The Boost Algorithm Library

Marshall Clow

Table of Contents

Description and Rationale

Boost.Algorithm is a collection of general purpose algorithms. While Boost contains many libraries of data structures, there is no single library for general purpose algorithms. Even though the algorithms are generally useful, many tend to be thought of as "too small" for Boost.

An implementation of Boyer-Moore searching, for example, might take a developer a week or so to implement, including test cases and documentation. However, scheduling a review to include that code into Boost might take several months, and run into resistance because "it is too small". Nevertheless, a library of tested, reviewed, documented algorithms can make the developer's life much easier, and that is the purpose of this library.

Future plans

I will be soliciting submissions from other developers, as well as looking through the literature for existing algorithms to include. The Adobe Source Library, for example, contains many useful algorithms that already have documentation and test cases. Knuth's The Art of Computer Programming is chock-full of algorithm descriptions, too.

My goal is to run regular algorithm reviews, similar to the Boost library review process, but with smaller chunks of code.

Dependencies

Boost.Algorithm uses Boost.Range, Boost.Assert, Boost.Array, Boost.TypeTraits, and Boost.StaticAssert.

Acknowledgements

Thanks to all the people who have reviewed this library and made suggestions for improvements. Steven Watanabe and Sean Parent, in particular, have provided a great deal of help.

Searching Algorithms

Boyer-Moore Search

Overview

The header file 'boyer_moore.hpp' contains an implementation of the Boyer-Moore algorithm for searching sequences of values.

The Boyer–Moore string search algorithm is a particularly efficient string searching algorithm, and it has been the standard benchmark for the practical string search literature. The Boyer-Moore algorithm was invented by Bob Boyer and J. Strother Moore, and published in the October 1977 issue of the Communications of the ACM , and a copy of that article is available at http://www.cs.utexas.edu/~moore/publications/fstrpos.pdf.

The Boyer-Moore algorithm uses two precomputed tables to give better performance than a naive search. These tables depend on the pattern being searched for, and give the Boyer-Moore algorithm larger a memory footprint and startup costs than a simpler algorithm, but these costs are recovered quickly during the searching process, especially if the pattern is longer than a few elements.

However, the Boyer-Moore algorithm cannot be used with comparison predicates like `std::search`.

Nomenclature: I refer to the sequence being searched for as the "pattern", and the sequence being searched in as the "corpus".

Interface

For flexibility, the Boyer-Moore algorithm has two interfaces; an object-based interface and a procedural one. The object-based interface builds the tables in the constructor, and uses operator () to perform the search. The procedural interface builds the table and does the search all in one step. If you are going to be searching for the same pattern in multiple corpora, then you should use the object interface, and only build the tables once.

Here is the object interface:

```
template <typename patIter>
class boyer_moore {
public:
    boyer_moore ( patIter first, patIter last );
    ~boyer_moore ();

    template <typename corpusIter>
    corpusIter operator () ( corpusIter corpus_first, corpusIter corpus_last );
    };
```

and here is the corresponding procedural interface:

```
template <typename patIter, typename corpusIter>
corpusIter boyer_moore_search (
        corpusIter corpus_first, corpusIter corpus_last,
        patIter pat_first, patIter pat_last );
```

Each of the functions is passed two pairs of iterators. The first two define the corpus and the second two define the pattern. Note that the two pairs need not be of the same type, but they do need to "point" at the same type. In other words, `patIter::value_type` and `curpusIter::value_type` need to be the same type.

The return value of the function is an iterator pointing to the start of the pattern in the corpus. If the pattern is not found, it returns the end of the corpus (`corpus_last`).

Performance

The execution time of the Boyer-Moore algorithm, while still linear in the size of the string being searched, can have a significantly lower constant factor than many other search algorithms: it doesn't need to check every character of the string to be searched, but rather skips over some of them. Generally the algorithm gets faster as the pattern being searched for becomes longer. Its efficiency derives from the fact that with each unsuccessful attempt to find a match between the search string and the text it is searching, it uses the information gained from that attempt to rule out as many positions of the text as possible where the string cannot match.

Memory Use

The algorithm allocates two internal tables. The first one is proportional to the length of the pattern; the second one has one entry for each member of the "alphabet" in the pattern. For (8-bit) character types, this table contains 256 entries.

Complexity

The worst-case performance to find a pattern in the corpus is *O(N)* (linear) time; that is, proportional to the length of the corpus being searched. In general, the search is sub-linear; not every entry in the corpus need be checked.

Exception Safety

Both the object-oriented and procedural versions of the Boyer-Moore algorithm take their parameters by value and do not use any information other than what is passed in. Therefore, both interfaces provide the strong exception guarantee.

Notes

- When using the object-based interface, the pattern must remain unchanged for during the searches; i.e, from the time the object is constructed until the final call to operator () returns.
- The Boyer-Moore algorithm requires random-access iterators for both the pattern and the corpus.

Customization points

The Boyer-Moore object takes a traits template parameter which enables the caller to customize how one of the precomputed tables is stored. This table, called the skip table, contains (logically) one entry for every possible value that the pattern can contain. When searching 8-bit character data, this table contains 256 elements. The traits class defines the table to be used.

The default traits class uses a `boost::array` for small 'alphabets' and a `tr1::unordered_map` for larger ones. The array-based skip table gives excellent performance, but could be prohibitively large when the 'alphabet' of elements to be searched grows. The unordered_map based version only grows as the number of unique elements in the pattern, but makes many more heap allocations, and gives slower lookup performance.

To use a different skip table, you should define your own skip table object and your own traits class, and use them to instantiate the Boyer-Moore object. The interface to these objects is described TBD.

Boyer-Moore-Horspool Search

Overview

The header file 'boyer_moore_horspool.hpp' contains an implementation of the Boyer-Moore-Horspool algorithm for searching sequences of values.

The Boyer-Moore-Horspool search algorithm was published by Nigel Horspool in 1980. It is a refinement of the Boyer-Moore algorithm that trades space for time. It uses less space for internal tables than Boyer-Moore, and has poorer worst-case performance.

The Boyer-Moore-Horspool algorithm cannot be used with comparison predicates like `std::search`.

Interface

Nomenclature: I refer to the sequence being searched for as the "pattern", and the sequence being searched in as the "corpus".

For flexibility, the Boyer-Moore-Horspool algorithm has two interfaces; an object-based interface and a procedural one. The object-based interface builds the tables in the constructor, and uses operator () to perform the search. The procedural interface builds the table and does the search all in one step. If you are going to be searching for the same pattern in multiple corpora, then you should use the object interface, and only build the tables once.

Here is the object interface:

```
template <typename patIter>
class boyer_moore_horspool {
public:
    boyer_moore_horspool ( patIter first, patIter last );
    ~boyer_moore_horspool ();

    template <typename corpusIter>
    corpusIter operator () ( corpusIter corpus_first, corpusIter corpus_last );
    };
```

and here is the corresponding procedural interface:

```
template <typename patIter, typename corpusIter>
corpusIter boyer_moore_horspool_search (
        corpusIter corpus_first, corpusIter corpus_last,
        patIter pat_first, patIter pat_last );
```

Each of the functions is passed two pairs of iterators. The first two define the corpus and the second two define the pattern. Note that the two pairs need not be of the same type, but they do need to "point" at the same type. In other words, `patIter::value_type` and `curpusIter::value_type` need to be the same type.

The return value of the function is an iterator pointing to the start of the pattern in the corpus. If the pattern is not found, it returns the end of the corpus (`corpus_last`).

Performance

The execution time of the Boyer-Moore-Horspool algorithm is linear in the size of the string being searched; it can have a significantly lower constant factor than many other search algorithms: it doesn't need to check every character of the string to be searched, but rather skips over some of them. Generally the algorithm gets faster as the pattern being searched for becomes longer. Its efficiency derives from the fact that with each unsuccessful attempt to find a match between the search string and the text it is searching, it uses the information gained from that attempt to rule out as many positions of the text as possible where the string cannot match.

Memory Use

The algorithm an internal table that has one entry for each member of the "alphabet" in the pattern. For (8-bit) character types, this table contains 256 entries.

Complexity

The worst-case performance is *O(m x n)*, where *m* is the length of the pattern and *n* is the length of the corpus. The average time is *O(n)*. The best case performance is sub-linear, and is, in fact, identical to Boyer-Moore, but the initialization is quicker and the internal loop is simpler than Boyer-Moore.

Exception Safety

Both the object-oriented and procedural versions of the Boyer-Moore-Horspool algorithm take their parameters by value and do not use any information other than what is passed in. Therefore, both interfaces provide the strong exception guarantee.

Notes

- When using the object-based interface, the pattern must remain unchanged for during the searches; i.e, from the time the object is constructed until the final call to operator () returns.

- The Boyer-Moore-Horspool algorithm requires random-access iterators for both the pattern and the corpus.

Customization points

The Boyer-Moore-Horspool object takes a traits template parameter which enables the caller to customize how the precomputed table is stored. This table, called the skip table, contains (logically) one entry for every possible value that the pattern can contain. When searching 8-bit character data, this table contains 256 elements. The traits class defines the table to be used.

The default traits class uses a `boost::array` for small 'alphabets' and a `tr1::unordered_map` for larger ones. The array-based skip table gives excellent performance, but could be prohibitively large when the 'alphabet' of elements to be searched grows. The unordered_map based version only grows as the number of unique elements in the pattern, but makes many more heap allocations, and gives slower lookup performance.

To use a different skip table, you should define your own skip table object and your own traits class, and use them to instantiate the Boyer-Moore-Horspool object. The interface to these objects is described TBD.

Knuth-Morris-Pratt Search

Overview

The header file 'knuth_morris_pratt.hpp' contains an implementation of the Knuth-Morris-Pratt algorithm for searching sequences of values.

The basic premise of the Knuth-Morris-Pratt algorithm is that when a mismatch occurs, there is information in the pattern being searched for that can be used to determine where the next match could begin, enabling the skipping of some elements of the corpus that have already been examined.

It does this by building a table from the pattern being searched for, with one entry for each element in the pattern.

The algorithm was conceived in 1974 by Donald Knuth and Vaughan Pratt, and independently by James H. Morris. The three published it jointly in 1977 in the SIAM Journal on Computing http://citeseer.ist.psu.edu/context/23820/0

However, the Knuth-Morris-Pratt algorithm cannot be used with comparison predicates like `std::search`.

Interface

Nomenclature: I refer to the sequence being searched for as the "pattern", and the sequence being searched in as the "corpus".

For flexibility, the Knuth-Morris-Pratt algorithm has two interfaces; an object-based interface and a procedural one. The object-based interface builds the table in the constructor, and uses operator () to perform the search. The procedural interface builds the table and does the search all in one step. If you are going to be searching for the same pattern in multiple corpora, then you should use the object interface, and only build the tables once.

Here is the object interface:

```
template <typename patIter>
class knuth_morris_pratt {
public:
    knuth_morris_pratt ( patIter first, patIter last );
    ~knuth_morris_pratt ();

    template <typename corpusIter>
    corpusIter operator () ( corpusIter corpus_first, corpusIter corpus_last );
    };
```

and here is the corresponding procedural interface:

```
template <typename patIter, typename corpusIter>
corpusIter knuth_morris_pratt_search (
        corpusIter corpus_first, corpusIter corpus_last,
        patIter pat_first, patIter pat_last );
```

Each of the functions is passed two pairs of iterators. The first two define the corpus and the second two define the pattern. Note that the two pairs need not be of the same type, but they do need to "point" at the same type. In other words, `patIter::value_type` and `curpusIter::value_type` need to be the same type.

The return value of the function is an iterator pointing to the start of the pattern in the corpus. If the pattern is not found, it returns the end of the corpus (`corpus_last`).

Performance

The execution time of the Knuth-Morris-Pratt algorithm is linear in the size of the string being searched. Generally the algorithm gets faster as the pattern being searched for becomes longer. Its efficiency derives from the fact that with each unsuccessful attempt to find a match between the search string and the text it is searching, it uses the information gained from that attempt to rule out as many positions of the text as possible where the string cannot match.

Memory Use

The algorithm an that contains one entry for each element the pattern, plus one extra. So, when searching for a 1026 byte string, the table will have 1027 entries.

Complexity

The worst-case performance is *O(2n)*, where *n* is the length of the corpus. The average time is *O(n)*. The best case performance is sub-linear.

Exception Safety

Both the object-oriented and procedural versions of the Knuth-Morris-Pratt algorithm take their parameters by value and do not use any information other than what is passed in. Therefore, both interfaces provide the strong exception guarantee.

Notes

- When using the object-based interface, the pattern must remain unchanged for during the searches; i.e, from the time the object is constructed until the final call to operator () returns.
- The Knuth-Morris-Pratt algorithm requires random-access iterators for both the pattern and the corpus. It should be possible to write this to use bidirectional iterators (or possibly even forward ones), but this implementation does not do that.

C++11 Algorithms

all_of

The header file 'boost/algorithm/cxx11/all_of.hpp' contains four variants of a single algorithm, `all_of`. The algorithm tests all the elements of a sequence and returns true if they all share a property.

The routine `all_of` takes a sequence and a predicate. It will return true if the predicate returns true when applied to every element in the sequence.

The routine `all_of_equal` takes a sequence and a value. It will return true if every element in the sequence compares equal to the passed in value.

Both routines come in two forms; the first one takes two iterators to define the range. The second form takes a single range parameter, and uses Boost.Range to traverse it.

interface

The function `all_of` returns true if the predicate returns true for every item in the sequence. There are two versions; one takes two iterators, and the other takes a range.

```
namespace boost { namespace algorithm {
template<typename InputIterator, typename Predicate>
 bool all_of ( InputIterator first, InputIterator last, Predicate p );
template<typename Range, typename Predicate>
 bool all_of ( const Range &r, Predicate p );
}}
```

The function `all_of_equal` is similar to `all_of`, but instead of taking a predicate to test the elements of the sequence, it takes a value to compare against.

```
namespace boost { namespace algorithm {
template<typename InputIterator, typename V>
 bool all_of_equal ( InputIterator first, InputIterator last, V const &val );
template<typename Range, typename V>
 bool all_of_equal ( const Range &r, V const &val );
}}
```

Examples

Given the container `c` containing `{ 0, 1, 2, 3, 14, 15 }`, then

```
bool isOdd ( int i ) { return i % 2 == 1; }
bool lessThan10 ( int i ) { return i < 10; }

using boost::algorithm;
all_of ( c, isOdd ) --> false
all_of ( c.begin (), c.end (), lessThan10 ) --> false
all_of ( c.begin (), c.begin () + 3, lessThan10 ) --> true
all_of ( c.end (), c.end (), isOdd ) --> true  // empty range
all_of_equal ( c, 3 ) --> false
all_of_equal ( c.begin () + 3, c.begin () + 4, 3 ) --> true
all_of_equal ( c.begin (), c.begin (), 99 ) --> true  // empty range
```

Iterator Requirements

`all_of` and `all_of_equal` work on all iterators except output iterators.

Complexity

All of the variants of `all_of` and `all_of_equal` run in *O(N)* (linear) time; that is, they compare against each element in the list once. If any of the comparisons fail, the algorithm will terminate immediately, without examining the remaining members of the sequence.

Exception Safety

All of the variants of `all_of` and `all_of_equal` take their parameters by value or const reference, and do not depend upon any global state. Therefore, all the routines in this file provide the strong exception guarantee.

Notes

- The routine `all_of` is part of the C++11 standard. When compiled using a C++11 implementation, the implementation from the standard library will be used.
- `all_of` and `all_of_equal` both return true for empty ranges, no matter what is passed to test against. When there are no items in the sequence to test, they all satisfy the condition to be tested against.
- The second parameter to `all_of_value` is a template parameter, rather than deduced from the first parameter (`std::iterator_traits<InputIterator>::value_type`) because that allows more flexibility for callers, and takes advantage of built-in comparisons for the type that is pointed to by the iterator. The function is defined to return true if, for all elements in the sequence, the expression `*iter == val` evaluates to true (where `iter` is an iterator to each element in the sequence)

any_of

The header file 'boost/algorithm/cxx11/any_of.hpp' contains four variants of a single algorithm, `any_of`. The algorithm tests the elements of a sequence and returns true if any of the elements has a particular property.

The routine `any_of` takes a sequence and a predicate. It will return true if the predicate returns true for any element in the sequence.

The routine `any_of_equal` takes a sequence and a value. It will return true if any element in the sequence compares equal to the passed in value.

Both routines come in two forms; the first one takes two iterators to define the range. The second form takes a single range parameter, and uses Boost.Range to traverse it.

interface

The function `any_of` returns true if the predicate returns true any item in the sequence. There are two versions; one takes two iterators, and the other takes a range.

```
namespace boost { namespace algorithm {
template<typename InputIterator, typename Predicate>
 bool any_of ( InputIterator first, InputIterator last, Predicate p );
template<typename Range, typename Predicate>
 bool any_of ( const Range &r, Predicate p );
}}
```

The function `any_of_equal` is similar to `any_of`, but instead of taking a predicate to test the elements of the sequence, it takes a value to compare against.

```
namespace boost { namespace algorithm {
template<typename InputIterator, typename V>
 bool any_of_equal ( InputIterator first, InputIterator last, V const &val );
template<typename Range, typename V>
 bool any_of_equal ( const Range &r, V const &val );
}}
```

Examples

Given the container `c` containing `{ 0, 1, 2, 3, 14, 15 }`, then

```
bool isOdd ( int i ) { return i % 2 == 1; }
bool lessThan10 ( int i ) { return i < 10; }

using boost::algorithm;
any_of ( c, isOdd ) --> true
any_of ( c.begin (), c.end (), lessThan10 ) --> true
any_of ( c.begin () + 4, c.end (), lessThan10 ) --> false
any_of ( c.end (), c.end (), isOdd ) --> false  // empty range
any_of_equal ( c, 3 ) --> true
any_of_equal ( c.begin (), c.begin () + 3, 3 ) --> false
any_of_equal ( c.begin (), c.begin (), 99 ) --> false  // empty range
```

Iterator Requirements

`any_of` and `any_of_equal` work on all iterators except output iterators.

Complexity

All of the variants of `any_of` and `any_of_equal` run in *O(N)* (linear) time; that is, they compare against each element in the list once. If any of the comparisons succeed, the algorithm will terminate immediately, without examining the remaining members of the sequence.

Exception Safety

All of the variants of `any_of` and `any_of_equal` take their parameters by value or const reference, and do not depend upon any global state. Therefore, all the routines in this file provide the strong exception guarantee.

Notes

- The routine `any_of` is part of the C++11 standard. When compiled using a C++11 implementation, the implementation from the standard library will be used.
- `any_of` and `any_of_equal` both return false for empty ranges, no matter what is passed to test against.
- The second parameter to `any_of_value` is a template parameter, rather than deduced from the first parameter (`std::iterator_traits<InputIterator>::value_type`) because that allows more flexibility for callers, and takes advantage of built-in comparisons for the type that is pointed to by the iterator. The function is defined to return true if, for any element in the sequence, the expression `*iter == val` evaluates to true (where `iter` is an iterator to each element in the sequence)

none_of

The header file 'boost/algorithm/cxx11/none_of.hpp' contains four variants of a single algorithm, `none_of`. The algorithm tests all the elements of a sequence and returns true if they none of them share a property.

The routine `none_of` takes a sequence and a predicate. It will return true if the predicate returns false when applied to every element in the sequence.

The routine `none_of_equal` takes a sequence and a value. It will return true if none of the elements in the sequence compare equal to the passed in value.

Both routines come in two forms; the first one takes two iterators to define the range. The second form takes a single range parameter, and uses Boost.Range to traverse it.

interface

The function `none_of` returns true if the predicate returns false for every item in the sequence. There are two versions; one takes two iterators, and the other takes a range.

```
namespace boost { namespace algorithm {
template<typename InputIterator, typename Predicate>
 bool none_of ( InputIterator first, InputIterator last, Predicate p );
template<typename Range, typename Predicate>
 bool none_of ( const Range &r, Predicate p );
}}
```

The function `none_of_equal` is similar to `none_of`, but instead of taking a predicate to test the elements of the sequence, it takes a value to compare against.

```
namespace boost { namespace algorithm {
template<typename InputIterator, typename V>
 bool none_of_equal ( InputIterator first, InputIterator last, V const &val );
template<typename Range, typename V>
 bool none_of_equal ( const Range &r, V const &val );
}}
```

Examples

Given the container `c` containing `{ 0, 1, 2, 3, 14, 15 }`, then

```
bool isOdd ( int i ) { return i % 2 == 1; }
bool lessThan10 ( int i ) { return i < 10; }

using boost::algorithm;

none_of ( c, isOdd ) --> false
none_of ( c.begin (), c.end (), lessThan10 ) --> false
none_of ( c.begin () + 4, c.end (), lessThan10 ) --> true
none_of ( c.end (), c.end (), isOdd ) --> true  // empty range
none_of_equal ( c, 3 ) --> false
none_of_equal ( c.begin (), c.begin () + 3, 3 ) --> true
none_of_equal ( c.begin (), c.begin (), 99 ) --> true  // empty range
```

Iterator Requirements

`none_of` and `none_of_equal` work on all iterators except output iterators.

Complexity

All of the variants of `none_of` and `none_of_equal` run in *O(N)* (linear) time; that is, they compare against each element in the list once. If any of the comparisons succeed, the algorithm will terminate immediately, without examining the remaining members of the sequence.

Exception Safety

All of the variants of `none_of` and `none_of_equal` take their parameters by value or const reference, and do not depend upon any global state. Therefore, all the routines in this file provide the strong exception guarantee.

Notes

- The routine `none_of` is part of the C++11 standard. When compiled using a C++11 implementation, the implementation from the standard library will be used.

- `none_of` and `none_of_equal` both return true for empty ranges, no matter what is passed to test against.
- The second parameter to `none_of_value` is a template parameter, rather than deduced from the first parameter (`std::iterator_traits<InputIterator>::value_type`) because that allows more flexibility for callers, and takes advantage of built-in comparisons for the type that is pointed to by the iterator. The function is defined to return true if, for all elements in the sequence, the expression `*iter == val` evaluates to false (where `iter` is an iterator to each element in the sequence)

one_of

The header file 'boost/algorithm/cxx11/one_of.hpp' contains four variants of a single algorithm, `one_of`. The algorithm tests the elements of a sequence and returns true if exactly one of the elements in the sequence has a particular property.

The routine `one_of` takes a sequence and a predicate. It will return true if the predicate returns true for one element in the sequence.

The routine `one_of_equal` takes a sequence and a value. It will return true if one element in the sequence compares equal to the passed in value.

Both routines come in two forms; the first one takes two iterators to define the range. The second form takes a single range parameter, and uses Boost.Range to traverse it.

interface

The function `one_of` returns true if the predicate returns true for one item in the sequence. There are two versions; one takes two iterators, and the other takes a range.

```
namespace boost { namespace algorithm {
template<typename InputIterator, typename Predicate>
 bool one_of ( InputIterator first, InputIterator last, Predicate p );
template<typename Range, typename Predicate>
 bool one_of ( const Range &r, Predicate p );
}}
```

The function `one_of_equal` is similar to `one_of`, but instead of taking a predicate to test the elements of the sequence, it takes a value to compare against.

```
namespace boost { namespace algorithm {
template<typename InputIterator, typename V>
 bool one_of_equal ( InputIterator first, InputIterator last, V const &val );
template<typename Range, typename V>
 bool one_of_equal ( const Range &r, V const &val );
}}
```

Examples

Given the container `c` containing `{ 0, 1, 2, 3, 14, 15 }`, then

```
bool isOdd ( int i ) { return i % 2 == 1; }
bool lessThan10 ( int i ) { return i < 10; }

using boost::algorithm;
one_of ( c, isOdd ) --> false
one_of ( c.begin (), c.end (), lessThan10 ) --> false
one_of ( c.begin () + 3, c.end (), lessThan10 ) --> true
one_of ( c.end (), c.end (), isOdd ) --> false  // empty range
one_of_equal ( c, 3 ) --> true
one_of_equal ( c.begin (), c.begin () + 3, 3 ) --> false
one_of_equal ( c.begin (), c.begin (), 99 ) --> false  // empty range
```

Iterator Requirements

`one_of` and `one_of_equal` work on all iterators except output iterators.

Complexity

All of the variants of `one_of` and `one_of_equal` run in *O(N)* (linear) time; that is, they compare against each element in the list once. If more than one of the elements in the sequence satisfy the condition, then algorithm will return false immediately, without examining the remaining members of the sequence.

Exception Safety

All of the variants of `one_of` and `one_of_equal` take their parameters by value or const reference, and do not depend upon any global state. Therefore, all the routines in this file provide the strong exception guarantee.

Notes

- `one_of` and `one_of_equal` both return false for empty ranges, no matter what is passed to test against.
- The second parameter to `one_of_equal` is a template parameter, rather than deduced from the first parameter (`std::iterator_traits<InputIterator>::value_type`) because that allows more flexibility for callers, and takes advantage of built-in comparisons for the type that is pointed to by the iterator. The function is defined to return true if, for one element in the sequence, the expression `*iter == val` evaluates to true (where `iter` is an iterator to each element in the sequence)

is_sorted

The header file `<boost/algorithm/cxx11/is_sorted.hpp>` contains functions for determining if a sequence is ordered.

is_sorted

The function `is_sorted(sequence)` determines whether or not a sequence is completely sorted according so some criteria. If no comparison predicate is specified, then std::less_equal is used (i.e, the test is to see if the sequence is non-decreasing)

```
namespace boost { namespace algorithm {
 template <typename Iterator, typename Pred>
 bool is_sorted ( Iterator first, Iterator last, Pred p );

 template <typename Iterator>
 bool is_sorted ( Iterator first, Iterator last );


 template <typename Range, typename Pred>
 bool is_sorted ( const Range &r, Pred p );

 template <typename Range>
 bool is_sorted ( const Range &r );
}}
```

Iterator requirements: The `is_sorted` functions will work on all kinds of iterators (except output iterators).

is_sorted_until

If `distance(first, last) < 2`, then `is_sorted ( first, last )` returns `last`. Otherwise, it returns the last iterator i in [first,last] for which the range [first,i) is sorted.

In short, it returns the element in the sequence that is "out of order". If the entire sequence is sorted (according to the predicate), then it will return `last`.

```
namespace boost { namespace algorithm {
 template <typename ForwardIterator, typename Pred>
 FI is_sorted_until ( ForwardIterator first, ForwardIterator last, Pred p );

 template <typename ForwardIterator>
 ForwardIterator is_sorted_until ( ForwardIterator first, ForwardIterator last );


 template <typename Range, typename Pred>
 typename boost::range_iterator<const R>::type is_sorted_until ( const Range &r, Pred p );

 template <typename Range>
 typename boost::range_iterator<const R>::type is_sorted_until ( const Range &r );
}}
```

Iterator requirements: The `is_sorted_until` functions will work on forward iterators or better. Since they have to return a place in the input sequence, input iterators will not suffice.

Complexity: `is_sorted_until` will make at most *N-1* calls to the predicate (given a sequence of length *N*).

Examples:

Given the sequence `{ 1, 2, 3, 4, 5, 3 }`, `is_sorted_until ( beg, end, std::less<int>())` would return an iterator pointing at the second `3`.

Given the sequence `{ 1, 2, 3, 4, 5, 9 }`, `is_sorted_until ( beg, end, std::less<int>())` would return `end`.

There are also a set of "wrapper functions" for is_ordered which make it easy to see if an entire sequence is ordered. These functions return a boolean indicating success or failure rather than an iterator to where the out of order items were found.

To test if a sequence is increasing (each element at least as large as the preceding one):

```
namespace boost { namespace algorithm {
 template <typename Iterator>
 bool is_increasing ( Iterator first, Iterator last );

 template <typename R>
 bool is_increasing ( const R &range );
}}
```

To test if a sequence is decreasing (each element no larger than the preceding one):

```
namespace boost { namespace algorithm {
 template <typename Iterator>
 bool is_decreasing ( Iterator first, Iterator last );

 template <typename R>
 bool is_decreasing ( const R &range );
}}
```

To test if a sequence is strictly increasing (each element larger than the preceding one):

```
namespace boost { namespace algorithm {
 template <typename Iterator>
 bool is_strictly_increasing ( Iterator first, Iterator last );

 template <typename R>
 bool is_strictly_increasing ( const R &range );
}}
```

To test if a sequence is strictly decreasing (each element smaller than the preceding one):

```
namespace boost { namespace algorithm {
 template <typename Iterator>
 bool is_strictly_decreasing ( Iterator first, Iterator last );

 template <typename R>
 bool is_strictly_decreasing ( const R &range );
}}
```

Complexity: Each of these calls is just a thin wrapper over `is_sorted`, so they have the same complexity as `is_sorted`.

Notes

- The routines `is_sorted` and `is_sorted_until` are part of the C++11 standard. When compiled using a C++11 implementation, the implementation from the standard library will be used.
- `is_sorted` and `is_sorted_until` both return true for empty ranges and ranges of length one.

is_partitioned

The header file 'is_partitioned.hpp' contains two variants of a single algorithm, `is_partitioned`. The algorithm tests to see if a sequence is partitioned according to a predicate; in other words, all the items in the sequence that satisfy the predicate are at the beginning of the sequence.

The routine `is_partitioned` takes a sequence and a predicate. It returns true if the sequence is partitioned according to the predicate.

`is_partitioned` come in two forms; the first one takes two iterators to define the range. The second form takes a single range parameter, and uses Boost.Range to traverse it.

interface

The function `is_partitioned` returns true if the items in the sequence are separated according to their ability to satisfy the predicate. There are two versions; one takes two iterators, and the other takes a range.

```
template<typename InputIterator, typename Predicate>
 bool is_partitioned ( InputIterator first, InputIterator last, Predicate p );
template<typename Range, typename Predicate>
 bool is_partitioned ( const Range &r, Predicate p );
```

Examples

Given the container c containing `{ 0, 1, 2, 3, 14, 15 }`, then

```
bool isOdd ( int i ) { return i % 2 == 1; }
bool lessThan10 ( int i ) { return i < 10; }

is_partitioned ( c, isOdd ) --> false
is_partitioned ( c, lessThan10 ) --> true
is_partitioned ( c.begin (), c.end (), lessThan10 ) --> true
is_partitioned ( c.begin (), c.begin () + 3, lessThan10 ) --> true
is_partitioned ( c.end (), c.end (), isOdd ) --> true  // empty range
```

Iterator Requirements

`is_partitioned` works on all iterators except output iterators.

Complexity

Both of the variants of `is_partitioned` run in *O(N)* (linear) time; that is, they compare against each element in the list once. If the sequence is found to be not partitioned at any point, the routine will terminate immediately, without examining the rest of the elements.

Exception Safety

Both of the variants of `is_partitioned` take their parameters by value or const reference, and do not depend upon any global state. Therefore, all the routines in this file provide the strong exception guarantee.

Notes

- The iterator-based version of the routine `is_partitioned` is part of the C++11 standard. When compiled using a C++11 implementation, the implementation from the standard library will be used.
- `is_partitioned` returns true for empty ranges, no matter what predicate is passed to test against.

is_permutation

The header file 'is_permutation.hpp' contains six variants of a single algorithm, `is_permutation`. The algorithm tests to see if one sequence is a permutation of a second one; in other words, it contains all the same members, possibly in a different order.

The routine `is_permutation` takes two sequences and an (optional) predicate. It returns true if the two sequences contain the same members. If it is passed a predicate, it uses the predicate to compare the elements of the sequence to see if they are the same.

`is_permutation` come in three forms. The first one takes two iterators to define the first range, and the starting iterator of the second range. The second form takes a two iterators to define the first range and two more to define the second range. The third form takes a single range parameter, and uses Boost.Range to traverse it.

Interface

The function `is_permutation` returns true if the two input sequences contain the same elements. There are six versions; two take three iterators, two take four iterators, and the other two take two ranges.

In general, you should prefer the four iterator versions over the three iterator ones. The three iterator version has to "create" the fourth iterator internally by calling `std::advance(first2, std::distance(first1,last1))`, and if the second sequence is shorter than the first, that's undefined behavior.

```
template< class ForwardIterator1, class ForwardIterator2 >
bool is_permutation ( ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2 );

template< class ForwardIterator1, class ForwardIterator2, class BinaryPredicate >
bool is_permutation ( ForwardIterator1 first1, ForwardIterator1 last1,
                      ForwardIterator2 first2, BinaryPredicate p );


template< class ForwardIterator1, class ForwardIterator2 >
bool is_permutation ( ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, For↵
wardIterator2 last2 );

template< class ForwardIterator1, class ForwardIterator2, class BinaryPredicate >
bool is_permutation ( ForwardIterator1 first1, ForwardIterator1 last1,
                      ForwardIterator2 first2, ForwardIterator2 last2,
                      BinaryPredicate p );

template <typename Range, typename ForwardIterator>
bool is_permutation ( const Range &r, ForwardIterator first2 );

template <typename Range, typename ForwardIterator, typename BinaryPredicate>
bool is_permutation ( const Range &r, ForwardIterator first2, BinaryPredicate pred );
```

Examples

Given the container `c1` containing `{ 0, 1, 2, 3, 14, 15 }`, and `c2` containing `{ 15, 14, 3, 1, 2 }`, then

```
is_permutation ( c1.begin(),     c1.end (), c2.begin(), c2.end ()) --> false
is_permutation ( c1.begin() + 1, c1.end (), c2.begin(), c2.end ()) --> true

is_permutation ( c1.end (), c1.end (), c2.end(), c2.end ()) --> true  // all empty ranges are ↵
permutations of each other
```

Iterator Requirements

`is_permutation` works on forward iterators or better.

Complexity

All of the variants of `is_permutation` run in *O(N^2)* (quadratic) time; that is, they compare against each element in the list (potentially) N times. If passed random-access iterators, `is_permutation` can return quickly if the sequences are different sizes.

Exception Safety

All of the variants of `is_permutation` take their parameters by value, and do not depend upon any global state. Therefore, all the routines in this file provide the strong exception guarantee.

Notes

- The three iterator versions of the routine `is_permutation` are part of the C++11 standard. When compiled using a C++11 implementation, the implementation from the standard library will be used.
- The four iterator versions of the routine `is_permutation` are part of the proposed C++14 standard. When C++14 standard libraries become available, the implementation should be changed to use the implementation from the standard library (if available).
- `is_permutation` returns true when passed a pair of empty ranges, no matter what predicate is passed to test with.

partition_point

The header file 'partition_point.hpp' contains two variants of a single algorithm, `partition_point`. Given a partitioned sequence and a predicate, the algorithm finds the partition point; i.e, the first element in the sequence that does not satisfy the predicate.

The routine `partition_point` takes a partitioned sequence and a predicate. It returns an iterator which 'points to' the first element in the sequence that does not satisfy the predicate. If all the items in the sequence satisfy the predicate, then it returns one past the final element in the sequence.

`partition_point` come in two forms; the first one takes two iterators to define the range. The second form takes a single range parameter, and uses Boost.Range to traverse it.

interface

There are two versions; one takes two iterators, and the other takes a range.

```
template<typename ForwardIterator, typename Predicate>
 ForwardIterator partition_point ( ForwardIterator first, ForwardIterator last, Predicate p );
template<typename Range, typename Predicate>
 boost::range_iterator<Range> partition_point ( const Range &r, Predicate p );
```

Examples

Given the container `c` containing `{ 0, 1, 2, 3, 14, 15 }`, then

```
bool lessThan10 ( int i ) { return i < 10; }
bool isOdd ( int i ) { return i % 2 == 1; }

partition_point ( c, lessThan10 ) --> c.begin () + 4  (pointing at 14)
partition_point ( c.begin (), c.end (), lessThan10 ) --> c.begin () + 4  (pointing at 14)
partition_point ( c.begin (), c.begin () + 3, lessThan10 ) -> c.begin () + 3 (end)
partition_point ( c.end (), c.end (), isOdd ) --> c.end ()  // empty range
```

Iterator Requirements

`partition_point` requires forward iterators or better; it will not work on input iterators or output iterators.

Complexity

Both of the variants of `partition_point` run in *O(log (N))* (logarithmic) time; that is, the predicate will be will be applied approximately *log(N)* times. To do this, however, the algorithm needs to know the size of the sequence. For forward and bidirectional iterators, calculating the size of the sequence is an *O(N)* operation.

Exception Safety

Both of the variants of `partition_point` take their parameters by value or const reference, and do not depend upon any global state. Therefore, all the routines in this file provide the strong exception guarantee.

Notes

- The iterator-based version of the routine `partition_point` is part of the C++11 standard. When compiled using a C++11 implementation, the implementation from the standard library will be used.
- For empty ranges, the partition point is the end of the range.

C++14 Algorithms

equal

The header file 'equal.hpp' contains two variants of a the stl algorithm `equal`. The algorithm tests to see if two sequences contain equal values;

Before (the proposed) C++14 the algorithm `std::equal` took three iterators and an optional comparison predicate. The first two iterators `[first1, last1)` defined a sequence, and the second one `first2` defined the start of the second sequence. The second sequence was assumed to be the same length as the first.

In C++14, two new variants were introduced, taking four iterators and an optional comparison predicate. The four iterators define two sequences `[first1, last1)` and `[first2, last2)` explicitly, rather than defining the second one implicitly. This leads to correct answers in more cases (and avoid undefined behavior in others).

Consider the two sequences:

```
auto seq1 = { 0, 1, 2 };
auto seq2 = { 0, 1, 2, 3, 4 };

std::equal ( seq1.begin (), seq1.end (), seq2.begin ()); // true
std::equal ( seq2.begin (), seq2.end (), seq1.begin ()); // Undefined behavior
std::equal ( seq1.begin (), seq1.end (), seq2.begin (), seq2.end ()); // false
```

You can argue that `true` is the correct answer in the first case, even though the sequences are not the same. The first N entries in `seq2` are the same as the entries in `seq1` - but that's not all that's in `seq2`. But in the second case, the algorithm will read past the end of `seq1`, resulting in undefined behavior (large earthquake, incorrect results, pregnant cat, etc).

However, if the two sequences are specified completely, it's clear that they are not equal.

interface

The function `equal` returns true if the two sequences compare equal; i.e, if each element in the sequence compares equal to the corresponding element in the other sequence. One version uses `std::equal_to` to do the comparison; the other lets the caller pass predicate to do the comparisons.

```
template <class InputIterator1, class InputIterator2>
bool equal ( InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, InputIterator2 last2 );

template <class InputIterator1, class InputIterator2, class BinaryPredicate>
bool equal ( InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, InputIterator2 last2, BinaryPredicate pred );
```

Examples

Given the container `c1` containing `{ 0, 1, 2, 3, 14, 15 }`, and `c2` containing `{ 1, 2, 3 }`, then

```
equal ( c1.begin (),     c1.end (),       c2.begin (), c2.end ()) --> false
equal ( c1.begin () + 1, c1.begin () + 3, c2.begin (), c2.end ()) --> true
equal ( c1.end (),       c1.end (),       c2.end (),   c2.end ()) --> true  // empty sequences ↵
are alway equal to each other
```

Iterator Requirements

`equal` works on all iterators except output iterators.

Complexity

Both of the variants of `equal` run in *O(N)* (linear) time; that is, they compare against each element in the list once. If the sequence is found to be not equal at any point, the routine will terminate immediately, without examining the rest of the elements.

Exception Safety

Both of the variants of `equal` take their parameters by value and do not depend upon any global state. Therefore, all the routines in this file provide the strong exception guarantee.

Notes

- The four iterator version of the routine `equal` is part of the C++14 standard. When C++14 standard library implementations become available, the implementation from the standard library should be used.
- `equal` returns true for two empty ranges, no matter what predicate is passed to test against.

mismatch

The header file 'mismatch.hpp' contains two variants of a the stl algorithm `mismatch`. The algorithm finds the first point in two sequences where they do not match.

Before (the proposed) C++14 the algorithm `std::mismatch` took three iterators and an optional comparison predicate. The first two iterators `[first1, last1)` defined a sequence, and the second one `first2` defined the start of the second sequence. The second sequence was assumed to be the same length as the first.

In C++14, two new variants were introduced, taking four iterators and an optional comparison predicate. The four iterators define two sequences `[first1, last1)` and `[first2, last2)` explicitly, rather than defining the second one implicitly. This leads to correct answers in more cases (and avoid undefined behavior in others).

Consider the two sequences:

```
auto seq1 = { 0, 1, 2 };
auto seq2 = { 0, 1, 2, 3, 4 };

std::mismatch ( seq1.begin (), seq1.end (), seq2.begin ()); // <3, 3>
std::mismatch ( seq2.begin (), seq2.end (), seq1.begin ()); // Undefined behavior
std::mismatch ( seq1.begin (), seq1.end (), seq2.begin (), seq2.end ()); // <3, 3>
```

The first N entries in `seq2` are the same as the entries in `seq1` - but that's not all that's in `seq2`. In the second case, the algorithm will read past the end of `seq1`, resulting in undefined behavior (large earthquake, incorrect results, pregnant cat, etc).

However, if the two sequences are specified completely, it's clear that where the mismatch occurs.

interface

The function `mismatch` returns a pair of iterators which denote the first mismatching elements in each sequence. If the sequences match completely, `mismatch` returns their end iterators. One version uses `std::equal_to` to do the comparison; the other lets the caller pass predicate to do the comparisons.

```
template <class InputIterator1, class InputIterator2>
std::pair<InputIterator1, InputIterator2>
mismatch ( InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2, InputIterator2 last2 );

template <class InputIterator1, class InputIterator2, class BinaryPredicate>
std::pair<InputIterator1, InputIterator2>
mismatch ( InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2, InputIterator2 last2, BinaryPredicate pred );
```

Examples

Given the container `c1` containing `{ 0, 1, 2, 3, 14, 15 }`, and `c2` containing `{ 1, 2, 3 }`, then

```
mismatch ( c1.begin(),     c1.end(),       c2.begin(), c2.end()) --> <c1.begin(), c2.begin()> // ↵
first elements do not match
mismatch ( c1.begin() + 1, c1.begin() + 4, c2.begin(), c2.end()) --> <c1.be↵
gin() + 4, c2.end ()> // all elements of `c2` match
mismatch ( c1.end(),       c1.end(),       c2.end(),   c2.end()) --> <c1.end(), c2.end()> // ↵
empty sequences don't match at the end.
```

Iterator Requirements

`mismatch` works on all iterators except output iterators.

Complexity

Both of the variants of `mismatch` run in *O(N)* (linear) time; that is, they compare against each element in the list once. If the sequence is found to be equal at any point, the routine will terminate immediately, without examining the rest of the elements.

Exception Safety

Both of the variants of `mismatch` take their parameters by value and do not depend upon any global state. Therefore, all the routines in this file provide the strong exception guarantee.

Notes

- If the sequences are equal (or both are empty), then mismatch returns the end iterators of both sequences.
- The four iterator version of the routine `mismatch` is part of the C++14 standard. When C++14 standard library implementations become available, the implementation from the standard library should be used.

Other Algorithms

clamp

The header file clamp.hpp contains two functions for "clamping" a value between a pair of boundary values.

clamp

The function `clamp (v, lo, hi)` returns:

- lo if v < lo
- hi if hi < v
- otherwise, v

Note: using `clamp` with floating point numbers may give unexpected results if one of the values is `NaN`.

There is also a version that allows the caller to specify a comparison predicate to use instead of `operator <`.

```
template<typename V>
V clamp ( V val, V lo, V hi );

template<typename V, typename Pred>
V clamp ( V val, V lo, V hi, Pred p );
```

The following code:

```
int foo = 23;
foo = clamp ( foo, 1, 10 );
```

will leave `foo` with a value of 10

Complexity: `clamp` will make either one or two calls to the comparison predicate before returning one of the three parameters.

clamp_range

There are also four range-based versions of clamp, that apply clamping to a series of values. You could write them yourself with std::transform and bind, like this: `std::transform ( first, last, out, bind ( clamp ( _1, lo, hi ) ))`, but they are provided here for your convenience.

```
template<typename InputIterator, typename OutputIterator>
OutputIterator clamp_range ( InputIterator first, InputIterator last, OutputIterator out,
    typename std::iterator_traits<InputIterator>::value_type lo,
    typename std::iterator_traits<InputIterator>::value_type hi );

template<typename Range, typename OutputIterator>
OutputIterator clamp_range ( const Range &r, OutputIterator out,
 typename std::iterator_traits<typename boost::range_iterator<const Range>::type>::value_type lo,

 typename std::iterator_traits<typename boost::range_iterator<const Range>::type>::value_type hi );

template<typename InputIterator, typename OutputIterator, typename Pred>
OutputIterator clamp_range ( InputIterator first, InputIterator last, OutputIterator out,
    typename std::iterator_traits<InputIterator>::value_type lo,
    typename std::iterator_traits<InputIterator>::value_type hi, Pred p );

template<typename Range, typename OutputIterator, typename Pred>
OutputIterator clamp_range ( const Range &r, OutputIterator out,
 typename std::iterator_traits<typename boost::range_iterator<const Range>::type>::value_type lo,

 typename std::iterator_traits<typename boost::range_iterator<const Range>::type>::value_type hi,

 Pred p );
```

gather

The header file 'boost/algorithm/gather.hpp' contains two variants of a single algorithm, `gather`.

`gather()` takes a collection of elements defined by a pair of iterators and moves the ones satisfying a predicate to them to a position (called the pivot) within the sequence. The algorithm is stable. The result is a pair of iterators that contains the items that satisfy the predicate.

Interface

The function `gather` returns a `std::pair` of iterators that denote the elements that satisfy the predicate.

There are two versions; one takes two iterators, and the other takes a range.

```
namespace boost { namespace algorithm {

template <typename BidirectionalIterator, typename Pred>
std::pair<BidirectionalIterator,BidirectionalIterator>
gather ( BidirectionalIterator first, BidirectionalIterator last, BidirectionalIterat↵
or pivot, Pred pred );

template <typename BidirectionalRange, typename Pred>
std::pair<typename boost::range_iterator<const BidirectionalRange>::type, type↵
name boost::range_iterator<const BidirectionalRange>::type>
gather ( const BidirectionalRange &range, typename boost::range_iterator<const Bidirection↵
alRange>::type pivot, Pred pred );

}}
```

Examples

Given an sequence containing:

```
0 1 2 3 4 5 6 7 8 9
```

a call to gather (arr, arr + 10, arr + 4, IsEven) will result in:

```
1 3 0 2 4 6 8 5 7 9
    |---|-----|
  first |  second
      pivot
```

where `first` and `second` are the fields of the pair that is returned by the call.

Iterator Requirements

`gather` work on bidirectional iterators or better. This requirement comes from the usage of `stable_partition`, which requires bidirectional iterators. Some standard libraries (libstdc++ and libc++, for example) have implementations of `stable_partition` that work with forward iterators. If that is the case, then `gather` will work with forward iterators as well.

Storage Requirements

`gather` uses `stable_partition`, which will attempt to allocate temporary memory, but will work in-situ if there is none available.

Complexity

If there is sufficient memory available, the run time is linear: `O(N)`

If there is not any memory available, then the run time is `O(N log N)`.

Exception Safety

Notes

hex

The header file `'boost/algorithm/hex.hpp'` contains three variants each of two algorithms, `hex` and `unhex`. They are inverse algorithms; that is, one undoes the effort of the other. `hex` takes a sequence of values, and turns them into hexadecimal characters. `unhex` takes a sequence of hexadecimal characters, and outputs a sequence of values.

`hex` and `unhex` come from MySQL, where they are used in database queries and stored procedures.

interface

The function `hex` takes a sequence of values and writes hexadecimal characters. There are three different interfaces, differing only in how the input sequence is specified.

The first one takes an iterator pair. The second one takes a pointer to the start of a zero-terminated sequence, such as a c string, and the third takes a range as defined by the Boost.Range library.

```
template <typename InputIterator, typename OutputIterator>
OutputIterator hex ( InputIterator first, InputIterator last, OutputIterator out );

template <typename T, typename OutputIterator>
OutputIterator hex ( const T *ptr, OutputIterator out );

template <typename Range, typename OutputIterator>
OutputIterator hex ( const Range &r, OutputIterator out );
```

`hex` writes only values in the range '0'..'9' and 'A'..'F', but is not limited to character output. The output iterator could refer to a wstring, or a vector of integers, or any other integral type.

The function `unhex` takes the output of `hex` and turns it back into a sequence of values.

The input parameters for the different variations of unhex are the same as hex.

```
template <typename InputIterator, typename OutputIterator>
OutputIterator unhex ( InputIterator first, InputIterator last, OutputIterator out );

template <typename T, typename OutputIterator>
OutputIterator unhex ( const T *ptr, OutputIterator out );

template <typename Range, typename OutputIterator>
OutputIterator unhex ( const Range &r, OutputIterator out );
```

Error Handling

The header 'hex.hpp' defines three exception classes:

```
struct hex_decode_error: virtual boost::exception, virtual std::exception {};
struct not_enough_input : public hex_decode_error;
struct non_hex_input : public hex_decode_error;
```

If the input to unhex does not contain an "even number" of hex digits, then an exception of type boost::algorithm::not_enough_input is thrown.

If the input to unhex contains any non-hexadecimal characters, then an exception of type boost::algorithm::non_hex_input is thrown.

If you want to catch all the decoding errors, you can catch exceptions of type boost::algorithm::hex_decode_error.

Examples

Assuming that out is an iterator that accepts char values, and wout accepts wchar_t values (and that sizeof (wchar_t) == 2)

```
hex ( "abcdef", out )  --> "616263646566"
hex ( "32", out )      --> "3332"
hex ( "abcdef", wout ) --> "006100620063006400650066"
hex ( "32", wout )     --> "00330032"

unhex ( "616263646566", out )  --> "abcdef"
unhex ( "3332", out )          --> "32"
unhex ( "616263646566", wout ) --> "\6162\6364\6566" ( i.e, a 3 character string )
unhex ( "3332", wout )         --> "\3233"    ( U+3332, SQUARE HUARADDO )

unhex ( "3", out )            --> Error - not enough input
unhex ( "32", wout )          --> Error - not enough input

unhex ( "ACEG", out )         --> Error - non-hex input
```

Iterator Requirements

hex and unhex work on all iterator types.

Complexity

All of the variants of hex and unhex run in *O(N)* (linear) time; that is, that is, they process each element in the input sequence once.

Exception Safety

All of the variants of hex and unhex take their parameters by value or const reference, and do not depend upon any global state. Therefore, all the routines in this file provide the strong exception guarantee. However, when working on input iterators, if an exception is thrown, the input iterators will not be reset to their original values (i.e, the characters read from the iterator cannot be un-read)

Notes

- hex and unhex both do nothing when passed empty ranges.

Reference

Header <boost/algorithm/clamp.hpp>

Clamp algorithm.

Marshall Clow
Suggested by olafvdspek in https://svn.boost.org/trac/boost/ticket/3215

```
namespace boost {
  namespace algorithm {
    template<typename T, typename Pred>
      T const & clamp(T const &,
                      typename boost::mpl::identity< T >::type const &,
                      typename boost::mpl::identity< T >::type const &, Pred);
    template<typename T>
      T const & clamp(const T &,
                      typename boost::mpl::identity< T >::type const &,
                      typename boost::mpl::identity< T >::type const &);
    template<typename InputIterator, typename OutputIterator>
      OutputIterator
      clamp_range(InputIterator, InputIterator, OutputIterator,
                  typename std::iterator_traits< InputIterator >::value_type,
                  typename std::iterator_traits< InputIterator >::value_type);
    template<typename Range, typename OutputIterator>
     boost::disable_if_c< boost::is_same< Range, OutputIterator >::value, OutputIterator >::type
      clamp_range(const Range &, OutputIterator,
                  typename std::iterator_traits< typename boost::range_iterat↵
or< const Range >::type >::value_type,
                  typename std::iterator_traits< typename boost::range_iterat↵
or< const Range >::type >::value_type);
    template<typename InputIterator, typename OutputIterator, typename Pred>
      OutputIterator
      clamp_range(InputIterator, InputIterator, OutputIterator,
                  typename std::iterator_traits< InputIterator >::value_type,
                  typename std::iterator_traits< InputIterator >::value_type,
                  Pred);
    template<typename Range, typename OutputIterator, typename Pred>
     boost::disable_if_c< boost::is_same< Range, OutputIterator >::value, OutputIterator >::type
      clamp_range(const Range &, OutputIterator,
                  typename std::iterator_traits< typename boost::range_iterat↵
or< const Range >::type >::value_type,
                  typename std::iterator_traits< typename boost::range_iterat↵
or< const Range >::type >::value_type,
                  Pred);
  }
}
```

Function template clamp

boost::algorithm::clamp

Synopsis

```
// In header: <boost/algorithm/clamp.hpp>


template<typename T, typename Pred>
  T const & clamp(T const & val,
                  typename boost::mpl::identity< T >::type const & lo,
                  typename boost::mpl::identity< T >::type const & hi,
                  Pred p);
```

Description

Parameters:	`hi`	The upper bound of the range to be clamped to
	`lo`	The lower bound of the range to be clamped to
	`p`	A predicate to use to compare the values. p (a, b) returns a boolean.
	`val`	The value to be clamped
Returns:	the value "val" brought into the range [lo, hi] using the comparison predicate p. If p (val, lo) return lo. If p (hi, val) return hi. Otherwise, return the original value.	

Function template clamp

boost::algorithm::clamp

Synopsis

```
// In header: <boost/algorithm/clamp.hpp>


template<typename T>
  T const & clamp(const T & val,
                  typename boost::mpl::identity< T >::type const & lo,
                  typename boost::mpl::identity< T >::type const & hi);
```

Description

Parameters:	`hi`	The upper bound of the range to be clamped to
	`lo`	The lower bound of the range to be clamped to
	`val`	The value to be clamped
Returns:	the value "val" brought into the range [lo, hi]. If the value is less than lo, return lo. If the value is greater than "hi", return hi. Otherwise, return the original value.	

Function template clamp_range

boost::algorithm::clamp_range

Synopsis

```
// In header: <boost/algorithm/clamp.hpp>


template<typename InputIterator, typename OutputIterator>
  OutputIterator
  clamp_range(InputIterator first, InputIterator last, OutputIterator out,
              typename std::iterator_traits< InputIterator >::value_type lo,
              typename std::iterator_traits< InputIterator >::value_type hi);
```

Description

Parameters:	`first`	The start of the range of values
	`hi`	The upper bound of the range to be clamped to
	`last`	One past the end of the range of input values
	`lo`	The lower bound of the range to be clamped to
	`out`	An output iterator to write the clamped values into
Returns:	clamp the sequence of values [first, last) into [lo, hi]	

Function template clamp_range

boost::algorithm::clamp_range

Synopsis

```
// In header: <boost/algorithm/clamp.hpp>


template<typename Range, typename OutputIterator>
  boost::disable_if_c< boost::is_same< Range, OutputIterator >::value, OutputIterator >::type
  clamp_range(const Range & r, OutputIterator out,
              typename std::iterator_traits< typename boost::range_iterat↵
or< const Range >::type >::value_type lo,
              typename std::iterator_traits< typename boost::range_iterat↵
or< const Range >::type >::value_type hi);
```

Description

Parameters:	`hi`	The upper bound of the range to be clamped to
	`lo`	The lower bound of the range to be clamped to
	`out`	An output iterator to write the clamped values into
	`r`	The range of values to be clamped
Returns:	clamp the sequence of values [first, last) into [lo, hi]	

Function template clamp_range

boost::algorithm::clamp_range

Synopsis

```
// In header: <boost/algorithm/clamp.hpp>


template<typename InputIterator, typename OutputIterator, typename Pred>
  OutputIterator
  clamp_range(InputIterator first, InputIterator last, OutputIterator out,
              typename std::iterator_traits< InputIterator >::value_type lo,
              typename std::iterator_traits< InputIterator >::value_type hi,
              Pred p);
```

Description

Parameters:	`first`	The start of the range of values
	`hi`	The upper bound of the range to be clamped to
	`last`	One past the end of the range of input values
	`lo`	The lower bound of the range to be clamped to
	`out`	An output iterator to write the clamped values into
	`p`	A predicate to use to compare the values. p (a, b) returns a boolean.
Returns:	clamp the sequence of values [first, last) into [lo, hi] using the comparison predicate p.	

Function template clamp_range

boost::algorithm::clamp_range

Synopsis

```
// In header: <boost/algorithm/clamp.hpp>


template<typename Range, typename OutputIterator, typename Pred>
  boost::disable_if_c< boost::is_same< Range, OutputIterator >::value, OutputIterator >::type
  clamp_range(const Range & r, OutputIterator out,
              typename std::iterator_traits< typename boost::range_iterat↵
or< const Range >::type >::value_type lo,
              typename std::iterator_traits< typename boost::range_iterat↵
or< const Range >::type >::value_type hi,
              Pred p);
```

Description

Parameters:	`hi`	The upper bound of the range to be clamped to
	`lo`	The lower bound of the range to be clamped to
	`out`	An output iterator to write the clamped values into
	`p`	A predicate to use to compare the values. p (a, b) returns a boolean.
	`r`	The range of values to be clamped
Returns:	clamp the sequence of values [first, last) into [lo, hi] using the comparison predicate p.	

Header <boost/algorithm/cxx11/all_of.hpp>

Test ranges to see if all elements match a value or predicate.

Marshall Clow

```
namespace boost {
  namespace algorithm {
    template<typename InputIterator, typename Predicate>
      bool all_of(InputIterator, InputIterator, Predicate);
    template<typename Range, typename Predicate>
      bool all_of(const Range &, Predicate);
    template<typename InputIterator, typename T>
      bool all_of_equal(InputIterator, InputIterator, const T &);
    template<typename Range, typename T>
      bool all_of_equal(const Range &, const T &);
  }
}
```

Function template all_of

boost::algorithm::all_of

Synopsis

```
// In header: <boost/algorithm/cxx11/all_of.hpp>


template<typename InputIterator, typename Predicate>
  bool all_of(InputIterator first, InputIterator last, Predicate p);
```

Description

Note

returns true on an empty range

Note

This function is part of the C++2011 standard library. We will use the standard one if it is available, otherwise we have our own implementation.

Parameters:	`first`	The start of the input sequence
	`last`	One past the end of the input sequence
	`p`	A predicate for testing the elements of the sequence
Returns:	true if all elements in [first, last) satisfy the predicate 'p'	

Function template all_of

boost::algorithm::all_of

Synopsis

```
// In header: <boost/algorithm/cxx11/all_of.hpp>


template<typename Range, typename Predicate>
  bool all_of(const Range & r, Predicate p);
```

Description

Note

returns true on an empty range

Parameters: `p` A predicate for testing the elements of the range
`r` The input range
Returns: true if all elements in the range satisfy the predicate 'p'

Function template all_of_equal

boost::algorithm::all_of_equal

Synopsis

```
// In header: <boost/algorithm/cxx11/all_of.hpp>


template<typename InputIterator, typename T>
  bool all_of_equal(InputIterator first, InputIterator last, const T & val);
```

Description

Note

returns true on an empty range

Parameters: `first` The start of the input sequence
`last` One past the end of the input sequence
`val` A value to compare against
Returns: true if all elements in [first, last) are equal to 'val'

Function template all_of_equal

boost::algorithm::all_of_equal

Synopsis

```
// In header: <boost/algorithm/cxx11/all_of.hpp>


template<typename Range, typename T>
  bool all_of_equal(const Range & r, const T & val);
```

Description

Note

returns true on an empty range

Parameters: `r` The input range
`val` A value to compare against
Returns: true if all elements in the range are equal to 'val'

Header <boost/algorithm/cxx11/any_of.hpp>

Test ranges to see if any elements match a value or predicate.

Marshall Clow

```
namespace boost {
  namespace algorithm {
    template<typename InputIterator, typename Predicate>
      bool any_of(InputIterator, InputIterator, Predicate);
    template<typename Range, typename Predicate>
      bool any_of(const Range &, Predicate);
    template<typename InputIterator, typename V>
      bool any_of_equal(InputIterator, InputIterator, const V &);
    template<typename Range, typename V>
      bool any_of_equal(const Range &, const V &);
  }
}
```

Function template any_of

boost::algorithm::any_of

Synopsis

```
// In header: <boost/algorithm/cxx11/any_of.hpp>


template<typename InputIterator, typename Predicate>
  bool any_of(InputIterator first, InputIterator last, Predicate p);
```

Description

Note

returns false on an empty range

Parameters: `first` The start of the input sequence
`last` One past the end of the input sequence
`p` A predicate for testing the elements of the sequence
Returns: true if any of the elements in [first, last) satisfy the predicate

Function template any_of

boost::algorithm::any_of

Synopsis

```
// In header: <boost/algorithm/cxx11/any_of.hpp>


template<typename Range, typename Predicate>
  bool any_of(const Range & r, Predicate p);
```

Description

Note

returns false on an empty range

Parameters: `p` A predicate for testing the elements of the range
`r` The input range
Returns: true if any elements in the range satisfy the predicate 'p'

Function template any_of_equal

boost::algorithm::any_of_equal

Synopsis

```
// In header: <boost/algorithm/cxx11/any_of.hpp>


template<typename InputIterator, typename V>
  bool any_of_equal(InputIterator first, InputIterator last, const V & val);
```

Description

Note

returns false on an empty range

Parameters: `first` The start of the input sequence
`last` One past the end of the input sequence
`val` A value to compare against
Returns: true if any of the elements in [first, last) are equal to 'val'

Function template any_of_equal

boost::algorithm::any_of_equal

Synopsis

```
// In header: <boost/algorithm/cxx11/any_of.hpp>


template<typename Range, typename V>
  bool any_of_equal(const Range & r, const V & val);
```

Description

Note

returns false on an empty range

Parameters: `r` The input range
`val` A value to compare against
Returns: true if any of the elements in the range are equal to 'val'

Header <boost/algorithm/cxx11/copy_if.hpp>

Copy a subset of a sequence to a new sequence.

Marshall Clow

```
namespace boost {
  namespace algorithm {
    template<typename InputIterator, typename OutputIterator,
             typename Predicate>
      OutputIterator
      copy_if(InputIterator, InputIterator, OutputIterator, Predicate);
    template<typename Range, typename OutputIterator, typename Predicate>
      OutputIterator copy_if(const Range &, OutputIterator, Predicate);
    template<typename InputIterator, typename OutputIterator,
             typename Predicate>
      std::pair< InputIterator, OutputIterator >
      copy_while(InputIterator, InputIterator, OutputIterator, Predicate);
    template<typename Range, typename OutputIterator, typename Predicate>
      std::pair< typename boost::range_iterator< const Range >::type, OutputIterator >
      copy_while(const Range &, OutputIterator, Predicate);
    template<typename InputIterator, typename OutputIterator,
             typename Predicate>
      std::pair< InputIterator, OutputIterator >
      copy_until(InputIterator, InputIterator, OutputIterator, Predicate);
    template<typename Range, typename OutputIterator, typename Predicate>
      std::pair< typename boost::range_iterator< const Range >::type, OutputIterator >
      copy_until(const Range &, OutputIterator, Predicate);
  }
}
```

Function template copy_if

boost::algorithm::copy_if — Copies all the elements from the input range that satisfy the predicate to the output range.

Synopsis

```
// In header: <boost/algorithm/cxx11/copy_if.hpp>


template<typename InputIterator, typename OutputIterator, typename Predicate>
  OutputIterator
  copy_if(InputIterator first, InputIterator last, OutputIterator result,
          Predicate p);
```

Description

Note

This function is part of the C++2011 standard library. We will use the standard one if it is available, otherwise we have our own implementation.

Parameters:	`first`	The start of the input sequence
	`last`	One past the end of the input sequence
	`p`	A predicate for testing the elements of the range
	`result`	An output iterator to write the results into
Returns:	The updated output iterator	

Function template copy_if

boost::algorithm::copy_if — Copies all the elements from the input range that satisfy the predicate to the output range.

Synopsis

```
// In header: <boost/algorithm/cxx11/copy_if.hpp>


template<typename Range, typename OutputIterator, typename Predicate>
  OutputIterator copy_if(const Range & r, OutputIterator result, Predicate p);
```

Description

Parameters:	`p`	A predicate for testing the elements of the range
	`r`	The input range
	`result`	An output iterator to write the results into
Returns:	The updated output iterator	

Function template copy_while

boost::algorithm::copy_while — Copies all the elements at the start of the input range that satisfy the predicate to the output range.

Synopsis

```
// In header: <boost/algorithm/cxx11/copy_if.hpp>


template<typename InputIterator, typename OutputIterator, typename Predicate>
  std::pair< InputIterator, OutputIterator >
  copy_while(InputIterator first, InputIterator last, OutputIterator result,
             Predicate p);
```

Description

Parameters:	`first`	The start of the input sequence
	`last`	One past the end of the input sequence
	`p`	A predicate for testing the elements of the range
	`result`	An output iterator to write the results into
Returns:	The updated input and output iterators	

Function template copy_while

boost::algorithm::copy_while — Copies all the elements at the start of the input range that satisfy the predicate to the output range.

Synopsis

```
// In header: <boost/algorithm/cxx11/copy_if.hpp>


template<typename Range, typename OutputIterator, typename Predicate>
  std::pair< typename boost::range_iterator< const Range >::type, OutputIterator >
  copy_while(const Range & r, OutputIterator result, Predicate p);
```

Description

Parameters:	`p`	A predicate for testing the elements of the range
	`r`	The input range
	`result`	An output iterator to write the results into
Returns:	The updated input and output iterators	

Function template copy_until

boost::algorithm::copy_until — Copies all the elements at the start of the input range that do not satisfy the predicate to the output range.

Synopsis

```
// In header: <boost/algorithm/cxx11/copy_if.hpp>


template<typename InputIterator, typename OutputIterator, typename Predicate>
  std::pair< InputIterator, OutputIterator >
  copy_until(InputIterator first, InputIterator last, OutputIterator result,
             Predicate p);
```

Description

Parameters:	`first`	The start of the input sequence
	`last`	One past the end of the input sequence
	`p`	A predicate for testing the elements of the range
	`result`	An output iterator to write the results into
Returns:	The updated output iterator	

Function template copy_until

boost::algorithm::copy_until — Copies all the elements at the start of the input range that do not satisfy the predicate to the output range.

Synopsis

```
// In header: <boost/algorithm/cxx11/copy_if.hpp>


template<typename Range, typename OutputIterator, typename Predicate>
  std::pair< typename boost::range_iterator< const Range >::type, OutputIterator >
  copy_until(const Range & r, OutputIterator result, Predicate p);
```

Description

Parameters:	`p`	A predicate for testing the elements of the range
	`r`	The input range
	`result`	An output iterator to write the results into
Returns:	The updated output iterator	

Header <boost/algorithm/cxx11/copy_n.hpp>

Copy n items from one sequence to another.

Marshall Clow

```
namespace boost {
  namespace algorithm {
    template<typename InputIterator, typename Size, typename OutputIterator>
      OutputIterator copy_n(InputIterator, Size, OutputIterator);
  }
}
```

Function template copy_n

boost::algorithm::copy_n — Copies exactly n (n > 0) elements from the range starting at first to the range starting at result.

Synopsis

```
// In header: <boost/algorithm/cxx11/copy_n.hpp>


template<typename InputIterator, typename Size, typename OutputIterator>
  OutputIterator copy_n(InputIterator first, Size n, OutputIterator result);
```

Description

Note

This function is part of the C++2011 standard library. We will use the standard one if it is available, otherwise we have our own implementation.

Parameters: `first` The start of the input sequence
`n` The number of elements to copy
`result` An output iterator to write the results into
Returns: The updated output iterator

Header <boost/algorithm/cxx11/find_if_not.hpp>

Find the first element in a sequence that does not satisfy a predicate.

Marshall Clow

```
namespace boost {
  namespace algorithm {
    template<typename InputIterator, typename Predicate>
      InputIterator find_if_not(InputIterator, InputIterator, Predicate);
    template<typename Range, typename Predicate>
      boost::range_iterator< const Range >::type
      find_if_not(const Range &, Predicate);
  }
}
```

Function template find_if_not

boost::algorithm::find_if_not — Finds the first element in the sequence that does not satisfy the predicate.

Synopsis

```
// In header: <boost/algorithm/cxx11/find_if_not.hpp>


template<typename InputIterator, typename Predicate>
  InputIterator
  find_if_not(InputIterator first, InputIterator last, Predicate p);
```

Description

Note

This function is part of the C++2011 standard library. We will use the standard one if it is available, otherwise we have our own implementation.

Parameters: `first` The start of the input sequence
`last` One past the end of the input sequence
`p` A predicate for testing the elements of the range
Returns: The iterator pointing to the desired element.

Function template find_if_not

boost::algorithm::find_if_not — Finds the first element in the sequence that does not satisfy the predicate.

Synopsis

```
// In header: <boost/algorithm/cxx11/find_if_not.hpp>


template<typename Range, typename Predicate>
  boost::range_iterator< const Range >::type
  find_if_not(const Range & r, Predicate p);
```

Description

Parameters: `p` A predicate for testing the elements of the range
`r` The input range
Returns: The iterator pointing to the desired element.

Header <boost/algorithm/cxx11/iota.hpp>

Generate an increasing series.

Marshall Clow

```
namespace boost {
  namespace algorithm {
    template<typename ForwardIterator, typename T>
      void iota(ForwardIterator, ForwardIterator, T);
    template<typename Range, typename T> void iota(Range &, T);
    template<typename OutputIterator, typename T>
      OutputIterator iota_n(OutputIterator, T, std::size_t);
  }
}
```

Function template iota

boost::algorithm::iota — Generates an increasing sequence of values, and stores them in [first, last)

Synopsis

```
// In header: <boost/algorithm/cxx11/iota.hpp>


template<typename ForwardIterator, typename T>
  void iota(ForwardIterator first, ForwardIterator last, T value);
```

Description

Note

This function is part of the C++2011 standard library. We will use the standard one if it is available, otherwise we have our own implementation.

Parameters:		
	`first`	The start of the input sequence
	`last`	One past the end of the input sequence
	`value`	The initial value of the sequence to be generated

Function template iota

boost::algorithm::iota — Generates an increasing sequence of values, and stores them in the input Range.

Synopsis

```
// In header: <boost/algorithm/cxx11/iota.hpp>


template<typename Range, typename T> void iota(Range & r, T value);
```

Description

Parameters:		
	`r`	The input range
	`value`	The initial value of the sequence to be generated

Function template iota_n

boost::algorithm::iota_n — Generates an increasing sequence of values, and stores them in the input Range.

Synopsis

```
// In header: <boost/algorithm/cxx11/iota.hpp>


template<typename OutputIterator, typename T>
  OutputIterator iota_n(OutputIterator out, T value, std::size_t n);
```

Description

Parameters:		
	`n`	The number of items to write
	`out`	An output iterator to write the results into
	`value`	The initial value of the sequence to be generated

Header <boost/algorithm/cxx11/is_partitioned.hpp>

Tell if a sequence is partitioned.

Marshall Clow

```
namespace boost {
  namespace algorithm {
    template<typename InputIterator, typename UnaryPredicate>
      bool is_partitioned(InputIterator, InputIterator, UnaryPredicate);
    template<typename Range, typename UnaryPredicate>
      bool is_partitioned(const Range &, UnaryPredicate);
  }
}
```

Function template is_partitioned

boost::algorithm::is_partitioned — Tests to see if a sequence is partitioned according to a predicate.

Synopsis

```
// In header: <boost/algorithm/cxx11/is_partitioned.hpp>


template<typename InputIterator, typename UnaryPredicate>
  bool is_partitioned(InputIterator first, InputIterator last,
                      UnaryPredicate p);
```

Description

Note

This function is part of the C++2011 standard library. We will use the standard one if it is available, otherwise we have our own implementation.

Parameters:

`first`	The start of the input sequence
`last`	One past the end of the input sequence
`p`	The predicate to test the values with

Function template is_partitioned

boost::algorithm::is_partitioned — Generates an increasing sequence of values, and stores them in the input Range.

Synopsis

```
// In header: <boost/algorithm/cxx11/is_partitioned.hpp>


template<typename Range, typename UnaryPredicate>
  bool is_partitioned(const Range & r, UnaryPredicate p);
```

Description

Parameters:

`p`	The predicate to test the values with
`r`	The input range

Header <boost/algorithm/cxx11/is_permutation.hpp>

```
namespace boost {
  namespace algorithm {
    template<typename ForwardIterator1, typename ForwardIterator2,
             typename BinaryPredicate>
      bool is_permutation(ForwardIterator1, ForwardIterator1,
                          ForwardIterator2, BinaryPredicate);
    template<typename ForwardIterator1, typename ForwardIterator2>
      bool is_permutation(ForwardIterator1, ForwardIterator1,
                          ForwardIterator2);
    template<typename ForwardIterator1, typename ForwardIterator2>
      bool is_permutation(ForwardIterator1, ForwardIterator1,
                          ForwardIterator2, ForwardIterator2);
    template<typename ForwardIterator1, typename ForwardIterator2,
             typename BinaryPredicate>
      bool is_permutation(ForwardIterator1, ForwardIterator1,
                          ForwardIterator2, ForwardIterator2,
                          BinaryPredicate);
    template<typename Range, typename ForwardIterator>
      bool is_permutation(const Range &, ForwardIterator);
    template<typename Range, typename ForwardIterator,
             typename BinaryPredicate>
      boost::disable_if_c< boost::is_same< Range, ForwardIterator >::value, bool >::type
      is_permutation(const Range &, ForwardIterator, BinaryPredicate);
  }
}
```

Function template is_permutation

boost::algorithm::is_permutation — Tests to see if the sequence [first,last) is a permutation of the sequence starting at first2.

Synopsis

```
// In header: <boost/algorithm/cxx11/is_permutation.hpp>


template<typename ForwardIterator1, typename ForwardIterator2,
         typename BinaryPredicate>
  bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                      ForwardIterator2 first2, BinaryPredicate p);
```

Description

Note

This function is part of the C++2011 standard library. We will use the standard one if it is available, otherwise we have our own implementation.

Parameters:

`first1`	The start of the input sequence
`first2`	The start of the second sequence
`last1`	One past the end of the input sequence
`p`	The predicate to compare elements with

Function template is_permutation

boost::algorithm::is_permutation — Tests to see if the sequence [first,last) is a permutation of the sequence starting at first2.

Synopsis

```
// In header: <boost/algorithm/cxx11/is_permutation.hpp>


template<typename ForwardIterator1, typename ForwardIterator2>
  bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                      ForwardIterator2 first2);
```

Description

Note

This function is part of the C++2011 standard library. We will use the standard one if it is available, otherwise we have our own implementation.

Parameters:

`first1`	The start of the input sequence
`first2`	The start of the second sequence

Function template is_permutation

boost::algorithm::is_permutation — Tests to see if the sequence [first,last) is a permutation of the sequence starting at first2.

Synopsis

```
// In header: <boost/algorithm/cxx11/is_permutation.hpp>


template<typename ForwardIterator1, typename ForwardIterator2>
  bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                      ForwardIterator2 first2, ForwardIterator2 last2);
```

Description

Note

This function is part of the C++2011 standard library. We will use the standard one if it is available, otherwise we have our own implementation.

Parameters:

`first1`	The start of the input sequence
`first2`	The start of the second sequence
`last1`	One past the end of the second sequence
`last2`	One past the end of the input sequence

Function template is_permutation

boost::algorithm::is_permutation — Tests to see if the sequence [first,last) is a permutation of the sequence starting at first2.

Synopsis

```
// In header: <boost/algorithm/cxx11/is_permutation.hpp>


template<typename ForwardIterator1, typename ForwardIterator2,
         typename BinaryPredicate>
  bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                      ForwardIterator2 first2, ForwardIterator2 last2,
                      BinaryPredicate pred);
```

Description

Note

This function is part of the C++2011 standard library. We will use the standard one if it is available, otherwise we have our own implementation.

Parameters:		
	`first1`	The start of the input sequence
	`first2`	The start of the second sequence
	`last1`	One past the end of the input sequence
	`last2`	One past the end of the second sequence
	`pred`	The predicate to compare elements with

Function template is_permutation

boost::algorithm::is_permutation — Tests to see if the sequence [first,last) is a permutation of the sequence starting at first2.

Synopsis

```
// In header: <boost/algorithm/cxx11/is_permutation.hpp>


template<typename Range, typename ForwardIterator>
  bool is_permutation(const Range & r, ForwardIterator first2);
```

Description

Parameters:		
	`first2`	The start of the second sequence
	`r`	The input range

Function template is_permutation

boost::algorithm::is_permutation — Tests to see if the sequence [first,last) is a permutation of the sequence starting at first2.

Synopsis

```
// In header: <boost/algorithm/cxx11/is_permutation.hpp>


template<typename Range, typename ForwardIterator, typename BinaryPredicate>
  boost::disable_if_c< boost::is_same< Range, ForwardIterator >::value, bool >::type
  is_permutation(const Range & r, ForwardIterator first2,
                 BinaryPredicate pred);
```

Description

Parameters:	`first2`	The start of the second sequence
	`pred`	The predicate to compare elements with
	`r`	The input range

Header <boost/algorithm/cxx14/is_permutation.hpp>

Header <boost/algorithm/cxx11/is_sorted.hpp>

```
namespace boost {
  namespace algorithm {
    template<typename ForwardIterator, typename Pred>
      ForwardIterator is_sorted_until(ForwardIterator, ForwardIterator, Pred);
    template<typename ForwardIterator>
      ForwardIterator is_sorted_until(ForwardIterator, ForwardIterator);
    template<typename ForwardIterator, typename Pred>
      bool is_sorted(ForwardIterator, ForwardIterator, Pred);
    template<typename ForwardIterator>
      bool is_sorted(ForwardIterator, ForwardIterator);
    template<typename R, typename Pred>
      boost::lazy_disable_if_c< boost::is_same< R, Pred >::value, typename boost::range_iterat↵
or< const R > >::type
      is_sorted_until(const R &, Pred);
    template<typename R>
      boost::range_iterator< const R >::type is_sorted_until(const R &);
    template<typename R, typename Pred>
      boost::lazy_disable_if_c< boost::is_same< R, Pred >::value, boost::mpl::iden↵
tity< bool > >::type
      is_sorted(const R &, Pred);
    template<typename R> bool is_sorted(const R &);
    template<typename ForwardIterator>
      bool is_increasing(ForwardIterator, ForwardIterator);
    template<typename R> bool is_increasing(const R &);
    template<typename ForwardIterator>
      bool is_decreasing(ForwardIterator, ForwardIterator);
    template<typename R> bool is_decreasing(const R &);
    template<typename ForwardIterator>
      bool is_strictly_increasing(ForwardIterator, ForwardIterator);
    template<typename R> bool is_strictly_increasing(const R &);
    template<typename ForwardIterator>
      bool is_strictly_decreasing(ForwardIterator, ForwardIterator);
    template<typename R> bool is_strictly_decreasing(const R &);
  }
}
```

Function template is_sorted_until

boost::algorithm::is_sorted_until

Synopsis

```
// In header: <boost/algorithm/cxx11/is_sorted.hpp>


template<typename ForwardIterator, typename Pred>
  ForwardIterator
  is_sorted_until(ForwardIterator first, ForwardIterator last, Pred p);
```

Description

Parameters:	`first`	The start of the sequence to be tested.
	`last`	One past the end of the sequence
	`p`	A binary predicate that returns true if two elements are ordered.
Returns:	the point in the sequence [first, last) where the elements are unordered (according to the comparison predicate 'p').	

Function template is_sorted_until

boost::algorithm::is_sorted_until

Synopsis

```
// In header: <boost/algorithm/cxx11/is_sorted.hpp>


template<typename ForwardIterator>
  ForwardIterator is_sorted_until(ForwardIterator first, ForwardIterator last);
```

Description

Parameters:	`first`	The start of the sequence to be tested.
	`last`	One past the end of the sequence
Returns:	the point in the sequence [first, last) where the elements are unordered	

Function template is_sorted

boost::algorithm::is_sorted

Synopsis

```
// In header: <boost/algorithm/cxx11/is_sorted.hpp>


template<typename ForwardIterator, typename Pred>
  bool is_sorted(ForwardIterator first, ForwardIterator last, Pred p);
```

Description

Parameters:	`first`	The start of the sequence to be tested.

	`last`	One past the end of the sequence
	`p`	A binary predicate that returns true if two elements are ordered.
Returns:	whether or not the entire sequence is sorted	

Function template is_sorted

boost::algorithm::is_sorted

Synopsis

```
// In header: <boost/algorithm/cxx11/is_sorted.hpp>


template<typename ForwardIterator>
  bool is_sorted(ForwardIterator first, ForwardIterator last);
```

Description

Parameters:	`first`	The start of the sequence to be tested.
	`last`	One past the end of the sequence
Returns:	whether or not the entire sequence is sorted	

Function template is_sorted_until

boost::algorithm::is_sorted_until

Synopsis

```
// In header: <boost/algorithm/cxx11/is_sorted.hpp>


template<typename R, typename Pred>
  boost::lazy_disable_if_c< boost::is_same< R, Pred >::value, typename boost::range_iterat↵
or< const R > >::type
  is_sorted_until(const R & range, Pred p);
```

Description

<ndash></ndash>
Range based versions of the C++11 functions

Parameters:	`p`	A binary predicate that returns true if two elements are ordered.
	`range`	The range to be tested.
Returns:	the point in the range R where the elements are unordered (according to the comparison predicate 'p').	

Function template is_sorted_until

boost::algorithm::is_sorted_until

Synopsis

```
// In header: <boost/algorithm/cxx11/is_sorted.hpp>


template<typename R>
  boost::range_iterator< const R >::type is_sorted_until(const R & range);
```

Description

Parameters: `range` The range to be tested.
Returns: the point in the range R where the elements are unordered

Function template is_sorted

boost::algorithm::is_sorted

Synopsis

```
// In header: <boost/algorithm/cxx11/is_sorted.hpp>


template<typename R, typename Pred>
  boost::lazy_disable_if_c< boost::is_same< R, Pred >::value, boost::mpl::identity< bool > >::type
   is_sorted(const R & range, Pred p);
```

Description

Parameters: `p` A binary predicate that returns true if two elements are ordered.
`range` The range to be tested.
Returns: whether or not the entire range R is sorted (according to the comparison predicate 'p').

Function template is_sorted

boost::algorithm::is_sorted

Synopsis

```
// In header: <boost/algorithm/cxx11/is_sorted.hpp>


template<typename R> bool is_sorted(const R & range);
```

Description

Parameters: `range` The range to be tested.
Returns: whether or not the entire range R is sorted

Function template is_increasing

boost::algorithm::is_increasing

Synopsis

```
// In header: <boost/algorithm/cxx11/is_sorted.hpp>


template<typename ForwardIterator>
  bool is_increasing(ForwardIterator first, ForwardIterator last);
```

Description

<ndash></ndash>
Range based versions of the C++11 functions

Note

This function will return true for sequences that contain items that compare equal. If that is not what you intended, you should use is_strictly_increasing instead.

Parameters: `first` The start of the sequence to be tested.
`last` One past the end of the sequence
Returns: true if the entire sequence is increasing; i.e, each item is greater than or equal to the previous one.

Function template is_increasing

boost::algorithm::is_increasing

Synopsis

```
// In header: <boost/algorithm/cxx11/is_sorted.hpp>


template<typename R> bool is_increasing(const R & range);
```

Description

Note

This function will return true for sequences that contain items that compare equal. If that is not what you intended, you should use is_strictly_increasing instead.

Parameters: `range` The range to be tested.
Returns: true if the entire sequence is increasing; i.e, each item is greater than or equal to the previous one.

Function template is_decreasing

boost::algorithm::is_decreasing

Synopsis

```
// In header: <boost/algorithm/cxx11/is_sorted.hpp>


template<typename ForwardIterator>
  bool is_decreasing(ForwardIterator first, ForwardIterator last);
```

Description

Note

This function will return true for sequences that contain items that compare equal. If that is not what you intended, you should use is_strictly_decreasing instead.

Parameters: `first` The start of the sequence to be tested.
`last` One past the end of the sequence
Returns: true if the entire sequence is decreasing; i.e, each item is less than or equal to the previous one.

Function template is_decreasing

boost::algorithm::is_decreasing

Synopsis

```
// In header: <boost/algorithm/cxx11/is_sorted.hpp>


template<typename R> bool is_decreasing(const R & range);
```

Description

Note

This function will return true for sequences that contain items that compare equal. If that is not what you intended, you should use is_strictly_decreasing instead.

Parameters: `range` The range to be tested.
Returns: true if the entire sequence is decreasing; i.e, each item is less than or equal to the previous one.

Function template is_strictly_increasing

boost::algorithm::is_strictly_increasing

Synopsis

```
// In header: <boost/algorithm/cxx11/is_sorted.hpp>


template<typename ForwardIterator>
  bool is_strictly_increasing(ForwardIterator first, ForwardIterator last);
```

Description

Note

This function will return false for sequences that contain items that compare equal. If that is not what you intended, you should use is_increasing instead.

Parameters: `first` The start of the sequence to be tested.
`last` One past the end of the sequence
Returns: true if the entire sequence is strictly increasing; i.e, each item is greater than the previous one

Function template is_strictly_increasing

boost::algorithm::is_strictly_increasing

Synopsis

```
// In header: <boost/algorithm/cxx11/is_sorted.hpp>


template<typename R> bool is_strictly_increasing(const R & range);
```

Description

Note

This function will return false for sequences that contain items that compare equal. If that is not what you intended, you should use is_increasing instead.

Parameters: `range` The range to be tested.
Returns: true if the entire sequence is strictly increasing; i.e, each item is greater than the previous one

Function template is_strictly_decreasing

boost::algorithm::is_strictly_decreasing

Synopsis

```
// In header: <boost/algorithm/cxx11/is_sorted.hpp>


template<typename ForwardIterator>
  bool is_strictly_decreasing(ForwardIterator first, ForwardIterator last);
```

Description

Note

This function will return false for sequences that contain items that compare equal. If that is not what you intended, you should use is_decreasing instead.

Parameters: `first` The start of the sequence to be tested.
`last` One past the end of the sequence
Returns: true if the entire sequence is strictly decreasing; i.e, each item is less than the previous one

Function template is_strictly_decreasing

boost::algorithm::is_strictly_decreasing

Synopsis

```
// In header: <boost/algorithm/cxx11/is_sorted.hpp>


template<typename R> bool is_strictly_decreasing(const R & range);
```

Description

Note

This function will return false for sequences that contain items that compare equal. If that is not what you intended, you should use is_decreasing instead.

Parameters: `range` The range to be tested.
Returns: true if the entire sequence is strictly decreasing; i.e, each item is less than the previous one

Header <boost/algorithm/cxx11/none_of.hpp>

Test ranges to see if no elements match a value or predicate.

Marshall Clow

```
namespace boost {
  namespace algorithm {
    template<typename InputIterator, typename Predicate>
      bool none_of(InputIterator, InputIterator, Predicate);
    template<typename Range, typename Predicate>
      bool none_of(const Range &, Predicate);
    template<typename InputIterator, typename V>
      bool none_of_equal(InputIterator, InputIterator, const V &);
    template<typename Range, typename V>
      bool none_of_equal(const Range &, const V &);
  }
}
```

Function template none_of

boost::algorithm::none_of

Synopsis

```
// In header: <boost/algorithm/cxx11/none_of.hpp>


template<typename InputIterator, typename Predicate>
  bool none_of(InputIterator first, InputIterator last, Predicate p);
```

Description

Note

returns true on an empty range

Parameters:
- `first` The start of the input sequence
- `last` One past the end of the input sequence
- `p` A predicate for testing the elements of the sequence

Returns: true if none of the elements in [first, last) satisfy the predicate 'p'

Function template none_of

boost::algorithm::none_of

Synopsis

```
// In header: <boost/algorithm/cxx11/none_of.hpp>


template<typename Range, typename Predicate>
  bool none_of(const Range & r, Predicate p);
```

Description

Note

returns true on an empty range

Parameters:
- `p` A predicate for testing the elements of the range
- `r` The input range

Returns: true if none of the elements in the range satisfy the predicate 'p'

Function template none_of_equal

boost::algorithm::none_of_equal

Synopsis

```
// In header: <boost/algorithm/cxx11/none_of.hpp>


template<typename InputIterator, typename V>
  bool none_of_equal(InputIterator first, InputIterator last, const V & val);
```

Description

Note

returns true on an empty range

Parameters:
- `first` The start of the input sequence
- `last` One past the end of the input sequence
- `val` A value to compare against

Returns: true if none of the elements in [first, last) are equal to 'val'

Function template none_of_equal

boost::algorithm::none_of_equal

Synopsis

```
// In header: <boost/algorithm/cxx11/none_of.hpp>


template<typename Range, typename V>
  bool none_of_equal(const Range & r, const V & val);
```

Description

Note

returns true on an empty range

Parameters:
- `r` The input range
- `val` A value to compare against

Returns: true if none of the elements in the range are equal to 'val'

Header <boost/algorithm/cxx11/one_of.hpp>

Test ranges to see if only one element matches a value or predicate.

Marshall Clow

```
namespace boost {
  namespace algorithm {
    template<typename InputIterator, typename Predicate>
      bool one_of(InputIterator, InputIterator, Predicate);
    template<typename Range, typename Predicate>
      bool one_of(const Range &, Predicate);
    template<typename InputIterator, typename V>
      bool one_of_equal(InputIterator, InputIterator, const V &);
    template<typename Range, typename V>
      bool one_of_equal(const Range &, const V &);
  }
}
```

Function template one_of

boost::algorithm::one_of

Synopsis

```
// In header: <boost/algorithm/cxx11/one_of.hpp>


template<typename InputIterator, typename Predicate>
  bool one_of(InputIterator first, InputIterator last, Predicate p);
```

Description

Parameters:	`first`	The start of the input sequence
	`last`	One past the end of the input sequence
	`p`	A predicate for testing the elements of the sequence
Returns:	true if the predicate 'p' is true for exactly one item in [first, last).	

Function template one_of

boost::algorithm::one_of

Synopsis

```
// In header: <boost/algorithm/cxx11/one_of.hpp>


template<typename Range, typename Predicate>
  bool one_of(const Range & r, Predicate p);
```

Description

Parameters:	`p`	A predicate for testing the elements of the range
	`r`	The input range
Returns:	true if the predicate 'p' is true for exactly one item in the range.	

Function template one_of_equal

boost::algorithm::one_of_equal

Synopsis

```
// In header: <boost/algorithm/cxx11/one_of.hpp>


template<typename InputIterator, typename V>
  bool one_of_equal(InputIterator first, InputIterator last, const V & val);
```

Description

Parameters:	`first`	The start of the input sequence
	`last`	One past the end of the input sequence
	`val`	A value to compare against
Returns:	true if the value 'val' exists only once in [first, last).	

Function template one_of_equal

boost::algorithm::one_of_equal

Synopsis

```
// In header: <boost/algorithm/cxx11/one_of.hpp>


template<typename Range, typename V>
  bool one_of_equal(const Range & r, const V & val);
```

Description

Parameters:	`r`	The input range
	`val`	A value to compare against
Returns:	true if the value 'val' exists only once in the range.	

Header <boost/algorithm/cxx11/partition_copy.hpp>

Copy a subset of a sequence to a new sequence.

Marshall Clow

```
namespace boost {
  namespace algorithm {
    template<typename InputIterator, typename OutputIterator1,
             typename OutputIterator2, typename UnaryPredicate>
      std::pair< OutputIterator1, OutputIterator2 >
      partition_copy(InputIterator, InputIterator, OutputIterator1,
                     OutputIterator2, UnaryPredicate);
    template<typename Range, typename OutputIterator1,
             typename OutputIterator2, typename UnaryPredicate>
      std::pair< OutputIterator1, OutputIterator2 >
      partition_copy(const Range &, OutputIterator1, OutputIterator2,
                     UnaryPredicate);
  }
}
```

Function template partition_copy

boost::algorithm::partition_copy — Copies the elements that satisfy the predicate p from the range [first, last) to the range beginning at d_first_true, and copies the elements that do not satisfy p to the range beginning at d_first_false.

Synopsis

```
// In header: <boost/algorithm/cxx11/partition_copy.hpp>


template<typename InputIterator, typename OutputIterator1,
         typename OutputIterator2, typename UnaryPredicate>
  std::pair< OutputIterator1, OutputIterator2 >
  partition_copy(InputIterator first, InputIterator last,
                 OutputIterator1 out_true, OutputIterator2 out_false,
                 UnaryPredicate p);
```

Description

Note

This function is part of the C++2011 standard library. We will use the standard one if it is available, otherwise we have our own implementation.

Parameters:		
	`first`	The start of the input sequence
	`last`	One past the end of the input sequence
	`out_false`	An output iterator to write the elements that do not satisfy the predicate into
	`out_true`	An output iterator to write the elements that satisfy the predicate into
	`p`	A predicate for dividing the elements of the input sequence.

Function template partition_copy

boost::algorithm::partition_copy

Synopsis

```
// In header: <boost/algorithm/cxx11/partition_copy.hpp>


template<typename Range, typename OutputIterator1, typename OutputIterator2,
         typename UnaryPredicate>
  std::pair< OutputIterator1, OutputIterator2 >
  partition_copy(const Range & r, OutputIterator1 out_true,
                 OutputIterator2 out_false, UnaryPredicate p);
```

Description

Parameters:		
	`out_false`	An output iterator to write the elements that do not satisfy the predicate into
	`out_true`	An output iterator to write the elements that satisfy the predicate into
	`p`	A predicate for dividing the elements of the input sequence.
	`r`	The input range

Header <boost/algorithm/cxx11/partition_point.hpp>

Find the partition point in a sequence.

Marshall Clow

```
namespace boost {
  namespace algorithm {
    template<typename ForwardIterator, typename Predicate>
      ForwardIterator
      partition_point(ForwardIterator, ForwardIterator, Predicate);
    template<typename Range, typename Predicate>
      boost::range_iterator< Range > partition_point(Range &, Predicate);
  }
}
```

Function template partition_point

boost::algorithm::partition_point — Given a partitioned range, returns the partition point, i.e, the first element that does not satisfy p.

Synopsis

```
// In header: <boost/algorithm/cxx11/partition_point.hpp>


template<typename ForwardIterator, typename Predicate>
  ForwardIterator
  partition_point(ForwardIterator first, ForwardIterator last, Predicate p);
```

Description

Note

This function is part of the C++2011 standard library. We will use the standard one if it is available, otherwise we have our own implementation.

Parameters:		
	`first`	The start of the input sequence
	`last`	One past the end of the input sequence
	`p`	The predicate to test the values with

Function template partition_point

boost::algorithm::partition_point — Given a partitioned range, returns the partition point.

Synopsis

```
// In header: <boost/algorithm/cxx11/partition_point.hpp>


template<typename Range, typename Predicate>
  boost::range_iterator< Range > partition_point(Range & r, Predicate p);
```

Description

Parameters:		
	`p`	The predicate to test the values with
	`r`	The input range

Header <boost/algorithm/cxx14/equal.hpp>

Test ranges to if they are equal.

Determines if one.

Marshall Clow

```
namespace boost {
  namespace algorithm {
    template<typename InputIterator1, typename InputIterator2,
             typename BinaryPredicate>
      bool equal(InputIterator1, InputIterator1, InputIterator2,
                 InputIterator2, BinaryPredicate);
    template<typename InputIterator1, typename InputIterator2>
      bool equal(InputIterator1, InputIterator1, InputIterator2,
                 InputIterator2);
  }
}
```

Function template equal

boost::algorithm::equal

Synopsis

```
// In header: <boost/algorithm/cxx14/equal.hpp>


template<typename InputIterator1, typename InputIterator2,
         typename BinaryPredicate>
  bool equal(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, InputIterator2 last2,
             BinaryPredicate pred);
```

Description

Parameters:	`first1`	The start of the first range.
	`first2`	The start of the second range.
	`last1`	One past the end of the first range.
	`last2`	One past the end of the second range.
	`pred`	A predicate for comparing the elements of the ranges
Returns:	true if all elements in the two ranges are equal	

Function template equal

boost::algorithm::equal

Synopsis

```
// In header: <boost/algorithm/cxx14/equal.hpp>


template<typename InputIterator1, typename InputIterator2>
  bool equal(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, InputIterator2 last2);
```

Description

Parameters:	`first1`	The start of the first range.
	`first2`	The start of the second range.
	`last1`	One past the end of the first range.
	`last2`	One past the end of the second range.
Returns:	true if all elements in the two ranges are equal	

Header <boost/algorithm/cxx14/mismatch.hpp>

Find the first mismatched element in a sequence.

Marshall Clow

```
namespace boost {
  namespace algorithm {
    template<typename InputIterator1, typename InputIterator2,
             typename BinaryPredicate>
      std::pair< InputIterator1, InputIterator2 >
      mismatch(InputIterator1, InputIterator1, InputIterator2, InputIterator2,
               BinaryPredicate);
    template<typename InputIterator1, typename InputIterator2>
      std::pair< InputIterator1, InputIterator2 >
      mismatch(InputIterator1, InputIterator1, InputIterator2, InputIterator2);
  }
}
```

Function template mismatch

boost::algorithm::mismatch

Synopsis

```
// In header: <boost/algorithm/cxx14/mismatch.hpp>


template<typename InputIterator1, typename InputIterator2,
         typename BinaryPredicate>
  std::pair< InputIterator1, InputIterator2 >
  mismatch(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2,
           InputIterator2 last2, BinaryPredicate pred);
```

Description

Parameters:	`first1`	The start of the first range.
	`first2`	The start of the second range.
	`last1`	One past the end of the first range.

	`last2`	One past the end of the second range.
	`pred`	A predicate for comparing the elements of the ranges
Returns:	a pair of iterators pointing to the first elements in the sequence that do not match	

Function template mismatch

boost::algorithm::mismatch

Synopsis

```
// In header: <boost/algorithm/cxx14/mismatch.hpp>


template<typename InputIterator1, typename InputIterator2>
  std::pair< InputIterator1, InputIterator2 >
  mismatch(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2,
           InputIterator2 last2);
```

Description

Parameters:	`first1`	The start of the first range.
	`first2`	The start of the second range.
	`last1`	One past the end of the first range.
	`last2`	One past the end of the second range.
Returns:	a pair of iterators pointing to the first elements in the sequence that do not match	

Header <boost/algorithm/gather.hpp>

```
namespace boost {
  namespace algorithm {

    // iterator-based gather implementation
    template<typename BidirectionalIterator, typename Pred>
      std::pair< BidirectionalIterator, BidirectionalIterator >
      gather(BidirectionalIterator first, BidirectionalIterator last,
             BidirectionalIterator pivot, Pred pred);

    // range-based gather implementation
    template<typename BidirectionalRange, typename Pred>
      std::pair< typename boost::range_iterator< const BidirectionalRange >::type, type↵
name boost::range_iterator< const BidirectionalRange >::type >
      gather(const BidirectionalRange & range,
             typename boost::range_iterator< const BidirectionalRange >::type pivot,
             Pred pred);
  }
}
```

Header <boost/algorithm/hex.hpp>

Convert sequence of integral types into a sequence of hexadecimal characters and back. Based on the MySQL functions HEX and UNHEX.

Marshall Clow

```
namespace boost {
  namespace algorithm {
    struct hex_decode_error;
    struct not_enough_input;
    struct non_hex_input;

    typedef boost::error_info< struct bad_char_, char > bad_char;
    template<typename InputIterator, typename OutputIterator>
      unspecified hex(InputIterator, InputIterator, OutputIterator);
    template<typename T, typename OutputIterator>
      boost::enable_if< boost::is_integral< T >, OutputIterator >::type
      hex(const T *, OutputIterator);
    template<typename Range, typename OutputIterator>
      unspecified hex(const Range &, OutputIterator);
    template<typename InputIterator, typename OutputIterator>
      OutputIterator unhex(InputIterator, InputIterator, OutputIterator);
    template<typename T, typename OutputIterator>
      OutputIterator unhex(const T *, OutputIterator);
    template<typename Range, typename OutputIterator>
      OutputIterator unhex(const Range &, OutputIterator);
    template<typename String> String hex(const String &);
    template<typename String> String unhex(const String &);
  }
}
```

Struct hex_decode_error

boost::algorithm::hex_decode_error — Base exception class for all hex decoding errors.

Synopsis

```
// In header: <boost/algorithm/hex.hpp>


struct hex_decode_error : public exception, public exception {
};
```

Struct not_enough_input

boost::algorithm::not_enough_input — Thrown when the input sequence unexpectedly ends.

Synopsis

```
// In header: <boost/algorithm/hex.hpp>


struct not_enough_input : public boost::algorithm::hex_decode_error {
};
```

Struct non_hex_input

boost::algorithm::non_hex_input — Thrown when a non-hex value (0-9, A-F) encountered when decoding. Contains the offending character.

Synopsis

```
// In header: <boost/algorithm/hex.hpp>


struct non_hex_input : public boost::algorithm::hex_decode_error {
};
```

Function template hex

boost::algorithm::hex — Converts a sequence of integral types into a hexadecimal sequence of characters.

Synopsis

```
// In header: <boost/algorithm/hex.hpp>


template<typename InputIterator, typename OutputIterator>
  unspecified hex(InputIterator first, InputIterator last, OutputIterator out);
```

Description

Note

Based on the MySQL function of the same name

Parameters:

`first`	The start of the input sequence
`last`	One past the end of the input sequence
`out`	An output iterator to the results into

Returns: The updated output iterator

Function template hex

boost::algorithm::hex — Converts a sequence of integral types into a hexadecimal sequence of characters.

Synopsis

```
// In header: <boost/algorithm/hex.hpp>


template<typename T, typename OutputIterator>
  boost::enable_if< boost::is_integral< T >, OutputIterator >::type
  hex(const T * ptr, OutputIterator out);
```

Description

Note

Based on the MySQL function of the same name

Parameters:

`out`	An output iterator to the results into

`ptr` A pointer to a 0-terminated sequence of data.

Returns: The updated output iterator

Function template hex

boost::algorithm::hex — Converts a sequence of integral types into a hexadecimal sequence of characters.

Synopsis

```
// In header: <boost/algorithm/hex.hpp>


template<typename Range, typename OutputIterator>
  unspecified hex(const Range & r, OutputIterator out);
```

Description

Note

Based on the MySQL function of the same name

Parameters: `out` An output iterator to the results into

`r` The input range

Returns: The updated output iterator

Function template unhex

boost::algorithm::unhex — Converts a sequence of hexadecimal characters into a sequence of integers.

Synopsis

```
// In header: <boost/algorithm/hex.hpp>


template<typename InputIterator, typename OutputIterator>
  OutputIterator
  unhex(InputIterator first, InputIterator last, OutputIterator out);
```

Description

Note

Based on the MySQL function of the same name

Parameters: `first` The start of the input sequence

`last` One past the end of the input sequence

`out` An output iterator to the results into

Returns: The updated output iterator

Function template unhex

boost::algorithm::unhex — Converts a sequence of hexadecimal characters into a sequence of integers.

Synopsis

```
// In header: <boost/algorithm/hex.hpp>


template<typename T, typename OutputIterator>
  OutputIterator unhex(const T * ptr, OutputIterator out);
```

Description

Note

Based on the MySQL function of the same name

Parameters: `out` An output iterator to the results into
`ptr` A pointer to a null-terminated input sequence.
Returns: The updated output iterator

Function template unhex

boost::algorithm::unhex — Converts a sequence of hexadecimal characters into a sequence of integers.

Synopsis

```
// In header: <boost/algorithm/hex.hpp>


template<typename Range, typename OutputIterator>
  OutputIterator unhex(const Range & r, OutputIterator out);
```

Description

Note

Based on the MySQL function of the same name

Parameters: `out` An output iterator to the results into
`r` The input range
Returns: The updated output iterator

Function template hex

boost::algorithm::hex — Converts a sequence of integral types into a hexadecimal sequence of characters.

Synopsis

```
// In header: <boost/algorithm/hex.hpp>


template<typename String> String hex(const String & input);
```

Description

Parameters: `input` A container to be converted
Returns: A container with the encoded text

Function template unhex

boost::algorithm::unhex — Converts a sequence of hexadecimal characters into a sequence of characters.

Synopsis

```
// In header: <boost/algorithm/hex.hpp>


template<typename String> String unhex(const String & input);
```

Description

Parameters: `input` A container to be converted
Returns: A container with the decoded text

Header <boost/algorithm/minmax.hpp>

```
namespace boost {
  template<typename T>
    tuple< T const &, T const & > minmax(T const & a, T const & b);
  template<typename T, typename BinaryPredicate>
    tuple< T const &, T const & >
    minmax(T const & a, T const & b, BinaryPredicate comp);
}
```

Header <boost/algorithm/minmax_element.hpp>

```
namespace boost {
  template<typename ForwardIter>
    std::pair< ForwardIter, ForwardIter >
    minmax_element(ForwardIter first, ForwardIter last);
  template<typename ForwardIter, typename BinaryPredicate>
    std::pair< ForwardIter, ForwardIter >
    minmax_element(ForwardIter first, ForwardIter last, BinaryPredicate comp);
  template<typename ForwardIter>
    ForwardIter first_min_element(ForwardIter first, ForwardIter last);
  template<typename ForwardIter, typename BinaryPredicate>
    ForwardIter first_min_element(ForwardIter first, ForwardIter last,
                                  BinaryPredicate comp);
  template<typename ForwardIter>
    ForwardIter last_min_element(ForwardIter first, ForwardIter last);
  template<typename ForwardIter, typename BinaryPredicate>
    ForwardIter last_min_element(ForwardIter first, ForwardIter last,
                                 BinaryPredicate comp);
  template<typename ForwardIter>
    ForwardIter first_max_element(ForwardIter first, ForwardIter last);
  template<typename ForwardIter, typename BinaryPredicate>
    ForwardIter first_max_element(ForwardIter first, ForwardIter last,
                                  BinaryPredicate comp);
  template<typename ForwardIter>
    ForwardIter last_max_element(ForwardIter first, ForwardIter last);
  template<typename ForwardIter, typename BinaryPredicate>
    ForwardIter last_max_element(ForwardIter first, ForwardIter last,
                                 BinaryPredicate comp);
  template<typename ForwardIter>
    std::pair< ForwardIter, ForwardIter >
    first_min_first_max_element(ForwardIter first, ForwardIter last);
  template<typename ForwardIter, typename BinaryPredicate>
    std::pair< ForwardIter, ForwardIter >
    first_min_first_max_element(ForwardIter first, ForwardIter last,
                                BinaryPredicate comp);
  template<typename ForwardIter>
    std::pair< ForwardIter, ForwardIter >
    first_min_last_max_element(ForwardIter first, ForwardIter last);
  template<typename ForwardIter, typename BinaryPredicate>
    std::pair< ForwardIter, ForwardIter >
    first_min_last_max_element(ForwardIter first, ForwardIter last,
                               BinaryPredicate comp);
  template<typename ForwardIter>
    std::pair< ForwardIter, ForwardIter >
    last_min_first_max_element(ForwardIter first, ForwardIter last);
  template<typename ForwardIter, typename BinaryPredicate>
    std::pair< ForwardIter, ForwardIter >
    last_min_first_max_element(ForwardIter first, ForwardIter last,
                               BinaryPredicate comp);
  template<typename ForwardIter>
    std::pair< ForwardIter, ForwardIter >
    last_min_last_max_element(ForwardIter first, ForwardIter last);
  template<typename ForwardIter, typename BinaryPredicate>
    std::pair< ForwardIter, ForwardIter >
    last_min_last_max_element(ForwardIter first, ForwardIter last,
                              BinaryPredicate comp);
}
```

Header <boost/algorithm/searching/boyer_moore.hpp>

```
namespace boost {
  namespace algorithm {
    template<typename patIter, typename traits = detail::BM_traits<patIter> >
      class boyer_moore;
    template<typename patIter, typename corpusIter>
      corpusIter boyer_moore_search(corpusIter, corpusIter, patIter, patIter);
    template<typename PatternRange, typename corpusIter>
      corpusIter boyer_moore_search(corpusIter corpus_first,
                                    corpusIter corpus_last,
                                    const PatternRange & pattern);
    template<typename patIter, typename CorpusRange>
      boost::lazy_disable_if_c< boost::is_same< CorpusRange, patIter >::value, type↵
name boost::range_iterator< CorpusRange > >::type
      boyer_moore_search(CorpusRange & corpus, patIter pat_first,
                         patIter pat_last);
    template<typename PatternRange, typename CorpusRange>
      boost::range_iterator< CorpusRange >::type
      boyer_moore_search(CorpusRange & corpus, const PatternRange & pattern);
    template<typename Range>
      boost::algorithm::boyer_moore< typename boost::range_iterator< const Range >::type >
      make_boyer_moore(const Range & r);
    template<typename Range>
      boost::algorithm::boyer_moore< typename boost::range_iterator< Range >::type >
      make_boyer_moore(Range & r);
  }
}
```

Class template boyer_moore

boost::algorithm::boyer_moore

Synopsis

```
// In header: <boost/algorithm/searching/boyer_moore.hpp>

template<typename patIter, typename traits = detail::BM_traits<patIter> >
class boyer_moore {
public:
  // construct/copy/destruct
  boyer_moore(patIter, patIter);
  ~boyer_moore();

  // public member functions
  template<typename corpusIter>
    corpusIter operator()(corpusIter, corpusIter) const;
  template<typename Range>
    boost::range_iterator< Range >::type operator()(Range &) const;
};
```

Description

boyer_moore public construct/copy/destruct

1. ```
   boyer_moore(patIter first, patIter last);
   ```
   ```

2.
```
~boyer_moore();
```

`boyer_moore` public member functions

1.
```
template<typename corpusIter>
  corpusIter operator()(corpusIter corpus_first, corpusIter corpus_last) const;
```

2.
```
template<typename Range>
  boost::range_iterator< Range >::type operator()(Range & r) const;
```

Function template boyer_moore_search

boost::algorithm::boyer_moore_search — Searches the corpus for the pattern.

Synopsis

```
// In header: <boost/algorithm/searching/boyer_moore.hpp>


template<typename patIter, typename corpusIter>
  corpusIter boyer_moore_search(corpusIter corpus_first,
                                corpusIter corpus_last, patIter pat_first,
                                patIter pat_last);
```

Description

Parameters:	`corpus_first`	The start of the data to search (Random Access Iterator)
	`corpus_last`	One past the end of the data to search
	`pat_first`	The start of the pattern to search for (Random Access Iterator)
	`pat_last`	One past the end of the data to search for

Header <boost/algorithm/searching/boyer_moore_horspool.hpp>

```
namespace boost {
  namespace algorithm {
    template<typename patIter, typename traits = detail::BM_traits<patIter> >
      class boyer_moore_horspool;
    template<typename patIter, typename corpusIter>
      corpusIter boyer_moore_horspool_search(corpusIter, corpusIter, patIter,
                                             patIter);
    template<typename PatternRange, typename corpusIter>
      corpusIter boyer_moore_horspool_search(corpusIter corpus_first,
                                             corpusIter corpus_last,
                                             const PatternRange & pattern);
    template<typename patIter, typename CorpusRange>
      boost::lazy_disable_if_c< boost::is_same< CorpusRange, patIter >::value, type↵
name boost::range_iterator< CorpusRange > >::type
      boyer_moore_horspool_search(CorpusRange & corpus, patIter pat_first,
                                  patIter pat_last);
    template<typename PatternRange, typename CorpusRange>
      boost::range_iterator< CorpusRange >::type
      boyer_moore_horspool_search(CorpusRange & corpus,
                                  const PatternRange & pattern);
    template<typename Range>
      boost::algorithm::boyer_moore_horspool< typename boost::range_iterat↵
or< const Range >::type >
      make_boyer_moore_horspool(const Range & r);
    template<typename Range>
      boost::algorithm::boyer_moore_horspool< typename boost::range_iterator< Range >::type >
      make_boyer_moore_horspool(Range & r);
  }
}
```

Class template boyer_moore_horspool

boost::algorithm::boyer_moore_horspool

Synopsis

```
// In header: <boost/algorithm/searching/boyer_moore_horspool.hpp>

template<typename patIter, typename traits = detail::BM_traits<patIter> >
class boyer_moore_horspool {
public:
  // construct/copy/destruct
  boyer_moore_horspool(patIter, patIter);
  ~boyer_moore_horspool();

  // public member functions
  template<typename corpusIter>
    corpusIter operator()(corpusIter, corpusIter) const;
  template<typename Range>
    boost::range_iterator< Range >::type operator()(Range &) const;
};
```

Description

`boyer_moore_horspool` public construct/copy/destruct

1.
```
boyer_moore_horspool(patIter first, patIter last);
```

2.
```
~boyer_moore_horspool();
```

`boyer_moore_horspool` public member functions

1.
```
template<typename corpusIter>
  corpusIter operator()(corpusIter corpus_first, corpusIter corpus_last) const;
```

2.
```
template<typename Range>
  boost::range_iterator< Range >::type operator()(Range & r) const;
```

Function template boyer_moore_horspool_search

boost::algorithm::boyer_moore_horspool_search — Searches the corpus for the pattern.

Synopsis

```
// In header: <boost/algorithm/searching/boyer_moore_horspool.hpp>


template<typename patIter, typename corpusIter>
  corpusIter boyer_moore_horspool_search(corpusIter corpus_first,
                                         corpusIter corpus_last,
                                         patIter pat_first, patIter pat_last);
```

Description

Parameters:	`corpus_first`	The start of the data to search (Random Access Iterator)
	`corpus_last`	One past the end of the data to search
	`pat_first`	The start of the pattern to search for (Random Access Iterator)
	`pat_last`	One past the end of the data to search for

Header <boost/algorithm/searching/knuth_morris_pratt.hpp>

```
namespace boost {
  namespace algorithm {
    template<typename patIter> class knuth_morris_pratt;
    template<typename patIter, typename corpusIter>
      corpusIter knuth_morris_pratt_search(corpusIter, corpusIter, patIter,
                                           patIter);
    template<typename PatternRange, typename corpusIter>
      corpusIter knuth_morris_pratt_search(corpusIter corpus_first,
                                           corpusIter corpus_last,
                                           const PatternRange & pattern);
    template<typename patIter, typename CorpusRange>
      boost::lazy_disable_if_c< boost::is_same< CorpusRange, patIter >::value, type↵
name boost::range_iterator< CorpusRange > >::type
      knuth_morris_pratt_search(CorpusRange & corpus, patIter pat_first,
                                patIter pat_last);
    template<typename PatternRange, typename CorpusRange>
      boost::range_iterator< CorpusRange >::type
      knuth_morris_pratt_search(CorpusRange & corpus,
                                const PatternRange & pattern);
    template<typename Range>
     boost::algorithm::knuth_morris_pratt< typename boost::range_iterator< const Range >::type >
      make_knuth_morris_pratt(const Range & r);
    template<typename Range>
      boost::algorithm::knuth_morris_pratt< typename boost::range_iterator< Range >::type >
      make_knuth_morris_pratt(Range & r);
  }
}
```

Class template knuth_morris_pratt

boost::algorithm::knuth_morris_pratt

Synopsis

```
// In header: <boost/algorithm/searching/knuth_morris_pratt.hpp>

template<typename patIter>
class knuth_morris_pratt {
public:
  // construct/copy/destruct
  knuth_morris_pratt(patIter, patIter);
  ~knuth_morris_pratt();

  // public member functions
  template<typename corpusIter>
    corpusIter operator()(corpusIter, corpusIter) const;
  template<typename Range>
    boost::range_iterator< Range >::type operator()(Range &) const;
};
```

Description

`knuth_morris_pratt` public construct/copy/destruct

1. ```
   knuth_morris_pratt(patIter first, patIter last);
   ```
   ```

2.
```
~knuth_morris_pratt();
```

`knuth_morris_pratt` public member functions

1.
```
template<typename corpusIter>
  corpusIter operator()(corpusIter corpus_first, corpusIter corpus_last) const;
```

2.
```
template<typename Range>
  boost::range_iterator< Range >::type operator()(Range & r) const;
```

Function template knuth_morris_pratt_search

boost::algorithm::knuth_morris_pratt_search — Searches the corpus for the pattern.

Synopsis

```
// In header: <boost/algorithm/searching/knuth_morris_pratt.hpp>


template<typename patIter, typename corpusIter>
  corpusIter knuth_morris_pratt_search(corpusIter corpus_first,
                                       corpusIter corpus_last,
                                       patIter pat_first, patIter pat_last);
```

Description

Parameters:	`corpus_first`	The start of the data to search (Random Access Iterator)
	`corpus_last`	One past the end of the data to search
	`pat_first`	The start of the pattern to search for (Random Access Iterator)
	`pat_last`	One past the end of the data to search for

Header <boost/algorithm/string.hpp>

Cumulative include for string_algo library

Header <boost/algorithm/string_regex.hpp>

Cumulative include for string_algo library. In addition to string.hpp contains also regex-related stuff.