Sequential access to data members and base sub-objects

Doc. no. N3326=12-0016
Date: 2011-12-19
Author: Andrzej Krzemieński
Contact: akrzemi1@gmail.com
Addresses: EWG

Abstract

This proposal describes a new language feature that enables the program to view any object of a (non-union) class type as a tuple (`std::tuple`), where each tuple element corresponds to the object's respective non-static data member. This enables class authors and library writers to express that some operation must be applied to all object's members (or base sub-objects), without the necessity to name each member explicitly.

Unlike a general-purpose compile-time reflection mechanism as proposed in N1471[1], this proposal addresses only one aspect of reflection (listing class's data members) by adding a fairly small section to the Standard Library (which requires certain 'compiler magic').

Motivation

Typically, when a programmer creates a regular type she has to repeat a lot of redundant code when specifying comparison operations:

```
struct Reg {
   int a;
   Rational b;
   std::string c;
};

bool operator==( Reg lhs, Reg rhs ) {
   return lhs.a == rhs.a
      && lhs.b == rhs.b
      && lhs.c == rhs.c;
}

bool operator<( Reg lhs, Reg rhs ) {
   if( lhs.a < rhs.a ) return true;
   if( lhs.a > rhs.a ) return false;
   if( lhs.b < rhs.b ) return true;
   if( lhs.b > rhs.b ) return false;
   if( lhs.c < rhs.c ) return true;
   if( lhs.c > rhs.c ) return false;
   return false;
}
```

There is an obvious pattern in those implementations, but there is currently no way in C++ to automate this definition. While it is possible to extend the current "defaulted implementation" syntax to work also for comparison operators:

```
bool operator==(Reg lhs, Reg rhs) = default;
bool operator< (Reg lhs, Reg rhs) = default;
```

The ability to specify that we want to perform some operation "for each member sub-object" (or "for each base sub-object") goes beyond comparison operators. For instance, it enables libraries to automatically stream out user defined types, automatically archive or marshal objects. Additionally, some libraries require an indexed access to object's members without the necessity to do something "for all members". For instance Boost.Spirit[2] library requires that for a given parser, say `(int_ << double_ << char_)`, it is given a corresponding attribute (object) whose first (at index 0) subobject is of type `int`, second of type `double`, third of type `char`. The following class:

```
struct S {
   int i;
   double d;
   char c;
};
```

does meet the requirement, but there is no way for the library to know it, unless the user provides an explicit mapping herself, but in that case, she has to repeat something that is already obvious.

Current workaround for Boost users is to use macro `BOOST_FUSION_ADAPT_STRUCT` and similar:

```
namespace demo {
   struct employee {
      std::string name;
      int age;
   };
}

BOOST_FUSION_ADAPT_STRUCT(
   demo::employee,
   (std::string, name)
   (int, age)
)
// demo::employee is now a Fusion sequence
```

This solution also enables the automatic implementation of comparison operators for structs. However, it has limitations. We still need to list all members twice. This is better than listing them *n* times, but is still redundant and error prone. For instance, if we add a third member to struct `employee`, no-one will remind us that we should have also added it also to the adapting macro. Second, it only works for members with public access; thus we are still unable to implement an assignment operator, which needs to be a member function, and assign also to private members.

Another goal of the proposed introspection library is to ensure that no member access is violated. That is, if someone does not have access to member `m` of our class, he shouldn't also be able to access the member via the tuple interface.

Description

With the functionality proposed here, the following syntax is possible:

```
struct Rec {
   int a;
   Rational b;
   std::string c;
};

Rec r{1, {3, 7}, "dog"};
auto tup = std::meta::members(r).value;
```

Now, tuple `tup` is a *view* of object `r`. The type of `tup` is `std::tuple<int&, Rational&, std::string&>`. It is somewhat similar to writing:

```
auto tup = std::tie(r.a, r.b, r.c);
```

The important difference is that you do not have to list all `Rec`'s members one by one. If you add a new member data to struct `Rec`, the former notation will automatically include it. It is more suitable in generic code, where you do not know the members of the class you are using.

If `r` was declared const, the type of the `tup` would have been `std::tuple<int const&, Rational const&, std::string const&>`. The first element of `tup` is a reference to `r.a`, the second element is a reference to `r.b`. Assigning to `std::get<0>(tup)` is equivalent to assigning to `r.a` directly; assigning new value to `std::get<1>(tup)` changes the value of `r.b`:

```
std::get<0>(tup) = 99;
assert( r.a == 99 );
```

LessThan comparison for type `Rec` can be now implemented as follows:

```
bool operator<( Rec const& l, Rec const& r ) {
   return std::meta::members(l).value < std::meta::members(r).value;
}
```

This compares two tuples that represent objects `l` and `r`. It takes advantage of tuple's `operator<`, which is defined according to most programmers' intuition. Again, it is similar to:

```
bool operator<( Rec const& l, Rec const& r ) {
   return std::tie(l.a, l.b, l.c) < std::tie(r.a, r.b, r.c);
}
```

But it doesn't require of the programmer to list, or even be aware of, all `Rec`'s members.

How does this work? Similarly to how type traits work. When a 'meta' expression (like `std::meta::members(l).value`) appears in the code, compiler specializes function template `meta` as well as the type this function returns. Continuing the above example with `operator<`, compiler generates a specialization for `Rec&`:

```
namespace std{
namespace meta{

   template<>
   auto members<Rec&>(Rec& obj) -> MEMBERS_T<Rec&>
   {
      return MEMBERS_T<Rec&>(obj);
   }

   template<>
   class MEMBERS_T<Rec&>
   {
      friend auto members<Rec&>( Rec& obj ) -> MEMBERS_T<Rec&>;
      MEMBERS_T(Rec& obj) : value{obj.a, obj.b, obj.c} {}

   public:
      std::tuple<int&, Rational&, std::string&> value;
   };

}} // namespace std::meta
```

When compiling the definition of `operator<`, compiler is allowed not to create and copy tuples, and simply perform member-wise comparison, because expression `std::meta::members(l).value` is a compiler intrinsic. `meta` is a special namespace in namespace `std`. It is special because every class or struct defined in the program or in the library implicitly declares classes and functions defined in `meta` as its friends. Thus, classes and functions from namespace `meta` have access to every member in every class/struct regardless if these members are declared with public, protected or private access. This does not violate encapsulation, though, because classes and functions from `meta` do not themselves read or write those members. They only provide back the access to the objects via the tuple member object `value`, in a manner that guarantees the preservation of encapsulation, as described below. `members` is a function template specialization. Compiler specializes this template for every class/struct `R` in a number of variants: for `R&`, `R const&`, `R volatile&` and `R volatile const&`, `R&&`, `R const&&`, `R volatile&&` and `R volatile const&&` (although we do not expect the last three to be used in practise). This function specialization returns an object of unspecified non-union class type (but different for each instantiation of `members`) (we refer to it as `MEMBERS_T<S>`, where `S` is `R&`, `R&&`, `R const&`, ...), which contains our data member `value`. `value` in the type `MEMBERS_T<S>` for different types `R` may be declared either private or public, based on the following rules. If all members in `R` are public, `value` is declared public, otherwise `value` is declared private, but in exchange class `MEMBERS_T<S>` declares as a friend class `R` and everyone else that `R` also declares as friend. Thus, whoever has direct access to *all* data members of `R`, has also access to `R` when viewed as tuple; and (although this is a bit more restrictive) if someone does not have access to at least one data member of `R`, he also does not have access to *any* member of `R` when viewed as a tuple. The following example illustrates that.

```
class Name {
public:
   std::string first;
   std::string second;
   void process();

private
   friend class IdManager;
   Id id;
};

Name gname;
std::meta::members(gname).value;     // Error: value is private

void Name::process() {
   std::meta::members(gname).value;  // OK: value is private but Name is a friend of
}                                    // type MEMBERS_T<Name&>

void IdManager::inspect() {
   std::meta::members(gname).value;  // OK: value is private but IdManager is a
}                                    // friend of type MEMBERS_T<Name&>
```

Note that the three above expressions `std::meta::members(gname)` return prvalues of the very same type `MEMBERS_T<Name&>`. The fact that the full expression is invalid in the first case and valid in the latter two is a consequence of ordinary C++ rules: private-vs-public member access and friendship. To illustrate it better, let's see how compiler magic works when the definition of class `Name` as described above is parsed. Compiler will modify the definition slightly and provide one or more specializations:

```
class Name {
public:
   std::string first;
   std::string second;
   void process();

private
   friend class IdManager;
   Id id;

   [[MAGIC]] template<class T>
      friend class std::meta::MEMBERS_T;

   [[MAGIC]] template<class T>
      friend auto std::meta::members(T&&) -> std::meta::MEMBERS_T<T&&>;
};

namespace std{ namespace meta{

   [[MAGIC]]
   template<> class std::meta::MEMBERS_T<Name&>
   {
   private: // everything private
      std::tuple<std::string&, std::string&, Id&> value;
      MEMBERS_T( Name& obj ) : value{ obj.first, opj.second, obj.id } {}

      [[MAGIC]] friend class ::Name;
      [[MAGIC]] friend class ::IdManager;
      [[MAGIC]] friend auto members<Name&>(Name&) -> MEMBERS_T<Name&>;
   }

}} // std::meta
```

In other words, we propose the notation that requires accessing (private or public) member `value` (and class `MEMBERS_T<T>` declaring friendship to `T` and `T`'s friends) to allow the abovementioned "context sensitivity". And owing to this context sensitivity, friends of `T` can use the proposed reflection mechanism to access `T`'s private data members.

This feature blends the language with the library, similarly to `typeinfo`, `initializer_list`, "for each" loop and `type_traits`. The entire expression `std::meta::members(e).value` should be thought of as a sort of keyword (say, `membersof(e)`). That is, we do not want to encourage any other usages, like

```
return std::meta::members(e); // No reason to do that
```

Therefore this proposal does not require meeting any requirements like `CopyConstructible` on 'intermediate' types (`MEMBERS_T<T>`) other than `MoveConstructible`, which is necessary to allow the proposed function call syntax. The expression is rather verbose: you spell it `std::meta::members(e).value` but one could expect a shorter `std::members(e)`. This is because we believe it will be easier for the programmers to understand the friendship (access) relationships that are hidden inside the feature. We need namespace `meta`, because it is its class templates that all other classes implicitly declare friendship to. We need member `value`, because it is this member that will be declared private or public: it will be easy to generate error message "Expression is invalid because `value` is declared private".

The following examples demonstrate the mechanics of the proposal.

```
class CC {
   int i,
   std::string s;

public:
   CC& operator=( CC const& rhs );
   CC& operator=( CC && rhs );
};
CC c;

auto tup = std::meta::members(c).value; // error: value is private

CC& CC::operator=( CC const& rhs ) {
   std::meta::members(*this).value = std::meta::members(rhs).value;
   // ok: meta::MEMBERS_T<CC const&> declares CC as friend
   return *this;
};

CC& CC::operator=( CC && rhs ) {
   meta::members(*this).value = meta::members(std::move(rhs)).value;
   return *this;
};
```

In the above example, we implement the assignment of `CC` in terms of tuple assignment. Even though in both cases the expression on the left-hand side of the assignment is a prvalue, tuple assignment is an overloaded function call, which is valid in this context. It is not possible to use this meta-programming feature to initialize members in the constructor.

Note that tuples created this way always contain references: no objects. They are not copies of objects, but *views*.

Operations like assignment or comparison involve not only member objects but also base classes. For these purposes, we provide a couple of other functions in namespace `meta`: `bases`, `bases_and_members`, `recursive_members` and `bases_with_virtuals`. Given the following struct `I`:

```
struct G { int mg; };
struct H { int mh; };
struct I : G, H { int mi, mj; };
```

We can define equality comparison in the following ways:

```
bool operator==( I l, I r ) {
   return std::meta::bases(l).value == std::meta::bases(r).value
      && std::meta::members(l).value == std::meta::members(r).value;
}
```

or, for short, because it is a common pattern:

```
bool operator==( I l, I r ) {
   return std::meta::bases_and_members(l).value
      == std::meta::bases_and_members(r).value;
}
```

The above solution calls `operator==` first for all `I`'s base classes, in order in which they are declared, and then for each member. It requires that classes `G` and `H` have equality operator already defined. If they do not, our proposal comes with another alternative:

```
bool operator==( I l, I r ) {
```

```
  return std::meta::recursive_members(l).value
    == std::meta::recursive_members(r).value;
}
```

function `recursive_members` creates a tuple that recursively extracts members from subclasses, so in our case `std::meta::recursive_members(l).value` renders a tuple containing references respectively to `i.mg`, `i.mh`, `i.mi`, `i.mj`. Function `meta::recursive_members` is expected to be used in situations where one wants to use inheritance to make one's classes' structure more clear:

```
struct Name {
  string first_name;
  string family_name;
};

struct Employee : Name {
  decimal salary;
}
```

but at the same time, when streaming, or when giving a sequential access, they want member objects from this class and a sub-class to be treated "equally" or "flatly". That is, if you want to treat struct `Employee` as a two element tuple, containing types `Name` and `decimal`, you need to use `meta::bases_and_members`; if you want to treat it as a three element tuple, containing two strings and a `decimal`, you need to use `std::recursive_members`.

Next example shows how we can 'automatically' implement streaming operation for our type. We need a generic function `apply` that would represent a compile-time recursion equivalent of for-each loop. It could be defined as follows.

```
template< typename Fun, int I, typename ... Args >
struct ApplyImpl {
  static void Run( Fun f, tuple<Args...> const& tup ) {
    f( get<(sizeof...(Args) - I)>(tup) );
    ApplyImpl<Fun, I - 1, Args...>::Run( f, tup );
  }
};

template< typename Fun, typename ... Args >
struct ApplyImpl<Fun, 0, Args...> {
  static void Run( Fun f, tuple<Args...> const& tup ) {
    ; // just to stop recursion
  }
};

template< typename Fun, typename ... Args >
void apply( Fun f, tuple<Args...> const& tup ) {
  ApplyImpl<Fun, sizeof...(Args), Args...>::Run( f, tup );
}
```

Thus defined function `apply` calls a generic functor for each tuple element starting from index 0 up to the number of tuple elements minus one. An example of such a generic functor for streaming out different types could look as follows.

```
struct StreamOut {
  std::ostream & out_;
  StreamOut( std::ostream & out ) : out_(out) {}

  template< typename T >
```

```
  void operator()( T const& v ) const {
    out_ << v << " ";
  }
};
```

Now, the streaming operator can be implemented as:

```
std::ostream & operator<<( std::ostream & out, CC c ) {
  apply( StreamOut{out}, meta::bases_and_members(c).value );
  return out;
}
```

In other words, the five functions from namespace `meta` provide a tuple interface for each user defined class or struct in a way that preserves encapsulation on one hand and on the other allows access to private members to those that are granted it already. Thus, any algorithm on tuples can be applied to any object of any (non-union) class type.

All 'meta' functions can be safely called in `noexcept` functions and in constant expressions:

```
struct Complex {
  double x, y;
  constexpr Complex( double x, double y ) noexcept : x{x}, y{y} {}
};

constexpr bool operator==( Complex a, Complex b ) noexcept {
  static_assert( noexcept(std::meta::members(a).value), "" );
  return std::meta::members(a).value == std::meta::members(b).value;
}
```

Details

Transformations of references

'Meta' expressions like `std::meta::members(e).value` return instances of `std::tuple` that contain only references. These reference types are deduced in the following way. For member sub-object `m` of object `o` it is `decltype((o.m))`. For base sub-object of type `S` of object `o`, it is `S` with the same reference and CV qualifiers as `decltype((o))`. This gives the same access (in terms of being const, volatile or rvalue-vs-lvalue) to subobjects as though they were accessed by names.

```
struct Person {
  const std::string name;
  int phone;
  int & reference;
  // ...
};

int global_ref = 0;

Person p { "John", 123456, global_ref };
auto pi = std::meta::members(p).value;
get<0>(pi) = "Jane"; // error: cannot assign to const std::string
get<1>(pi) = 999999; // ok
get<2>(pi) = 999999; // ok

const Person cp { "Jack", 4567890, global_ref };
auto cpi = std::meta::members(cp).value;
get<0>(cpi) = "Jane"; // error
```

```
get<1>(cpi) = 999999; // error
get<2>(cpi) = 999999; // ok: const-ness does not apply to member references
```

Limitations

The solution has limitations in case of protected sub-objects. That is, derived classes that have normal (by member name) access to base class's sub-objects do not have indexed access (using `std::meta`) to those members. We did not find any practical use case for protected member access. The use cases we want to address is for anyone to access a POD class, or for a class to access its own (private, protected or public) members.

Another limitation is that in case a type `T` has some non-static data members declared as private and some as public, the access via a tuple is private for all members, although a 'public' subset of those could be provided to everyone.

Virtual inheritance

Functions from namespace `meta` provide limited functionality for classes that inherit virtually, or inherit from classes that inherit virtually. This is because in those cases the unusual order of subobjects might render results surprising to the programmer. Consider the following example:

```
struct Name{
   std::string first;
   std::string last;
};

template< typename Base2 >
struct Record : Name, Base2
{
   std::string getFirstName() const {
      return get<0>(meta::recursive_members(*this).value);
   }
};
```

It is not uncommon to expect that base subobject of type `Name`, along with its members, will be ordered in memory before any members of the subobject of type `Base2`. But in case `Base2` inherits virtually, some other objects may come first. We are not even able to check that inside the template. If this other object that is stored as first happens to be of type `std::string` the compiler is not even allowed to report this as an error.

Similarly, in case of function `meta::bases`, a programmer might incorrectly assume that the two bases are disjoint and cause an operation that is supposed to be called only once for each base to be in fact called twice:

```
struct B { /*...*/ };

struct D1 : virtual B { /*...*/ };

struct D2 : virtual B { /*...*/ };

struct init_once // a generic functor
{
   template< typename T >
   void operator()( T& res ) const {
      res.initialize_once();
```

```
      apply( *this, std::meta::bases(res).value );
      // apply() as defined in one of the previous examples
   }
};

template< typename Base1, typename Base2 >
struct Resource: Base1, Base2
{
   void init() {
      init_once{}(*this);
   }
   // ...
};
```

In this case we want to initialize recursively each base subobject (only once). The algorithm will work fine as long as neither `Base1` nor `Base2` inherit virtually (directly or indirectly). But if they do, as in the example, `B` is initialized twice.

Not every virtual inheritance is *bad*, though. The notable exception is when virtual inheritance is used to indicate "implementation of the interface" in OO sense, and the class that we inherit virtually from is an "interface": it has no non-static member sub-objects, and some other constraints (like, has only public and protected member functions - but this is beyond our interest). We define a term *thin class*, which indicates a class with no non-static member sub-objects, except for zero-length anonymous bit-fields, whose base classes (if any) (derived virtually or not) are thin classes. While name "empty" appears more suitable it would collide with `std::is_empty` trait which has a slightly different meaning. Next term we define is *straight-member-order class:* it is a class whose all sub-classes (if any) are either (inherited virtually or not) thin classes or inherited non-virtually straight-member-order classes. This means that straight-member-order class does not have virtual bases (directly or indirectly), or it has virtual bases that are (almost) empty. Function `meta::recursive_members` is only specialized for straight-member-order class; for others a compile-time error is triggered. This solution prevents using virtual inheritance in general, but allows particular cases like the following:

```
struct Iface { // an OO interface
   virtual void fun() = 0;
   virtual ~Iface(){}
   Iface(Iface const&) = delete;
   void operator=(Iface) = delete;
};

struct G : virtual Iface {
   void fun() override;
   int mg;
};

struct H : G, virtual Iface {
   void fun() override;
   int mh;
};

G g;
meta::recursive_members(g)::value; // contains {mg, mh}
```

In order to avoid unexpected problems with classes that inherit virtually (directly or indirectly) on the one hand, and on the other give the possibility in rare cases to obtain the list of all classes that we derive from, we propose the following solution. Functions `meta::bases_and_members` and `meta::bases` do not compile if applied to objects not of straight-member-order class. In exchange,

there is a fifth function: `meta::bases_and_virtuals` that lists all direct bases for every class w/o constraints. There is no counterpart for `meta::bases_and_members` because the same effect can be obtained by concatenating the result of `meta::bases_and_virtuals` and `meta::members`. Thus we have a simple syntax for typical task, and a more verbose, but working, syntax for an unusual task; and we force the user to specify explicitly that he is aware of potential virtual inheritance surprises.

Bit-fields

Since it is not possible to bind a reference to a bit-field, the described functionality is not capable of handling bit-fields. When an attempt is made to use classes that contain bit-fields, we choose to explicitly signal an error at compilation time rather than silently skip the bit-field members. The only exception is made for zero-length anonymous bit-fields because they do not store any data that could be silently dropped. In this case such a zero-length bit-field is not treated as a 'member' when used with the 'meta' functionality.

Possible extensions

While we do not propose it, it is possible to extend the proposal so that any function, function template, class or class template defined in namespace `std::meta` automatically obtains access to every private or protected member of any class/struct. This would address the common problem of 'meta' libraries like Boost.Serialization[3] or ODB,[4] which require that if the programmer wants her type to participate in the framework effectively, she needs to put a friendship declaration in her class that would allow access to the framework. Quoting Boost.Serialization tutorial:

```
class gps_position
{
private:
   friend class boost::serialization::access;
   int degrees;
   int minutes;
   float seconds;
   // [...]
};
```

In contrast, if the 'meta' frameworks could just declare a function template in namespace `std::meta` (in this case language should allow extending the namespace by anyone) users of the frameworks wouldn't be forced to specify this additional friendship. This would violate the encapsulation of class types to some degree, however it would follow the rule that C++ saves programmers from inadvertent errors: not from malicious intentions.

Also, one could expect more of a compile-time introspection library. For instance instead of a tuple of references to members one could expect a tuple of pairs: member name and a reference to member. This way we would also be able to query the member name as string at compile time. Similarly, one could expect to be able to query any object of class type for its every member function that matches a given signature:

```
struct CalculatorSuite : Suite
{
   void TestCreation();
   void TestDestruction();
   void TestAssignment();
   // …
} mySuite;
```

```
// later:
using Test = std::function<void()>;
std::vector<Test> tests = std::meta::functions<void()>(mySuite).value;
for( auto & test : tests ) test();
```

This would be helpful in simplifying the generation of unit test frameworks.

One could also expect more member access functions like `public_members` which would provide access only to a subset of members of a given class.

We choose to limit our proposal to providing only five function templates for indexed access to class's subobjects at this time.

Alternatives

Daveed Vandevoorde presented a full compile-time reflection utility – Metacode – in N1471. Our proposal is a strict subset of N1471. In contrast, our proposal focuses only on one aspect of reflection and suggests only localized changes in the Standard Library.

A similar effect could be achieved with run-time reflection facility, where information about every class is contained in a run-time facility and using it incurs a certain runtime cost. We chose a trade-off typical to other common C++ solutions: one that uses template instantiations to favour run-time execution optimization at the cost of extending the compilation time.

Library interface

The proposed interface for obtaining tuples has a certain flaw. Programmers can mistakenly type a fairly meaningless expression `std::meta::members(o)` (without `.value` at the end). This may lead to some confusion. Even the authors of the proposal occasionally forgot to put the final "`.value`".

As an alternative, we could propose that the tuple (that represent a view of object `o`) could be accessed by shorter expression `std::meta::members(o)`. This would limit the functionality however. Without the additional member that can be declared private ("private" means "accessible only to friends") there is no way, in case of class with private members to guarantee that all friends have unlimited access via tuples, whereas at the same time non-friends are not allowed access. We could re-enable this encapsulation-preserving functionality if we introduced yet another, unprecedented, kind of compiler magic, where the same function may be not found during overload resolution in one place and may be found in another based only on the function in which the expression is invoked. This, in our opinion, would be hard to teach and learn. The solution we propose also requires compiler magic (implicit template specialization based on well-defined rules) but this magic is already familiar to C++ programmers due to type traits library. Also, implicit friend declaration appears easy to learn.

An even cleaner alternative would be to use new reserved keywords for getting the tuples. Keywords are a natural and elegant way of introducing compiler magic. However, this would require five new keywords, and since this proposal has the potential to expand, this may potentially require a big number of keywords. Library approach (even if it is magical) appears more suitable.

We could also propose a pure-library (no compiler magic) solution with limited capabilities. For every class that the programmer wants to participate in the reflection framework, she has to define (a perhaps member-) function `as_tuple` defined as:

```
auto as_tuple( R& r ) -> decltype(std::tie(r.a, r.b, r.c))
{
   return std::tie(r.a, r.b, r.c);
}
```

This is more-less the approach that Boost.Spirit took with macros like `BOOST_FUSION_ADAPT_STRUCT`, which we described in the introduction. Such solution has a number of limitations:

1. You have to specify members of your every type twice: first time when you define the type; second, when you define the adaptor function/macro. Compiler cannot guarantee that the two are in sync.
2. You cannot have access to private members, unless you are declaring `as_tuple` as a member function; but in that case, you cannot apply this technique retroactively to third-party types that you cannot modify.
3. In fact, you have to specify more than one function for each of your type for different "reference variants": `R&&`, `R const&`, etc. Or you could use a perfect-forwarding template and limit the range of allowed arguments with `std::enable_if`, but requiring this of ordinary programmers appears not to be the right way.

Dependencies

This proposal requires that changes proposed in N3305[5] are accepted into C++. The paper adds `constexpr` specifiers to a number of operations on tuples. If N3305 is not accepted this proposal can still be added to C++ in a limited form: by dropping the requirement that expressions involving 'meta' functions are core constants expressions. Alternatively, the requirements that expressions like `meta::members(e).value` are core constant expressions can be treated as implicit requirements on tuples.

Wording

In Table 14 — C++ library headers – add header `<meta>`.

Add new paragraph 20.10 after 20.9.7. (Current paragraph 20.10 (Compile-time rational arithmetic) becomes 20.11.)

20.10 Indexed access to class subobjcts [meta.indexed]

This sub-clause describes templates that may be used to obtain access to user defined non-union class types' subobjects in form of class template `std::tuple` (20.4).

20.10.1 Definitions [meta.indexed.def]

A *thin class* is a non-union class that has no non-static data members, except for zero-length anonymous bit-fields, and that has no non-thin base classes.

A *meta-compatible class* is a non-union class type that does not contain bit-field members of size different than zero.

A *straight-member-order class* is a meta-compatible class whose all base classes, if any, are either (inherited virtually or not) thin classes or inherited non-virtually straight-member-order classes.

A straight-member-order class is *all-publically-accessed* if all its non-static data members are public and all its non-virtual base classes are derived with public access and are *all-publically-accessed.*

Given `tuple` types P_1, P_2, ..., P_N, let `CONCAT<`P_1`, `P_2`, ..., `P_N`>` denote a `tuple` type `decltype(tuple_cat(declval<`P_1`>(), declval<`P_2`>(), ..., declval<`P_N`>()))`.

Given that `T` denotes a reference type to a possibly cv-qualified non-union class type `U`, and `B` denotes a non-union class type that `U` is derived from, let `BASE_TYPE(B, T)` denote a reference type defined in the following way:

— if `T` is an rvalue reference then it is an r-value reference to `B` with the same cv-qualifiers as `T`, otherwise

— it is an lvalue reference to `B` with the same cv-qualifiers as `T`.

Given that `OBJREF` denotes a reference to an object of a possibly cv-qualified type `U`, and `B` denotes a base class of `U`, let `FORWARD_BASE(B, OBJREF)` denote a reference to base subobject of `OBJREF` of type `BASE_TYPE(B, REFTYPE)`, where `REFTYPE` is the type of reference `OJBREF`.

20.10.2 Header `<meta>` synopsis [meta.indexed.synop]

```
namespace std {
  namespace meta {
    template <class T>
    constexpr auto members(T&& obj) noexcept -> see below;

    template <class T>
    constexpr auto bases(T&& obj) noexcept -> see below;

    template <class T>
    constexpr auto bases_and_members(T&& obj) noexcept -> see below;

    template <class T>
    constexpr auto bases_with_virtuals(T&& obj) noexcept -> see below;

    template <class T>
    constexpr auto recursive_members(T&& obj) noexcept -> see below;

  } // namespace meta
} // namespace std
```

20.10.3 Function template `members` [meta.indexed.mem]

For every type `T` where `decay<T>::type`, denoted by `U` in this subclause, names a meta-compatible class, let `MEMBERS_T<T>` denote a non-union class type with the following properties.

- It is declared in namespace `std::meta`.
- It satisfies requirements `MoveConstructible`.
- It is a literal type.
- It has a non-static data member `value` of type `std::tuple<`Arg_1`, `Arg_2`, ..., `Arg_N`>` with N equal to the number of non-static non-bit-field data members declared in type `U`, and Arg_i equal to `decltype((std::declval<T>().`mem_i`))`, and mem_i is the name of the *i*-th non-static non-bit-field data member defined in `U`.

– If all non-static data members of `U` are public, member `value` is declared with public access, otherwise it is defined with private access.

– It declares as friend class `U` as well as any function, function template, class or class template that `U` declares as friend.

```
template <class T>
  constexpr auto members(T&& obj) noexcept -> MEMBERS_T<T&&>;
```

Returns: prvalue of type `MEMBERS_T<T&&>` containing object `value` whose every *i*-th element is a reference to `std::forward<T>(obj).`mem_i, where mem_i is the name of *i*-th non-static non-bit-field data member of `U`. [*Note:* satisfying this requirement requires breaking the usual access rules if mem_i is declared as private or protected -- *end note*]

Remarks: if `decay<T>::type` does not denote a meta-compatible class, or if it is an incomplete type, the program is ill-formed.

For any core constant expression (5.19) `ce` whose type is a meta-compatible class expression `meta::members(ce).value` shall be a core constant expression. For any expression `e` whose type is a meta-compatible class for which expression `noexcept(e)` evaluates to true, expression `noexcept(meta::members(e).value)` shall evaluate to true.

20.10.4 Function template `bases` [meta.indexed.base]

For every type `T` where `decay<T>::type`, denoted by `U` in this subclause, names a meta-compatible class, let `BASES_T<T>` denote a non-union class type with the following properties.

– It is declared in namespace `std::meta`.

– It satisfies requirements `MoveConstructible`.

– It is a literal type.

– It has a non-static data member `value` of type `std::tuple<`Arg_1`, `Arg_2`, ..., `Arg_N`>` with *N* equal to the number of base subobjects of `U`, and Arg_i equal to `BASE_TYPE(base`$_i$`, T&&)`, and `base`$_i$ being the name of the *i*-th base subobject of `U`.

– If all base subobjects of `U` are public, member `value` is declared with public access, otherwise it is defined with private access.

– It declares as friend class `U` as well as any function, function template, class or class template that `U` declares as friend.

```
template <class T>
  constexpr auto bases(T&& obj) noexcept -> BASES_T<T&&>;
```

Returns: prvalue of type `BASES_T<T&&>` containing object `value` whose each *i*-th element is a reference to `FORWARD_BASE(`B_i`, obj)`, where B_i is the name of *i*-th base class of `U`. [*Note:* satisfying this requirement requires breaking the usual access rules if U inherits from B_i with private or protected access -- *end note*]

Remarks: if `decay<T>::type` does not denote a straight-member-order class, or if it is an incomplete type, the program is ill-formed.

For any core constant expression `ce` whose type is a straight-member-order class expression `meta::bases(ce).value` shall be a core constant expression. For any expression `e` whose type is a straight-member-order class for which expression `noexcept(e)` evaluates to true, expression `noexcept(meta::bases(e).value)` shall evaluate to true.

20.10.5 Function template `bases_and_members` [meta.indexed.basemem]

For every type `T` where `decay<T>::type`, denoted by `U` in this subclause, names a straight-member-order class, let `BASES_AND_MEMBERS_T<T>` denote a non-union class type with the following properties.

- It is declared in namespace `std::meta`.
- It satisfies requirements `MoveConstructible`.
- It is a literal type.
- It has a non-static data member `value` of type `CONCAT<TB, TM>`, where `TB` is equal to `BASES_T<T&&>` and `TM` is equal to `MEMBERS_T<T&&>`.
- If all non-static data members and all base subobjects of `U` are public, member `value` is declared with public access; otherwise it is defined with private access.
- It declares as friend class `U` as well as any function, function template, class or class template that `U` declares as friend.

```
template <class T>
  constexpr auto bases_and_members(T&& obj) noexcept
  -> BASES_AND_MEMBERS_T<T&&>;
```

Returns: prvalue of type `BASES_AND_MEMBERS_T<T&&>` containing object `value` whose every *i*-th element is a reference to `FORWARD_BASE(`B_i`, obj)` for $1 \le i \le M$, and a reference to `std::forward<T>(obj).`mem_{i-M}, for $M < i \le M+N$, where M is the number of base subobjects of `U`, mem_{i-M} is the name of $(i-M)$-th non-static non-bit-field data member of `U`, B_i is the name of *i*-th base class of `U`. [*Note:* satisfying this requirement requires breaking the usual access rules if `U` inherits from B_i with private or protected access or if mem_{i-M} is a private or protected data member -- *end note*]

Remarks: if `decay<T>::type` does not denote a straight-member-order class, or if it is an incomplete type, the program is ill-formed.

For any core constant expression `ce` whose type is a straight-member-order class expression `meta::bases_and_members(ce).value` shall be a core constant expression. For any expression `e` whose type is a straight-member-order class for which expression `noexcept(e)` evaluates to true, expression `noexcept(meta::bases_and_members(e).value)` shall evaluate to true.

20.10.6 Function template `bases_with_virtuals` [meta.indexed.virt]

```
template <class T>
  constexpr auto bases_with_virtuals(T&& obj) noexcept -> BASES_T<T&&>;
```

Returns: prvalue of type `BASES_T<T&&>` containing object `value` whose each *i*-th element is a reference to `FORWARD_BASE(`B_i`, obj)`, where B_i is the name of *i*-th base class of `U`, and `U` is `decay<T>::type`. [*Note:* satisfying this requirement requires breaking the usual access rules if U inherits from B_i with private or protected access -- *end note*]

Remarks: if `decay<T>::type` does not denote a meta-compatible class, or if it is an incomplete type, the program is ill-formed.

For any core constant expression `ce` whose type is a meta-compatible class expression `meta::bases_with_virtuals(ce).value` shall be a core constant expression. For any expression `e` whose type is a meta-compatible class for which expression `noexcept(e)` evaluates to true, expression `noexcept(meta::bases_with_virtuals(e).value)` shall evaluate to true.

20.10.7 Function template `recursive_members` [meta.indexed.recurs]

For every type `T` where `decay<T>::type`, denoted by `U` in this subclause, names a straight-member-order class, let `RECURSIVE_MEMBERS_T<T>` denote a non-union class type with the following properties.

- It is declared in namespace `std::meta`.
- It satisfies requirements `MoveConstructible`.
- It is a literal type.
- It has a non-static data member `value` of type `CONCAT<`TB_1`, `TB_2`, ..., `TB_N`, TM>`, where `TM` is equal to `MEMBERS_T<T&&>`, TB_i is equal to `RECURSIVE_MEMBERS_T<BASE_TYPE(`B_i`, T&&)>`, B_i is the *i*-th base subobject of `U`.
- If `U` is *all-publically-accessed*, member `value` is declared with public access; otherwise it is defined with private access.
- It declares as friend class `U` as well as any function, function template, class or class template that `U` declares as friend.

```
template<class T>
  constexpr auto recursive_members(T&& obj) noexcept
  -> RECURSIVE_MEMBERS_T<T&&>;
```

Returns: prvalue of type `RECURSIVE_MEMBERS_T<T&&>` containing object `value` equal to the concatenation of tuples $tupB_1$, $tupB_2$, ..., $tupB_N$, `tupM`, where `tupM` is equal to `members(forward<T>(obj)).value`, $tupB_i$ is equal to `recursive_members(FORWARD_BASE(`B_i`, forward<T>(obj))).value`, and B_i is the *i*-th base subobject of `U`. [*Note:* satisfying this requirement requires breaking the usual access rules if any of the member subobjects referenced by the returned tuple is declared as private or protected -- *end note*]

Remarks: if `decay<T>::type` does not denote a straight-member-order class, or if it is an incomplete type, the program is ill-formed.

For any core constant expression `ce` whose type is a straight-member-order class expression `meta::recursive_members(ce).value` shall be a core constant expression. For any expression `e` whose type is a straight-member-order class for which expression `noexcept(e)`

evaluates to true, expression `noexcept(meta::recursive_members(e).value)` shall evaluate to true.

20.10.8 Special semantics [meta.indexed.spec]

Implementation need not specialize and/or instantiate functions `members`, `bases`, `bases_and_members`, `bases_with_virtuals` and `recursive_members`, or create instances of classes these functions return, if it is able to provide in an another way the correct `tuple` return value for expression `meta::members(e).value`, `meta::bases(e).value`, and so on, and ensure that access to member `value` obeys normal access rules.

Implementation need not instantiate `tuple` types returned by sub-expressions such as `meta::members(e).value`, `meta::bases(e).value`, and so on, if it can compute the result and side effects of the full expression by other means [*Example:* for the following code:

```
struct TT{
   int a, b;
};

bool operator==( TT const& x, TT const& y ) {
   return std::meta::members(x).value == std::meta::members(y).value;
}
```

the definition of operator== can be replaced by the implementation with

```
bool operator==( TT const& x, TT const& y ) {
   return x.a == y.a  &&  y.a == y.b;
}
```

-- *end example*]

Impementability

This language feature has not been implemented. It cannot be implemented as a library because of two kinds of required compiler magic:

1) The proposed classes require access to all members of all user-defined class types that violates the current class member access rules.
2) Class specializations need to be automatically generated based on non-trivial (although automatic) rules.

In order to emulate the proposal with a library, we emulate 1) by putting explicit friend declarations inside every tested class and 2) by explicitly writing each specialization required by the example.

Acknowlegements

Daniel Krügler provided many helpful suggestions, corrections and comments on this paper.

References

1. Daveed Vandevoorde: N1471 – “Reflective Metaprogramming in C++”.
2. Joel de Guzman, Hartmut Kaiser: Boost.Spirit library v. 2.5.1 – documentation at http://www.boost.org/doc/libs/1_48_0/libs/spirit/doc/html/index.html.

3. Robert Ramey: Boost.Serialization library – documentation at http://www.boost.org/doc/libs/1_48_0/libs/serialization/doc/index.html.
4. ODB library by Code Synthesis Tools CC – documentation at http://www.codesynthesis.com/products/odb/doc/manual.xhtml.
5. Benjamin Kosnik, Daniel Krügler: N3305 – "Constexpr Library Additions: utilities, v2"