    product_type::make_from<tuple<product_type::element_t<0, PT>,...>>(pt);
```

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::assign(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::assign(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::assign(std::forward<PT>(u), *this); return *this`. end note]

Implementability

This is a library proposal. There is an implementation [PT_impl] of the basic ProductType algorithms. The standard library has not been adapted yet.

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? If a TS which one?*
- *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`*

Future work

Add other parts of the current standard if we have missed them.

Acknowledgments

Thanks to all those that help on [P0327R1].

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

References

- *[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

- *[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

- *[P0095R1] Pattern Matching and Language Variants*

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0095r1.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

- *[P0338R0] C++ generic factories*

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

- *[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