Document number: N3021=10-0011
Date: 2010-02-10
Author: Martin Sebor
Contact: msebor@gmail.com
Organization: Cisco Systems, Inc.

Harmonizing *Effects* and *Returns* Elements in Clause 21

Introduction

Library issue 625 notes a general problem pervasive in Clause 21, the Strings Library: The descriptions of the semantic effects of a considerable number of `basic_string` member functions make use of what Clause 17, Library Introduction, refers as a *Returns* element, rather than the more appropriate *Effects* element. The distinction between the two elements is essential since their roles are quite different. As described in the `[structure.specifications]` section of Clause 17:

- A ***Returns*** element provides "*a description of the value(s) returned by the function.*"
- An ***Effects*** element describes "*the actions performed by the function.*"

By the way of an example, since function `f()` below is described using a *Returns* element, only the return value of the function for non-zero arguments is specified. Since function `g()` doesn't return a value when its argument is zero, the behavior of function `f()` when its argument is zero is unspecified. Specifically, whether `f()` throws an exception when its argument is zero, returns some unknown value, or calls `std::abort()`, is left unspecified.

`double f(double x);`

-1- *Returns*: `g(x)`.

`double g(double x);`

-2- *Throws:* `invalid_argument` if `x == 0`.

-3- *Returns*: `1 / x`.

This is obviously undesirable in general and in the case of many member functions of `basic_string` it is not intended. The correct way to specify function `f`'s semantics is either by duplicating the *Throws* element from `g()` or by making use of the *Effects* element.

In addition, a number of non-member functions are required by means of newly introduced *Remarks* elements that replaced the original *Notes*, to make use of the `traits::length()` function in cases where calling the function may be profitably avoided for better efficiency without changing the specified result.

This paper describes the necessary changes to the latest Draft Working Paper in order to fully specify the behavior of all `basic_string` functions without imposing unnecessary inefficiencies.

Overview of Changes

In the process of correcting the descriptions of the member functions a number of other, largely editorial issues were corrected:

- Inconsistent references to `basic_string`. The signatures as well as descriptions of some member functions specified the full set of template arguments when referring to the basic_string template (`basic_string<charT, traits, Allocator>`), others only a subset (`basic_string<charT, traits>`), and others omitted them altogether (`basic_string`). While the first and the last forms are equivalent, the second one isn't and it constituted a defect in the specification. The changes below use the last form consistently.
- Inconsistent treatment of `max_size()` versus `npos`. A small number of member functions specified that an exception of type `length_error` be thrown if the size of the resulting object were to exceed the value of `npos`, while others specified the same effect for `max_size()`. The changes adopted the latter for consistency.
- A handful of functions used the *Remarks* element to impose an additional requirement on the semantics of the functions: to make use of the `traits::length()` member function. Fully specifying the functions' effects via the *Effects* element made it possible to express the requirement in code and eliminate the *Remarks*.
- A small number of functions lacked a *Requirements* element expressing the implicit preconditions on the functions arguments

(such as a pointer argument referring to a character array). These were added where appropriate.

- When describing one function's Effects by reference to another function, the approach used was to always refer to a function with more specific arguments than those of the function being described. For example, the effects of function `f(const basic_string& str)` would be described in terms of `g(const char *s, size_type n)` like so:

  `void f(const basic_string& str);`

  -1- *Effects*: Calls (or is equivalent to) `g(str.data(), str.size())`.

  Never the other way around to avoid circular definitions. The phrase "*is equivalent to*" was used in cases where invoking the other function involved constructing a temporary `basic_string` object for notational convenience. The intent was to indicate that the temporary `basic_string` object is not required to be constructed. This practice is common throughout the rest of the standard.

Changes To The Draft Working Paper

In the following section, paragraph numbers refer to those in the Working Draft, Standard for Programming Language C++, document number N3000=09-0150. Additions to the text are denoted using an underlined typeface on green background, while deletions using a ~~strike-through typeface on light red background~~.

Changes to `[string::op+=]`

`basic_string~~<charT, traits, Allocator>~~&`
`operator+=(const basic_string~~<charT, traits, Allocator>~~& str);`

-?- *Effects*: Calls `append(str.data(), str.size())`.
-1- *Returns*: `*this` ~~`append(str)`~~.

`basic_string~~<charT, traits, Allocator>~~&`
`operator+=(const charT* s);`

-?- *Requires*: `s` points to an array of at least `traits::length(s)` elements of `charT`
-?- *Effects*: Calls `append(s, traits::length(s))`.
-2- *Returns*: `*this` ~~`+= basic_string<charT,traits,Allocator>(s)`~~.
~~-3- *Remarks*: Uses `traits::length()`.~~

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

-?- *Effects*: Calls `append(1, c)`.
-4- *Returns*: `*this` ~~`+= basic_string<charT,traits,Allocator>(1,c)`~~.

`basic_string&`
`operator+=(initializer_list<charT> il);`

-?- *Effects*: Calls `append(il.first(), il.size())`.
-5- *Returns*: `*this` ~~The result of `append(il)`~~.

Changes to `[string::append]`

`basic_string~~<charT, traits, Allocator>~~&`
`append(const basic_string~~<charT, traits>~~& str);`

-?- *Effects*: Calls `append(str, str.data(), str.size())`.
-1- *Returns*: `*this` ~~`append(str, 0, npos)`~~.

`basic_string~~<charT, traits, Allocator>~~&`
`append(const basic_string~~<charT, traits>~~& str, size_type pos, size_type n);`

-2- *Requires*: `pos <= str.size()`
-3- *Throws*: `out_of_range` if `pos > str.size()`.
-4- *Effects*: Determines the effective length `rlen` of the string to append as the smaller of `n` and `str.size()` `- pos` and calls `append(str.data() + pos, rlen)`. ~~The function then throws `length_error` if `size() >= npos`~~

~~rlen. Otherwise, the function replaces the string controlled by *this with a string of length size() + rlen whose first size() elements are a copy of the original string controlled by *this and whose remaining elements are a copy of the initial elements of the string controlled by str beginning at position pos.~~

-5- *Returns*: *this.

```
basic_string<charT, traits, Allocator>&
append(const charT* s, size_type n);
```

-?- *Requires*: s points to an array of at least n elements of charT

-?- *Throws*:

- out_of_range if pos > str.size().
- length_error if size() + n > max_size()

-?- *Effects*: The function replaces the string controlled by *this with a string of length size() + n whose first size() elements are a copy of the original string controlled by *this and whose remaining elements are a copy of the initial n elements of s.

-6- *Returns*: *this ~~append(basic_string<charT, traits, Allocator>(s,n))~~.

```
basic_string<charT, traits, Allocator>&
append(const charT* s);
```

-?- *Requires*: s points to an array of at least traits::length(s) + 1 elements of charT

-?- *Effects*: Calls append(s, traits::length(s)).

-7- *Returns*: *this ~~append(basic_string<charT, traits, Allocator>(s))~~.

~~-8- *Remarks*: Uses traits::length().~~

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

-?- *Effects*: Equivalent to append(basic_string(n, c)).

-9- *Returns*: *this ~~append(basic_string<charT ,traits, Allocator>(n, c))~~.

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

-?- *Requires*: [first, last) is a valid range.

-?- *Effects*: Equivalent to append(basic_string(first, last)).

-10- *Returns*: *this ~~append(basic_string<charT ,traits, Allocator>(first, last))~~.

```
basic_string&
append(initializer_list<charT> il);
```

-?- *Effects*: Calls append(il.begin(), il.size()).

-11- Returns: *this ~~append(basic_string(il))~~.

Changes to `[string::assign]`

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

-4- *Requires*: pos <= str.size()

-5- *Throws*: out_of_range if pos > str.size().

-6- *Effects*: Determines the effective length rlen of the string to assign as the smaller of n and str.size() - pos and calls assign(str.data() + pos, rlen).

~~The function then replaces the string controlled by *this with a string of length rlen whose elements are a copy of the string controlled by str beginning at position pos.~~

-7- *Returns*: *this.

```
basic_string<charT, traits, Allocator>&
assign(const charT* s, size_type n);
```

-?- *Requires*: s points to an array of at least n elements of charT

-?- *Throws*: length_error if size() + n > max_size()

-?- *Effects*: Replaces the string controlled by *this with a string of length n whose elements are a copy of those pointed to by s.

-8- *Returns*: *this ~~assign(basic_string<charT,traits,Allocator>(s,n))~~.

```
basic_string<charT, traits, Allocator>&
assign(const charT* s);
```

-?- *Requires*: s points to an array of at least `traits::length(s) + 1` elements of `charT`
-?- *Effects*: Calls `assign(s, traits::length(s))`.
-9- *Returns*: `*this` ~~`assign(basic_string<charT, traits, Allocator>(s))`~~.
~~10 *Remarks*: Uses `traits::length()`.~~

```
basic_string&
assign(initializer_list<charT> il);
```

-?- *Effects*: Calls `assign(il.begin(), il.size())`.
-11- *Returns*: `*this` ~~`assign(basic_string(il))`~~.

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

-?- *Effects*: Equivalent to `assign(basic_string(n, c))`.
-12- *Returns*: `*this` ~~`assign(basic_string<charT,traits,Allocator>(n,c))`~~.

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

-?- *Effects*: Equivalent to `assign(basic_string(first, last))`.
-13- *Returns*: `*this` ~~`assign(basic_string<charT,traits,Allocator>(first,last))`~~.

Changes to `[string::insert]`

```
basic_string<charT, traits, Allocator>&
insert(size_type pos1, const basic_string<charT, traits, Allocator>& str);
```

-?- *Requires*: `pos <= size()`
-?- *Throws*: `out_of_range` if `pos > size()`
-?- *Effects*: Calls `insert(pos, str.data(), str.size())`.
-1- *Returns*: `*this` ~~`insert(pos1, str, 0, npos)`~~.

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

-2- *Requires*: `pos1 <= size()` and `pos2 <= str.size()`.
-3- *Throws*: `out_of_range` if `pos1 > size()` or `pos2 > str.size()`.
-4- *Effects*: Determines the effective length `rlen` of the string to insert as the smaller of `n` and `str.size() - pos2` and calls `insert(pos1, str.data() + pos2, rlen)`. ~~Then throws `length_error` if `size() >= npos - rlen`.~~
~~Otherwise, the function replaces the string controlled by `*this` with a string of length `size() + rlen` whose first `pos1` elements are a copy of the initial elements of the original string controlled by `*this`, whose next `rlen` elements are a copy of the elements of the string controlled by `str` beginning at position `pos2`, and whose remaining elements are a copy of the remaining elements of the original string controlled by `*this`~~.
-5- *Returns*: `*this`.

```
basic_string<charT, traits, Allocator>&
insert(size_type pos, const charT* s, size_type n);
```

-?- *Requires*: `pos <= size()`.
-?- *Throws*:
- `out_of_range` if `pos > size()`
- `length_error` if `size() + n > max_size()`

-?- *Effects*: Replaces the string controlled by `*this` with a string of length `size() + n` whose first `pos` elements are a copy of the initial elements of the original string controlled by `*this`, whose next `n` elements are a copy of the elements in the array `s`, and whose remaining elements are a copy of the remaining elements of the original string controlled by `*this`.
-5- *Returns*: `*this` ~~`insert(pos, basic_string<charT, traits, Allocator>(s,n))`~~.

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

insert(size_type pos, const charT* s);

-?- *Requires*: pos <= size()
-?- *Throws*: out_of_range if pos > size()
-?- Effects: Calls insert(pos, s, traits::length(s)).
-7- *Returns*: *this. ~~insert(pos,basic_string<charT, traits, Allocator>(s)).~~
~~-8- *Remarks*: Uses traits::length().~~

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

-?- *Effects*: Equivalent to insert(pos, basic_string(n, c)).
-9- *Returns*: *this ~~insert(pos,basic_string<charT, traits, Allocator>(n, c))~~.

Changes to `[string::replace]`

basic_string~~<charT, traits, Allocator>~~&
replace(size_type pos~~1~~, size_type n~~1~~,
const basic_string~~<charT, traits, Allocator>~~& str);

-?- *Requires*: pos <= size()
-?- *Throws*: out_of_range if pos > size()
-?- *Effects*: Calls replace(pos, n, str.data(), str.size()).
-1- *Returns*: *this ~~replace(pos1, n1, str, 0, npos)~~.

basic_string~~<charT, traits, Allocator>~~&
replace(size_type pos1, size_type n1,
const basic_string~~<charT, traits, Allocator>~~& str,
size_type pos2, size_type n2);

-2- *Requires*: pos1 <= size() && pos2 <= str.size().
-3- *Throws*: out_of_range if pos1 > size() or pos2 > str.size()~~, or length_error if the length of the resulting string would exceed max_size() (see below)~~.
-4- *Effects*: Determines ~~the effective length xlen of the string to be removed as the smaller of n1 and size() - pos1. Also determines~~ the effective length rlen of the string to be inserted as the smaller of n2 and str.size() - pos2 and calls replace(pos1, n1, str.data() + pos2, rlen) ~~. If size() - xlen >= max_size() - rlen, throws length_error. Otherwise, the function replaces the string controlled by *this with a string of length size() - xlen + rlen whose first pos1 elements are a copy of the initial elements of the original string controlled by *this, whose next rlen elements are a copy of the initial elements of the string controlled by str beginning at position pos2, and whose remaining elements are a copy of the elements of the original string controlled by *this beginning at position pos1 + xlen~~.
-5- *Returns*: *this.

basic_string~~<charT, traits, Allocator>~~&
replace(size_type pos, size_type n1, const charT* s, size_type n2);

-?- *Requires*: pos <= size()
-?- *Throws*:
- out_of_range if pos > size()
- length_error if the length of the resulting string would exceed max_size() (see below).

-?- *Effects*: Determines the effective length xlen of the string to be removed as the smaller of n1 and size() - pos. If size() - xlen >= max_size() - n2, throws length_error. Otherwise, the function replaces the string controlled by *this with a string of length size() - xlen + n2 whose first pos elements are a copy of the initial elements of the original string controlled by *this, whose next n2 elements are a copy of the initial n2 elements of s, and whose remaining elements are a copy of the elements of the original string controlled by *this beginning at position pos + xlen.
-6- *Returns*: *this ~~replace(pos, n1, basic_string<charT, traits, Allocator>(s, n2))~~.

basic_string~~<charT, traits, Allocator>~~&
replace(size_type pos, size_type n~~1~~, const charT* s);

-?- *Requires*: pos <= size()
-?- *Throws*: out_of_range if pos > size()
-?- *Effects*: Calls replace(pos, n, s, traits::length(s)).
-7- *Returns*: *this ~~replace(pos,n1,basic_string<charT, traits, Allocator>(s))~~.
~~-8- *Remarks*: Uses traits::length().~~

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

-?- Effects: Equivalent to `replace(pos, n1, basic_string(n2, c))`.
-9- *Returns*: `*this` `replace(pos, n1, basic_string<charT, traits, Allocator>(n2, c))`.

basic_string&
replace(iterator i1, iterator i2, const basic_string& str);

-10- *Requires*: `[begin(), i1)` and `[i1, i2)` are valid ranges. The iterators `i1` and `i2` are valid iterators on `*this`, defining a range `[i1, i2)`.
-11- *Effects*: Calls `replace(i1 - begin(), i2 - i1, str)`. Replaces the string controlled by `*this` with a string of length `size() - (i2 - i1) + str.size()` whose first `i1 - begin()` elements are a copy of the initial elements of the original string controlled by `*this`, whose next `str.size()` elements are a copy of the string controlled by `str`, and whose remaining elements are a copy of the elements of the original string controlled by `*this` beginning at position `i2`.
-12- *Returns*: `*this`.
-13- *Remarks*: After the call, the length of the string will be changed by: `str.size() - (i2 - i1)`.

basic_string&
replace(iterator i1, iterator i2, const charT* s, size_type n);

-??- *Requires*: `[begin(), i1)` and `[i1, i2)` are valid ranges.
-??- *Effects*: Calls `replace(i1 - begin(), i2 - i1, s, n)`.
-14- *Returns*: `*this` `replace(i1, i2, basic_string(s, n))`.
-15- *Remarks*: After the call, the length of the string will be changed by Length change: `n - (i2 - i1)`.

basic_string&
replace(iterator i1, iterator i2, const charT* s);

-??- *Requires*: `[begin(), i1)` and `[i1, i2)` are valid ranges.
-??- *Effects*: Calls `replace(i1 - begin(), i2 - i1, s, traits::length(s))`.
-16- *Returns*: `*this` `replace(i1, i2, basic_string(s))`.
-17- *Remarks*: After the call, the length of the string will be changed by Length change: `traits::length(s) - (i2 - i1)`.
Uses `traits::length()`.

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

-??- *Requires*: `[begin(), i1)` and `[i1, i2)` are valid ranges.
-??- *Effects*: Equivalent to `replace(i1 - begin(), i2 - i1, basic_string(n, c))`.
-18- *Returns*: `*this` `replace(i1, i2, basic_string(n, c))`.
-19- *Remarks*: After the call, the length of the string will be changed by Length change: `n - (i2 - i1)`.

template<class InputIterator>
basic_string&
replace(iterator i1, iterator i2, InputIterator j1, InputIterator j2);

-??- *Requires*: `[begin(), i1)`, `[i1, i2)` and `[j1, j2)` are valid ranges.
-??- *Effects*: Equivalent to `replace(i1 - begin(), i2 - i1, basic_string(j1, j2))`.
-20- *Returns*: `*this` `replace(i1, i2, basic_string(j1, j2))`.
-21- *Remarks*: After the call, the length of the string will be changed by Length change: `j2 - j1 - (i2 - i1)`.

basic_string&
replace(iterator i1, iterator i2, initializer_list<charT> il);

-??- *Requires*: `[begin(), i1)` and `[i1, i2)` are valid ranges.
-??- *Effects*: Calls `replace(i1 - begin(), i2 - i1, il.begin(), il.size())`.
-22- *Returns*: `*this` `replace(i1, i2, il.begin(), il.end())`.

Changes to `[string::find]`

Note: Although the standard specifies the return value of all overloads of `find()` in terms of `find(const basic_string&, size_type)`, an implementation in which some or all overloads call `find(const charT*, size_type, size_type)`

instead is more efficient since it can avoid calling `traits::length()`. Such efficient implementations conform to C++ 2003 but would be rendered non-conforming to C++ 1x due to the change from non-normative *Notes* to normative *Remarks* introduced during the development of the Draft Working Paper. The changes below restore the permission to implement this optimization.

size_type find(const charT* s, size_type pos = 0) const;

-5- *Returns*: find(basic_string~~<charT, traits, Allocator>~~(s), pos).
~~-6- *Remarks*: Uses traits::length().~~

Changes to *[string::rfind]*

size_type rfind(const charT* s, size_type pos = 0) const;

-5- *Returns*: rfind(basic_string~~<charT, traits, Allocator>~~(s), pos).
~~-6- *Remarks*: Uses traits::length().~~

Changes to *[string::find.first.of]*

size_type find_first_of(const charT* s, size_type pos = 0) const;

-5- *Returns*: find_first_of(basic_string~~<charT, traits, Allocator>~~(s), pos).
~~-6- *Remarks*: Uses traits::length().~~

Changes to *[string::find.last.of]*

size_type find_last_of(const charT* s, size_type pos = 0) const;

-5- *Returns*: find_last_of(basic_string~~<charT, traits, Allocator>~~(s), pos).
~~-6- *Remarks*: Uses traits::length().~~

Changes to *[string::find.first.not.of]*

size_type find_first_not_of(const charT* s, size_type pos = 0) const;

-5- *Returns*: find_first_not_of(basic_string~~<charT, traits, Allocator>~~(s), pos).
~~-6- *Remarks*: Uses traits::length().~~

Changes to *[string::find.last.not.of]*

size_type find_last_not_of(const charT* s, size_type pos = 0) const;

-5- *Returns*: find_last_not_of(basic_string~~<charT, traits, Allocator>~~(s), pos).
~~-6- *Remarks*: Uses traits::length().~~

Changes to *[string::operator==]*

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

-3- *Returns*: lhs.compare(rhs) == 0 ~~lhs == basic_string<charT, traits, Allocator>(rhs)~~.
~~-4- *Remarks*: Uses traits::length().~~

Changes to *[string::op!=]*

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

-3- *Returns*: lhs.compare(rhs) != 0 ~~lhs != basic_string<charT, traits, Allocator>(rhs)~~.
~~-4- *Remarks*: Uses traits::length().~~