Doc. No.: X3J16/95-0159
WG21/ N0759
Date: July 28, 1995
Project: Programming Language C++
Reply To: Richard K. Wilhelm
Andersen Consulting
rkw@chi.andersen.com

Clause 21 (Strings Library) Issues List
Revision 7

Revision History

Version 1 - January 30, 1995: Distributed in pre-Austin mailing.
Version 2 - March 6, 1995: Distributed at Austin meeting.
Version 3 - March 24, 1995: Distributed in post-Austin mailing. Several issues added. Several issues updated to reflect decisions at Austin meeting.
Version 4 - May 19, 1995: Distributed in pre-Monetery mailing.
Version 5 - July 9, 1995: Distributed at the Monterey meeting. Includes many issues added from public comments.
Version 6 - July 11, 1995: Distributed at the Monterey meeting. Added no new issues from previous version. Included issues prepared for formal vote. Added solutions for issues 8, 21,31, 38, 69, 71. Made only changes to reflect the decisions of the string sub-group, correct working paper text and to correct typographical errors.
Version 7 - July 27, 1995: Distributed in the post-Monterey mailing. Reflects the resolutions and discussions of the Monterey meeting.

Introduction

This document is a summary of the issues identified in Clause 21. 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 record for Strings issues and as a foundation of proposals for resolving specific issues.

For clarity, active issues are seperated from issues recently closed. Closed issues are retained for one revision of the paper to serve as a record of recent resolutions. Subsequenly, they will be removed from the paper for brevity. Any issue which has been removed will include the document number of the final paper in which it was included.

Active Issues

Issue Number: 21-002
Title: Are string_traits members char_in() and char_out() necessary?
Section: 21.1.1.2 [lib.string.char.traits]
Status: active
Description:

In lib-3398, Nathan Myers writes:

Looking at Clause 21, Strings, I find some string_traits static members:

```
static basic_istream<charT>
  string_char_traits::char_in(basic_istream<charT>& is,
                              charT& a)
{ return is >> a; }
```

```
static basic_istream<charT>
  string_char_traits::char_out(basic_ostream<charT>& os,
                               charT& a)
{ return os << a; }
```

Are they necessary? If so, shouldn't they be parameterized on ios_traits? And shouldn't they default to use streambuf put() and get()?

[Note: lib-3398 contained a typo in which `char_in()` and `char_out()` were incorrectly specified as being members of basic_string. The slight error is corrected above.]

See issue 21-008 for additional comments on this subject.

Proposed Resolution:

Remove the members `string_char_traits::char_in()` and `string_char_traits::char_out()`.

Requester: Nathan Myers: myersn@roguewave.com
Owner:
Emails: lib-3398
Papers: (none)

Issue Number: 21-012
Title: Why are character parameters to the string functions passed by value?
Section: 21.1.1.2 [lib.string.char.traits]
Status: active
Description:

In the string functions, character parameters are specified as being passed by "charT". In the past, the LWG had decided that char-like types should be considered cheap enough to pass by value.

However during discussions at the Austin meeting, the LWG formed the consensus that characters should be passed by reference. The rationale was: for most character types, on most architectures, it was as efficient for characters to be passed by references instead of by value. The importance of reference parameters arrived in atypical character types which might be considerably larger than ASCII characters

Proposed Resolution:

All character parameters to all string functions will be passed by const reference.

Requester: Rick Wilhelm: rkw@chi.andersen.com
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-013
Title: There is no provision for errors caused by implementation limits.
Section: 21.1.1.2 [lib.basic.string]
Status: active
Description:

In private email, John Dlugosz wrote:
"There is no provision for errors caused by implementation limits. The class handles strings up to length NPOS-1, with no specified way to throw an error saying "I can't do that!" for shorter values. In my implementation I'm simulating an out-of-memory error if an operation exceeds a `maxcount' length, since that's what would presumably happen anyway. The maxcount arises due to arithmetic overflow: I'm limited to size_t-(small constant) _bytes_, not

elements, and an element may be any size. I can't compute the memory requirments without getting an unreported arithmetic overflow, so I have to check in advance for this instantiation-specific maxcount.

"In order to simulate the out of memory condition, I just call `new' on NPOS bytes. That way I get the "correct" behavior for any installed new_handler or replacement operator new() that may exist. However, that is not the best solution for a few reasons. First, it will fail if the implementation _does_ in fact allocate NPOS bytes without error. Second, an out-of-memory exception might not be the appropriate way for a program to recover from this problem. Third, it is less efficient, since by spec I must test for an argument of NPOS anyway, and take one action and _then_ test for the smaller maxcount and take another action. To summarize, I think that a "length error" should be allowed at an implementation defined size limit which is less than or equal to NPOS. There should also be a function available to return this value.cause."

Proposed Resolution:
Requester: John Dlugosz: jdlugosz@objectspace.com
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-014
Title: Argument order for copy() is incorrect..
Section: 21.1.1.8.7 [lib.string::copy]
Status: active
Description:

In private email, John Dlugosz wrote:
"In copy() the arguments are in a different order than on other functions. I suppose this was to provide for a default on pos. However, if someone does specify both he will be likely to get them backwards and the compiler will not catch this. I feel it is a point of usability that is not worth the default argument.
Provide two forms of copy() instead:

```
copy (dest, pos, len);
copy (dest,len);
```

Note: The current interface to copy is:

```
size_type copy(charT* s, size_type n, size_type pos=0);
```

Proposed Resolution:

Provide two forms of copy():

```
size_type copy(charT* s, size_type pos, size_type n);
```

This function differs from the current copy only in the order of its last two arguments and the lack of a default argument.

```
size_type copy(charT* s, size_type n);
```

Returns:

```
copy(s, 0, n);
```

.Requester: John Dlugosz: jdlugosz@objectspace.com
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-017
Title: Can `reserve()` cause construction of characters?.
Section: 21.1.1.6 [lib.string.capacity]

Status: active
Description:

In private email, John Dlugosz wrote:
"Also, totally unspecified, is the treatment of the `reserve' area with respect to element creation and destruction. I chose to construct elements in the reserve area right away, and then the string grows into the reserve area using assignment semantics. This causes dramatic simplification in several areas, and allows me to implement it without the need for in-place construction and explicit destructor calls (important when targeting cfront-based compilers)."

Proposed Resolution:
Requester: John Dlugosz: jdlugosz@objectspace.com
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-018
Title: Specification of traits class is constraining.
Section: 21.1.1.2 [lib.string.char.traits]
Status: active
Description:

In private email, John Dlugosz wrote:
"The austerity of the traits class strongly suggests certain implementations and prevents certain optimizations. For a simple example, the copy() function does not provide for overlapping copies. Say I have a string "ABr" where A and B represent substrings of some length, and r is unused reserve area. I want to insert "C" into the string, and the length of "ACB" fits into the pre-existing allocation (because C is shorter or equal in size to r). I can't just copy B down to the tail end. Instead, I have to reallocate the whole string and copy the A part also.

"More significantly, the find() functions pretty much have to be implemented by a brute-force approach as they are defined-- locate a place where the match occurs. In short, I wish the traits available were richer. It seems inconsistant w.r.t. copy semantics, as explained in [issue 23-017], and it is so simple as to force inefficiencies in the implementation. In addition, it would be nice if additional implementation-specific stuff could be placed in the traits class. This can be done and still allow for user-defined "custom" traits to be created that only have the standard members, by using inheritance."

Proposed Resolution:
Requester: John Dlugosz: jdlugosz@objectspace.com
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-024
Title: Name of traits delimiter function is confusing
Section: 21.1.1.1 [lib.string.char.traits]
Status: active
Description:

The name of the string_char_traits function is "is_del". This has the connotation of "is delete".

Proposed Resolution:

Change the name of this member to "is_delim".

Requester: John Hinke: jhinke@qds.com
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-025
Title: Does `string_char_traits` need a locale?
Section: 21.1.1.2 [lib.string.char.traits.members]
Status: active
Description:

The description of the member `string_char_traits::is_del()` says it returns: `isspace()`. This function is subject to localization. Does this mean that `string_char_traits` is locale sensitive?

Proposed Resolution:
Requester: John Hinke: jhinke@qds.com
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-026
Title: Description of `string_char_traits::compare()` is expressed in code.
Section: 21.1.1.2 [lib.string.char.traits.members]
Status: active
Description:

The description of the `string_char_traits` member:

```
static int compare(const char_type* s1, const char_type* s2,
                   size_t n);
```

is expressed in code as follows:

```
for (size_t i=0; i<n; ++i, ++s1, ++s2)
       if (ne(*s1, *s2))
              return (lt(*s1, *s2) ? -1 : 1;
return 0;
```

It should be expressed in prose.

Proposed Resolution:

Replace the description with the following:

Returns: 0 iff for each i: 0<i<n the expression eq(*(s1+i), *(s2+i)) is true. Otherwise, returns -1 given i and j such that for j: 0 <= j < n, the expression lt(*(s1+j), *(s2+j)) is true and for each i: 0 < i < j the expression eq(*(s1+i), *(s2+i)) is true.

Otherwise returns 1.

Requester: Rick Wilhelm: rkw@chi.andersen.com
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-027
Title: Description of `string_char_traits::compare()` overspecifies return value.
Section: 21.1.1.2 [lib.string.char.traits.members]
Status: active
Description:

The description of the string_char_traits member:

```
static int compare(const char_type* s1, const char_type* s2,
                   size_t n);
```

is expressed in code as follows:

```
for (size_t i=0; i<n; ++i, ++s1, ++s2)
       if (ne(*s1, *s2))
```

```
                return (lt(*s1, *s2) ? -1 : 1;
return 0;
```

Specifiying the exact return values when the comparison returns “less than” or “greater than” is too constraining.

Proposed Resolution:

Replace “-1” with “an integer less than zero” and replace “1” with “an integer greater than 0”

Requester: Rick Wilhelm: rkw@chi.andersen.com
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-028

Title: Description of `string_char_traits::length()` is expressed in code.
Section: 21.1.1.2 [lib.string.char.traits.members]
Status: active
Description:

The description of the `string_char_traits` member:

```
static int length(const char_type* s);
```

is expressed in code as follows:

```
size_t len = 0;
while (ne(*s++, eos())) ++len;
return len;
```

It should be expressed in prose.

Proposed Resolution:

Replace the description with the following:

> Returns: the lowest value of i such that for i, i>=0, the expression ne(*(s+i), eos()) returns false and for each j, 0 <= j <= i the expression ne(*(s+j), eos()) returns true and.

Requester: Rick Wilhelm: rkw@chi.andersen.com
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-029

Title: Description of `string_char_traits::copy()` is overconstraining.
Section: 21.1.1.2 [lib.string.char.traits.members]
Status: active
Description:

The description of the member `string_char_traits::copy ()`

```
char_type* s = s1;
for (size_t i=0; i<n; ++i) assign(*s1++, *s2++);
```

This overconstrains implementations, in that there is no particular reason to do the operations in the order specified. (Box 78).

Proposed Resolution:

Replace the description as follows:

> Effects: Copies elements. For each non-negative integer i < n, performs
> *(s1 + i) = *(s2 + i).
> Returns: s1.
> Requires: s1 shall not be in the range [s2, s2+n).

Requester: Rick Wilhelm: rkw@chi.andersen.com
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-030

Title: Description of `string_char_traits::copy()` is silent on overlapping strings.

Section: 21.1.1.2 [lib.string.char.traits.members]

Status: active

Description:

The description of the member string_char_traits::copy ()

```
char_type* s = s1;
for (size_t i=0; i<n; ++i) assign(*s1++, *s2++);
```

Doesn't explicitly address the issue of overlapping strings.

Proposed Resolution:

Add the following to the description of string_char_traits::copy():

> Requires: s1 shall not be in the range [s2, s2+n).

This is similar to the approach followed by `copy()` in 25.2.1 [lib.alg.copy].

Requester: Rick Wilhelm: rkw@chi.andersen.com

Owner:

Emails: (none)

Papers: (none)

Issue Number: 21-031

Title: Copy constructor takes extra argument to switch allocator but does not allow allocator to remain the same.

Section: 21.1.1.4 [lib.string.cons]

Status: active

Description:

The copy constructor:

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

takes an extra argument, so that it can be used to copy a string while changing its allocator. Is this the best way to do this? (Box 79).

This copy constructor does not allow the user to retain the same allocator as the current string. Additionally, the string class does not provide a member to access a string's allocator.

Proposed Resolution:

The solution to this issue exactly mirrors the solution to a general containers issue.

At the Monterey meeting, the following change was approved and inserted into the WP:

In section 21.1.1.9 [lib.string.ops], add the member:

```
const allocator_type& get_allocator() const;
```

Returns: a reference to the string's allocator object.

The resolution to the default Allocator argument is pending the resolution to a similar issue in Clause 23.

Requester: Rick Wilhelm: rkw@chi.andersen.com. See also public comment T21 (p. 108)

Owner:

Emails: (none)

Papers: (none)

Issue Number: 21-034

Title: Inconsistency in requirements statements involving npos

Section: 21.1.1.4 [lib.string.cons] and 21.1.1.6 [lib.string.capacity]

Status: active

Description:

In the current draft, the requirements for
`basic_string(size_type n, charT c, Allocator& = Allocator());`
read:

Requires: `n < npos`.

and the requirements for
`void resize(size_type n, charT c);`
read:

Requires: `n != npos`.

These should be expressed in terms of max_size()

Proposed Resolution:

Change the description of both these members to:

Requires: n <= max_size()

Throws: length_error if n > max_size()

Requester: Rick Wilhelm: rkw@chi.andersen.com See also public commnet T21 (p. 109)
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-034a
Title: Expand ability to throw length_error
Section: 21.1.1.3 [lib.basic.string]
Status: active
Description:

The specification carefully dictates that a string should be able to hold the number of entities indexed by a size_type. This is evidenced, for example, in the strict specification of when a length_error exception is thrown in basic_string::replace.

Strictly interpreted, this prevents storage of other information in the same memory block as the data (e.g., reference counts of string lengths). It should be possible to throw a length_error when the resulting data size *plus the size of the overhead information* exceeds the capacity of a size_type.

It may be convenient to specify length_error conditions in terms of the max_size() value.

Proposed Resolution:
Requester: Judy Ward: ward@roguewave.com
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-037
Title: Traits needs a move() for overlapping copies.
Section: 21.1.1.4 [lib.string.cons]
Status: active
Description:

A move() member for overlapping copies would be a useful addition to the string_char_traits class.

Proposed Resolution:
Requester: Judy Ward: ward@roguewave.com
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-059

Title: String traits have no relationship to iostream traits.
Section: 21.1.1.1 [lib.string.char.traits]
Status: active
Description:

I would like to propose (whether officially or not) to modify the current CD:

```
template <class charT> struct ios_traits {};
```

to

```
template <class charT> struct ios_traits :
    public string_char_traits<charT> {};
```

in order to integrate the closely related traits, 'ios_traits' and 'string_char_traits'.

We can expect the integration of the common features, such as 'eq', 'eos', 'length', and 'copy' which is now inappropreately separated with no explicit reasons.

In lib-3832, Nathan Myers wrote:
“I have been careful to avoid getting too involved with Clause 21, thus far, because I have been quite busy with other chapters. However, it would be my recommendation to eliminate most of the string character traits: eq(), ne(), lt(), assign(), char_in(), char_out(), and is_del(). Also, I would either add a few "speed-up functions" needed to efficently implement strings without specialization, such as a move() member, or eliminate them all, and let the implementation specialize speedups for types known to it.”

A public comment included the follwoing:
“string_char_traits is missing three important speed-up functions, the generalizations of memchr, memmove, and memset. Nearly all the mutator functions in basic_string can be expressed as calls to these three primitives, to good advantage.”

See also issue 21-018.

Proposed Resolution:

More detailed work needed on this topic.
Requester: Norihiro Kumagai: kuma@slab.tnr.sharp.co.jp.
See also Public Comment T21 (p. 108).
Owner:
Emails: lib-3832
Papers: (none)

Issue Number: 21-060

Title: string_char_traits::ne not needed
Section: 21.1.1.1 [lib.string.char.traits]
Status: active
Description:

A public comment included:
“string_char_traits::ne is hardly needed given the member eq. It should be removed.
Proposed Resolution:

Remove the member string_char_traits.
Requester: Public comment T21 (p. 107)
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-061
Title: Missing explanation of traits specialization
Section: 21.1.1.2 [lib.string.char.traits.members]
Status: active
Description:

A public comment noted:
"No explanation is given for why the descriptions of the members of template class string_char_traits are "default definitions." If it is meant to suggest that the program can supply an explicit specialization, provided the specialization satisfies the semantics of the class, then the text should say so (here and several other places as well).

Proposed Resolution:

None.

Requester: Public comment T21 (p. 108).
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-062
Title: Missing explanation of requirements on charT.
Section: 21.1.1.3 [lib.basic.string]
Status: active
Description:

A public comment noted:
Paragraph 1 doesn't say enough about the properties of a "char-like object." It should say that it doesn't need to be constructred or destroyed (otherwise, the primitives in string_char_traits are woefully inadequate).
string_char_traits::assign (and copy) must suffice either to copy or initialize a char_like element. The definition should also say than an allocator must have the same definitions for the types size_type, difference_type, pinter, const_pointer, reference, and const_reference as class allocator::types<charT> (again because string_char_traits has no provision for funny address types).

Proposed Resolution:

None.

Requester: Public comment T21 (p. 108).
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-063
Title: No constraints on constructor parameter.
Section: 21.1.1.4 [lib.string.cons]
Status: active
Description:

The description of the constructor
`basic_string(const charT* s, size_type n, Allocator&);`
Doesn't constrain the size_type parameter.

Proposed Resolution:

Modify the description of the constructor as follows:

Requires: s shall not be a null pointer and n <= max_size().
Throws: length_error if n > max_size()

Requester: Public comment T21 (p. 108)
Owner:
Emails: (none)

Papers: (none)

Issue Number: 21-067
Title: Traits specializations are over-constrained for eos() member
Section: 21.1.1.2 [lib.string.char.traits.members]
Status: active
Description:

The current description is:

> Returns: The null character, char_type()

However, if the traits are specialized, the specialization should not be required to return the result of the default constructor.

Proposed Resolution:

Change the description to be:

> Returns: The null character.

Requester: Public comment T21 (p. 108).
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-068
Title: What is the proper role of the "Notes" section in Clause 21.
Section: 21.1.1.6 [lib.string.capacity] (and several other sections in the clause)
Status: active
Description:

Clause 21 currently contains several sections which include the text:

> Notes:

The draft already says that notes are non-normative. However, the contents of these sections are often normative. Should the contents of these sections be moved into other sections.

Also, the Notes sections currently give information on the use of some traits. The Japanese delegation would like to see information on the use of traits expanded to give the user more information about the impact of traits on the string template. However, one public comment described these sorts of notes on traits as over-specification.

Proposed Resolution:
Requester: Public comment T21 (p. 108).
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-074
Title: Should basic_string have a member semantically equivalent to strlen()
Section: 21.1.1.6 [lib.string.capacity])
Status: active
Description:

The basic_string template contains two member functions which return the number of characters in the string: size() and length(). Issue 21-054 proposed changing the semantics of length() to return the number of characters in the string which are positioned before the first traits::eos() character.

In discussions in Monterey, the LWG rejected the notion of changing the semantics of length(), but agreed to discuss addiing a new member which is semantically equivalent to C's strlen().

Proposed Resolution:

Add the following member to 21.1.1.6 [lib.string.capacity]

```
size_type c_strlen() const;
```

Returns: the minimum of length() and the number of char-like objects currently in the string which appear before the first traits::eos() character.

Requester: LWG
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-074
Title: Incomplete specification for assignment operator
Section: 21.1.1.4 [lib.string.cons]
Status: active
Description:

The current description of the basic_string assignment operator does not handle the case of a string being assigned to itself.

Proposed Resolution:

In the basic_string assignment operator's Effects description, add the following after the table:

If *this and str are the same object, no effect.

Requester: LWG
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-075
Title: Inconsistent pattern of arguments in basic_string overloads
Section: 21.1.1.3 [lib.template.string]
Status: active
Description:

During discussions at the Monterey meeting, the LWG determined that the pattern of arguments and overloads used in member functions is often inconsistent and confusing.

Most of these inconsistencies relate to size_type parameters referring either to the lvalue (this) or the rvalue (a parameter passed to the member function.

Proposed Resolution:

A paper with a proposed solution is forthcoming.

Requester: LWG
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-076
Title: Inconsistent pattern of arguments in basic_string overloads
Section: 21.1.1.3 [lib.template.string]
Status: active
Description:

Although basic_string has been modified to conform to the requirements for Sequences specified in Clause 23, no language in the WP specifically states that basic_string is a Sequence.

Proposed Resolution:

Add appropriate language to Clause 23. [More detail forthcoming.]

Requester: LWG

Owner:
Emails: (none)
Papers: (none)

Closed Issues

Issues which have been recently closed are included in their entirety. Issues which have appeared in a previous version of the issues list as "closed" have the bulk of their content deleted for brevity. The document number of the paper in which they last appeared is included for reference.

Issue Number: 21-001
Title: Should basic_string have a getline() function?
Last Doc.: N0721=95-0121

Issue Number: 21-003
Title: Character-oriented assign function has incorrect signature
Last Doc.: N0721=95-0121

Issue Number: 21-004
Title: Character-oriented replace function has incorrect signature
Section: 21.1.1.8.6 [lib.string::replace]
Status: closed
Description:

As specified in N0557=94-0170, which was accepted in Valley Forge, the character-oriented replace member has the interface:

```
basic_string<charT,traits,Allocator>&
replace(size_type pos, size_type n, const T c = T());
```

This interface should be as follows:

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

This change was inadvertently introduced and should be removed.

Resolution:

Replace the text:

```
basic_string<charT,traits,Allocator>&
replace(size_type pos, size_type n, const T c = T());
```

Returns: replace(pos, n,basic_string<charT,traits,Allocator>(c, n)).

with the following:

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

Returns: replace(pos, n1, basic_string<charT, traits, Allocator>(n2, c));

Requester: Rick Wilhelm: rkw@chi.andersen.com
Owner: Rick Wilhelm
Emails: (none)
Papers: 95-0028=N0628

Issue Number: 21-005
Title: How come the string class does not have a prepend() function?
Section: 21.1.1.8.2 [lib.string::append]
Status: closed
Description:

Judy thinks the prepend interface(s) should look just like the append() interfaces described in [lib.string::append] with the appropriate wording changes.

Resolution:

No change. There was no support or concrete proposal forthcoming.

Requester: Judy Ward: ward@roguewave.com
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-006
Title: Should the Allocator be the last template argument to basic_string?
Last Doc.: N0721=95-0121

Issue Number: 21-007
Title: Should the string_char_traits speed-up functions be specified as inline?
Section: 21.1.1.2 [lib.string.char.traits]
Status: closed
Description:

The string_char_traits speed-up functions:

```
static int compare(const char_type* s1, const char_type* s2,
                   size_t n);
static size_t length(chonst char_type*);
static char_type* copy(char_type*, const char_type*, size_t);
```

were originally proposed as being inline for efficiency. In the WP (dated 1 February 1995), they are not specified as inline.

Resolution:

No change, close the issue. The general consensus of library reflector messages and discussion in Austin was: inlining functions was an implementation detail and that functions could not be specified as inline in the Standard.

Requester: Takanori Adachi (taka@miwa.co.jp)
Owner:
Emails: lib-3519, lib-3520, lib-3522, lib-3523
Papers: (none)

Issue Number: 21-008
Title: Should an iostream inserter and extractor be specified for basic_string?
Section: 21.1.1.2 [lib.string.char.traits] and 21.1.1.10.8 (“Inserters and extractors”, no concordance entry)
Status: closed
Description:

In private email, Takanori Adachi wrote:
“In my original basic_string paper, I gave up trying to introduce the inserter and extractor operators since I felt that there is a traits-passing problem from basic_string to basic_iostream. But in the present WP, they are introduced as:

```
template<class charT, class traits, class Allocator>
basic_istream<charT>
operator>>(basic_istream<charT>& is,
           basic_string<charT,traits,Allocator>& a);

template<class charT, class traits, class Allocator>
basic_ostream<charT>
operator<<(basic_ostream<charT>& os,
           basic_string<charT,traits,Allocator>& a);
```

without considering the ios_char_traits, which seems to me to be a partial solution.

"I think, in order not to lose the power of traits, they should be replaced with the following:

```
template<class charT, class traits, class Allocator,
         class ios_traits = ios_char_traits(traits)>
   basic_istream<charT, ios_traits>
      operator>>(basic_istream<charT, ios_traits>& is,
                 basic_string<charT,traits,Allocator>& a);
template<class charT, class traits, class Allocator,
         class ios_traits = ios_char_traits(traits)>
   basic_ostream<charT, ios_traits>
      operator<<(basic_ostream<charT, ios_traits>& os,
                 basic_string<charT,traits,Allocator>& a);
```

when those operators are included in the basic_string.

"By the way, if you accept the above solution, you will realize there still need to be additional changes for the classes, ios_char_traits and string_char_traits. For the ios_char_traits, there will need to be a constructor like:

```
template<class string_traits>
   ios_char_traits(string_traits traits);
```

and the mechanism to reflect members of traits to the behaviors of the default functions of ios_char_traits, causing some new overhead in the iostream library.

"For the string_char_traits, two members, char_in and char_out will be parameterized with ios_traits like:

```
template<class ios_traits>
static basic_istream<charT,ios_traits>&
   char_in(basic_istream<charT,ios_traits>& is, charT& a);
template<class ios_traits>
static basic_ostream<charT,ios_traits>&
   char_out(basic_ostream<charT,ios_traits>& os, charT& a);
```

"My position is on the side of removing those operators from the basic_string. But if they remain, we should prepare to accept a somewhat complicated, full solution like the above."

The public comment included the text: "It seems to me that, to be useful, operator>>() must eat zero or more delimiters specified by basic_string<...>::traits::is_del() prior to reading each string."

Resolution:

Some changes changes required to address the issue. The full templatization of `operator<<()` and `operator>>()` was accomplished with iostreams resolutions in Austin. However, these changes were omitted from the July 95 draft. In the July 95 draft, there is no explanation of the insertion and extration operators. See issue 21-002 regarding the `char_in()` and `char_out()` members of traits.

In section 21.1.1.10.8 (Inserters and extractors), replace the descriptions of operator<<() and operator>>() with the following:

```
template<class charT, class IS_traits,
         class STR_traits, class STR_Alloc>
  basic_istream<charT,IS_traits>&
    operator>>(basic_istream<charT,IS_traits>& is,
               basic_string<charT,STR_traits,STR_Alloc>& str);
```

Effects: The function begins execution by calling is.ipfx(true). If that function returns true, the function endeavors to obtain the requested input.
The function extracts characters and appends them to str as if by calling str.append(1,c) If is.width() is greater than zero, the maximum number of characters stored n is is.width(); otherwise it is str.max_size()
Characters are extracted and stored until any of the following occurs:

— n characters are stored;
— end-of-file occurs on the input sequence;
— IS_traits::is_whitespace(c,ctype) is true for the next available input character c. In the above code fragment, the argument ctype is acquired by getloc().use<ctype<charT> >().

In any case, the function ends by calling is.isfx()
Returns: is

```
template<class charT, class IS_traits,
         class STR_traits, class STR_Alloc>
  basic_istream<charT,IS_traits>&
    operator<<(basic_istream<charT,OS_traits>& os,
               basic_string<charT,STR_traits,STR_Alloc>& str);
```

Effects: Behaves as if the function calls
`os.write(str.data(), str.size())`
Returns: os

Requester: Takanori Adachi (taka@miwa.co.jp)
See also public comment 6.12 and T21 (p. 107 and p. 111).
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-009
Title: Why are character parameters passed as “const charT”?
Last Doc.: N0721=95-0121

Issue Number: 21-010
Title: Should member parameters passed as “const_pointer”?
Last Doc.: N0721=95-0121

Issue Number: 21-011
Title: Why are character parameters to the string traits functions passed by reference?
Last Doc.: N0721=95-0121

Issue Number: 21-015
Title: The copy() member should be const.
Section: 21.1.1.8.7 [lib.string::copy]
Status: closed
Description:

In private email, John Dlugosz wrote:
“In copy(), I see no reason for not making the function const. In my implementation, I made it so.”

Note: The current interface to copy is:

```
size_type copy(charT* s, size_type n, size_type pos=0);
```

Resolution:

In 21.1.1.3 [lib.basic.string] and 21.1.1.9.7 [lib.string::copy], change to declaration of the memeber:

```
size_type copy(charT* s, size_type n, size_type pos=0);
```

to:

```
size_type copy(charT* s, size_type n, size_type pos=0) const;
```

Requester: John Dlugosz: jdlugosz@objectspace.com (Public Comment 6.6)

Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-016
Title: The error conditions are not well-specified for the find() and rfind() functions.
Section: 21.1.1.9.1 [lib.string::find]
Status: closed
Description:

In private email, John Dlugosz wrote:
"The error conditions are not very well specified for the find() and rfind() functions, nor do I feel that they are the most appropriate choice.

"My interpretation of 21.1.1.9.1 [lib.string::find] is that
1. an empty string will be found anywhere, so will always return `pos'.
2. passing in a pos that is too large is not an error, unlike most other functions in this class. Instead, it fails to match and returns NPOS. This is not explicit, but requires careful reading of the definition to figure out. However, rule 2 takes precedence over rule 1, so that searching for the empty string at an illegal position is _not_ found.

"I have three problems with this. First, making such boundary conditions or error conditions implicit rather than explicit will mean that users don't get a clear quick answer, and implementors may miss something and implement it incorrectly. I doubt many will realize that 2 takes precidence over 1 above, for example, and may happen to get it backwards. Second, the treatment of `pos' values out of range is inconsistant with the rest of the class. Third, it saves nothing in the implementation. Although as written it would seem that the boundary condition of pos out of range is handled naturally if you implement it the way it reads, that is not the case. The size_t domain cannot handle negative numbers, and the "natural" behavior is an incorrect result. Instead, an explicit test for the value of pos is needed in the code, before proceeding with the real work. As long as this test is necessary anyway, why not just throw a range error? Returning NPOS saves nothing in the implementation efficiency for normal in-range searches."

Resolution:

No change. See also 21-052.
Requester: John Dlugosz: jdlugosz@objectspace.com
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-019
Title: The `Allocator` template parameter is not reflected in a member typedef.
Section: 21.1.1.3 [lib.basic.string]
Status: closed
Description:

In lib-3593, Nathan Myers wrote:
"Looking through the Containers clause of the WP, I notice that, unlike all other class template parameters in the library, the `Allocator` parameter is not reflected in a member typedef.

"The reason for this is, I believe, historical; in earlier versions this parameter was a template template parameter, and the language offers no equivalent of typedef for templates."

Resolution:

Now that the parameter is a regular class type, it should be reflected in a member typedef. Note: this change is being made to all other containers which use run-time variable allocators. Add the following to the 21.1.3 [lib.basic.string].

```
typedef Allocator allocator_type;
```

Requester: Nathan Myers: myersn@roguewave.com
Owner:
Emails: lib-3593
Papers: (none)

Issue Number: 21-020
Title: Header for Table 42 is incorrect.
Section: 21.1.1.4 [lib.string.cons]
Status: closed
Description:

The header for Table 42 shows the arguments to a basic_string constructor in the incorrect order:
"Table 42 - basic_string(charT, size)type effects"

Resolution:

In 21.1.1.4 [lib.string.const], change the title of Table 42 as follows:
"Table 42 - basic_string(size_type, charT) effects"

Requester: afk@ElSegundoCA.ATTGIS.COM
(also pointed out by Richard Minner in lib-3711)
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-021
Title: `compare()` has unexpected results
Section: 21.1.1.9.8 [lib.string::compare]
Status: closed
Description:

The current wording for compare() is:
"Returns:
the nonzero result if the result of the comparison is nonzero. Otherwise, returns a value as indicated in Table 44:"

This causes the unexpected result of:
string("abcfoo").compare(string("abcx"),0,3) returns > 0, while
string("abcfoo").compare(string("abcbar"),0,3) returns 0.

A public comment noted:
"basic_string::compare has nonsensical semantics. Unfortunately, the last version approved, in July 94 resolution 16, is also nonsensical in a different way. The description should be restored to the earlier version, which at least has the virtue of captureing the intent of the original string class proposal:
1) If n < str.size(), it is replaced by str.size()
2) Compare the smaller of n and size() - pos with traits::compare().
3) If that result is nonzero, return it.
4) Otherwise, return negative for size() - pos < n, zero for size() - pos == n, or positive for size() - pos > n"

The proposed resolution should be compared to that of the public_comment.

Resolution:

Replace the following basic_string::compare members:

```
int compare(
      const basic_string<charT,traits,Allocator>& str,
      size_type pos = 0,  size_type n = npos) const;
int compare(const charT* s, size_type pos,
            size_type n) const;
int compare(const charT* s, size_type pos = 0) const;
```

with the following members:

```
int compare(
   const basic_string<charT,traits,Allocator>& str) const;
int compare(size_type pos1, size_type n1
      const basic_string<charT,traits,Allocator>& str,
      size_type pos2 = 0, size_type n2 = npos) const;
int compare(charT* s) const;
int compare(size_type pos, size_type n,
      const charT* s, size_type n = npos) const;
```

Replace the descriptions of the removed members with the following descriptions of the added members:

`int compare(const basic_string<charT, traits, Allocator>& str)`

Effects: Determines the effective length rlen of the strings to compare as the smallest of size() and str.size(). The function then compares the two strings by calling traits::compare(data(), str.data(), rlen).

Returns: the nonzero result if the result of the comparison is nonzero. Otherwise, returns a value as indicated in Table 44:

Table 44:

Condition	Return Value
size()< str.size()	< 0
size() == str.size()	0
size()> str.size()	> 0

`int compare(size_type pos1, size_type n1,`
`const basic_string<charT, traits, Allocator>& str,`
`size_type pos2 = 0, size_type n2 = npos) const;`

Returns: basic_string<charT,traits,Allocator>(*this, pos1,n1).compare(basic_string<charT,traits,Allocator>(str, pos2, n2))

`int compare(charT* s) const;`

Returns: *this.compare(basic_string<charT,traits,Allocator>(s))

`int compare(size_type pos, size_type n1,`
`charT* s, size_type n2 = npos) const;`

Returns: basic_string<charT,traits,Allocator>(*this, pos, n1).compare(basic_string<charT,traits,Allocator>(s, n2))

Requester: Jason Merrill: jason@cygnus.com
Public comment T21 (p 110)

Owner:

Emails: lib-3709, lib-3712

Papers: (none)

Issue Number: 21-022

Title: s.append('c') appends 99 nulls.

Section: 21.1.1.8.2 [lib.string::append]

Status: closed

Description:

In lib-3709, Jason Merrill writes:
"Is it really necessary to have any of the charT = charT() default arguments? They seem like much more a source of errors than a useful shortcut. How often are you going to want to add a lot of nulls to your string? Is it really such a hardship to make it explicit when you do?

"When I write s.append('c'), I expect it to add a 'c' to the end of the string, not to add 99 nulls. Is there some requirement that prevents it from doing what I want? The default argument doesn't seem to be part of the container or sequence requirements, and having an

```
append (charT c)
```

in addition to the

```
append (size_type n, charT c)
```

(and similar additional functions for assign, insert and replace) would be analogous to the iterator insert methods that *are* part of the sequence requirements."

Resolution:

With the acceptance of a modified version of 95-0091=N0691, this issue can be closed.

Requester: Jason Merrill: jason@cygnus.com
Owner:
Emails: lib-3709, lib-3711, lib-3712, lib-3722, lib-3723, lib-3724,
Papers: 95-0091 / N0691

Issue Number: 21-023
Title: Non-conforming default `Allocator` arguments
Section: 21.1.1.4 [lib.string.cons]
Status: closed
Description:

The defaulted `Allocator` arguments as declared do not conform to the language specification. The specification:

```
explicit basic_string(Allocator& = Allocator());
```

causes a compiler warning.

In lib-3731, Nathan Myers wrote:
"The line of reasoning for making that argument non-const was that one would need to call non-const members of it. However, that interferes with passing it as a default value. The solution becomes evident when you consider that basic_string, or any other collection, must make a copy of the Allocator argument anyway; they can use the (non-const) copy."

Resolution:

Change the type of all Allocator arguments in all basic_string members from

```
Allocator&
```

to

```
const Allocator&
```

Requester: Judy Ward: ward@roguewave.com
Owner:
Emails: lib-3730, lib-3731
Papers: (none)

Issue Number: 21-032
Title: Description for operator+() is incorrect

Section: 21.1.1.10.1 [lib.string.string::op+]
Status: closed
Description:

In the current draft:

```
template<class charT, class traits, class Allocator>
 basic_string<charT, traits, Allocator>
  operator+(const basic_string<charT, traits, Allocator>& lhs,
            const basic_string<charT, traits, Allocator>& rhs);
```

is described by:

Returns: lhs.append(rhs)

These are the incorrect semantics. The lhs argument is not modified, a new string object is created and returned.
(This was pointed out by bob_kline@stream.com in comp.std.c++.)

Note: this issue is the same as 21-050.

Resolution:

Change the description of this operator to be:

Returns: `basic_string<charT,traits,Allocator>(lhs).append(rhs)`

Requester: Rick Wilhelm: rkw@chi.andersen.com
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-033
Title: Requirements for `const charT*` arguments not specified
Section: throughout clause 21
Status: closed
Description:

In the current draft, `basic_string` and `string_char_traits` members which take an argument of type `const charT*` fail to specify that the argument shall not be null. The appropriate constructors specify:

Requires: `s` shall not be a null pointer.

but most of the other members do not.

Resolution:

No change. Close the issue. Other working paper text addresses this issue.

Requester: Rick Wilhelm: rkw@chi.andersen.com
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-035
Title: Character replacement does not change length.
Section: 21.1.1.7 [lib.string.access]
Status: close
Description:

If a character in the middle is replaced with eos() using the char-type accessors / modifiers this currently does not change the length of the string as returned by length().

It seems a string is acting more like a container for data than a string of characters.

Resolution:

No change. Close this issue. The LWG has concluded in the past that basic_string can hold null characters and these characters are included in the length of the string.

Requester: Judy Ward: ward@roguewave.com
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-036
Title: Character case disregarded during common operations.
Section: 21.1.1.4 [lib.string.cons]
Status: closed
Description:

The case of a character is not adequately addressed despite being a very common use of a string class. For example, searches and comparisons should have programmable case dependence. There should be members/functions to operate on the case (toupper, etc.).

This would require extending the traits class to allow a user's new char type to reflect case.

This issue can be addressed by creating separate traits classes, but this approach is not intuitive to the average user. It is also costly and inflexible when mixing different case handling properties.

Resolution:

No change. Close the issue. Character case is a locale-dependent consideration.

Requester: Judy Ward: ward@roguewave.com
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-038
Title: Operator < clashes cause ambiguity
Section: 21.1.1.10.4 [lib.string::op<]
Status: closed
Description:

operator< clashes between that defined directly for basic_string in the specification and the one provided for generally by <functional> in the STL, .e.g. this will give an ambiguity error:

```
template <class charT>
struct basic_string {
};

// simplified from string
template <class charT>
inline int operator>(
  const basic_string<charT>& lhs,
  const basic_string<charT>& rhs
) { return 0; }

// simplified from the STL
template <class T>
inline int operator>(const T& x, const T& y) {
    return 0;
}

int main() {
       basic_string<char> s1,s2;
       if (s1 > s2) ;
       return 0;
```

```
}
```

Resolution:

No change. Close this issue. Core language rules are now sufficient to handle these situations.

Requester: Judy Ward: ward@roguewave.com
See also public comment T21 p. 110-111

Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-039

Title: Iterator parameters can get confused with size_type parameters.
Section: 21.1.1.4 [lib.string.cons]
Status: closed
Description:

Parameters of type iterator can get confused in typical usage with size_type parameters. When a programmer uses a literal constant, the compiler can cast to either type and confusion results.

For example, with the standard basic_string<char>:

```
string s;
s.replace(0, 1, "test");
```

could be either:

```
s.replace(size_type pos, size_type n1, const charT * s)
s.replace(iterator i1, iterator i2, const charT * s)
```

if char pointers are used to implement the iterators.

This leads to ambiguity errors unless the users uses casts.

Resolution:

No change. Close the issue. Overloading rules have been refined to handle these cases.

Requester: Judy Ward: ward@roguewave.com
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-040

Title: Repetition parameter non-intuitive
Section: 21.1.1.8.2 [lib.string.::append]
Status: closed
Description:

The placement of the repetition parameter before the character parameter makes for some very non-intuitive usages. For example, s.append(1, 'a') is required now instead of s.append('a').

Resolution:

No change. Close the issue. Also see issue 21-022.

Requester: Judy Ward: ward@roguewave.com
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-041

Title: Assignment operator defined in terms of itself
Section: 21.1.1.4 [lib.string.cons]
Status: closed

Description:

basic_string assignment operator seems to be defined in terms of itself:

```
basic_string<...>& operator=(const basic_string<...>& str);
```

Returns:

```
*this = basic_string<...>(str);
```

Resolution:

Change the description as follows:

Effects: If *this and str are not the same object, modifies *this such that:

Element	Value
data()	points at the first element of an allocated copy of the array whose first element is pointed at by str.data()
size()	str.size()
capacity()	a value at least as large as size()

If *this and str are the same object, the member has no effect.

Returns: *this

Requester: Sean Corfield: sean_corfield@prqa.co.uk
Owner:
Emails: lib-3789
Papers: (none)

Issue Number: 21-042
Title: Character assignment defined in terms of non-existent constructor
Section: 21.1.1.4 [lib.string.cons]
Status: closed
Description:

The basic_string character assignment operator is defined in terms of a constructor that does not exist:

```
basic_string<...>& operator=(charT c);
```

Returns:

```
*this = basic_string<...>(c);
```

Resolution:

Change the description of the member:

```
basic_string<charT,traits,Allocator>& operator=(charT c)
```

as follows:

Returns: *this = basic_string<charT,traits,Allocator>(1, c)

Requester: Sean Corfield: sean_corfield@prqa.co.uk (Public Comment 6.2)
Owner:
Emails: lib-3789
Papers: (none)

Issue Number: 21-043
Title: Character append operator defined in terms of non-existent constructor
Section: 21.1.1.8.1 [lib.string::op+=]
Status: closed
Description:

The basic_string character append operator is defined in terms of a constructor that does not exist:

```
basic_string<...>& operator+=(charT c);
```

Returns:

```
*this = basic_string<...>(c);
```

Resolution:

Change the description of the member:

```
basic_string<charT,traits,Allocator>& operator+=(charT c)
```

to:

Returns: *this += basic_string<charT,traits,Allocator>(1, c)

Requester: Sean Corfield: sean_corfield@prqa.co.uk (Public Comment 6.2)
Owner:
Emails: lib-3789
Papers: (none)

Issue Number: 21-044
Title: Character modifiers defined in terms of non-existent constructor
Section: 21.1.1.8.[2-4,6] [lib.string::append] [lib.string::assign] [lib.string::insert] [lib.string::replace]
21.1.1.9.[1-6] [lib.string::find] [lib.string::rfind] lib.string::find.first.of] [lib.string::find.last.of] [lib.string::find.first.not.of] [lib.string::find.last.not.of]
21.1.1.10.1 [lib.string::op+]
Status: closed
Description:

Several members in these sections are defined in terms of a non-existent constructor. These descriptions are incorrect:

```
basic_string<...>& append(size_type n, charT c = charT());
```
Returns: append(basic_string<...>(c,n));
```
basic_string<...>& assign(size_type n, charT c = charT());
```
Returns: assign(basic_string<...>(c,n));
```
basic_string<...>&
       insert(size_type pos, size_type n, charT c = charT());
```
Returns: insert(pos, basic_string<...>(c,n));
```
basic_string<...>&
       replace(size_type pos, size_type n, charT c = charT());
```
Returns: replace(pos, n, basic_string<...>(c,n));
```
size_type find(charT c, size_type pos = 0) const;
```
Returns: find(basic_string<...>(c), pos);
```
size_type rfind(charT c, size_type pos = npos) const;
```
Returns: find(basic_string<...>(c,n), pos);
```
size_type find_first_of(charT c, size_type pos = 0) const;
```
Returns: find_first_of(basic_string<...>(c), pos);
```
size_type find_last_of(charT c, size_type pos = npos) const;
```
Returns: find_last_of(basic_string<...>(c), pos);
```
size_type find_first_not_of(charT c, size_type pos = 0) const;
```
Returns: find_first_of(basic_string<...>(c), pos);
```
size_type find_last_not_of(charT c, size_type pos = npos) const;
```
Returns: find_last_not_of(basic_string<...>(c), pos);
```
template<class charT, class traits, class Allocator>
  basic_string<...>
    operator+(charT lhs, const basic_string<...>& rhs);
```
Returns: basic_string<...>(lhs) + rhs;
```
template<class charT, class traits, class Allocator>
  basic_string<...>
    operator+(const basic_string<...>& lhs, charT rhs);
```
Returns: lhs + basic_string<...>(rhs);

Resolution:

In the following description, the text: “basic_string<...>“ represents “basic_string<charT,traits,Allocator>“. It is substituted here for clarity.
Change the descriptions as follows

```
basic_string<...>& append(size_type n, charT c = charT());
```
Returns: append(basic_string<...>(n,c));
```
basic_string<...>& assign(size_type n, charT c = charT());
```
Returns: assign(basic_string<...>(n,c));
```
basic_string<...>&
       insert(size_type pos, size_type n, charT c = charT());
```
Returns: insert(pos, basic_string<...>(n,c));
```
basic_string<...>&
       replace(size_type pos, size_type n, charT c = charT());
```

Returns: replace(pos, n, basic_string<...>(n,c));

```
size_type find(charT c, size_type pos = 0) const;
```

Returns: find(basic_string<...>(1,c), pos);

```
size_type rfind(charT c, size_type pos = npos) const;
```

Returns: find(basic_string<...>(1,c), pos);

```
size_type find_first_of(charT c, size_type pos = 0) const;
```

Returns: find_first_of(basic_string<...>(1,c), pos);

```
size_type find_last_of(charT c, size_type pos = npos) const;
```

Returns: find_last_of(basic_string<...>(1,c), pos);

```
size_type find_first_not_of(charT c, size_type pos = 0) const;
```

Returns: find_first_of(basic_string<...>(1,c), pos);

```
size_type find_last_not_of(charT c, size_type pos = npos) const;
```

Returns: find_last_not_of(basic_string<...>(1,c), pos);

```
template<class charT, class traits, class Allocator>
  basic_string<...>
    operator+(charT lhs, const basic_string<...>& rhs);
```

Returns: basic_string<...>(1, lhs) + rhs;

```
template<class charT, class traits, class Allocator>
  basic_string<...>
    operator+(const basic_string<...>& lhs, charT rhs);
```

Returns: lhs + basic_string<...>(1, rhs);

Requester: Sean Corfield: sean_corfield@prqa.co.uk
See also public comments 6.2 and T21 (p. 109)
Owner:
Emails: lib-3789
Papers: (none)

Issue Number: 21-045
Title: Iterator typenames overspecified
Section: 21.1.1.3 [lib.basic.string]
Status: closed
Description:

The declarations for iterator and const_iterator are over-constraining. They are:

```
typedef typename Allocator::types<charT>::pointer iterator;
typedef typename Allocator::types<charT>::const_pointer
const_iterator;
```

Resolution:

The exact type equivalency is left implementation-defined, but the presence of these typenames is required. Add the following to the synopsis in 21.1.1.3 [lib.basic.string]:

```
typedef typename implementation_defined iterator;
typedef typename implementation_defined const_iterator;
```

Requester: Nathan Myers: myersn@roguewave.com
Owner:
Emails: lib-3810
Papers: (none)

Issue Number: 21-046
Title: basic_string type syntactically incorrect in some descriptions
Section: 21.1.1.8.[2,3,5,6] [lib.string::append] [lib.string::assign] [lib.string::remove] [lib.string::replace]
Status: closed
Description:

The return types for several member functions in these sections is incorrectly specified as:

```
basic_string&
```

This should be fixed in other areas of the clause where the term does not refer to a construtor or a destructor.

Resolution:

Change the return type of all overloads of the members: append(), assign(), remove(), and replace() to be:

```
basic_string<charT,traits,Allocator>&
```

Requester: Rick Wilhelm: rkw@chi.andersen.com (Public Comment 6.1)
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-047
Title: Error in description of replace() member
Section: 21.1.1.8.6 [lib.string::replace]
Status: closed
Description:

In the 'Effects:' section for the first replace() function, in the first sentence, there is a '&' character in front of the name 'pos1'. This would have the undesired effect of taking the address of the parameter.

Resolution:

In the first section of the 'Effects:' section of the first replace() member, remove the '&' from in front of the argument name 'pos1'.

Requester: Rick Wilhelm: rkw@chi.andersen.com (Public Comment 6.3)
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-048
Title: Inconsistency in const-ness of compare() declarations
Section: 21.1.1.9.8 [lib.string::compare]
Status: closed
Description:

The declaration of the first compare function does not indicate the function is const. The function is correctly declared const in 21.1.1.3 [lib.basic.string]

Resolution:

Change the declaration of the first compare member in 21.1.1.9.8 [lib.string::compare] to make the member const:

```
int compare(const basic_string<charT,traits,Allocator>& str,
            size_type pos = 0, size_type n = npos) const;
```

Requester: Rick Wilhelm: rkw@chi.andersen.com (Public Comment 6.4)
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-049
Title: Inconsistency constructor effects and semantics of data()
Section: 21.1.1.4 [lib.string.cons]
Status: closed
Description:

Table 38, which describes the effects of basic_string(Allocator& = Allocator()) , indicates the value of data() is “an unspecified value” and the value of size() is 0. This contradicts the semantics of data specified in 21.1.1.9 [lib.string.ops]. This section states:

> Returns: c_str if size() is non-zero, otherwise a null pointer.

For a related issue on date(), see issue 21-058.

Resolution:

Change Table 38 to show the value of data() to be "a non-null pointer that is copyable and can have 0 added to it.".

Requester: Rick Wilhelm: rkw@chi.andersen.com (Public Comment 6.5)
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-050
Title: Incorrect semantics for operator+()
Section: 21.1.1.10.1 [lib.string::op+]
Status: closed
Description:

The semantics of the operator

```
template<class charT, class traits, class Allocator>
  basic_string<...>
       operator+(const basic_string<...>& lhs,
                 const basic_string<...>& rhs);
```

are incorrectly given as:

Returns: lhs.append(rhs).

Note: This issue is the same as 21-032. It was added inadvertently.

Resolution:

Change the description to:

Returns: basic_string<charT,traits,Allocator>(lhs).append(rhs).

Requester: Rick Wilhelm: rkw@chi.andersen.com (Public Comment 6.7 and T26.13)
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-051
Title: Incorrect return type for insert() member
Section: 21.1.1.8.4 [lib.string::insert]
Status: closed
Description:

The following member has a return type as specified.

```
iterator insert( iterator p, size_type n, charT c = charT() );
```

As specified in 23.1.1 [lib.sequence.reqmts] Table 52, this should be void.

Resolution:

Change the return type of the member:

```
iterator insert( iterator p, size_type n, charT c = charT() );
```

to:

```
void insert( iterator p, size_type n, charT c = charT() );
```

Requester: Rick Wilhelm: rkw@chi.andersen.com (Public Comment 6.14)
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-052
Title: Unconstrained position arguments for find members.
Section: 21.1.1.9.[1-6] [lib.string::find] [lib.string::rfind] lib.string::find.first.of] [lib.string::find.last.of] [lib.string::find.first.not.of] [lib.string::find.last.not.of]
Status: closed
Description:

There are no constraints on the pos arguments to the member functions in these sections.

Resolution:

No change. Close the issue. The return values of these members are capable of indicating an unsuccessful search.
Note: See also issue 21-016

Requester: Rick Wilhelm: rkw@chi.andersen.com (Public Comment 6.16)
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-053
Title: Semantics of size() prevents null characters in string
Section: 21.1.1.6 [lib.string.capacity]
Status: closed
Description:

The description of size() includes the following:

> Notes: Uses traits::length()

This prevents the handling of a null character as part of the string.

Resolution:

Remove the following from 21.1.1.6 [lib.string.capacity]:

> Notes: Uses traits::length()

Requester: Takanori Adachi: taka@miwa.co.jp
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-054
Title: Change the semantics of length()
Section: 21.1.1.6 [lib.string.capacity]
Status: closed
Description:

The member length() will be more useful if it defines to return traits::length(c_str()) not just as a synonym of size().

Resolution:

No change. Close the issue. length() and size() should remain synonymous.

Requester: Takanori Adachi: taka@miwa.co.jp
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-055
Title: append(), assign() have incorrect requirements
Section: 21.1.1.8.[2-3] [lib.string::append] [lib.string::assign]
Status: closed
Description:

The description for:

```
basic_string<...>&
  append(const basic_string<...>& str,
         size_type pos =0, size_type n =npos);
basic_string<...>&
  assign(const basic_string<...>& str,
         size_type pos =0, size_type n =npos);
```

includes:

> Requires: pos <= size()

Since the pos argument refers to the str argument, this statement makes no sense in this context.

Resolution:

In sections 21.1.1.8.2[lib.string::append] and 21.1.1.8.3 [lib.string::assign], substitute the sentence:

Requires: pos <= size().

with

Requires: pos <= str.size().

Requester: Takanori Adachi: taka@miwa.co.jp
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-056
Title: Requirements for insert() are too weak.
Section: 21.1.1.8.4 [lib.string::insert]
Status: closed
Description:

The requirements for:

```
  basic_string<charT,traits,Allocator>&
    insert(size_type pos1,
           const basic_string<charT,traits,Allocator>& str,
           size_type pos2 = 0, size_type n = npos);
```

are too weak. They make no constraints on the requirements for the relationship between str and pos2.

Resolution:

In section 21.1.1.8.4 [lib.string::insert], substitute the sentence:

Requires: pos1 <= size().

with

Requires: pos1 <= size() and pos2 <= str.size().

Requester: Takanori Adachi: taka@miwa.co.jp
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-057
Title: replace has incorrect requirements
Section: 21.1.1.8.6 [lib.string::replace]
Status: closed
Description:

The description for:

```
basic_string<charT,traits,Allocator>&
    replace(size_type pos1, size_type n1,
            const basic_string<charT,traits,Allocator>& str,
            size_type pos2 = 0, size_type n2 = npos);
```

includes:

Requres: pos1 <= size() && pos2 <= size().

Since the pos2 argument refers to the str argument, this statement makes no sense in this context.

Resolution:

In section 21.1.1.8.6 [lib.string::replace], replace the sentence:

Requres: pos1 <= size() && pos2 <= size().

with

Requres: pos1 <= size() && pos2 <= str.size().

Requester: Takanori Adachi: taka@miwa.co.jp
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-058
Title: Description of data() is over-constraining.
Section: 21.1.1.9 [lib.string.ops]
Status: closed
Description:

The description for:
`const charT* data() const`
includes:

> Returns: c_str() if size() is nonzero, otherwise a null pointer.

This prevents this function from being used on strings with null characters.

Note: the original proposed change was:

> Returns: A pointer to the initial element of an array of length >= size() whose first size() elements equal the corresponding elements of the string controlled by *this.

This was modified as indicated below. See also: issue 21-049.

Resolution:

In this section, replace the sentence:

> Returns: c_str() if size() is nonzero, otherwise a null pointer.

with

> Returns: If size() is nonzero, the member returns a pointer to the initial element of an array whose first size() elements equal the corresponding elements of the string controlled by *this. If size() is zero, the member returns a non-null pointer that is copyable and can have zero added to it.

Requester: Takanori Adachi: taka@miwa.co.jp
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-064
Title: Miscellaneous errors in resize(size_type n)
Section: 21.1.1.6 [lib.string.capacity]
Status: closed
Description:

In the current draft, the description for
`void resize(size_type n);`
reads:

> Returns: `resize(n, eos());`

Since this is a void function, there should be no "returns" section. Also, it should append the default character and there should be constraints on the parameter.

Resolution:

Change the description of this member to:

> Effects: resize(n, charT());

Also, since the traits::eos() member is not used, the note refering to it should be removed.

Requester: Rick Wilhelm: rkw@chi.andersen.com See also public comment T21 (p. 109)
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-065
Title: Incorrect return value for insert()
Section: 21.1.1.8.4 [lib.string::insert]

Status: closed
Description:

In the current draft, the description for
`iterator insert(iterator p, charT c);`
includes:

> Returns: `p`

Since the iterator p may have been invalidated by the insertion it should not be returned.

Proposed Resolution:

Change the description of the member:
`iterator insert(iterator p, charT c);`
to:

> Returns: an iterator which refers to the copy of inserted character.

Requester: Rick Wilhelm: rkw@chi.andersen.com See public comment T21 (p. 109)
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-066
Title: Description of remove() is over-specific
Section: 21.1.1.8.5 [lib.string::remove]
Status: closed
Description:

In the current draft, the description for
`basic_string& remove(iterator p);`
includes:

> Effects: ... calls the character's destructor

The description for
`basic_string& remove(iterator first, iterator last);`
includes:

> Effects: ... calls the character's destructor
> Complexity: the desctructor is called a number of times exactly equal to the size of the range.

These descriptions are over-specific. Nowhere else in the clause is charactor construction or destruction mentioned.

Resolution:

In section 21.1.1.8.5 [lib.string::remove], remove the occurances of the phrase "and calls the character's destructor" and the Complexity clause.

Requester: Rick Wilhelm: rkw@chi.andersen.com See public comment T21 (p. 109-110)
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-069
Title: Swap complexity underspecified.
Section: 21.1.1.8.8 [lib.string::swap]
Status: closed
Description:

A public comment contained:
"Swap complexity says 'constant time.' It doesn't say with respect to what. Should probably say, 'with respect to the lengths of the two strings, assuming that their two allocator objects compare equal.' (This assumes added wording describing how to compare two allocator objects for equality.)

Resolution:

Any resolution should be examined in the context of the rest of the containers library since this member was added for compatibility.

Replace the text:
Complexity: constant time
with the text:
Complexity: linear in general, constant if a.get_allocator() == b.get_allocator()

Note: this resolution depends on the adoption of the get_allocator() member to retrieve the current allocator object for a container. See issue 21-031.

Requester: Public comment T21 (p. 110).
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-070
Title: operator>= described incorrectly
Section: 21.1.1.10.7 [lib.string::op>=]
Status: closed
Description:

The description of the third operator has an incorrect operation "<=" in the Returns section.

Resolution:

Change the description of the operator:

```
template<class charT, class traits, class Allocator>
bool operator>=(const basic_string<charT,traits,Allocator>& lhs,
                const charT* rhs);
```

to:

Returns: lhs >= basic_string<charT,traits,Allocator>(rhs)

Requester: Public comment T21 (p. 111).
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-071
Title: Does getline() have the correct semantics?
Section: 21.1.1.10.8 (no concordance entry)
Status: closed
Description:

A public comment noted:
"getline for basic_string reflects none of the changes adopted by July 94 resolution 26. It should not fail if a line exactly fills, and it should set failbit if it *extracts* no characters, not if it *appends* no characters. Should be changed to match 27.6.1.3"

and also:
"getline for basic_string says that extraction stops when npos - 1 characters are extracted. The proper value is str.max_size() (which is less than allocator.max_size(), but shouldn't be constrinted more precisely than that). Should be changed."

Resolution:

In section 21.1.1.10.8 (Inserters and extractors), replace the "Effects" section of the getline() description with the following:

Effects: The function begins by calling `is.ipfx(true)`. If that function returns true, the function endeavors to extract the requested input. It also counts the number of characters extracted. The string is initially made empty by calling str.remove(). Characters are extracted from the stream and appended to the string as if by calling str.append(1, c). Characters are extracted and appended until one of the following occurs:

1) end-of_file occurs on the input sequence (in which case, the function calls is.setstate(ios_base::eofbit)
2) c == delim for the next available input character c (in which case, c is extracted but not appended) (27.4.4.3)
3) str.max_size() characters are stored (in which case, the function calls is.setstate(ios_base::failbit) (27.4.4.3)

These conditions are tested in the order shown.

In any case, the function ends by storing the count in `is` and calling `is.isfx()`, then returning the value specified.

If the function extracts no characters, it calls is.setstate(ios_base::failbit) which may throw ios_base::failure (27.4.4.3)

Requester: Public comment T21 (p. 111).
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-072
Title: Incorrect use of size_type in third table in section
Section: 21.2 [lib.c.strings]
Status: closed
Description:

The third table in this section makes a reference to “size_type”. This is not defined in the cstring header and should be changed.

Resolution:

In the third table in 21.2 [lib.c.strings], change the occurance of “size_type” to “size_t”

Requester: Public comment T21 (p. 111).
Owner:
Emails: (none)
Papers: (none)

Issue Number: 21-073
Title: Add overloads to functions that take default character object.
Section: 21.1.1.3 [lib.basic.string]
Status: closed
Description:

In lib-3824, Taka writes:
There are seven members using charT() as their default arguments in the class template basic_string. I think it is problematic in two points: one is on the possibility of defining eos() as the element which is different from charT(). The other point is on the unclearness of their dependency on traits.

The usage of charT() as default arguments are not adequate in the following seven members:

```
basic_string& append(size_type n, charT c = charT());
basic_string& assign(size_type n, charT c = charT());
basic_string& insert(size_type pos, size_type n,
                     charT c = charT() );
iterator insert(iterator p, charT c = charT());
iterator insert(iterator p, size_type n, charT c = charT());
basic_string& replace(size_type pos, size_type n,
                      charT c = charT());
basic_string& replace(iterator i1, iterator i2,
                      size_type n, charT c = charT());
```

Resolution:

The default arguments charT() should be eliminated by separating those members into two forms and by using traits::eos() instead of charT() as the following:

```
basic_string& append(size_type n, charT c);
basic_string& append(size_type n);

basic_string& assign(size_type n, charT c);
basic_string& assign(size_type n);

basic_string& insert(size_type pos, size_type n, charT c);
basic_string& insert(size_type pos, size_type n);

iterator insert(iterator p, charT c);
iterator insert(iterator p);

iterator insert(iterator p, size_type n, charT c);
iterator insert(iterator p, size_type n);

basic_string& replace(size_type pos, size_type n, charT c);
basic_string& replace(size_type pos, size_type n);

basic_string& replace(iterator i1, iterator I2,
                      size_type n, charT c);
basic_string& replace(iterator i1, iterator i2, size_type n);
```

Proposed Resolution:

No change. Close the issue. The changes introduced by the resolution of issue 21-022 eliminate the default argument for these functions.

Requester: Takanori Adachi: taka@miwa.co.jp

Owner:

Emails: lib-3824

Papers: (none)