Authors: Jonathan Wakely and Bjarne Stroustrup
Date: 2011-02-26
N3257=11-0027

Range-based `for` statements and ADL

This paper summarizes of an issue raised on the committee reflectors [1] which identified a potential problem with the new range-based `for` statement in C++0x. The problem can be demonstrated by this example:

```
#include <vector>

namespace n
{
  struct X { void begin(); };
  struct Y { void begin(); };

  template<typename T> void begin(T& t) { t.begin(); }
}

int main()
{
  std::vector<n::X> v;

  for (auto i : v)  // error
  {
    // ...
  }
}
```

This produces the following error on the line indicated:

```
error: call of overloaded 'begin(std::vector<n::X>)&' is ambiguous
```

The ambiguity arisees because 6.5.4 [stmt.ranged] specifies that a range-based `for` statement calls `begin` and `end` and that:

> `begin` and `end` are looked up with argument-dependent lookup (3.4.2). For the purposes of this name lookup, namespace `std` is an associated namespace.

In the example `n` is also an associated namespace, so ADL finds `std::begin` and `n::begin`. Both templates are "greedy" and accept any type, so neither is more specialized and overload resolution cannot select the best function. Because the unqualified calls are not present in the code but are inserted by the implementation the author cannot qualify them to resolve the ambiguity. The only easy way to make the code compile is to ensure `n::begin` is not in scope, which may not be possible if the contents of `namespace n` are in a third-party header.

Although in the example only the call to `begin` is ambiguous, the same problem exists with `end`. The problem is not purely speculative and has already been encountered by users of Boost.Range and compilers providing C++0x features.

Problems due to ADL have been known for some time (see [2] and [3]) but this case is caused by the introduction of a new C++0x core language feature and is difficult for users to avoid. Changing the specification of range-based `for` would not break any existing code so it would be unfortunate to publish a new standard with this problem present.

There are two sides of the problem.

- The new C++0x functions `std::begin` and `std::end` are "greedy" and match anything. Often ambiguities with such functions can be resolved by qualifying calls with a namespace, but this isn't possible for generic code which wants to use those functions as customization points and therefore relies on ADL to find suitable user-defined overloads. This means that namespace `std` effectively claims ownership of the customization point, similarly to `std::swap`. This makes sense for `swap` where the default version provided by `std::swap` is a sensible default. The C++0x draft makes `begin` and `end` into similar curomization points, owned by namespace `std`. Because those functions weren't customization points in C++03 this can break existing code.
- The second problem is that the range-based `for` statement relies on ADL to find the `begin`/`end` functions, with no way to disable ADL in cases where it doesn't work.

As a result of these two problems libraries should not define their own greedy `begin`/`end` functions.

Suggested Solutions

Solutions where offered in the original thread following [1], then those solutions and several others where suggested, refined and/or discarded on the lib reflector in the thread beginning with [4]. The most popular suggestions are listed below.

N.B. none of these options alter how built-in arrays are handled by range-based `for`, there is no suggestion to change that, so these only apply to the second bullet at the end of [stmt.range].

0. Do nothing

It's very late to make a change to core wording. Users of for-range and library writers would have to avoid writing "greedy" templates called `begin` or `end`. This option did not seem to have a lot of support on the reflector.

1. Provide more-specialized overloads of `begin()` and `end()`.

The example would not be ambiguous if ADL found more specialized functions in namespace `std`, which could be defined for each standard container e.g.

```
template <class T, class A> typename vector<T,A>::iterator begin(vector<T,A>&
c);
template <class T, class A> typename vector<T,A>::const_iterator begin(const
vector<T,A>& c);
template <class T, class A> typename vector<T,A>::iterator end(vector<T,A>&
c);
template <class T, class A> typename vector<T,A>::const_iterator end(const
vector<T,A>& c);
```

These overloads would be selected instead of either of the "greedy" `begin` overloads.

This solution doesn't change the fact that the `begin`/`end` customization points are owned by namespace `std`, nor that range-based `for` uses ADL. It solves the problem for the given example, but doesn't help user-defined containers which meet the general container requirements (23.3.1) so containers authors would be required to provide their own specialized overloads of every container they write. This makes user-defined containers "second-class citizens" because (unlike the standard containers) they cannot be used with range-based `for` unless they provide non-member `begin` and `end` overloads for each type, in addition to providing `begin` and `end` member functions to meet the general container requirements. This is quite a significant burden for users just to enable a way of iterating over a range that works automatically for the standard containers.

2. Make qualified calls and define a new customization point.

The range-based `for` statement could be changed to make qualified calls so that ADL is not used.

This avoids ambiguities in range-based `for` statements, but means that an alternative technique must be used to enable `begin` and `end` to behave as customization points. The canonical way to do so is to allow users to specialize a class template such as:

```
template<class T>
struct range_traits
{
 static typename T::iterator begin(T& t) { return t.begin(); }
 static typename T::iterator end(T& t) { return t.end(); }

 // similar overloads for const T
};
```

This template could be used in two ways, either by changing range-based `for` to call `range_traits<_RangeT>::begin(__range)` and `range_traits<_RangeT>::end(__range)` directly or more generally by changing range-based `for` to call `std::begin` and `std::end` and making those functions use `range_traits`. The latter option is a smaller change to the core language and allows improvements or fixes to be made to the library only, by altering the definitions of `std::begin`/`std::end` if such fixes become necessary later on.

This supports user-defined containers that meet the container requirements without needing any additional code. Customization for other types is possible by specializing `range_traits`. Such specializations can choose to call `begin` and `end` from user-defined namespaces if desired, by making qualified calls e.g.

```
template<>
struct range-traits<std::vector<foo::bar>>
{
    static auto begin(std::vector<foo::bar>& v) -> decltype(foo::begin(v))
    { return foo::begin(v); }
    // similarly for const begin and for end ...
};
```

The downside of this option is that specialization is quite complicated and verbose compared to providing function overloads. It also claims ownership of the customization point: all customization related to `begin`/`end` must be done via the traits template. This option also fails to address the problem of ADL finding the greedy `std::begin` and `std::end` templates in contexts outside of range-based `for` statements.

3. Option 2 plus disabling ADL.

In addition to defining `range_traits` and making range-based `for` use qualified calls, the greedy overloads in namespace `std` could be moved into a nested namespace:

```
namespace std
 {
   namespace unspecified
   {
      // for exposition only
      template<class C>
        using RT = range_traits<typename remove_cv<typename
remove_reference<C>::type>::type>

      // std::begin() on any range-enabled container will get the begin
iterator
      template<class C>  auto begin(C&&  c) ->
decltype(RT::begin(forward<C>(c)));
      template<class C>  auto end(C&&  c) ->
decltype(RT::end(forward<C>(c)));
   }

   using namespace unspecified; // using directive suppresses ADL.
 }
```

This has the additional advantage that unqualified calls to `begin` and `end` will not find the greedy overloads, so that namespace `std` does not own the "begin" and "end" customization points. Qualified calls can be made if `std::begin` and `std::end` are really desired but they won't be found by ADL.

This is a fairly large change and alters the API in non-trivial ways using an idiom which is not used elsewhere in the standard library. Whatever faults the current design has, it went through the standardization process and was voted into the current draft.

4. Replace unqualified calls with calls to member functions.

Another suggestion early in the thread was to change range-based `for` to call `begin` and `end` member functions on the range instead of making unqualified calls with the range as an argument. This avoids ADL entirely, relying instead of the general container requirements. Types which do not provide those member functions can be used via an adaptor e.g.

```
template<typename T>
struct NonStdContainer {
    T* getFirst();
    T* getLast();
};

template<typename T>
struct adaptor {
    NonStdContainer<T>* ptr;
    T* begin() { return ptr->getFirst(); }
    T* end() { return ptr->getLast() + 1; }
};

void f(NonStdContainer<int>& c)
{
    for (auto i : adaptor<int>{&c})
    { /* ... */ }
}
```

Such adaptors are simple, and several such adaptors can be provided without conflicting, with the user choosing which one (if any) to use. This avoids problems present in the previous options that arise when two different libraries both try to specialize a customization point for the same type, introducing a new problem that is difficult for users of those libraries to resolve e.g. if two libraries both provide overloads of `begin(NonStdContainer<T>&)` or both provide specializations of `range_traits<NonStdContainer<T>>` then a user has no way to pick between them. If the libraries instead provide `bar::adaptor` and `baz::adaptor` they don't conflict and the user can choose whichever one they want.

5. Option 4, with fallback to unqualified calls.

A further suggestion was to call `begin()` and `end()` members if they exist, and if not then to make unqualified calls as specified in the current draft. This has the benefits of option 4 (absolute simplicity and absence of ambiguity opportunity for the simplest cases), while also allowing customization (at the risk of ambiguities or hijacking) via ADL. Users would have the option of using an adaptor or non-member overloads to resolve problems if necessary. This has a slightly higher burden on implementers to detect the presence of the member functions. On the

other hand, it preserves status quo for all containers that do not define member `begin()` and `end()`. It resembles the C# rules for range for.

What happens if a class provides a `begin` or an `end` that isn't suitable for iteration? My suggestion is that is a compile-time error. For example:

```
struct C {
   int* begin();
   // I forgot end() or misspelled end() or gave end the wrong type for end
};

C cont;
// ...
template<class T> T* begin(T);    // the nastiest example I could come up with in a hurry
template<class T> T* end(T);
// ...
for (auto x : cont ) // I'd prefer this to fail
```

I prefer the error to be caught rather than the likely user error be ignored and a non-local `begin(c)/end(c)` chosen. I think that this will lead to by far the fewest errors and confusions. This suggested rule does mean that if

(1) you have a class with a member `begin` or a member `end` that is not part of a `begin()/end()` pair and
(2) you want to use an object of that class as a sequence in a range-for loop

then you have to write and adapter as shown for "option 4" rather than simply a `begin()/end()` pair.

If you don't have begin or end members you can use a simple `begin()/end()` pair as currently.

Given that change the various range access `begin()` and `end()` standard library functions can be removed. For example:

```
// 18.9.3 initializer list range access
template<class E> const E* begin(initializer_list<E> il);

template<class E> const E* end(initializer_list<E> il);
```

24.6.5 range access [iterator.range]

```
template <class C> auto begin(C& c) -> decltype(c.begin());
template <class C> auto begin(const C& c) -> decltype(c.begin());
        Returns: c.begin().

template <class C> auto end(C& c) -> decltype(c.end());
```

template <class C> auto end(const C& c) -> decltype(c.end());
Returns: c.end().

template <class T, size_t N> T* begin(T (&array)[N]);
Returns: array.
template <class T, size_t N> T* end(T (&array)[N]);
Returns: array + N.

This removal is of course a separable from giving preference to member `begin()` and `end()`, but I think it would be a good idea to eliminate these potentially troublesome namespace functions when they are no longer essential.

Possible Solutions

Of the options 1 and 3 elicited the most opposition in the –lib reflector discussion. Option 1 is not general enough, only helping for the standard library containers types and not solving either of the underlying problems. Option 3 is too far-reaching and tries to solve potential ADL problems with "begin" and "end" but doesn't do the same for any other names which can potentially be affected in the same way.

Option 2 is a significant change to how a customization point is defined and used, while such a change might be desirable it is very late to make such a change. There has been interest in a "range" feature being added to the library in future, but even Boost has only recently acquired such a library and so the idea is not ready for standardization. Adding `range_traits` to the standard library now would make that the preferred customization point and would probably make it much more difficult to provide a better range API at a later date.

Option 4 is a very small change, requiring a simple, localized change to the core wording and no library changes, and doesn't rule out adding a richer range-based API in the future. It consists of a change of a single language rule.

Option 5 is a also very small change, requiring a simple, localized change to the core wording and no library changes, and doesn't rule out adding a richer range-based API in the future. It consists of the addition of a single language rule and the removal of now-redundant `begin()` and `end()` functions.

Proposed Wording

Option 4:

Make the following changes to 6.5.4 [stmt.ranged] paragraph 1:

> — otherwise, *begin-expr* and *end-expr* are `__range.begin()` and `__range.end()`, respectively. ~~`begin(__range)` and `end(__range)`, respectively, where `begin` and `end` are looked up with argument-dependent lookup (3.4.2). For the purposes of this name lookup, namespace `std` is an associated namespace.~~

Option 5:

Add the following as the middle (second) bullet point in the list in 6.5.4 [stmt.ranged] paragraph 1:

— otherwise, if _rangeT has a member `begin` or a member `end`, *begin-expr* and *end-expr* are `__range.begin()` and `__range.end()`, respectively.

In 18.9 Initializer lists [support.initlist] [1], delete

```
// 18.9.3 initializer list range access
template<class E> const E* begin(initializer_list<E> il);
template<class E> const E* end(initializer_list<E> il);
```

Delete section 18.9.3 Initializer list range access [support.initlist.range].

Delete section 4.6.5 range access [iterator.range].

Authorship comment

Jonathon Wakely wasn't able to complete this note in time and (with permission) Bjarne Stroustrup cleaned up the last few paragraphs and added comments base on the late February reflector discussion..

References

[1] c++std-ext-11387, "ADL bites", Dave Abrahams, 20 Dec 2010
http://accu.org/cgi-bin/wg21/message?wg=ext&msg=11387

[2] N1691, "Explicit Namespaces", David Abrahams, 7 Sep 2004
http://www.boostpro.com/writing/n1691.html

[3] N2103, "A Modest Proposal: Fixing ADL (revision 2)", Herb Sutter, 31 Oct 2006
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2103.pdf

[4] c++std-lib-29687, "Avoiding unqualified begin()/end() hijacking", Alberto Ganesh Barbati, 28 Dec 2010
http://accu.org/cgi-bin/wg21/message?wg=lib&msg=29687