Document number: **N3644**
Date: 2013-04-18
Project: Programming Language C++
Reference: N3485
Reply to: **Alan Talbot**
cpp@alantalbot.com

Null Forward Iterators

I have on several occasions found it quite awkward that you cannot create a valid iterator without a container instance. This is important because containers are usually accessed by means of ranges, either implicitly in the form of pairs of iterators, or explicitly using some form of range class. A range is self-consistent: it has no connection to the container instance into which it refers. I should therefore be able to create an empty range *without* an instance of the container. This can make a significant difference to a design, particularly since having a range of iterators on an actual instance of a container implies that instance must have a lifetime that encompasses the lifetime of the range.

For example, suppose I have a class hierarchy that provides iterator access to a member vector, along with other features. The base class does not actually have a vector, but some derived classes do:

```
struct A {
    virtual vector<int>::const_iterator begin();
    virtual vector<int>::const_iterator end();
    . . .
};

struct B : public A {
    virtual vector<int>::const_iterator begin();
    virtual vector<int>::const_iterator end();
    vector<int> v;
};

const A& ar = get_an_A(...);
for (int x : ar)
    do_something(x);
do_something_else(ar);
```

This is the case that actually came up in my code, but I can think of other use cases. I might want a container of ranges on vectors, and some of the elements in my container are “null”, meaning not only is the range empty, but there is no vector to refer to. I suspect that there are also interesting use cases involving strings.

But to implement any design that involves a range that may not always be able to refer to an actual live container, I have to resort to what feels like a kludge and is probably at least slightly less than optimally efficient.

The solution is to recognize the validity of null iterators by allowing value-initialized forward iterators to be compared, and ensuring that all value-initialized iterators for a particular container type compare equal. The result of comparing a value-initialized iterator to an iterator with a non-singular value is undefined.

```
vector<int> v = {1,2,3};
auto ni = vector<int>::iterator();
auto nd = vector<double>::iterator();

ni == ni;         // True.
nd != nd;         // False.
v.begin() == ni; // Undefined behavior (likely false in practice).
v.end() == ni;   // Undefined behavior (likely false in practice).
ni == nd;         // Won't compile.
```

Proposed Wording

24.2.5 Forward iterators **[forward.iterators]**

2 The domain of == for forward iterators is that of iterators over the same underlying sequence. However, value-initialized iterators may be compared and shall compare equal to other value-initialized iterators of the same type. [Note: value initialized iterators behave as if they refer past the end of the same empty sequence - end note]

Acknowledgements

Alisdair Meredith suggested using value-initialized iterators, and provided the note in the wording.