Document No: P1294R0
Revisions of: N3254=11-0024
Date: 2018-10-06
Authors :
Michael Spertus - mike_spertus@symantec.com
John Maddock - boost.regex@virgin.net
Audience: LWG

Proposed resolution for US104: Allocator-aware regular expressions (rev 3)

What's New?

Although this paper has languished for a while, I was pleased when I recently asked to update it for consideration as, in our opinion, the lack of allocator support in regular expressions has only become more problematic, and we have updated the use cases to reflect that.

I have also updated the proposal to include appropriate deduction guides, an explicit allocator constructor, and a pmr type alias

Rationale

The standard library contains many data structures that hold user data, such as containers, strings, string streams, and string buffers. These data structures are allocator-aware, using storage allocators to manage their storage requirements and internal pointer representations. However, regular expressions are not allocator-aware.

Such inconsistent treatment creates complexity by giving C++ programmers another inconsistency to remember. More importantly, the reasons for making the above data structures allocator-aware apply equally well to regular expressions. At Symantec, we have had to engineer around the lack of allocator support in regular expressions because `regex` objects cannot be placed in shared memory as they cannot be assigned a shared memory allocator. While the overall notion of allocators has generated some controversy within the C++ community, it seems clear that as long as other C++ containers are allocator-aware, regular expressions should be also.

Use Case 1: Shared memory regular expressions
At Symantec, we ran into the following problem that seems quite general. We had an application that ran in a number of processes. As one would naturally expect, the application used a configuration, which contained vectors and maps of strings and regular expressions.

This led us to consider how to share a configuration among multiple processes. This is unquestionably a common problem, and in the years since, we have found that such multi-process applications implementations have only become more common with the growth in core and need for memory isolation. If each process managed the configuration separately, they would have to setup complex configuration change listeners in separate thread that would reload the file and recompile the regular

expressions. A much better solution is to simply map a representation of the configuration in memory, using something like boost::interprocess. As long as the configuration is protected by an interprocess mutex, the configuration can always be used directly in shared memory by normal STL algorithms, string, and regular expression algorithms in place. In our experience, this turned out to be a fantastic solution to a common problem.

Use Case 2: Polymorphic Memory Resources

As there is a `std::pmr::match_results`, what better to use it with than a `std::pmr::regex`? This would allow all text processing to be done with the chosen allocator.

Approach

The proposed wording below was arrived at *mutatis mutandis* from the corresponding wording from `string`. However, a few comments are in order.

1. Class `match_results` currently uses an allocator. This allocator has no relationship to the allocator used internally in the regular expression, as has always been the case (regarding the current `regex` as regular expression using the standard allocator). Similar comments apply to string allocators.
2. Although most C++ containers consistently use pointer traits internally, regex_traits use locales, so they cannot be, say, shared using an interprocess allocator. Note that `basic_stringstream` and `basic_stringbuf` already use both allocators and locales, so supporting allocators in regular expressions do not introduce any new problems.

   Although these locale considerations prevent regular expressions using std::regex_traits from being shared between processes, there is no reason to prevent users from defining their own allocator-aware regex traits. To facilitate this, we require that if the regex_traits class is constructible from an allocator, then `std::basic_regex` initializes its traits object from the allocator. If the regex_traits class is not constructible from an allocator (e.g. `std::regex_traits`) then it is default constructed. We allow imbue to throw a regex_error (although std::regex_traits doesn't) as user-defined regex_traits may not be able to handle all possible locales (e.g., some custom locales).

Status

A working implementation has been implemented. In addition, an exemplary user-defined regex_traits was validated that allows locales to be passed between process based on their locale names.

Wording

In allocator.requirements/1, change the final sentence to

All of the string types (Clause 24), containers (Clause 26) (except array), string buffers and string streams (Clause 30), regular expressions (clause 31), and match_results (Clause 31) are parameterized in terms of allocators.

re.gen

Add the following paragraph

A deduction guide for `basic_regex` only participates in overload resolution if any `Allocator` argument satisfies the requirements for an allocator and any `ForwardIterator` argument satisfies the requirements for an iterator

re.req

In the entry for u.imbue in table 121 add: Report error by throwing exception of type `regex_error`

At the end of re.req/3, add the following paragraph.

Class template `basic_regex` satisfies the requirements for an Allocator-aware container (Table 78).

re.syn

...

```
// 31.8, class template basic_regex:
template <class charT, class traits = regex_traits<charT>,
class Allocator = allocator<charT> > class basic_regex;
```

…

```
// 31.8.6, basic_regex swap:
template <class charT, class traits, class Allocator> void
swap(basic_regex<charT, traits, Allocator>& e1,
basic_regex<charT, traits, Allocator>& e2);
```

…

```
// 31.11.2, function template regex_match:
template <class BidirectionalIterator, class AllocatorAllocMatch, class charT,
class traits, class Allocator>
  bool regex_match(BidirectionalIterator first,BidirectionalIterator
                   match_results<BidirectionalIterator, AllocatorAllocMatch>& m,
                   const basic_regex<charT, traits, Allocator>& e,
                   regex_constants::match_flag_type flags
                         = regex_constants::match_default);
template <class BidirectionalIterator, class charT, class traits, class Allocator>
  bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
                   const basic_regex<charT, traits, Allocator>& e,
                   regex_constants::match_flag_type flags
                         = regex_constants::match_default);
template <class charT, class  AllocatorAllocMatch, class traits, class Allocator>
  bool regex_match(const charT* str, class AllocatorAllocMatch
                   match_results<const charT*, AllocatorAllocMatch >& m,
                   const basic_regex<charT, traits, Allocator>& e,
                   regex_constants::match_flag_type flags
                         = regex_constants::match_default);
template <class ST, class SA, class AllocatorAllocMatch, class charT,
          class traits, class Allocator>
  bool regex_match(const basic_string<charT, ST, SA>& s,
                   match_results<
                     typename basic_string<charT, ST, SA>::const_iterator,
                     AllocatorAllocMatch>& m,
                   const basic_regex<charT, traits, Allocator>& e,
                   regex_constants::match_flag_type flags
                        = regex_constants::match_default);
template <class charT, class traits, class Allocator >
 bool regex_match(const charT* str,
```

```
                  const basic_regex<charT, traits, Allocator >& e,
                  regex_constants::match_flag_type flags
                    = regex_constants::match_default);
template <class ST, class SA, class charT, class traits, class Allocator >
 bool regex_match(const basic_string<charT, ST, SA>& s,
                  const basic_regex<charT, traits, Allocator>& e,
                  regex_constants::match_flag_type flags
                     =  regex_constants::match_default);
```

// 31.11.3, function template regex_search:

```
template <class BidirectionalIterator, class~~Allocator~~AllocMatch,
          class charT, class traits, class Allocator >
bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
match_results<BidirectionalIterator, ~~Allocator~~AllocMatch >& m,
                  const basic_regex<charT, traits, Allocator >& e,
                  regex_constants::match_flag_type flags
                     = regex_constants::match_default);
template <class BidirectionalIterator, class charT, class traits, class Allocator >
 bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
                   const basic_regex<charT, traits, Allocator >& e,
                   regex_constants::match_flag_type flags
                     = regex_constants::match_default);
template <class charT, class ~~Allocator~~AllocMatch, class traits, class Allocator >
 bool regex_search(const charT* str,
                  match_results<const charT*, ~~Allocator~~AllocMatch >& m,
                  const basic_regex<charT, traits, Allocator >& e,
                  regex_constants::match_flag_type flags
                     =  regex_constants::match_default);
template <class charT, class traits, class Allocator >
 bool regex_search(const charT* str,
                   const basic_regex<charT, traits, Allocator >& e,
                   regex_constants::match_flag_type flags
                      =  regex_constants::match_default);
template <class ST, class SA, class charT, class traits, class Allocator > bool
regex_search(const basic_string<charT, ST, SA>& s,
                   const basic_regex<charT, traits, Allocator >& e,
                   regex_constants::match_flag_type flags
                     =  regex_constants::match_default);
template <class ST, class SA, class ~~Allocator~~AllocMatch, class charT,
          class traits, class Allocator >
bool regex_search(const basic_string<charT, ST, SA>& s,
                  match_results<
                   typename basic_string<charT, ST, SA>::const_iterator,
                                 ~~Allocator~~AllocMatch >& m,
                   const basic_regex<charT, traits, Allocator >& e,
                    regex_constants::match_flag_type flags
                       = regex_constants::match_default);
```

// 31.11.4, function template regex_replace:

```
template <class OutputIterator, class BidirectionalIterator,
          class traits, class charT, class ST,
          class SA, class Allocator >
OutputIterator
  regex_replace(OutputIterator out,
                BidirectionalIterator first, BidirectionalIterator last,
                const basic_regex<charT, traits, Allocator >& e,
                const basic_string<charT, ST, SA>& fmt,
                regex_constants::match_flag_type flags
                  = regex_constants::match_default);
template <class OutputIterator, class BidirectionalIterator,
          class traits, class charT, class Allocator >
OutputIterator
  regex_replace(OutputIterator out,
                BidirectionalIterator first, BidirectionalIterator last,
```

```
                const basic_regex<charT, traits, Allocator >& e,
                const charT* fmt,
                regex_constants::match_flag_type flags
                 = regex_constants::match_default);
template <class traits, class charT, class ST, class SA,
          class FST, class FSA, class Allocator >
basic_string<charT, ST, SA>
  regex_replace(const basic_string<charT, ST, SA>& s,
                const basic_regex<charT, traits, Allocator >& e,
                const basic_string<charT, FST, FSA>& fmt,
                regex_constants::match_flag_type flags
                 = regex_constants::match_default);
template <class traits, class charT, class ST, class SA, class Allocator >
basic_string<charT, ST, SA>
  regex_replace(const basic_string<charT, ST, SA>& s,
                const basic_regex<charT, traits, Allocator >& e,
                const charT* fmt,
                regex_constants::match_flag_type flags
                  = regex_constants::match_default);
template <class traits, class charT, class ST, class SA, class Allocator >
basic_string<charT>
 regex_replace(const charT* s,
               const basic_regex<charT, traits, Allocator >& e,
               const basic_string<charT, ST, SA>& fmt,
               regex_constants::match_flag_type flags
                = regex_constants::match_default);
template <class traits, class charT, class Allocator >
basic_string<charT>
 regex_replace(const charT* s,
               const basic_regex<charT, traits, Allocator >& e,
               const charT* fmt,
               regex_constants::match_flag_type flags
                 = regex_constants::match_default);
…
```

re.err

Add an error type to regex_constants and table 124 called error_locale with Error Condition: Unable to imbue with a locale.

re.regex

…

```
namespace std {
template <class charT,
          class traits = regex_traits<charT>,
          class Allocator >
class basic_regex {
public:
       // types:
       using value_type = charT;
       using traits_type = traits;
       using string_type = typename traits::string_type;
       using flag_type = regex_constants::syntax_option_type;
       using locale_type = typename traits::locale_type;
       using allocator_type = Allocator;
       …
```

```
    // 31.8.2, construct/copy/destroy:
    basic_regex() noexcept(noexcept(Allocator()));
    explicit basic_regex(const Allocator &);
    explicit basic_regex(const charT* p,
                         flag_type f = regex_constants::ECMAScript,
                         const Allocator &a = Allocator());
    basic_regex(const charT* p, size_t len, flag_type f,
                const Allocator &a = Allocator());
    basic_regex(const basic_regex&); basic_regex(basic_regex&&);
    template <class ST, class SA>
      explicit basic_regex(const basic_string<charT, ST, SA>& p,
                           flag_type f = regex_constants::ECMAScript,
                           const Allocator &a = Allocator());
    template <class ForwardIterator>
      basic_regex(ForwardIterator first, ForwardIterator last,
                  flag_type f = regex_constants::ECMAScript,
                  const Allocator &a = Allocator());
    basic_regex(initializer_list<charT>,
                flag_type = regex_constants::ECMAScript,
                const Allocator &a = Allocator());

    // 31.8.10, allocator (Section number may be chosen editorially):
    allocator_type get_allocator() const noexcept;


  ...
  };
```

Editorial note: First guide in blue depends on P1021R1 partial specialization

```
template<class CharT, class Traits = std::regex_traits<CharT>, class Allocator>
explicit basic_regex(const Allocator) -> basic_regex<CharT, Traits, Allocator>;
template<class CharT, class Allocator>
explicit basic_regex(const CharT*, flag_type, Allocator)
  -> basic_regex<CharT, std::regex_traits<CharT>, Allocator>;
template<class CharT, class Allocator>
basic_regex(const CharT*, size_t, flag_type, Allocator)
  -> basic_regex<CharT, std::regex_traits<CharT>, Allocator>;
template <class CharT, class ST, class SA, class Allocator>
explicit basic_regex(const basic_string<charT, ST, SA>, flag_type, Allocator)
  -> basic_regex<CharT, std::regex_traits<CharT>, Allocator>;
template <class ForwardIterator, class Allocator>
basic_regex(ForwardIterator, ForwardIterator, flag_type, Allocator)
  -> basic_regex<typename std::iterator_traits<ForwardIterator>::value_type,
           std::regex_traits<typename std::iterator_traits<ForwardIterator>::value_type>,
           Allocator>;
template <class CharT, class Allocator>
basic_regex(initializer_list<CharT>, flag_type, Allocator)
  -> basic_regex<CharT, std:: regex_traits<CharT>, Allocator>;
```

```
namespace pmr {
  template <class charT, class traits = std::regex_traits<charT>>
  using basic_regex = std::basic_regex<chart, traits, pmr::allocator<charT>>;
  using regex = basic_regex<char>;
  using wregex = basic_regex<wchar_t>;
}
```

re.regex.construct

```
basic_regex() noexcept(noexcept(Allocator()));
basic_regex(const Allocator &a = Allocator);
```

Effects: Constructs an object of class basic_regex that does not match any character sequence.

```
basic_regex(const charT* p, flag_type f = regex_constants::ECMAScript,
            const Allocator &a = Allocator);
```

```
…
basic_regex(const charT* p, size_t len, flag_type f,
            const Allocator &a = Allocator);
…
template <class ST, class SA>
  basic_regex(const basic_string<charT, ST, SA>& s,
              flag_type f = regex_constants::ECMAScript,
              const Allocator &a = Allocator);
…
template <class ForwardIterator>
  basic_regex(ForwardIterator first, ForwardIterator last,
              flag_type f = regex_constants::ECMAScript
              const Allocator &a = Allocator);
…
basic_regex(initializer_list<charT> il,
            flag_type f = regex_constants::ECMAScript,
             const Allocator &a = Allocator);
```

31.8.5 basic_regex **locale [re.regex.locale]**

locale_type imbue(locale_type loc);

Effects: Returns the result of traits_inst.imbue(loc) where traits_inst is a (default initialized if is_convertible<Allocator, traits> has a base characteristic (20.7.1) of false and initialized from `get_allocator()` if `is_convertible<Allocator, traits>` has a base characteristic of true) instance of the template type argument traits stored within the object. After a call to imbue the basic_regex object does not match any character sequence.

locale_type getloc() const;

Effects: Returns the result of traits_inst.getloc() where traits_inst is a (default initialized if is_convertible<Allocator, traits> has a base characteristic (20.7.1) of false and initialized from `get_allocator()` if `is_convertible<Allocator, traits>` has a base characteristic of true) instance of the template parameter traits stored within the object.

31.8.8 re.regex.allocator

allocator_type get_allocator() const noexcept;

Returns: a copy of the Allocator object used to construct the basic_regex or, if that allocator has been replaced, a copy of the most recent replacement.

Note: The next few sections simply update signatures as in the header

28.11.2 regex_match [re.alg.match]

```
template <class BidirectionalIterator, class AllocatorAllocMatch,
          class charT, class traits, class Allocator>
  bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
                   match_results<BidirectionalIterator, AllocatorAllocMatch>& m,
                   const basic_regex<charT, traits, Allocator>& e,
                   regex_constants::match_flag_type flags
                     = regex_constants::match_default);
```

) (until after paragraph 3)

```
template <class BidirectionalIterator, class charT, class traits, class Allocator>
bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
```

```
                const basic_regex<charT, traits, Allocator>& e,
                regex_constants::match_flag_type flags
                  = regex_constants::match_default);
```

) (until after paragraph 4)

```
template <class charT, class AllocatorAllocMatch, class traits, class Allocator>
bool regex_match(const charT* str,
                 match_results<const charT*, AllocatorAllocMatch >& m,
const basic_regex<charT, traits, Allocator>& e,
regex_constants::match_flag_type flags =
regex_constants::match_default);
```

) (until after paragraph 5)

```
template <class ST, class SA, class AllocatorAllocMatch, class charT,
          class traits, class Allocator>
  bool regex_match(const basic_string<charT, ST, SA>& s,
                   match_results<
                     typename basic_string<charT, ST, SA>::const_iterator,
                     AllocatorAllocMatch >& m,
                  const basic_regex<charT, traits, Allocator>& e,
                  regex_constants::match_flag_type flags
                    = regex_constants::match_default);
```

) (until after paragraph 6)

```
template <class charT, class traits, class Allocator >
bool regex_match(const charT* str,
                   const basic_regex<charT, traits, Allocator >& e,
                   regex_constants::match_flag_type flags
                     = regex_constants::match_default);
```

) (until after paragraph 7)

```
 template <class ST, class SA, class charT, class traits, class Allocator
>   bool regex_match(const basic_string<charT, ST, SA>& s,
                 const basic_regex<charT, traits, Allocator>& e,
                 regex_constants::match_flag_type flags
                   = regex_constants::match_default);
```

re.alg.search

```
template <class BidirectionalIterator, class AllocatorAllocMatch,
          class charT, class traits, class Allocator >
bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
match_results<BidirectionalIterator, AllocatorAllocMatch>& m,
              const basic_regex<charT, traits, Allocator >& e,
              regex_constants::match_flag_type flags
                = regex_constants::match_default);
```

) (until after paragraph 3)

```
template <class charT, class AllocatorAllocMatch,
          class traits, class Allocator >
bool regex_search(const charT* str,
```

```
                match_results<const charT*, AllocatorAllocMatch >& m,
                const basic_regex<charT, traits, Allocator >& e,
                regex_constants::match_flag_type flags
                  =  regex_constants::match_default);
```

,(until after paragraph 4)

```
template <class ST, class SA, class charT, class traits, class Allocator >
bool regex_search(const basic_string<charT, ST, SA>& s,
                const basic_regex<charT, traits, Allocator >& e,
                regex_constants::match_flag_type flags
                  =  regex_constants::match_default);
```

,(until after paragraph 5)

```
template <class BidirectionalIterator, class charT, class traits, class Allocator >
bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
                const basic_regex<charT, traits, Allocator >& e,
                regex_constants::match_flag_type flags
                 = regex_constants::match_default);
```

,(until after paragraph 6)

```
template <class charT, class traits, class Allocator >
bool regex_search(const charT* str,
                const basic_regex<charT, traits, Allocator >& e,
                regex_constants::match_flag_type flags
                  = regex_constants::match_default);
```

,(until after paragraph 7)

```
template <class ST, class SA, class charT, class traits, class Allocator >
bool regex_search(const basic_string<charT, ST, SA>& s,
                const basic_regex<charT, traits, Allocator >& e,
                regex_constants::match_flag_type flags
                  = regex_constants::match_default);
```

re.alg.replace

```
template <class OutputIterator, class BidirectionalIterator,
class traits, class charT, class ST, class SA, class Allocator >
OutputIterator
  regex_replace(OutputIterator out,
                BidirectionalIterator first, BidirectionalIterator last,
const basic_regex<charT, traits, Allocator >& e,
                const basic_string<charT, ST, SA>& fmt,
                regex_constants::match_flag_type flags
                  = regex_constants::match_default);
template <class OutputIterator, class BidirectionalIterator,
          class traits, class charT, class Allocator >
OutputIterator
  regex_replace(OutputIterator out,
                BidirectionalIterator first, BidirectionalIterator last,
                const basic_regex<charT, traits, Allocator >& e,
                const charT* fmt,
                regex_constants::match_flag_type flags
                  = regex_constants::match_default);
```

)(until after paragraph 2)

```
template <class traits, class charT, class ST, class SA,
          class FST, class FSA, class Allocator >
basic_string<charT, ST, SA>
  regex_replace(const basic_string<charT, ST, SA>& s,
                const basic_regex<charT, traits, Allocator >& e,
                const basic_string<charT, FST, FSA>& fmt,
                regex_constants::match_flag_type flags
                 = regex_constants::match_default);
template <class traits, class charT, class ST, class SA, class Allocator >
basic_string<charT, ST, SA>
  regex_replace(const basic_string<charT, ST, SA>& s,
                const basic_regex<charT, traits, Allocator >& e,
                const charT* fmt,
                regex_constants::match_flag_type flags
                 = regex_constants::match_default);
```

,(until after paragraph 4)

```
template <class traits, class charT, class ST, class SA, class Allocator >
basic_string<charT>
  regex_replace(const charT* s,
                const basic_regex<charT, traits, Allocator >& e,
                const basic_string<charT, ST, SA>& fmt,
                regex_constants::match_flag_type flags
                 = regex_constants::match_default);
template <class traits, class charT, class Allocator >
basic_string<charT>
  regex_replace(const charT* s,
                const basic_regex<charT, traits, Allocator >& e,
                const charT* fmt,
                regex_constants::match_flag_type flags
                 = regex_constants::match_default);
```

re.grammar

)

Objects of type specialization of basic_regex store within themselves a ~~default-constructed~~n instance of their traits template parameter, henceforth referred to as traits_inst. It is default constructed if is_convertible<Allocator, traits> has a base characteristic (20.7.1) of false and constructed from `get_allocator()` if `is_convertible<Allocator, traits>` has a base characteristic of true This traits_inst object is used to support localization of the regular expression; basic_regex object member functions shall not call any locale dependent C or C++ API, including the formatted string input functions. Instead they shall call the appropriate traits member function to achieve the required effect.