Doc. No.: X3J16/95-0080
WG21/ N0680
Date: March 24, 1995
Project: Programming Language C++
Reply To: Richard K. Wilhelm
Andersen Consulting
rkw@chi.andersen.com

Clause 21 (Strings Library) Issues List
Revision 3

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.

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.

Issues

Issue Number: 21-001
Title: Should basic_string have a getline() function?
Section: 21.1.1.4.5 (new) [lib.string::getline]
Status: closed
Description:

As identified by Beman Dawes in lib-3367, the 20 September 1994 draft of the WP does not include getline() . It was part of the 27 May 1994 draft of the WP. Beman suggested that getline() be reinstated with the semantics as specified in the earlier WP draft.

In lib-3408, Nathan Myers responded as follows:
"I'm quite concerned about the semantics implied in the string traits. There, it seems to be assumed that the end-of-line character is the same for all encodings of a character type. But, of course, even in ASCII we see an amazing variety of line-end conventions. Unicode is worse, with all the ASCII control characters and (as I recall) two more line-end characters.

"I fear that we cannot provide internationalized getline semantics with the same interface that we have had. I can imagine a getline() which takes the user's choice of line ending, but I can imagine you may want any of the available choices to end a line. The locale object's ctype facet does not provide an 'is_eol()' member, and POSIX does not provide the underlying support necessary to implement it in any case.

"It seems clear to me that the getline operation depends on the character-encoding in use, and that makes it a locale-dependent operation. It is not clear

to me how to propagate the information to the place where it is needed. It would like to avoid a 'virtual-function-call-per-character' when reading lines of text, because of performance problems."

Resolution: Resolved with the acceptance of N0611=95-0011 in Austin.
Requester: Beman Dawes: beman@dawes.win.net
Owner:
Emails: lib-3367, lib-3408, lib-3411, lib-3417, lib-3421
Papers: N0611=95-0011

Issue Number: 21-002
Title: Are string_traits members char_in() and char_out() necessary?
Section: 21.1.1.1 [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.]

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

Issue Number: 21-003
Title: Character-oriented assign function has incorrect signature
Section: 21.1.3.6 [lib.string::assign]
Status: closed
Description:

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

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

This interface should not take have its first parameter. This change was inadvertently introduced and should be removed. This issue will be closed if 2.5.4 of N0628=95-0028 is accepted.

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());
```

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

Issue Number: 21-004
Title: Character-oriented replace function has incorrect signature
Section: 21.1.1.4.8 [lib.string::replace]
Status: active
Description:

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

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

This interface should be as follows:

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

This change was inadvertently introduced and should be removed.

Resolution:
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: 21.1.3.5 [lib.string::append]
Status: active
Description:

Judy thinks the prepend interface(s) should look just like the append() interfaces described in 21.1.1.3.5 with the appropriate wording changes.

Resolution:
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?
Section: 21.1.3 [lib.basic.string]
Status: closed
Description:

The current form of the basic_string template is:

```
template<class charT, class Allocator, class traits>
```

The order of these template arguments should be changed to:

```
template<class charT, class traits, class Allocator>
```

because it is more common to change the traits, without changing the allocator. In this case, the default template arguments allow for a simpler declaration of the template, such as:

```
basic_string<wchar_t, my_traits>
```

rather than

```
basic_string<wchar_t, allocator, my_traits> .
```

The rationale mentioned in N0557 (the paper which added the Allocator as a template parameter) indicated that the Allocator is the second template parameter for similarity with other STL container template. However, this is not true. The set and map templates take the Allocator as the final template argument.

Resolution:

Amend the WP in section 21.1.1.3 [lib.basic.string] to change the order of the template arguments for basic_string as follows:

```
template<class charT, class traits = string_char_traits<charT>,
class Allocator = allocator>
class basic_string { // ...
```

Accepted at Austin meeting.

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

Issue Number: 21-007
Title: Should the string_char_traits speed-up functions be specified as inline?
Section: 21.1.1.1 [lib.string.char.traits]
Status: active
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.

The general consensus of library reflector messages was that inlining functions was an implementation detail and that functions could not be specified as inline in the Standard.

Resolution:
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: 21.1.1.1 [lib.string.char.traits]
Status: active
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."

As an additional note, only the operator**<<** is in the current WP (1 Feb. 1995). Both operator**<<** and operator**>>** were in the WP (20 Sept. 1994)

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

Issue Number: 21-009
Title: Why are character parameters passed as "const charT"?
Section: 21.1.1.4.{4-6} [lib.string::append], [lib.string::assign], [lib.string::insert]
Status: closed
Description:

The basic_string members append, assign, and insert have overloaded variations which take a single character as the final parameter. These parameters are specified as "const charT". Since these parameters are passed by value, the const qualification should be removed.

Resolution:

Amend to WP in section 21.1.1.4.{4-6} [lib.string::append], [lib.string::assign], [lib.string::insert] to remove the const qualifier from the members which take a character is the final parameter. The member functions should take the following form:

```
basic_string<charT,Allocator,traits>&
append(size_type n, charT c = charT());

basic_string<charT,Allocator,traits>&
assign(size_type pos, size_type n, charT c = charT());

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

Note: These const qualifiers were inadvertently introduced in N0557=94-0170.

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

Issue Number: 21-010
Title: Should member parameters passed as "const_pointer"?
Section: 21.1.3 [lib.basic.string]
Status: closed
Description:

In N0557=94-0170, basic_string was given typedefs for const_pointer, the pointer type supplied by the allocator. Along with this change, the type of pointer arguments return values was changed from const charT* to const_pointer, uniformly.

Unfortunately, this change prevents a string produced by a non-standard allocator from interfacing with C-style arrays of character objects. For example, the following code is no longer legal if the allocators const_pointer type is not const charT.:

```
basic_string<char, string_char_traits<char>, XAlloc> s("foo");
```

The const_pointer typedef should only be used when considering character objects which are part of an existing string, not for interfacing with C-style arrays of character objects.

Resolution:

Amend the WP as follows: Arguments to the functions which are currently of type "const_pointer" should be changed to "const charT*". Arguments to functions which are of type "pointer" should be changed to "charT*":

These member functions include:

```
basic_string(const charT* s, size_type n);
basic_string(const charT* s);
basic_string& operator=(const charT* s);
basic_string& operator+=(const charT* s);
basic_string& append(const charT* s, size_type n);
basic_string& append(const charT* s);
basic_string& assign(const charT* s, size_type n);
basic_string& assign(const charT* s);
basic_string& insert(const charT* s, size_type n);
basic_string& insert(const charT* s);
basic_string& replace(size_type pos, size_type n1,
                      const charT* s);
```

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

Also, change the return type of the data() and cstr() members as follows:

```
const charT* data() const;
const charT* cstr() const;
```

(Note: due to an editorial problem the current WP, these members have this interface in 21.1.1.4.11 [lib.string::cstr] and 21.1.1.4.12 [lib.string::data]. They have the const_pointer return type in 21.1.1.3 [lib.basic.string].

This change also applies to the appropriate algorithm members. They are not listed here for brevity.

Closed at Austin meeting.

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

Issue Number: 21-011
Title: Why are character parameters to the string traits functions passed by reference?
Section: 21.1.1.1 [lib.string.char.traits]
Status: closed
Description:

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

Resolution:

All character parameters will be passed by reference, see issue 21-012.

In section 21.1.1.1 [lib.string.char.traits], modify the specification for the string_char_traits members as follows:

```
static void assign(char_type& c1, char_type c2)
static bool eq(char_type c1, char_type c2)
static bool ne(char_type c1, char_type c2)
static bool lt(char_type c1, char_type c2)
```

Also, modify the following sections as appropriate: 21.1.1.2.{1-4} [lib.char.traits::assign], [lib.char.traits::eq], [lib.char.traits::ne], [lib.char.traits::lt].

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

Issue Number: 21-012
Title: Why are character parameters to the string functions passed by value?
Section: 21.1.1.1 [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 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: 21.1.1.4.19 [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() instead:

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

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

Issue Number: 21-015
Title: The copy() member should be const.
Section: 21.1.1.4.18 [lib.string::copy]
Status: active
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);
```

Proposed Resolution:

Remove the const qualifier from the copy member.

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

Issue Number: 21-016
Title: The error conditions are not well-specified for the find() and rfind() functions.
Section: 21.1.1.4.20 [lib.string::find]
Status: active
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 21.1.1.4.20 [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 precidence 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."

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

Issue Number: 21-017
Title: Can reserve() cause construction of characters?.
Section: 21.1.1.4.16 [lib.string::reserve]
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: 21.1.1.1 [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: 21-019

Title: The Allocator template parameter is not reflected in a member typedef.

Section: 21.1.1.3 [lib.basic.string]

Status: active

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."

Proposed Resolution:

Now that the parameter is a regular class type, it should be reflected in a member typedef:

```
typedef Allocator allocator_type;
```

in each standard container, and in basic_string as well.

Requester: Nathan Myers: myersn@roguewave.com

Owner:

Emails: lib-3593

Papers: (none)