Document: N2980=09-0170
Date: 2009-10-20
Authors: Robert Klarer
klarer@ca.ibm.com
Bjarne Stroustrup
bs@cs.tamu.edu
Dan Tsafrir
dan.tsafrir@gmail.com
Michael Wong
michaelw@ca.ibm.com

SCARY Iterator Assignment and Initialization Revision 1

1 Revision History

This revision supports the allocator member types void_pointer and const_void_pointer, as described in N2982, "Allocators post Removal of C++ Concepts (Rev 1)."

2 Abstract

We propose a change to the standard that specifies the types on which a container's iterator templates are dependent. In particular, the types of a standard container's iterators should not depend on the container's `key_compare`, `hasher`, `key_equal`, or `allocator` types.

3 Background

This paper is a companion to document N2911, "Minimizing Dependencies Within Generic Classes for Faster and Smaller Programs," by Tsafrir, et al, which will be presented at OOPSLA '09.

3.1 What is a SCARY iterator initialization?

N2911 gives the following examples of SCARY[1] initializations:

```
set<int, C1, A1>::iterator i1;
set<int, C2, A1>::iterator i2 = i1; // different comparator
set<int, C1, A2>::iterator i3 = i1; // different allocator
```

[1] N2911 explains that the acronym SCARY "describes assignments and initializations that are **S**eemingly erroneous (**C**onstrained by conflicting generic parameters), but **A**ctually work with the **R**ight implementation (unconstrained b**Y** the conflict due to minimized dependencies)."

The initializations of i2 and i3 are valid if they have the same type as i1, which is currently an implementation-dependent issue that is not addressed by the standard.

3.2 Are SCARY assignments and initializations dangerous?

No, they just look dangerous. The comparator and allocator are properties of the container, not the iterator itself. There's no particular reason for a standard container's nested iterator type to depend upon the container's comparator or allocator type.

Often, standard container iterators are implemented as classes nested inside the corresponding container template's definition. An iterator type's dependency on the container's comparator and allocator types is the unintended result of this implementation technique.

3.3 Who supports SCARY iterator assignment/initialization today?

Implementations that support SCARY assignment and initialization do so by dispensing with the use of a nested class to represent the iterator template. Example:

```
template <class T>
struct _ListIterator {  // not actually nested
        // …
};

template <class T, class Allocator = allocator<T> >
struct  list {
        typedef _ListIterator<T> iterator;
        // …
};
```

N2911 uses the term *independent* to describe a standard container iterator whose type does not depend on the container's allocator, comparator, hasher, etc.

Container iterators are independent in implementations based on SGI STL, including libstdc++ and STLPort. Container iterators are not independent in implementations based on Dinkumware STL and early versions of Rogue Wave STL.

The most recent version of Rogue Wave STL has independent container iterators in its production mode, but some of the standard containers' iterators are not independent in debug mode. N2911 explains that these dependencies are not actually required for debugging purposes and can easily be removed.

3.4 Why support SCARY iterator operations?

N2911 explains the advantages of independent container iterators in detail and quantifies them with academic rigor. In particular, it demonstrates that SCARY operations are crucial to the performant implementation of common design patterns using STL components. It further shows that implementations that support SCARY operations reduce object code bloat by eliminating redundant specializations of iterator and algorithm templates.

Other reasons to support SCARY operations are:

1. to clarify behavior that is currently implementation defined
2. to eliminate a known source of portability problems that have arisen in practice
3. to reduce excessive compile times by eliminating redundant template instantiations
4. the C++ Standard Library is an important showpiece for contemporary C++ style, and it should reflect the best known C++ programming practices; we should seek to avoid gratuitous type distinctions in the library

3.5 What are the possible disadvantages of independent iterators?

If container iterators are not independent, certain program errors can be detected at compile time:

```
std::deque<int> d1 = {1, 2, 3};
std::deque<int, my_allocator<int>> d2 = {3, 4, 5};
p = d1.begin();
q = d2.end();
std::sort(p, q); // type error?
```

However, range-based algorithms will be a more general and reliable solution to this kind of problem. Reliance on the type system to detect iterator mismatch errors has a number of serious drawbacks:

1. it can’t detect errors in which both iterators have the same type but they refer to different container objects
2. it can’t detect errors in which both iterators refer to the same container object, but they don’t delimit a range (eg. `std::sort(d1.end(), d1.begin());`)
3. some containers’ nested iterator types (eg `std::array<T, N>::iterator`) will probably be implemented as typedefs to pointer types, so checking won’t be uniform; some containers may support type-based iterator mismatch checking, but `std::array` and perhaps others probably won’t.

4 Proposal

We propose that all of the standard containers be required to support independent iterators. We do not propose the same requirement for non-standard containers.

Add the following text after paragraph 10 in [container.requirements.general]:

All container types defined in this Clause meet the following additional requirements:

Type `X::iterator`, where `X` is a container class, represents either a pointer type or an instantiated class whose type arguments include `X::value_type` and any of the following (and nothing else):

- `X::difference_type`
- `X::pointer`
- `X::allocator_type::void_pointer`

Type `X::const_iterator`, where `X` is a container class, represents either a pointer type or an instantiated class whose type arguments include `X::value_type` and any of the following (and nothing else):

- `X::difference_type`
- `X::pointer`
- `X::const_pointer`
- `X::allocator_type::void_pointer`
- `X::allocator_type::const_void_pointer`

[*Note:* in particular, `X's` `iterator` and `const_iterator` types may not vary with changes to `X::allocator`, `X::key_compare`, `X::hasher`, or `X::key_equal`. *– end note.*]

Add the following text to [unord.req]:

All unordered associative container types defined in this Clause meet the following additional requirements:

Type `X::local_iterator`, where `X` is an unordered associative container class, represents either a pointer type or an instantiated class whose type arguments include `X::value_type` and any of the following (and nothing else):

- `X::difference_type`
- `X::pointer`
- `X::allocator_type::void_pointer`

Type `X::const_local_iterator`, where `X` is an unordered associative container class, represents either a pointer type or an instantiated class whose type arguments include `X::value_type` and any of the following (and nothing else):

- `X::difference_type`
- `X::pointer`
- `X::const_pointer`
- `X::allocator_type::void_pointer`
- `X::allocator_type::const_void_pointer`

5 Implications to backwards compatibility

5.1 Source compatibility

We do not propose changing the general container requirements, so user-defined containers that do not currently support SCARY operations will continue to satisfy these requirements.

Since this proposal merely relaxes a restriction by ensuring the availability of SCARY operations, it won't render incorrect any use of the standard containers in user source code.

5.2 Link compatibility

If an implementation of the C++ Standard Library that did not previously support SCARY operations is modified to support this proposal, then the linkage name of any library function or user function that has a parameter whose type is one of the standard container iterators will change.