Document number: N3951
Date: 2014-02-07
Project: Programming Language C++, SG7, Reflection
Reply-to: Cleiton Santoia Silva <cleitonsantoia@gmail.com> and Daniel Auresco <auresco@gmail.com>

C++ type reflection via variadic template expansion

Cleiton Santoia Silva and Daniel Auresco

Abstract

From a type T, gather member name and member type information, via variadic template expansion.

Contents

1 Introduction

1.1 Keywords

From a type T, obtain static typed reflection adding 2 language constructs:

1. An instruction `typename<T>...` that expands members identifiers of type T into a variadic template. Each type of n-th element of `typename<T>...` is a const char* and each n-th value is the identifier of n-th member of T, expressed in UTF-8 encoded;

2. An instruction `typedef<T>...` that expands members of type T into a variadic template (in the same order of `typename<T>...`). Each n-th type of `typedef<T>...` is the type of the n-th member of T and each n-th value is a pointer to n-th member of T, or a value if member is a constexpr member or enum item;

1.2 Simple Sample Code

1. Given:

```
using namespace std;
namespace some_namespace {
 struct some_struct {
  int x, y;
 };
}
vector<string> names{ typename<some_namespace::some_struct>... };
auto mytuple = make_tuple(typedef<some_namespace::some_struct>...);
```

2. Compiler will turn the last two lines into this:

```
vector<string> names { "some_namespace::some_struct","x","y" };
auto mytuple = make_tuple(
                (some_namespace::some_struct*)nullptr,
                &some_struct::x,
                &some_struct::y);
```

3. What it's not:

   - we don't need no new keywords
   - we don't need no AST control
   - no magic namespaces or runtime structures
   - all we need by now just another keyword syntax
   - some type traits are welcome, but we are not trying to standardize them now

4. That's it.

2 Motivation and Scope

The *use cases* session of [Spe12, N3403] sums this:

- Serialization
- Parallel hierarchies
- Delegates
- Getter/Setter generation
- Generating user interfaces to call functions and constructors

Adding motivation scope from [SC13, N3814]:

- Generation of common functions
- Type transformations
- Compile-time context information
- Enumeration of other entities

Except "Compile-time context information" we can get all of above

3 Impact on standard

1. This is a proposal of static-compile-time-reflection, totally strongly typed, all work made by compiler using variadic template expansion language feature;
2. `typename<T>...` and `typedef<T>...` will be new ways to use existing keywords, that are not used today, only *`typename type`* and *`typedef type id`* are used in C++11;
3. The use of ellipsis here helps to indicate a variadic expansion.

4 Design Decisions

4.1 Initial considerations

1. Clearly, nobody expects to get full power of reflection just by using new feature in its raw form. Some type traits `is_virtual_base<>` see [Spe09, N2965] or `is_protected<>` or `is_inline<>` are welcome. Same about libraries, like `std::tuple` and BOOST META libraries or future lib of run time structures, possibly some `std::class_info` and `std:member_info` things. Again, it doesn't mean they are not useful, surely they are and surely they must be standardized as well, but since they are not needed by now, and there is no "current practice" of C++ introspection, we just didn't give this step yet;
2. An important performance requirement from [Spe12, N3403]: the size of variadic parameter list may be huge, so we present some expected minimum limits your compiler should handle about classes taken from Annex B from [ISO13][N3690]
   - Data members in a single class [16 384].
   - Enumeration constants in a single enumeration [4 096].
   - Levels of nested class definitions in a single member-specification [256].
   - Direct and indirect base classes [16 384].
   - Direct base classes for a single class [1 024].
   - Members declared in a single class [4 096].
   - Final overriding virtual functions in a class, accessible or not [16 384].

- Direct and indirect virtual bases of a class [1 024].
- Static members of a class [1 024].

Now, some template limits

- Template arguments in a template declaration [1 024].
- Recursively nested template instantiations, including substitution during template argument deduction (14.8.2) [1 024].

It doesn't mention directly variadic template parameter expansion but, anyway, one can see a big difference, even if these are "minimum" and in practice we succeeded in compile variadic templates up to 32K parameters on GCC 4.8.1, would be nice if we had a way of picking from `typename<T>...` and `typedef<T>...` what is needed and skip unnecessary things. We considered an instruction `typename<T requires U>...` and `typedef<T requires U>...` where U is a boolean constexpr like a "concept" that will be applied to each element of T, to choose if it will be retrieved or not. This way, we really hope to solve the problem hijacking keywords and semantics from "concepts" without need for another new reserved word.

3. In effort to recurrently and orthogonally apply `typedef<T>...` in wherever item that `typedef<T>...` returns, in this document, we reach a point to dig a bit down in some aspects that may be a postponed or ripped off, like introspecting functions,constructors, templates and namespaces, but anyway it's better bring these ideas to appreciation and critique then just skipped and hidden because they are "too problematic".

4. About `typename<T>...`

   - Encoding should be utf-8, but remember: utf-8 != ASCII, only chars in range 0x00-0x7f are equal (the most common C++ ones in english language), utf-8 may expands to 16, 24 and 32bits; see annex E [charname.allowed] [ISO13, N3690]
   - The type of template parameters as `const char*` may lead to few issues, one important is that parameter value of this kind of template must have extern linkage "compiler magic string" is the technical proper name of it, so the names of types and member must be that way;
   - Type names must be fully qualified, expanding all namespaces to avoid ambiguity if you will create a map of classes by name. see "simple member function case" code sample 5;
   - Member names may not be qualified, we don't see the need for full namespaces into each member, that you already knows what class it belongs to, when you call `typename<T>...` see code sample "simple member function case" 5 and see code sample 6 "More complete class use case";

5. Basically we propose to gather everything from a type, not only members, but base classes, internal classes, typedefs enums and constant values, it appear to be a mess (may be it is), calling it brings everything in one shot and then, you pick what you need or by using `typedef<T requires U>...` explained above on item 2 or using something like Boost::MPL `filter_view<s, pred>` where pred is some type trait. Otherwise we may need many keywords, one for base classes, another for members, another for inner classes etc, even if those are type traits, all of that must return a variadic expansion, its not the intended way of using type traits today that is: single response, either a boolean for "is...something" traits or a type sanitizer like "remove_cv";

6. IMHO, comparing to other related documents:

   (a) Variadic templates includes all of [TK13, N3815] functionality that enumerate "enums", and their names, but in a different way;
      - It avoids out-of-bounds problems that enumerators have;
      - It gets enum and other types in a same "standard" way;
      - A drawback is that template expansion may be huge;

   (b) Here we got same basic idea of [Krz13, N3326], but few relevant differences, the major is: this is statically resolved, without the need (or the possibility) of an instance of object to get a reflection, and no namespaces needed either. However, you can easily use member pointers returned by `typedef<T>...` to get those information, once you have an object instance;

   (c) This proposal includes all of [Spe09, N2965] functionality, but;

- [Spe09, N2965] has way of get base classes from a class, this feature is clearly easier than what we are proposing here, but, we steak to the idea to standardize the same "method" for base classes and other things;
- We did not addressed directly how we get information if a base class is virtual or protected or public, etc. It would be better to use type traits like new `is_virtual_base<>` and `is_protected_base<>` and so on;

(d) About "From a struct to a struct of arrays" topic in ISOCPP reflection forums, really interesting idea, we think that a Boost::MPL `filter_view<s, pred>` where pred is `is_member_object<>` can be used to pick only object members from `typedef<T>...` and implement a tuple-like class that permit some "empty" columns for skipped members, to make an struct of arrays or vectors. Passing the barrier of introspection and making this a "really" reflection case;

(e) Almost nothing of [BCAS12, N3410] dynamic approach you will find here, this is completely different;

7. We think it's important talk about this: We are NOT proposing any kind of run-time structure for deal with data, let us consider, a replacement for our variadic expansion, a struct like:

```
template <typename _CharT, typename _MemberT>
struct reflect_member {
 typedef _Char      id_char_type;
 typedef _MemberT   member_type;

 id_char_type       identifier;
 member_type        member_ptr;
};
```

That allows good things like easy-to-use variadic locked expansion, of each member type information together with member name, but that will impose compiler magic to fill this particular structure, so all vendors must " deal with it".We don't think that's a good idea, so we really prefer not impose any kind of structure and left vendors choose their own, if it's necessary;

4.2 More code

We used this code fragment in every sample to make code shorter.

Listing 1: Check macro

```
#include <string>
#include <tuple>
#include <vector>
#include <type_traits>
#include <assert.h>

#define CHECK(expected_, index_, tuple_)  \
     static_assert(std::is_same<expected_, \
     std::tuple_element<index_, decltype(tuple_)> \
     ::type>::value, "Problem ?")
using namespace std;
```

Listing 2: Simple struct use case

```
namespace space {
 struct some_struct {
  int  count;
  char delimiter;
 };

 void reflect_check_struct() {
  auto mytuple = make_tuple(typedef<some_struct>...);
```

```
  CHECK(some_struct*, 0, mytuple);
  CHECK(int some_struct::*, 1, mytuple);
  CHECK(char some_struct::*, 2, mytuple);

  vector<string> names{ typename<some_struct>... };

  assert(names[0] == "space::some_struct");
  assert(names[1] == "count");
  assert(names[2] == "delimiter");
 }
}
```

Listing 3: Simple enum case

```
namespace space {
enum Y : unsigned {
 A = 10,
 B = 88
};

enum class Z : int {
 A = 33,
 B = 34
};

void reflect_check_enum() {
 auto mytupleY = make_tuple(typedef<Y>...);

 CHECK(Y*, 0, mytupleY);
 CHECK(unsigned, 1, mytupleY);
 CHECK(unsigned, 2, mytupleY);

 assert( nullptr == std::get<0>(mytupleY));
 assert( 10 == std::get<1>(mytupleY) );
 assert( 88 == std::get<2>(mytupleY) );

 vector<string> namesY{ typename<Y>... };

 assert( namesY[0] == "space::Y" );
 assert( namesY[1] == "::A" ); // A is global
 assert( namesY[2] == "::B" ); // B is global

 auto mytupleZ = make_tuple(typedef<Z>...);

 CHECK(Z*, 0, mytupleZ);
 CHECK(int, 1, mytupleZ);
 CHECK(int, 2, mytupleZ);

 assert( nullptr == std::get<0>(mytupleZ));
 assert( 33 == std::get<1>(mytupleZ) );
 assert( 34 == std::get<2>(mytupleZ) );

 vector<string> namesZ{ typename<Z>... };

 assert( namesZ[0] == "space::Z" );
 assert( namesZ[1] == "A" ); // A is local
 assert( namesZ[2] == "B" ); // B is local
}
```

```
}
```

Listing 4: Simple function type use case

```
int foo_int( int val ) {
 return val + 1;
}

void reflect_check_fun() {
 auto mytuple = make_tuple(typedef<foo_int>...);

 CHECK(int*, 0, mytuple);
 CHECK(int*, 1, mytuple);

 vector<string> names{ typename<foo_int>... };

 assert( names[0] == "::foo_int" );
 assert( names[1] == "val" );
}
```

Listing 5: Simple member function use case

```
struct X {
 int foo_int( int val ) const {
     return val + 1;
 }
};

struct W : X {
 static float foo_float(int h) {
  return h * h;
 }
};

void reflect_check_mem_fun() {
 auto mytuple = make_tuple(typedef<X::foo_int>...);

 CHECK(int*, 0, mytuple);
 // we need to get the constness and the function of member
 CHECK(const X*, 1, mytuple);
 CHECK(int*, 2, mytuple);

 std::vector<std::string> names{ typename<X::foo_int>... };

 // name of function comes in position 0
 assert ( names[0] == "X::foo_int" );
 // since it is a member function, we should put the class that it belongs
 // somewhere, commonly under-the-hood, compilers passes 'this' pointer as
 // first parameter to non-static member functions.
 assert ( names[1] == "this" );
 assert ( names[2] == "val" );

 // get foo_int from W, instead of X
 auto mytuple = make_tuple(typedef<W::foo_int>...);

 CHECK(int*, 0, mytuple2);
 CHECK(const X*, 1, mytuple2);
 CHECK(int*, 2, mytuple2);

 std::vector<std::string> names2{ typename<W::foo_int>... };
```

```
 assert ( names2[0] == "X::foo_int" ); // exactly as X::foo_int
 assert ( names2[1] == "this" );
 assert ( names2[2] == "val" );

 auto mytuple = make_tuple(typedef<W::foo_float>...);

 CHECK(float*, 0, mytuple3);
 CHECK(int*, 1, mytuple3);

 std::vector<std::string> names3{ typename<W::foo_int>... };
 assert ( names3[0] == "W::foo_float" );
 assert ( names3[1] == "h" );
}
```

Listing 6: Simple base class use case

```
struct A {
 int a;
 int foo_int( int v ) { return v + 1; };
};

struct B : A {
 int b;
 float foo_float( float val ) { return val + 1; };
};

void reflect_check_base_class() {
 auto mytuple = make_tuple(typedef<B>...);
// auto mytuple = std::make_tuple( (A*)nullptr, (B*)nullptr, &B::b, &B::foo_float );

 CHECK(A*, 0, mytuple);
 CHECK(B*, 1, mytuple);
 CHECK(decltype(&B::b), 2, mytuple);
 CHECK(decltype(&B::foo_float), 3, mytuple);

 std::vector<std::string> names{ typename<X::foo_int>... };
// std::vector<std::string> names{ "A", "B", "b", "B::foo_float" };

 assert ( names[0] == "::A" );
 assert ( names[1] == "::B" );
 assert ( names[3] == "b" );
 assert ( names[4] == "foo_float" );
}
```

Listing 7: Using it wrongly

```
struct point {
        int x, y;
        decl_type(typedef<point>...) z; // illegal [point] is not fully defined
};

struct Ok {
        int x;
        int z;
};

struct Ok_son : Ok {
        int x;
        int z;
```

```
};

Ok* ok_var = new Ok_son();
vector<string> names { typename<*ok_var>...  }; // illegal dynamic call
```

4.3 More considerations

4.3.1 What T can be in `typedef<T>...`:

1. "T" must be a typename, enum, typedef, a class, a struct, a concrete template instantiation, function, member function, union, or a constant wherever that is or has a compile time defined type expression, everything you can put in a `decltype()`;
2. T is constexpr, as you expect `typedef<T>...` must return it's type and value;
3. T is a primitive constant, as `typedef<4>...` or `typedef<"hi">...`, it must return it's type and value;
4. T is a member of enum, it must be treated as constexpr;
5. T is a constant object, as `typedef<const_obj>...`, it return type must be a pointer type of const_obj and return value must be a pointer to that constant, so you can freely define the value of that const in other compilation unit;
6. T parameter in `typedef<T>...` cannot be an instance. So we must pick real type of variable in run time, this is way harder than `typeid(instanceX)` construct that already exists, this changes everything to runtime, so compiler must deals to all descendant of declared type of "instanceX" variable, that may be different classes, leading to different signatures of `typedef<T>...` that cannot be "compiled" at "runtime", right ? If you want this, you may defend [BCAS12, N3410]. Anyway you could call `typedef<decltype(instanceX)>...`, that's legal;
7. T is constant data member it return type must be a member-pointer type of member and return value must be a pointer to that member-typed constant;
8. Since the call on `typename<T>...` gets typed member function pointers, one may argue that is not need to reflect a single function itself, but introspecting a function is important thing, if you need to reflective call a function with many parameters in RPC (remote procedure call) system when you need to serialize/deserialize and set parameters by name, you surely need to know the parameter names and types of those functions, `typename<T>...` when T is a function is not a big deal even when we get nameless parameters, simply return zero-sized strings, but `typedef<T>...` when T itself is a function leads to many problems:

   - How reflect the type of a parameter, clearly it's not a good idea return a type itself, since the value part of `typedef<T>...` should be of that type so, some instance should be created, what is no good;
   - How reflect const parameters ?
   - How reflect & parameters ?
   - How reflect const& parameters ?
   - How reflect && parameters ?
   - How reflect if a function is a member function or static or global ?
   - How reflect if a function is a constant member function ?
   - In which order should go the return and the parameters of a function ?

   Then we present some solutions:

   - Global functions will be reflected is this order:`typename<T>...` returns first the name of function prefixed with "'::", than the parameter names, while `typedef<T>...` returns pointer to type of return of function than pointers to the types of parameters; We cannot return the parameters themselves, because that will impose the creation of an object of each type of each parameter of the function, in the variadic return. We cannot do this. What leads to another problem: pointer types will become pointer to pointer types,

reference types now become pointer to reference types, and if the function parameter is a &&, to create a pointer to a && as result of type `typedef<T>...` the `v` parameter in `void f(int&& v)` become a `int*&&` this type is allowed in GCC 4.8.1. It's important get a type that might not be the correct type of the function, but it in a way that can be easily and, without any ambiguity, mapped into correct parameter type to make a call to that function, finally, all values of `typedef<T>...` when t is a function should be nullptr;

- Member functions will be reflect is this order:`typename<T>...` returns first the name of function unqualified inside the class, than "this" string, than the parameter names, while `typedef<T>...` returns pointer to type of return of function, than the class pointer that the function is enclosed together with the constness of the function, than the pointers of types of parameters. see code listing "simple member function" 5
- Static member functions will be reflect is this order:`typename<T>...` returns first the name of function unqualified inside the class, than the parameter names, while `typedef<T>...` gets pointer to type of return of function than, the pointer to types of parameters. see code listing "simple member function"5
- Inherited members will be addressed with it's respective parents, if class A has a member X and class B inherits from A and you call for `typedef<B::X>` you should get the same as `typedef<A::X>`, the names and types will reflect A class, not B, see code listing "simple member function" 5
- Functions with default parameters, should be reflect only by full parameter list;
- No mention to throws clause neither "noexcept" they are not part of "common" type system of C++, even if you can use noexcept operator as a kind of type_trait, its not returned by `typedef<T>...`;

9. We think that should be possible use on "primary types", `typedef<T>...` will get cast operators, common operators, constructors and destructors. We think (not to deeply) that casts to convertible types must appear as well, if you call `typedef<int>...` you must receive cast operators to double/float/long... etc. Since if you got an user defined cast operator in an user class you will expect that member in the type list. The problem here is, how the compiler will generate function pointers to these member operators. We do not have a good understanding of the guts of compilers to give a good insight on this. Anyway we prefer to put some insights of what should be returned here, and after some discussion let them be sorted out of proposal than just skip this difficult thing, that other papers should deal anyway. The directive here is to skip members that is unnecessary or impossible to use in that type, like assignment operators in a constant type;

   - `typedef<void>...` this should not return any data members neither function members, only constructors, and void should be only "incomplete type" allowed;
   - `typedef<nullptr_t>...` this should not return any data members neither function members, may be not even constructors here;
   - `typedef<T*>...` this should return assignment operators, dereference operator*, ++,– and conversions from it to it's base classes, but not return any members of T;
   - `typedef<const T>...` this should return same as `typedef<T>...` but all members as const (either functions or data) and no assignment operator;
   - `typedef<const T*>...` this should return same as `typedef<T*>...` ++, – and dereference operator* changed to const, conversions from it to it's base const classes, but not assignment operators;
   - `typedef<T&>...` this should return same as T, but without constructors;
   - `typedef<const T&>...` this should return same as `typedef<const T>...`, but without constructors;
   - If the member is `volatile` and `mutable` does not changes if a member should be retrieved or not, but all data members should be "volatized" or "mutabilized" in the return of `typedef<volatile T>...`

10. Abstract classes should be allowed. You should be able to get concrete and pure virtual members pointer of an abstract class, but not constructors.

11. Reflect lambdas like `typedef<decl_type([](int i)->int return i+1; )>...` are fully problematic.

    - We may Reflect the compiler-vendor-specific type of lambda, it does not sound as a good solution, it may lead to undefined-behavior situation, or worse, someone may try to standardize underline structures for lambdas, however, since they are typed, it must return it's type information.

- The most important here is that whatever returns from above lambda, somewhere must exists a function pointer, that may be called, with one integer and will return a integer (in this case). Which leads to another problem, if you use that syntax above it must compile the lambda inline in order to make a "reflective" call to it.
- Lambdas that capture values are a bigger problem, they must have a capture point inside an "alive" functions, what will not happens in compilation context.
- If lambdas that captures nothing can be treated as static functions we can avoid the problem with them:

Listing 8: Without capturing anything

```
auto lamb = [](int i) -> int { return i*i; };
auto tuple = make_tuple( typedef<decl_type(lamb)>... );
int x = 10;
int y = tuple.get<0>(x); // we donÂ´t know exactly if it's in pos 0
```

Listing 9: Using it wrongly

```
auto lamb = [&x](int i) -> int { return i+x; }; // nope... it captures
auto tuple = make_tuple( typedef<decl_type(lamb)>... );
int x = 10;
int y = tuple.get<0>(x);
```

We think this case will be possible

  - The standard says that we should get a function call operator, but not where: [ISO13, N3690] `5.1.2.5 [expr.prim.lambda]` The closure type for a non-generic lambda-expression has a public inline function call operator (13.5.4) whose parameters and return type are described by the lambda-expression's parameter-declaration-clause and trailing-return-type respectively.
  - We can also use as a pointer to function: [ISO13, N3690] `5.1.2.6 [expr.prim.lambda]` For a generic lambda with no lambda-capture, the closure type has a public non-virtual non-explicit const conversion function template to pointer to function. The conversion function template has the same invented template-parameter-list, and the pointer to function has the same parameter types, as the function call operator template. The return type of the pointer to function shall behave as if it were a decltype-specifier denoting the return type of the corresponding function call operator template specialization. [Note: If the generic lambda has no trailing-return-type or the trailing-return-type contains a placeholder type, return type deduction of the corresponding function call operator template specialization has to be done. The corresponding specialization is that instantiation of the function call operator template with the same template arguments as those deduced for the conversion function template.]
  - [ISO13, N3690] `5.1.2.20 and 5.1.2.21 [expr.prim.lambda]` 20. The closure type associated with a lambda-expression has a deleted (8.4.3) default constructor and a deleted copy assignment operator. It has an implicitly-declared copy constructor (12.8) and may have an implicitly declared move constructor (12.8). [Note: The copy or move constructor is implicitly defined in the same way as any other implicitly declared copy or move constructor would be implicitly defined. -end note] 21. The closure type associated with a lambda-expression has an implicitly-declared destructor (12.4).

12. This is not a "core feature" of proposal, but if the case of T to be a variadic "T...", we can sequentially expand all types of "T...", even if the result will be more difficult to interpret by libraries, many classes at once may be a good thing if you want to "register" all of them into your serialization engine, and on top of that, we don't know why, but we like orthogonality :

    - `typedef<typedef<T>...>...` will be allowed;
    - `typedef<typename<T>...>...` will to;
    - `typename<typename<T>...>...` will to;
    - `typename<typedef<T>...>...` will to;

13. Templates are hard, try to reflect template functions or classes are difficult; here some thoughts: First of all, it's pointless to include only the declaration of a template function or a template class that has not a concrete type, since they don't generate code anyway.

(a) For class templates

- plan A: We thing that calling `typedef<std::vector>...` without any parameters we might return `vector<int>, vector<char>, vector<myclass>... etc` but the problem here is: when the compiler is compiling one compilation unit, like one .cpp file, that includes <vector> in one to one .obj file, the compiler does not knows all instances of vector<X> that you are using in other compilation units. But if the compiler return only instances in the current compilation unit someone will find a use for it, like including all widget classes headers inside one big .cpp in the way that compiler will knows all important instances;
- plan B: allow only concrete types on `typedef<T>...`

(b) Same for function templates:

- plan A : template functions that was instantiated until "now" into compilation unit
- plan B : forget about them and reflect only functions that are not templates

(c) For declaration of member depending of a template parameter, just pick it's concrete type:

Listing 10: Member template function

```
template<typename V>
struct T {
        V value;
};
```

For above struct example, if we ask for `typedef< T<int> >...` than the member value will be a `int T::*` concrete type;

14. `typedef<T>...` where T is a "namespace" have kind of same problem as "Template functions" because they are not fully expanded at time you call to `typedef<T>...`, they can be extended anywhere after.

(a) But... that will be nice if you could enumerate all types of a namespace !

- plan A : types that was instantiated until "now" into compilation unit (same problem as Plan A - of templates)
- plan B : forget about them
- plan C : Implement Plan A you can be self-organized and have two projects: compile a lib X with your full expanded namespace and a "stub" outer executable project that you include your lib X this way you guarantee all namespace expansions are done;

(b) It also have new problems, like inner namespaces. For solving this, `typedef<T>...` could recurrently dig into inner namespaces to bring all inner types classes and functions, since class names are fully qualified wher you call `typename<T>...`. For global namespace, we could define a `typedef<::>...` syntax, but how many entries this would have ?

(c) Modules may help a lot to define the exact the scope of what `typedef<module>...` should reflect, but the discussion of what is a module, has just started, so we will wait until it's better defined;

4.3.2 Returns of `typedef<T>...`:

1. It should respect access rights to members of the function that calls `typename<T>...` or `typedef<T>...`, have access to private/protected via derivation or `friend`;

2. For each class, the types of `typedef<T>...` should contains the pointer types of its direct base classes, than the pointer to class itself, than its member objects pointers, member functions pointers, operator functions pointer, typedefs, contructors and destructors enumerations, const member objects and friend functions, in order of definition inside the class; see code sample 6 "More complete class use case"

3. For each class, the values of `typedef<T>...` should be nullptr for base classes, for the class itself, and typedefs and enum definition, and a pointer to member for each declared member, and the value of member if it's a constexpr or an enumeration element (which are constexpr by the way, just saying);

4. Member typedefs are treated as they are inner classes, if T has a member `typedef X Y`, than `typedef<T>...` returns the new name of the type "T::Y" and a "typed" nullptr in it's position, the type that was introduced into scope of T. But if you call `typedef<T::Y>...` it will be the same as calling `typedef<X>...`;

5. One could say "you may skip a pointer to class itself in return of `typedef<T>...`, since it's useless, you already have that !" right, but we need to bring the name of class in `typename<T>...` and to maintain the same number of elements in `typedef<T>...` and `typename<T>...` we fulfill that slot with a FILLER member (for those old enough to know COBOL).Also, this way, putting "T" base classes before "T" and inner classes after "T", we can differentiate if a class T has a parent P and also a field P inside itself;

6. About the member functions, the main target here is, as far as possible, to be able to call any function of a class that is returned by `typedef<T>...`;

    - For common functions, operators cast operators static and friend functions, just pick their pointers plain and simple.
    - Constructors are problematic, since C++ standard says that we cannot get their addresses. Cleverly (bad cleverly), they can be reflected by lambda proxies; see code sample 6 "More complete class use case". We know that this idea may impose a big problem, but, after you realize, that you must somehow be able to use lambdas as parameter of `typedef<T>...`, this proxy hack will become a little bit smaller problem. Even if this will be completely ripped out form the proposal, we stand the effort to be able to make string-mapped class factories with all possible constructor calls also mapped via reflection. This also brings another set of problems: how implement this lambda constructor hack, should we return a heap-allocated or a stack allocated ? Should we use move-semantics ? Should we use placement new ?. In our sample we returned a simple local object and let RVO works; Or may be we simply may be able to get constructor addresses...
    - Destructor are problematic too, since C++ standard also says that we cannot get their addresses. But you can call a destructor without reflecting it. So we may not need them in reflected elements; We think that may be, some obscure case of virtual classes without virtual destructors may benefit of this, but this is already a stretch
    - Inline functions must have a "not inlined" version so you could call them via reflection, the compilers already take care of this today if you pick a pointer do member inline function;
    - Constexpr must be treated by it's context, if you ask for `typename<strlen>...` that it will be reflect as a function, but if you try ask for `typename<strlen("test")>...` the result will be the same as if you call for `typename<(size_t)4>...`;

4.3.3 Returns of `typename<T>...`:

1. For classes it returns the qualified names of classes with all namespaces, inner classes and typedefs fully qualified, but for members, only it's names;

2. For a constant of some type X, well, we think that `typename<T>...` should return the name of the type of X;

3. For enums it returns the names of members qualified into global namespace, more precisely, they are prefixed with "::"

4. For class enums it returns the names of members not qualified, more precisely, they are not prefixed with "::" nor class name, nor namespaces;

5. For non member functions it returns the name of the function prefixed with it's namespace or "::" if it's a global function, without parentheses and without any noexcept/exception declaration or inline or other modifiers, only the qualified name of the function, then the names of each parameter;

6. For member functions it returns the qualified name of the function, without parentheses and without any noexcept/exception declaration or inline or other modifiers like virtual or public or protected constness or volatile, etc..., only the qualified name of the function, then the names of each parameter;

7. For non member operators, it returns in same syntax as they are called by `this->operator @()` idiom for example, for `this->operator +(x)` the result should be "operator +" without one space;

8. Member typedefs are treated as they are inner classes, if T has a member `typedef X Y`, than `typename<T>...` returns the new name of the type "T::Y", the name that was introduced into scope of T; But if you call `typename<T::Y>...` it will be the same as calling `typename<X>...`

9. Result of `typename<T>...` must apply name look-up rules to find what is "T", but once it finds, the result must not be changed by the scope the call is placed, even if the call is placed inside a member function, it means, the results that represent names of classes, including base classes, enums, unions, and typedefs, and inner classes should be fully qualified, member data and functions and should not be qualified, and global objects should be prefixed with "::", even if the call is in same namespace than the class of in global namespace;

5 Out of proposal important topics

1. Attributes: It's a long way from what we got today as attributes and something usable in reflection. Even if we define another instruction like `attribute<T>...` that expands attributes of T into a variadic template, like `typedef<T>...`.That would be a good idea if attributes was:

   - typed;
   - scoped into some namespace (like any type);
   - their parameters should be typed as well;
   - their parameters values should be constexpr;
   - they must be freely user-declared and user-defined;

   Well, if attribute is a common type, that we become able to "attach" via [[attribute]] to target, initialize it's member values via standard initialization and constexpr, so get them would be possible:

   - Call `typename<T>...` where T is an attribute to pick names of attribute and it's members of defined attribute;
   - Call `typedef<T>...` where T is an attribute to pick types of attribute and it's members and const values of members of defined attribute;

   Among many problems, attributes are not typed nor part of type system (characteristic they share with macros and throws specification) According to [ISO13, N3690]: 7.6.1.3. [dcl.attr.grammar] If a keyword (2.12) or an alternative token (2.6) that satisfies the syntactic requirements of an identifier (2.11) is contained in an attribute-token, it is considered an identifier. No name lookup (3.4) is performed on any of the identifiers contained in an attribute-token. The attribute-token determines additional requirements on the attribute-argument-clause (if any). The use of an attribute-scoped-token is conditionally-supported, with implementation-defined behavior. [Note: Each implementation should choose a distinctive name for the attribute-namespace in an attribute-scoped-token. - end note]. They are a kind of "naked identifier", a little bit far of "typed things" and again, we prefer to maintain the discussion of reflection separated of this issues, for now;

2. In serialization, how to choose what serialize:

   - Plan A: Left to user register wherever he wants into "serialization engine" and library implementers use some idiom like create a "inner class with name serial_info" or just "pick member functions that starts with get or set" or a `typedef start_serialize start_t` the members and then `typedef end_serialize end_t;` inside class as start and end serialization members. see "Weired idiom"6.2
   - Plan B: Create a attribute, like `[[serializable]]` to define what and how should be serialized, and a corresponding type-trait like `has_attribute<>` to discover if that member is serializable.The problem with this approach is : how you get the parameters of the attribute ?

3. Would be nice if member modifiers like inline, explicit, virtual, friend, override, final, constexpr, default, noexcept, private, protected, public, thread_local, throws specification be extracted, we think that type traits would be the best way to get these since they are not part of the type;

4. Concepts, they are an intention message from developer to compiler "please, ensure this for me", since the compiler use them only before compile a template as a restriction "can I compile this algorithm with this parameter ?" and our reflective mechanism only picks concrete type information, after compilation, so we think they are not a part of type. The best use we could give to them are to choose what to return in `typedef<T requires U>...` as said in "Initial considerations" 4.1.2;

5. Few compiler issues:

   - About instruction `typedef<T>...`: if we understand it as a meta-function call that have a variadic template return type but not a variadic call, contrary as when you instantiate some variadic template class, when you put all types in the call, the tokens are not there and they must be generated by compiler.

6 More complete class use case

6.1 Full case

Listing 11: Full sample

```
#include <string>
#include <tuple>
#include <vector>
#include <type_traits>
#include <assert.h>
#include <iostream>

#define CHECK(expected_, index_, tuple_)  \
      static_assert(std::is_same<expected_, \
      std::tuple_element<index_, decltype(tuple_)> \
      ::type>::value, "Problem ?")

namespace reflect {
using namespace std;

template <typename T> struct Pod1 {            // 0
 typedef T value_t;
 string nm;
 T      value;
};

struct Pod2 : public Pod1<int> {               // 1
 Pod1<int>::value_t another_value;             // 2
 static int SOME_STATIC;                       // 3
 static const int SOME_CONSTANT;               // 4
 static constexpr int SOME_OTHER_CONS = 55;    // 5

 struct inner_class {              // inner class 6
  int value;
 };
 inner_class inner_member;                     // 7

 enum enum_type : unsigned { // inner enum type  8
  V1 = 1,
  V2 = 2000
 } enum_member;                                // 9

 virtual int callable(int x) const {return x + 1;}// 10 common function
 void set_value(int) {}                        // 11 getter/setter idiom
 operator int() const {return 0;}              // 12 cast operator
 Pod2& operator=(const Pod2&) { return *this; }  // 13 more operator
```

```
 Pod2& operator+(const Pod2&) { return *this; }   // 14 more operator

 explicit Pod2():
       another_value(0), enum_member(V1) {}       // 15 constructor
 Pod2(int x, int y):
       another_value(x), enum_member(V2) {}       // 16 other constructor
 virtual ~Pod2() {}
};

Pod1<int>::value_t Pod2::SOME_STATIC = 20;        // 5
extern const int Pod2::SOME_CONSTANT = 12;        // 6
constexpr int Pod2::SOME_OTHER_CONS;              // 7

void reflect_check1() {

 // even if constructors cannot have their address taken, we can make
 // a 'lambda hack' to a callable function ctor/dtor

 auto lambda_ctor1 = []()->Pod2 { return Pod2(); };
 auto lambda_ctor2 = [](int x, int y)->Pod2 {return Pod2(x,y);};

 auto mytuple = std::make_tuple(typedef<Pod2>...);
// auto mytuple = std::make_tuple(
// /* 0*/ (Pod1<int>*)nullptr,
// /* 1*/ (Pod2*)nullptr,
// /* 2*/ &Pod2::another_value,
// /* 3*/ &Pod2::SOME_STATIC,
// /* 4*/ &Pod2::SOME_CONSTANT,
// /* 5*/ Pod2::SOME_OTHER_CONS,
// /* 6*/ (Pod2::inner_class*)nullptr,
// /* 7*/ &Pod2::inner_member,
// /* 8*/ (Pod2::enum_type*)nullptr,
// /* 9*/ &Pod2::enum_member,
// /*10*/ &Pod2::callable,
// /*11*/ &Pod2::set_value,
// /*12*/ &Pod2::operator int,
// /*13*/ &Pod2::operator=,
// /*14*/ &Pod2::operator+,
// /*15*/ lambda_ctor1,
// /*16*/ lambda_ctor2
// );

 std::vector<std::string> names = { typename<Pod2>... };
// std::vector<std::string> names = {
// /* 0*/ "reflect::Pod1<int>",
// /* 1*/ "reflect::Pod2",
// /* 2*/ "another_value",
// /* 3*/ "SOME_STATIC",
// /* 4*/ "SOME_CONSTANT",
// /* 5*/ "SOME_OTHER_CONS",
// /* 6*/ "reflect::Pod2::inner_class",
// /* 7*/ "inner_member",
// /* 8*/ "reflect::Pod2::enum_type",
// /* 9*/ "enum_member",
// /*10*/ "callable",
// /*11*/ "set_value",
// /*12*/ "operator int",
// /*13*/ "operator=",
// /*14*/ "operator+",
// /*15*/ "reflect::Pod2::Pod2",
```

```
// /*16*/ "reflect::Pod2::Pod2"
// };

 // ********************** Types ***************************
 CHECK(Pod1<int>*,                          0, mytuple);
 CHECK(Pod2*,                               1, mytuple);
 CHECK(int Pod2::*,                         2, mytuple);  // templated typedef
 CHECK(decltype(&Pod2::SOME_STATIC),        3, mytuple);
 CHECK(decltype(&Pod2::SOME_CONSTANT),      4, mytuple);
 CHECK(int,                                 5, mytuple);
 CHECK(Pod2::inner_class*,                  6, mytuple);
 CHECK(decltype(&Pod2::inner_member),       7, mytuple);
 CHECK(Pod2::enum_type*,                    8, mytuple);
 CHECK(decltype(&Pod2::enum_member),        9, mytuple);
 CHECK(decltype(&Pod2::callable),          10, mytuple);
 CHECK(decltype(&Pod2::set_value),         11, mytuple);
 CHECK(decltype(&Pod2::operator int),      12, mytuple);
 CHECK(decltype(&Pod2::operator=),         13, mytuple);
 CHECK(decltype(&Pod2::operator+),         14, mytuple);

 // ********************** Values **************************
 assert(nullptr                == std::get<0>(mytuple) ); // parent
 assert(nullptr                == std::get<1>(mytuple) ); // itself
 assert(&Pod2::another_value   == std::get<2>(mytuple) );
 assert(10                     == *std::get<3>(mytuple) ); // pointer
 assert(12                     == *std::get<4>(mytuple) ); // pointer
 assert(55                     == std::get<5>(mytuple) );  // constexpr
 assert(nullptr                == std::get<6>(mytuple) );
 assert(&Pod2::inner_member    == std::get<7>(mytuple) );
 assert(nullptr                == std::get<8>(mytuple) );
 assert(&Pod2::enum_member     == std::get<9>(mytuple) );
 assert(&Pod2::callable        == std::get<10>(mytuple) );
 assert(&Pod2::set_value       == std::get<11>(mytuple) );
 assert(&Pod2::operator int    == std::get<12>(mytuple) );
 assert(&Pod2::operator=       == std::get<13>(mytuple) );
 assert(&Pod2::operator+       == std::get<14>(mytuple) );

 Pod2 new_pod_A = std::get<15>(mytuple)();        // constructs new Pod2
 Pod2 new_pod_B = std::get<16>(mytuple)(10, 20); // constructs another new Pod2

 auto call = std::get<10>(mytuple);
 auto result = (new_pod_B.*call)(10);            // calls 'callable'
 assert(result == 11);
}
}
```

6.2 Weird idioms for serialization

Listing 12: Start/End serialization idiom

```
class component {
 friend class serializer;
 int     a; // this member will be ignored
 int     b; // this member will be ignored
 // serialization engine will start to pick members here;
 typedef start_serialize start_t;
 int     c;
 string  d;
 float   get_f() const { return f; }
```

```
 void     set_f(int val) { f = val; }
 void     set_f(float val) { f = val; }
 // serialization engine will stop to pick members here;
 typedef end_serialize end_t;
public:
 component() {}
 void bind() {}
 float    e; // this member will be ignored
 float    f; // this member will be ignored
}
```

Fields a, b, and e will not be serialized, but the fields c and d will, the intriguing is that field f will be serialized via function call `get_f()` and `set_f()` and the "serialize engine" should be able to set it via integer parameter;

References

[BCAS12] Dean Berris, Lawrence Crowl, Matt Austern, and Lally Singh. N3410 - rich pointers with dynamic and static introspection. Technical report, Programming Language C++/Reflection, 2012.

[ISO13] C++ ISO. N3690 programming languages c++. Technical report, Programming Language C++, 2013.

[Krz13] Andrzej Krzemienski. N3326 - sequential access to data members and base sub-objects. Technical report, Programming Language C++/Reflection, 2013.

[SC13] Jeff Snyder and Chandler Carruth. N3814 - call for compile-time reflection proposals. Technical report, Symantec, 2013.

[Spe09] Michael Spertus. N2965 - type traits and base classes. Technical report, Symantec, 2009.

[Spe12] Michael Spertus. N3403 - use cases for compile-time reflection. Technical report, SG7 - Reflection Study Group, 2012.

[TK13] Andrew Tomazos and Christian Kaser. N3815 - enumerator list property queries. Technical report, Programming Language C++/Reflection, 2013.