

Document number: **N3450=12-0140**
Date: 2012-09-23
Project: Programming Language C++
Reference: N3376=12-0066
Reply to: **Alan Talbot**
cpp@alantalbot.com

Iterator-Related Improvements to Containers

Abstract

This proposal recommends several small enhancements to the way containers interact with iterators. While none of these introduces functionality that cannot be achieved by other means, they make containers easier to use and teach, and make user code smaller and easier to read.

Last

I frequently find that I need an iterator to the last element of a container. Accessing the first and last elements directly is fully supported by **front** and **back**, but complementary access through an iterator is available only for the first element with **begin**. **last** provides the same semantics as **begin**, but for the last element of the container. If the container is empty, **last** returns **end**, otherwise it returns the last element. Note that **last** completes the symmetry of the design:

<i>Iterators</i>	<i>Element Access</i>
begin	front
last	back
end	

I propose adding **last** and **clast** to all containers (except **forward_list**), and to **basic_string** (mostly for symmetry and to avoid surprises). I do not find reverse iterators useful in very many situations, so I have not found a use for **rlast**, but **rlast** and **crlast** are included for consistency.

Proposed Wording

The wording will add the following signatures to **basic_string** and all containers except **forward_list**.

```
iterator last() noexcept;  
const_iterator last() const noexcept;  
reverse_iterator rlast() noexcept;  
const_reverse_iterator rlast() const noexcept;  
const_iterator clast() const noexcept;  
const_reverse_iterator crlast() const noexcept;
```

Complete wording will be provided in a revision of this paper.

Conversion between iterators and indices

I sometimes find that I have an iterator to a sequence (perhaps the result of **find**), but I need an index to that position. At other times I have an index and need an iterator. The Standard containers do not provide an obvious way to convert between iterator and index. It is fairly easy to do, but I find that people (including myself) are not quite sure how to do it correctly, and the code doesn't clearly express the intent. (At least, I would tend to comment it.) Here is one way to write it:

```
vector<int> v = ...;
vector<int>::size_type index1 = 3;
auto iter1 = find_if(v.begin(), v.end(), ...);

auto iter2 = next(v.begin(), index1); // Convert index to iter.
auto index2 = distance(v.begin(), iter1); // Convert iter to index.
```

But is this actually correct? **distance** returns a **distance_type**, but I need a **size_type** for an index. It takes a pretty careful reading of the Standard to determine that it's valid to assume that a container will never be larger than a number representable by a positive **distance_type** (which I find rather surprising and which could be a problem in a certain environments, but that's another discussion). I'm also going to have some trouble with this code because I'm going to end up mixing signed and unsigned if (for example) I compare `index1` and `index2`.

To solve this I propose adding two member functions to containers that do the conversion for me: **to_iterator** and **to_index**. With these member functions, the return types will be consistent with indices, the underlying code will be optimal without my having to think about it, and my code will be more obvious and expressive of its intent:

```
auto iter2 = v.to_iterator(index1);
auto index2 = v.to_index(iter1);
```

I am not overly attached to these names if there are other suggestions (in fact, I originally had the "c" and "r" names fully spelled out).

Proposed Wording

The wording will add the following signatures to **basic_string**, **array**, **deque**, and **vector**.

```
size_type to_index(const_iterator) noexcept;
size_type to_index(const_reverse_iterator) noexcept;

iterator to_iterator(size_type) noexcept;
const_iterator to_iterator(size_type) const noexcept;
const_iterator to_citerator(size_type) const noexcept;

reverse_iterator to_riterator(size_type) noexcept;
const_reverse_iterator to_riterator(size_type) const noexcept;
const_reverse_iterator to_criterator(size_type) const noexcept;
```

Complete wording will be provided in a revision of this paper.

Null Iterators

I have found it quite awkward on several occasions 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. The clients of a range never see or care about the particular container instance. 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 a range containing 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 container to refer to. I suspect that there are also many 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 create the concept of a *null iterator*, and provide containers with a static member function that returns a null iterator. Null iterators are singular (they are not associated with a container) and they are not dereferenceable, but they *are* valid for a very limited number of operations. Null iterators of a particular type always test equal to themselves. They may *not* be legally compared to any valid iterator that is not a null iterator (such as **end**), because such a comparison would be between iterators that do not refer to the same underlying sequence.

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

ni == ni;           // True.
nd != nd;           // False.
v.begin() == ni;    // Undefined behavior (likely false in practice).
v.end() == ni;      // Undefined behavior (likely true in practice).
ni == nd;           // Undefined behavior (likely true in practice).
```

Proposed Wording

The wording will add the following signature to all containers and **basic_string**.

```
static iterator null_iterator() noexcept;
```

24.2.1 In general [iterator.requirements.general]

9½ A *null* iterator is an iterator that is not dereferenceable and has a singular value. If *n* and *m* are null iterators of a container type, *n* == *m* is a valid expression that evaluates to true, and *n* != *m* is a valid expression that evaluates to false. In all other respects, null iterators behave as other singular iterators. Null iterators are obtainable from a static member function of each container type.

10 An *invalid* iterator is an iterator that is not a null iterator and may be singular.²⁶⁶

Complete wording will be provided in a revision of this paper.

Mapped Type Iterators

I use maps a lot, for a lot of different things, but my most common use case is to implement a database-like table, with the primary key (ID) as the key type of the map and a record class as the mapped type. For this use, almost all of my iterator operations involve only the mapped type of the value pair—the key is used only to look up a record when lookup is needed, or occasionally to access the ID (but my records almost always have to know their own ID). Because of the nature of the map interface, this means my code always looks like this (or will, once I have a compiler with range-based for loops):

```
for (auto i : m)
{
    i->second.foo();
    i->second.bar();
    // etc...
    if (i->first == x) ... // Access to the key is fairly rare.
}
```

This is a notational nuisance, but the problem becomes much worse in generic contexts:

```
template<typename C>
void print(const C& c)
{
    for (auto i : c)
        cout << *i << endl;
}
```

I would like to call this with whatever container I happen to be using, but `operator<<` isn't overloaded on pair so it won't compile for maps. And if I implement `operator<<` for pairs, I still want to be able choose whether to print the key/mapped pair or only the mapped type.

The solution I'm proposing is to create a `mapped_adaptor_t` wrapper for maps that provides iterators that dereference to the mapped type, rather than the container's value type (the pair). These iterators will provide a `key()` member to access the key type. There will be a convenience function `mapped_adaptor` which provides automatic creation of the wrapper object. This solves my problem without making any changes to map. Now my code can look like this:

```
for (auto i : mapped_adaptor(m))
{
    i->foo();
    i->bar();
    // etc...
    if (i.key() == x) ...
}

foo(mapped_adaptor(m));    // Print only the mapped type.
foo(m);                   // Print the pair.
```

I'm not attached to these names if there are other suggestions.

Proposed Wording

Complete wording will be provided in a revision of this paper.