N3403=12-0093
2012-09-22
Mike Spertus, Symantec
mike_spertus@symantec.com

Use Cases for Compile-Time Reflection

Overview

There have been many proposals around compile-time reflection, some accepted, like type traits, some proposed like n2965 and n3326, and some still to come, perhaps like static reflection (which develops a number of the ideas in this paper much further than here). In order to ensure a coherent and powerful framework for compile-time reflection in C++1y, we need to have an understanding of questions like what are its goals and what use cases does it need to support.

The purpose of this paper is to start to collect use cases for compile-time reflection that we can use to help evaluate proposals and seek out additional proposals. I am well aware that the list is incomplete and does not represent any agreed on understanding. I am want this paper to serve as a tool to solicit use cases, requirements, and discussion about compile-time reflection, to help brainstorm how to solve the most difficult problems and get the most benefits.

For each use case, I include a list of requirements. Not that these requirements are not requirements on the use case but are instead requirements that the use case is likely to impose on compile-time reflection. I.e., I do not give a list of requirements that a serialization framework needs to satisfy but instead list requirements that a serialization framework may be reasonably expected to impose on the reflection framework.

Use cases

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

Serialization

Reflection can contribute to C++ serialization by making use of the information about data structures that is already implicit in the program. For example, it is easy to imagine that something simple like the following could just work out of the box.

```
struct S {
  int i;
  double d;
};
/* ... */
archive a("data.arch");
S s;
a["s"] >> s;
```

```
s.i = 10;
a["s"] << s;
```

Requirements

While there are many questions about the design of a serialization framework (e.g., should serializable classes have to inherit from `serializable`?) that can only be decided by the designer of the facility, there are some tools that the reflection facility can reasonably be expected to provide to make their job easier.

- The reflection system needs to enumerate all of the fields and base classes of a class, even the non-public ones, so the serialization framework can infer (at least a good approximation of) what fields need to be serialized.
  - Questions of visibility need to be thought through because the serialization system will require member pointers to non-public fields. For example, a `constexpr` function `fields<C>` that returns a `constexpr tuple` of member pointers for all the fields of `C` would be very useful for serialization but allows client code to evade access controls.

    ```
    struct S {
      int i;
    private:
      double d;
    };
    /* ... */
    // fields<S> is a tuple containing &S::i and &S::d
    S s;
    s.*get<1>(fields<S>());  // Accessing private member without S' permiss
    ```

    One approach, inspired by that in n3326, would be for `fields` to return a class containing two tuples, a private tuple and a public tuple for private and public fields respectively. The members template would list the serialization framework as a friend. However, that would restrict us to one blessed serialization framework, which clearly isn't satisfactory. One idea is to require the reflection framework to be able to enumerate friend declarations, then `fields<S>` could add all the friends of `S` to its own friend list. Now, `S` could just make the serialization framework a friend.

- If `D` is derived directly from `B`, we need to be able to find the `B` "part" of `D`. Non-public inheritance presents a challenge here because if `D` does not inherit publicly from `B`, an external routine cannot cast from `D` to `B` to find the `B` part of `D`.

  One approach might be to allow member pointers to point to base classes, so we could have something like `B (D::*p) = &D::B;` This would also be a generally helpful feature for working with multiple inheritance.

Parallel hierarchies

Description

This is proposed in . The basic idea from that paper is that if you have a hierarchy of classes `Cn` with various inheritance relationships among the `Cn` (E.g., `C1` inherits directly non-virtually from `C2` and `C3`), then we may want to construct a parallel hierarchy of classes with the same inheritance relationships.

An example from the paper are if the `Cn` are a hierarchy of interface classes, then `Impl<Cn>` is a parallel hierarchy of implementation classes such that `Impl<Cx>` should inherit from `Cx` and from `Impl<Cy>` for all `Cy` that `Cx` inherits from. Having such a parallel implementation hierarchy is an extremely popular programming pattern. See the paper for more details and other uses of parallel hierarchies.

Requirement

As in the paper, the only requirement is that we can enumerate all the direct bases of a class. The paper includes a type trait `direct_bases` for that purpose. Even if we have a full-blown reflection framework, the `direct_bases` type trait may be a useful convenience class due to its simplicity and good instantiation performance.

Delegation

Description

Along with the rise of "prefer delegation to inheritance" mantra, creating wrapper classes that forward all methods to the wrapped class has become extremely popular. In this reflection use case, we would like to be able to, given a class `T`, construct a new class `delegate<T>` containing all the public methods of `T` and forward them to a contained `T *`.

```
#include <delegate>

class C {
public:
  virtual void f();
  int i;
  virtual vector<int> g(double);
private:
  void h(int);
};

// The generic delegate<C> should be identical
to the manual specialization below
template<>
class delegate<C> : public C {
public:
  delegate(C *c) : wrapped(c) {}
  virtual void f() { return wrapped->f(); }
  virtual vector<int> g(double d) {
    return wrapped->g(d);
  }
};
```

Of course, this is most useful with interface classes, but then parallel hierarchies will help with that...

Requirements

- It should be possible to enumerate all of the virtual public methods of a class (including inherited ones). Since producing a full list of members can be expensive in unnecessary instantiation costs and may run into template instantiation limits. For example, if `members<C>::type` is the same as `tuple<`*`pointers to all members of C`*`>`, then a large

class might have more members than `tuple` can take as parameters (Only 64 template parameters are available in Visual Studio). This suggests that the member enumeration facility should be flexible enough enumerate only virtual public methods without enumerating all members.

- Metaprograms need to be able to create methods whose name is given by some kind of "compile-time string." Since C++ does not have such a facility, this will require a language extension.

Getters and Setters

It is an oft-quoted best practice to provide getters and setters to modify all fields. While this is sometimes taken too far, it is often good advice. For example, if a data structure is being edited in a GUI using a model-view-controller operation, the setter functions can update the view. Unfortunately, the dictum to provide getters and setters is typically honored in the breach due to the painful boilerplate required. This use case is to use templates to automate such routine boilerplate

Unfortunately, using ordinary templates as getters and setters is awkward, mainly because of the need to fully qualify the field name.

```
template<class T, class FieldType>
void set(T *obj, FieldType T::*m, const FieldType &value)
{
  obj->*m = value;
  Code to fire change notifications
}

struct A {
  int i;
  double d;
};

A a;
set(&a, &A::i, 4);  // Much worse than a.i = 4;
```

This use case envisions a template class `accessor<T>` such that the following specialization is redundant.

```
template<>
struct accessor<A> : public A {
  int get_i() { return i; }
  void set_i(int new_i) { i = new_i; }
  double get_d() { return d; }
  void set_d(double new_d) { d = new_d; }
```

Even more ambitiously, the `accessor` template could create proxy field that overload `operator=()`.

Requirements

This also has the requirement of being able to generate methods whose name is computed at compile-time.

An ambitious solution that doesn't use reflection is to stick with the generic `get` template above, but create some way for more simply referring to fields. E.g., if you could somehow say `get(a, i)` instead of `get(a, &A::i);`, but that would require a major change in symbol lookup.

Generate user interfaces for classes, functions, and constructors

Description

(Note: See static reflection for a similar (and different) approach in this vein). Consider a class

```
class Person {
public:
        string name;
        int age;
        int weight;
};
```

It would be very nice to be able to use code like

```
Person p = ...;
getCanvas().edit(p);
```

to get an editable form like

Of course, it would be better if we could customize the form. For example,

```
template<typename T> string prompt() {
  return T::name;
}
template<>
string prompt<describe<&Person::weight>>
{
        return "Weight in pounds? Be honest";
}
```

to get the form

Applying the same techniques to functions creates new challenges but the payoffs are also high

```
string getEmployeeTitle(int employeeID);
```

Again,it would be very nice to be able to use code like

```
string title = getCanvas().invoke(getEmployeeTitle);
```

to automatically produce a form like:

Of course, it would be better if we could customize the form (along the same lines as the class example) with something like

```
template<typename T> string prompt() {
  return T::name;
}
template<>
string prompt<describe<&getEmployeeTitle::employeeID>>() // Not legal C++
{
```

```
        return "What is the employee's ID?";
}
```

to get

Requirement

- The reflection framework should be able to enumerate the names of function parameters from the declaration (heavy reliance on ODR). Likewise, there should be some way of referring to function parameters, as desired in the mythical use of `&getEmployeeTitle::employeeID` above.
- It might be useful for reflection to enumerate comments, especially doxygen comments. Those could be displayed when mousing over the fields. There are many other benefits to this (e.g., producing online help in a single pass). The ability to understand comments has served Java well.

Performance requirements

This doesn't really fit into any of the above use cases, or rather it fits into all of them. Enumerating all of the members of something in a kind of template sequence can be very expensive. Consider something like `members<namespace std>`. Furthermore, it may exceed template argument limits. Therefore, we probably need

1. membership enumerators that only enumerate the requested types of members (e.g., enumerate public methods) rather than filtering the entire list of members
2. We need some mechanism to get arbitrarily long compile-time sequences. One approach would be to return chunks. See n3416 for another approach.