Boost String Algorithms Library

Pavol Droba

Table of Contents

Introduction 2
Release Notes 2
Usage 3
- First Example 3
- Case conversion 3
- Predicates and Classification 4
- Trimming 4
- Find algorithms 5
- Replace Algorithms 5
- Find Iterator 6
- Split 7

Quick Reference 7
- Algorithms 7
- Finders and Formatters 12
- Iterators 12
- Classification 13

Design Topics 13
- String Representation 13
- Sequence Traits 14
- Find Algorithms 14
- Replace Algorithms 14
- Find Iterators & Split Algorithms 15
- Exception Safety 15

Concepts 15
- Definitions 15
- Finder Concept 15
- Formatter concept 16

Reference 17
- Header <boost/algorithm/string.hpp> 17
- Header <boost/algorithm/string/case_conv.hpp> 17
- Header <boost/algorithm/string/classification.hpp> 22
- Header <boost/algorithm/string/compare.hpp> 39
- Header <boost/algorithm/string/concept.hpp> 45
- Header <boost/algorithm/string/constants.hpp> 47
- Header <boost/algorithm/string/erase.hpp> 48
- Header <boost/algorithm/string/find.hpp> 72
- Header <boost/algorithm/string/find_format.hpp> 83
- Header <boost/algorithm/string/find_iterator.hpp> 87
- Header <boost/algorithm/string/finder.hpp> 93
- Header <boost/algorithm/string/formatter.hpp> 100
- Header <boost/algorithm/string/iter_find.hpp> 103
- Header <boost/algorithm/string/join.hpp> 105
- Header <boost/algorithm/string/predicate.hpp> 107
- Header <boost/algorithm/string/regex.hpp> 119

Header <boost/algorithm/string/regex_find_format.hpp> 133
Header <boost/algorithm/string/replace.hpp> 135
Header <boost/algorithm/string/sequence_traits.hpp> 159
Header <boost/algorithm/string/split.hpp> 163
Header <boost/algorithm/string/std_containers_traits.hpp> 166
Header <boost/algorithm/string/trim.hpp> 166
Header <boost/algorithm/string_regex.hpp> 179
Rationale 179
Locales 179
Regular Expressions 179
Environment 179
Build 179
Examples 180
Tests 180
Portability 180
Credits 180
Acknowledgments 180

Introduction

The String Algorithm Library provides a generic implementation of string-related algorithms which are missing in STL. It is an extension to the algorithms library of STL and it includes trimming, case conversion, predicates and find/replace functions. All of them come in different variants so it is easier to choose the best fit for a particular need.

The implementation is not restricted to work with a particular container (like `std::basic_string`), rather it is as generic as possible. This generalization is not compromising the performance since algorithms are using container specific features when it means a performance gain.

Important note: In this documentation we use term string to designate a sequence of characters stored in an arbitrary container. A string is not restricted to `std::basic_string` and character does not have to be `char` or `wchar_t`, although these are most common candidates. Consult the design chapter to see precise specification of supported string types.

The library interface functions and classes are defined in namespace `boost::algorithm`, and they are lifted into namespace `boost` via using declaration.

The documentation is divided into several sections. For a quick start read the Usage section followed by Quick Reference. The Design Topics, Concepts and Rationale provide some explanation about the library design and structure an explain how it should be used. See the Reference for the complete list of provided utilities and algorithms. Functions and classes in the reference are organized by the headers in which they are defined. The reference contains links to the detailed description for every entity in the library.

Release Notes

- **1.32**

  Initial release in Boost

- **1.33**

  Internal version of collection traits removed, library adapted to Boost.Range

- **1.34**

  - `lexicographical_compare()`
  - `join()` and `join_if()`
  - New comparison predicates `is_less`, `is_not_greater`

- Negative indexes support (like Perl) in various algorihtms (`*_head/tail`, `*_nth`).

Usage

First Example

Using the algorithms is straightforward. Let us have a look at the first example:

```
#include <boost/algorithm/string.hpp>
using namespace std;
using namespace boost;

// ...

string str1(" hello world! ");
to_upper(str1);  // str1 == " HELLO WORLD! "
trim(str1);      // str1 == "HELLO WORLD!"

string str2=
   to_lower_copy(
      ireplace_first_copy(
         str1,"hello","goodbye")); // str2 == "goodbye world!"
    ↵
```

This example converts str1 to upper case and trims spaces from the start and the end of the string. str2 is then created as a copy of str1 with "hello" replaced with "goodbye". This example demonstrates several important concepts used in the library:

- **Container parameters:** Unlike in the STL algorithms, parameters are not specified only in the form of iterators. The STL convention allows for great flexibility, but it has several limitations. It is not possible to *stack* algorithms together, because a container is passed in two parameters. Therefore it is not possible to use a return value from another algorithm. It is considerably easier to write `to_lower(str1)`, than `to_lower(str1.begin(), str1.end())`.

  The magic of Boost.Range provides a uniform way of handling different string types. If there is a need to pass a pair of iterators, `boost::iterator_range` can be used to package iterators into a structure with a compatible interface.

- **Copy vs. Mutable:** Many algorithms in the library are performing a transformation of the input. The transformation can be done in-place, mutating the input sequence, or a copy of the transformed input can be created, leaving the input intact. None of these possibilities is superior to the other one and both have different advantages and disadvantages. For this reason, both are provided with the library.

- **Algorithm stacking:** Copy versions return a transformed input as a result, thus allow a simple chaining of transformations within one expression (i.e. one can write `trim_copy(to_upper_copy(s))`). Mutable versions have `void` return, to avoid misuse.

- **Naming:** Naming follows the conventions from the Standard C++ Library. If there is a copy and a mutable version of the same algorithm, the mutable version has no suffix and the copy version has the suffix *_copy*. Some algorithms have the prefix *i* (e.g. `ifind_first()`). This prefix identifies that the algorithm works in a case-insensitive manner.

To use the library, include the `boost/algorithm/string.hpp` header. If the regex related functions are needed, include the `boost/algorithm/string_regex.hpp` header.

Case conversion

STL has a nice way of converting character case. Unfortunately, it works only for a single character and we want to convert a string,

```
string str1("HeLlO WoRld!");
to_upper(str1); // str1=="HELLO WORLD!"
    ↵
```

`to_upper()` and `to_lower()` convert the case of characters in a string using a specified locale.

For more information see the reference for `boost/algorithm/string/case_conv.hpp`.

Predicates and Classification

A part of the library deals with string related predicates. Consider this example:

```
bool is_executable( string& filename )
{
    return
        iends_with(filename, ".exe") ||
        iends_with(filename, ".com");
}

// ...
string str1("command.com");
cout
    << str1
    << (is_executable("command.com")? "is": "is not")
    << "an executable"
    << endl; // prints "command.com is an executable"

//..
char text1[]="hello world!";
cout
    << text1
    << (all( text1, is_lower() )? "is": "is not")
    << " written in the lower case"
    << endl; // prints "hello world! is written in the lower case"
    ↵
```

The predicates determine whether if a substring is contained in the input string under various conditions. The conditions are: a string starts with the substring, ends with the substring, simply contains the substring or if both strings are equal. See the reference for `boost/algorithm/string/predicate.hpp` for more details.

In addition the algorithm `all()` checks all elements of a container to satisfy a condition specified by a predicate. This predicate can be any unary predicate, but the library provides a bunch of useful string-related predicates and combinators ready for use. These are located in the `boost/algorithm/string/classification.hpp` header. Classification predicates can be combined using logical combinators to form a more complex expressions. For example: `is_from_range('a','z') || is_digit()`

Trimming

When parsing the input from a user, strings usually have unwanted leading or trailing characters. To get rid of them, we need trim functions:

```
string str1="     hello world!     ";
string str2=trim_left_copy(str1);   // str2 == "hello world!     "
string str3=trim_right_copy(str1);  // str3 == "     hello world!"
trim(str1);                         // str1 == "hello world!"

string phone="00423333444";
// remove leading 0 from the phone number
trim_left_if(phone,is_any_of("0")); // phone == "423333444"
    ↵
```

It is possible to trim the spaces on the right, on the left or on both sides of a string. And for those cases when there is a need to remove something else than blank space, there are *_if* variants. Using these, a user can specify a functor which will select the *space* to be removed. It is possible to use classification predicates like `is_digit()` mentioned in the previous paragraph. See the reference for the `boost/algorithm/string/trim.hpp`.

Find algorithms

The library contains a set of find algorithms. Here is an example:

```
char text[]="hello dolly!";
iterator_range<char*> result=find_last(text,"ll");

transform( result.begin(), result.end(), result.begin(), bind2nd(plus<char>(), 1) );
// text = "hello dommy!"

to_upper(result); // text == "hello doMMy!"

// iterator_range is convertible to bool
if(find_first(text, "dolly"))
{
    cout << "Dolly is there" << endl;
}
    ↵
```

We have used `find_last()` to search the `text` for "ll". The result is given in the `boost::iterator_range`. This range delimits the part of the input which satisfies the find criteria. In our example it is the last occurrence of "ll". As we can see, input of the `find_last()` algorithm can be also char[] because this type is supported by Boost.Range. The following lines transform the result. Notice that `boost::iterator_range` has familiar `begin()` and `end()` methods, so it can be used like any other STL container. Also it is convertible to bool therefore it is easy to use find algorithms for a simple containment checking.

Find algorithms are located in `boost/algorithm/string/find.hpp`.

Replace Algorithms

Find algorithms can be used for searching for a specific part of string. Replace goes one step further. After a matching part is found, it is substituted with something else. The substitution is computed from the original, using some transformation.

```
string str1="Hello  Dolly,   Hello World!"
replace_first(str1, "Dolly", "Jane");      // str1 == "Hello  Jane,   Hello World!"
replace_last(str1, "Hello", "Goodbye");    // str1 == "Hello  Jane,   Goodbye World!"
erase_all(str1, " ");                      // str1 == "HelloJane,GoodbyeWorld!"
erase_head(str1, 6);                       // str1 == "Jane,GoodbyeWorld!"
    ↵
```

For the complete list of replace and erase functions see the reference. There is a lot of predefined function for common usage, however, the library allows you to define a custom `replace()` that suits a specific need. There is a generic `find_format()` function which takes two parameters. The first one is a Finder object, the second one is a Formatter object. The Finder object is a functor which performs the searching for the replacement part. The Formatter object takes the result of the Finder (usually a reference to the found substring) and creates a substitute for it. Replace algorithm puts these two together and makes the desired substitution.

Check `boost/algorithm/string/replace.hpp`, `boost/algorithm/string/erase.hpp` and `boost/algorithm/string/find_format.hpp` for reference.

Find Iterator

An extension to find algorithms it the Find Iterator. Instead of searching for just a one part of a string, the find iterator allows us to iterate over the substrings matching the specified criteria. This facility is using the Finder to incrementally search the string. Dereferencing a find iterator yields an `boost::iterator_range` object, that delimits the current match.

There are two iterators provided `find_iterator` and `split_iterator`. The former iterates over substrings that are found using the specified Finder. The latter iterates over the gaps between these substrings.

```
string str1("abc-*-ABC-*-aBc");
// Find all 'abc' substrings (ignoring the case)
// Create a find_iterator
typedef find_iterator<string::iterator> string_find_iterator;
for(string_find_iterator It=
        make_find_iterator(str1, first_finder("abc", is_iequal()));
    It!=string_find_iterator();
    ++It)
{
    cout << copy_range<std::string>(*It) << endl;
}

// Output will be:
// abc
// ABC
// aBC

typedef split_iterator<string::iterator> string_split_iterator;
for(string_split_iterator It=
    make_split_iterator(str1, first_finder("-*-", is_iequal()));
    It!=string_split_iterator();
    ++It)
{
    cout << copy_range<std::string>(*It) << endl;
}

// Output will be:
// abc
// ABC
// aBC
    ↵
```

Note that the find iterators have only one template parameter. It is the base iterator type. The Finder is specified at runtime. This allows us to typedef a find iterator for common string types and reuse it. Additionally make_*_iterator functions help to construct a find iterator for a particular range.

See the reference in `boost/algorithm/string/find_iterator.hpp`.

Split

Split algorithms are an extension to the find iterator for one common usage scenario. These algorithms use a find iterator and store all matches into the provided container. This container must be able to hold copies (e.g. `std::string`) or references (e.g. `iterator_range`) of the extracted substrings.

Two algorithms are provided. `find_all()` finds all copies of a string in the input. `split()` splits the input into parts.

```
    string str1("hello abc-*-ABC-*-aBc goodbye");

    typedef vector< iterator_range<string::iterator> > find_vector_type;

    find_vector_type FindVec; // #1: Search for separators
    ifind_all( FindVec, str1, "abc" ); // FindVec == { [abc],[ABC],[aBc] }

    typedef vector< string > split_vector_type;

    split_vector_type SplitVec; // #2: Search for tokens
    split( SplitVec, str1, is_any_of("-*"), token_compress_on ); // SplitVec == { "hello ↵
abc","ABC","aBc goodbye" }
        ↵
```

`[hello]` designates an `iterator_range` delimiting this substring.

First example show how to construct a container to hold references to all extracted substrings. Algorithm `ifind_all()` puts into FindVec references to all substrings that are in case-insensitive manner equal to "abc".

Second example uses `split()` to split string str1 into parts separated by characters '-' or '*'. These parts are then put into the SplitVec. It is possible to specify if adjacent separators are concatenated or not.

More information can be found in the reference: `boost/algorithm/string/split.hpp`.

Quick Reference

Algorithms

Table 1. Case Conversion

Algorithm name	Description	Functions
`to_upper`	Convert a string to upper case	`to_upper_copy()` `to_upper()`
`to_lower`	Convert a string to lower case	`to_lower_copy()` `to_lower()`

Table 2. Trimming

Algorithm name	Description	Functions

`trim_left`	Remove leading spaces from a string	`trim_left_copy_if()`
		`trim_left_if()`
		`trim_left_copy()`
		`trim_left()`

`trim_right`	Remove trailing spaces from a string	`trim_right_copy_if()`
		`trim_right_if()`
		`trim_right_copy()`
		`trim_right()`

`trim`	Remove leading and trailing spaces from a string	`trim_copy_if()`
		`trim_if()`
		`trim_copy()`
		`trim()`

Table 3. Predicates

Algorithm name	Description	Functions
`starts_with`	Check if a string is a prefix of the other one	`starts_with()` `istarts_with()`
`ends_with`	Check if a string is a suffix of the other one	`ends_with()` `iends_with()`
`contains`	Check if a string is contained of the other one	`contains()` `icontains()`
`equals`	Check if two strings are equal	`equals()` `iequals()`
`lexicographical_compare`	Check if a string is lexicographically less then another one	`lexicographical_compare()` `ilexicographical_compare()`
`all`	Check if all elements of a string satisfy the given predicate	`all()`

Table 4. Find algorithms

Algorithm name	Description	Functions
find_first	Find the first occurrence of a string in the input	`find_first()` `ifind_first()`
find_last	Find the last occurrence of a string in the input	`find_last()` `ifind_last()`
find_nth	Find the nth (zero-indexed) occurrence of a string in the input	`find_nth()` `ifind_nth()`
find_head	Retrieve the head of a string	`find_head()`
find_tail	Retrieve the tail of a string	`find_tail()`
find_token	Find first matching token in the string	`find_token()`
find_regex	Use the regular expression to search the string	`find_regex()`
find	Generic find algorithm	`find()`

Table 5. Erase/Replace

Algorithm name	Description	Functions

replace/erase_first	Replace/Erase the first occurrence of a string in the input	`replace_first()`
		`replace_first_copy()`
		`ireplace_first()`
		`ireplace_first_copy()`
		`erase_first()`
		`erase_first_copy()`
		`ierase_first()`
		`ierase_first_copy()`

replace/erase_last	Replace/Erase the last occurrence of a string in the input	`replace_last()`
		`replace_last_copy()`
		`ireplace_last()`
		`ireplace_last_copy()`
		`erase_last()`
		`erase_last_copy()`
		`ierase_last()`
		`ierase_last_copy()`

replace/erase_nth	Replace/Erase the nth (zero-indexed) occurrence of a string in the input	`replace_nth()`
		`replace_nth_copy()`
		`ireplace_nth()`
		`ireplace_nth_copy()`
		`erase_nth()`
		`erase_nth_copy()`
		`ierase_nth()`
		`ierase_nth_copy()`

replace/erase_all	Replace/Erase the all occurrences of a string in the input	`replace_all()`
		`replace_all_copy()`
		`ireplace_all()`
		`ireplace_all_copy()`
		`erase_all()`
		`erase_all_copy()`
		`ierase_all()`
		`ierase_all_copy()`

replace/erase_head	Replace/Erase the head of the input	`replace_head()`
		`replace_head_copy()`
		`erase_head()`
		`erase_head_copy()`

replace/erase_tail	Replace/Erase the tail of the input	`replace_tail()`
		`replace_tail_copy()`
		`erase_tail()`
		`erase_tail_copy()`

replace/erase_regex	Replace/Erase a substring matching the given regular expression	`replace_regex()`
		`replace_regex_copy()`
		`erase_regex()`
		`erase_regex_copy()`

Algorithm name	Description	Functions

replace/erase_regex_all	Replace/Erase all substrings matching the given regular expression	`replace_all_regex()`
		`replace_all_regex_copy()`
		`erase_all_regex()`
		`erase_all_regex_copy()`

find_format	Generic replace algorithm	`find_format()`
		`find_format_copy()`
		`find_format_all()`
		`find_format_all_copy()()`

Table 6. Split

Algorithm name	Description	Functions

find_all	Find/Extract all matching substrings in the input	`find_all()`
		`ifind_all()`
		`find_all_regex()`

split	Split input into parts	`split()` `split_regex()`
iter_find	Iteratively apply the finder to the input to find all matching substrings	`iter_find()`
iter_split	Use the finder to find matching substrings in the input and use them as separators to split the input into parts	`iter_split()`

Table 7. Join

Algorithm name	Description	Functions
join	Join all elements in a container into a single string	`join`
join_if	Join all elements in a container that satisfies the condition into a single string	`join_if()`

Finders and Formatters

Table 8. Finders

Finder	Description	Generators
first_finder	Search for the first match of the string in an input	`first_finder()`
last_finder	Search for the last match of the string in an input	`last_finder()`
nth_finder	Search for the nth (zero-indexed) match of the string in an input	`nth_finder()`
head_finder	Retrieve the head of an input	`head_finder()`
tail_finder	Retrieve the tail of an input	`tail_finder()`
token_finder	Search for a matching token in an input	`token_finder()`
range_finder	Do no search, always returns the given range	`range_finder()`
regex_finder	Search for a substring matching the given regex	`regex_finder()`

Table 9. Formatters

Formatter	Description	Generators
const_formatter	Constant formatter. Always return the specified string	`const_formatter()`
identity_formatter	Identity formatter. Return unmodified input input	`identity_formatter()`
empty_formatter	Null formatter. Always return an empty string	`empty_formatter()`
regex_formatter	Regex formatter. Format regex match using the specification in the format string	`regex_formatter()`

Iterators

Table 10. Find Iterators

Iterator name	Description	Iterator class
find_iterator	Iterates through matching substrings in the input	`find_iterator`
split_iterator	Iterates through gaps between matching substrings in the input	`split_iterator`

Classification

Table 11. Predicates

Predicate name	Description	Generator
is_classified	Generic `ctype` mask based classification	`is_classified()`
is_space	Recognize spaces	`is_space()`
is_alnum	Recognize alphanumeric characters	`is_alnum()`
is_alpha	Recognize letters	`is_alpha()`
is_cntrl	Recognize control characters	`is_cntrl()`
is_digit	Recognize decimal digits	`is_digit()`
is_graph	Recognize graphical characters	`is_graph()`
is_lower	Recognize lower case characters	`is_lower()`
is_print	Recognize printable characters	`is_print()`
is_punct	Recognize punctuation characters	`is_punct()`
is_upper	Recognize uppercase characters	`is_upper()`
is_xdigit	Recognize hexadecimal digits	`is_xdigit()`

Design Topics

String Representation

As the name suggest, this library works mainly with strings. However, in the context of this library, a string is not restricted to any particular implementation (like `std::basic_string`), rather it is a concept. This allows the algorithms in this library to be reused for any string type, that satisfies the given requirements.

Definition: A string is a range of characters accessible in sequential ordered fashion. Character is any value type with "cheap" copying and assignment.

First requirement of string-type is that it must accessible using Boost.Range. This facility allows to access the elements inside the string in a uniform iterator-based fashion. This is sufficient for our library

Second requirement defines the way in which the characters are stored in the string. Algorithms in this library work with an assumption that copying a character is cheaper then allocating extra storage to cache results. This is a natural assumption for common character types. Algorithms will work even if this requirement is not satisfied, however at the cost of performance degradation.

In addition some algorithms have additional requirements on the string-type. Particularly, it is required that an algorithm can create a new string of the given type. In this case, it is required that the type satisfies the sequence (Std §23.1.1) requirements.

In the reference and also in the code, requirement on the string type is designated by the name of template argument. `RangeT` means that the basic range requirements must hold. `SequenceT` designates extended sequence requirements.

Sequence Traits

The major difference between `std::list` and `std::vector` is not in the interfaces they provide, but rather in the inner details of the class and the way how it performs various operations. The problem is that it is not possible to infer this difference from the definitions of classes without some special mechanism. However, some algorithms can run significantly faster with the knowledge of the properties of a particular container.

Sequence traits allow one to specify additional properties of a sequence container (see Std.§32.2). These properties are then used by algorithms to select optimized handling for some operations. The sequence traits are declared in the header `boost/algorithm/string/sequence_traits.hpp`.

In the table C denotes a container and c is an object of C.

Table 12. Sequence Traits

Trait	Description
`has_native_replace<C>`::value	Specifies that the sequence has std::string like replace method
`has_stable_iterators<C>`::value	Specifies that the sequence has stable iterators. It means, that operations like `insert`/`erase`/`replace` do not invalidate iterators.
`has_const_time_insert<C>`::value	Specifies that the insert method of the sequence has constant time complexity.
`has_const_time_erase<C>`::value	Specifies that the erase method of the sequence has constant time complexity

Current implementation contains specializations for std::list<T> and std::basic_string<T> from the standard library and SGI's std::rope<T> and std::slist<T>.

Find Algorithms

Find algorithms have similar functionality to `std::search()` algorithm. They provide a different interface which is more suitable for common string operations. Instead of returning just the start of matching subsequence they return a range which is necessary when the length of the matching subsequence is not known beforehand. This feature also allows a partitioning of the input sequence into three parts: a prefix, a substring and a suffix.

Another difference is an addition of various searching methods besides find_first, including find_regex.

It the library, find algorithms are implemented in terms of Finders. Finders are used also by other facilities (replace,split). For convenience, there are also function wrappers for these finders to simplify find operations.

Currently the library contains only naive implementation of find algorithms with complexity O(n * m) where n is the size of the input sequence and m is the size of the search sequence. There are algorithms with complexity O(n), but for smaller sequence a constant overhead is rather big. For small m << n (m by magnitude smaller than n) the current implementation provides acceptable efficiency. Even the C++ standard defines the required complexity for search algorithm as O(n * m). It is possible that a future version of library will also contain algorithms with linear complexity as an option

Replace Algorithms

The implementation of replace algorithms follows the layered structure of the library. The lower layer implements generic substitution of a range in the input sequence. This layer takes a Finder object and a Formatter object as an input. These two functors define what to replace and what to replace it with. The upper layer functions are just wrapping calls to the lower layer. Finders are shared with the find and split facility.

As usual, the implementation of the lower layer is designed to work with a generic sequence while taking advantage of specific features if possible (by using Sequence traits)

Find Iterators & Split Algorithms

Find iterators are a logical extension of the find facility. Instead of searching for one match, the whole input can be iteratively searched for multiple matches. The result of the search is then used to partition the input. It depends on the algorithms which parts are returned as the result. They can be the matching parts (`find_iterator`) of the parts in between (`split_iterator`).

In addition the split algorithms like `find_all()` and `split()` can simplify the common operations. They use a find iterator to search the whole input and copy the matches they found into the supplied container.

Exception Safety

The library requires that all operations on types used as template or function arguments provide the *basic exception-safety guarantee*. In turn, all functions and algorithms in this library, except where stated otherwise, will provide the *basic exception-safety guarantee*. In other words: The library maintains its invariants and does not leak resources in the face of exceptions. Some library operations give stronger guarantees, which are documented on an individual basis.

Some functions can provide the *strong exception-safety guarantee*. That means that following statements are true:

- If an exception is thrown, there are no effects other than those of the function
- If an exception is thrown other than by the function, there are no effects

This guarantee can be provided under the condition that the operations on the types used for arguments for these functions either provide the strong exception guarantee or do not alter the global state .

In the reference, under the term *strong exception-safety guarantee*, we mean the guarantee as defined above.

For more information about the exception safety topics, follow this link

Concepts

Definitions

Table 13. Notation

`F`	A type that is a model of Finder
`Fmt`	A type that is a model of Formatter
`Iter`	Iterator Type
`f`	Object of type `F`
`fmt`	Object of type `Fmt`
`i,j`	Objects of type `Iter`

Finder Concept

Finder is a functor which searches for an arbitrary part of a container. The result of the search is given as an `iterator_range` delimiting the selected part.

Table 14. Valid Expressions

Expression	Return Type	Effects
`f(i,j)`	Convertible to `iterator_range<Iter>`	Perform the search on the interval [i,j) and returns the result of the search

Various algorithms need to perform a search in a container and a Finder is a generalization of such search operations that allows algorithms to abstract from searching. For instance, generic replace algorithms can replace any part of the input, and the Finder is used to select the desired one.

Note, that it is only required that the finder works with a particular iterator type. However, a Finder operation can be defined as a template, allowing the Finder to work with any iterator.

Examples

- Finder implemented as a class. This Finder always returns the whole input as a match. `operator()` is templated, so that the finder can be used on any iterator type.

```
struct simple_finder
{
    template<typename ForwardIteratorT>
    boost::iterator_range<ForwardIteratorT> operator()(
        ForwardIteratorT Begin,
        ForwardIteratorT End )
    {
        return boost::make_range( Begin, End );
    }
};
        ↵
```

- Function Finder. Finder can be any function object. That is, any ordinary function with the required signature can be used as well. However, such a function can be used only for a specific iterator type.

```
boost::iterator_range<std::string> simple_finder(
    std::string::const_iterator Begin,
    std::string::const_iterator End )
{
    return boost::make_range( Begin, End );
}
        ↵
```

Formatter concept

Formatters are used by replace algorithms. They are used in close combination with finders. A formatter is a functor, which takes a result from a Finder operation and transforms it in a specific way. The operation of the formatter can use additional information provided by a specific finder, for example `regex_formatter()` uses the match information from `regex_finder()` to format the result of formatter operation.

Table 15. Valid Expressions

Expression	Return Type	Effects
`fmt(f(i,j))`	A container type, accessible using container traits	Formats the result of the finder operation

Similarly to finders, formatters generalize format operations. When a finder is used to select a part of the input, formatter takes this selection and performs some formating on it. Algorithms can abstract from formating using a formatter.

Examples

- Formatter implemented as a class. This Formatter does not perform any formating and returns the match, repackaged. `operator()` is templated, so that the Formatter can be used on any Finder type.

```
struct simple_formatter
{
    template<typename FindResultT>
    std::string operator()( const FindResultT& Match )
    {
        std::string Temp( Match.begin(), Match.end() );
        return Temp;
    }
};
               ↵
```

- Function Formatter. Similarly to Finder, Formatter can be any function object. However, as a function, it can be used only with a specific Finder type.

```
std::string simple_formatter( boost::iterator_range<std::string::const_iterator>& Match )
{
    std::string Temp( Match.begin(), Match.end() );
    return Temp;
}
                    ↵
```

Reference

Header <boost/algorithm/string.hpp>

Cumulative include for string_algo library

Header <boost/algorithm/string/case_conv.hpp>

Defines sequence case-conversion algorithms. Algorithms convert each element in the input sequence to the desired case using provided locales.

```
namespace boost {
  namespace algorithm {
    template<typename OutputIteratorT, typename RangeT>
      OutputIteratorT
      to_lower_copy(OutputIteratorT, const RangeT &,
                    const std::locale & = std::locale());
    template<typename SequenceT>
      SequenceT to_lower_copy(const SequenceT &,
                              const std::locale & = std::locale());
    template<typename WritableRangeT>
      void to_lower(WritableRangeT &, const std::locale & = std::locale());
    template<typename OutputIteratorT, typename RangeT>
      OutputIteratorT
      to_upper_copy(OutputIteratorT, const RangeT &,
                    const std::locale & = std::locale());
    template<typename SequenceT>
      SequenceT to_upper_copy(const SequenceT &,
                              const std::locale & = std::locale());
    template<typename WritableRangeT>
      void to_upper(WritableRangeT &, const std::locale & = std::locale());
  }
}
```

Function to_lower_copy

boost::algorithm::to_lower_copy — Convert to lower case.

Synopsis

```
// In header: <boost/algorithm/string/case_conv.hpp>


template<typename OutputIteratorT, typename RangeT>
  OutputIteratorT
  to_lower_copy(OutputIteratorT Output, const RangeT & Input,
                const std::locale & Loc = std::locale());
template<typename SequenceT>
  SequenceT to_lower_copy(const SequenceT & Input,
                          const std::locale & Loc = std::locale());
```

Description

Each element of the input sequence is converted to lower case. The result is a copy of the input converted to lower case. It is returned as a sequence or copied to the output iterator.

Parameters:	`Input`	An input range
	`Loc`	A locale used for conversion
	`Output`	An output iterator to which the result will be copied
Returns:	An output iterator pointing just after the last inserted character or a copy of the input	
Notes:	The second variant of this function provides the strong exception-safety guarantee	

Function template to_lower

boost::algorithm::to_lower — Convert to lower case.

Synopsis

```
// In header: <boost/algorithm/string/case_conv.hpp>


template<typename WritableRangeT>
  void to_lower(WritableRangeT & Input,
                const std::locale & Loc = std::locale());
```

Description

Each element of the input sequence is converted to lower case. The input sequence is modified in-place.

Parameters:	`Input`	A range
	`Loc`	a locale used for conversion

Function to_upper_copy

boost::algorithm::to_upper_copy — Convert to upper case.

Synopsis

```
// In header: <boost/algorithm/string/case_conv.hpp>


template<typename OutputIteratorT, typename RangeT>
  OutputIteratorT
  to_upper_copy(OutputIteratorT Output, const RangeT & Input,
                const std::locale & Loc = std::locale());
template<typename SequenceT>
  SequenceT to_upper_copy(const SequenceT & Input,
                          const std::locale & Loc = std::locale());
```

Description

Each element of the input sequence is converted to upper case. The result is a copy of the input converted to upper case. It is returned as a sequence or copied to the output iterator

Parameters:	`Input`	An input range
	`Loc`	A locale used for conversion
	`Output`	An output iterator to which the result will be copied
Returns:	An output iterator pointing just after the last inserted character or a copy of the input	
Notes:	The second variant of this function provides the strong exception-safety guarantee	

Function template to_upper

boost::algorithm::to_upper — Convert to upper case.

Synopsis

```
// In header: <boost/algorithm/string/case_conv.hpp>


template<typename WritableRangeT>
  void to_upper(WritableRangeT & Input,
                const std::locale & Loc = std::locale());
```

Description

Each element of the input sequence is converted to upper case. The input sequence is modified in-place.

Parameters:	`Input`	An input range
	`Loc`	a locale used for conversion

Header <boost/algorithm/string/classification.hpp>

Classification predicates are included in the library to give some more convenience when using algorithms like `trim()` and `all()`. They wrap functionality of STL classification functions (e.g. `std::isspace()`) into generic functors.

```
namespace boost {
  namespace algorithm {
    unspecified is_classified(std::ctype_base::mask,
                              const std::locale & = std::locale());
    unspecified is_space(const std::locale & = std::locale());
    unspecified is_alnum(const std::locale & = std::locale());
    unspecified is_alpha(const std::locale & = std::locale());
    unspecified is_cntrl(const std::locale & = std::locale());
    unspecified is_digit(const std::locale & = std::locale());
    unspecified is_graph(const std::locale & = std::locale());
    unspecified is_lower(const std::locale & = std::locale());
    unspecified is_print(const std::locale & = std::locale());
    unspecified is_punct(const std::locale & = std::locale());
    unspecified is_upper(const std::locale & = std::locale());
    unspecified is_xdigit(const std::locale & = std::locale());
    template<typename RangeT> unspecified is_any_of(const RangeT &);
    template<typename CharT> unspecified is_from_range(CharT, CharT);
    template<typename Pred1T, typename Pred2T>
      unspecified operator&&(const predicate_facade< Pred1T > &,
                             const predicate_facade< Pred2T > &);
    template<typename Pred1T, typename Pred2T>
      unspecified operator||(const predicate_facade< Pred1T > &,
                             const predicate_facade< Pred2T > &);
    template<typename PredT>
      unspecified operator!(const predicate_facade< PredT > &);
  }
}
```

Function is_classified

boost::algorithm::is_classified — is_classified predicate

Synopsis

```
// In header: <boost/algorithm/string/classification.hpp>


unspecified is_classified(std::ctype_base::mask Type,
                          const std::locale & Loc = std::locale());
```

Description

Construct the `is_classified` predicate. This predicate holds if the input is of specified `std::ctype` category.

Parameters:	`Loc`	A locale used for classification
	`Type`	A `std::ctype` category
Returns:	An instance of the `is_classified` predicate	

Function is_space

boost::algorithm::is_space — is_space predicate

Synopsis

```
// In header: <boost/algorithm/string/classification.hpp>


unspecified is_space(const std::locale & Loc = std::locale());
```

Description

Construct the `is_classified` predicate for the `ctype_base::space` category.

Parameters: `Loc` A locale used for classification
Returns: An instance of the `is_classified` predicate

Function is_alnum

boost::algorithm::is_alnum — is_alnum predicate

Synopsis

```
// In header: <boost/algorithm/string/classification.hpp>


unspecified is_alnum(const std::locale & Loc = std::locale());
```

Description

Construct the `is_classified` predicate for the `ctype_base::alnum` category.

Parameters: `Loc` A locale used for classification
Returns: An instance of the `is_classified` predicate

Function is_alpha

boost::algorithm::is_alpha — is_alpha predicate

Synopsis

```
// In header: <boost/algorithm/string/classification.hpp>


unspecified is_alpha(const std::locale & Loc = std::locale());
```

Description

Construct the `is_classified` predicate for the `ctype_base::alpha` category.

Parameters: `Loc` A locale used for classification
Returns: An instance of the `is_classified` predicate

Function is_cntrl

boost::algorithm::is_cntrl — is_cntrl predicate

Synopsis

```
// In header: <boost/algorithm/string/classification.hpp>


unspecified is_cntrl(const std::locale & Loc = std::locale());
```

Description

Construct the `is_classified` predicate for the `ctype_base::cntrl` category.

Parameters: `Loc` A locale used for classification
Returns: An instance of the `is_classified` predicate

Function is_digit

boost::algorithm::is_digit — is_digit predicate

Synopsis

```
// In header: <boost/algorithm/string/classification.hpp>


unspecified is_digit(const std::locale & Loc = std::locale());
```

Description

Construct the `is_classified` predicate for the `ctype_base::digit` category.

Parameters: `Loc` A locale used for classification
Returns: An instance of the `is_classified` predicate

Function is_graph

boost::algorithm::is_graph — is_graph predicate

Synopsis

```
// In header: <boost/algorithm/string/classification.hpp>


unspecified is_graph(const std::locale & Loc = std::locale());
```

Description

Construct the `is_classified` predicate for the `ctype_base::graph` category.

Parameters: `Loc` A locale used for classification
Returns: An instance of the `is_classified` predicate

Function is_lower

boost::algorithm::is_lower — is_lower predicate

Synopsis

```
// In header: <boost/algorithm/string/classification.hpp>


unspecified is_lower(const std::locale & Loc = std::locale());
```

Description

Construct the `is_classified` predicate for the `ctype_base::lower` category.

Parameters: `Loc` A locale used for classification
Returns: An instance of `is_classified` predicate

Function is_print

boost::algorithm::is_print — is_print predicate

Synopsis

```
// In header: <boost/algorithm/string/classification.hpp>


unspecified is_print(const std::locale & Loc = std::locale());
```

Description

Construct the `is_classified` predicate for the `ctype_base::print` category.

Parameters: `Loc` A locale used for classification
Returns: An instance of the `is_classified` predicate

Function is_punct

boost::algorithm::is_punct — is_punct predicate

Synopsis

```
// In header: <boost/algorithm/string/classification.hpp>


unspecified is_punct(const std::locale & Loc = std::locale());
```

Description

Construct the `is_classified` predicate for the `ctype_base::punct` category.

Parameters: `Loc` A locale used for classification
Returns: An instance of the `is_classified` predicate

Function is_upper

boost::algorithm::is_upper — is_upper predicate

Synopsis

```
// In header: <boost/algorithm/string/classification.hpp>


unspecified is_upper(const std::locale & Loc = std::locale());
```

Description

Construct the `is_classified` predicate for the `ctype_base::upper` category.

Parameters: `Loc` A locale used for classification
Returns: An instance of the `is_classified` predicate

Function is_xdigit

boost::algorithm::is_xdigit — is_xdigit predicate

Synopsis

```
// In header: <boost/algorithm/string/classification.hpp>


unspecified is_xdigit(const std::locale & Loc = std::locale());
```

Description

Construct the `is_classified` predicate for the `ctype_base::xdigit` category.

Parameters: `Loc` A locale used for classification
Returns: An instance of the `is_classified` predicate

Function template is_any_of

boost::algorithm::is_any_of — is_any_of predicate

Synopsis

```
// In header: <boost/algorithm/string/classification.hpp>


template<typename RangeT> unspecified is_any_of(const RangeT & Set);
```

Description

Construct the `is_any_of` predicate. The predicate holds if the input is included in the specified set of characters.

Parameters: `Set` A set of characters to be recognized
Returns: An instance of the `is_any_of` predicate

Function template is_from_range

boost::algorithm::is_from_range — is_from_range predicate

Synopsis

```
// In header: <boost/algorithm/string/classification.hpp>


template<typename CharT> unspecified is_from_range(CharT From, CharT To);
```

Description

Construct the `is_from_range` predicate. The predicate holds if the input is included in the specified range. (i.e. From <= Ch <= To)

Parameters:	`From`	The start of the range
	`To`	The end of the range
Returns:	An instance of the `is_from_range` predicate	

Function template operator&&

boost::algorithm::operator&& — predicate 'and' composition predicate

Synopsis

```
// In header: <boost/algorithm/string/classification.hpp>


template<typename Pred1T, typename Pred2T>
  unspecified operator&&(const predicate_facade< Pred1T > & Pred1,
                         const predicate_facade< Pred2T > & Pred2);
```

Description

Construct the `class_and` predicate. This predicate can be used to logically combine two classification predicates. `class_and` holds, if both predicates return true.

Parameters:	`Pred1`	The first predicate
	`Pred2`	The second predicate
Returns:	An instance of the `class_and` predicate	

Function template operator||

boost::algorithm::operator|| — predicate 'or' composition predicate

Synopsis

```
// In header: <boost/algorithm/string/classification.hpp>


template<typename Pred1T, typename Pred2T>
  unspecified operator||(const predicate_facade< Pred1T > & Pred1,
                         const predicate_facade< Pred2T > & Pred2);
```

Description

Construct the `class_or` predicate. This predicate can be used to logically combine two classification predicates. `class_or` holds, if one of the predicates return true.

Parameters:	`Pred1`	The first predicate
	`Pred2`	The second predicate
Returns:	An instance of the `class_or` predicate	

Function template operator!

boost::algorithm::operator! — predicate negation operator

Synopsis

```
// In header: <boost/algorithm/string/classification.hpp>


template<typename PredT>
  unspecified operator!(const predicate_facade< PredT > & Pred);
```

Description

Construct the `class_not` predicate. This predicate represents a negation. `class_or` holds if of the predicates return false.

Parameters:	`Pred`	The predicate to be negated
Returns:	An instance of the `class_not` predicate	

Header <boost/algorithm/string/compare.hpp>

Defines element comparison predicates. Many algorithms in this library can take an additional argument with a predicate used to compare elements. This makes it possible, for instance, to have case insensitive versions of the algorithms.

```
namespace boost {
  namespace algorithm {
    struct is_equal;
    struct is_iequal;
    struct is_less;
    struct is_iless;
    struct is_not_greater;
    struct is_not_igreater;
  }
}
```

Struct is_equal

boost::algorithm::is_equal — is_equal functor

Synopsis

```
// In header: <boost/algorithm/string/compare.hpp>


struct is_equal {

  // public member functions
  template<typename T1, typename T2>
    bool operator()(const T1 &, const T2 &) const;
};
```

Description

Standard STL equal_to only handle comparison between arguments of the same type. This is a less restrictive version which wraps operator ==.

`is_equal` public member functions

1. ```
   template<typename T1, typename T2>
     bool operator()(const T1 & Arg1, const T2 & Arg2) const;
   ```

   Function operator.

   Compare two operands for equality
   ```

Struct is_iequal

boost::algorithm::is_iequal — case insensitive version of is_equal

Synopsis

```
// In header: <boost/algorithm/string/compare.hpp>


struct is_iequal {
  // construct/copy/destruct
  is_iequal(const std::locale & = std::locale());

  // public member functions
  template<typename T1, typename T2>
    bool operator()(const T1 &, const T2 &) const;
};
```

Description

Case insensitive comparison predicate. Comparison is done using specified locales.

`is_iequal` public construct/copy/destruct

1. ```
   is_iequal(const std::locale & Loc = std::locale());
   ```

   Constructor.

   Parameters: `Loc` locales used for comparison

`is_iequal` public member functions

1. ```
   template<typename T1, typename T2>
     bool operator()(const T1 & Arg1, const T2 & Arg2) const;
   ```

   Function operator.

   Compare two operands. Case is ignored.

Struct is_less

boost::algorithm::is_less — is_less functor

Synopsis

```
// In header: <boost/algorithm/string/compare.hpp>


struct is_less {

  // public member functions
  template<typename T1, typename T2>
    bool operator()(const T1 &, const T2 &) const;
};
```

Description

Convenient version of standard std::less. Operation is templated, therefore it is not required to specify the exact types upon the construction

`is_less` public member functions

1. ```
   template<typename T1, typename T2>
     bool operator()(const T1 & Arg1, const T2 & Arg2) const;
   ```

   Functor operation.

   Compare two operands using > operator
   ```

Struct is_iless

boost::algorithm::is_iless — case insensitive version of is_less

Synopsis

```
// In header: <boost/algorithm/string/compare.hpp>


struct is_iless {
  // construct/copy/destruct
  is_iless(const std::locale & = std::locale());

  // public member functions
  template<typename T1, typename T2>
    bool operator()(const T1 &, const T2 &) const;
};
```

Description

Case insensitive comparison predicate. Comparison is done using specified locales.

`is_iless` public construct/copy/destruct

1. ```
   is_iless(const std::locale & Loc = std::locale());
   ```

   Constructor.

   Parameters: `Loc` locales used for comparison

`is_iless` public member functions

1. ```
   template<typename T1, typename T2>
     bool operator()(const T1 & Arg1, const T2 & Arg2) const;
   ```

   Function operator.

   Compare two operands. Case is ignored.

Struct is_not_greater

boost::algorithm::is_not_greater — is_not_greater functor

Synopsis

```
// In header: <boost/algorithm/string/compare.hpp>


struct is_not_greater {

  // public member functions
  template<typename T1, typename T2>
    bool operator()(const T1 &, const T2 &) const;
};
```

Description

Convenient version of standard std::not_greater_to. Operation is templated, therefore it is not required to specify the exact types upon the construction

`is_not_greater` public member functions

1. ```
   template<typename T1, typename T2>
     bool operator()(const T1 & Arg1, const T2 & Arg2) const;
   ```

   Functor operation.

   Compare two operands using > operator
   ```

Struct is_not_igreater

boost::algorithm::is_not_igreater — case insensitive version of is_not_greater

Synopsis

```
// In header: <boost/algorithm/string/compare.hpp>


struct is_not_igreater {
  // construct/copy/destruct
  is_not_igreater(const std::locale & = std::locale());

  // public member functions
  template<typename T1, typename T2>
    bool operator()(const T1 &, const T2 &) const;
};
```

Description

Case insensitive comparison predicate. Comparison is done using specified locales.

`is_not_igreater` public construct/copy/destruct

1. ```
   is_not_igreater(const std::locale & Loc = std::locale());
   ```

   Constructor.

   Parameters: `Loc` locales used for comparison

`is_not_igreater` public member functions

1. ```
   template<typename T1, typename T2>
     bool operator()(const T1 & Arg1, const T2 & Arg2) const;
   ```

   Function operator.

   Compare two operands. Case is ignored.

Header <boost/algorithm/string/concept.hpp>

Defines concepts used in string_algo library

```
namespace boost {
  namespace algorithm {
    template<typename FinderT, typename IteratorT> struct FinderConcept;
    template<typename FormatterT, typename FinderT, typename IteratorT>
      struct FormatterConcept;
  }
}
```

Struct template FinderConcept

boost::algorithm::FinderConcept — Finder concept.

Synopsis

```
// In header: <boost/algorithm/string/concept.hpp>

template<typename FinderT, typename IteratorT>
struct FinderConcept {

  // public member functions
  void constraints();
};
```

Description

Defines the Finder concept. Finder is a functor which selects an arbitrary part of a string. Search is performed on the range specified by starting and ending iterators.

Result of the find operation must be convertible to iterator_range.

`FinderConcept` public member functions

1. ```
   void constraints();
   ```
   ```

Struct template FormatterConcept

boost::algorithm::FormatterConcept — Formatter concept.

Synopsis

```
// In header: <boost/algorithm/string/concept.hpp>

template<typename FormatterT, typename FinderT, typename IteratorT>
struct FormatterConcept {

  // public member functions
  void constraints();
};
```

Description

Defines the Formatter concept. Formatter is a functor, which takes a result from a finder operation and transforms it in a specific way.

Result must be a container supported by container_traits, or a reference to it.

`FormatterConcept` public member functions

1. ```
   void constraints();
   ```

Header <boost/algorithm/string/constants.hpp>

```
namespace boost {
  namespace algorithm {
    enum token_compress_mode_type;
  }
}
```
```

Type token_compress_mode_type

boost::algorithm::token_compress_mode_type — Token compression mode.

Synopsis

```
// In header: <boost/algorithm/string/constants.hpp>


enum token_compress_mode_type { token_compress_on, token_compress_off };
```

Description

Specifies token compression mode for the token_finder.

`token_compress_on`	Compress adjacent tokens.
`token_compress_off`	Do not compress adjacent tokens.

Header <boost/algorithm/string/erase.hpp>

Defines various erase algorithms. Each algorithm removes part(s) of the input according to a searching criteria.

```
namespace boost {
  namespace algorithm {
    template<typename OutputIteratorT, typename RangeT>
      OutputIteratorT
      erase_range_copy(OutputIteratorT, const RangeT &,
                       const iterator_range< typename range_const_iterator< RangeT >::type > &);
    template<typename SequenceT>
      SequenceT erase_range_copy(const SequenceT &,
                                 const iterator_range< typename range_const_iterator< Sequen↵
ceT >::type > &);
    template<typename SequenceT>
      void erase_range(SequenceT &,
                       const iterator_range< typename range_iterator< SequenceT >::type > &);
    template<typename OutputIteratorT, typename Range1T, typename Range2T>
      OutputIteratorT
      erase_first_copy(OutputIteratorT, const Range1T &, const Range2T &);
    template<typename SequenceT, typename RangeT>
      SequenceT erase_first_copy(const SequenceT &, const RangeT &);
    template<typename SequenceT, typename RangeT>
      void erase_first(SequenceT &, const RangeT &);
    template<typename OutputIteratorT, typename Range1T, typename Range2T>
      OutputIteratorT
      ierase_first_copy(OutputIteratorT, const Range1T &, const Range2T &,
                        const std::locale & = std::locale());
    template<typename SequenceT, typename RangeT>
      SequenceT ierase_first_copy(const SequenceT &, const RangeT &,
                                  const std::locale & = std::locale());
    template<typename SequenceT, typename RangeT>
      void ierase_first(SequenceT &, const RangeT &,
                        const std::locale & = std::locale());
    template<typename OutputIteratorT, typename Range1T, typename Range2T>
      OutputIteratorT
      erase_last_copy(OutputIteratorT, const Range1T &, const Range2T &);
    template<typename SequenceT, typename RangeT>
      SequenceT erase_last_copy(const SequenceT &, const RangeT &);
    template<typename SequenceT, typename RangeT>
      void erase_last(SequenceT &, const RangeT &);
    template<typename OutputIteratorT, typename Range1T, typename Range2T>
      OutputIteratorT
      ierase_last_copy(OutputIteratorT, const Range1T &, const Range2T &,
                       const std::locale & = std::locale());
    template<typename SequenceT, typename RangeT>
      SequenceT ierase_last_copy(const SequenceT &, const RangeT &,
                                 const std::locale & = std::locale());
    template<typename SequenceT, typename RangeT>
      void ierase_last(SequenceT &, const RangeT &,
                       const std::locale & = std::locale());
    template<typename OutputIteratorT, typename Range1T, typename Range2T>
      OutputIteratorT
      erase_nth_copy(OutputIteratorT, const Range1T &, const Range2T &, int);
    template<typename SequenceT, typename RangeT>
      SequenceT erase_nth_copy(const SequenceT &, const RangeT &, int);
    template<typename SequenceT, typename RangeT>
      void erase_nth(SequenceT &, const RangeT &, int);
    template<typename OutputIteratorT, typename Range1T, typename Range2T>
      OutputIteratorT
      ierase_nth_copy(OutputIteratorT, const Range1T &, const Range2T &, int,
                      const std::locale & = std::locale());
    template<typename SequenceT, typename RangeT>
      SequenceT ierase_nth_copy(const SequenceT &, const RangeT &, int,
                                const std::locale & = std::locale());
    template<typename SequenceT, typename RangeT>
      void ierase_nth(SequenceT &, const RangeT &, int,
```

```
                    const std::locale & = std::locale());
    template<typename OutputIteratorT, typename Range1T, typename Range2T>
      OutputIteratorT
      erase_all_copy(OutputIteratorT, const Range1T &, const Range2T &);
    template<typename SequenceT, typename RangeT>
      SequenceT erase_all_copy(const SequenceT &, const RangeT &);
    template<typename SequenceT, typename RangeT>
      void erase_all(SequenceT &, const RangeT &);
    template<typename OutputIteratorT, typename Range1T, typename Range2T>
      OutputIteratorT
      ierase_all_copy(OutputIteratorT, const Range1T &, const Range2T &,
                    const std::locale & = std::locale());
    template<typename SequenceT, typename RangeT>
      SequenceT ierase_all_copy(const SequenceT &, const RangeT &,
                                const std::locale & = std::locale());
    template<typename SequenceT, typename RangeT>
      void ierase_all(SequenceT &, const RangeT &,
                    const std::locale & = std::locale());
    template<typename OutputIteratorT, typename RangeT>
      OutputIteratorT erase_head_copy(OutputIteratorT, const RangeT &, int);
    template<typename SequenceT>
      SequenceT erase_head_copy(const SequenceT &, int);
    template<typename SequenceT> void erase_head(SequenceT &, int);
    template<typename OutputIteratorT, typename RangeT>
      OutputIteratorT erase_tail_copy(OutputIteratorT, const RangeT &, int);
    template<typename SequenceT>
      SequenceT erase_tail_copy(const SequenceT &, int);
    template<typename SequenceT> void erase_tail(SequenceT &, int);
  }
}
```

Function erase_range_copy

boost::algorithm::erase_range_copy — Erase range algorithm.

Synopsis

```
// In header: <boost/algorithm/string/erase.hpp>


template<typename OutputIteratorT, typename RangeT>
  OutputIteratorT
  erase_range_copy(OutputIteratorT Output, const RangeT & Input,
                   const iterator_range< typename range_const_iterator< RangeT >::type > & Sear↵
chRange);
template<typename SequenceT>
  SequenceT erase_range_copy(const SequenceT & Input,
                             const iterator_range< typename range_const_iterator< Sequen↵
ceT >::type > & SearchRange);
```

Description

Remove the given range from the input. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters:	`Input`	An input sequence
	`Output`	An output iterator to which the result will be copied
	`SearchRange`	A range in the input to be removed
Returns:	An output iterator pointing just after the last inserted character or a modified copy of the input	
Notes:	The second variant of this function provides the strong exception-safety guarantee	

Function template erase_range

boost::algorithm::erase_range — Erase range algorithm.

Synopsis

```
// In header: <boost/algorithm/string/erase.hpp>


template<typename SequenceT>
  void erase_range(SequenceT & Input,
                   const iterator_range< typename range_iterator< SequenceT >::type > & Sear↵
chRange);
```

Description

Remove the given range from the input. The input sequence is modified in-place.

Parameters:	`Input`	An input sequence
	`SearchRange`	A range in the input to be removed

Function erase_first_copy

boost::algorithm::erase_first_copy — Erase first algorithm.

Synopsis

```
// In header: <boost/algorithm/string/erase.hpp>


template<typename OutputIteratorT, typename Range1T, typename Range2T>
  OutputIteratorT
  erase_first_copy(OutputIteratorT Output, const Range1T & Input,
                   const Range2T & Search);
template<typename SequenceT, typename RangeT>
  SequenceT erase_first_copy(const SequenceT & Input, const RangeT & Search);
```

Description

Remove the first occurrence of the substring from the input. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters:	`Input`	An input string
	`Output`	An output iterator to which the result will be copied
	`Search`	A substring to be searched for
Returns:	An output iterator pointing just after the last inserted character or a modified copy of the input	
Notes:	The second variant of this function provides the strong exception-safety guarantee	

Function template erase_first

boost::algorithm::erase_first — Erase first algorithm.

Synopsis

```
// In header: <boost/algorithm/string/erase.hpp>


template<typename SequenceT, typename RangeT>
  void erase_first(SequenceT & Input, const RangeT & Search);
```

Description

Remove the first occurrence of the substring from the input. The input sequence is modified in-place.

Parameters:		
	`Input`	An input string
	`Search`	A substring to be searched for.

Function ierase_first_copy

boost::algorithm::ierase_first_copy — Erase first algorithm (case insensitive).

Synopsis

```
// In header: <boost/algorithm/string/erase.hpp>


template<typename OutputIteratorT, typename Range1T, typename Range2T>
  OutputIteratorT
  ierase_first_copy(OutputIteratorT Output, const Range1T & Input,
                    const Range2T & Search,
                    const std::locale & Loc = std::locale());
template<typename SequenceT, typename RangeT>
  SequenceT ierase_first_copy(const SequenceT & Input, const RangeT & Search,
                              const std::locale & Loc = std::locale());
```

Description

Remove the first occurrence of the substring from the input. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator. Searching is case insensitive.

Parameters:	`Input`	An input string
	`Loc`	A locale used for case insensitive comparison
	`Output`	An output iterator to which the result will be copied
	`Search`	A substring to be searched for
Returns:	An output iterator pointing just after the last inserted character or a modified copy of the input	
Notes:	The second variant of this function provides the strong exception-safety guarantee	

Function template ierase_first

boost::algorithm::ierase_first — Erase first algorithm (case insensitive).

Synopsis

```
// In header: <boost/algorithm/string/erase.hpp>


template<typename SequenceT, typename RangeT>
  void ierase_first(SequenceT & Input, const RangeT & Search,
                    const std::locale & Loc = std::locale());
```

Description

Remove the first occurrence of the substring from the input. The input sequence is modified in-place. Searching is case insensitive.

Parameters:	`Input`	An input string
	`Loc`	A locale used for case insensitive comparison
	`Search`	A substring to be searched for

Function erase_last_copy

boost::algorithm::erase_last_copy — Erase last algorithm.

Synopsis

```
// In header: <boost/algorithm/string/erase.hpp>


template<typename OutputIteratorT, typename Range1T, typename Range2T>
  OutputIteratorT
  erase_last_copy(OutputIteratorT Output, const Range1T & Input,
                  const Range2T & Search);
template<typename SequenceT, typename RangeT>
  SequenceT erase_last_copy(const SequenceT & Input, const RangeT & Search);
```

Description

Remove the last occurrence of the substring from the input. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters:	`Input`	An input string
	`Output`	An output iterator to which the result will be copied
	`Search`	A substring to be searched for.
Returns:	An output iterator pointing just after the last inserted character or a modified copy of the input	
Notes:	The second variant of this function provides the strong exception-safety guarantee	

Function template erase_last

boost::algorithm::erase_last — Erase last algorithm.

Synopsis

```
// In header: <boost/algorithm/string/erase.hpp>


template<typename SequenceT, typename RangeT>
  void erase_last(SequenceT & Input, const RangeT & Search);
```

Description

Remove the last occurrence of the substring from the input. The input sequence is modified in-place.

Parameters:		
	`Input`	An input string
	`Search`	A substring to be searched for

Function ierase_last_copy

boost::algorithm::ierase_last_copy — Erase last algorithm (case insensitive).

Synopsis

```
// In header: <boost/algorithm/string/erase.hpp>


template<typename OutputIteratorT, typename Range1T, typename Range2T>
  OutputIteratorT
  ierase_last_copy(OutputIteratorT Output, const Range1T & Input,
                   const Range2T & Search,
                   const std::locale & Loc = std::locale());
template<typename SequenceT, typename RangeT>
  SequenceT ierase_last_copy(const SequenceT & Input, const RangeT & Search,
                             const std::locale & Loc = std::locale());
```

Description

Remove the last occurrence of the substring from the input. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator. Searching is case insensitive.

Parameters:	`Input`	An input string
	`Loc`	A locale used for case insensitive comparison
	`Output`	An output iterator to which the result will be copied
	`Search`	A substring to be searched for
Returns:	An output iterator pointing just after the last inserted character or a modified copy of the input	
Notes:	The second variant of this function provides the strong exception-safety guarantee	

Function template ierase_last

boost::algorithm::ierase_last — Erase last algorithm (case insensitive).

Synopsis

```
// In header: <boost/algorithm/string/erase.hpp>


template<typename SequenceT, typename RangeT>
  void ierase_last(SequenceT & Input, const RangeT & Search,
                   const std::locale & Loc = std::locale());
```

Description

Remove the last occurrence of the substring from the input. The input sequence is modified in-place. Searching is case insensitive.

Parameters:	`Input`	An input string
	`Loc`	A locale used for case insensitive comparison
	`Search`	A substring to be searched for

Function erase_nth_copy

boost::algorithm::erase_nth_copy — Erase nth algorithm.

Synopsis

```
// In header: <boost/algorithm/string/erase.hpp>


template<typename OutputIteratorT, typename Range1T, typename Range2T>
  OutputIteratorT
  erase_nth_copy(OutputIteratorT Output, const Range1T & Input,
                 const Range2T & Search, int Nth);
template<typename SequenceT, typename RangeT>
  SequenceT erase_nth_copy(const SequenceT & Input, const RangeT & Search,
                           int Nth);
```

Description

Remove the Nth occurrence of the substring in the input. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters:	`Input`	An input string
	`Nth`	An index of the match to be replaced. The index is 0-based. For negative N, matches are counted from the end of string.
	`Output`	An output iterator to which the result will be copied
	`Search`	A substring to be searched for
Returns:	An output iterator pointing just after the last inserted character or a modified copy of the input	
Notes:	The second variant of this function provides the strong exception-safety guarantee	

Function template erase_nth

boost::algorithm::erase_nth — Erase nth algorithm.

Synopsis

```
// In header: <boost/algorithm/string/erase.hpp>


template<typename SequenceT, typename RangeT>
  void erase_nth(SequenceT & Input, const RangeT & Search, int Nth);
```

Description

Remove the Nth occurrence of the substring in the input. The input sequence is modified in-place.

Parameters:		
	`Input`	An input string
	`Nth`	An index of the match to be replaced. The index is 0-based. For negative N, matches are counted from the end of string.
	`Search`	A substring to be searched for.

Function ierase_nth_copy

boost::algorithm::ierase_nth_copy — Erase nth algorithm (case insensitive).

Synopsis

```
// In header: <boost/algorithm/string/erase.hpp>


template<typename OutputIteratorT, typename Range1T, typename Range2T>
  OutputIteratorT
  ierase_nth_copy(OutputIteratorT Output, const Range1T & Input,
                  const Range2T & Search, int Nth,
                  const std::locale & Loc = std::locale());
template<typename SequenceT, typename RangeT>
  SequenceT ierase_nth_copy(const SequenceT & Input, const RangeT & Search,
                            int Nth, const std::locale & Loc = std::locale());
```

Description

Remove the Nth occurrence of the substring in the input. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator. Searching is case insensitive.

Parameters:	`Input`	An input string
	`Loc`	A locale used for case insensitive comparison
	`Nth`	An index of the match to be replaced. The index is 0-based. For negative N, matches are counted from the end of string.
	`Output`	An output iterator to which the result will be copied
	`Search`	A substring to be searched for.
Returns:	An output iterator pointing just after the last inserted character or a modified copy of the input	
Notes:	The second variant of this function provides the strong exception-safety guarantee	

Function template ierase_nth

boost::algorithm::ierase_nth — Erase nth algorithm.

Synopsis

```
// In header: <boost/algorithm/string/erase.hpp>


template<typename SequenceT, typename RangeT>
  void ierase_nth(SequenceT & Input, const RangeT & Search, int Nth,
                  const std::locale & Loc = std::locale());
```

Description

Remove the Nth occurrence of the substring in the input. The input sequence is modified in-place. Searching is case insensitive.

Parameters:	`Input`	An input string
	`Loc`	A locale used for case insensitive comparison
	`Nth`	An index of the match to be replaced. The index is 0-based. For negative N, matches are counted from the end of string.
	`Search`	A substring to be searched for.

Function erase_all_copy

boost::algorithm::erase_all_copy — Erase all algorithm.

Synopsis

```
// In header: <boost/algorithm/string/erase.hpp>


template<typename OutputIteratorT, typename Range1T, typename Range2T>
  OutputIteratorT
  erase_all_copy(OutputIteratorT Output, const Range1T & Input,
                 const Range2T & Search);
template<typename SequenceT, typename RangeT>
  SequenceT erase_all_copy(const SequenceT & Input, const RangeT & Search);
```

Description

Remove all the occurrences of the string from the input. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters:	`Input`	An input sequence
	`Output`	An output iterator to which the result will be copied
	`Search`	A substring to be searched for.
Returns:	An output iterator pointing just after the last inserted character or a modified copy of the input	
Notes:	The second variant of this function provides the strong exception-safety guarantee	

Function template erase_all

boost::algorithm::erase_all — Erase all algorithm.

Synopsis

```
// In header: <boost/algorithm/string/erase.hpp>


template<typename SequenceT, typename RangeT>
  void erase_all(SequenceT & Input, const RangeT & Search);
```

Description

Remove all the occurrences of the string from the input. The input sequence is modified in-place.

Parameters:		
	`Input`	An input string
	`Search`	A substring to be searched for.

Function ierase_all_copy

boost::algorithm::ierase_all_copy — Erase all algorithm (case insensitive).

Synopsis

```
// In header: <boost/algorithm/string/erase.hpp>


template<typename OutputIteratorT, typename Range1T, typename Range2T>
  OutputIteratorT
  ierase_all_copy(OutputIteratorT Output, const Range1T & Input,
                  const Range2T & Search,
                  const std::locale & Loc = std::locale());
template<typename SequenceT, typename RangeT>
  SequenceT ierase_all_copy(const SequenceT & Input, const RangeT & Search,
                            const std::locale & Loc = std::locale());
```

Description

Remove all the occurrences of the string from the input. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator. Searching is case insensitive.

Parameters:	`Input`	An input string
	`Loc`	A locale used for case insensitive comparison
	`Output`	An output iterator to which the result will be copied
	`Search`	A substring to be searched for
Returns:	An output iterator pointing just after the last inserted character or a modified copy of the input	
Notes:	The second variant of this function provides the strong exception-safety guarantee	

Function template ierase_all

boost::algorithm::ierase_all — Erase all algorithm (case insensitive).

Synopsis

```
// In header: <boost/algorithm/string/erase.hpp>


template<typename SequenceT, typename RangeT>
  void ierase_all(SequenceT & Input, const RangeT & Search,
                  const std::locale & Loc = std::locale());
```

Description

Remove all the occurrences of the string from the input. The input sequence is modified in-place. Searching is case insensitive.

Parameters:	`Input`	An input string
	`Loc`	A locale used for case insensitive comparison
	`Search`	A substring to be searched for.

Function erase_head_copy

boost::algorithm::erase_head_copy — Erase head algorithm.

Synopsis

```
// In header: <boost/algorithm/string/erase.hpp>


template<typename OutputIteratorT, typename RangeT>
  OutputIteratorT
  erase_head_copy(OutputIteratorT Output, const RangeT & Input, int N);
template<typename SequenceT>
  SequenceT erase_head_copy(const SequenceT & Input, int N);
```

Description

Remove the head from the input. The head is a prefix of a sequence of given size. If the sequence is shorter then required, the whole string is considered to be the head. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters:	`Input`	An input string
	`N`	Length of the head. For N>=0, at most N characters are extracted. For N<0, size(Input)-|N| characters are extracted.
	`Output`	An output iterator to which the result will be copied
Returns:	An output iterator pointing just after the last inserted character or a modified copy of the input	
Notes:	The second variant of this function provides the strong exception-safety guarantee	

Function template erase_head

boost::algorithm::erase_head — Erase head algorithm.

Synopsis

```
// In header: <boost/algorithm/string/erase.hpp>


template<typename SequenceT> void erase_head(SequenceT & Input, int N);
```

Description

Remove the head from the input. The head is a prefix of a sequence of given size. If the sequence is shorter then required, the whole string is considered to be the head. The input sequence is modified in-place.

Parameters:		
	`Input`	An input string
	`N`	Length of the head For N>=0, at most N characters are extracted. For N<0, size(Input)-|N| characters are extracted.

Function erase_tail_copy

boost::algorithm::erase_tail_copy — Erase tail algorithm.

Synopsis

```
// In header: <boost/algorithm/string/erase.hpp>


template<typename OutputIteratorT, typename RangeT>
  OutputIteratorT
  erase_tail_copy(OutputIteratorT Output, const RangeT & Input, int N);
template<typename SequenceT>
  SequenceT erase_tail_copy(const SequenceT & Input, int N);
```

Description

Remove the tail from the input. The tail is a suffix of a sequence of given size. If the sequence is shorter then required, the whole string is considered to be the tail. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters:	`Input`	An input string
	`N`	Length of the tail. For N>=0, at most N characters are extracted. For N<0, size(Input)-|N| characters are extracted.
	`Output`	An output iterator to which the result will be copied
Returns:	An output iterator pointing just after the last inserted character or a modified copy of the input	
Notes:	The second variant of this function provides the strong exception-safety guarantee	

Function template erase_tail

boost::algorithm::erase_tail — Erase tail algorithm.

Synopsis

```
// In header: <boost/algorithm/string/erase.hpp>


template<typename SequenceT> void erase_tail(SequenceT & Input, int N);
```

Description

Remove the tail from the input. The tail is a suffix of a sequence of given size. If the sequence is shorter then required, the whole string is considered to be the tail. The input sequence is modified in-place.

Parameters:		
	`Input`	An input string
	`N`	Length of the tail For N>=0, at most N characters are extracted. For N<0, size(Input)-|N| characters are extracted.

Header <boost/algorithm/string/find.hpp>

Defines a set of find algorithms. The algorithms are searching for a substring of the input. The result is given as an `iterator_range` delimiting the substring.

```
namespace boost {
  namespace algorithm {
    template<typename RangeT, typename FinderT>
      iterator_range< typename range_iterator< RangeT >::type >
      find(RangeT &, const FinderT &);
    template<typename Range1T, typename Range2T>
      iterator_range< typename range_iterator< Range1T >::type >
      find_first(Range1T &, const Range2T &);
    template<typename Range1T, typename Range2T>
      iterator_range< typename range_iterator< Range1T >::type >
      ifind_first(Range1T &, const Range2T &,
                  const std::locale & = std::locale());
    template<typename Range1T, typename Range2T>
      iterator_range< typename range_iterator< Range1T >::type >
      find_last(Range1T &, const Range2T &);
    template<typename Range1T, typename Range2T>
      iterator_range< typename range_iterator< Range1T >::type >
      ifind_last(Range1T &, const Range2T &,
                 const std::locale & = std::locale());
    template<typename Range1T, typename Range2T>
      iterator_range< typename range_iterator< Range1T >::type >
      find_nth(Range1T &, const Range2T &, int);
    template<typename Range1T, typename Range2T>
      iterator_range< typename range_iterator< Range1T >::type >
      ifind_nth(Range1T &, const Range2T &, int,
                const std::locale & = std::locale());
    template<typename RangeT>
      iterator_range< typename range_iterator< RangeT >::type >
      find_head(RangeT &, int);
    template<typename RangeT>
      iterator_range< typename range_iterator< RangeT >::type >
      find_tail(RangeT &, int);
    template<typename RangeT, typename PredicateT>
      iterator_range< typename range_iterator< RangeT >::type >
      find_token(RangeT &, PredicateT,
                 token_compress_mode_type = token_compress_off);
  }
}
```

Function template find

boost::algorithm::find — Generic find algorithm.

Synopsis

```
// In header: <boost/algorithm/string/find.hpp>


template<typename RangeT, typename FinderT>
  iterator_range< typename range_iterator< RangeT >::type >
  find(RangeT & Input, const FinderT & Finder);
```

Description

Search the input using the given finder.

Parameters:	`Finder`	Finder object used for searching.
	`Input`	A string which will be searched.
Returns:	An `iterator_range` delimiting the match. Returned iterator is either `RangeT::iterator` or `RangeT::const_iterator`, depending on the constness of the input parameter.	

Function template find_first

boost::algorithm::find_first — Find first algorithm.

Synopsis

```
// In header: <boost/algorithm/string/find.hpp>


template<typename Range1T, typename Range2T>
  iterator_range< typename range_iterator< Range1T >::type >
  find_first(Range1T & Input, const Range2T & Search);
```

Description

Search for the first occurrence of the substring in the input.

Parameters:	`Input`	A string which will be searched.
	`Search`	A substring to be searched for.
Returns:	An `iterator_range` delimiting the match. Returned iterator is either `RangeT::iterator` or `RangeT::const_iterator`, depending on the constness of the input parameter.	
Notes:	This function provides the strong exception-safety guarantee	

Function template ifind_first

boost::algorithm::ifind_first — Find first algorithm (case insensitive).

Synopsis

```
// In header: <boost/algorithm/string/find.hpp>


template<typename Range1T, typename Range2T>
  iterator_range< typename range_iterator< Range1T >::type >
  ifind_first(Range1T & Input, const Range2T & Search,
              const std::locale & Loc = std::locale());
```

Description

Search for the first occurence of the substring in the input. Searching is case insensitive.

Parameters:	`Input`	A string which will be searched.
	`Loc`	A locale used for case insensitive comparison
	`Search`	A substring to be searched for.
Returns:	An `iterator_range` delimiting the match. Returned iterator is either `Range1T::iterator` or `Range1T::const_iterator`, depending on the constness of the input parameter.	
Notes:	This function provides the strong exception-safety guarantee	

Function template find_last

boost::algorithm::find_last — Find last algorithm.

Synopsis

```
// In header: <boost/algorithm/string/find.hpp>


template<typename Range1T, typename Range2T>
  iterator_range< typename range_iterator< Range1T >::type >
  find_last(Range1T & Input, const Range2T & Search);
```

Description

Search for the last occurrence of the substring in the input.

Parameters:	`Input`	A string which will be searched.
	`Search`	A substring to be searched for.
Returns:	An `iterator_range` delimiting the match. Returned iterator is either `Range1T::iterator` or `Range1T::const_iterator`, depending on the constness of the input parameter.	
Notes:	This function provides the strong exception-safety guarantee	

Function template ifind_last

boost::algorithm::ifind_last — Find last algorithm (case insensitive).

Synopsis

```
// In header: <boost/algorithm/string/find.hpp>


template<typename Range1T, typename Range2T>
  iterator_range< typename range_iterator< Range1T >::type >
  ifind_last(Range1T & Input, const Range2T & Search,
             const std::locale & Loc = std::locale());
```

Description

Search for the last match a string in the input. Searching is case insensitive.

Parameters:	`Input`	A string which will be searched.
	`Loc`	A locale used for case insensitive comparison
	`Search`	A substring to be searched for.
Returns:	An `iterator_range` delimiting the match. Returned iterator is either `Range1T::iterator` or `Range1T::const_iterator`, depending on the constness of the input parameter.	
Notes:	This function provides the strong exception-safety guarantee	

Function template find_nth

boost::algorithm::find_nth — Find n-th algorithm.

Synopsis

```
// In header: <boost/algorithm/string/find.hpp>


template<typename Range1T, typename Range2T>
  iterator_range< typename range_iterator< Range1T >::type >
  find_nth(Range1T & Input, const Range2T & Search, int Nth);
```

Description

Search for the n-th (zero-indexed) occurrence of the substring in the input.

Parameters:	`Input`	A string which will be searched.
	`Nth`	An index (zero-indexed) of the match to be found. For negative N, the matches are counted from the end of string.
	`Search`	A substring to be searched for.
Returns:	An `iterator_range` delimiting the match. Returned iterator is either `Range1T::iterator` or `Range1T::const_iterator`, depending on the constness of the input parameter.	

Function template ifind_nth

boost::algorithm::ifind_nth — Find n-th algorithm (case insensitive).

Synopsis

```
// In header: <boost/algorithm/string/find.hpp>


template<typename Range1T, typename Range2T>
  iterator_range< typename range_iterator< Range1T >::type >
  ifind_nth(Range1T & Input, const Range2T & Search, int Nth,
            const std::locale & Loc = std::locale());
```

Description

Search for the n-th (zero-indexed) occurrence of the substring in the input. Searching is case insensitive.

Parameters:	`Input`	A string which will be searched.
	`Loc`	A locale used for case insensitive comparison
	`Nth`	An index (zero-indexed) of the match to be found. For negative N, the matches are counted from the end of string.
	`Search`	A substring to be searched for.
Returns:	An `iterator_range` delimiting the match. Returned iterator is either `Range1T::iterator` or `Range1T::const_iterator`, depending on the constness of the input parameter.	
Notes:	This function provides the strong exception-safety guarantee	

Function template find_head

boost::algorithm::find_head — Find head algorithm.

Synopsis

```
// In header: <boost/algorithm/string/find.hpp>


template<typename RangeT>
  iterator_range< typename range_iterator< RangeT >::type >
  find_head(RangeT & Input, int N);
```

Description

Get the head of the input. Head is a prefix of the string of the given size. If the input is shorter then required, whole input if considered to be the head.

Parameters:	`Input`	An input string
	`N`	Length of the head For N>=0, at most N characters are extracted. For N<0, size(Input)-|N| characters are extracted.
Returns:	An `iterator_range` delimiting the match. Returned iterator is either `Range1T::iterator` or `Range1T::const_iterator`, depending on the constness of the input parameter.	
Notes:	This function provides the strong exception-safety guarantee	

Function template find_tail

boost::algorithm::find_tail — Find tail algorithm.

Synopsis

```
// In header: <boost/algorithm/string/find.hpp>


template<typename RangeT>
  iterator_range< typename range_iterator< RangeT >::type >
  find_tail(RangeT & Input, int N);
```

Description

Get the tail of the input. Tail is a suffix of the string of the given size. If the input is shorter then required, whole input if considered to be the tail.

Parameters:	`Input`	An input string
	`N`	Length of the tail. For N>=0, at most N characters are extracted. For N<0, size(Input)-|N| characters are extracted.
Returns:	An `iterator_range` delimiting the match. Returned iterator is either `RangeT::iterator` or `RangeT::const_iterator`, depending on the constness of the input parameter.	
Notes:	This function provides the strong exception-safety guarantee	

Function template find_token

boost::algorithm::find_token — Find token algorithm.

Synopsis

```
// In header: <boost/algorithm/string/find.hpp>


template<typename RangeT, typename PredicateT>
  iterator_range< typename range_iterator< RangeT >::type >
  find_token(RangeT & Input, PredicateT Pred,
             token_compress_mode_type eCompress = token_compress_off);
```

Description

Look for a given token in the string. Token is a character that matches the given predicate. If the "token compress mode" is enabled, adjacent tokens are considered to be one match.

Parameters:	`Input`	A input string.
	`Pred`	An unary predicate to identify a token
	`eCompress`	Enable/Disable compressing of adjacent tokens
Returns:	An `iterator_range` delimiting the match. Returned iterator is either `RangeT::iterator` or `RangeT::const_iterator`, depending on the constness of the input parameter.	
Notes:	This function provides the strong exception-safety guarantee	

Header <boost/algorithm/string/find_format.hpp>

Defines generic replace algorithms. Each algorithm replaces part(s) of the input. The part to be replaced is looked up using a Finder object. Result of finding is then used by a Formatter object to generate the replacement.

```
namespace boost {
  namespace algorithm {
    template<typename OutputIteratorT, typename RangeT, typename FinderT,
             typename FormatterT>
      OutputIteratorT
      find_format_copy(OutputIteratorT, const RangeT &, FinderT, FormatterT);
    template<typename SequenceT, typename FinderT, typename FormatterT>
      SequenceT find_format_copy(const SequenceT &, FinderT, FormatterT);
    template<typename SequenceT, typename FinderT, typename FormatterT>
      void find_format(SequenceT &, FinderT, FormatterT);
    template<typename OutputIteratorT, typename RangeT, typename FinderT,
             typename FormatterT>
      OutputIteratorT
      find_format_all_copy(OutputIteratorT, const RangeT &, FinderT,
                           FormatterT);
    template<typename SequenceT, typename FinderT, typename FormatterT>
      SequenceT find_format_all_copy(const SequenceT &, FinderT, FormatterT);
    template<typename SequenceT, typename FinderT, typename FormatterT>
      void find_format_all(SequenceT &, FinderT, FormatterT);
  }
}
```

Function find_format_copy

boost::algorithm::find_format_copy — Generic replace algorithm.

Synopsis

```
// In header: <boost/algorithm/string/find_format.hpp>


template<typename OutputIteratorT, typename RangeT, typename FinderT,
         typename FormatterT>
  OutputIteratorT
  find_format_copy(OutputIteratorT Output, const RangeT & Input,
                   FinderT Finder, FormatterT Formatter);
template<typename SequenceT, typename FinderT, typename FormatterT>
  SequenceT find_format_copy(const SequenceT & Input, FinderT Finder,
                             FormatterT Formatter);
```

Description

Use the Finder to search for a substring. Use the Formatter to format this substring and replace it in the input. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters:	`Finder`	A Finder object used to search for a match to be replaced
	`Formatter`	A Formatter object used to format a match
	`Input`	An input sequence
	`Output`	An output iterator to which the result will be copied
Returns:	An output iterator pointing just after the last inserted character or a modified copy of the input	
Notes:	The second variant of this function provides the strong exception-safety guarantee	

Function template find_format

boost::algorithm::find_format — Generic replace algorithm.

Synopsis

```
// In header: <boost/algorithm/string/find_format.hpp>


template<typename SequenceT, typename FinderT, typename FormatterT>
  void find_format(SequenceT & Input, FinderT Finder, FormatterT Formatter);
```

Description

Use the Finder to search for a substring. Use the Formatter to format this substring and replace it in the input. The input is modified in-place.

Parameters:	`Finder`	A Finder object used to search for a match to be replaced
	`Formatter`	A Formatter object used to format a match
	`Input`	An input sequence

Function find_format_all_copy

boost::algorithm::find_format_all_copy — Generic replace all algorithm.

Synopsis

```
// In header: <boost/algorithm/string/find_format.hpp>


template<typename OutputIteratorT, typename RangeT, typename FinderT,
         typename FormatterT>
  OutputIteratorT
  find_format_all_copy(OutputIteratorT Output, const RangeT & Input,
                       FinderT Finder, FormatterT Formatter);
template<typename SequenceT, typename FinderT, typename FormatterT>
  SequenceT find_format_all_copy(const SequenceT & Input, FinderT Finder,
                                 FormatterT Formatter);
```

Description

Use the Finder to search for a substring. Use the Formatter to format this substring and replace it in the input. Repeat this for all matching substrings. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters:	`Finder`	A Finder object used to search for a match to be replaced
	`Formatter`	A Formatter object used to format a match
	`Input`	An input sequence
	`Output`	An output iterator to which the result will be copied
Returns:	An output iterator pointing just after the last inserted character or a modified copy of the input	
Notes:	The second variant of this function provides the strong exception-safety guarantee	

Function template find_format_all

boost::algorithm::find_format_all — Generic replace all algorithm.

Synopsis

```
// In header: <boost/algorithm/string/find_format.hpp>


template<typename SequenceT, typename FinderT, typename FormatterT>
  void find_format_all(SequenceT & Input, FinderT Finder,
                       FormatterT Formatter);
```

Description

Use the Finder to search for a substring. Use the Formatter to format this substring and replace it in the input. Repeat this for all matching substrings.The input is modified in-place.

Parameters:		
	`Finder`	A Finder object used to search for a match to be replaced
	`Formatter`	A Formatter object used to format a match
	`Input`	An input sequence

Header <boost/algorithm/string/find_iterator.hpp>

Defines find iterator classes. Find iterator repeatedly applies a Finder to the specified input string to search for matches. Dereferencing the iterator yields the current match or a range between the last and the current match depending on the iterator used.

```
namespace boost {
  namespace algorithm {
    template<typename IteratorT> class find_iterator;
    template<typename IteratorT> class split_iterator;
    template<typename RangeT, typename FinderT>
      find_iterator< typename range_iterator< RangeT >::type >
      make_find_iterator(RangeT &, FinderT);
    template<typename RangeT, typename FinderT>
      split_iterator< typename range_iterator< RangeT >::type >
      make_split_iterator(RangeT &, FinderT);
  }
}
```

Class template find_iterator

boost::algorithm::find_iterator — find_iterator

Synopsis

```
// In header: <boost/algorithm/string/find_iterator.hpp>

template<typename IteratorT>
class find_iterator {
public:
  // construct/copy/destruct
  find_iterator();
  find_iterator(const find_iterator &);
  template<typename FinderT> find_iterator(IteratorT, IteratorT, FinderT);
  template<typename FinderT, typename RangeT> find_iterator(RangeT &, FinderT);

  // public member functions
  bool eof() const;

  // private member functions
  const match_type & dereference() const;
  void increment();
  bool equal(const find_iterator &) const;
};
```

Description

Find iterator encapsulates a Finder and allows for incremental searching in a string. Each increment moves the iterator to the next match.

Find iterator is a readable forward traversal iterator.

Dereferencing the iterator yields an iterator_range delimiting the current match.

`find_iterator` public construct/copy/destruct

1. ```
   find_iterator();
   ```

   Default constructor.

   Construct null iterator. All null iterators are equal.

   Postconditions: eof()==true

2. ```
   find_iterator(const find_iterator & Other);
   ```

   Copy constructor.

   Construct a copy of the find_iterator

3. ```
   template<typename FinderT>
     find_iterator(IteratorT Begin, IteratorT End, FinderT Finder);
   ```

   Constructor.

   Construct new find_iterator for a given finder and a range.
   ```

4.
```
template<typename FinderT, typename RangeT>
  find_iterator(RangeT & Col, FinderT Finder);
```

Constructor.

Construct new find_iterator for a given finder and a range.

`find_iterator` public member functions

1.
```
bool eof() const;
```

Eof check.

Check the eof condition. Eof condition means that there is nothing more to be searched i.e. find_iterator is after the last match.

`find_iterator` private member functions

1.
```
const match_type & dereference() const;
```

2.
```
void increment();
```

3.
```
bool equal(const find_iterator & Other) const;
```

Class template split_iterator

boost::algorithm::split_iterator — split_iterator

Synopsis

```
// In header: <boost/algorithm/string/find_iterator.hpp>

template<typename IteratorT>
class split_iterator {
public:
  // construct/copy/destruct
  split_iterator();
  split_iterator(const split_iterator &);
  template<typename FinderT> split_iterator(IteratorT, IteratorT, FinderT);
  template<typename FinderT, typename RangeT>
    split_iterator(RangeT &, FinderT);

  // public member functions
  bool eof() const;

  // private member functions
  const match_type & dereference() const;
  void increment();
  bool equal(const split_iterator &) const;
};
```

Description

Split iterator encapsulates a Finder and allows for incremental searching in a string. Unlike the find iterator, split iterator iterates through gaps between matches.

Find iterator is a readable forward traversal iterator.

Dereferencing the iterator yields an iterator_range delimiting the current match.

`split_iterator` public construct/copy/destruct

1. ```
   split_iterator();
   ```

   Default constructor.

   Construct null iterator. All null iterators are equal.

   Postconditions: eof()==true

2. ```
   split_iterator(const split_iterator & Other);
   ```

   Copy constructor.

   Construct a copy of the split_iterator

3. ```
   template<typename FinderT>
     split_iterator(IteratorT Begin, IteratorT End, FinderT Finder);
   ```

   Constructor.

   Construct new split_iterator for a given finder and a range.
   ```

4.
```
template<typename FinderT, typename RangeT>
  split_iterator(RangeT & Col, FinderT Finder);
```

Constructor.

Construct new split_iterator for a given finder and a collection.

`split_iterator` public member functions

1.
```
bool eof() const;
```

Eof check.

Check the eof condition. Eof condition means that there is nothing more to be searched i.e. find_iterator is after the last match.

`split_iterator` private member functions

1.
```
const match_type & dereference() const;
```

2.
```
void increment();
```

3.
```
bool equal(const split_iterator & Other) const;
```

Function template make_find_iterator

boost::algorithm::make_find_iterator — find iterator construction helper

Synopsis

```
// In header: <boost/algorithm/string/find_iterator.hpp>


template<typename RangeT, typename FinderT>
  find_iterator< typename range_iterator< RangeT >::type >
  make_find_iterator(RangeT & Collection, FinderT Finder);
```

Description

Construct a find iterator to iterate through the specified string

Function template make_split_iterator

boost::algorithm::make_split_iterator — split iterator construction helper

Synopsis

```
// In header: <boost/algorithm/string/find_iterator.hpp>


template<typename RangeT, typename FinderT>
  split_iterator< typename range_iterator< RangeT >::type >
  make_split_iterator(RangeT & Collection, FinderT Finder);
```

Description

Construct a split iterator to iterate through the specified collection

Header <boost/algorithm/string/finder.hpp>

Defines Finder generators. Finder object is a functor which is able to find a substring matching a specific criteria in the input. Finders are used as a pluggable components for replace, find and split facilities. This header contains generator functions for finders provided in this library.

```
namespace boost {
  namespace algorithm {
    template<typename RangeT> unspecified first_finder(const RangeT &);
    template<typename RangeT, typename PredicateT>
      unspecified first_finder(const RangeT &, PredicateT);
    template<typename RangeT> unspecified last_finder(const RangeT &);
    template<typename RangeT, typename PredicateT>
      unspecified last_finder(const RangeT &, PredicateT);
    template<typename RangeT> unspecified nth_finder(const RangeT &, int);
    template<typename RangeT, typename PredicateT>
      unspecified nth_finder(const RangeT &, int, PredicateT);
    unspecified head_finder(int);
    unspecified tail_finder(int);
    template<typename PredicateT>
      unspecified token_finder(PredicateT,
                               token_compress_mode_type = token_compress_off);
    template<typename ForwardIteratorT>
      unspecified range_finder(ForwardIteratorT, ForwardIteratorT);
    template<typename ForwardIteratorT>
      unspecified range_finder(iterator_range< ForwardIteratorT >);
  }
}
```

Function first_finder

boost::algorithm::first_finder — "First" finder

Synopsis

```
// In header: <boost/algorithm/string/finder.hpp>


template<typename RangeT> unspecified first_finder(const RangeT & Search);
template<typename RangeT, typename PredicateT>
  unspecified first_finder(const RangeT & Search, PredicateT Comp);
```

Description

Construct the `first_finder`. The finder searches for the first occurrence of the string in a given input. The result is given as an `iterator_range` delimiting the match.

Parameters: `Search` A substring to be searched for.
Returns: An instance of the `first_finder` object

Function last_finder

boost::algorithm::last_finder — "Last" finder

Synopsis

```
// In header: <boost/algorithm/string/finder.hpp>


template<typename RangeT> unspecified last_finder(const RangeT & Search);
template<typename RangeT, typename PredicateT>
  unspecified last_finder(const RangeT & Search, PredicateT Comp);
```

Description

Construct the `last_finder`. The finder searches for the last occurrence of the string in a given input. The result is given as an `iterator_range` delimiting the match.

Parameters:	`Search`	A substring to be searched for.
Returns:	An instance of the `last_finder` object	

Function nth_finder

boost::algorithm::nth_finder — "Nth" finder

Synopsis

```
// In header: <boost/algorithm/string/finder.hpp>


template<typename RangeT>
  unspecified nth_finder(const RangeT & Search, int Nth);
template<typename RangeT, typename PredicateT>
  unspecified nth_finder(const RangeT & Search, int Nth, PredicateT Comp);
```

Description

Construct the `nth_finder`. The finder searches for the n-th (zero-indexed) occurrence of the string in a given input. The result is given as an `iterator_range` delimiting the match.

Parameters:	`Nth`	An index of the match to be find
	`Search`	A substring to be searched for.
Returns:	An instance of the `nth_finder` object	

Function head_finder

boost::algorithm::head_finder — "Head" finder

Synopsis

```
// In header: <boost/algorithm/string/finder.hpp>


unspecified head_finder(int N);
```

Description

Construct the `head_finder`. The finder returns a head of a given input. The head is a prefix of a string up to n elements in size. If an input has less then n elements, whole input is considered a head. The result is given as an `iterator_range` delimiting the match.

Parameters: `N` The size of the head
Returns: An instance of the `head_finder` object

Function tail_finder

boost::algorithm::tail_finder — "Tail" finder

Synopsis

```
// In header: <boost/algorithm/string/finder.hpp>


unspecified tail_finder(int N);
```

Description

Construct the `tail_finder`. The finder returns a tail of a given input. The tail is a suffix of a string up to n elements in size. If an input has less then n elements, whole input is considered a head. The result is given as an `iterator_range` delimiting the match.

Parameters: `N` The size of the head
Returns: An instance of the `tail_finder` object

Function template token_finder

boost::algorithm::token_finder — "Token" finder

Synopsis

```
// In header: <boost/algorithm/string/finder.hpp>


template<typename PredicateT>
  unspecified token_finder(PredicateT Pred,
                           token_compress_mode_type eCompress = token_compress_off);
```

Description

Construct the `token_finder`. The finder searches for a token specified by a predicate. It is similar to std::find_if algorithm, with an exception that it return a range of instead of a single iterator.

If "compress token mode" is enabled, adjacent matching tokens are concatenated into one match. Thus the finder can be used to search for continuous segments of characters satisfying the given predicate.

The result is given as an `iterator_range` delimiting the match.

Parameters:	`Pred`	An element selection predicate
	`eCompress`	Compress flag
Returns:	An instance of the `token_finder` object	

Function range_finder

boost::algorithm::range_finder — "Range" finder

Synopsis

```
// In header: <boost/algorithm/string/finder.hpp>


template<typename ForwardIteratorT>
  unspecified range_finder(ForwardIteratorT Begin, ForwardIteratorT End);
template<typename ForwardIteratorT>
  unspecified range_finder(iterator_range< ForwardIteratorT > Range);
```

Description

Construct the `range_finder`. The finder does not perform any operation. It simply returns the given range for any input.

Parameters:	`Begin`	Beginning of the range
	`End`	End of the range
Returns:	An instance of the `range_finger` object	

Header <boost/algorithm/string/formatter.hpp>

Defines Formatter generators. Formatter is a functor which formats a string according to given parameters. A Formatter works in conjunction with a Finder. A Finder can provide additional information for a specific Formatter. An example of such a cooperation is regex_finder and regex_formatter.

Formatters are used as pluggable components for replace facilities. This header contains generator functions for the Formatters provided in this library.

```
namespace boost {
  namespace algorithm {
    template<typename RangeT> unspecified const_formatter(const RangeT &);
    template<typename RangeT> unspecified identity_formatter();
    template<typename RangeT> unspecified empty_formatter(const RangeT &);
  }
}
```

Function template const_formatter

boost::algorithm::const_formatter — Constant formatter.

Synopsis

```
// In header: <boost/algorithm/string/formatter.hpp>


template<typename RangeT> unspecified const_formatter(const RangeT & Format);
```

Description

Construct the `const_formatter`. Const formatter always returns the same value, regardless of the parameter.

Parameters:	`Format`	A predefined value used as a result for formating
Returns:	An instance of the `const_formatter` object.	

Function template identity_formatter

boost::algorithm::identity_formatter — Identity formatter.

Synopsis

```
// In header: <boost/algorithm/string/formatter.hpp>

template<typename RangeT> unspecified identity_formatter();
```

Description

Construct the `identity_formatter`. Identity formatter always returns the parameter.

Returns: An instance of the `identity_formatter` object.

Function template empty_formatter

boost::algorithm::empty_formatter — Empty formatter.

Synopsis

```
// In header: <boost/algorithm/string/formatter.hpp>


template<typename RangeT> unspecified empty_formatter(const RangeT &);
```

Description

Construct the `empty_formatter`. Empty formatter always returns an empty sequence.

Returns: An instance of the `empty_formatter` object.

Header <boost/algorithm/string/iter_find.hpp>

Defines generic split algorithms. Split algorithms can be used to divide a sequence into several part according to a given criteria. Result is given as a 'container of containers' where elements are copies or references to extracted parts.

There are two algorithms provided. One iterates over matching substrings, the other one over the gaps between these matches.

```
namespace boost {
  namespace algorithm {
    template<typename SequenceSequenceT, typename RangeT, typename FinderT>
      SequenceSequenceT & iter_find(SequenceSequenceT &, RangeT &, FinderT);
    template<typename SequenceSequenceT, typename RangeT, typename FinderT>
      SequenceSequenceT & iter_split(SequenceSequenceT &, RangeT &, FinderT);
  }
}
```

Function template iter_find

boost::algorithm::iter_find — Iter find algorithm.

Synopsis

```
// In header: <boost/algorithm/string/iter_find.hpp>


template<typename SequenceSequenceT, typename RangeT, typename FinderT>
  SequenceSequenceT &
  iter_find(SequenceSequenceT & Result, RangeT & Input, FinderT Finder);
```

Description

This algorithm executes a given finder in iteration on the input, until the end of input is reached, or no match is found. Iteration is done using built-in find_iterator, so the real searching is performed only when needed. In each iteration new match is found and added to the result.

Parameters:	`Finder`	A Finder object used for searching
	`Input`	A container which will be searched.
	`Result`	A 'container container' to contain the result of search. Both outer and inner container must have constructor taking a pair of iterators as an argument. Typical type of the result is `std::vector<boost::iterator_range<iterator>>` (each element of such a vector will container a range delimiting a match).
Returns:	A reference the result	
Notes:	Prior content of the result will be overwritten.	

Function template iter_split

boost::algorithm::iter_split — Split find algorithm.

Synopsis

```
// In header: <boost/algorithm/string/iter_find.hpp>


template<typename SequenceSequenceT, typename RangeT, typename FinderT>
  SequenceSequenceT &
  iter_split(SequenceSequenceT & Result, RangeT & Input, FinderT Finder);
```

Description

This algorithm executes a given finder in iteration on the input, until the end of input is reached, or no match is found. Iteration is done using built-in find_iterator, so the real searching is performed only when needed. Each match is used as a separator of segments. These segments are then returned in the result.

Parameters:	`Finder`	A finder object used for searching
	`Input`	A container which will be searched.
	`Result`	A 'container container' to container the result of search. Both outer and inner container must have constructor taking a pair of iterators as an argument. Typical type of the result is `std::vector<boost::iterator_range<iterator>>` (each element of such a vector will container a range delimiting a match).
Returns:	A reference the result	
Notes:	Prior content of the result will be overwritten.	

Header <boost/algorithm/string/join.hpp>

Defines join algorithm.

Join algorithm is a counterpart to split algorithms. It joins strings from a 'list' by adding user defined separator. Additionally there is a version that allows simple filtering by providing a predicate.

```
namespace boost {
  namespace algorithm {
    template<typename SequenceSequenceT, typename Range1T>
      range_value< SequenceSequenceT >::type
      join(const SequenceSequenceT &, const Range1T &);
    template<typename SequenceSequenceT, typename Range1T,
             typename PredicateT>
      range_value< SequenceSequenceT >::type
      join_if(const SequenceSequenceT &, const Range1T &, PredicateT);
  }
}
```

Function template join

boost::algorithm::join — Join algorithm.

Synopsis

```
// In header: <boost/algorithm/string/join.hpp>


template<typename SequenceSequenceT, typename Range1T>
  range_value< SequenceSequenceT >::type
  join(const SequenceSequenceT & Input, const Range1T & Separator);
```

Description

This algorithm joins all strings in a 'list' into one long string. Segments are concatenated by given separator.

Parameters:	`Input`	A container that holds the input strings. It must be a container-of-containers.
	`Separator`	A string that will separate the joined segments.
Returns:	Concatenated string.	
Notes:	This function provides the strong exception-safety guarantee	

Function template join_if

boost::algorithm::join_if — Conditional join algorithm.

Synopsis

```
// In header: <boost/algorithm/string/join.hpp>


template<typename SequenceSequenceT, typename Range1T, typename PredicateT>
  range_value< SequenceSequenceT >::type
  join_if(const SequenceSequenceT & Input, const Range1T & Separator,
          PredicateT Pred);
```

Description

This algorithm joins all strings in a 'list' into one long string. Segments are concatenated by given separator. Only segments that satisfy the predicate will be added to the result.

Parameters:	`Input`	A container that holds the input strings. It must be a container-of-containers.
	`Pred`	A segment selection predicate
	`Separator`	A string that will separate the joined segments.
Returns:	Concatenated string.	
Notes:	This function provides the strong exception-safety guarantee	

Header <boost/algorithm/string/predicate.hpp>

Defines string-related predicates. The predicates determine whether a substring is contained in the input string under various conditions: a string starts with the substring, ends with the substring, simply contains the substring or if both strings are equal. Additionaly the algorithm `all()` checks all elements of a container to satisfy a condition.

All predicates provide the strong exception guarantee.

```
namespace boost {
  namespace algorithm {
    template<typename Range1T, typename Range2T, typename PredicateT>
      bool starts_with(const Range1T &, const Range2T &, PredicateT);
    template<typename Range1T, typename Range2T>
      bool starts_with(const Range1T &, const Range2T &);
    template<typename Range1T, typename Range2T>
      bool istarts_with(const Range1T &, const Range2T &,
                        const std::locale & = std::locale());
    template<typename Range1T, typename Range2T, typename PredicateT>
      bool ends_with(const Range1T &, const Range2T &, PredicateT);
    template<typename Range1T, typename Range2T>
      bool ends_with(const Range1T &, const Range2T &);
    template<typename Range1T, typename Range2T>
      bool iends_with(const Range1T &, const Range2T &,
                      const std::locale & = std::locale());
    template<typename Range1T, typename Range2T, typename PredicateT>
      bool contains(const Range1T &, const Range2T &, PredicateT);
    template<typename Range1T, typename Range2T>
      bool contains(const Range1T &, const Range2T &);
    template<typename Range1T, typename Range2T>
      bool icontains(const Range1T &, const Range2T &,
                     const std::locale & = std::locale());
    template<typename Range1T, typename Range2T, typename PredicateT>
      bool equals(const Range1T &, const Range2T &, PredicateT);
    template<typename Range1T, typename Range2T>
      bool equals(const Range1T &, const Range2T &);
    template<typename Range1T, typename Range2T>
      bool iequals(const Range1T &, const Range2T &,
                   const std::locale & = std::locale());
    template<typename Range1T, typename Range2T, typename PredicateT>
      bool lexicographical_compare(const Range1T &, const Range2T &,
                                   PredicateT);
    template<typename Range1T, typename Range2T>
      bool lexicographical_compare(const Range1T &, const Range2T &);
    template<typename Range1T, typename Range2T>
      bool ilexicographical_compare(const Range1T &, const Range2T &,
                                    const std::locale & = std::locale());
    template<typename RangeT, typename PredicateT>
      bool all(const RangeT &, PredicateT);
  }
}
```

Function starts_with

boost::algorithm::starts_with — 'Starts with' predicate

Synopsis

```
// In header: <boost/algorithm/string/predicate.hpp>


template<typename Range1T, typename Range2T, typename PredicateT>
  bool starts_with(const Range1T & Input, const Range2T & Test,
                   PredicateT Comp);
template<typename Range1T, typename Range2T>
  bool starts_with(const Range1T & Input, const Range2T & Test);
```

Description

This predicate holds when the test string is a prefix of the Input. In other words, if the input starts with the test. When the optional predicate is specified, it is used for character-wise comparison.

Parameters:	`Comp`	An element comparison predicate
	`Input`	An input sequence
	`Test`	A test sequence
Returns:	The result of the test	
Notes:	This function provides the strong exception-safety guarantee	

Function template istarts_with

boost::algorithm::istarts_with — 'Starts with' predicate (case insensitive)

Synopsis

```
// In header: <boost/algorithm/string/predicate.hpp>


template<typename Range1T, typename Range2T>
  bool istarts_with(const Range1T & Input, const Range2T & Test,
                    const std::locale & Loc = std::locale());
```

Description

This predicate holds when the test string is a prefix of the Input. In other words, if the input starts with the test. Elements are compared case insensitively.

Parameters:	`Input`	An input sequence
	`Loc`	A locale used for case insensitive comparison
	`Test`	A test sequence
Returns:	The result of the test	
Notes:	This function provides the strong exception-safety guarantee	

Function ends_with

boost::algorithm::ends_with — 'Ends with' predicate

Synopsis

```
// In header: <boost/algorithm/string/predicate.hpp>


template<typename Range1T, typename Range2T, typename PredicateT>
  bool ends_with(const Range1T & Input, const Range2T & Test, PredicateT Comp);
template<typename Range1T, typename Range2T>
  bool ends_with(const Range1T & Input, const Range2T & Test);
```

Description

This predicate holds when the test string is a suffix of the Input. In other words, if the input ends with the test. When the optional predicate is specified, it is used for character-wise comparison.

Parameters:	`Comp`	An element comparison predicate
	`Input`	An input sequence
	`Test`	A test sequence
Returns:	The result of the test	
Notes:	This function provides the strong exception-safety guarantee	

Function template iends_with

boost::algorithm::iends_with — 'Ends with' predicate (case insensitive)

Synopsis

```
// In header: <boost/algorithm/string/predicate.hpp>


template<typename Range1T, typename Range2T>
  bool iends_with(const Range1T & Input, const Range2T & Test,
                  const std::locale & Loc = std::locale());
```

Description

This predicate holds when the test container is a suffix of the Input. In other words, if the input ends with the test. Elements are compared case insensitively.

Parameters:	`Input`	An input sequence
	`Loc`	A locale used for case insensitive comparison
	`Test`	A test sequence
Returns:	The result of the test	
Notes:	This function provides the strong exception-safety guarantee	

Function contains

boost::algorithm::contains — 'Contains' predicate

Synopsis

```
// In header: <boost/algorithm/string/predicate.hpp>


template<typename Range1T, typename Range2T, typename PredicateT>
  bool contains(const Range1T & Input, const Range2T & Test, PredicateT Comp);
template<typename Range1T, typename Range2T>
  bool contains(const Range1T & Input, const Range2T & Test);
```

Description

This predicate holds when the test container is contained in the Input. When the optional predicate is specified, it is used for character-wise comparison.

Parameters:	`Comp`	An element comparison predicate
	`Input`	An input sequence
	`Test`	A test sequence
Returns:	The result of the test	
Notes:	This function provides the strong exception-safety guarantee	

Function template icontains

boost::algorithm::icontains — 'Contains' predicate (case insensitive)

Synopsis

```
// In header: <boost/algorithm/string/predicate.hpp>


template<typename Range1T, typename Range2T>
  bool icontains(const Range1T & Input, const Range2T & Test,
                 const std::locale & Loc = std::locale());
```

Description

This predicate holds when the test container is contained in the Input. Elements are compared case insensitively.

Parameters:	`Input`	An input sequence
	`Loc`	A locale used for case insensitive comparison
	`Test`	A test sequence
Returns:	The result of the test	
Notes:	This function provides the strong exception-safety guarantee	

Function equals

boost::algorithm::equals — 'Equals' predicate

Synopsis

```
// In header: <boost/algorithm/string/predicate.hpp>


template<typename Range1T, typename Range2T, typename PredicateT>
  bool equals(const Range1T & Input, const Range2T & Test, PredicateT Comp);
template<typename Range1T, typename Range2T>
  bool equals(const Range1T & Input, const Range2T & Test);
```

Description

This predicate holds when the test container is equal to the input container i.e. all elements in both containers are same. When the optional predicate is specified, it is used for character-wise comparison.

Parameters:	`Comp`	An element comparison predicate
	`Input`	An input sequence
	`Test`	A test sequence
Returns:	The result of the test	
Notes:	This is a two-way version of `std::equal` algorithm	
	This function provides the strong exception-safety guarantee	

Function template iequals

boost::algorithm::iequals — 'Equals' predicate (case insensitive)

Synopsis

```
// In header: <boost/algorithm/string/predicate.hpp>


template<typename Range1T, typename Range2T>
  bool iequals(const Range1T & Input, const Range2T & Test,
               const std::locale & Loc = std::locale());
```

Description

This predicate holds when the test container is equal to the input container i.e. all elements in both containers are same. Elements are compared case insensitively.

Parameters:	`Input`	An input sequence
	`Loc`	A locale used for case insensitive comparison
	`Test`	A test sequence
Returns:	The result of the test	
Notes:	This is a two-way version of `std::equal` algorithm	
	This function provides the strong exception-safety guarantee	

Function lexicographical_compare

boost::algorithm::lexicographical_compare — Lexicographical compare predicate.

Synopsis

```
// In header: <boost/algorithm/string/predicate.hpp>


template<typename Range1T, typename Range2T, typename PredicateT>
  bool lexicographical_compare(const Range1T & Arg1, const Range2T & Arg2,
                               PredicateT Pred);
template<typename Range1T, typename Range2T>
  bool lexicographical_compare(const Range1T & Arg1, const Range2T & Arg2);
```

Description

This predicate is an overload of std::lexicographical_compare for range arguments

It check whether the first argument is lexicographically less then the second one.

If the optional predicate is specified, it is used for character-wise comparison

Parameters:	`Arg1`	First argument
	`Arg2`	Second argument
	`Pred`	Comparison predicate
Returns:	The result of the test	
Notes:	This function provides the strong exception-safety guarantee	

Function template ilexicographical_compare

boost::algorithm::ilexicographical_compare — Lexicographical compare predicate (case-insensitive).

Synopsis

```
// In header: <boost/algorithm/string/predicate.hpp>


template<typename Range1T, typename Range2T>
  bool ilexicographical_compare(const Range1T & Arg1, const Range2T & Arg2,
                                const std::locale & Loc = std::locale());
```

Description

This predicate is an overload of std::lexicographical_compare for range arguments. It check whether the first argument is lexicographically less then the second one. Elements are compared case insensitively

Parameters:	`Arg1`	First argument
	`Arg2`	Second argument
	`Loc`	A locale used for case insensitive comparison
Returns:	The result of the test	
Notes:	This function provides the strong exception-safety guarantee	

Function template all

boost::algorithm::all — 'All' predicate

Synopsis

```
// In header: <boost/algorithm/string/predicate.hpp>


template<typename RangeT, typename PredicateT>
  bool all(const RangeT & Input, PredicateT Pred);
```

Description

This predicate holds it all its elements satisfy a given condition, represented by the predicate.

Parameters:	`Input`	An input sequence
	`Pred`	A predicate
Returns:	The result of the test	
Notes:	This function provides the strong exception-safety guarantee	

Header <boost/algorithm/string/regex.hpp>

Defines regex variants of the algorithms.

```
namespace boost {
  namespace algorithm {
    template<typename RangeT, typename CharT, typename RegexTraitsT>
      iterator_range< typename range_iterator< RangeT >::type >
      find_regex(RangeT &, const basic_regex< CharT, RegexTraitsT > &,
                 match_flag_type = match_default);
    template<typename OutputIteratorT, typename RangeT, typename CharT,
             typename RegexTraitsT, typename FormatStringTraitsT,
             typename FormatStringAllocatorT>
      OutputIteratorT
      replace_regex_copy(OutputIteratorT, const RangeT &,
                         const basic_regex< CharT, RegexTraitsT > &,
                        const std::basic_string< CharT, FormatStringTraitsT, FormatStringAlloc↵
atorT > &,
                         match_flag_type = match_default|format_default);
    template<typename SequenceT, typename CharT, typename RegexTraitsT,
             typename FormatStringTraitsT, typename FormatStringAllocatorT>
      SequenceT replace_regex_copy(const SequenceT &,
                                   const basic_regex< CharT, RegexTraitsT > &,
                                  const std::basic_string< CharT, FormatStringTraitsT, Format↵
StringAllocatorT > &,
                                   match_flag_type = match_default|format_default);
    template<typename SequenceT, typename CharT, typename RegexTraitsT,
             typename FormatStringTraitsT, typename FormatStringAllocatorT>
      void replace_regex(SequenceT &,
                         const basic_regex< CharT, RegexTraitsT > &,
                        const std::basic_string< CharT, FormatStringTraitsT, FormatStringAlloc↵
atorT > &,
                         match_flag_type = match_default|format_default);
    template<typename OutputIteratorT, typename RangeT, typename CharT,
             typename RegexTraitsT, typename FormatStringTraitsT,
             typename FormatStringAllocatorT>
      OutputIteratorT
      replace_all_regex_copy(OutputIteratorT, const RangeT &,
                             const basic_regex< CharT, RegexTraitsT > &,
                            const std::basic_string< CharT, FormatStringTraitsT, FormatStringAl↵
locatorT > &,
                             match_flag_type = match_default|format_default);
    template<typename SequenceT, typename CharT, typename RegexTraitsT,
             typename FormatStringTraitsT, typename FormatStringAllocatorT>
      SequenceT replace_all_regex_copy(const SequenceT &,
                                       const basic_regex< CharT, RegexTraitsT > &,
                                      const std::basic_string< CharT, FormatStringTraitsT, Format↵
StringAllocatorT > &,
                                       match_flag_type = match_default|format_default);
    template<typename SequenceT, typename CharT, typename RegexTraitsT,
             typename FormatStringTraitsT, typename FormatStringAllocatorT>
      void replace_all_regex(SequenceT &,
                             const basic_regex< CharT, RegexTraitsT > &,
                            const std::basic_string< CharT, FormatStringTraitsT, FormatStringAl↵
locatorT > &,
                             match_flag_type = match_default|format_default);
    template<typename OutputIteratorT, typename RangeT, typename CharT,
             typename RegexTraitsT>
      OutputIteratorT
      erase_regex_copy(OutputIteratorT, const RangeT &,
                       const basic_regex< CharT, RegexTraitsT > &,
                       match_flag_type = match_default);
    template<typename SequenceT, typename CharT, typename RegexTraitsT>
      SequenceT erase_regex_copy(const SequenceT &,
                                 const basic_regex< CharT, RegexTraitsT > &,
                                 match_flag_type = match_default);
    template<typename SequenceT, typename CharT, typename RegexTraitsT>
```

```
      void erase_regex(SequenceT &,
                       const basic_regex< CharT, RegexTraitsT > &,
                       match_flag_type = match_default);
    template<typename OutputIteratorT, typename RangeT, typename CharT,
             typename RegexTraitsT>
      OutputIteratorT
      erase_all_regex_copy(OutputIteratorT, const RangeT &,
                           const basic_regex< CharT, RegexTraitsT > &,
                           match_flag_type = match_default);
    template<typename SequenceT, typename CharT, typename RegexTraitsT>
      SequenceT erase_all_regex_copy(const SequenceT &,
                                     const basic_regex< CharT, RegexTraitsT > &,
                                     match_flag_type = match_default);
    template<typename SequenceT, typename CharT, typename RegexTraitsT>
      void erase_all_regex(SequenceT &,
                           const basic_regex< CharT, RegexTraitsT > &,
                           match_flag_type = match_default);
    template<typename SequenceSequenceT, typename RangeT, typename CharT,
             typename RegexTraitsT>
      SequenceSequenceT &
      find_all_regex(SequenceSequenceT &, const RangeT &,
                     const basic_regex< CharT, RegexTraitsT > &,
                     match_flag_type = match_default);
    template<typename SequenceSequenceT, typename RangeT, typename CharT,
             typename RegexTraitsT>
      SequenceSequenceT &
      split_regex(SequenceSequenceT &, const RangeT &,
                  const basic_regex< CharT, RegexTraitsT > &,
                  match_flag_type = match_default);
    template<typename SequenceSequenceT, typename Range1T, typename CharT,
             typename RegexTraitsT>
      range_value< SequenceSequenceT >::type
      join_if(const SequenceSequenceT &, const Range1T &,
              const basic_regex< CharT, RegexTraitsT > &,
              match_flag_type = match_default);
  }
}
```

Function template find_regex

boost::algorithm::find_regex — Find regex algorithm.

Synopsis

```
// In header: <boost/algorithm/string/regex.hpp>


template<typename RangeT, typename CharT, typename RegexTraitsT>
  iterator_range< typename range_iterator< RangeT >::type >
  find_regex(RangeT & Input, const basic_regex< CharT, RegexTraitsT > & Rx,
             match_flag_type Flags = match_default);
```

Description

Search for a substring matching the given regex in the input.

Parameters:	`Flags`	Regex options
	`Input`	A container which will be searched.
	`Rx`	A regular expression
Returns:	An `iterator_range` delimiting the match. Returned iterator is either `RangeT::iterator` or `RangeT::const_iterator`, depending on the constness of the input parameter.	
Notes:	This function provides the strong exception-safety guarantee	

Function replace_regex_copy

boost::algorithm::replace_regex_copy — Replace regex algorithm.

Synopsis

```
// In header: <boost/algorithm/string/regex.hpp>


template<typename OutputIteratorT, typename RangeT, typename CharT,
         typename RegexTraitsT, typename FormatStringTraitsT,
         typename FormatStringAllocatorT>
  OutputIteratorT
  replace_regex_copy(OutputIteratorT Output, const RangeT & Input,
                     const basic_regex< CharT, RegexTraitsT > & Rx,
                     const std::basic_string< CharT, FormatStringTraitsT, FormatStringAlloc↵
atorT > & Format,
                     match_flag_type Flags = match_default|format_default);
template<typename SequenceT, typename CharT, typename RegexTraitsT,
         typename FormatStringTraitsT, typename FormatStringAllocatorT>
  SequenceT replace_regex_copy(const SequenceT & Input,
                               const basic_regex< CharT, RegexTraitsT > & Rx,
                              const std::basic_string< CharT, FormatStringTraitsT, FormatStrin↵
gAllocatorT > & Format,
                               match_flag_type Flags = match_default|format_default);
```

Description

Search for a substring matching given regex and format it with the specified format. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters:	`Flags`	Regex options
	`Format`	Regex format definition
	`Input`	An input string
	`Output`	An output iterator to which the result will be copied
	`Rx`	A regular expression
Returns:	An output iterator pointing just after the last inserted character or a modified copy of the input	
Notes:	The second variant of this function provides the strong exception-safety guarantee	

Function template replace_regex

boost::algorithm::replace_regex — Replace regex algorithm.

Synopsis

```
// In header: <boost/algorithm/string/regex.hpp>


template<typename SequenceT, typename CharT, typename RegexTraitsT,
         typename FormatStringTraitsT, typename FormatStringAllocatorT>
  void replace_regex(SequenceT & Input,
                     const basic_regex< CharT, RegexTraitsT > & Rx,
                     const std::basic_string< CharT, FormatStringTraitsT, FormatStringAlloc↵
atorT > & Format,
                     match_flag_type Flags = match_default|format_default);
```

Description

Search for a substring matching given regex and format it with the specified format. The input string is modified in-place.

Parameters:	`Flags`	Regex options
	`Format`	Regex format definition
	`Input`	An input string
	`Rx`	A regular expression

Function replace_all_regex_copy

boost::algorithm::replace_all_regex_copy — Replace all regex algorithm.

Synopsis

```
// In header: <boost/algorithm/string/regex.hpp>


template<typename OutputIteratorT, typename RangeT, typename CharT,
         typename RegexTraitsT, typename FormatStringTraitsT,
         typename FormatStringAllocatorT>
  OutputIteratorT
  replace_all_regex_copy(OutputIteratorT Output, const RangeT & Input,
                         const basic_regex< CharT, RegexTraitsT > & Rx,
                        const std::basic_string< CharT, FormatStringTraitsT, FormatStringAlloc↵
atorT > & Format,
                         match_flag_type Flags = match_default|format_default);
template<typename SequenceT, typename CharT, typename RegexTraitsT,
         typename FormatStringTraitsT, typename FormatStringAllocatorT>
  SequenceT replace_all_regex_copy(const SequenceT & Input,
                                   const basic_regex< CharT, RegexTraitsT > & Rx,
                                  const std::basic_string< CharT, FormatStringTraitsT, Format↵
StringAllocatorT > & Format,
                                   match_flag_type Flags = match_default|format_default);
```

Description

Format all substrings, matching given regex, with the specified format. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters:	`Flags`	Regex options
	`Format`	Regex format definition
	`Input`	An input string
	`Output`	An output iterator to which the result will be copied
	`Rx`	A regular expression
Returns:	An output iterator pointing just after the last inserted character or a modified copy of the input	
Notes:	The second variant of this function provides the strong exception-safety guarantee	

Function template replace_all_regex

boost::algorithm::replace_all_regex — Replace all regex algorithm.

Synopsis

```
// In header: <boost/algorithm/string/regex.hpp>


template<typename SequenceT, typename CharT, typename RegexTraitsT,
         typename FormatStringTraitsT, typename FormatStringAllocatorT>
  void replace_all_regex(SequenceT & Input,
                         const basic_regex< CharT, RegexTraitsT > & Rx,
                        const std::basic_string< CharT, FormatStringTraitsT, FormatStringAlloc↵
atorT > & Format,
                         match_flag_type Flags = match_default|format_default);
```

Description

Format all substrings, matching given regex, with the specified format. The input string is modified in-place.

Parameters:	`Flags`	Regex options
	`Format`	Regex format definition
	`Input`	An input string
	`Rx`	A regular expression

Function erase_regex_copy

boost::algorithm::erase_regex_copy — Erase regex algorithm.

Synopsis

```
// In header: <boost/algorithm/string/regex.hpp>


template<typename OutputIteratorT, typename RangeT, typename CharT,
         typename RegexTraitsT>
  OutputIteratorT
  erase_regex_copy(OutputIteratorT Output, const RangeT & Input,
                   const basic_regex< CharT, RegexTraitsT > & Rx,
                   match_flag_type Flags = match_default);
template<typename SequenceT, typename CharT, typename RegexTraitsT>
  SequenceT erase_regex_copy(const SequenceT & Input,
                             const basic_regex< CharT, RegexTraitsT > & Rx,
                             match_flag_type Flags = match_default);
```

Description

Remove a substring matching given regex from the input. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters:	`Flags`	Regex options
	`Input`	An input string
	`Output`	An output iterator to which the result will be copied
	`Rx`	A regular expression
Returns:	An output iterator pointing just after the last inserted character or a modified copy of the input	
Notes:	The second variant of this function provides the strong exception-safety guarantee	

Function template erase_regex

boost::algorithm::erase_regex — Erase regex algorithm.

Synopsis

```
// In header: <boost/algorithm/string/regex.hpp>


template<typename SequenceT, typename CharT, typename RegexTraitsT>
  void erase_regex(SequenceT & Input,
                   const basic_regex< CharT, RegexTraitsT > & Rx,
                   match_flag_type Flags = match_default);
```

Description

Remove a substring matching given regex from the input. The input string is modified in-place.

Parameters:	`Flags`	Regex options
	`Input`	An input string
	`Rx`	A regular expression

Function erase_all_regex_copy

boost::algorithm::erase_all_regex_copy — Erase all regex algorithm.

Synopsis

```
// In header: <boost/algorithm/string/regex.hpp>


template<typename OutputIteratorT, typename RangeT, typename CharT,
         typename RegexTraitsT>
  OutputIteratorT
  erase_all_regex_copy(OutputIteratorT Output, const RangeT & Input,
                       const basic_regex< CharT, RegexTraitsT > & Rx,
                       match_flag_type Flags = match_default);
template<typename SequenceT, typename CharT, typename RegexTraitsT>
  SequenceT erase_all_regex_copy(const SequenceT & Input,
                                 const basic_regex< CharT, RegexTraitsT > & Rx,
                                 match_flag_type Flags = match_default);
```

Description

Erase all substrings, matching given regex, from the input. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters:	`Flags`	Regex options
	`Input`	An input string
	`Output`	An output iterator to which the result will be copied
	`Rx`	A regular expression
Returns:	An output iterator pointing just after the last inserted character or a modified copy of the input	
Notes:	The second variant of this function provides the strong exception-safety guarantee	

Function template erase_all_regex

boost::algorithm::erase_all_regex — Erase all regex algorithm.

Synopsis

```
// In header: <boost/algorithm/string/regex.hpp>


template<typename SequenceT, typename CharT, typename RegexTraitsT>
  void erase_all_regex(SequenceT & Input,
                       const basic_regex< CharT, RegexTraitsT > & Rx,
                       match_flag_type Flags = match_default);
```

Description

Erase all substrings, matching given regex, from the input. The input string is modified in-place.

Parameters:	`Flags`	Regex options
	`Input`	An input string
	`Rx`	A regular expression

Function template find_all_regex

boost::algorithm::find_all_regex — Find all regex algorithm.

Synopsis

```
// In header: <boost/algorithm/string/regex.hpp>


template<typename SequenceSequenceT, typename RangeT, typename CharT,
         typename RegexTraitsT>
  SequenceSequenceT &
  find_all_regex(SequenceSequenceT & Result, const RangeT & Input,
                 const basic_regex< CharT, RegexTraitsT > & Rx,
                 match_flag_type Flags = match_default);
```

Description

This algorithm finds all substrings matching the give regex in the input.

Each part is copied and added as a new element to the output container. Thus the result container must be able to hold copies of the matches (in a compatible structure like std::string) or a reference to it (e.g. using the iterator range class). Examples of such a container are `std::vector<std::string>` or `std::list<boost::iterator_range<std::string::iterator>>`

Parameters:	`Flags`	Regex options
	`Input`	A container which will be searched.
	`Result`	A container that can hold copies of references to the substrings.
	`Rx`	A regular expression
Returns:	A reference to the result	
Notes:	Prior content of the result will be overwritten.	
	This function provides the strong exception-safety guarantee	

Function template split_regex

boost::algorithm::split_regex — Split regex algorithm.

Synopsis

```
// In header: <boost/algorithm/string/regex.hpp>


template<typename SequenceSequenceT, typename RangeT, typename CharT,
         typename RegexTraitsT>
  SequenceSequenceT &
  split_regex(SequenceSequenceT & Result, const RangeT & Input,
              const basic_regex< CharT, RegexTraitsT > & Rx,
              match_flag_type Flags = match_default);
```

Description

Tokenize expression. This function is equivalent to C strtok. Input sequence is split into tokens, separated by separators. Separator is an every match of the given regex. Each part is copied and added as a new element to the output container. Thus the result container must be able to hold copies of the matches (in a compatible structure like std::string) or a reference to it (e.g. using the iterator range class). Examples of such a container are `std::vector<std::string>` or `std::list<boost::iterator_range<std::string::iterator>>`

Parameters:	`Flags`	Regex options
	`Input`	A container which will be searched.
	`Result`	A container that can hold copies of references to the substrings.
	`Rx`	A regular expression
Returns:	A reference to the result	
Notes:	Prior content of the result will be overwritten.	
	This function provides the strong exception-safety guarantee	

Function template join_if

boost::algorithm::join_if — Conditional join algorithm.

Synopsis

```
// In header: <boost/algorithm/string/regex.hpp>


template<typename SequenceSequenceT, typename Range1T, typename CharT,
         typename RegexTraitsT>
  range_value< SequenceSequenceT >::type
  join_if(const SequenceSequenceT & Input, const Range1T & Separator,
          const basic_regex< CharT, RegexTraitsT > & Rx,
          match_flag_type Flags = match_default);
```

Description

This algorithm joins all strings in a 'list' into one long string. Segments are concatenated by given separator. Only segments that match the given regular expression will be added to the result

This is a specialization of join_if algorithm.

Parameters:	`Flags`	Regex options
	`Input`	A container that holds the input strings. It must be a container-of-containers.
	`Rx`	A regular expression
	`Separator`	A string that will separate the joined segments.
Returns:	Concatenated string.	
Notes:	This function provides the strong exception-safety guarantee	

Header <boost/algorithm/string/regex_find_format.hpp>

Defines the `regex_finder` and `regex_formatter` generators. These two functors are designed to work together. `regex_formatter` uses additional information about a match contained in the regex_finder search result.

```
namespace boost {
  namespace algorithm {
    template<typename CharT, typename RegexTraitsT>
      unspecified regex_finder(const basic_regex< CharT, RegexTraitsT > &,
                               match_flag_type = match_default);
    template<typename CharT, typename TraitsT, typename AllocT>
      unspecified regex_formatter(const std::basic_string< CharT, TraitsT, AllocT > &,
                                  match_flag_type = format_default);
  }
}
```

Function template regex_finder

boost::algorithm::regex_finder — "Regex" finder

Synopsis

```
// In header: <boost/algorithm/string/regex_find_format.hpp>


template<typename CharT, typename RegexTraitsT>
  unspecified regex_finder(const basic_regex< CharT, RegexTraitsT > & Rx,
                           match_flag_type MatchFlags = match_default);
```

Description

Construct the `regex_finder`. Finder uses the regex engine to search for a match. Result is given in `regex_search_result`. This is an extension of the iterator_range. In addition it containes match results from the `regex_search` algorithm.

Parameters:	`MatchFlags`	Regex search options
	`Rx`	A regular expression
Returns:	An instance of the `regex_finder` object	

Function template regex_formatter

boost::algorithm::regex_formatter — Regex formatter.

Synopsis

```
// In header: <boost/algorithm/string/regex_find_format.hpp>


template<typename CharT, typename TraitsT, typename AllocT>
  unspecified regex_formatter(const std::basic_string< CharT, TraitsT, AllocT > & Format,
                              match_flag_type Flags = format_default);
```

Description

Construct the `regex_formatter`. Regex formatter uses the regex engine to format a match found by the `regex_finder`. This formatted it designed to closely cooperate with `regex_finder`.

Parameters:	`Flags`	Format flags
	`Format`	Regex format definition
Returns:	An instance of the `regex_formatter` functor	

Header <boost/algorithm/string/replace.hpp>

Defines various replace algorithms. Each algorithm replaces part(s) of the input according to set of searching and replace criteria.

```
namespace boost {
  namespace algorithm {
    template<typename OutputIteratorT, typename Range1T, typename Range2T>
      OutputIteratorT
      replace_range_copy(OutputIteratorT, const Range1T &,
                         const iterator_range< typename range_const_iterator< Range1T >::type > &,
                            const Range2T &);
    template<typename SequenceT, typename RangeT>
      SequenceT replace_range_copy(const SequenceT &,
                                   const iterator_range< typename range_const_iterator< Sequen↵
ceT >::type > &,
                                   const RangeT &);
    template<typename SequenceT, typename RangeT>
      void replace_range(SequenceT &,
                         const iterator_range< typename range_iterator< SequenceT >::type > &,
                         const RangeT &);
    template<typename OutputIteratorT, typename Range1T, typename Range2T,
             typename Range3T>
      OutputIteratorT
      replace_first_copy(OutputIteratorT, const Range1T &, const Range2T &,
                         const Range3T &);
    template<typename SequenceT, typename Range1T, typename Range2T>
      SequenceT replace_first_copy(const SequenceT &, const Range1T &,
                                   const Range2T &);
    template<typename SequenceT, typename Range1T, typename Range2T>
      void replace_first(SequenceT &, const Range1T &, const Range2T &);
    template<typename OutputIteratorT, typename Range1T, typename Range2T,
             typename Range3T>
      OutputIteratorT
      ireplace_first_copy(OutputIteratorT, const Range1T &, const Range2T &,
                          const Range3T &,
                          const std::locale & = std::locale());
    template<typename SequenceT, typename Range2T, typename Range1T>
      SequenceT ireplace_first_copy(const SequenceT &, const Range2T &,
                                    const Range1T &,
                                    const std::locale & = std::locale());
    template<typename SequenceT, typename Range1T, typename Range2T>
      void ireplace_first(SequenceT &, const Range1T &, const Range2T &,
                          const std::locale & = std::locale());
    template<typename OutputIteratorT, typename Range1T, typename Range2T,
             typename Range3T>
      OutputIteratorT
      replace_last_copy(OutputIteratorT, const Range1T &, const Range2T &,
                        const Range3T &);
    template<typename SequenceT, typename Range1T, typename Range2T>
      SequenceT replace_last_copy(const SequenceT &, const Range1T &,
                                  const Range2T &);
    template<typename SequenceT, typename Range1T, typename Range2T>
      void replace_last(SequenceT &, const Range1T &, const Range2T &);
    template<typename OutputIteratorT, typename Range1T, typename Range2T,
             typename Range3T>
      OutputIteratorT
      ireplace_last_copy(OutputIteratorT, const Range1T &, const Range2T &,
                         const Range3T &,
                         const std::locale & = std::locale());
    template<typename SequenceT, typename Range1T, typename Range2T>
      SequenceT ireplace_last_copy(const SequenceT &, const Range1T &,
                                   const Range2T &,
                                   const std::locale & = std::locale());
    template<typename SequenceT, typename Range1T, typename Range2T>
      void ireplace_last(SequenceT &, const Range1T &, const Range2T &,
                         const std::locale & = std::locale());
    template<typename OutputIteratorT, typename Range1T, typename Range2T,
```

```
            typename Range3T>
      OutputIteratorT
      replace_nth_copy(OutputIteratorT, const Range1T &, const Range2T &, int,
                       const Range3T &);
    template<typename SequenceT, typename Range1T, typename Range2T>
      SequenceT replace_nth_copy(const SequenceT &, const Range1T &, int,
                                 const Range2T &);
    template<typename SequenceT, typename Range1T, typename Range2T>
      void replace_nth(SequenceT &, const Range1T &, int, const Range2T &);
    template<typename OutputIteratorT, typename Range1T, typename Range2T,
            typename Range3T>
      OutputIteratorT
      ireplace_nth_copy(OutputIteratorT, const Range1T &, const Range2T &,
                        int, const Range3T &,
                        const std::locale & = std::locale());
    template<typename SequenceT, typename Range1T, typename Range2T>
      SequenceT ireplace_nth_copy(const SequenceT &, const Range1T &, int,
                                  const Range2T &,
                                  const std::locale & = std::locale());
    template<typename SequenceT, typename Range1T, typename Range2T>
      void ireplace_nth(SequenceT &, const Range1T &, int, const Range2T &,
                        const std::locale & = std::locale());
    template<typename OutputIteratorT, typename Range1T, typename Range2T,
            typename Range3T>
      OutputIteratorT
      replace_all_copy(OutputIteratorT, const Range1T &, const Range2T &,
                       const Range3T &);
    template<typename SequenceT, typename Range1T, typename Range2T>
      SequenceT replace_all_copy(const SequenceT &, const Range1T &,
                                 const Range2T &);
    template<typename SequenceT, typename Range1T, typename Range2T>
      void replace_all(SequenceT &, const Range1T &, const Range2T &);
    template<typename OutputIteratorT, typename Range1T, typename Range2T,
            typename Range3T>
      OutputIteratorT
      ireplace_all_copy(OutputIteratorT, const Range1T &, const Range2T &,
                        const Range3T &, const std::locale & = std::locale());
    template<typename SequenceT, typename Range1T, typename Range2T>
      SequenceT ireplace_all_copy(const SequenceT &, const Range1T &,
                                  const Range2T &,
                                  const std::locale & = std::locale());
    template<typename SequenceT, typename Range1T, typename Range2T>
      void ireplace_all(SequenceT &, const Range1T &, const Range2T &,
                        const std::locale & = std::locale());
    template<typename OutputIteratorT, typename Range1T, typename Range2T>
      OutputIteratorT
      replace_head_copy(OutputIteratorT, const Range1T &, int,
                        const Range2T &);
    template<typename SequenceT, typename RangeT>
      SequenceT replace_head_copy(const SequenceT &, int, const RangeT &);
    template<typename SequenceT, typename RangeT>
      void replace_head(SequenceT &, int, const RangeT &);
    template<typename OutputIteratorT, typename Range1T, typename Range2T>
      OutputIteratorT
      replace_tail_copy(OutputIteratorT, const Range1T &, int,
                        const Range2T &);
    template<typename SequenceT, typename RangeT>
      SequenceT replace_tail_copy(const SequenceT &, int, const RangeT &);
    template<typename SequenceT, typename RangeT>
      void replace_tail(SequenceT &, int, const RangeT &);
  }
}
```

Function replace_range_copy

boost::algorithm::replace_range_copy — Replace range algorithm.

Synopsis

```
// In header: <boost/algorithm/string/replace.hpp>


template<typename OutputIteratorT, typename Range1T, typename Range2T>
  OutputIteratorT
  replace_range_copy(OutputIteratorT Output, const Range1T & Input,
                     const iterator_range< typename range_const_iterator< Range1T >::type > & Sear↵
chRange,
                        const Range2T & Format);
template<typename SequenceT, typename RangeT>
  SequenceT replace_range_copy(const SequenceT & Input,
                               const iterator_range< typename range_const_iterator< Sequen↵
ceT >::type > & SearchRange,
                               const RangeT & Format);
```

Description

Replace the given range in the input string. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters:	`Format`	A substitute string
	`Input`	An input string
	`Output`	An output iterator to which the result will be copied
	`SearchRange`	A range in the input to be substituted
Returns:	An output iterator pointing just after the last inserted character or a modified copy of the input	
Notes:	The second variant of this function provides the strong exception-safety guarantee	

Function template replace_range

boost::algorithm::replace_range — Replace range algorithm.

Synopsis

```
// In header: <boost/algorithm/string/replace.hpp>


template<typename SequenceT, typename RangeT>
  void replace_range(SequenceT & Input,
                     const iterator_range< typename range_iterator< SequenceT >::type > & Sear↵
chRange,
                     const RangeT & Format);
```

Description

Replace the given range in the input string. The input sequence is modified in-place.

Parameters:		
	`Format`	A substitute string
	`Input`	An input string
	`SearchRange`	A range in the input to be substituted

Function replace_first_copy

boost::algorithm::replace_first_copy — Replace first algorithm.

Synopsis

```
// In header: <boost/algorithm/string/replace.hpp>


template<typename OutputIteratorT, typename Range1T, typename Range2T,
         typename Range3T>
  OutputIteratorT
  replace_first_copy(OutputIteratorT Output, const Range1T & Input,
                     const Range2T & Search, const Range3T & Format);
template<typename SequenceT, typename Range1T, typename Range2T>
  SequenceT replace_first_copy(const SequenceT & Input,
                               const Range1T & Search,
                               const Range2T & Format);
```

Description

Replace the first match of the search substring in the input with the format string. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters:	`Format`	A substitute string
	`Input`	An input string
	`Output`	An output iterator to which the result will be copied
	`Search`	A substring to be searched for
Returns:	An output iterator pointing just after the last inserted character or a modified copy of the input	
Notes:	The second variant of this function provides the strong exception-safety guarantee	

Function template replace_first

boost::algorithm::replace_first — Replace first algorithm.

Synopsis

```
// In header: <boost/algorithm/string/replace.hpp>


template<typename SequenceT, typename Range1T, typename Range2T>
  void replace_first(SequenceT & Input, const Range1T & Search,
                     const Range2T & Format);
```

Description

replace the first match of the search substring in the input with the format string. The input sequence is modified in-place.

Parameters:		
	`Format`	A substitute string
	`Input`	An input string
	`Search`	A substring to be searched for

Function ireplace_first_copy

boost::algorithm::ireplace_first_copy — Replace first algorithm (case insensitive).

Synopsis

```
// In header: <boost/algorithm/string/replace.hpp>


template<typename OutputIteratorT, typename Range1T, typename Range2T,
         typename Range3T>
  OutputIteratorT
  ireplace_first_copy(OutputIteratorT Output, const Range1T & Input,
                      const Range2T & Search, const Range3T & Format,
                      const std::locale & Loc = std::locale());
template<typename SequenceT, typename Range2T, typename Range1T>
  SequenceT ireplace_first_copy(const SequenceT & Input,
                                const Range2T & Search,
                                const Range1T & Format,
                                const std::locale & Loc = std::locale());
```

Description

Replace the first match of the search substring in the input with the format string. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator. Searching is case insensitive.

Parameters:	`Format`	A substitute string
	`Input`	An input string
	`Loc`	A locale used for case insensitive comparison
	`Output`	An output iterator to which the result will be copied
	`Search`	A substring to be searched for
Returns:	An output iterator pointing just after the last inserted character or a modified copy of the input	
Notes:	The second variant of this function provides the strong exception-safety guarantee	

Function template ireplace_first

boost::algorithm::ireplace_first — Replace first algorithm (case insensitive).

Synopsis

```
// In header: <boost/algorithm/string/replace.hpp>


template<typename SequenceT, typename Range1T, typename Range2T>
  void ireplace_first(SequenceT & Input, const Range1T & Search,
                      const Range2T & Format,
                      const std::locale & Loc = std::locale());
```

Description

Replace the first match of the search substring in the input with the format string. Input sequence is modified in-place. Searching is case insensitive.

Parameters:		
	`Format`	A substitute string
	`Input`	An input string
	`Loc`	A locale used for case insensitive comparison
	`Search`	A substring to be searched for

Function replace_last_copy

boost::algorithm::replace_last_copy — Replace last algorithm.

Synopsis

```
// In header: <boost/algorithm/string/replace.hpp>


template<typename OutputIteratorT, typename Range1T, typename Range2T,
         typename Range3T>
  OutputIteratorT
  replace_last_copy(OutputIteratorT Output, const Range1T & Input,
                    const Range2T & Search, const Range3T & Format);
template<typename SequenceT, typename Range1T, typename Range2T>
  SequenceT replace_last_copy(const SequenceT & Input, const Range1T & Search,
                              const Range2T & Format);
```

Description

Replace the last match of the search string in the input with the format string. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters:	`Format`	A substitute string
	`Input`	An input string
	`Output`	An output iterator to which the result will be copied
	`Search`	A substring to be searched for
Returns:	An output iterator pointing just after the last inserted character or a modified copy of the input	
Notes:	The second variant of this function provides the strong exception-safety guarantee	

Function template replace_last

boost::algorithm::replace_last — Replace last algorithm.

Synopsis

```
// In header: <boost/algorithm/string/replace.hpp>


template<typename SequenceT, typename Range1T, typename Range2T>
  void replace_last(SequenceT & Input, const Range1T & Search,
                    const Range2T & Format);
```

Description

Replace the last match of the search string in the input with the format string. Input sequence is modified in-place.

Parameters:		
	`Format`	A substitute string
	`Input`	An input string
	`Search`	A substring to be searched for

Function ireplace_last_copy

boost::algorithm::ireplace_last_copy — Replace last algorithm (case insensitive).

Synopsis

```
// In header: <boost/algorithm/string/replace.hpp>


template<typename OutputIteratorT, typename Range1T, typename Range2T,
         typename Range3T>
  OutputIteratorT
  ireplace_last_copy(OutputIteratorT Output, const Range1T & Input,
                     const Range2T & Search, const Range3T & Format,
                     const std::locale & Loc = std::locale());
template<typename SequenceT, typename Range1T, typename Range2T>
  SequenceT ireplace_last_copy(const SequenceT & Input,
                               const Range1T & Search,
                               const Range2T & Format,
                               const std::locale & Loc = std::locale());
```

Description

Replace the last match of the search string in the input with the format string. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator. Searching is case insensitive.

Parameters:	`Format`	A substitute string
	`Input`	An input string
	`Loc`	A locale used for case insensitive comparison
	`Output`	An output iterator to which the result will be copied
	`Search`	A substring to be searched for
Returns:	An output iterator pointing just after the last inserted character or a modified copy of the input	
Notes:	The second variant of this function provides the strong exception-safety guarantee	

Function template ireplace_last

boost::algorithm::ireplace_last — Replace last algorithm (case insensitive).

Synopsis

```
// In header: <boost/algorithm/string/replace.hpp>


template<typename SequenceT, typename Range1T, typename Range2T>
  void ireplace_last(SequenceT & Input, const Range1T & Search,
                     const Range2T & Format,
                     const std::locale & Loc = std::locale());
```

Description

Replace the last match of the search string in the input with the format string.The input sequence is modified in-place. Searching is case insensitive.

Parameters:	`Format`	A substitute string
	`Input`	An input string
	`Loc`	A locale used for case insensitive comparison
	`Search`	A substring to be searched for
Returns:	A reference to the modified input	

Function replace_nth_copy

boost::algorithm::replace_nth_copy — Replace nth algorithm.

Synopsis

```
// In header: <boost/algorithm/string/replace.hpp>


template<typename OutputIteratorT, typename Range1T, typename Range2T,
         typename Range3T>
  OutputIteratorT
  replace_nth_copy(OutputIteratorT Output, const Range1T & Input,
                   const Range2T & Search, int Nth, const Range3T & Format);
template<typename SequenceT, typename Range1T, typename Range2T>
  SequenceT replace_nth_copy(const SequenceT & Input, const Range1T & Search,
                             int Nth, const Range2T & Format);
```

Description

Replace an Nth (zero-indexed) match of the search string in the input with the format string. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters:	`Format`	A substitute string
	`Input`	An input string
	`Nth`	An index of the match to be replaced. The index is 0-based. For negative N, matches are counted from the end of string.
	`Output`	An output iterator to which the result will be copied
	`Search`	A substring to be searched for
Returns:	An output iterator pointing just after the last inserted character or a modified copy of the input	
Notes:	The second variant of this function provides the strong exception-safety guarantee	

Function template replace_nth

boost::algorithm::replace_nth — Replace nth algorithm.

Synopsis

```
// In header: <boost/algorithm/string/replace.hpp>


template<typename SequenceT, typename Range1T, typename Range2T>
  void replace_nth(SequenceT & Input, const Range1T & Search, int Nth,
                   const Range2T & Format);
```

Description

Replace an Nth (zero-indexed) match of the search string in the input with the format string. Input sequence is modified in-place.

Parameters:

`Format`	A substitute string
`Input`	An input string
`Nth`	An index of the match to be replaced. The index is 0-based. For negative N, matches are counted from the end of string.
`Search`	A substring to be searched for

Function ireplace_nth_copy

boost::algorithm::ireplace_nth_copy — Replace nth algorithm (case insensitive).

Synopsis

```
// In header: <boost/algorithm/string/replace.hpp>


template<typename OutputIteratorT, typename Range1T, typename Range2T,
         typename Range3T>
  OutputIteratorT
  ireplace_nth_copy(OutputIteratorT Output, const Range1T & Input,
                    const Range2T & Search, int Nth, const Range3T & Format,
                    const std::locale & Loc = std::locale());
template<typename SequenceT, typename Range1T, typename Range2T>
  SequenceT ireplace_nth_copy(const SequenceT & Input, const Range1T & Search,
                              int Nth, const Range2T & Format,
                              const std::locale & Loc = std::locale());
```

Description

Replace an Nth (zero-indexed) match of the search string in the input with the format string. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator. Searching is case insensitive.

Parameters:	`Format`	A substitute string
	`Input`	An input string
	`Loc`	A locale used for case insensitive comparison
	`Nth`	An index of the match to be replaced. The index is 0-based. For negative N, matches are counted from the end of string.
	`Output`	An output iterator to which the result will be copied
	`Search`	A substring to be searched for
Returns:	An output iterator pointing just after the last inserted character or a modified copy of the input	
Notes:	The second variant of this function provides the strong exception-safety guarantee	

Function template ireplace_nth

boost::algorithm::ireplace_nth — Replace nth algorithm (case insensitive).

Synopsis

```
// In header: <boost/algorithm/string/replace.hpp>


template<typename SequenceT, typename Range1T, typename Range2T>
  void ireplace_nth(SequenceT & Input, const Range1T & Search, int Nth,
                    const Range2T & Format,
                    const std::locale & Loc = std::locale());
```

Description

Replace an Nth (zero-indexed) match of the search string in the input with the format string. Input sequence is modified in-place. Searching is case insensitive.

Parameters:		
	`Format`	A substitute string
	`Input`	An input string
	`Loc`	A locale used for case insensitive comparison
	`Nth`	An index of the match to be replaced. The index is 0-based. For negative N, matches are counted from the end of string.
	`Search`	A substring to be searched for

Function replace_all_copy

boost::algorithm::replace_all_copy — Replace all algorithm.

Synopsis

```
// In header: <boost/algorithm/string/replace.hpp>


template<typename OutputIteratorT, typename Range1T, typename Range2T,
         typename Range3T>
  OutputIteratorT
  replace_all_copy(OutputIteratorT Output, const Range1T & Input,
                   const Range2T & Search, const Range3T & Format);
template<typename SequenceT, typename Range1T, typename Range2T>
  SequenceT replace_all_copy(const SequenceT & Input, const Range1T & Search,
                             const Range2T & Format);
```

Description

Replace all occurrences of the search string in the input with the format string. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters:	`Format`	A substitute string
	`Input`	An input string
	`Output`	An output iterator to which the result will be copied
	`Search`	A substring to be searched for
Returns:	An output iterator pointing just after the last inserted character or a modified copy of the input	
Notes:	The second variant of this function provides the strong exception-safety guarantee	

Function template replace_all

boost::algorithm::replace_all — Replace all algorithm.

Synopsis

```
// In header: <boost/algorithm/string/replace.hpp>


template<typename SequenceT, typename Range1T, typename Range2T>
  void replace_all(SequenceT & Input, const Range1T & Search,
                   const Range2T & Format);
```

Description

Replace all occurrences of the search string in the input with the format string. The input sequence is modified in-place.

Parameters:	`Format`	A substitute string
	`Input`	An input string
	`Search`	A substring to be searched for
Returns:	A reference to the modified input	

Function ireplace_all_copy

boost::algorithm::ireplace_all_copy — Replace all algorithm (case insensitive).

Synopsis

```
// In header: <boost/algorithm/string/replace.hpp>


template<typename OutputIteratorT, typename Range1T, typename Range2T,
         typename Range3T>
  OutputIteratorT
  ireplace_all_copy(OutputIteratorT Output, const Range1T & Input,
                    const Range2T & Search, const Range3T & Format,
                    const std::locale & Loc = std::locale());
template<typename SequenceT, typename Range1T, typename Range2T>
  SequenceT ireplace_all_copy(const SequenceT & Input, const Range1T & Search,
                              const Range2T & Format,
                              const std::locale & Loc = std::locale());
```

Description

Replace all occurrences of the search string in the input with the format string. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator. Searching is case insensitive.

Parameters:	`Format`	A substitute string
	`Input`	An input string
	`Loc`	A locale used for case insensitive comparison
	`Output`	An output iterator to which the result will be copied
	`Search`	A substring to be searched for
Returns:	An output iterator pointing just after the last inserted character or a modified copy of the input	
Notes:	The second variant of this function provides the strong exception-safety guarantee	

Function template ireplace_all

boost::algorithm::ireplace_all — Replace all algorithm (case insensitive).

Synopsis

```
// In header: <boost/algorithm/string/replace.hpp>


template<typename SequenceT, typename Range1T, typename Range2T>
  void ireplace_all(SequenceT & Input, const Range1T & Search,
                    const Range2T & Format,
                    const std::locale & Loc = std::locale());
```

Description

Replace all occurrences of the search string in the input with the format string.The input sequence is modified in-place. Searching is case insensitive.

Parameters:		
	`Format`	A substitute string
	`Input`	An input string
	`Loc`	A locale used for case insensitive comparison
	`Search`	A substring to be searched for

Function replace_head_copy

boost::algorithm::replace_head_copy — Replace head algorithm.

Synopsis

```
// In header: <boost/algorithm/string/replace.hpp>


template<typename OutputIteratorT, typename Range1T, typename Range2T>
  OutputIteratorT
  replace_head_copy(OutputIteratorT Output, const Range1T & Input, int N,
                    const Range2T & Format);
template<typename SequenceT, typename RangeT>
  SequenceT replace_head_copy(const SequenceT & Input, int N,
                              const RangeT & Format);
```

Description

Replace the head of the input with the given format string. The head is a prefix of a string of given size. If the sequence is shorter then required, whole string if considered to be the head. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters:	`Format`	A substitute string
	`Input`	An input string
	`N`	Length of the head. For N>=0, at most N characters are extracted. For N<0, size(Input)-|N| characters are extracted.
	`Output`	An output iterator to which the result will be copied
Returns:	An output iterator pointing just after the last inserted character or a modified copy of the input	
Notes:	The second variant of this function provides the strong exception-safety guarantee	

Function template replace_head

boost::algorithm::replace_head — Replace head algorithm.

Synopsis

```
// In header: <boost/algorithm/string/replace.hpp>


template<typename SequenceT, typename RangeT>
  void replace_head(SequenceT & Input, int N, const RangeT & Format);
```

Description

Replace the head of the input with the given format string. The head is a prefix of a string of given size. If the sequence is shorter then required, the whole string is considered to be the head. The input sequence is modified in-place.

Parameters:	`Format`	A substitute string
	`Input`	An input string
	`N`	Length of the head. For N>=0, at most N characters are extracted. For N<0, size(Input)-|N| characters are extracted.

Function replace_tail_copy

boost::algorithm::replace_tail_copy — Replace tail algorithm.

Synopsis

```
// In header: <boost/algorithm/string/replace.hpp>


template<typename OutputIteratorT, typename Range1T, typename Range2T>
  OutputIteratorT
  replace_tail_copy(OutputIteratorT Output, const Range1T & Input, int N,
                    const Range2T & Format);
template<typename SequenceT, typename RangeT>
  SequenceT replace_tail_copy(const SequenceT & Input, int N,
                              const RangeT & Format);
```

Description

Replace the tail of the input with the given format string. The tail is a suffix of a string of given size. If the sequence is shorter then required, whole string is considered to be the tail. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters:	`Format`	A substitute string
	`Input`	An input string
	`N`	Length of the tail. For N>=0, at most N characters are extracted. For N<0, size(Input)-|N| characters are extracted.
	`Output`	An output iterator to which the result will be copied
Returns:	An output iterator pointing just after the last inserted character or a modified copy of the input	
Notes:	The second variant of this function provides the strong exception-safety guarantee	

Function template replace_tail

boost::algorithm::replace_tail — Replace tail algorithm.

Synopsis

```
// In header: <boost/algorithm/string/replace.hpp>


template<typename SequenceT, typename RangeT>
  void replace_tail(SequenceT & Input, int N, const RangeT & Format);
```

Description

Replace the tail of the input with the given format sequence. The tail is a suffix of a string of given size. If the sequence is shorter then required, the whole string is considered to be the tail. The input sequence is modified in-place.

Parameters:		
	`Format`	A substitute string
	`Input`	An input string
	`N`	Length of the tail. For N>=0, at most N characters are extracted. For N<0, size(Input)-|N| characters are extracted.

Header <boost/algorithm/string/sequence_traits.hpp>

Traits defined in this header are used by various algorithms to achieve better performance for specific containers. Traits provide fail-safe defaults. If a container supports some of these features, it is possible to specialize the specific trait for this container. For lacking compilers, it is possible of define an override for a specific tester function.

Due to a language restriction, it is not currently possible to define specializations for stl containers without including the corresponding header. To decrease the overhead needed by this inclusion, user can selectively include a specialization header for a specific container. They are located in boost/algorithm/string/stl directory. Alternatively she can include boost/algorithm/string/std_collection_traits.hpp header which contains specializations for all stl containers.

```
namespace boost {
  namespace algorithm {
    template<typename T> class has_native_replace;
    template<typename T> class has_stable_iterators;
    template<typename T> class has_const_time_insert;
    template<typename T> class has_const_time_erase;
  }
}
```

Class template has_native_replace

boost::algorithm::has_native_replace — Native replace trait.

Synopsis

```
// In header: <boost/algorithm/string/sequence_traits.hpp>

template<typename T>
class has_native_replace {
public:
  // types
  typedef mpl::bool_< has_native_replace< T >::value > type;
  static const bool value;
};
```

Description

This trait specifies that the sequence has `std::string` like replace method

Class template has_stable_iterators

boost::algorithm::has_stable_iterators — Stable iterators trait.

Synopsis

```
// In header: <boost/algorithm/string/sequence_traits.hpp>

template<typename T>
class has_stable_iterators {
public:
  // types
  typedef mpl::bool_< has_stable_iterators< T >::value > type;
  static const bool value;
};
```

Description

This trait specifies that the sequence has stable iterators. It means that operations like insert/erase/replace do not invalidate iterators.

Class template has_const_time_insert

boost::algorithm::has_const_time_insert — Const time insert trait.

Synopsis

```
// In header: <boost/algorithm/string/sequence_traits.hpp>

template<typename T>
class has_const_time_insert {
public:
  // types
  typedef mpl::bool_< has_const_time_insert< T >::value > type;
  static const bool value;
};
```

Description

This trait specifies that the sequence's insert method has constant time complexity.

Class template has_const_time_erase

boost::algorithm::has_const_time_erase — Const time erase trait.

Synopsis

```
// In header: <boost/algorithm/string/sequence_traits.hpp>

template<typename T>
class has_const_time_erase {
public:
  // types
  typedef mpl::bool_< has_const_time_erase< T >::value > type;
  static const bool value;
};
```

Description

This trait specifies that the sequence's erase method has constant time complexity.

Header <boost/algorithm/string/split.hpp>

Defines basic split algorithms. Split algorithms can be used to divide a string into several parts according to given criteria.

Each part is copied and added as a new element to the output container. Thus the result container must be able to hold copies of the matches (in a compatible structure like std::string) or a reference to it (e.g. using the iterator range class). Examples of such a container are `std::vector<std::string>` or `std::list<boost::iterator_range<std::string::iterator>>`

```
namespace boost {
  namespace algorithm {
    template<typename SequenceSequenceT, typename Range1T, typename Range2T>
      SequenceSequenceT &
      find_all(SequenceSequenceT &, Range1T &, const Range2T &);
    template<typename SequenceSequenceT, typename Range1T, typename Range2T>
      SequenceSequenceT &
      ifind_all(SequenceSequenceT &, Range1T &, const Range2T &,
                const std::locale & = std::locale());
    template<typename SequenceSequenceT, typename RangeT, typename PredicateT>
      SequenceSequenceT &
      split(SequenceSequenceT &, RangeT &, PredicateT,
            token_compress_mode_type = token_compress_off);
  }
}
```

Function template find_all

boost::algorithm::find_all — Find all algorithm.

Synopsis

```
// In header: <boost/algorithm/string/split.hpp>


template<typename SequenceSequenceT, typename Range1T, typename Range2T>
  SequenceSequenceT &
  find_all(SequenceSequenceT & Result, Range1T & Input,
           const Range2T & Search);
```

Description

This algorithm finds all occurrences of the search string in the input.

Each part is copied and added as a new element to the output container. Thus the result container must be able to hold copies of the matches (in a compatible structure like std::string) or a reference to it (e.g. using the iterator range class). Examples of such a container are `std::vector<std::string>` or `std::list<boost::iterator_range<std::string::iterator>>`

Parameters:	`Input`	A container which will be searched.
	`Result`	A container that can hold copies of references to the substrings
	`Search`	A substring to be searched for.
Returns:	A reference the result	
Notes:	Prior content of the result will be overwritten.	
	This function provides the strong exception-safety guarantee	

Function template ifind_all

boost::algorithm::ifind_all — Find all algorithm (case insensitive).

Synopsis

```
// In header: <boost/algorithm/string/split.hpp>


template<typename SequenceSequenceT, typename Range1T, typename Range2T>
  SequenceSequenceT &
  ifind_all(SequenceSequenceT & Result, Range1T & Input,
            const Range2T & Search, const std::locale & Loc = std::locale());
```

Description

This algorithm finds all occurrences of the search string in the input. Each part is copied and added as a new element to the output container. Thus the result container must be able to hold copies of the matches (in a compatible structure like std::string) or a reference to it (e.g. using the iterator range class). Examples of such a container are `std::vector<std::string>` or `std::list<boost::iterator_range<std::string::iterator>>`

Searching is case insensitive.

Parameters:	`Input`	A container which will be searched.
	`Loc`	A locale used for case insensitive comparison
	`Result`	A container that can hold copies of references to the substrings
	`Search`	A substring to be searched for.
Returns:	A reference the result	
Notes:	Prior content of the result will be overwritten.	
	This function provides the strong exception-safety guarantee	

Function template split

boost::algorithm::split — Split algorithm.

Synopsis

```
// In header: <boost/algorithm/string/split.hpp>


template<typename SequenceSequenceT, typename RangeT, typename PredicateT>
  SequenceSequenceT &
  split(SequenceSequenceT & Result, RangeT & Input, PredicateT Pred,
        token_compress_mode_type eCompress = token_compress_off);
```

Description

Tokenize expression. This function is equivalent to C strtok. Input sequence is split into tokens, separated by separators. Separators are given by means of the predicate.

Each part is copied and added as a new element to the output container. Thus the result container must be able to hold copies of the matches (in a compatible structure like std::string) or a reference to it (e.g. using the iterator range class). Examples of such a container are `std::vector<std::string>` or `std::list<boost::iterator_range<std::string::iterator>>`

Parameters:	`Input`	A container which will be searched.
	`Pred`	A predicate to identify separators. This predicate is supposed to return true if a given element is a separator.
	`Result`	A container that can hold copies of references to the substrings
	`eCompress`	If eCompress argument is set to token_compress_on, adjacent separators are merged together. Otherwise, every two separators delimit a token.
Returns:	A reference the result	
Notes:	Prior content of the result will be overwritten.	
	This function provides the strong exception-safety guarantee	

Header <boost/algorithm/string/std_containers_traits.hpp>

This file includes sequence traits for stl containers.

Header <boost/algorithm/string/trim.hpp>

Defines trim algorithms. Trim algorithms are used to remove trailing and leading spaces from a sequence (string). Space is recognized using given locales.

Parametric (`_if`) variants use a predicate (functor) to select which characters are to be trimmed.. Functions take a selection predicate as a parameter, which is used to determine whether a character is a space. Common predicates are provided in classification.hpp header.

```
namespace boost {
  namespace algorithm {
    template<typename OutputIteratorT, typename RangeT, typename PredicateT>
      OutputIteratorT
      trim_left_copy_if(OutputIteratorT, const RangeT &, PredicateT);
    template<typename SequenceT, typename PredicateT>
      SequenceT trim_left_copy_if(const SequenceT &, PredicateT);
    template<typename SequenceT>
      SequenceT trim_left_copy(const SequenceT &,
                               const std::locale & = std::locale());
    template<typename SequenceT, typename PredicateT>
      void trim_left_if(SequenceT &, PredicateT);
    template<typename SequenceT>
      void trim_left(SequenceT &, const std::locale & = std::locale());
    template<typename OutputIteratorT, typename RangeT, typename PredicateT>
      OutputIteratorT
      trim_right_copy_if(OutputIteratorT, const RangeT &, PredicateT);
    template<typename SequenceT, typename PredicateT>
      SequenceT trim_right_copy_if(const SequenceT &, PredicateT);
    template<typename SequenceT>
      SequenceT trim_right_copy(const SequenceT &,
                                const std::locale & = std::locale());
    template<typename SequenceT, typename PredicateT>
      void trim_right_if(SequenceT &, PredicateT);
    template<typename SequenceT>
      void trim_right(SequenceT &, const std::locale & = std::locale());
    template<typename OutputIteratorT, typename RangeT, typename PredicateT>
      OutputIteratorT
      trim_copy_if(OutputIteratorT, const RangeT &, PredicateT);
    template<typename SequenceT, typename PredicateT>
      SequenceT trim_copy_if(const SequenceT &, PredicateT);
    template<typename SequenceT>
      SequenceT trim_copy(const SequenceT &,
                          const std::locale & = std::locale());
    template<typename SequenceT, typename PredicateT>
      void trim_if(SequenceT &, PredicateT);
    template<typename SequenceT>
      void trim(SequenceT &, const std::locale & = std::locale());
  }
}
```

Function trim_left_copy_if

boost::algorithm::trim_left_copy_if — Left trim - parametric.

Synopsis

```
// In header: <boost/algorithm/string/trim.hpp>


template<typename OutputIteratorT, typename RangeT, typename PredicateT>
  OutputIteratorT
  trim_left_copy_if(OutputIteratorT Output, const RangeT & Input,
                    PredicateT IsSpace);
template<typename SequenceT, typename PredicateT>
  SequenceT trim_left_copy_if(const SequenceT & Input, PredicateT IsSpace);
```

Description

Remove all leading spaces from the input. The supplied predicate is used to determine which characters are considered spaces. The result is a trimmed copy of the input. It is returned as a sequence or copied to the output iterator

Parameters:	`Input`	An input range
	`IsSpace`	An unary predicate identifying spaces
	`Output`	An output iterator to which the result will be copied
Returns:	An output iterator pointing just after the last inserted character or a copy of the input	
Notes:	The second variant of this function provides the strong exception-safety guarantee	

Function template trim_left_copy

boost::algorithm::trim_left_copy — Left trim - parametric.

Synopsis

```
// In header: <boost/algorithm/string/trim.hpp>


template<typename SequenceT>
  SequenceT trim_left_copy(const SequenceT & Input,
                           const std::locale & Loc = std::locale());
```

Description

Remove all leading spaces from the input. The result is a trimmed copy of the input.

Parameters:	`Input`	An input sequence
	`Loc`	a locale used for 'space' classification
Returns:	A trimmed copy of the input	
Notes:	This function provides the strong exception-safety guarantee	

Function template trim_left_if

boost::algorithm::trim_left_if — Left trim.

Synopsis

```
// In header: <boost/algorithm/string/trim.hpp>


template<typename SequenceT, typename PredicateT>
  void trim_left_if(SequenceT & Input, PredicateT IsSpace);
```

Description

Remove all leading spaces from the input. The supplied predicate is used to determine which characters are considered spaces. The input sequence is modified in-place.

Parameters:	`Input`	An input sequence
	`IsSpace`	An unary predicate identifying spaces

Function template trim_left

boost::algorithm::trim_left — Left trim.

Synopsis

```
// In header: <boost/algorithm/string/trim.hpp>


template<typename SequenceT>
  void trim_left(SequenceT & Input, const std::locale & Loc = std::locale());
```

Description

Remove all leading spaces from the input. The Input sequence is modified in-place.

Parameters:	`Input`	An input sequence
	`Loc`	A locale used for 'space' classification

Function trim_right_copy_if

boost::algorithm::trim_right_copy_if — Right trim - parametric.

Synopsis

```
// In header: <boost/algorithm/string/trim.hpp>


template<typename OutputIteratorT, typename RangeT, typename PredicateT>
  OutputIteratorT
  trim_right_copy_if(OutputIteratorT Output, const RangeT & Input,
                     PredicateT IsSpace);
template<typename SequenceT, typename PredicateT>
  SequenceT trim_right_copy_if(const SequenceT & Input, PredicateT IsSpace);
```

Description

Remove all trailing spaces from the input. The supplied predicate is used to determine which characters are considered spaces. The result is a trimmed copy of the input. It is returned as a sequence or copied to the output iterator

Parameters:	`Input`	An input range
	`IsSpace`	An unary predicate identifying spaces
	`Output`	An output iterator to which the result will be copied
Returns:	An output iterator pointing just after the last inserted character or a copy of the input	
Notes:	The second variant of this function provides the strong exception-safety guarantee	

Function template trim_right_copy

boost::algorithm::trim_right_copy — Right trim.

Synopsis

```
// In header: <boost/algorithm/string/trim.hpp>


template<typename SequenceT>
  SequenceT trim_right_copy(const SequenceT & Input,
                            const std::locale & Loc = std::locale());
```

Description

Remove all trailing spaces from the input. The result is a trimmed copy of the input

Parameters:	`Input`	An input sequence
	`Loc`	A locale used for 'space' classification
Returns:	A trimmed copy of the input	
Notes:	This function provides the strong exception-safety guarantee	

Function template trim_right_if

boost::algorithm::trim_right_if — Right trim - parametric.

Synopsis

```
// In header: <boost/algorithm/string/trim.hpp>


template<typename SequenceT, typename PredicateT>
  void trim_right_if(SequenceT & Input, PredicateT IsSpace);
```

Description

Remove all trailing spaces from the input. The supplied predicate is used to determine which characters are considered spaces. The input sequence is modified in-place.

Parameters:	`Input`	An input sequence
	`IsSpace`	An unary predicate identifying spaces

Function template trim_right

boost::algorithm::trim_right — Right trim.

Synopsis

```
// In header: <boost/algorithm/string/trim.hpp>


template<typename SequenceT>
  void trim_right(SequenceT & Input, const std::locale & Loc = std::locale());
```

Description

Remove all trailing spaces from the input. The input sequence is modified in-place.

Parameters:	`Input`	An input sequence
	`Loc`	A locale used for 'space' classification

Function trim_copy_if

boost::algorithm::trim_copy_if — Trim - parametric.

Synopsis

```
// In header: <boost/algorithm/string/trim.hpp>


template<typename OutputIteratorT, typename RangeT, typename PredicateT>
  OutputIteratorT
  trim_copy_if(OutputIteratorT Output, const RangeT & Input,
               PredicateT IsSpace);
template<typename SequenceT, typename PredicateT>
  SequenceT trim_copy_if(const SequenceT & Input, PredicateT IsSpace);
```

Description

Remove all trailing and leading spaces from the input. The supplied predicate is used to determine which characters are considered spaces. The result is a trimmed copy of the input. It is returned as a sequence or copied to the output iterator

Parameters:	`Input`	An input range
	`IsSpace`	An unary predicate identifying spaces
	`Output`	An output iterator to which the result will be copied
Returns:	An output iterator pointing just after the last inserted character or a copy of the input	
Notes:	The second variant of this function provides the strong exception-safety guarantee	

Function template trim_copy

boost::algorithm::trim_copy — Trim.

Synopsis

```
// In header: <boost/algorithm/string/trim.hpp>


template<typename SequenceT>
  SequenceT trim_copy(const SequenceT & Input,
                      const std::locale & Loc = std::locale());
```

Description

Remove all leading and trailing spaces from the input. The result is a trimmed copy of the input

Parameters:	`Input`	An input sequence
	`Loc`	A locale used for 'space' classification
Returns:	A trimmed copy of the input	
Notes:	This function provides the strong exception-safety guarantee	

Function template trim_if

boost::algorithm::trim_if — Trim.

Synopsis

```
// In header: <boost/algorithm/string/trim.hpp>


template<typename SequenceT, typename PredicateT>
  void trim_if(SequenceT & Input, PredicateT IsSpace);
```

Description

Remove all leading and trailing spaces from the input. The supplied predicate is used to determine which characters are considered spaces. The input sequence is modified in-place.

Parameters:	`Input`	An input sequence
	`IsSpace`	An unary predicate identifying spaces

Function template trim

boost::algorithm::trim — Trim.

Synopsis

```
// In header: <boost/algorithm/string/trim.hpp>


template<typename SequenceT>
  void trim(SequenceT & Input, const std::locale & Loc = std::locale());
```

Description

Remove all leading and trailing spaces from the input. The input sequence is modified in-place.

Parameters:		
	`Input`	An input sequence
	`Loc`	A locale used for 'space' classification

Header <boost/algorithm/string_regex.hpp>

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

Rationale

Locales

Locales have a very close relation to string processing. They contain information about the character sets and are used, for example, to change the case of characters and to classify the characters.

C++ allows to work with multiple different instances of locales at once. If an algorithm manipulates some data in a way that requires the usage of locales, there must be a way to specify them. However, one instance of locales is sufficient for most of the applications, and for a user it could be very tedious to specify which locales to use at every place where it is needed.

Fortunately, the C++ standard allows to specify the *global* locales (using static member function `std:locale::global()`). When instantiating an `std::locale` class without explicit information, the instance will be initialized with the *global* locale. This implies, that if an algorithm needs a locale, it should have an `std::locale` parameter defaulting to `std::locale()`. If a user needs to specify locales explicitly, she can do so. Otherwise the *global* locales are used.

Regular Expressions

Regular expressions are an essential part of text processing. For this reason, the library also provides regex variants of some algorithms. The library does not attempt to replace Boost.Regex; it merely wraps its functionality in a new interface. As a part of this library, regex algorithms integrate smoothly with other components, which brings additional value.

Environment

Build

The whole library is provided in headers. Regex variants of some algorithms, however, are dependent on the Boost.Regex library. All such algorithms are separated in `boost/algorithm/string_regex.hpp`. If this header is used, the application must be linked with the Boost.Regex library.

Examples

Examples showing the basic usage of the library can be found in the libs/algorithm/string/example directory. There is a separate file for the each part of the library. Please follow the boost build guidelines to build examples using the bjam. To successfully build regex examples the Boost.Regex library is required.

Tests

A full set of test cases for the library is located in the libs/algorithm/string/test directory. The test cases can be executed using the boost build system. For the tests of regular expression variants of algorithms, the Boost.Regex library is required.

Portability

The library has been successfully compiled and tested with the following compilers:

- Microsoft Visual C++ 7.0
- Microsoft Visual C++ 7.1
- GCC 3.2
- GCC 3.3.1

See Boost regression tables for additional info for a particular compiler.

There are known limitation on platforms not supporting partial template specialization. Library depends on correctly implemented `std::iterator_traits` class. If a standard library provided with compiler is broken, the String Algorithm Library cannot function properly. Usually it implies that primitive pointer iterators are not working with the library functions.

Credits

Acknowledgments

The author would like to thank everybody who gave suggestions and comments. Especially valuable were the contributions of Thorsten Ottosen, Jeff Garland and the other boost members who participated in the review process, namely David Abrahams, Daniel Frey, Beman Dawes, John Maddock, David B.Held, Pavel Vozenilek and many other.

Additional thanks go to Stefan Slapeta and Toon Knapen, who have been very resourceful in solving various portability issues.