Doc. No.:	X3J16 / 95-0081 WG21 / N0681
Date:	March 8, 1995
Project:	Programming Language C++
Reply To:	Richard K. Wilhelm Andersen Consulting rkw@chi.andersen.com

Refinements to basic_string (Revised)

0. Introduction to Revision[1]

At the Valley Forge meeting, we modified the basic_string template to make it partially STL compatible. In the interest of minimizing impact on the string class, the proposal was somewhat conservative in its approach: it changed a few member function names, reordered some arguments, and added an Allocator template parameter.

During discussions in Valley Forge, a small group decided that basic_string could be made more STL-compatible without losing the functionality provided by the previous basic_string interface. The case for compatibility was successfully made to the LWG and subsequently to the full committee. The previous version of this paper ("Refinements to basic_string" 95-0028=N0628) proposed changes which would make basic_string more STL-compatible.

However, during (very) recent discussions with Uwe Steinmuller and other members of the LWG, it became apparent that 95-0028 contained an approach that was quite different than expected. This is due to an honest difference in interpretation of the LWG guidance. The goal of "equivalent functionality" was taken to mean *functional equivalence* and not *interface compatibility*. That is, the functionality of searching, comparing, and appending (etc.) was retained in basic_string. But it was given a very different interface.

In this revised proposal, the basic_string template retains its existing position-based interface. This proposal would only add additional members to make basic_string STL-compatible; it is a strict extension. It would also add overloads of some functions to provide an "STL-style" interface on them. The goal of this revised proposal is to give the basic_string template two interfaces: one for those who approach it from the position-based viewpoint of commonly available C++ strings, and one for those who approach it from the STL viewpoint.

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-

[1] This document was distributed at the Austin meeting with the temporary number of N0628R1 / 95-0028R1.

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: `basic_string` lacks an iterator-based constructor required for Sequence compliance.

Proposed Resolution: In 21.1.1.4.1 [lib.string.cons], add the constructor:

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

with semantics specified for Sequences in 23.1.1 [lib.sequence.reqmts], Table 50.

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;
```

The members have semantics as specified by STL requirements stated in 23.1 [lib.container.requirements], Table 49. 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 Mutators

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

Proposed Resolution: In 21.1.1.4.6 [lib.string::insert], add the following members:

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

with semantics specified for Sequences in 23.1.1 [lib.sequence.reqmts], Table 50.

Rationale: Satisfies STL Sequence requirements

2.4.2 Issue: The append() members do not provide an iterator-based interface. In STL, the append operation is done by using insert() and inserting at the end of a sequence.

Proposed Resolution: In 21.1.1.4.4 [lib.string::append], add the following member:

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

This function returns:

```
append(basic_string<charT,Allocator,traits>(first, last));
```

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

2.4.3 Issue: Interfaces to assign() member functions do not provide an iterator-based interface:

Proposed Resolution: In 21.1.1.4.5 [lib.string::assign], add the following member:

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

This function returns:

```
assign(basic_string<charT,Allocator,traits>(first, last));
```

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

2.4.4 Issue: Interfaces to character append() member and assign() members have incorrect interfaces:

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

Proposed Resolution: In 21.1.1.4.4 [lib.sring::append] and 21.1.4.5 [lib.string::assign], change the interfaces as follows:

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

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

Rationale: The current interfaces do not make sense. These interfaces accidentally changed during recent revisions to the WP.

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

Proposed Resolution: In 21.1.1.4.8 [lib.string::replace], add the following members:

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

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

```
basic_string<T>&
replace(iterator i1, iterator i2, const charT* s, size_type n);
// length change: n - (i2 - i1)
basic_string<T>&
replace(iterator i1, iterator i2, const charT* s);
// length change: traits.length(s) - (i2 - I1)
// uses traits::length()
basic_string<T>&
replace(iterator i1, iterator i2, size_type n, T c = T());
// length change: n - (i2 - i1)
template <class InputIterator>
basic_string<T>&
replace(iterator i1, iterator i2,
     InputIterator j1, InputIterator j2);
// length change: j2 - j1 - (i2 - i1)
```

In the above functions, the iterators `i1` and `i2` are valid iterators on the string. They represent the range of characters that will be replaced After the call, the length of the string will be change as indicated in the comment.

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.

2.4.6 Issue: The interfaces to the `remove()` members do not conform to STL conventions.

Proposed Resolution: In 21.1.4.7 [lib.string::remove], add the following members:

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

with semantics specified for Sequences in 23.1.1 [lib.sequence.reqmts], Table 50.

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.

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 / 95-0029=WG21 / N0629.

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.

R. Wilhelm, “Refinements to basic_string”, X3J16 / 95-0028-WG21 / N0628.