Controlling the instantiation of vtables and RTTI

Document #: D1263R1
Date: 2018-11-06
Project: Programming Language C++
Audience: EWG

Reply-to: Louis Dionne <ldionne@apple.com>

1 Abstract

It is currently not possible to control where the RTTI and the vtable of a type are instantiated. It is also impossible to apply attributes to the vtable and the RTTI of a type. This leads to difficulties when trying to control the ABI of a shared library: one must apply attributes to and export the whole class from the shared library, which is not an option some members of the class should not be exported.

This proposal suggests providing explicit control over where the vtable and the RTTI of a type are instantiated. By introducing syntax that allows referring to those objects, it also becomes possible to apply attributes to them:

```
template <typename T>
struct Foo { virtual ~Foo() { } };

struct Bar { virtual ~Bar() { } };

// Explicit instantiation declaration of RTTI/vtable
extern template Foo<int>::typeid;
extern template Foo<int>::virtual;
extern Bar::typeid;
extern Bar::virtual;

// Explicit instantiation of RTTI/vtable
template Foo<int>::typeid;
template Foo<int>::virtual;
Bar::typeid;
Bar::virtual;
```

2 Motivation

As outlined in the abstract, this proposal is relevant for libraries and applications wishing to control the symbols exported in their ABI. In this section, we outline a couple of motivating examples related to ABI control.

1. **Controlling whether the vtable/RTTI is explicitly instantiated**
   Consider the following situation where we want to instantiate some member functions and the vtable of (a specialization of) a class template in a shared library:

```
// in a header
template <class CharT, class Traits = std::char_traits<CharT>>
class basic_ios {
public:
  bool operator!() const { /* ... */ }
  basic_ios& copyfmt(basic_ios const&) { /* ... */ }
  virtual ~basic_ios() { }
  // ...
};

extern template class basic_ios<char>;

// in the shared library
template class basic_ios<char>;
```

   Here, we end up explicitly instantiating the RTTI, the vtable, and all the member functions of `basic_ios<char>` in the shared library. However, if we don't want `basic_ios<char>::operator!` to be instantiated in the shared library, we might be tempted to write the following (note that we don't externally instantiate `basic_ios<char>::operator!`):

```
// in a header
template <class CharT, class Traits = std::char_traits<CharT>>
class basic_ios {
public:
  bool operator!() const { /* ... */ }
  basic_ios& copyfmt(basic_ios const&) { /* ... */ }
  virtual ~basic_ios() { }
  // ...
};

extern template basic_ios<char>& basic_ios<char>::copyfmt(basic_ios<char> const&);
extern template basic_ios<char>::~basic_ios();

// in the shared library
template basic_ios<char>& basic_ios<char>::copyfmt(basic_ios<char> const&);
template basic_ios<char>::~basic_ios();
```

   Unfortunately, this solution does not work because the RTTI and the vtable are not instantiated in the shared library anymore. We are not aware of any way to solve this problem in the language.

2. **Applying attributes to the vtable/RTTI**

Another related problem is the application of attributes to the vtable and the RTTI of a class. Since there is no way to syntactically refer to those entities, it is impossible to apply attributes to them. Unfortunately, controlling the ABI of a shared library requires applying visibility attributes to those entities – right now one can only apply those attributes to the whole class. For example, we'd like to write the following:

```
// in a header
#define HIDDEN __attribute__((__visibility__("hidden")))
#define VISIBLE __attribute__((__visibility__("default")))

namespace std HIDDEN { // hide everything from the ABI by default
  template <class CharT, class Traits = std::char_traits<CharT>>
  class basic_ios {
  public:
    bool operator!() const { /* ... */ }
    basic_ios& copyfmt(basic_ios const&) { /* ... */ }
    virtual ~basic_ios() { }
    // ...
  };

  // cherry-pick symbols to export from the shared library
  extern template VISIBLE basic_ios<char>::virtual;
  extern template VISIBLE basic_ios<char>::typeid;
  extern template VISIBLE basic_ios<char>&
                          basic_ios<char>::copyfmt(basic_ios<char> const&);
  extern template VISIBLE basic_ios<char>::~basic_ios();
}

// in the shared library
template std::basic_ios<char>::virtual;
template std::basic_ios<char>::typeid;
template std::basic_ios<char>&
         std::basic_ios<char>::copyfmt(std::basic_ios<char> const&);
template std::basic_ios<char>::~basic_ios();
```

3. **Controlling which translation unit the vtable/RTTI are defined in**
   Another benefit of this proposal would be to eliminate the need for a common technique called "anchor functions". This technique is used to control the translation unit in which the vtable and the RTTI of a type are instantiated. The technique consists in declaring (but not defining) a non-inline virtual function in the class, and defining that function in exactly one translation unit:

```
// in header
struct Foo {
  virtual void a() { }
  virtual void b() { }
  virtual void anchor();
};

// in exactly one TU
void Foo::anchor() { }
```

This results in the vtable and RTTI being instantiated in the translation unit where the anchor function is defined. This technique relies on the fact that the vtable and RTTI are always defined in the translation unit where the first virtual function of the vtable is defined. Since the class contains a non-inline virtual function, it knows that this virtual function must be defined in exactly one translation unit, and so it can emit the vtable and RTTI in that translation unit only instead of instantiating those in each translation unit and given `linkonce_odr` linkage.

This technique is unfortunate as it relies on detailed knowledge of how common toolchains work, and it may also require creating a dummy virtual function.

3 FAQ

1. How does this interact with regular (non-virtual) inheritance?
   Derived classes can decide to explicitly instantiate (or not) their vague linkage objects however they want: they are unaffected by what their bases do in that respect.

2. How does this interact with virtual inheritance?
   In the event of virtual inheritance, implementations use another vtable called the VTT (Virtual Table Table). We suggest that instantiating the vtable would also control where the VTT is instantiated for that type, if that type happens to need a VTT.

3. Could I make the vtable/RTTI constexpr, inline, const, etc?
   I don't think that makes sense, because that would imply that we can name the type of the vtable/RTTI. Consider:

   ```
   template constexpr TYPE? Foo<int>::virtual;
   template const TYPE? Foo<int>::virtual;
   template inline TYPE? Foo<int>::virtual;
   ```

   Also, vtable/RTTI today is morally `constexpr` (initialized at compile-time), `inline` (ODR-merged across TUs except when there is an anchor function), and `const` (can't be modified). It's not clear to me that it makes sense for it to be anything else.

4. Does it really make sense for the RTTI and the VTable not to necessarily be in the same TU?
   Yes, at least in the Itanium ABI. While there is a referende to the RTTI from the vtable, it is not required to appear in the same TU.

4 Proposed wording

Will be provided on request.

5 Acknowledgments

Thanks to John McCall for the initial idea.

6 References

[N4762] Richard Smith, *Working Draft, Standard for Programming Language C++*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4762.pdf