Doc. No.: X3J16 / 95-0028
WG21 / N0628
Date: January 30, 1995
Project: Programming Language C++
Reply To: Richard K. Wilhelm
Andersen Consulting
rkw@chi.andersen.com

Refinements to basic_string

1. Introduction

This proposal seeks to increase the compatibility of basic_string with the style of STL. It proposes changes required to make basic_string an STL-compliant Reversible Sequence.

Many of the changes contained in this paper were first suggested in 94-0145=N0532 (post-Waterloo mailing). However, subsequent implementation experience lead us to believe it would be impossible to maintain STL Sequence compliance while permitting a reference-counted implementation. Consequently, in 94-0170=N0557 (pre-Valley Forge mailing) I proposed a more conservative approach to revising basic_string. That proposal sought to eliminate gratuitous incompatibilities with STL and provide interfaces as consistent as possible with STL conventions, without changing the essentials of basic_string.

During discussions at the Waterloo meeting, Andrew Koenig suggested an implementation which would allow an STL-compliant interface for basic_string while permitting a reference-counted implementation. Essentially, the strategy consisted of marking a string representation as unsharable when a non-const iterator is created.

After this insight, the members of the Library Working Group voiced support for making basic_string a fully compliant STL Sequence.

2. Issues and Proposed Resolutions

2.1 Typedefs

2.1.1 Issue: Reversible Sequence requires typedefs for `iterator`, `const_iterator`, `reverse_iterator`, and `const_reverse_iterator`.

Proposed Resolution: add these typedefs to basic_string, as follows:

```
typedef allocator::iterator iterator;
typedef allocator::const_iterator const_iterator;
typedef allocator::reverse_iterator reverse_iterator;
typedef allocator::const_reverse_iterator const_reverse_iterator;
```

The type `iterator` is a random access iterator referring to `T`. The exact type is implementation dependent and determined by `Allocator`. The type `const_iterator` is a constant random access iterator referring to `const T`. The exact type is implementation dependent and determined by `Allocator`. It is guaranteed that there is a constructor for `const_iterator` out of `iterator`.

Rationale: These typedefs provide users with access to the information provided by the allocator. Additionally, these typedefs increase code portability. They also provide consistency with the rest of the library. Since the iterator type of basic string belongs to the random access iterator category, the container is called *reversible* and must provide reverse iterators.

2.2 Constructors

2.2.1 Issue: The copy constructor

```
basic_string(const basic_string& str, size_type pos = 0,
             size_type n = NPOS);
```

contains three arguments for additional flexibility in copying strings. Sequence copy constructors have one argument.

Proposed Resolution: eliminate the additional arguments. The copy constructor should the interface:

```
basic_string(const basic_string& str);
```

Rationale: STL provides an alternative with equivalent functionality. The flexibility provided by the additional arguments will be available through constructors which use iterators (2.2.2).

2.2.2 Issue: basic_string lacks a constructor required for Sequence compliance:

```
template <class InputIterator>
basic_string(InputIterator begin, InputIterator end);
```

Proposed Resolution: add this constructor with a semantics specified for Sequence containers.

Rationale: The new constructor provides functionality currently contained in basic_string for building a string from a portion of an existing string. It also conforms with STL sequence requirements for constructors using iterators.

2.3 Iterators

2.3.1 Issue: basic_string does not have the member functions `begin()` and `end()` as required by STL Sequences.

Proposed Resolution: Add the member functions:

```
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;

reverse_iterator rbegin();
const reverse_iterator rbegin() const;
reverse_iterator rend();
const reverse_iterator rend() const;
```

as specified by STL requirements. `begin()` returns an iterator referring to the first character in the string. `end()` returns an iterator which is the past-the-end value. `rbegin()` returns an iterator which is semantically equivalent to `reverse_iterator(end())`. `rend()` returns an iterator which is semantically equivalent to `reverse_iterator(begin())`.

Rationale: Iterators returned by basic_string allow it to fit into the STL framework. Since basic_string is a reversible container, it must provide reverse iterators.

2.4 Accessors

2.4.1 Issue: basic_string provides a member for building a substring from a string:

```
basic_string substr(size_type pos, size_type n);
```

which is not provided by any STL class.

Proposed Resolution: remove the `substr()` member from basic_string.

Rationale: The functionality of `substr()` can be efficiently duplicated within the STL framework. In general, iterators provide a means of accessing a portion of a string. The functionality of creating a substring is not required due to the resolution of issue 2.5.3 below. For example, the code:

```
basic_string<char> s, t;
int start, len
// ...
s = t.substr(start, len);
```

can be replaced with:

```
basic_string<char> s, t;
// ...
s.assign(t.begin() + start, t.begin() + start + len);
```

See below for modifications to the `assign()` members which accept iterator parameters.

2.5 Mutators

2.5.1 Issue: The basic_string member functions for insert lack consistency with the required STL Sequence insert member functions.

The basic_string members are:

```
basic_string<T>& insert(size_type pos, const basic_string<T>& str,
                        size_type pos2 = 0, size_type n = NPOS);
basic_string<T>& insert(size_type pos, const_pointer s,
                        size_type n);
basic_string<T>& insert(size_type pos, const_pointer s);
basic_string<T>& insert(size_type pos, size_type n, T c = T());
```

STL sequence members are:

```
iterator insert(iterator position, const T c = T());
template <class InputIterator>
insert(iterator position, InputIterator first, InputIterator last);
insert(iterator position, size_type n, T c = T());
```

Proposed Resolution: Change the interface of the member:

```
basic_string<T>& insert(size_type pos, const basic_string<T>& str,
                        size_type pos2 = 0, size_type n = NPOS);
```

to the following:

```
basic_string<T>& insert(iterator p, const basic_string<T>& str);
```

Change the interface of the members:

```
basic_string<T>& insert(size_type pos, const_pointer s,
                        size_type n);
basic_string<T>& insert(size_type pos, const_pointer s);
basic_string<T>& insert(size_type pos, size_type n, T c = T());
```

to the following:

```
basic_string<T>& insert(iterator p, const_pointer s, size_type n);
basic_string<T>& insert(iterator p, const_pointer s);
basic_string<T>& insert(iterator p, size_type n, T c = T());
```

And add the following members:

```
iterator insert(iterator p, const T c = T());
template <class InputIterator>
basic_string<T>&
```

```
    insert(iterator p, InputIterator first, InputIterator last);
```

Rationale: Satisfies STL Sequence requirements. The revised interface provides equivalent functionality to that currently present in basic_string using iterators instead of position-based indexes.

Note: Unlike, the insert members defined by STL, the proposed member `insert(iterator, InputIterator, InputIterator)` returns a reference the current object. The return value is retained as being part of existing practice.

2.5.2 Issue: The append members:

```
    basic_string<T>& append(const basic_string<T>& str,
                            size_type pos = 0, size_type n = NPOS);
    basic_string<T>& append(const_pointer, size_type n);
    basic_string<T>& append(const_pointer);
    basic_string<T>& append(size_type pos, size_type n, T c = T());
```

do not appear in sequences. In STL, the append operation is done by using `insert()` and inserting at the end of a sequence.

Proposed Resolution: Retain the append() members which use `const_pointer` objects and characters. Remove the member:

```
    basic_string<T>& append(const basic_string<T>& str,
                                size_type pos = 0, size_type n = NPOS);
```

And replace it with the following members:

```
    basic_string<T>& append(const basic_string<T>& str);

    template <class InputIterator>
    basic_string<T>& append(InputIterator first, InputIterator last);
```

Rationale: consistency with STL and proposed change to the constructor and other mutators. Removal would break with existing practice.

Note: The following member to append characters to a string is already part of the class interface and will not be changed by this proposal:

```
    basic_string<T>& append(size_type pos, size_type n, T c = T());
```

2.5.3 Issue: Interfaces to `assign()` member functions do not conform to those on other sequences:

```
    basic_string<T>& assign(const basic_string& str,
                                size_type pos = 0, size_type n = NPOS);
    basic_string<T>& assign(const_pointer, size_type n);
    basic_string<T>& assign(const_pointer);
    basic_string<T>& assign(size_type pos, size_type n, T c = T());
```

Proposed Resolution: Retain the `assign()` members which use `const_pointer` objects and characters. Remove the member:

```
    basic_string<T>& assign(const basic_string& str,
                                size_type pos = 0, size_type n = NPOS);
```

And replace it with the following members:

```
    basic_string<T>& assign(const basic_string<T>& str);

    template <class InputIterator>
    basic_string<T>& assign(InputIterator first, InputIterator last);
```

Rationale: consistency with STL and proposed change to the constructor and other mutators. Removal would break with existing practice.

2.5.4 Issue: Interface to character assign member has incorrect interface:

```
    basic_string<T>& assign(size_type pos, size_type n, T c = T());
```

Proposed Resolution: Change the interface as follows:

```
basic_string<T>& assign(size_type n, T c = T());
```

Rationale: The interface to this member was incorrectly specified in N0557/94-0170. This change might be considered editorial.

2.5.5 Issue: The basic_string replace() member functions do not appear in STL sequences and the interfaces do not conform to STL conventions. The members are:

```
basic_string<T>&
replace(size_type pos1, size_type n1, const basic_string& str,
        size_type pos2 = 0, size_type n2 = NPOS);

basic_string<T>&
replace(size_type pos, size_type n1, const_pointer s, size_type n2);

basic_string<T>&
replace(size_type pos, size_type n1, const_pointer s);

basic_string<T>&
replace(size_type pos, size_type n1, size_type n2, T c = T());
```

Proposed Resolution: Retain this functionality and modify their interfaces as follows:

Replace:

```
basic_string<T>&
replace(size_type pos1, size_type n1, const basic_string& str,
       size_type pos2 = 0, size_type n2 = NPOS);
```

with:

```
basic_string<T>&
replace(iterator i1, iterator i2, const basic_string& str);

basic_string<T>&
replace(iterator i1, iterator I2,
        const_iterator j1, const_iterator j2);
```

Replace:

```
basic_string<T>&
replace(size_type pos, size_type n1, const_pointer s, size_type n2);

basic_string<T>&
replace(size_type pos, size_type n1, const_pointer s);
```

with:

```
basic_string<T>&
replace(iterator i1, iterator i2, const_pointer s, size_type n2);

basic_string<T>&
replace(iterator i1, iterator i2, const_pointer s);
```

The iterators `i1` and `i2` are valid iterators on the string. They represent the range that will be replaced by the values between the iterators `j1` and `j2`. After the call, the length of the string will be changed by: `j2 -j1- (i2 -i1)`.

Replace:

```
basic_string<T>&
replace(size_type pos, size_type n1, size_type n, T c = T());
```

with:

```
basic_string<T>&
replace(iterator i1, iterator i2, size_type n, T c = T());
```

The iterators `i1` and `i2` are valid iterators on `this`. They represent the range that will be replaced by `n` copies of the object `c`. After the call, the length of the string will be changed by: `n - (i2 - i1)`.

Rationale: The `replace()` members are part of existing practice. The substring replacement functionality provided by the replace member functions is not provided by STL with equivalent efficiency. The revised interfaces conform to STL conventions.

2.5.6 Issue: The interface to the `remove()` members does not conform to STL conventions.

The basic_string member is:

```
   basic_string<T>& remove(size_type pos = 0, size_type n = NPOS);
```

The sequence members are:

```
   void remove(iterator position);
   void remove(iterator first, iterator last);
```

Proposed Resolution: Replace the member

```
   basic_string<T>& remove(size_type pos = 0, size_type n = NPOS);
```

with the members:

```
   basic_string<T>& remove(iterator position);
   basic_string<T>& remove(iterator first, iterator last);
```

Rationale: Conformance to STL sequences. The return values are retained for conformance to existing practice

Note: The STL members used be called “erase”. The name was changed with the acceptance of 94-0155=N0542 in Valley Forge.

2.6 Algorithms

Most STL containers have few algorithms. However, these string algorithms are retained because they are a part of existing practice. Additionally, the presence of these algorithms in basic_string reduces its reliance on the STL algorithms for common functionality. This is especially desirable because it reduces the learning burden for new programmers who might make heavy use of strings before learning the STL.

2.6.1 Issue: basic_string provides functions for string comparison. STL sequences provide no members for comparison. The basic_string members are:

```
   int compare(const basic_string& str, size_type pos = 0,
               size_type n = NPOS) const;
   int compare(const_pointer s, size_type pos, size_type n) const;
   int compare(const_pointer s, size_type pos = 0);
```

The interfaces are not consistent with other interfaces in basic_string nor with STL.

Resolution: Retain the functionality in basic_string. Change the interfaces as follows:

```
   int compare(const basic_string& str);
   int compare(const basic_string& str, const_iterator first1) const;
   int compare(const basic_string& str, const_iterator first1,
               size_type n) const;
   int compare(const_pointer cp, const_iterator first1,
               size_type n) const;
   // Returns:   compare(basic_string(cp, cp + n), first1).

   int compare(const_pointer cp);
   // Returns:  compare(basic_string(first2)). Uses traits::length().

   int compare(const_pointer first2, const_iterator first1);
   // Returns: compare(basic_string(first2), first1).
   // Uses traits::length().
```

In these prototypes, `first1` are valid `const_iterator` objects on `this`. The parameters `first2` and `last2` are valid `const_iterator` objects on the string compared against.

Rationale: The `compare()` members represent a gradual progression from simple to complex. The specified interface retains and expands the considerable flexibility of the current `compare()` members while maintaining appropriate consistency with STL conventions. The inability to use default arguments causes the increase in the number of comparison members.

2.6.2 Issue: The interfaces to the basic_string searching algorithms do not conform to STL conventions in two respects. First, basic_string members return an index, the relevant STL algorithms return an iterator. Second, basic_string members take index parameters, the relevant STL algorithms take iterator parameters. The interfaces are:

```
size_type F(const basic_string& str, size_type pos = 0) const;
size_type F(const_pointer s, size_type pos, size_type n) const;
size_type F(const_pointer s, size_type pos = 0) const;
size_type F(T c, size_type pos) const;
```

where F is the name of one of the four functions: `find()`, `rfind()`, `find_first_of()`, `find_last_of()`, `find_first_not_of()`, and `find_last_not_of()`.

Proposed Resolution: Modify the interfaces as follows:

Replace the interface:

```
size_type F(const basic_string& str, size_type pos = 0) const;
```

with the interfaces:

```
// first1 is an iterator on 'this'
const_iterator F(const basic_string& str) const;
const_iterator F(const basic_string& str, const_iterator first1);
```

Replace the interfaces:

```
size_type F(const_pointer s, size_type pos, size_type n) const;
size_type F(const_pointer s, size_type pos = 0) const;
```

with the interfaces:

```
const_iterator F(const_pointer s, const_iterator first1,
                 size_type n) const;
const_iterator F(const_pointer s) const;
const_iterator F(const_pointer s, const_iterator first1) const;
```

Replace the interface:

```
size_type F(T c, size_type pos) const;
```

with the interfaces:

```
const_iterator F(T c) const;
const_iterator F(T c, const_iterator first1) const;
```

Rationale: These interfaces more closely map to the iterator-based style without substantially changing those which currently exist in basic_string. The increase in the number of members is due to the lack of default arguments.

Comment: An alternative which was considered was to reorder the arguments to some of the functions, providing the iterator on `this` as the first parameter. The approach had the advantages of keeping the number of interfaces at four and being more in the style of STL algorithms. However, since these algorithms are being provided to give compatibility with existing practice, I think a better case can be made for the interfaces which map closely to those currently in basic_string. Since all functionality can be accomplished through the STL algorithms, those wanting to code in the STL style can work exclusively in that style by avoiding the algorithms in basic_string. Additionally, the more compact interface would have been less convenient for the user in common cases, forcing the use of extra arguments.

2.6.3 Issue: The names of the `find()` and `rfind()` algorithms that search for substrings do not correspond with STL algorithms of equivalent functionality. Bundled under the name of `find()` is the functionality of the STL algorithms `find()` and `search()`. In

STL, the function `find()` seeks a matching element of a sequence. The function `search()` seeks a subsequence of equal values in a sequence.

Proposed resolution: (given the interfaces specified in 2.6.2 above) replace the basic_string members:

```
   const_iterator find(const basic_string& str) const;
   const_iterator find(const basic_string& str, const_iterator first1);
   const_iterator find(const_pointer s, const_iterator first1,
                       size_type n) const;
   const_iterator find(const_pointer s) const;
   const_iterator find(const_pointer s, const_iterator first1) const;

   const_iterator rfind(const basic_string& str) const;
   const_iterator rfind(const basic_string& str,
                       const_iterator first1);
   const_iterator rfind(const_pointer s, const_iterator first1,
                        size_type n) const;
   const_iterator rfind(const_pointer s) const;
   const_iterator rfind(const_pointer s, const_iterator first1) const;
```

with the members:

```
   const_iterator search(const basic_string& str) const;
   const_iterator search(const basic_string& str,
                         const_iterator first1);
   const_iterator search(const_pointer s, const_iterator first1,
                         size_type n) const;
   const_iterator search(const_pointer s) const;
   const_iterator search(const_pointer s, const_iterator first1) const;

   const_iterator rsearch(const basic_string& str) const;
   const_iterator rsearch(const basic_string& str,
                          const_iterator first1);
   const_iterator rsearch(const_pointer s, const_iterator first1,
                          size_type n) const;
   const_iterator rsearch(const_pointer s) const;
   const_iterator rsearch(const_pointer s,
                          const_iterator first1) const;
```

Rationale: The STL-based names provide consistency with the rest of the library. Internal consistency makes the library easier to understand for new users.

3. Relevant Documents

B. Dawes, “Small Library Changes”, X3J16/94-0155=WG21/N0542.
A. Koenig (ed.), "Working Paper for Draft Proposed International Standard for Information Systems -- Programming Language C++", X3J16/94-0158=WG21/N0545.
R. Wilhelm, “Integrating basic_string with STL”, X3J16/94-0145=WG21/N0532.
R. Wilhelm, “Integrating basic_string with STL (revised)”, X3J16/94-0170=WG21/N0557.
R. Wilhelm, “Minor Library Modifications”, X3J16/94-0171=WG21/N0558.