Doc. No: P0590R0
Date: 2017-02-05
Group: SG7
Contact: Andrew Sutton <asutton@uakron.edu>
Herb Sutter <hsutter@microsoft.com>

A design for static reflection

This document gives an overview of the design and implementation of static reflection for C++. The design presented here is closely related to the approach described in P0385R1, except that reflections are regular objects and not types. This enables a style of metaprogramming without the need for template metaprogramming. This paper presents a set of motivating examples, describes how the feature is implemented in Clang, and includes discussion of related work.

Table of Contents

Introduction and examples

Static reflection enables programmers to write code that queries the compile-time properties of types and declarations[1]. These queries can be used in conjunction with traditional programming techniques to support a number of common use cases. The examples presented in this section are adapted from those presented in P0385R1.

Logging

The need to generate readable declaration names for the purpose of logging, tracing, and debugging is a frequent concern. Typical approaches require the use demangling `typeid` names

[1] Reflecting expressions is reserved for future work.

through some compiler-specific ABI-provided library. Both of these operations incur runtime overhead.

With static reflection, we should be able to synthesize human readable declaration names without the associated runtime cost. Here is how we can do this:

```
template<typename T>
T min(T a, T b) {
  log() << “min” << ‘<’ << $T.qualified_name() << “>(“
        << $a.name() << ‘:’ << $a.type().name() << ‘,’
        << $b.name() << ‘:’ << $b.type().name() << “) = “;
  T r = a < b ? a : b;
  log() << r << ‘\n’;
  return r;
}
```

The $ operator is the *reflection operator*. It returns a class object of unspecified type, called a *reflection*, that describes the compile-time properties of the operand (here a type T). The type of a declaration’s reflection is unique (i.e., `$T` has a different type than `$a`). Note that the operator is applied only when the algorithm is instantiated; we do not reflect properties of dependent names.

The expression `$T.qualified_name()` queries the compiler for the qualified name of type T. It returns a C-string containing the human-readable name of the substituted type. Strings are only generated if the function is called; they are not added to the translation unit unless used.

Generation of common functions

The need to generate boilerplate code for functions on simple structures is a common problem. It appears in a very, very wide number of applications, and language-based solutions to the problem have proven to be difficult to navigate through committee. For example, here is a simple class and its associated `hash` function, based on Howard Hinnant’s proposal (N3980):

```
struct S {
  int i;
  long l;
  float f;
};

template<HashAlgorithm H>
void hash_append(H& h, S const& s) {
  hash_append(h, s.i);
  hash_append(h, s.l);
  hash_append(h, s.f);
}
```

For simple `struct`-like types, the definitions of these overloads follows a simple pattern: append each member of the `struct` to the hash in their order of declaration. We can use static reflection to do this for us. Here is a generic algorithm that works for all `SimpleStruct` types (defined in the next section).

```
template<HashAlgorithm H, SimpleStruct T>
void hash_append(H& h, T const& s) {
  for (MemberVariable v : $T.member_variables())
    hash_append(h, s.*(v.pointer()));
}
```

This algorithm[2] simply applies `hash_append` to each non-static data member of the class, which are accessed by the range expression `$T.member_variables()`. This yields a compile-time sequence of objects (i.e., a tuple) that describe the member variables of type T.

Here, `MemberVariable` is a constrained-type-specifier from the concepts TS. Recall that the type of reflections is an unspecified class type that is unique to a declaration. However, all such member variable reflections share the same properties. In this case, we can use the `pointer()` method to return a pointer-to-data-member, which can be applied to access the runtime-data member of the structure.

This technique can be used to provide specialized implementations of generic facilities that require the traversal of complex data types. Hashing is a straightforward example; equality and order comparison are closely related and left to the reader as an exercise.

Advanced constraints

The `SimpleStruct` concept in the previous example is indicative of the kinds of constraints that we can write using reflection. It is entirely possible that many existing type traits, whose definitions that rely on language wording, can be replaced by library features expressing the same ideas programmatically. `SimpleStruct` is such an example.

A simple structure is a POD class type with no base classes. We might define it like this:

```
template<typename T>
concept bool SimpleStruct() {
  return is_class_v<T> && is_pod_v<T> && $T.bases().empty();
}
```

The `$T.bases()` method returns a compile-time sequence of base classes. For a simple `struct`, this sequence must be empty[3].

Stringification

The generation of strings from enumerators is a very common feature request. Current options are to define a host of specialized functions that return the string representations of enumerated values, or macro-based definitions. Static reflection can be used to provide a simple three line function to generate the strings of for all enumeration types.

```
template<Enum E>
const char* to_string(E value) {
  for (Enumerator e : $E.enumerators())
```

[2] This example uses the tuple-based for loop described in P0589R0.

[3] This concept is probably subsumed by `StandardLayout`. However, generic algorithms requiring that concept would need to account for data members defined in base classes.

```
      if (e.value() == value)
        return e.name();
  }
```

Again, the simplicity of this implementation relies on the tuple-based `for` loop and concepts. We can write an equivalent version this function without those features, but it's quite a bit longer.

Implementation

This feature is being implemented in a fork of the Clang compiler, which can be found at https://github.com/asutton/clang-reflect. Note that this is a work in progress.

The reflection operator

The reflection operator $ can be applied to any name (variable, function, parameter, type, enumerator, namespace). It returns a prvalue class object whose type is determined by the entity whose name follows the operator. The type returned is a specialization of a template, definined in the `meta` namespace, whose template argument (given as X below) binds to the compile-time definition of the reflected entity. The value of this argument is implementation defined.

The mapping of entities to names is given in the following table:

Entity	*Reflection type*
Variable	`meta::variable<X>`
Member variable	`meta::member_variable<X>`
Function	`meta::function<X>`
Constructor	`meta::constructor<X>`
Destructor	`meta::destructor<X>`
Member function	`meta::member_function<X>`
Conversion operator	`meta::conversion<X>`
Function parameter	`meta::parameter<X>`
Enumerator	`meta::enumerator<X>`
Class type	`meta::class_type<X>`
Union type	`meta::union_type<X>`
Enum type	`meta::enum_type<X>`
Fundamental type	`meta::fundamental_type<X>`
Qualified type	`meta::qualified_type<X>`
Namespace	`meta::ns<X>`
Translation unit	`meta::tu<X>`
…	

Each class in the table above has the following form:

```
template<reflection_t X>
struct reflection-class {
  // member operations
};
```

The `reflection_t` type is the type is an implementation-defined handle to an internal structure in the compiler that describes the reflected entity. In our implementation, this is an integer type large enough to hold a pointer to a node in the compiler's AST[4]. When a name is reflected, the AST node pointer is converted into this opaque representation and used as a template argument for the corresponding class template in the table above. That is, this expression

```
$main
```

is an alias for this expression, which simply constructs temporary object.

```
meta::function</*AST-reference-to-main*/>{}
```

The opaque value for "AST-reference-to-main" is supplied by the implementation.

Reflected properties

The properties of reflected entity depend on its declaration, its definition, the language, and compiler options. In general, there are three kinds of information that can be requested of any reflection

- *Specifiers* are flags and values that indicate how a declaration was written.
- *Attributes* correspond to the written C++ attributes of a declaration.
- *Traits* are the computed from specifiers, attributes, and language rules.

The implementation does not currently support queries for specifiers or attributes. Only queries for traits are supported.

Reflection classes have no non-static data members. All properties are defined as static member functions and variables. The properties of a reflection class depend on the entity they reflect. These can be grouped into concepts, defined by the table below.

[4] This is actually a node pointer where unused low-order bits contain a small descriptor describing the kind of node (type, expression, or declaration). This is needed so we can re-cast the reflection to an object in the appropriate hierarchy.

Concept	*Members*
NamedEntity	const char* name() const char* qualified_name() ScopeEntity declaration_context() ScopeEntity lexical_context() linkage_t linkage() access_t access()
ScopeEntity	Tuple members()
Type	NamedEntity typename type;
UserDefinedType	Type, ScopedEnity
MemberType	bool is_complete() Tuple member_variables() Tuple member_functions() Tuple constructors() Destructor destrutors()
ClassType	MemberType bool is_polymorphic() bool is_abstract() bool is_final() bool is_empty()
UnionType	
EnumType	UserDefinedType bool is_complete() bool is_scoped()
TypedEntity	auto type()
Variable	NamedEntity, TypedEntity storage_t storage() bool is_inline() bool is_constexpr() T* pointer()
MemberVariable	NamedEntity, TypedEntity bool is_mutable() T C::* pointer()
Function	NamedEntity, TypedEntity bool is_constexpr() bool is_noexcept() bool is_defined() bool is_inline() bool is_deleted()

	Tuple parameters() T(*)(...) pointer()
Method	NamedEntity, TypedEntity bool is_noexcept() bool is_defined() bool is_inline() bool is_deleted() Tuple parameters() T (C::*)(...) pointer()
PolymorphicMethod	MemberFunction bool is_virtual() bool is_pure_virtual() bool is_final() bool is_override()
Constructor	MemberFunction bool is_constexpr() bool is_explicit() bool is_defaulted() bool is_trivial()
Destructor	PolymorphicMemberFunction bool is_defaulted() bool is_trivial()
MemberFunction	PolymorphicMethod bool is_constexpr()
ConversionFunction	MemberFunction bool is_explicit()
Parameter	NamedEntity, TypedEntity
Enumerator	NamedEntity, TypedEntity T value()

This tables associates a concept name with a set of required concepts, partial signatures for member functions and, in the case of the Type concept, a nested type name.

This is an initial concept/class design for the reflection hiearchy. We will almost certainly find better ways of constructing this hierarchy to accommodate things like arbitrary declaration specifiers, access specifiers, and other attributes.

Properties are accessed via member functions of the classes above. These member functions require the use of intrinsics to access their requested data. For example, the `function` template might provide access to its name like this:

```
template<reflection_t X>
struct function {
  static constexpr const char* name() const {
    return __reflect_name(X);
  }
};
```

The `__reflect_name` intrinsic takes the opaque entity descriptor to the compiler as a constant expression. Internally, the compiler converts this back to a node pointer, generates a human readable name for the reflected variable, and the rewrites the entire expression as a string literal containing the characters of the variable's names. The effect is that the expression $main.name() results in the following definition for main:

```
static constexpr const char* name() const {
  return “main”;
}
```

Here is a list of intrinsics that can be used to access scalar or string properties.

The entirety of this design is driven by an implementation specific interface exposed by the compiler that allows library writer to query internal properties of the AST. This interface is can be viewed as an extension of a compiler's type trait intrinsics, which give library writers a limited view into the compiler's understanding of the type system. This interface makes that view much larger and more fine-grained. A partial listing of that interface is given in the table below.

Intrinsic name	Return type	Description
`__reflect_name(n)`	`const char*`	Returns the declared name of a declaration.
`__reflect_qual_name(n)`	`const char *`	Returns the fully qualified name of a declaration.
`__reflect_type(n)`	`Type`	Depends on the named entity
`__reflect_scope(n)`	`ScopedEntity`	Depends on the named entity
`__reflect_member(n)`	`Tuple`	A tuple containing the objects/types of members of a scope
`__reflect_traits(n)`	`unspecified`	A bitfield object that encodes a series of flags and values to be interpreted by property accessor functions.

Member sequences

Properties that return `Tuples` (e.g., `members()`) are heterogeneous containers of declarations. This is not a `std:tuple` (the number of members could be very large). The current implementation uses a *generated tuple*, built on the back of two intrinsics: `__reflect_num_members`, and `__reflect_ith_member`. The first tells the program how many elements the tuple contains, and the second yields the ith element. Note that this design does not actually produce an object containing all elements.

These are combined to define a data type that works like a tuple. Here is a simplified version of that class; it is parameterized by a data type R (for reflections) that provides two operations: size and get.

```
template<typename R>
struct generated_tuple { };

template<typename R>
struct std::tuple_size<generated_tuple<R>>
  : std::integral_constant<size_t, R::size()>
{ };

template<int I, typename R>
auto get(generated_tuple<R>) {
  return R::template get<I>();
}
```

To use the generated tuple, an element must define the reflection data type to access the required elements. Here is a fragment of the `ns` template.

```
template<reflection X>
struct ns {
  // Provides operations for a generated tuple
  struct member_reflection {
    static size_t size() { return __reflect_num_members(X); }
    template<int I>
    static auto get() { return __reflect_ith_member(X, I); }
  };
  using member_list = generated_tuple<member_reflection>;

  // The member accessor
  static member_list members() { return {}; }
};
```

Note that we need to define these two intrinsics for each kind of sequence returned. There are currently four: members of scope, bases of a class, parameters of a function, and enumerators of an enumeration.

Of course, there are also interesting subsets of those sequences. For example, when querying a class, we might only be interested in non-static member variables or constructors. Or we might be interested in the virtual bases of a class. One option is to define an intrinsic for each sequence of interest, but that wouldn't scale. The other option is to apply a compile-time filter to those sequences.

Filtered sequences

Instead of defining an even more expansive intrinsic interface, we can apply a little metaprogramming and build compile-time filters on our generated tuples. The "filter" is simply a type trait that determines the inclusion of an element in the result set. The filtered tuple interface is:

```
template<typename R, template<typename> typename P>
struct filtered_tuple { };

template<typename R, template<typename> typename P>
struct std::tuple_size<filtered_tuple<R, P>
  : std::integral_constant<size_t, tuple_count_if_v<R, P>> { };
template<int I, typename R, template<typename> typename P>
auto get(filtered_tuple<R, P> t) {
  return tuple_get_if<I, P>(t);
}
```

The R parameter provides access to the underlying elements of a generated tuple, and the predicate is passed as a template template parameter. The `tuple_size` specialization counts the number of elements X for which P<X>`::value` is true, and `tuple_get_if` returns the ith element for which P<X>`::value` is true.

The implementation of this facility would be *much* easier if concepts were available. Fortunately, it's use is not particularly difficult. Here is how to implement `member_variables()` for a class.

```
template<reflection_t X>
struct class_type {
  // define member_reflection as above
  using memvar_list
    = filtered_tuple<member_reflection, is_member_variable>;

  static memver_list member_variables() { return {}; }
};
```

Discussion

This section discusses the evolution of this design, including comparisons with prior work and tradeoffs of different approaches.

Related work

The most comprehensive work done on static reflection to date is by Matúš Chochlík, Axel Naumann, and David Sankel. The most documents capturing the state of this work are:

- P0385R1 – "Design document for static reflection" by Chochlík, Naumann, and Sankel.
- P0194R2 – "Proposed wording for static reflection" by Chochlík, Naumann, and Sankel.

- P0255R0 – "C++ Static Reflection via template pack expansion" by Cleiton Santoia Silva and Daniel Auresco

There is also an initial implementation of the work by Chochlík and Axel Naumann in Clang, which can be found here:

https://github.com/matus-chochlik/clang/tree/mirror-reflection

This approach Chochlík, Naumann, and Sankel uses a `reflexpr` operator to associate a unique implementation-defined class with each reflected type (fundamental, compound, user-defined), namespace, and specifier (public, virtual, etc.). A set of queries, in the form of type traits, is used to access the name, members, and other properties of the reflected class. Certain queries (i.e., `get_pointer<>`) can be used to access the reflected objects or class members. Reflection of templates, expressions, and names with no linkage is not currently supported.

The proposal described in this document is largely equivalent to the design in P0385R1 and P0194R2 except in one significant way: the `reflexpr` operator yields types and our `$` operator yields objects. Many other aspects of both proposals are directly analogous, including the compiler/library interaction for creating new reflection types or objects.

In fact, the proposals are so closely related that, if the library aspects were unified, then we have the following equivalencies for the reflection operator:

```
reflexpr(x){} /* is equivalent to */ $x
decltype($x) /* is equivalent to */ reflexpr(x)
```

A second major difference in the design is how properties are accessed. P0385R1 relies on type traits; we use member functions. The requirement for type traits follows from the fact that `reflexpr` yields a type, although properties could also be accessed via nested members. In this proposal, methods and members of reflection classes are all static. Although not directly usable with type traits (because $ yields an object), that information is nonetheless accessible at the type level.

One notable difference is that our implementation relies on intrinsics to generate property values when requested. The implementation described in P0385R1 appears to populate reflection types at the time they are defined. There are pros and cons to both approaches; we suspect that the implementation techniques are interchangeable.

It is also worth noting P0255R0 by Cleiton Santoia Silva and Daniel Auresco. This proposal is less fully developed than that of Matúš and Naumannn, but has some interesting ideas.

This approach uses a set of operators, `typedef<T, P>`, `typename<T, P>`, and `typeid<T, P>`, to denote a parameter pack of "things" associated with T and filtered by some predicate P. I say "things" because I believe the queries can yield either type or non-type argument packs. For example:

```
typedef<string, is_member_object_pointer>...
```

Selects all members from string and expands that as a pack of member pointer objects. The different operators select different kinds of properties.

Under this proposal, it appears that the type trait used to select elements of the resulting pack would actually determine the type of those elements. Here, for example, the elements are member pointers. It's not entirely clear how I might select the names of those members, for example. I'm not sure the proposal scales.

However, the idea of returning an unexpanded pack of objects is not fundamentally different than how this proposal returns and generates tuples. However, the uniform language support for tuples, packs, and classes is well beyond the scope of this proposal, but nonetheless intriguing.

Missing features

This proposal is lacking a *reification* operator; that is an operator that takes a reflection and yields an object, function, type, or namespace. Initial circulations of the proposal included a postfix operator that did just that. For example:

```
int x = 0;
$x.type()$ y = y; // the type of y is int.
```

Here, the postfix reification operator is applied to the reflection `$x.type()` (this yields a `meta::` object) to produce the type of `y`. However, there were some concerns about parsing this style of operator, and so we have temporarily put its design aside.

Specification

2. Lexical conventions

In Section 2.3, add $ to the list of characters in the basic character set.

In Section 2.12 add $ to the *preprocessing-op-or-punc* production.

5. Expressions

In 5.3, modify the *unary-expression* grammar as follows, and add a new production *reflection-argument.*

unary-expression:
  postfix-expression
  ++ cast-expression
  -- cast-expression
  $ reflection-argument
  unary-operator cast-expression
  …

reflection-argument:
  id-expression
  type-id
  namespace-name

Add a new section 5.3.8, Reflection with the following text

The $ operator returns an prvalue class object of unspecified type called a *reflection*. The type of the reflection depends on the entity denoted by the operand and satisfies one of the concepts in [lib.meta.reflect] as specified table [ref].

Table 1 - Reflections and the concepts they satisfy.

Reflected entity	Satisfied concepts
Function	Function
TODO: Finish this table.	

20.15. Reflection concepts [meta.reflect]

Add this section to clause 20. It contains the definitions of the concepts describing reflections.

TODO: Define these concepts.