Doc. No.: X3J16/95-0097
WG21/N0697
Date: May 28, 1995
Project: Programming Language C++
Reply To: J. Lawrence Podmolik
Andersen Consulting
jlp@chi.andersen.com

Clause 23 (Containers Library) Issues List
Revision 3

Revision History

Revision 1 - January 31, 1995. Distributed in pre-Austin mailing.

Revision 2 - March 2, 1995. To be distributed at the Austin meeting.

Revision 3 - May 28, 1995. Distributed in pre-Monterey mailing.

Notes: some discussion was condensed or elided for closed issues to keep the list to a reasonable size. Also, some compound issues were split into several separate issues and some problems with issue numbering were corrected.

Introduction

This document is a summary of the issues identified in Clause 23. For each issue the status, a short description, and pointers to relevant reflector messages and papers are given. This evolving document will serve as a basis of discussion and historical for Containers issues and as a foundation of proposals for resolving specific issues.

Issues

Work Group: Library
Issue Number: 23-001
Title: propose to add convenience functions to stl containers
Sections: 23.1.5 through 23.1.8 and 23.2.1 through 23.2.4
Status: closed

Description:

There are some common needs that are not currently provided as a class members of the stl container classes. These additional methods can be provided by individual programmers as needed, but in our experience are so commonly wanted that they deserve to be included as class members

Proposed Resolution:

Add the following method to all containers

```
size_type Container<T>::clear() {return erase(begin(),end());}
```

Note: the returned size_type matches another proposal that all methods which change the size of a container by "some" amount return that amount.

Add the following methods to containers that provide the equivalent void pop_something():

```
sections 23.1.7 through 23.1.8 (list, deque)
T pop_front_value()
        {T ret = front(); pop_front(); return ret; }

section 23.1.5 through 23.1.8 (vectors, list deque)
T pop_back_value()
        {T ret = back(); pop_back(); return ret; }

sections 23.1.9 and 23.1.11 (stack and priority_queue)
T pop_value()
        {T ret = top(); pop(); return ret; }

section 23.1.10 (queue)
T pop_value()
        {T ret = front(); pop(); return ret; }
```

Note: The method names are a suggestion only. It is the returned T that is wanted.

Discussion:

<< deleted -- see Revision 2 for details >>

Resolution:

This issue was discussed in the LWG at the Austin meeting. It was noted that there are two distinct issues: adding clear() and adding push/pop functions. For clarity, this issue is being closed and split into two new, separate issues:

23-017 adding clear()
23-018 adding pop() functions

Requestor: Frank Griswold: griswolf@roguewave.com
Owner:

Work Group: Library
Issue Number: 23-002
Title: should some STL members return an iterator?
Also: minor clarification of member insert()
Sections: 23.1.5 through 23.1.8 and 23.2.1 through 23.2.4
Status: closed

Description:

There are some methods in all the STL containers that currently return void. These methods can be used "in a more natural way" for certain coding techniques if they return an iterator.

Also: The current description of the method

```
iterator insert(iterator location, const T&)
```

does not specify which iterator is returned.

Proposed Resolution:

If a container method might change the size of the container by exactly one, then it should return an iterator that points to the item inserted, or "just past" the item removed. (This covers both the needed clarification and the proposed change)

The method listed below should return an iterator providing the location of the change called for by the method. These changes should be made in parallel on all the containers mentioned in the WP sections listed above.

```
iterator Container<T>::erase(iterator);
```

The push and pop methods should return an iterator because doing so keeps the conceptual model consistent. Also note that for containers with only a forward iterator, the proposed version of push_back() returns information that could be expensive to recover.

```
(Sections 2.3.1.5 through 2.3.1.8)
iterator Container<T>::push_front(const T&) {... return begin();}
iterator Container<T>::push_back(const T&)
{... no way to do cheaply unless bidirectional ... return --end()}
iterator Container<T>::pop_front() { ... return begin(); }
iterator Container<T>::pop_back() { ... return end(); }
```

the following method should return an iterator because doing so keeps the conceptual model consistent.

```
iterator list<T>::splice(iterator position,
                         list<T>& donor, iterator donor_location)
```

Note that Container<T>::insert(iterator,const T&) already returns an iterator for all classes in the STL.

Discussion:

<< deleted -- see Revision 2 for details >>

Resolution:

This issue was discussed by the LWG at the Austin meeting. The general proposal to add iterator return values to various container member functions did not generate enough support to be brought before the full committee. Therefore, this issue was closed.

However, it was noted that the description of insert() must specify which iterator value is returned. The intent it to return an iterator to the element just inserted.

Requestor: Frank Griswold: griswolf@roguewave.com
Owner:

Work Group: Library
Issue Number: 23-003
Title: problems of nomenclature in STL classes
Sections: 23.1.5 through 23.1.8 and 23.2.1 through 23.2.4
Status: closed

Description:

-- list<type,allocator> has the only method that takes a predicate object: remove_if(Predicate). This method's name should be changed to "remove."

-- The STL container classes provide methods to "erase" or "remove" data; but the names of those methods are not consistent with their semantics across classes.

-- Section 17.2.2.4.2 refers to "associative containers" These are

not all associative (in the sense of an association between a key and a value). They are containers which internally mediate the location of their contained data rather than allowing the user to place the data. This request for renaming may be editorial.

Proposed Resolution:

-- No container class member name contain a trailing "_if," since overloading based on the signature of the method is sufficient.

This will change the WP only for class list
23.1.7 class declaration; change remove_if to remove
23.1.7.2 paragraph 8 remains unchanged

-- Also propose that the following conceptual model be used in naming methods which take data out of the container:

If all the data that in some sense "matches" a key or a predicate object is removed, then the name of the method is "remove" parallel to list<T>::remove()) of section 23.1.7.2 paragraph 8. By another proposal, these methods would also return a size_type instead of void.

If data is erased at an iterator or in the range between two iterators, the method is named "erase" parallel to the erase() defined in 17.2.2.4.1 table 22. By another proposal, these methods would return either a size_type (range) or an iterator "just past" the point of erasure (single iterator).

In summary: methods named erase "just erase right here" but methods named remove "search out and destroy."

This will change the WP in sections:
17.2.2.4.2 table 25 associative containers change the first a.erase(k) to a.remove(k). Note the returned size_type which is already in conformance with the other proposal.
23.2.1, 23.2.2, 23.2.3, 23.2.4:
rename erase(const key_type&) to remove...

-- Also propose that section 17.2.2.4.2 be changed to refer to classes which are "not externally sequenced" or which are "internally sequenced" . This would suggest a parallel change in section 17.2.2.4.1 from "sequence" to "externally sequenced". We can live with any name which seems suitable to the editor or committee; and which does categorize the kind of container without linguistic traps.

Discussion:

<< deleted -- see Revision 2 for details >>

Resolution:

This issue was discussed by the LWG at the Austin meeting. None of these changes garnered sufficient support among the LWG to be brought before the full committee for a formal vote. So this issue was closed.

Requestor: Frank Griswold: griswolf@roguewave.com
Owner:

Work Group: Library
Issue Number: 23-004
Title: should STL classes have fixed comparator semantics?
Section: 17.2.2.4 (table 22), 23.1.5 through 23.2.4
Status: active

Description:

Table 22 specifies that the semantics of
operator==(const Container<T> a, const Container<T> b)
is "a.size() == b.size() && equal(a.begin(), a.end(), b.end())"

This use of the algorithm equal() forces containers to hold data in the same order if they are to be ==. While this is often reasonable, it is not always the meaning that is wanted, particularly for data that is being used as an unordered (logical, not stl) set.

Table 22 also specifies that the semantics of
operator< (const Container<T>, const Container<T>)
is "lexicographical_compare(a.begin(),a.end(),b.begin(),b.end())"

As with operator==, this requirement for lexicographic ordering among containers, while often useful is not invariably what is wanted, particularly when the container is being used as an unordered (logical) set.

Proposed Resolution:

Provide suitable specialized meanings for the various operators by providing a traits class for each container. The standard should require (in sections 23.1.5 through 23.2.4) that these traits classes be specialized as follows:
list, vector, deque: lexical comparison semantics
(multi)set, (multi)map: set inclusion semantics

In section 17.2.2.4 there should be a discussion like this:

Any container which meets the STL specification must have an associated specialization of the container_traits class which provides a typedef
binary_function<const container<T>&, const container<T>&, int) comparitor;
which may be used to provide the 6 comparison operators on that container. A complete description of the container must include a description of the comparison semantics provided for that container. Table 22 of section 17.2.2.4 would change in boxes defining operational semantics of operator==() and operator<() to say something like
"either lexically comparison or set-inclusion comparison, depending on the container"

There needs to be a discussion/definition of
template <class container, class T> lexical_comparitor;
and
template <class container, class T> set_comparitor

I'm not sure where these go. Chapter 25? lexical_comparitor makes use of lexicographical_compare (25.3.8), set_comparitor might make use of includes (25.3.5.1) but might not, depending on the container.

Here is container_traits:

```
template <class container, class T> struct container_traits {
  typedef binary_function<const container<T>&, const container<T>&, int>
              comparitor;
```

```
  };

  Here is a required (partial) specialization:
  template <list, class T> struct container_traits {
    typedef lexical_comparitor<list,T>  comparitor;
  };

Discussion:

        << deleted -- see Revision 2 for details >>

Resolution:

        NOTE: Discussion of this issue was tabled by the LWG in Austing
        pending a discussion of the STL hash table proposal.  The hash
        table proposal was not accepted, but the LWG did not have time
        to return to this issue.

Requestor:      Frank Griswold: griswolf@roguewave.com
Owner:

------------------------------------------------------------------------

Work Group:     Library
Issue Number:   23-005
Title:          should some STL members return a size_type?
Sections:       23.1.5 through 23.1.8 and 23.2.1 through 23.2.4
Status:         closed

Description:
  There are some methods in the STL containers that currently return
  void. These methods can be used "in a more natural way" if they
  return a count of the number of items removed or added. Note similar
  proposal requesting that some methods return iterators.

Proposed Resolution:
  If a container method might change the size of the container by
  an unknown amount, it should return the amount of the change.

  The methods listed below should return a size_type indicating the
  number of items added to or removed from the container.  These
  changes should be made in parallel on all the containers mentioned
  in the WP sections listed.

    section 23.1.5 through 23.1.8 and section 23.2.1 through 23.2.4
    size_type Container<T>::erase(iterator start, iterator boundary);

    section 23.1.5 through 23.1.8
    template <class InputIterator>
    size_type Container<T>::insert(iterator location
                InputIterator first, InputIterator boundary);

    section 23.2.1 through 23.2.4
    template <class InputIterator>
    size_type Container<T>::insert(
                inputIterator first, InputIterator boundary);

    section 23.1.5 through 23.1.8
    size_type Container::insert(iterator location, size_type, const T&);

    section 23.2.1 through 23.2.4
    size_type Container::insert(size_type, const T&);

  The methods listed below should return a size_type indicating the
```

number of items added to or removed from the container. These changes apply only to the class: list<T>

section 23.1.7 and 23.1.7.2 para 10
size_type list<T>::merge(list<T>&);

section 23.1.7 and 23.1.7.2 para 5
size_type list<T>::splice(iterator position, list<T>&);

section 23.1.7 and 23.1.7.2 para 7
size_type list<T>::splice(iterator position,
iterator start, iterator boundary);

section 23.1.7 and 23.1.7.2 para 9
size_type list<T>::unique();

section 23.1.7 and 23.1.7.2 para 9
template <class BinaryPredicate>
size_type list<T>::unique(BinaryPredicate);

section 23.1.7 and 23.1.7.2 para 8
size_type list<T>::remove(const T&);

section 23.1.7 and 23.1.7.2 para 8
template <class Predicate>
size_type list<T>::remove_if(Predicate);
Note: this method's name is the subject of another proposal

Discussion:

<< deleted -- see Revision 2 for details >>

Resolution:

This issue was discussed by the LWG at the Austin meeting. Changing these return types did not result in a consensus among the LWG. So this issue was closed.

Requestor: Frank Griswold: griswolf@roguewave.com
Owner:

Work Group: Library
Issue Number: 23-006
Title: naming inconsistencies in bits<T>
Sections: 23.1.1 [lib.template.bits]
Status: closed

Description:

vector<bool> uses "flip()" to toggle a bit, bits<N> uses "toggle()". I asked this question almost 3 years ago, and people didn't care much either way. The international folks then present mildly favored "toggle". Not wanting to waste time on such trivia, and trusting that Alex has as much right to speak for international users as anyone, I suggest that we use "flip" in place of "toggle" in bits<N> (it occurs in two places).

Likewise, let's rename "length()" to "size()" (if not done already), to be in-sync with the rest of STL.

I have for some time suggested that we rename bits<N> to

bitset<N>, making it less awkward to talk about ("bits" is not a singular noun). This is not a priority, but I would surely like to hear some feedback at least once in a row.

I have also suggested that "to_ushort()" is redundant, since we have to_ulong().

Resolution:

Discussed in the LWG at Austin. All four of these changes were discussed, accepted and incorporated into the WP.

Requestor: Chuck Allison <72640.1507@compuserve.com>
Owner:
Emails: (none)
Papers: (none)

Work Group: Library
Issue Number: 23-007
Title: adding vector<bool>::flip that toggles all bits
Sections: 23.1.1 [lib.template.bits]
Status: closed

Description:

vector<bool> uses flip() only as something one can do with a "reference", a surrogate for single-bit access. Is there anything wrong with adding a vector<bool>::flip to toggle all bits, like bits<N> does?

Resolution:

Discussed and approved in the LWG at Austin. Adding the member function

```
void vector<bool>::flip()
```

was proposed and accepted into the WP.

Requestor: Chuck Allison <72640.1507@compuserve.com>
Owner:
Emails: (none)
Papers: (none)

Work Group: Library
Issue Number: 23-008
Title: add a nested reference class to bits<T>
Sections: 23.1.1 [lib.template.bits]
Status: closed

Description:

I propose that we add a nested reference class to bits<N>, similar to vector<bool>, to allow for bits<N>::operator[]. To be precise, add:

```
template<size_t N>
class bits
{
public:
```

```
                class reference
                {
                public:
                        reference& operator=(bool);     // for b[i] = x;
                        reference& operator=(const reference&);
                                                        // for b[i] = b[j];
                        bool operator~() const;         // for ~b[i]
                        operator bool();                // for x = b[i];
                        reference& flip();              // for b[i].flip();
                };

                reference operator[](size_t);           // for b[i]

                // the rest as-is in bits...
        };
```

Note no public constructor for reference. I have implemented this successfully in Borland C++. Note also that there is still no reason for introducing an iterator class for bits<N>.

Resolution:

Discussed in the LWG at Austin. The proposed change passed a LWG straw vote unanimously and was voted into the WP by the full committee.

One new issue arose: that the nested reference classes in vector<bool> and bits (now "bitset") should have explicit private destructors. See issue 23-016.

Requestor: Chuck Allison <72640.1507@compuserve.com>
Owner:
Emails: (none)
Papers: (none)

Work Group: Library
Issue Number: 23-009
Title: adding a "default value" argument to map/multimap constructors
Sections: 23.2.3 [lib.map], 23.2.4 [lib.multimap]
Status: closed

Description:

As currently defined, when operator[] is applied to a map or a multimap and a new entry is inserted into the map as a result, the new value is initialized using the default constructor T(). This is not always desirable - sometimes it is useful to specify another default value.

The USL library solved this problem by providing an alternate constructor wherein the user could specify the value to be used when new ("empty") entries were automatically inserted into the map.

Such an option could be added to map and multimap in the current WP. The analogous map/multimap constructors might look something like this:

```
map(const T& = T(), const Compare& comp = Compare());

multimap(const T& = T(), const Compare& comp = Compare());
```

Resolution:

Discussed in the LWG at Austin. There was some discussion about reversing the order of the T() and Compare() arguments, but a LWG straw vote revealed the proposed change as a whole lacked sufficient support to bring before the full committee. The issue was closed.

Requestor: Larry Podmolik <jlp@chi.andersen.com>
Owner:
Emails: (none)
Papers: (none)

Work Group: Library
Issue Number: 23-010
Title: Should the WP specify requirements for container template class T's?
Sections: 23 [lib.containers]
Status: active
Description:

The current WP does not place explicit requirements (that I could find) on class T (the value_type) for container classes.

It appears that for (most? many? all?) containers, T must either have an accessible default constructor, copy constructor, operator=, and destructor, or the compiler must be able to generate them.

Where present, similar requirements probably apply to class Key and other template arguments throughout the library clauses.

Implementors need to know the requirements so that they can avoid use of class member functions not required to be present (or compiler generatable) and accessible.

Proposed Resolution:

The WP should specify requirements for template class T and class Key arguments for containers.

Resolution:

Discussed in the LWG at Austin. Beman Dawes said he would try to write a proposal for the next meeting that would address the general issue of requirements for template arguments.

So: the issue remains open pending further analysis.

Requestor: Beman Dawes <beman@dawes.win.net>
Owner:
Emails: (none)
Papers: (none)

Work Group: Library
Issue Number: 23-011
Title: Bits inserters/extractors need updating
Sections: 23.1.1.26 [lib.ext.bt], 23.1.1.27 [lib.ins.bt]
Status: active

Description:

The bits library's insertion/extraction operators need to be updated for the new iostreams

Proposed Resolution:

Change the operator interfaces from

```
istream& operator>>(istream& is,bits<N>& x);
ostream& operator<<(ostream& os,const bits<N>& x);
```

to:

```
basic_istream<charT,ios_traits<charT> >&
        operator>>(basic_istream<charT, ios_traits<charT> >& is,
                        bits<N>& x);

basic_ostream<charT,ios_traits<charT> >&
        operator<<(basic_ostream<charT, ios_traits<charT> >& os,
                        const bits<N>& x);
```

Resolution:

Discussed in the LWG at Austin. P.J. Plauger noted that making these kinds of changes required a thorough review of all the library sections to determine where these types of changes were required, and should not be done in an ad hoc fashion. There was also some uncertainty about syntax issues and the compiler's ability to deduce all the correct types in the general case.

So: the issue remains open pending further analysis or a more detailed proposal.

Requestor: Judy Ward <ward@roguewave.com>
Owner:
Emails: (none)
Papers: (none)

Work Group: Library
Issue Number: 23-012
Title: Templatize bits members that interact with basic_string
Sections: 23.2.1.1 [lib.cons.bits], 23.2.1.13 [lib.bits::to.string]
Status: active

Description:

The members of bits that take arguments of basic_string must be updated to be template members

Proposed Resolution:

Replace the following two bits signatures:

```
bits(const string& str, size_t pos = 0, size_t n = NPOS);

string to_string() const;
```

with:

```
        template <class charT>
        bits(const basic_string<charT,string_char_traits<charT> >& str,
                size_t pos = 0, size_t n = NPOS);

        template <class charT>
        basic_string<charT,string_char_traits<charT> > to_string() const;

Resolution

        Discussed in the LWG at Austin.  Nathan Myers said that the
        string_char_traits<> should not be specified in the function
        declaration, but rather should be an additional template
        parameter that default to the arguments shown in the current
        signatures.  Also, the references to size_t and NPOS need
        to be corrected.

        So: the issue remains open pending further analysis.

Requestor:      Judy Ward <ward@roguewave.com>
Owner:
Emails:         (none)
Papers:         (none)

----------------------------------------------------------------------

Work Group:     Library
Issue Number:   23-013
Title:          Return values from library class member functions
Sections:       23 [lib.containers]
Status:         closed

Description:

        Why do many member functions have void return values rather
        than returning *this? The STL does not seem to follow this
        standard idiom, which is unfortunate as statements like

                sequence.sort().reverse();

        seem to make perfect sense. Has this been looked at / noticed?

Resolution:

        Discussed in the LWG at Austin, but failed to generate
        sufficient support.  Straw vote to close this issue passed
        unanimously.

Requestor:      Kevlin A P Henney <kevlin@wslint.demon.co.uk>
Owner:
Emails:         (none)
Papers:         (none)

----------------------------------------------------------------------

Work Group:     Library
Issue Number:   23-015
Title:          reference counted strings and begin()/end()
Sections:       21.1.1.2 [lib.basic.string]
Status:         active

Description:

        In Valley Forge, we accepted a version of basic_string<T>
        that was modified to be compatible with STL.  One of the
```

issues that arose was ensuring that a reference counted version of basic_string could be efficiently implemented.

One suggested implementation prevented structural sharing of strings whenever a non-const iterator was allowed to escape from the string. This can make innocent-looking code like the following inefficient:

```
string s;
string::const_iterator iter = s.begin();
...
```

The non-const version of begin() gets invoked, so we would have to prohibit all future sharing of s, even though we don't intent to modify it. To force the return of a const iterator involves a messy looking const_cast to a reference type.

So: should we create distinct names for the functions that return const iterators?

Resolution:

Discussed in the LWG at Austin but no consensus was reached.

So: the issue remains open pending further analysis.

Requestor: Larry Podmolik <jlp@chi.andersen.com>
Owner:
Emails: c++std-lib-2981, c++std-lib-2985
Papers: (none)

Work Group: Library
Issue Number: 23-016
Title: adding explicit private constructors to vector<bool>::reference and bitset<N>::reference
Sections: 23.2.1 [lib.template.bitset], 23.2.6 [lib.vector.bool]
Status: active

Description:

(This issue arose while discussing issue 23-008 in Austin.)

The nested reference classes in vector<bool> and bitset currently do not specify any constructors. During LWG discussions in Austin, we agreed that both these nested classes should have private constructors so that objects of the reference types could not be created in user code.

Proposed Resolution:

Modify the declarations of vector<bool>::reference and bitset<N>::reference to add a private constructor. Also add a friend declaration for the corresponding (enclosing) container class to allow the containers to create instances of the reference types.

Resolution:

Requestor: Larry Podmolik <jlp@chi.andersen.com>
Owner:
Emails: (none)

Papers: (none)

Work Group: Library
Issue Number: 23-017
Title: add clear() to all containers
Sections: 23.2.2 [lib.deque], 23.2.3 [lib.list],
23.2.5 [lib.vector], 23.2.6 [lib.vector.bool],
23.3.1 through 23.3.4
Status: active

Description:

Add a convenience function clear() to all containers in Clause 23 that currently have an erase() member function.

Proposed Resolution:

Add the following member function to all of the containers in Clause 23 that current define erase (deque, list, vector, vector<bool>, set, multiset, map and multimap):

```
void clear();
```

whose semantics are

```
void clear() {return erase(begin(),end());}
```

Discussion:

This is taken directly from an earlier issue (23-001), except that the return type is now void (instead of size_type), since the proposal to change these return types (see 23-005) was not accepted.

Resolution:

Requestor: Frank Griswold: griswolf@roguewave.com
Owner:

Work Group: Library
Issue Number: 23-018
Title: add additional pop() functions to containers
Sections: 23.2.2 [lib.deque], 23.2.3 [lib.list],
23.2.4.1 [lib.queue], 23.2.4.2 [lib.priority.queue],
23.2.4.3 [lib.stack], 23.2.5 [lib.vector],
23.2.6 [lib.vector.bool]
Status: active

Description:

Add additional pop() members that return the popped value as well as modifying the container.

Proposed Resolution:

Add the following methods to containers that provide the equivalent void pop_something():

--> To 23.2.2 [lib.deque] and 23.2.3 [lib.list]:

```
                T pop_front_value()
                        {T ret = front(); pop_front(); return ret; }

        --> To 23.2.2 [lib.deque], 23.2.3 [lib.list],
            23.2.5 [lib.vector] and  23.2.6 [lib.vector.bool]:

                T pop_back_value()
                        {T ret = back(); pop_back(); return ret; }

        --> To 23.2.4.2 [lib.priority.queue] and 23.2.4.3 [lib.stack]:

                T pop_value()
                        {T ret = top(); pop(); return ret; }

        --> To 23.2.4.1 [lib.queue]:

                T pop_value()
                        {T ret = front(); pop(); return ret; }

Discussion:

        This is one part of an earlier issue (23-001).

Resolution:

Requestor:      Frank Griswold: griswolf@roguewave.com
Owner:

------------------------------------------------------------------------

Work Group:     Library
Issue Number:   23-019
Title:          make Allocator argument in containers const refs
Sections:       23.2.2 [lib.deque], 23.2.3 [lib.list],
                23.2.5 [lib.vector], 23.2.6 [lib.vector.bool],
                23.3.1 through 23.3.4
Status:         active

Description:

        Default Allocator arguments for containers must be const
        references (vs. the non-const references currently in the
        WP).

Proposed Resolution:

        Change all defaulted Allocator arguments in the Clause 23
        containers from
                Allocator& = Allocator()
        to
                const Allocator& = Allocator()

Discussion:

        The WP currently says that the value returned by a constructor
        type call is an rvalue.  A non-const reference argument must
        be initialized by an lvalue.

        Note: the same change applies to basic_string<T> in Clause 21.

Resolution:

Requestor:      Judy Ward <ward@roguewave.com>
Owner:
```

Emails: c++std-lib-3730, c++std-lib-3731, c++std-lib-3732, c++std-lib-3733
Papers: (none)

Work Group: Library
Issue Number: 23-020
Title: Stack and Queue interface is limited and unnatural
Sections: 23.2.4
Status: active

Description:

The current definitions of stack and queue look like:

```
template <class Container>  class stack { ... };
template <class Container>  class queue { ... };
```

To use one requires code like

```
stack< deque<T> > myStack;
```

This differs sharply from "real" collections:

```
vector<T> myVector;
```

and leads to confusion. Furthermore, it requires the user to specify more than is necessary, by not providing a useful default choice of container.

Worse, it prevents use of collections that use a runtime-variable allocator, so such objects could not be placed in object databases.

Proposed Resolution:

1. Simply change each declaration above to read:

```
template <class T, class Container = deque<T>,
                   Allocator = allocator>
class stack { ... };

template <class T, class Container = deque<T>,
                   Allocator = allocator>
class queue { ... };
```

   and provide each with a constructor:

```
explicit stack(const Allocator& = Allocator());
explicit queue(const Allocator& = Allocator());
```

2. If we accept (1), we should also vote on a change to priority_queue as follows:

```
template <class T, class Comp = less<T>,
        class Container = vector<T>, Allocator = allocator>
class priority_queue { ... };
```

   and provide a constructor:

```
explicit priority_queue(const Comp& = Comp(),
        const Allocator& = Allocator());
```

The description of the constructor simply says that they pass along the Allocator argument to the member collection.

Discussion:

These changes allow usage from the very natural:

```
stack<T> myStack;
```

to the fully general:

```
stack<T,vector<T,ODBAllocator>,ODBAllocator> myStack(myODB);
```

In the last line above, a stack has been declared based on a vector, using a memory model appropriate to placement in an object database.

The increase in generality comes at very small cost in complexity, and with a great improvement in consistency with the rest of the library.

NOTE: Plauger notes the following in c++std-lib-3741 (paraphrased):

> A container fixates on an allocator when you construct it -- it's type is even a template parameter. There is no mechanism that lets you ``pass along'' an Allocator argument to an already constructed container, even if you have some assurance that the types of the allocators are the same.

Resolution:

Requestor: Nathan Myers <myersn@roguewave.com>
Owner:
Emails: c++std-lib-3735, c++std-lib-3737, c++std-lib-3738, c++std-lib-3741, c++std-lib-3743
Papers: (none)
