Doc. No.: WG21/N1014
X3J16/96-0196
Date: November 11, 1996
Project: C++ Standard Library
Reply to: Pete Becker
pbecker@oec.com

# Clause 23 (Containers Library) Issues List

# Revision 10

Revision History

---

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

Revision 2 - March 2, 1995. 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.

Revision 4 - July 11, 1995. Updated and distributed at the Monterey meeting.
Includes several issues generated from the first round of X3J16 public review comments, as well as issues resulting from editorial boxes in the April 28, 1995 version of the WP.

Revision 5 - July 31, 1995. Distributed in post-Monterey mailing.
Updated to reflect issues closed at the Monterey meeting, Also includes several new issues resulting from the X3J16 public review comments and from discussions at Monterey.

Revision 6 - October 29, 1995. Distributed at the Tokyo meeting.
Includes issues that remained open following the Monterey meeting, plus a significant number of new issues. For brevity, this revision lists the full text only of ongoing and new issues; issues closed up to and including the Monterey meeting are summarized below.
Note: Working Paper references in this revision are to the pre-Tokyo draft dated 26 September 1995.

Revision 7 - November 30, 1995. Distributed in the post-Tokyo mailing.
Updated to reflect issues closed at the Tokyo meeting. Also includes new issues raised (but not addressed) at the Tokyo meeting and any issues identified since that meeting.

Revision 8 - May 28, 1996. Distributed in the pre-Stockholm mailing.

Revision 9 - July 5, 1996. Distributed at the Stockholm meeting.

Revision 10 - November 11, 1996. Distributed at the Kona meeting. Pete Becker took over editing from Larry Podmolik.

## 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.

## Summary of Open Issues

23-043 Fix container ambiguities when T == size_type
23-063 Should set / multiset define mapped_type?
23-064 Are comparators held by value or by reference?
23-065 Can comparators be function pointers vs. objects?
23-066 Need comparator copy / assign semantics
23-067 Fix description of bitset operator<<()
23-068 Make bitset constructor signatures consistent
23-069 Add pop_value() to container adapters
23-070 Clean up descriptions for capacity() and reserve()
23-071 Do adapters need allocator arguments?
23-072 Clean up vector<bool> declarations
23-073 Map / multimap::value_compare::operator() should be const
23-074 Why no copy constructor or assignment for bitset?
23-075 Add resize() to optional operations, fix description
23-076 Fix reverse_iterator typedefs in deque and vector
23-077 Why doesn't queue have a top() member function?
23-078 Naming: difference_type vs. distance type
23-079 insert(p,t) should not have default argument
23-080 Resolve map::mapped_type vs. map::referent_type

## Summary of Closed Issues

23-001 Add convenience functions to STL containers
23-002 Should some STL members return an iterator?
23-003 Nomenclature problems in STL classes
23-004 Should STL classes have fixed comparator semantics?
23-005 Should some STL members return a size_type?
23-006 Naming inconsistencies in bits<T>
23-007 Adding vector<bool>::flip that toggles all bits
23-008 Add a nested reference class to bits<T>
23-009 Add "default value" arg to map / multimap constructors
23-010 Requirements for type T in template containers
23-011 Bitset inserters / extractors need updating
23-012 Templatize bits members for basic_string
23-013 Return values from library class member functions
23-014 Add hash tables to standard library
23-015 Reference counted strings and begin() / end()
23-016 Adding constructors to nested reference types

23-017 Add clear() to all containers
23-018 Add additional pop() functions to containers
23-019 Make Allocator argument in containers const refs
23-020 Change container adapter interfaces
23-021 Modify complexity of swap() due to allocators
23-022 Add typedef, member to retrieve allocator type
23-023 Specify container iterators as opaque types
23-024 Fix copy constructors w.r.t. allocators
23-025 Remove bitset exposition implementation
23-026 Update vector<bool> with partial specialization
23-027 Make vector<bool> bit ref swap a static member
23-028 Clean up empty sections in Clause 23
23-029 Fix vector constructor signatures in description
23-030 Update descriptions of deque operations
23-031 Specialize swap() algorithm for containers
23-032 Non-const top() missing in priority_queue?
23-033 Clean up resize() effects for deque, list and vector
23-034 Reverse iterator types for list
23-035 Correct argument list to vector<bool>::insert
23-036 Need semantics for at() member deque / vector
23-037 Semantics for a.back() in sequence requirements
23-038 Specify iterator properties for Clauses 21 & 23
23-039 Reconsider return type of erase(iterator)
23-040 Need typedefs for map / multimap T type
23-041 Possible solutions for map::insert()
23-042 Fix default container for priority_queue
23-044 Inconsistent insert() return types for assoc. containers
23-045 Remove <stdexcept> from <bitset> synopsis
23-046 Clean up bitset element access methods
23-047 Clarify complexity for deque::erase()
23-049 Clarify complexity for vector::insert(p,i,j)
23-048 Improve description of list::sort()
23-050 Add additional constructors to Container requirements
23-051 Fix description of list::unique()
23-052 Fix description of list::merge()
23-053 vector<bool>::const_reference should be bool
23-054 Define vector<bool>::reference::operator==()
23-055 Fix return type of map::operator[]()
23-056 Remove const version of map::operator[]()
23-057 Need semantics for associative containers
23-058 Fix reverse iterator typedef arguments
23-059 Wrong reverse iterator type for associative containers
23-060 Fix postcondition for (&a)->~X() in requirements table
23-061 Reorganize Clause 23 sections
23-062 Remove() algorithm doesn't work on map / multimap

Issues

---

Work Group: Library
Issue Number: 23-043
Title: Fix container ambiguities when T == size_type
Sections: 23 [lib.containers]
Status: Active
Description:

Various types of calls to constructors & member functions are ambiguous for the case that the element of the container is a size_type: as long as C++ does not have constraints, the templates on InputIterator may conflict with the size / value methods.

A note should be added to explain how to disambiguate the constructors (do not default the allocator argument). A solution (possibly involving a defaultable dummy argument?) should be found for assign() and insert().

Proposed Resolution:
Requester: German delegation comments
Owner:
Emails: c++std-edit-579
Papers: (none)

---

Work Group: Library
Issue Number: 23-063
Title: Should set / multiset define mapped_type?
Sections: 23.3.3 [lib.set], 23.3.4 [lib.multiset]
Status: Active
Description:

For consistency with map / multimap, set and multiset define both key_type and value_type, even though these both refer to the same underlying type (T).

Following this line of logic, should set and multiset also define mapped_type?

Proposed Resolution:

No, mapped_type doesn't make any sense for set or multiset, as nothing is being mapped. Recommend closing this issue with no changes to the WP.

Requester: Angelika Langer (langer@roguewave.com)
Owner:
Emails: (none)

Papers: (none)

---

Work Group: Library
Issue Number: 23-064
Title: Are comparators held by value or by reference?
Sections: 23.1.2 [lib.associative.reqmts]
Status: Active
Description:

Are containers supposed to hold compare functions as values or as references?

Again, I couldn't find anything in the draft that specifies whether compare functions of a container are values or references internally, which of course makes a difference for the user.

The fact that the compare functions are constant references when provided to a container constructor seems to imply that they are internally held as references. Hence the user has to pay attention to the lifetime of the compare object.

On the other hand, the container constructors have a compare parameter which is defaulted by a temporary object. This gives the impression that the compare parameter will probably be copied and internally held as a value. In this case the user cannot work with polymorphic function objects because of the inevitable slicing, or has to find workarounds.

In any case, the users needs to know what the exact requirements to the compare function of an associative container are.

Proposed Resolution:
Requester: Angelika Langer (langer@roguewave.com)
Owner:
Emails: c++std-lib-4356
Papers: (none)

---

Work Group: Library
Issue Number: 23-065
Title: Can comparators be function pointers vs. objects?
Sections: 23.1.2 [lib.associative.reqmts]
Status: Active
Description:

I could not find any requirements imposed on the type of a

compare function in the working paper. The text tends to talk of "function object", but it is nowhere specified that a compare function needs to be an object. With most algorithms that take a compare function it seems to be reasonable to allow function objects as well as function pointers.

On the other hand with the associative containers the intent seems to be a little bit different. E.g. the constructors of those containers take a compare argument, which has a default value of Compare(), which is the default constructor of the Compare type provided as template parameter of the container. If a user wants to work with a function pointer instead of a function object he/she can do so. The only inconvenience is that he/she cannot rely on the default value and has to explicitly provide the compare parameter in all cases. (This would be true for function compare objects that have no default constructor as well.)

So, there seems to be no reason to assume that a compare function could not be a function pointer.

But then, a library implementer has the latitude to offer two constructors instead of one with a defaulted argument. In that case the default constructor would make some assumption about the default value for the compare function, which probably would be Compare() again. Hence with such an implementation it would not be possible to use function pointers.

Proposed Resolution:

Requester: Angelika Langer (langer@roguewave.com)
Owner:
Emails: c++std-lib-4356
Papers: (none)

---

Work Group: Library
Issue Number: 23-066
Title: Need comparator copy/assign semantics
Sections: 23.1.2 [lib.associative.reqmts]
Status: Active

Description:

What is the role of the compare function in copy constructors, assignments and swap functions of containers?

Imagine you had two associative containers of the same type holding two different compare objects. What is supposed to happen when you assign the one to the other? Which compare

object will be used when inserting the values from the source container into the target container? Will the compare object itself be copied as well, along with other internal data?

I tried to check out what HP's STL does. The example was the assignment operator of set. The result was fascinating ... and the target set was corrupted after this assignment. :-(

It is definitely necessary to clarify what the semantics of copy construction, assignment and swap are when compare objects are involved. (I can imagine that the same would be true when allocators are involved, too.)

Proposed Resolution:

Requester: Angelika Langer (langer@roguewave.com)
Owner:
Emails: c++std-lib-4356
Papers: (none)

---

Work Group: Library
Issue Number: 23-067
Title: Fix description of bitset operator<<()
Sections: 23.2.1.3 [lib.bitset.operators]
Status: Active

Description:

The description for bitset's operator<<() function currently reads:

Returns:

os << x.to_string() (_lib.ostream.formatted_).

This should be changed to:

Returns:

os << x.to_string<charT,traits>()

Proposed Resolution:

Change the description of bitset<N>::operator<<() in 23.2.1.3 [lib.bitset.operators] as described above.

Requester: Andy Sawyer (andys@thone.demon.co.uk)
Owner:
Emails: (none)
Papers: (none)

---

Work Group: Library
Issue Number: 23-068
Title: Make bitset constructor signatures consistent
Sections: 23.2.1 [lib.template.bitset],
23.2.1.1 [lib.bitset.cons]
Status: Active
Description:

The following bitset constructor signature appears in [lib.template.bitset]:

```
explicit bitset(const string& str, size_t pos = 0,
                size_t n = size_t(-1));
```

Yet in [lib.bitset.cons] it reads:

```
template <class charT, class traits, class Allocator>
explicit
bitset(const basic_string<charT, traits, Allocator>& str,
       basic_string<charT, traits, Allocator>::size_type pos = 0,
       basic_string<charT, traits, Allocator>::size_type n =
         basic_string<charT, traits, Allocator>::npos);
```

The latter is correct.

Proposed Resolution:

Change the declaration of the explicit bitset constructor in 23.2.1 [lib.template.bitset] as described above.

Requester: Andy Sawyer (andys@thone.demon.co.uk)
Owner:
Emails: (none)
Papers: (none)

---

Work Group: Library
Issue Number: 23-069
Title: Add pop_value() to container adapters
Sections: 23.2.4 [lib.container.adapters]
Status: Active
Description:

Due to time penalties, the STL container adaptor classes have no function that removes the next element AND returns it. Instead two different functions top() and pop() have to get called. As the normal interaction with stacks and queues is to

process the next element I suggest to introduce as add on a function pop_value() that does the job.

Proposed Resolution:

A proposal very similar to this was presented as issue 23-018 and discussed at Monterey. The LWG decided not to introduce any new pop() members. Therefore, in keeping with this earlier decision, close this issue with no changes to the WP. Refer to 23-018 for rationale.

Requester: Konrad Kiefer (kiefer@gecko.zfe.siemens.de)
Owner:
Emails: (none)
Papers: (none)

---

Work Group: Library
Issue Number: 23-070
Title: Clean up descriptions for capacity() and reserve()
Sections: 23.2.5.4 [lib.vector.capacity]
Status: Active
Description:

The current WP descriptions for capacity() and reserve() in vector are imprecise. Suggest changing them as follows:

Change the return value description for capacity() to the following:

Returns:
the number of elements in the vector for which the size of the allocated storage is enough for.

Change the last sentence of the description for reserve() as follows:

It is guaranteed that no reallocation during a insertion that happens after reserve() takes place until the time when the size of the vector becomes greater than the size specified by reserve().

Proposed Resolution:

Change the description for capacity() in 23.2.5.4 [lib.vector.capacity] to read as follows:

Returns: the number of elements that can be stored

in the vector without requiring reallocation.

Change the last sentence of the Notes section for reserve() in 23.2.5.4 [lib.vector.capacity] to read as follows:

It is guaranteed that no reallocation takes place during insertions that happen after reserve() takes place until the time when the size of the vector becomes greater than the size specified by reserve().

Requester: Konrad Kiefer (kiefer@gecko.zfe.siemens.de)
Owner:
Emails: (none)
Papers: (none)

---

Work Group: Library
Issue Number: 23-071
Title: Do adapters need allocator arguments?
Sections: 23.2.4 [lib.container.adapters]
Status: Active
Description:

Bjarne writes:

A container adapter, such as stack, can use its allocator template argument or it can extract its container argument's allocator. Has the question whether it needs both (or the allocator template parameter could be eliminated) been discussed?

Separately, Nathan Myers wrote:

>Judy Ward, Message c++std-lib-4575:
>
> I have an issue with the way the standard adaptor containers
> (stack, queue, priority_queue) are defined in the current
> standard. I'll use stack as an example:
>
> template <class T, class C=deque<T>, class A=allocator<T> > >
> class stack;
>
> If a user declared, for example:
>
> stack<int, deque<int>, myallocator<int> > s;
>
> Wouldn't they expect that myallocator is being used for

> allocation in the stack class? I don't think it would because
> stack uses deque and deque would be using the default
> allocator. In fact the results of the get_allocator() function
> would be misleading. Wouldn't they have to say:
>
> stack<int, deque<int, myallocator<int> >, myallocator<int> >;
> (This seems a little redundant, does anyone have a better idea?)

I agree. This was pointed out to me by somebody else yesterday, and I promised to write an issue for it. Luckily, Judy beat me to it. The correct way to do this, now that part of the requirements on Container is a typedef member allocator_type, is for the adaptor constructor to be declared in terms of that member:

```
template <class T, class Container = deque<T> >
class stack {
    / / ...
   typedef typename Container::allocator_type allocator_type;
   explicit stack(const allocator_type& = allocator_type());
/ / ...
};
```

> Also, how does the allocator that is passed in as a
> constructor argument for stack become the same constructor
> used by the container? If it doesn't what is the use of it?
> I don't think stack itself does any allocation.

stack<> etc. have member data containers, to which the allocator argument must be passed:

```
template <class T, class Container = deque<T> >
stack(const allocator_type& a = allocator_type()) : c_(a) {}
```

Proposed Resolution:

Change the declaration of queue in 23.2.4.1 [lib.queue] to read as follows (only changes shown):

```
   template <class T, class Container = deque<T> >
   class queue {
      ...
      typedef Container::allocator_type allocator_type;
         ...
         explicit queue(const allocator_type& = allocator_type());
      ...
   };

   template <class T, class Container>
```

```
bool operator==(const queue<T, Container>& x,
        const queue<T, Container>& y);

template <class T, class Container>
bool operator< (const queue<T, Container>& x,
        const queue<T, Container>& y);
```

Change the declaration of priority_queue in 23.2.4.2 [lib..priority.queue] to read as follows (only changes shown):

```
template <class T, class Container = vector<T>,
      class Compare = less<Container::value_type> >
class priority_queue {
   ...
   typedef Container::allocator_type allocator_type;
      ...
      explicit priority_queue(const Compare& x = Compare(),
        const allocator_type& = allocator_type());
   ...
};
```

Change the declaration of stack in 23.2.4.3 [lib.stack] to read as follows (only changes shown):

```
template <class T, class Container = deque<T> >
class queue {
   ...
   typedef Container::allocator_type allocator_type;
      ...
      explicit stack(const allocator_type& = allocator_type());
   ...
};

template <class T, class Container>
bool operator==(const stack<T, Container>& x,
        const stack<T, Container>& y);

template <class T, class Container>
bool operator< (const stack<T, Container>& x,
        const stack<T, Container>& y);
```

Requester: Bjarne Stroustrup (bs@research.att.com) et. al.
Owner:
Emails: c++std-lib-4575, c++std-lib-4577, c++std-lib-4677
Papers: (none)

---

Work Group: Library

Issue Number: 23-072
Title: Clean up vector<bool> declarations
Sections: 23.2.6 [lib.vector.bool]
Status: Active
Description:

1. Type "pointer" is missing from vector<bool>.

2. vector<bool>::assign is declared as:

```
template <class Size, Class T>
void assign(Size n, const T& t = T());
```

It should be:

```
template <class Size, Class T>
void assign(Size n, const bool& x = bool());
```

3. Should the vector<bool>::operator[] and vector<bool>::at functions return reference &, instead of reference ?

Proposed Resolution:

The issue with missing "pointer" typedefs for all the container types is addressed in 23-058.

Change the declaration of the two-argument version of vector<bool>::assign to read as follows:

```
template <class Size, class T>
void assign(Size n, const bool& x = bool());
```

Finally, the current specifications for operator[] and at() are correct; they should *not* be changed to return reference&.

Requester: Harold Seigel (seigel@decc.ENET.dec.com)
Owner:
Emails: (none)
Papers: (none)

---

Work Group: Library
Issue Number: 23-073
Title: Map / multimap::value_compare::operator() should be const
Sections: 23.3.1 [lib.map], 23.3.2 [lib.multimap]
Status: Active
Description:

Why is std::map::value_compare::operator() not a const member function? Same for multimap.

Proposed Resolution:

Change the definition of map::value_compare::operator() in 23.3.1 [lib.map] and of multimap::value_compare::operator() in 23.3.2 [lib.multimap] to be const.

Requester: Michael Klobe (mklobe@objectspace.com)
Owner:
Emails: (none)
Papers: (none)

---

Work Group: Library
Issue Number: 23-074
Title: Why no copy constructor or assignment for bitset?
Sections: 23.2.1 [lib.bitset]
Status: Active

Description:

Why is there not a templatized copy constructor for bitset? E.g.,

```
template<size_t N>
bitset(const bitset<N>& original);
```

Effect:

```
*this = bitset<N>(original.to_string());
```

It seems like a similar operator=() would be useful as well.

Also, I was surprised to find that bitset doesn't have container semantics. Iterators could be used with the copy algorithm and the assign() methods to move subranges of bits around, for example. If bitset was never intended to have container semantics, why put it in chapter 23? Why not chapter 26?

Proposed Resolution:

The omission of a bitset copy constructor and assignment operator appears to be a simple oversight. Add the following two signatures to the definition of bitset in 23.2.1 [lib.template.bitset]:

```
bitset(const bitset<N>& x);
bitset<N>& operator=(const bitset<N>& x);
```

Add the following text to 23.2.1.1 [lib.bitset.cons]:

bitset(const bitset<N>& x);

Effects:
Constructs an object of class bitset<N>, initializing each bit position to the corresponding bit values in x.

Add the following text to 23.2.1.2 [lib.bitset.members]:

bitset<N>& operator=(const bitset<N>& x);

Effects:
Sets each bit in *this to the corresponding bit value in x.

(The issue about where bitset should be placed in Clause 23 is addressed separately in issue 23-061.)

Requester: Michael Klobe (mklobe@objectspace.com)
Owner:
Emails: (none)
Papers: (none)

---

Work Group: Library
Issue Number: 23-075
Title: Add resize() to optional operations, fix description
Sections: 23.1 [lib.container.requirements],
23.2.2.4 [lib.deque.capacity],
23.2.3.4 [lib.list.capacity],
23.2.5.4 [lib.vector.capacity]
Status: Active
Description:

All three sequential containers exhibit a resize() member function, but resize() is not considered a required or an optional operation. Moreover, the definition of resize() given in the CD is incorrect. I think it should be:

```
void resize(size_t sz, T c = T())
{
  if (sz>size())
    insert(end(), size()-sz, c);
  else if (sz<size())
    erase(begin()+sz, end());
```

}

Proposed Resolution:

Add an entry for resize() to Table 77 (Sequence Requirements), in 23.1.1 [lib.sequence.reqmts] as follows:

| | | |
|---|---|---|
| a.resize(n,t) | void | post: size() == n.<br>if (n > a.size())<br>  insert(a.end(),n-a.size(),t);<br>else if (n < a.size())<br>  erase(a.begin()+n,a.end()); |

If this recommendation is adopted, then the current definitions of the resize() semantics in 23.2.2.4, 23.2.3.4 and 23.2.5.4 can be removed in favor of the equivalent definition added to the table, as shown above.

However, if resize() is not added to Table 77:

As for the correctness of the resize() definition, a resolution to correct the definition was written up as issue 23-033 and passed at the Tokyo meeting. However, a minor typo remains in the current WP. The line

```
erase(begin()+sz, s.end());
```

Should be changed to

```
erase(begin().sz, end());
```

in all three sections. I.e., remove the "s." from the second argument. (This is an editorial change.)

Requester: Graziano Lo Russo (via Andy Koenig)
Owner:
Emails: c++std-lib-4500
Papers: (none)

---

Work Group: Library
Issue Number: 23-076
Title: Fix reverse_iterator typedefs in deque and vector
Sections: 23.2.2 [lib.deque], 23.2.5 [lib.vector], 23.2.6 [lib.vector.bool]
Status: Active
Description:

Although only the EDG compiler currently gives a complaint, this is invalid C++ according to John Spicer:

```
template <class T>
class reverse_iterator {};

class vector {
    typedef reverse_iterator<...> reverse_iterator;
    typedef reverse_iterator<...>  const_reverse_iterator;
        / / this reverse iterator refers to the member
        / / reverse_iterator declared above
};
```

Proposed Resolution:

Change the reverse_iterator typedefs for deque (23.2.2), vector (23.2.5) and vector<bool> (23.2.6) to:

```
typedef std::reverse_iterator<...> reverse_iterator;
typedef std::reverse_iterator<...> const_reverse_iterator;
```

where the "..." is as before. In other words, simply add the "std::" to each typedef.

Requester: Judy Ward (j_ward@zko.dec.com)
Owner:
Emails: (none)
Papers: (none)

---

Work Group: Library
Issue Number: 23-077
Title: Why doesn't queue have a top() member function?
Sections: 23.2.4.1 [lib.queue]
Status: Active

Description:

Priority_queue has a top() member functions, why doesn't queue? Wouldn't it be strange if you wrote some code for priority_queue which wouldn't work with a queue?

Proposed Resolution:

Add the following declarations to the declaration of class queue in 23.2.4.1 [lib.queue]:

```
value_type& top() { return c.front(); }
const value_type& top() const { return c.front(); }
```

Requester: Judy Ward (j_ward@zko.dec.com)
Owner:
Emails: (none)
Papers: (none)

---

Work Group: Library
Issue Number: 23-078
Title: Naming: difference_type vs. distance type
Sections: 23.1 [lib.container.requirements]
Status: Active
Description:

Table 75 (Container Requirements) in the May Draft states that a container must define

| | | |
|---|---|---|
| X::difference_type | signed integral type | is identical to the distance type of X::iterator and X::const_iterator |

Is there a reason why these identical types are called difference_type in one place and distance_type in the other?

Proposed Resolution:

This should be discussed and resolved by the LWG. Although not a major issue, the different names are indeed confusing and there does not appear to be any compelling reason to name them differently. A single name should be chosen; either "difference_type" or "distance_type" is satisfactory.

Requester: Dave Dodgson (dsd@tr.unisys.com)
Owner:
Emails: (none)
Papers: (none)

---

Work Group: Library
Issue Number: 23-079
Title: insert(p,t) should not have default argument
Sections: 23.1.1 [lib.sequence.reqmts], 23.2.2 [lib.deque], 23.2.3 [lib.list], 23.2.5 [lib.vector], 23.2.6 [lib.vector.bool]
Status: Active
Description:

(Note: the following comment was originally raised w.r.t. Clause 21, but since it involves STL it has been moved

here.)

While it is a minor aside to the above, I note that I consider the function signature

insert(iterator position, const T& x = T())

incorrect and silly. The silliness comes from being able to write

insert(it);

which specifies where something is to be inserted, but not what (that defaults). I note that this is consistent with similar function signatures in the STL, but I think they are wrong also. An insertion should always have to specify what is being inserted.

If the STL functions would also accept the recommendation, then I would suggest that the default parameter be removed. If the STL is not changed(and I do not expect that it will be), then I would recommend that basic_string<> be kept consistent (even if it is silly).

Proposed Resolution:

Close this issue with no change to the WP. Although the default argument is arguably more confusing than useful, the current signature has been in STL from the beginning and is not clearly "broken".

Requester: Jack Reeves
Owner:
Emails: (none)
Papers: (none)

---

Work Group: Library
Issue Number: 23-080
Title: Resolve map::mapped_type vs. map::referent_type
Sections: 23.3.1 [lib.map], 23.3.2 [lib.multimap]
Status: Active
Description:

The resolution to issue 23-040 included adding the typedefs

typedef T mapped_type

to both map and multimap. However, apparently paper N0845 (accepted at the Santa Cruz meeting) contained a provision to add a typedef "referent_type" defined the same way.

This duplication is noted in Boxes 76 and 77 of the May 1996 revision of the WP.

Proposed Resolution:

Reject the addition of referent_type specified in N0845 in favor of the existing typedefs (mapped_type). Remove boxes 76 and 77 from the WP.

Requester: Larry Podmolik (podmolik@str.com)
Owner:
Emails: (none)
Papers: (none)