Doc No: J16/97-0075 = WG21/N1113
Date: 30 September, 1997
Project: Programming Language C++
Ref Doc:
Reply to: Nico Josuttis
(nico@bredex.de)

Swapping of containers

The standard library lacks some specification about how a swap() works for containers.

Actual situation

The definitions in the draft actually say the following:

- Any container member function a.swap(b) has the post-condition swap(a,b) (see 23.1).
- The global algorithm swap() "exchanges values stored in two locations" (25.2.2).
- For every container it is specified that a specialized global algorithms swap(x,y) exist, which effects s.swap(y) (see e.g. 23.2.4.4).
- For containers swap() should have constant complexity (23.1).
- Regarding exceptions 23.1 states the following:

  > All container types defined in this clause meet the additional requirements that no swap() function throws an exception unless that exception is thrown by the copy constructor or assignment operator of the container's Compare object (if any, see 23.1.2).
- For basic_string the following is defined:

  > References, pointers, and iterators referring to the elements of a basic_string sequence may be invalidated by the following uses of that basic_string object:
  > - As an argument to non-member functions swap() (21.3.7.8)...
  > - As an argument to basic_string::swap().

Defects

The current specifications of swap() have the following problems:

- There is no general statement about what happens with references, pointers, and iterators.
  Following the idea of the specialized swap()s it seems to make sense that they are swapped accordingly and thus stay valid.
  As (except for string) there is no statement about that, it implicitly means that swap() currently invalidates references, pointers, and iterators.
  Changing that would enable significant advantages as people could swap containers without loosing references to individual elements.
- Another related aspect is the question, whether a vector might reallocate due to swap().
  Actually there is no statement about that, what means that reallocation might occur.

If we specify to swap iterators, pointers and references accordingly, then it may follow that the memory (and the capacity) is also swapped. However some wording could clarify that.

Proposal

I suggest the following change:

In 23.1 Container requirements [lib.container.requirements] replace the last paragraph (which actually has no number but should probably have paragraph number 10):

```
OLD> All container types defined in this clause meet the
OLD> additional requirements that no swap() function throws
OLD> an exception unless that exception is thrown by the
OLD> copy constructor or assignment operator of the
OLD> container's Compare object (if any, see 23.1.2).
```

with the following:

> All container types defined in this clause meet the following additional requirements for swap() functions:
>
> - No swap() function throws an exception unless that exception is thrown by the copy constructor or assignment operator of the container's Compare object (if any, see 23.1.2).
> - No swap() function invalidates any references, pointers, or iterators referring to the elements of the containers being swapped. All references, pointers and iterators to the elements of a swapped container afterwards refer to the same elements as on entry what means that they swap the containers they refer to accordingly.
> - No swap() function does reallocate memory for the elements of a container. The memory used by the two swapped containers, and their capacities (if any, see 23.2.4.2) are swapped.

Special proposal for basic_string

The proposal above has no change in basic_string as right now some special words exist there yet.
But I wonder if they are correct.
Right now swapping a string means that all references, pointers, or iterators became invalid.
For the same reasons as above it would be nice to introduce the possibility to swap references, iterators, and pointers accordingly.
But I am not sure if any problem might occur.

So as supplementary proposal I suggest:

In 21.3 Template class basic_string [lib.basic.string] replace the paragraph 5:

```
OLD> References, pointers, and iterators referring
OLD> to the elements of a basic_string sequence may
OLD> be invalidated by the following uses of that
OLD> basic_string object:
OLD>  - As an argument to non-member functions swap()
OLD>    (21.3.7.8), operator>>() (21.3.7.9), and
OLD>    getline() (21.3.7.9).
OLD>  -  As an argument to basic_string::swap().
OLD>  -  Calling data() and c_str() member functions.
OLD>  -  Calling non-const member functions, except
OLD>     operator[](), at(), begin(), rbegin(), end(), and rend().
OLD>  -  Subsequent to any of the above uses except the forms
OLD>     of insert() and erase() which return iterators,
OLD>     the first call to non-const member functions
OLD>     operator[](), at(), begin(), rbegin(), end(), and rend().
```

with the following:

```
       > References, pointers, and iterators referring
       > to the elements of a basic_string sequence may
       > be invalidated by the following uses of that
       > basic_string object:
    NEW>  - As an argument to non-member functions
    NEW>    operator>>() (21.3.7.9), and getline() (21.3.7.9).
    DEL>
       >  -  Calling data() and c_str() member functions.
       >  -  Calling non-const member functions, except
       >     operator[](), at(), begin(), rbegin(), end(), and rend().
       >  -  Subsequent to any of the above uses except the forms
       >     of insert() and erase() which return iterators,
       >     the first call to non-const member functions
       >     operator[](), at(), begin(), rbegin(), end(), and rend().
    NEW> No swap() function invalidates any references,
    NEW> pointers, or iterators referring to the elements
    NEW> of the string being swapped.
    NEW> All references, pointers and iterators to the
    NEW> elements of a swapped string afterwards refer
    NEW> to the same elements as on entry what means that
    NEW> they swap the string they refer to accordingly.
```