N2409 Proposed Resolutions for the Outstanding Issues in Chapter 28: Regular expressions library

John Maddock

Document: N2409=07-0269.
Date: 2007-09-7

Table of Contents

Issues With Proposed Resolutions 1
- Issue 524. regex named character classes and case-insensitivity don't mix 1
- Issue 650. regex_token_iterator and const correctness 3
- Issue 651. Missing preconditions for regex_token_iterator c'tors 3
- Issue 652. regex_iterator and const correctness 3
- Issue 682. basic_regex ctor takes InputIterator or ForwardIterator? 3
- Issue 684. Unclear which members of match_results should be used in comparison 4
- Issue 645. Missing members in match_results 5
- Issue 681. Operator functions impossible to compare are defined in [re.submatch.op] 7

Issues Still With no Proposed Resolution 11
- Issue 523. regex case-insensitive character ranges are unimplementable as specified 11

Issues With Proposed Resolutions

Issue 524. regex named character classes and case-insensitivity don't mix

Previous Discussion

This defect is also being discussed on the Boost developers list. The full discussion can be found here: http://lists.boost.org/boost/2005/07/29546.php

-- Begin original message --

Also, I may have found another issue, closely related to the one under discussion. It regards case-insensitive matching of named character classes. The regex_traits<> provides two functions for working with named char classes: lookup_classname and isctype. To match a char class such as [[:alpha:]], you pass "alpha" to lookup_classname and get a bitmask. Later, you pass a char and the bitmask to isctype and get a bool yes/no answer.

But how does case-insensitivity work in this scenario? Suppose we're doing a case-insensitive match on [[:lower:]]. It should behave as if it were [[:lower:][:upper:]], right? But there doesn't seem to be enough smarts in the regex_traits interface to do this.

Imagine I write a traits class which recognizes [[:fubar:]], and the "fubar" char class happens to be case-sensitive. How is the regex engine to know that? And how should it do a case-insensitive match of a character against the [[:fubar:]] char class? John, can you confirm this is a legitimate problem?

I see two options:

1) Add a bool icase parameter to lookup_classname. Then, lookup_classname("upper", true) will know to return lower|upper instead of just upper.

2) Add a isctype_nocase function

I prefer (1) because the extra computation happens at the time the pattern is compiled rather than when it is executed.

-- End original message --

Further Comments

This is relatively trivial to work around for [[:lower:]] and [[:upper:]] - you simply check whether your initial bitmask contains the same mask as returned by [[:lower:]] or [[:upper:]], and if it does, then bitwise OR your mask with the one for [[:alpha:]].

However, this strategy doesn't work for more advanced character classes such as [[:Lu:]] in Unicode for example. Eric's fix appears to be the correct approach for this.

Proposed Resolution

In 28.2 change the table entry:

Expression	Return Type	Notes
v.lookup_classname(F1,F2)	X::char_class_type	Converts the character sequence designated by the iterator range [F1,F2) into a value of a bitmask type that can subsequently be passed to isctype. Values returned from lookup_classname can be bitwise or'ed together; the resulting value represents membership in either of the corresponding character classes. Returns 0 if the character sequence is not the name of a character class recognized by X. The value returned shall be independent of the case of the characters in the sequence.

To:

Expression	Return Type	Notes
v.lookup_classname(F1,F2, b)	X::char_class_type	Converts the character sequence designated by the iterator range [F1,F2) into a value of a bitmask type that can subsequently be passed to isctype. Values returned from lookup_classname can be bitwise or'ed together; the resulting value represents membership in either of the corresponding character classes. If b is true, then returns a bitmask suitable for matching characters without regard to their case. Returns 0 if the character sequence is not the name of a character class recognized by X. The value returned shall be independent of the case of the characters in the sequence.

In 28.7 change:

```
template <class ForwardIterator>
char_class_type lookup_classname(
ForwardIterator first, ForwardIterator last) const;
```

to

```
template <class ForwardIterator>
char_class_type lookup_classname(
ForwardIterator first, ForwardIterator last, bool icase = false) const;
```

In 28.7 para 7 change:

```
template <class ForwardIterator>
char_class_type lookup_classname(
ForwardIterator first, ForwardIterator last) const;
```

Returns: an unspecified value that represents the character classification named by the character sequence designated by the iterator range [first, last). The value returned shall be independent of the case of the characters in the character sequence. If the name is not recognized then returns a value that compares equal to 0.

To:

```
template <class ForwardIterator>
char_class_type lookup_classname(
ForwardIterator first, ForwardIterator last, bool icase = false) const;
```

Returns: an unspecified value that represents the character classification named by the character sequence designated by the iterator range [first, last). If the parameter icase is true then the returned mask identifies the character classification without regard to the case of the characters being matched, otherwise it does honour the case of the character being matched. (Insert footnote 1 here) The value returned shall be independent of the case of the characters in the character sequence. If the name is not recognized then returns a value that compares equal to 0.

Added Footnote 1: For example if the parameter *icase* is true then [[:lower:]] is regarded the same as [[:alpha:]].

In 28.13 para 14 Replace:

During matching of a regular expression finite state machine against a sequence of characters, a character c is a member of a character class designated by an iterator range [first, last) if traits_inst.isctype(c, traits_inst.lookup_classname(first, last)) is true.

With:

During matching of a regular expression finite state machine against a sequence of characters, a character c is a member of a character class designated by an iterator range [first, last) if traits_inst.isctype(c, traits_inst.lookup_classname(first, last, flags() & icase)) is true.

Issue 650. regex_token_iterator and const correctness

The currently proposed resolution to issue 650 appears to be correct.

Issue 651. Missing preconditions for regex_token_iterator c'tors

The currently proposed resolution to issue 651 appears to be correct.

Issue 652. regex_iterator and const correctness

The currently proposed resolution to issue 652 appears to be correct.

Issue 682. basic_regex ctor takes InputIterator or ForwardIterator?

Existing Discussion

Looking at N2284, 28.8 [re.regex], p3 basic_regex class template synopsis shows this constructor:

```
template <class InputIterator>
     basic_regex(InputIterator first, InputIterator last,
                 flag_type f = regex_constants::ECMAScript);
```

In 28.8.2 [re.regex.construct], p15, the constructor appears with this signature:

```
template <class ForwardIterator>
     basic_regex(ForwardIterator first, ForwardIterator last,
                 flag_type f = regex_constants::ECMAScript);
```

ForwardIterator is probably correct, so the synopsis is wrong.

John adds:

I think either could be implemented? Although an input iterator would probably require an internal copy of the string being made.

I have no strong feelings either way, although I think my original intent was InputIterator.

Further Comments

I've double checked and Boost.Regex implements ForwardIterator, likewise the corresponding constructor in `basic_string` is templated on ForwardIterator. Eric is therefore correct, and we should do the same here.

Proposed Resolution

In 28.8 para 3 change:

```
template <class InputIterator>
basic_regex(InputIterator first, InputIterator last,
flag_type f = regex_constants::ECMAScript);
```

To:

```
template <class ForwardIterator>
basic_regex(ForwardIterator first, ForwardIterator last,
flag_type f = regex_constants::ECMAScript);
```

Issue 684. Unclear which members of match_results should be used in comparison

Previous Discussion

In 28.4 [re.syn] of N2284, two template functions are declared here:

```
// 28.10, class template match_results:
  <snip>
// match_results comparisons
  template <class BidirectionalIterator, class Allocator>
    bool operator== (const match_results<BidirectionalIterator, Allocator>& m1,
                     const match_results<BidirectionalIterator, Allocator>& m2);
  template <class BidirectionalIterator, class Allocator>
    bool operator!= (const match_results<BidirectionalIterator, Allocator>& m1,
                     const match_results<BidirectionalIterator, Allocator>& m2);

// 28.10.6, match_results swap:
```

But the details of these two bool operator functions (i.e., which members of match_results should be used in comparison) are not described in any following sections.

John adds:

That looks like a bug: operator== should return true only if the two objects refer to the same match - i.e. if one object was constructed as a copy of the other.

Proposed Resolution

Add a new section after 28.10.6, which reads:

28.10.7 match_results non-member functions.

```
template <class BidirectionalIterator, class Allocator>
bool operator== (const match_results<BidirectionalIterator, Allocator>& m1,
const match_results<BidirectionalIterator, Allocator>& m2);
```

Returns: true only if the two objects refer to the same match.

```
template <class BidirectionalIterator, class Allocator>
bool operator!= (const match_results<BidirectionalIterator, Allocator>& m1,
const match_results<BidirectionalIterator, Allocator>& m2);
```

Returns: `!(m1 == m2)`.

```
template <class BidirectionalIterator, class Allocator>
void swap(match_results<BidirectionalIterator, Allocator>& m1,
match_results<BidirectionalIterator, Allocator>& m2);
```

Returns: `m1.swap(m2)`.

Issue 645. Missing members in match_results

previous comment

According to the description given in 28.10 [re.results]/2 the class template match_results "shall satisfy the requirements of a Sequence, [..], except that only operations defined for const-qualified Sequences are supported". Comparing the provided operations from 28.10 [re.results]/3 with the sequence/container tables 80 and 81 one recognizes the following missing operations:

1) The members

```
const_iterator rbegin() const;
const_iterator rend() const;
```

should exists because 23.1/10 demands these for containers (all sequences are containers) which support bidirectional iterators. Aren't these supported by match_result? This is not explicitly expressed, but it's somewhat implied by two arguments:

(a) Several typedefs delegate to iterator_traits<BidirectionalIterator>.

(b) The existence of const_reference operator[](size_type n) const implies even random-access iteration. I also suggest, that match_results should explicitly mention, which minimum iterator category is supported and if this does not include random-access the existence of operator[] is somewhat questionable.

2) The new "convenience" members

```
const_iterator cbegin() const;
const_iterator cend() const;
const_iterator crbegin() const;
const_iterator crend() const;
```

should be added according to tables 80/81.

Further Comment

This report confuses two concepts:

- The type of iterator stored in the container (actually in the `sub_match` structure): this must be at least a BidirectionIterator.
- The type of iterator used to iterate over the container (this is currently unspecified).

In effect `match_results` is a *container of a structure containing Bidirectional Iterators*.

The container is not specified as *reversible* and so `rbegin()` and `rend()` members are not required, i.e. only table 65 applies and not table 66.

The presence of operator[] might imply that the container is an associative one - but none of the requirements for associative containers are particularly useful here other than operator[].

Likewise we don't want to specify that operator[] implies Random Access Iterators, since the container could be implemented internally using an associative data structure.

Therefore we rely on operator[] being an optional member of Sequences in Table 68: perhaps match_results should be mentioned here.

Item 2, the new convenience members should be added.

Proposed Resolution

Add match_results to the single row for operator[] in Table 68:

expression	return type	optional semantics	container
a[n]	reference; const_reference	*(a.begin() + n)	vector, deque, match_results

Add the following members to the match_results synopsis after `end()` in 28.10 para 3:

```
const_iterator cbegin() const;
const_iterator cend() const;
```

In section 28.10.3 change:

```
const_iterator begin()const;
```

Returns: A starting iterator that enumerates over all the sub-expressions stored in *this.

```
const_iterator end()const;
```

Returns: A terminating iterator that enumerates over all the sub-expressions stored in *this.

To:

```
const_iterator begin()const;
const_iterator cbegin()const;
```

Returns: A starting iterator that enumerates over all the sub-expressions stored in *this.

```
const_iterator end()const;
const_iterator cend()const;
```

Returns: A terminating iterator that enumerates over all the sub-expressions stored in *this.

Issue 681. Operator functions impossible to compare are defined in [re.submatch.op]

Previous Comment

In 28.9.2 [re.submatch.op] of N2284, operator functions numbered 31-42 seem impossible to compare. E.g.:

```
template <class BiIter>
    bool operator==(typename iterator_traits<BiIter>::value_type const& lhs,
                    const sub_match<BiIter>& rhs);
```

Returns: lhs == rhs.str().

When char* is used as BiIter, iterator_traits<BiIter>::value_type would be char, so that lhs == rhs.str() ends up comparing a char value and an object of std::basic_string<char>. However, the behaviour of comparison between these two types is not defined in 21.3.8 [string.nonmembers] of N2284. This applies when wchar_t* is used as BiIter.

Further Comment

This is a real issue, that was missed because of the way in which Boost.Regex is implemented: using the "as if" rule the exact comparisons given in the text aren't used, and as noted some are indeed invalid code. However, these comparisons are believed to be genuinely useful in practice.

Proposed Resolution

In 28.9.2 replace:

```
template <class BiIter>
bool operator==(typename iterator_traits<BiIter>::value_type const& lhs,
const sub_match<BiIter>& rhs);
```

31 Returns: lhs == rhs.str().

```
template <class BiIter>
bool operator!=(typename iterator_traits<BiIter>::value_type const& lhs,
const sub_match<BiIter>& rhs);
```

32 Returns: lhs != rhs.str().

```
template <class BiIter>
bool operator<(typename iterator_traits<BiIter>::value_type const& lhs,
const sub_match<BiIter>& rhs);
```

33 Returns: lhs < rhs.str().

```
template <class BiIter>
bool operator>(typename iterator_traits<BiIter>::value_type const& lhs,
const sub_match<BiIter>& rhs);
```

34 Returns: lhs > rhs.str().

```
template <class BiIter>
bool operator>=(typename iterator_traits<BiIter>::value_type const& lhs,
const sub_match<BiIter>& rhs);
```

35 Returns: lhs >= rhs.str().

```
template <class BiIter>
bool operator<=(typename iterator_traits<BiIter>::value_type const& lhs,
const sub_match<BiIter>& rhs);
```

36 Returns: lhs <= rhs.str().

```
template <class BiIter>
bool operator==(const sub_match<BiIter>& lhs,
typename iterator_traits<BiIter>::value_type const& rhs);
```

37 Returns: lhs.str() == rhs.

```
template <class BiIter>
bool operator!=(const sub_match<BiIter>& lhs,
typename iterator_traits<BiIter>::value_type const& rhs);
```

38 Returns: lhs.str() != rhs.

```
template <class BiIter>
bool operator<(const sub_match<BiIter>& lhs,
typename iterator_traits<BiIter>::value_type const& rhs);
```

39 Returns: lhs.str() < rhs.

```
template <class BiIter>
bool operator>(const sub_match<BiIter>& lhs,
typename iterator_traits<BiIter>::value_type const& rhs);
```

40 Returns: lhs.str() > rhs.

```
template <class BiIter>
bool operator>=(const sub_match<BiIter>& lhs,
typename iterator_traits<BiIter>::value_type const& rhs);
```

41 Returns: lhs.str() >= rhs.

```
template <class BiIter>
bool operator<=(const sub_match<BiIter>& lhs,
typename iterator_traits<BiIter>::value_type const& rhs);
```

42 Returns: lhs.str() <= rhs.

With:

```
template <class BiIter>
bool operator==(typename iterator_traits<BiIter>::value_type const& lhs,
const sub_match<BiIter>& rhs);
```

31 Returns: basic_string<typename iterator_traits<BiIter>::value_type>(1, lhs) == rhs.str().

```
template <class BiIter>
bool operator!=(typename iterator_traits<BiIter>::value_type const& lhs,
const sub_match<BiIter>& rhs);
```

32 Returns: basic_string<typename iterator_traits<BiIter>::value_type>(1, lhs) != rhs.str().

```
template <class BiIter>
bool operator<(typename iterator_traits<BiIter>::value_type const& lhs,
const sub_match<BiIter>& rhs);
```

33 Returns: basic_string<typename iterator_traits<BiIter>::value_type>(1, lhs) < rhs.str().

```
template <class BiIter>
bool operator>(typename iterator_traits<BiIter>::value_type const& lhs,
const sub_match<BiIter>& rhs);
```

34 Returns: basic_string<typename iterator_traits<BiIter>::value_type>(1, lhs) > rhs.str().

```
template <class BiIter>
bool operator>=(typename iterator_traits<BiIter>::value_type const& lhs,
const sub_match<BiIter>& rhs);
```

35 Returns: basic_string<typename iterator_traits<BiIter>::value_type>(1, lhs) >= rhs.str().

```
template <class BiIter>
bool operator<=(typename iterator_traits<BiIter>::value_type const& lhs,
const sub_match<BiIter>& rhs);
```

36 Returns: basic_string<typename iterator_traits<BiIter>::value_type>(1, lhs) <= rhs.str().

```
template <class BiIter>
bool operator==(const sub_match<BiIter>& lhs,
typename iterator_traits<BiIter>::value_type const& rhs);
```

37 Returns: lhs.str() == basic_string<typename iterator_traits<BiIter>::value_type>(1, rhs).

```
template <class BiIter>
bool operator!=(const sub_match<BiIter>& lhs,
typename iterator_traits<BiIter>::value_type const& rhs);
```

38 Returns: lhs.str() != basic_string<typename iterator_traits<BiIter>::value_type>(1, rhs).

```
template <class BiIter>
bool operator<(const sub_match<BiIter>& lhs,
typename iterator_traits<BiIter>::value_type const& rhs);
```

39 Returns: lhs.str() < basic_string<typename iterator_traits<BiIter>::value_type>(1, rhs).

```
template <class BiIter>
bool operator>(const sub_match<BiIter>& lhs,
typename iterator_traits<BiIter>::value_type const& rhs);
```

40 Returns: lhs.str() > basic_string<typename iterator_traits<BiIter>::value_type>(1, rhs).

```
template <class BiIter>
bool operator>=(const sub_match<BiIter>& lhs,
typename iterator_traits<BiIter>::value_type const& rhs);
```

41 Returns: lhs.str() >= basic_string<typename iterator_traits<BiIter>::value_type>(1, rhs).

```
template <class BiIter>
bool operator<=(const sub_match<BiIter>& lhs,
typename iterator_traits<BiIter>::value_type const& rhs);
```

42 Returns: lhs.str() <= basic_string<typename iterator_traits<BiIter>::value_type>(1, rhs).

Issues Still With no Proposed Resolution

Issue 523. regex case-insensitive character ranges are unimplementable as specified

There's been a great deal of discussion on this already, see Issue 523.

The basic problem is this:

How do we match a character range such as [a-z] in a case insensitive manner?

Currently the regex specification supports *case-folding*: so we can convert any character to it's case folded equivalent, but to use this to match a range we would need to enumerate each character in the range, and build an equivalence class containing all of the case folded equivalents. Then to determine whether a character *c* is in the range, we simply convert *c* to it's case folded equivalent and see if it's in the equivalence class or not.

However, this is expensive if the range `[x-y]` contains a lot of characters - if our intension is to support arbitrarily large character sets - or at least Unicode then this may be slow.

The possible options appear to be:

1. Do nothing: implementations are required to use the existing API. If the user provides a very large range, and wants case insensitivity as well, then they'll have to accept the poor performance in that case.

2. Provide an API in the traits class that allows us to enumerate for a given character *c* all the characters that are equivalent to it, then to test if a character *c* is in the range `[a-b]` we enumerate all the characters equivalent to *c* and return true if one of them is in the numeric range 'a' to 'b'. Unfortunately, there is no way to implement this for Unicode using the current ctype facet.

3. Either outlaw, or make implementation defined, the behaviour of case insensitive ranges if the two ends of the range are of different case, or come from different character blocks. Unfortunately, it's not clear whether this is implementable either: we can check that the two ends of the range have the same case, but not whether they come from the same block of characters. For example `[a-\xEF]` has both ends of the range the same case, but contains a mixture of uppercase, lowercase, and non-character code points.

This is a real issue, people are quite likely to want to specify large character ranges - for example `[\x0370-\x03FF]` - which encompass a complete language block (Greek in this case).

One option might be to do nothing at present - basically option 1 above - and seek further implementation (and user) experience.

Note that it is possible for an implementation to detect whether a traits class has any implementation-specific extensions (using SFINAE), and choose between implementation-specific or std-defined interfaces at compile time. This gives implementers the leeway to experiment with new regex algorithms, and/or fixes to tricky issues such as this, where there is no clear solution.