Document number:	P0197R0
Date:	2016-02-11
Project:	ISO/IEC JTC1 SC22 WG21 Programming Language C++
Audience:	Evolution Working Group
Reply-to:	Vicente J. Botet Escriba <vicente.botet@wanadoo.fr>

Default Tuple-like Access

Abstract

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.

I propose to (implicitly) supply default versions of these operation and traits, if needed. The meaning of `get<I>` is to return a reference to the I^{th} member.

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.

I propose to (implicitly) supply default versions of these operation and traits, if needed. The meaning of `get<I>` is to return a reference to the I^{th} member.

If the simple defaults are unsuitable for a class, a programmer can, as ever, define more suitable ones or suppress the defaults. The proposal is to add the operations as an integral part of C++ (like the assignment operator), rather than as a library feature.

The proposal follows the same approach as Default Comparison (N4475), that is, that having default generated code for these basic operations only when needed and possible would make the language simpler.

The concerned classes would be the same as the ones on which Structured Binding (P0144R0) can be applied.

This paper contains no proposed wording. This is a discussion paper to determine EWG interest in the feature, and if there is interest to get direction for a follow-up paper with wording.

Motivation

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

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.

Proposals such as Structured Bindings P0144R0 would already provide positional access. This proposal and structured binding should use the same restrictions on the types that can be applied.

Proposal

I propose to generate default implementations for `tuple_size`, `tuple_element` and `get<I>/get<T>` when needed. If the defaults are unsuitable for a type, the user may be explicitly delete (`=delete`) them. If the default implementation is not suitable, the user may provide a definition (as always). If an operation is already declared, a default implementation is not generated for it. This is exactly

the way assignment and constructors work today and as comparison operators would work if N4475 is adopted.

A library solution could be an alternative once we have the necessary reflection traits.

Common restrictions

A default implementation of tuple-like access for a class can be generated unless the class

- contains protected or private non-static data members, or
- contains protected or private or virtual base classes, or
- contains a public base class that fulfills some of these restrictions.

We expect that a follow-up for P0144R0 will define the same restrictions but we are not sure the first version would allow inheritance.

Tuple-like access

Given a class `C` that doesn't define the tuple-like access `get<I>` and it is not restricted by the previous conditions, let

- `N` be the number of public non-static data members,
- `Ti` the type of the i^{th} data member (0 based).

The following could be defined

```
namespace std {
    template <>
      struct tuple_size<C> : integral_constant<size_t, N> {}
    template <>
      struct tuple_element<i, C> { using type = Ti; }    // for i in 0..N-1

    template <size_t I>
      constexpr tuple_element_t<I, C>& get(C& c) noexcept;
    template <size_t I>
      constexpr tuple_element_t<I, C> const& get(C const& c) noexcept;
    template <size_t I>
      constexpr tuple_element_t<I, C> && get(C && c) noexcept;

    template <class T>
      constexpr T& get(C& c) noexcept;
    template <class T>
      constexpr T const& get(C const& c) noexcept;
    template <class T>
      constexpr T && get(C && c) noexcept;
}
```

This definition would be in line with the tuple-like access as defined for `std::tuple`.

Explicit conversion to `std::pair<T,U>` and `std::tuple<Ts...>`

We could make any class `C` satisfying the constraints and having 2 data members of types `T` and `U` to be explicitly convertible to `std::pair<T,U>`.

We could make any class `C` satisfying the constraints and having `N` data members of types `T1` ... `Tn` explicitly convertible to `std::tuple<T1,..., Tn>`.

We could generate the explicit conversions `operator std::pair<T,U>` and `operator std::tuple<Ts...>`.

However the syntax would be less friendly than a non-member conversion functions `to_pair` and `to_tuple` if P0091R0 is not adopted.

Given `c++ TupleLike<int, string> tpl= ...;`

compare

```
f(std::tuple<int, string>(tpl)); // if [P0091R0] is not adopted
f(std::tuple(tpl)); // if [P0091R0] is adopted
f(std::to_tuple(tpl)); // complementary if [P0091R0] is adopted
```

If P0091R0 is not adopted, in addition of the explicit conversions we propose to overload the new `to_pair` and `to_tuple` functions in the namespace of the class `C`.

```
namespace Cns {
    class C;
    auto to_pair(C const&);
    auto to_tuple(C const&);
}
```

Concerned classes in the standard

The definition of the tuple-like access could be removed in the text of the standard for the following classes:

- `std::pair<T,U>`
- `std::tuple<Ts...>`

Design Rationale

What about inheritance?

With the adoption of Extension to Aggregate Initialization [P0017R0], it is coherent to permit public inheritance as we want the tuple-like access to be generated for aggregates.

Do we want to consider base classes or its data members as elements of the tuple-like class ?

P0017R1 considers the base classes as element of the aggregation.

We don't know yet what Structured Bindings P0144R0 will do, but suspect that it would consider base classes as elements of the structured binding as P0017R1 does.

The tuple-like access to `std::tuple` is member oriented even if the order is not required. Having coherent order of members and template parameters could permit conversion from a tuple to a tuple having some of the last members removed.

This is why this proposal gives recursive access to the member of the base classes.

Do we need conversions?

We can consider `std::tuple` as the underlying type of such structures. It seems reasonable to have a conversion to it.

Having this conversion will allow the user to use functions that work with `std::tuple`.

Why not implicit conversions?

So what kind of conversions are desired? `std` `tuple<T,U>` is implicitly convertible to `std::pair<T,U>`.

Note that the cost of the conversion means a copy and that this is more expensive. The cost of copying two elements could be less expensive than copying more.

Working paper wording

This wording is very “drafty” and has not gone through expert review. It is intended to reflect the design decisions described above.

The author was not aware of the new wording for Default Comparison in N4532. There is a lot there that should inspire the wording for this function.

The wording that follows is based on the wording of the current standard in particular (N4527), and on initial wording in N4475.

Wording for explicit conversions to be added if the idea is retained.

Add a "Tuple-like access expression" section in 5

Tuple-like access expression [expr.get]

A *tuple-like access expression* is a particular case of a *function call expression* when the function name is `get<I>`.

If an operand is of class type and no suitable function is found in the class namespace, the implicitly-declared `get<I>` non-member operation as described in over.generate_get is used.

Add a "Special non-member tuple-like access operations" section after 13.6

Special non-member `get` operation [over.generate_get]

Implicitly-declared `get` non-member operation

If no user-defined `get` operation is provided for a class type `T` (`struct`, `class` but not `union`), and all of the following is true:

- doesn't contain protected or private non-static data members, or
- doesn't contain protected or private or virtual base classes, or
- doesn't contain a public base class that fulfills some of these restrictions.

then the compiler will declare a `get` operation with the signature

```
template <size_t I>
requires I < tuple_size<T>{}
friend tuple_element<T, I> get(T&) noexcept(see below);
```

The generated implementation is not considered a function so it cannot have its address taken [Note: like the `=` operator.].

Explicitly defaulted `get` non-member operation

The user cannot force the generation of the implicitly declared `get` operation declaring it with the keyword `default`.

Deleted implicitly-declared `get` non-member operation

The implicitly-declared or defaulted `get` operation for class `T` is never defined as deleted.

Implicitly-defined `get` non-member operation

The implicitly-declared `get` operation is always defined (that is, a function's body is generated and compiled) by the compiler if odr-used. The `get<I>` non-member operation gets the I^{th} of the concerned non-static data member of the object, in their initialization order.

Alternative solutions

Based on a future reflection library (e.g. N4428 or N4451), we could define the tuple-like access instead of generating it (of course, for classes satisfying the tuple-like access generation requirements).

Next follows a incomplete implementation when inheritance is not considered.

```
namespace std {namespace experimental {namespace reflect { inline namespace v1 {

    template <class C>
    struct is_tuple_like_generation_enabled : bool_constant<
        is_class<C>{} &&
        //   no private or protected non-static data members
        class_protected_non_static_data_members<C>::size==0 &&
        class_private_non_static_data_members<C>::size==0 &&
        //   no base classes (this is the restriction of the draft - no inheritance)
        class_base_classes<C>::size==0
        > {};

    template <class C, class Enabler=void>
    struct tuple_size
    template <class C, class Enabler=enable_if_t<is_tuple_like_generation_enabled<C>
    struct tuple_size<C> : size_t_constant<class_public_non_static_data_members<C>::

    template <size_t N, class C, class Enabler=void>>
    struct tuple_element;
    template <size_t N, class C, class Enabler=enable_if_t<is_tuple_like_generation_
    struct tuple_element<N, C> {
        using pointer = class_public_non_static_data_members<C>::get<N>::pointer
        using type = decltype(std::declval<C>().*declval<pointer>());
    };
}}}
```

```
// default definition for tuple_size changed
template <class C>
struct tuple_size : reflect::tuple_size<C> {}

// default definition for tuple_element changed
template <size_t N, class C>
struct tuple_element { using type = typename reflect::tuple_element<N, C>::type; }

// overloads of get when reflect::is_tuple_like_generation_enabled<C>
template <size_t N, class C, class Enabler=enable_if_t< N < tuple_size<C>{} and refl
 constexpr reflect::tuple_element_t<N,C>& get(C & that) noexcept
{
    typename reflect::tuple_element<N,C>::pointer pm;
    return that.*pm;
}

template <size_t N, class C, class Enabler=enable_if_t< N < tuple_size<C>{} and refl
constexpr const tuple_element_t<N,C>& get(const C & that) noexcept
{
    typename reflect::tuple_element<N,C>::pointer pm;
    return that.*pm;
}

template <size_t N, class C, class Enabler=enable_if_t< N < tuple_size<C>{} and refl
constexpr reflect::tuple_element_t<N,C>&& get(C && that) noexcept
{
    typename reflect::tuple_element<N,C>::pointer pm;
    return std::forward<reflect::tuple_element_t<N,C>&&>(that.*pm);
}

} // namespaces
```

Note that the default traits `tuple_size` and `tuple_element` have been redefined. Maybe using Concepts this could be relaxed.

```
// default definition for tuple_size changed
template <class C>
requires reflect::is_tuple_like_generation_enabled<C>{}
struct tuple_size : reflect::tuple_size<C> {}

// default definition for tuple_element changed
template <size_t N, class C>
requires N < tuple_size<C>{} and reflect::is_tuple_like_generation_enabled<C>{}
struct tuple_element { using type = typename reflect::tuple_element<N, C>::type; }
```

Implementability

This proposal needs some compiler magic, either by generating directly the tuple-like access or by providing the reflection traits as e.g. in N4428 or N4451.

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 a default or a reflection solution?
- Do we want an explicit conversions to `std::pair` / `std::tuple` ?

Acknowledgments

Thanks to Matthew Woehlke for championing this proposal during the C++ standard committee meetings and its global review.

Thanks to all those that have commented the idea on the std-proposals ML better helping to identify the constraints, in particular to Nicol Bolas and Matthew Woehlke.

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

- 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

- P0091R0 - Template parameter deduction for constructors (Rev. 3)

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0091r0.html

- P0144R0 Structured Bindings

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0144r0.pdf

- P0151R0 Proposal of Multi-Declarators

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0151r0.pdf