Range 2.0

Thorsten Ottosen

Neil Groves

Table of Contents

Boost.Range is a collection of concepts and utilities, range-based algorithms, as well as range adaptors that allow for efficient and expressive code.

Using Boost.Range inplace of the standard library alternatives results in more readable code and in many cases greater efficiency.

Introduction

Generic algorithms have so far been specified in terms of two or more iterators. Two iterators would together form a range of values that the algorithm could work on. This leads to a very general interface, but also to a somewhat clumsy use of the algorithms with redundant specification of container names. Therefore we would like to raise the abstraction level for algorithms so they specify their interface in terms of Ranges as much as possible.

The most common form of ranges used throughout the C++ community are standard library containers. When writing algorithms however, one often finds it desirable for the algorithm to accept other types that offer enough functionality to satisfy the needs of the generic code **if a suitable layer of indirection is applied** . For example, raw arrays are often suitable for use with generic code that works with containers, provided a suitable adapter is used. Likewise, null terminated strings can be treated as containers of characters, if suitably adapted.

This library therefore provides the means to adapt standard-like containers, null terminated strings, `std::pairs` of iterators, and raw arrays (and more), such that the same generic code can work with them all. The basic idea is to add another layer of indirection using metafunctions and free-standing functions so syntactic and/or semantic differences can be removed.

The main advantages are

- simpler implementation and specification of generic range algorithms
- more flexible, compact and maintainable client code
- safe use of built-in arrays (for legacy code; why else would you use built-in arrays?)

Example - Iterate over the values in a map

```
using namespace boost;
using namespace boost::adaptors;
for_each( my_map | map_values, fn );
```

Example - Iterate over the keys in a map

```
using namespace boost;
using namespace boost::adaptors;
for_each( my_map | map_keys, fn );
```

Example - Push the even values from a map in reverse order into the container `target`

```
using namespace boost;
using namespace boost::adaptors;
// Assume that is_even is a predicate that has been implemented elsewhere...
push_back(target, my_map | map_values | filtered(is_even()) | reversed);
```

Range Concepts

Overview

A Range is a **concept** similar to the STL Container concept. A Range provides iterators for accessing a half-open range `[first,one_past_last)` of elements and provides information about the number of elements in the Range. However, a Range has fewer requirements than a Container.

The motivation for the Range concept is that there are many useful Container-like types that do not meet the full requirements of Container, and many algorithms that can be written with this reduced set of requirements. In particular, a Range does not necessarily

- own the elements that can be accessed through it,
- have copy semantics,

Because of the second requirement, a Range object must be passed by (const or non-const) reference in generic code.

The operations that can be performed on a Range is dependent on the traversal category of the underlying iterator type. Therefore the range concepts are named to reflect which traversal category its iterators support. See also terminology and style guidelines. for more information about naming of ranges.

The concepts described below specifies associated types as metafunctions and all functions as free-standing functions to allow for a layer of indirection.

Single Pass Range

Notation

`X`	A type that is a model of Single Pass Range.
`a`	Object of type X.

Description

A range `X` where `boost::range_iterator<X>::type` is a model of Single Pass Iterator.

Associated types

Iterator type	`boost::range_iterator<X>::type`	The type of iterator used to iterate through a Range's elements. The iterator's value type is expected to be the Range's value type. A conversion from the iterator type to the `const` iterator type must exist.
Const iterator type	`boost::range_iterator<const X>::type`	A type of iterator that may be used to examine, but not to modify, a Range's elements.

Valid expressions

The following expressions must be valid.

Name	Expression	Return type
Beginning of range	`boost::begin(a)`	`boost::range_iterator<X>::type` if a is mutable, `boost::range_iterator<const X>::type` otherwise
End of range	`boost::end(a)`	`boost::range_iterator<X>::type` if a is mutable, `boost::range_iterator<const X>::type` otherwise

Expression semantics

Expression	Semantics	Postcondition
`boost::begin(a)`	Returns an iterator pointing to the first element in the Range.	`boost::begin(a)` is either dereferenceable or past-the-end. It is past-the-end if and only if `boost::distance(a) == 0`.
`boost::end(a)`	Returns an iterator pointing one past the last element in the Range.	`boost::end(a)` is past-the-end.

Complexity guarantees

`boost::end(a)` is at most amortized linear time, `boost::begin(a)` is amortized constant time. For most practical purposes, one can expect both to be amortized constant time.

Invariants

Valid range	For any Range `a`, `[boost::begin(a),boost::end(a))` is a valid range, that is, `boost::end(a)` is reachable from `boost::begin(a)` in a finite number of increments.
Completeness	An algorithm that iterates through the range `[boost::begin(a),boost::end(a))` will pass through every element of `a`.

See also

Extending the library for UDTs

Implementation of metafunctions

Implementation of functions

Container

Forward Range

Notation

`X`	A type that is a model of Forward Range.
`a`	Object of type X.

Description

A range `X` where `boost::range_iterator<X>::type` is a model of Forward Traversal Iterator.

Refinement of

Single Pass Range

Associated types

Distance type	`boost::range_difference<X>::type`	A signed integral type used to represent the distance between two of the Range's iterators. This type must be the same as the iterator's distance type.
Size type	`boost::range_size<X>::type`	An unsigned integral type that can represent any nonnegative value of the Range's distance type.

See also

Implementation of metafunctions

Implementation of functions

Bidirectional Range

Notation

`X`	A type that is a model of Bidirectional Range.
`a`	Object of type X.

Description

This concept provides access to iterators that traverse in both directions (forward and reverse). The `boost::range_iterator<X>::type` iterator must meet all of the requirements of Bidirectional Traversal Iterator.

Refinement of

Forward Range

Associated types

Reverse Iterator type	`boost::range_reverse_iterator<X>::type`	The type of iterator used to iterate through a Range's elements in reverse order. The iterator's value type is expected to be the Range's value type. A conversion from the reverse iterator type to the const reverse iterator type must exist.
Const reverse iterator type	`boost::range_reverse_iterator<const X>::type`	A type of reverse iterator that may be used to examine, but not to modify, a Range's elements.

Valid expressions

Name	Expression	Return type	Semantics
Beginning of range	boost::rbegin(a)	boost::range_reverse_iterator<X>::type if a is mutable boost::range_reverse_iterator<const X>::type otherwise.	Equivalent to boost::range_reverse_iterator<X>::type(boost::end(a)).
End of range	boost::rend(a)	boost::range_reverse_iterator<X>::type if a is mutable, boost::range_reverse_iterator<const X>::type otherwise.	Equivalent to boost::range_reverse_iterator<X>::type(boost::begin(a)).

Complexity guarantees

boost::rbegin(a) has the same complexity as boost::end(a) and boost::rend(a) has the same complexity as boost::begin(a) from Forward Range.

Invariants

Valid reverse range	For any Bidirectional Range a, [boost::rbegin(a),boost::rend(a)) is a valid range, that is, boost::rend(a) is reachable from boost::rbegin(a) in a finite number of increments.
Completeness	An algorithm that iterates through the range [boost::rbegin(a),boost::rend(a)) will pass through every element of a.

See also

Implementation of metafunctions

Implementation of functions

Random Access Range

Description

A range X where boost::range_iterator<X>::type is a model of Random Access Traversal Iterator.

Refinement of

Bidirectional Range

Valid expressions

Name	Expression	Return type
Size of range	boost::size(a)	boost::range_size<X>::type

Expression semantics

Expression	Semantics	Postcondition
`boost::size(a)`	Returns the size of the Range, that is, its number of elements. Note `boost::size(a) == 0u` is equivalent to `boost::empty(a)`.	`boost::size(a) >= 0`

Complexity guarantees

`boost::size(a)` completes in amortized constant time.

Invariants

Range size	`boost::size(a)` is equal to the `boost::end(a) - boost::begin(a)`.

Concept Checking

Each of the range concepts has a corresponding concept checking class in the file `<boost/range/concepts.hpp>`. These classes may be used in conjunction with the Boost Concept Check library to ensure that the type of a template parameter is compatible with a range concept. If not, a meaningful compile time error is generated. Checks are provided for the range concepts related to iterator traversal categories. For example, the following line checks that the type `T` models the Forward Range concept.

```
BOOST_CONCEPT_ASSERT(( ForwardRangeConcept<T> ));
```

An additional concept check is required for the value access property of the range based on the range's iterator type. For example to check for a ForwardReadableRange, the following code is required.

```
BOOST_CONCEPT_ASSERT(( ForwardRangeConcept<T> ));
BOOST_CONCEPT_ASSERT(( ReadableIteratorConcept<typename range_iterator<T>::type> ));
```

The following range concept checking classes are provided.

- Class SinglePassRangeConcept checks for Single Pass Range
- Class ForwardRangeConcept checks for Forward Range
- Class BidirectionalRangeConcept checks for Bidirectional Range
- Class RandomAccessRangeConcept checks for Random Access Range

See also

Range Terminology and style guidelines

Iterator concepts

Boost Concept Check library

Reference

Overview

Three types of objects are currently supported by the library:

- standard-like containers
- `std::pair<iterator,iterator>`
- built-in arrays

Even though the behavior of the primary templates are exactly such that standard containers will be supported by default, the requirements are much lower than the standard container requirements. For example, the utility class `iterator_range` implements the minimal interface required to make the class a Forward Range.

Please also see Range concepts for more details.

Range concept implementation

Synopsis

```
namespace boost
{
    //
    // Single Pass Range metafunctions
    //

    template< class T >
    struct range_iterator;

    template< class T >
    struct range_value;

    template< class T >
    struct range_reference;

    template< class T >
    struct range_pointer;

    template< class T >
    struct range_category;

    //
    // Forward Range metafunctions
    //

    template< class T >
    struct range_difference;

    //
    // Bidirectional Range metafunctions
    //

    template< class T >
    struct range_reverse_iterator;

    //
    // Single Pass Range functions
    //

    template< class T >
    typename range_iterator<T>::type
    begin( T& r );

    template< class T >
    typename range_iterator<const T>::type
    begin( const T& r );

    template< class T >
    typename range_iterator<T>::type
    end( T& r );

    template< class T >
    typename range_iterator<const T>::type
    end( const T& r );

    template< class T >
    bool
    empty( const T& r );
```

```
//
// Forward Range functions
//

template< class T >
typename range_difference<T>::type
distance( const T& r );

//
// Bidirectional Range functions
//

template< class T >
typename range_reverse_iterator<T>::type
rbegin( T& r );

template< class T >
typename range_reverse_iterator<const T>::type
rbegin( const T& r );

template< class T >
typename range_reverse_iterator<T>::type
rend( T& r );

template< class T >
typename range_reverse_iterator<const T>::type
rend( const T& r );

//
// Random Access Range functions
//

template< class T >
typename range_difference<T>::type
size( const T& r );

//
// Special const Range functions
//

template< class T >
typename range_iterator<const T>::type
const_begin( const T& r );

template< class T >
typename range_iterator<const T>::type
const_end( const T& r );

template< class T >
typename range_reverse_iterator<const T>::type
const_rbegin( const T& r );

template< class T >
typename range_reverse_iterator<const T>::type
const_rend( const T& r );

//
// String utilities
//

template< class T >
iterator_range< ... see below ... >
```

```
    as_literal( T& r );

    template< class T >
    iterator_range< ... see below ... >
    as_literal( const T& r );

    template< class T >
    iterator_range< typename range_iterator<T>::type >
    as_array( T& r );

    template< class T >
    iterator_range< typename range_iterator<const T>::type >
    as_array( const T& r );

} // namespace 'boost'
```

Semantics

notation

Type	Object	Describes
`X`	`x`	any type
`T`	`t`	denotes behavior of the primary templates
`P`	`p`	denotes `std::pair<iterator,iterator>`
`A[sz]`	`a`	denotes an array of type `A` of size `sz`
`Char*`	`s`	denotes either `char*` or `wchar_t*`

Metafunctions

Expression	Return type	Complexity
range_iterator<X>::type	T::iterator P::first_type A*	compile time
range_iterator<const X>::type	T::const_iterator P::first_type const A*	compile time
range_value<X>::type	boost::iterator_value<range_iterator<X>::type>::type	compile time
range_reference<X>::type	boost::iterator_reference<range_iterator<X>::type>::type	compile time
range_pointer<X>::type	boost::iterator_pointer<range_iterator<X>::type>::type	compile time
range_category<X>::type	boost::iterator_category<range_iterator<X>::type>::type	compile time
range_difference<X>::type	boost::iterator_category<range_iterator<X>::type>::type	compile time
range_reverse_iterator<X>::type	boost::reverse_iterator<range_iterator<X>::type>	compile time
range_reverse_iterator<const X>::type	boost::reverse_iterator<range_iterator<const X>::type	compile time
has_range_iterator<X>::type	mpl::true_ if range_mutable_iterator<X>::type is a valid expression, mpl::false_ otherwise	compile time
has_range_const_iterator<X>::type	mpl::true_ if range_const_iterator<X>::type is a valid expression, mpl::false_ otherwise	compile time

Functions

Expression	**Return type**	**Returns**	**Complexity**
`begin(x)`	`range_iterator<X>::type`	`p.first` if `p` is of type `std::pair<T>` `a` if `a` is an array `range_begin(x)` if that expression would invoke a function found by ADL `t.begin()` otherwise	constant time
`end(x)`	`range_iterator<X>::type`	`p.second` if `p` is of type `std::pair<T>` `a + sz` if `a` is an array of size `sz` `range_end(x)` if that expression would invoke a function found by ADL `t.end()` otherwise	constant time
`empty(x)`	`bool`	`boost::begin(x) == boost::end(x)`	constant time
`distance(x)`	`range_difference<X>::type`	`std::distance(boost::begin(x),boost::end(x))`	-
`size(x)`	`range_size<X>::type`	`range_calculate_size(x)` which by default is `boost::end(x) - boost::begin(x)`. Users may supply alternative implementations by implementing `range_calculate_size(x)` so that it will be found via ADL	constant time
`rbegin(x)`	`range_reverse_iterator<X>::type`	`range_reverse_iterator<X>::type(boost::end(x))`	constant time
`rend(x)`	`range_reverse_iterator<X>::type`	`range_reverse_iterator<X>::type(boost::begin(x))`	constant time
`const_begin(x)`	`range_iterator<const X>::type`	`range_iterator<const X>::type(boost::begin(x))`	constant time
`const_end(x)`	`range_iterator<const X>::type`	`range_iterator<const X>::type(boost::end(x))`	constant time
`const_rbegin(x)`	`range_reverse_iterator<const X>::type`	`range_reverse_iterator<const X>::type(boost::rbegin(x))`	constant time
`const_rend(x)`	`range_reverse_iterator<const X>::type`	`range_reverse_iterator<const X>::type(boost::rend(x))`	constant time

Expression	Return type	Returns	Complexity
`as_literal(x)`	`iterator_range<U>` where `U` is `Char*` if `x` is a pointer to a string and `U` is `range_iterator<X>::type` otherwise	`[s, s + std::char_traits<X>::length(s))` if `s` is a `Char*` or an array of `Char` `[boost::begin(x),boost::end(x))` otherwise	linear time for pointers to a string or arrays of `Char`, constant time otherwise
`as_array(x)`	`iterator_range<X>`	`[boost::begin(x),boost::end(x))`	

The special `const_`-named functions are useful when you want to document clearly that your code is read-only.

`as_literal()` can be used ***internally*** in string algorithm libraries such that arrays of characters are handled correctly.

`as_array()` can be used with string algorithm libraries to make it clear that arrays of characters are handled like an array and not like a string.

Notice that the above functions should always be called with qualification (`boost::`) to prevent ***unintended*** Argument Dependent Lookup (ADL).

Range Adaptors

Introduction and motivation

A **Range Adaptor** is a class that wraps an existing Range to provide a new Range with different behaviour. Since the behaviour of Ranges is determined by their associated iterators, a Range Adaptor simply wraps the underlying iterators with new special iterators. In this example

```
#include <boost/range/adaptors.hpp>
#include <boost/range/algorithm.hpp>
#include <iostream>
#include <vector>

std::vector<int> vec;
boost::copy( vec | boost::adaptors::reversed,
             std::ostream_iterator<int>(std::cout) );
```

the iterators from `vec` are wrapped `reverse_iterator`s. The type of the underlying Range Adapter is not documented because you do not need to know it. All that is relevant is that the expression

```
vec | boost::adaptors::reversed
```

returns a Range Adaptor where the iterator type is now the iterator type of the range `vec` wrapped in `reverse_iterator`. The expression `boost::adaptors::reversed` is called an **Adaptor Generator**.

There are two ways of constructing a range adaptor. The first is by using `operator|()`. This is my preferred technique, however while discussing range adaptors with others it became clear that some users of the library strongly prefer a more familiar function syntax, so equivalent functions of the present tense form have been added as an alternative syntax. The equivalent to `rng | reversed` is `adaptors::reverse(rng)` for example.

Why do I prefer the `operator|` syntax? The answer is readability:

```
std::vector<int> vec;
boost::copy( boost::adaptors::reverse(vec),
             std::ostream_iterator<int>(std::cout) );
```

This might not look so bad, but when we apply several adaptors, it becomes much worse. Just compare

```
std::vector<int> vec;
boost::copy( boost::adaptors::unique( boost::adaptors::reverse( vec ) ),
             std::ostream_iterator<int>(std::cout) );
```

to

```
std::vector<int> vec;
boost::copy( vec | boost::adaptors::reversed
                 | boost::adaptors::uniqued,
             std::ostream_iterator<int>(std::cout) );
```

Furthermore, some of the adaptor generators take arguments themselves and these arguments are expressed with function call notation too. In those situations, you will really appreciate the succinctness of `operator|()`.

Composition of Adaptors

Range Adaptors are a powerful complement to Range algorithms. The reason is that adaptors are ***orthogonal*** to algorithms. For example, consider these Range algorithms:

- `boost::copy( rng, out )`
- `boost::count( rng, pred )`

What should we do if we only want to copy an element `a` if it satisfies some predicate, say `pred(a)`? And what if we only want to count the elements that satisfy the same predicate? The naive answer would be to use these algorithms:

- `boost::copy_if( rng, pred, out )`
- `boost::count_if( rng, pred )`

These algorithms are only defined to maintain a one to one relationship with the standard library algorithms. This approach of adding algorithm suffers a combinatorial explosion. Inevitably many algorithms are missing `_if` variants and there is redundant development overhead for each new algorithm. The Adaptor Generator is the design solution to this problem.

Range Adaptor alternative to copy_if algorithm

```
boost::copy_if( rng, pred, out );
```

can be expressed as

```
boost::copy( rng | boost::adaptors::filtered(pred), out );
```

Range Adaptor alternative to count_if algorithm

```
boost::count_if( rng, pred );
```

can be expressed as

```
boost::count( rng | boost::adaptors::filtered(pred), out );
```

What this means is that ***no*** algorithm with the `_if` suffix is needed. Furthermore, it turns out that algorithms with the `_copy` suffix are not needed either. Consider the somewhat misdesigned `replace_copy_if()` which may be used as

```
std::vector<int> vec;
boost::replace_copy_if( rng, std::back_inserter(vec), pred, new_value );
```

With adaptors and algorithms we can express this as

```
std::vector<int> vec;
boost::push_back(vec, rng | boost::adaptors::replaced_if(pred, new_value));
```

The latter code has several benefits:

1. it is more ***efficient*** because we avoid extra allocations as might happen with `std::back_inserter`

2. it is ***flexible*** as we can subsequently apply even more adaptors, for example:

```
boost::push_back(vec, rng | boost::adaptors::replaced_if(pred, new_value)
                          | boost::adaptors::reversed);
```

3. it is ***safer*** because there is no use of an unbounded output iterator.

In this manner, the ***composition*** of Range Adaptors has the following consequences:

1. we no longer need `_if`, `_copy`, `_copy_if` and `_n` variants of algorithms.

2. we can generate a multitude of new algorithms on the fly, for example, above we generated `reverse_replace_copy_if()`

In other words:

Range Adaptors are to algorithms what algorithms are to containers

General Requirements

In the description of generator expressions, the following notation is used:

- `fwdRng` is an expression of a type `R` that models `ForwardRange`
- `biRng` is an expression of a type `R` that models `BidirectionalRange`
- `rndRng` is an expression of a type `R` that models `RandomAccessRange`
- `pred` is an expression of a type that models `UnaryPredicate`
- `bi_pred` is an expression of a type that models `BinaryPredicate`
- `fun` is an expression of a type that models `UnaryFunction`
- `value`, `new_value` and `old_value` are objects convertible to `boost::range_value<R>::type`
- `n,m` are integer expressions convertible to `range_difference<R>::type`

Also note that `boost::range_value<R>::type` must be implicitly convertible to the type arguments to `pred`, `bi_pred` and `fun`.

Range Category in the following adaptor descriptions refers to the minimum range concept required by the range passed to the adaptor. The resultant range is a model of the same range concept as the input range unless specified otherwise.

Returned Range Category is the concept of the returned range. In some cases the returned range is of a lesser category than the range passed to the adaptor. For example, the `filtered` adaptor returns only a `ForwardRange` regardless of the input.

Furthermore, the following rules apply to any expression of the form

```
rng | boost::adaptors::adaptor_generator
```

1. Applying `operator|()` to a range `R` (always left argument) and a range adapter `RA` (always right argument) yields a new range type which may not conform to the same range concept as `R`.

2. The return-type of `operator|()` is otherwise unspecified.

3. `operator|()` is found by Argument Dependent Lookup (ADL) because a range adaptor is implemented in namespace `boost::adaptors`.

4. `operator|()` is used to add new behaviour ***lazily*** and never modifies its left argument.

5. All iterators extracted from the left argument are extracted using qualified calls to `boost::begin()` and `boost::end()`.

6. In addition to the `throw`-clauses below, `operator|()` may throw exceptions as a result of copying iterators. If such copying cannot throw an exception, then neither can the whole expression.

Reference

adjacent_filtered

Syntax	Code
Pipe	`rng	boost::adaptors::adjacent_filtered(bi_pred)`
Function	`boost::adaptors::adjacent_filter(rng, bi_pred)`

- **Precondition:** The `value_type` of the range is convertible to both argument types of `bi_pred`.
- **Postcondition:** For all adjacent elements `[x,y]` in the returned range, `bi_pred(x,y)` is `true`.
- **Throws:** Whatever the copy constructor of `bi_pred` might throw.
- **Range Category:** Single Pass Range
- **Return Type:** `boost::adjacent_filtered_range<typeof(rng)>`
- **Returned Range Category:** The minimum of the range category of `rng` and Forward Range

adjacent_filtered example

```
#include <boost/range/adaptor/adjacent_filtered.hpp>
#include <boost/range/algorithm/copy.hpp>
#include <boost/assign.hpp>
#include <iterator>
#include <functional>
#include <iostream>
#include <vector>

int main(int argc, const char* argv[])
{
    using namespace boost::assign;
    using namespace boost::adaptors;

    std::vector<int> input;
    input += 1,1,2,2,2,3,4,5,6;

    boost::copy(
        input | adjacent_filtered(std::not_equal_to<int>()),
        std::ostream_iterator<int>(std::cout, ","));

    return 0;
}
```

This would produce the output:

```
1,2,3,4,5,6,
```

copied

Syntax	Code
Pipe	`rng | boost::adaptors::copied(n, m)`
Function	`boost::adaptors::copy(rng, n, m)`

- **Precondition:** `0 <= n && n <= m && m < distance(rng)`
- **Returns:** A new `iterator_range` that holds the sliced range `[n,m)` of the original range.
- **Range Category:** Random Access Range
- **Returned Range Category:** Random Access Range

copied example

```
#include <boost/range/adaptor/copied.hpp>
#include <boost/range/algorithm/copy.hpp>
#include <boost/assign.hpp>
#include <iterator>
#include <iostream>
#include <vector>

int main(int argc, const char* argv[])
{
    using namespace boost::assign;
    using namespace boost::adaptors;

    std::vector<int> input;
    input += 1,2,3,4,5,6,7,8,9,10;

    boost::copy(
        input | copied(1, 5),
        std::ostream_iterator<int>(std::cout, ","));

    return 0;
}
```

This would produce the output:

```
2,3,4,5,
```

filtered

Syntax	Code
Pipe	`rng	boost::adaptors::filtered(pred)`
Function	`boost::adaptors::filter(rng, pred)`

- **Precondition:** The `value_type` of the range is convertible to the argument type of `pred`.
- **Postcondition:** For all adjacent elements `[x]` in the returned range, `pred(x)` is `true`.
- **Throws:** Whatever the copy constructor of `pred` might throw.
- **Range Category:** Forward Range
- **Range Return Type:** `boost::filtered_range<typeof(rng)>`
- **Returned Range Category:** The minimum of the range category of `rng` and Bidirectional Range

filtered example

```
#include <boost/range/adaptor/filtered.hpp>
#include <boost/range/algorithm/copy.hpp>
#include <boost/assign.hpp>
#include <iterator>
#include <iostream>
#include <vector>

struct is_even
{
    bool operator()( int x ) const { return x % 2 == 0; }
};

int main(int argc, const char* argv[])
{
    using namespace boost::assign;
    using namespace boost::adaptors;

    std::vector<int> input;
    input += 1,2,3,4,5,6,7,8,9;

    boost::copy(
        input | filtered(is_even()),
        std::ostream_iterator<int>(std::cout, ","));

    return 0;
}
```

This would produce the output:

```
2,4,6,8,
```

indexed

Syntax	Code
Pipe	`rng	boost::adaptors::indexed(start_index)`
Function	`boost::adaptors::index(rng, start_index)`

- **Returns:** A range adapted to return both the element and the associated index. The returned range consists of iterators that have in addition to the usual iterator member functions an `index()` member function that returns the appropriate index for the element in the sequence corresponding with the iterator.
- **Range Category:** Single Pass Range
- **Range Return Type:** `boost::indexed_range<typeof(rng)>`
- **Returned Range Category:** The range category of `rng`

indexed example

```
#include <boost/range/adaptor/indexed.hpp>
#include <boost/range/algorithm/copy.hpp>
#include <boost/assign.hpp>
#include <iterator>
#include <iostream>
#include <vector>

template<class Iterator>
void display_element_and_index(Iterator first, Iterator last)
{
    for (Iterator it = first; it != last; ++it)
    {
        std::cout << "Element = " << *it << " Index = " << it.index() << std::endl;
    }
}

template<class SinglePassRange>
void display_element_and_index(const SinglePassRange& rng)
{
    display_element_and_index(boost::begin(rng), boost::end(rng));
}

int main(int argc, const char* argv[])
{
    using namespace boost::assign;
    using namespace boost::adaptors;

    std::vector<int> input;
    input += 10,20,30,40,50,60,70,80,90;

    display_element_and_index( input | indexed(0) );

    return 0;
}
```

This would produce the output:

```
Element = 10 Index = 0
Element = 20 Index = 1
Element = 30 Index = 2
Element = 40 Index = 3
Element = 50 Index = 4
Element = 60 Index = 5
Element = 70 Index = 6
Element = 80 Index = 7
Element = 90 Index = 8
```

indirected

Syntax	Code
Pipe	`rng	boost::adaptors::indirected`
Function	`boost::adaptors::indirect(rng)`

- **Precondition:** The `value_type` of the range defines unary `operator*()`

- **Postcondition:** For all elements `x` in the returned range, `x` is the result of `*y` where `y` is the corresponding element in the original range.
- **Range Category:** Single Pass Range
- **Range Return Type:** `boost::indirected_range<typeof(rng)>`
- **Returned Range Category:** The range category of `rng`

indirected example

```
#include <boost/range/adaptor/indirected.hpp>
#include <boost/range/algorithm/copy.hpp>
#include <boost/shared_ptr.hpp>
#include <iterator>
#include <iostream>
#include <vector>

int main(int argc, const char* argv[])
{
    using namespace boost::adaptors;

    std::vector<boost::shared_ptr<int> > input;

    for (int i = 0; i < 10; ++i)
        input.push_back(boost::shared_ptr<int>(new int(i)));

    boost::copy(
        input | indirected,
        std::ostream_iterator<int>(std::cout, ","));

    return 0;
}
```

This would produce the output:

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

map_keys

Syntax	Code
Pipe	`rng | boost::adaptors::map_keys`
Function	`boost::adaptors::keys(rng)`

- **Precondition:** The `value_type` of the range is an instantiation of `std::pair`.
- **Postcondition:** For all elements `x` in the returned range, `x` is the result of `y.first` where `y` is the corresponding element in the original range.
- **Range Category:** Single Pass Range
- **Range Return Type:** `boost::select_first_range<typeof(rng)>`
- **Returned Range Category:** The range category of `rng`.

map_keys example

```
#include <boost/range/adaptor/map.hpp>
#include <boost/range/algorithm/copy.hpp>
#include <boost/assign.hpp>
#include <iterator>
#include <iostream>
#include <map>
#include <vector>

int main(int argc, const char* argv[])
{
    using namespace boost::assign;
    using namespace boost::adaptors;

    std::map<int,int> input;
    for (int i = 0; i < 10; ++i)
        input.insert(std::make_pair(i, i * 10));

    boost::copy(
        input | map_keys,
        std::ostream_iterator<int>(std::cout, ","));

    return 0;
}
```

This would produce the output:

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

map_values

Syntax	Code
Pipe	`rng | boost::adaptors::map_values`
Function	`boost::adaptors::values(rng)`

- **Precondition:** The `value_type` of the range is an instantiation of `std::pair`.
- **Postcondition:** For all elements `x` in the returned range, `x` is the result of `y.second` where `y` is the corresponding element in the original range.
- **Range Category:** Single Pass Range
- **Range Return Type:** for constant ranges, `boost::select_second_const<typeof(rng)>` otherwise `boost:select_second_mutable<typeof(rng)>`
- **Returned Range Category:** The range category of `rng`.

map_values example

```
#include <boost/range/adaptor/map.hpp>
#include <boost/range/algorithm/copy.hpp>
#include <boost/assign.hpp>
#include <iterator>
#include <iostream>
#include <map>
#include <vector>

int main(int argc, const char* argv[])
{
    using namespace boost::assign;
    using namespace boost::adaptors;

    std::map<int,int> input;
    for (int i = 0; i < 10; ++i)
        input.insert(std::make_pair(i, i * 10));

    boost::copy(
        input | map_values,
        std::ostream_iterator<int>(std::cout, ","));

    return 0;
}
```

This would produce the output:

```
0,10,20,30,40,50,60,70,80,90,
```

replaced

Syntax	Code
Pipe	`rng | boost::adaptors::replaced(new_value, old_value)`
Function	`boost::adaptors::replace(rng, new_value, old_value)`

- **Precondition:**
  - `new_value` is convertible to the `value_type` of the range.
  - `old_value` is convertible to the `value_type` of the range.
- **Postcondition:** For all elements `x` in the returned range, the value `x` is equal to the value of `(y == old_value) ? new_value : y` where `y` is the corresponding element in the original range.
- **Range Category:** Forward Range
- **Range Return Type:** `boost::replaced_range<typeof(rng)>`
- **Returned Range Category:** The range category of `rng`.

replaced example

```
#include <boost/range/adaptor/replaced.hpp>
#include <boost/range/algorithm/copy.hpp>
#include <boost/assign.hpp>
#include <iterator>
#include <iostream>
#include <vector>

int main(int argc, const char* argv[])
{
    using namespace boost::adaptors;
    using namespace boost::assign;

    std::vector<int> input;
    input += 1,2,3,2,5,2,7,2,9;

    boost::copy(
        input | replaced(2, 10),
        std::ostream_iterator<int>(std::cout, ","));

    return 0;
}
```

This would produce the output:

```
1,10,3,10,5,10,7,10,9,
```

replaced_if

Syntax	Code
Pipe	`rng	boost::adaptors::replaced_if(pred, new_value)`
Function	`boost::adaptors::replace_if(rng, pred, new_value)`

- **Precondition:**
  - The range `value_type` is convertible to the argument type of `pred`.
  - `new_value` is convertible to the `value_type` of the range.
- **Postconditions:** For all elements `x` in the returned range, the value `x` is equal to the value of `pred(y) ? new_value : y` where `y` is the corresponding element in the original range.
- **Range Category:** Forward Range
- **Range Return Type:** `boost::replaced_if_range<typeof(rng)>`
- **Returned Range Category:** The range category of `rng`.

replaced_if example

```
#include <boost/range/adaptor/replaced_if.hpp>
#include <boost/range/algorithm/copy.hpp>
#include <boost/assign.hpp>
#include <iterator>
#include <iostream>
#include <vector>

struct is_even
{
    bool operator()(int x) const { return x % 2 == 0; }
};

int main(int argc, const char* argv[])
{
    using namespace boost::adaptors;
    using namespace boost::assign;

    std::vector<int> input;
    input += 1,2,3,4,5,6,7,8,9;

    boost::copy(
        input | replaced_if(is_even(), 10),
        std::ostream_iterator<int>(std::cout, ","));

    return 0;
}
```

This would produce the output:

```
1,10,3,10,5,10,7,10,9,
```

reversed

Syntax	Code
Pipe	`rng | boost::adaptors::reversed`
Function	`boost::adaptors::reverse(rng)`

- **Returns:** A range whose iterators behave as if they were the original iterators wrapped in `reverse_iterator`.
- **Range Category:** Bidirectional Range
- **Range Return Type:** `boost::reversed_range<typeof(rng)>`
- **Returned Range Category:** The range category of `rng`.

reversed example

```
#include <boost/range/adaptor/reversed.hpp>
#include <boost/range/algorithm/copy.hpp>
#include <boost/assign.hpp>
#include <iterator>
#include <iostream>
#include <vector>

int main(int argc, const char* argv[])
{
    using namespace boost::adaptors;
    using namespace boost::assign;

    std::vector<int> input;
    input += 1,2,3,4,5,6,7,8,9;

    boost::copy(
        input | reversed,
        std::ostream_iterator<int>(std::cout, ","));

    return 0;
}
```

This would produce the output:

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

sliced

Syntax	Code
Pipe	`rng | boost::adaptors::sliced(n, m)`
Function	`boost::adaptors::slice(rng, n, m)`

- **Precondition:** `0 <= n && n <= m && m < distance(rng)`
- **Returns:** `make_range(rng, n, m)`
- **Range Category:** Random Access Range
- **Range Return Type:** `boost::sliced_range<typeof(rng)>`
- **Returned Range Category:** Random Access Range

sliced example

```
#include <boost/range/adaptor/sliced.hpp>
#include <boost/range/algorithm/copy.hpp>
#include <boost/assign.hpp>
#include <iterator>
#include <iostream>
#include <vector>

int main(int argc, const char* argv[])
{
    using namespace boost::adaptors;
    using namespace boost::assign;

    std::vector<int> input;
    input += 1,2,3,4,5,6,7,8,9;

    boost::copy(
        input | sliced(2, 5),
        std::ostream_iterator<int>(std::cout, ","));

    return 0;
}
```

This would produce the output:

```
3,4,5,
```

strided

Syntax	Code
Pipe	`rng | boost::adaptors::strided(n)`
Function	`boost::adaptors::stride(rng, n)`

- **Precondition:** `0 <= n`.
- **Returns:** A new range based on `rng` where traversal is performed in steps of `n`.
- **Range Category:** Single Pass Range
- **Returned Range Category:** The range category of `rng`.

strided example

```
#include <boost/range/adaptor/strided.hpp>
#include <boost/range/algorithm/copy.hpp>
#include <boost/assign.hpp>
#include <iterator>
#include <iostream>
#include <vector>

int main(int argc, const char* argv[])
{
    using namespace boost::adaptors;
    using namespace boost::assign;

    std::vector<int> input;
    input += 1,2,3,4,5,6,7,8,9,10;

    boost::copy(
        input | strided(2),
        std::ostream_iterator<int>(std::cout, ","));

    return 0;
}
```

This would produce the output:

```
1,3,5,7,9,
```

type_erased

Syntax	Code
Pipe	`rng | boost::adaptors::type_erased<Value, Traversal, Reference, Difference, Buffer>()`
Function	`boost::adaptors::type_erase(rng, boost::adaptors::type_erased<Value, Traversal, Reference, Difference, Buffer>)`

Please note that it is frequently unnecessary to use the `type_erased` adaptor. It is often better to use the implicit conversion to `any_range`.

Let `Rng` be the type of `rng`.

- **Template parameters:**
  - `Value` is the `value_type` for the `any_range`. If this is set to boost::use_default, `Value` will be calculated from the range type when the adaptor is applied.
  - `Traversal` is the tag used to identify the traversal of the resultant range. Frequently it is desirable to set a traversal category lower than the source container or range to maximize the number of ranges that can convert to the `any_range`. If this is left as boost::use_default then `Traversal` will be `typename boost::iterator_traversal<boost::range_iterator<Rng>::type>::type`
  - `Reference` is the `reference` for the `any_range`. `boost::use_default` will equate to `typename range_reference<Rng>::type`.
  - `Difference` is the `difference_type` for the `any_range`. `boost::use_default` will equate to `typename boost::range_difference<Rng>::type`

  - `Buffer` is the storage used to allocate the underlying iterator wrappers. This can typically be ignored, but is available as a template parameter for customization. Buffer must be a model of the `AnyIteratorBufferConcept`.

- **Precondition:** `Traversal` is one of `{ boost::use_default, boost::single_pass_traversal_tag, boost::forward_traversal_tag, boost::bidirectional_traversal_tag, boost::random_access_traversal_tag }`

- **Returns:** The returned value is the same as `typename any_range_type_generator< Rng, Value, Traversal, Reference, Difference, Buffer >` that represents `rng` in a type-erased manner.

- **Range Category:** Single Pass Range

- **Returned Range Category:** if `Traversal` was specified as `boost::use_default` then `typename boost::iterator_traversal<boost::range_iterator<Rng>::type>::type`, otherwise `Traversal`.

AnyIteratorBufferConcept

```
class AnyIteratorBufferConcept
{
public:
    AnyIteratorBufferConcept();
    ~AnyIteratorBufferConcept();

    // bytes is the requested size to allocate. This function
    // must return a pointer to an adequate area of memory.
    // throws: bad_alloc
    //
    // The buffer will only ever have zero or one
    // outstanding memory allocations.
    void* allocate(std::size_t bytes);

    // deallocate this buffer
    void deallocate();
};
```

type-erased example

```
#include <boost/range/adaptor/type_erased.hpp>
#include <boost/range/algorithm/copy.hpp>
#include <boost/assign.hpp>
#include <boost/foreach.hpp>
#include <iterator>
#include <iostream>
#include <list>
#include <vector>

// The client interface from an OO perspective merely requires a sequence
// of integers that can be forward traversed
typedef boost::any_range<
    int
  , boost::forward_traversal_tag
  , int
  , std::ptrdiff_t
> integer_range;

namespace server
{
    void display_integers(const integer_range& rng)
    {
        boost::copy(rng,
                    std::ostream_iterator<int>(std::cout, ","));

        std::cout << std::endl;
    }
}

namespace client
{
    void run()
    {
        using namespace boost::assign;
        using namespace boost::adaptors;

        // Under most conditions one would simply use an appropriate
        // any_range as a function parameter. The type_erased adaptor
        // is often superfluous. However because the type_erased
        // adaptor is applied to a range, we can use default template
        // arguments that are generated in conjunction with the
        // range type to which we are applying the adaptor.

        std::vector<int> input;
        input += 1,2,3,4,5;

        // Note that this call is to a non-template function
        server::display_integers(input);

        std::list<int> input2;
        input2 += 6,7,8,9,10;

        // Note that this call is to the same non-tempate function
        server::display_integers(input2);

        input2.clear();
        input2 += 11,12,13,14,15;

        // Calling using the adaptor looks like this:
        // Notice that here I have a type_erased that would be a
        // bidirectional_traversal_tag, but this is convertible
```

```
        // to the forward_traversal_tag equivalent hence this
        // works.
        server::display_integers(input2 | type_erased<>());

        // However we may simply wish to define an adaptor that
        // takes a range and makes it into an appropriate
        // forward_traversal any_range...
        typedef boost::adaptors::type_erased<
            boost::use_default
          , boost::forward_traversal_tag
        > type_erased_forward;

        // This adaptor can turn other containers with different
        // value_types and reference_types into the appropriate
        // any_range.

        server::display_integers(input2 | type_erased_forward());
    }
}

int main(int argc, const char* argv[])
{
    client::run();
    return 0;
}
```

This would produce the output:

```
1,2,3,4,5,
6,7,8,9,10,
11,12,13,14,15,
11,12,13,14,15,
```

tokenized

Syntax	Code

Pipe	`rng | boost::adaptors::tokenized(regex)`
	`rng | boost::adaptors::tokenized(regex, i)`
	`rng | boost::adaptors::token↵ized(regex, rndRng)`
	`rng | boost::adaptors::token↵ized(regex, i, flags)`
	`rng | boost::adaptors::token↵ized(regex, rndRng, flags)`

Function	`boost::adaptors::tokenize(rng, regex)`
	`boost::adaptors::tokenize(rng, regex, i)`
	`boost::adaptors::tokenize(rng, regex, rndRng)`
	`boost::adaptors::token↵ize(rng, regex, i, flags)`
	`boost::adaptors::token↵ize(rng, regex, rndRng, flags)`

- **Precondition:**

  - Let `T` denote `typename range_value<decltype(rng)>::type`, then `regex` has the type `basic_regex<T>` or is implicitly convertible to one of these types.
  - `i` has the type `int`.
  - the `value_type` of `rndRng` is `int`.
  - `flags` has the type `regex_constants::syntax_option_type`.

- **Returns:** A range whose iterators behave as if they were the original iterators wrapped in `regex_token_iterator`. The first iterator in the range would be constructed by forwarding all the arguments of `tokenized()` to the `regex_token_iterator` constructor.
- **Throws:** Whatever constructing and copying equivalent `regex_token_iterators` might throw.
- **Range Category:** Random Access Range
- **Range Return Type:** `boost::tokenized_range<typeof(rng)>`
- **Returned Range Category:** Random Access Range

tokenized_example

```
#include <boost/range/adaptor/tokenized.hpp>
#include <boost/range/algorithm/copy.hpp>
#include <boost/assign.hpp>
#include <iterator>
#include <iostream>
#include <vector>

int main(int argc, const char* argv[])
{
    using namespace boost::adaptors;

    typedef boost::sub_match< std::string::iterator > match_type;

    std::string input = " a b c d e f g hijklmnopqrstuvwxyz";
    boost::copy(
        input | tokenized(boost::regex("\\w+")),
        std::ostream_iterator<match_type>(std::cout, "\n"));

    return 0;
}
```

This would produce the output:

```
a
b
c
d
e
f
g
hijklmnopqrstuvwxyz
```

transformed

Syntax	Code
Pipe	`rng | boost::adaptors::transformed(fun)`
Function	`boost::adaptors::transform(rng, fun)`

- **Precondition:** The `value_type` of the range is convertible to the argument type of `fun`.
- **Postcondition:** For all elements `x` in the returned range, `x` is the result of `fun(y)` where `y` is the corresponding element in the original range.
- **Throws:** Whatever the copy-constructor of `fun` might throw.
- **Range Category:** Single Pass Range
- **Range Return Type:** `boost::transformed_range<typeof(rng)>`
- **Returned Range Category:** The range category of `rng`.

transformed example

```
#include <boost/range/adaptor/transformed.hpp>
#include <boost/range/algorithm/copy.hpp>
#include <boost/assign.hpp>
#include <iterator>
#include <iostream>
#include <vector>

struct double_int
{
    typedef int result_type;
    int operator()(int x) const { return x * 2; }
};

int main(int argc, const char* argv[])
{
    using namespace boost::adaptors;
    using namespace boost::assign;

    std::vector<int> input;
    input += 1,2,3,4,5,6,7,8,9,10;

    boost::copy(
        input | transformed(double_int()),
        std::ostream_iterator<int>(std::cout, ","));

    return 0;
}
```

This would produce the output:

```
2,4,6,8,10,12,14,16,18,20,
```

uniqued

Syntax	Code
Pipe	`rng	boost::adaptors::uniqued`
Function	`boost::adaptors::unique(rng)`

- **Precondition:** The `value_type` of the range is comparable with `operator==()`.
- **Postcondition:** For all adjacent elements `[x,y]` in the returned range, `x==y` is false.
- **Range Category:** Forward Range
- **Range Return Type:** `boost::uniqued_range<typeof(rng)>`
- **Returned Range Category:** The minimum of the range concept of `rng` and Forward Range.

uniqued example

```
#include <boost/range/adaptor/uniqued.hpp>
#include <boost/range/algorithm/copy.hpp>
#include <boost/assign.hpp>
#include <iterator>
#include <iostream>
#include <vector>

int main(int argc, const char* argv[])
{
    using namespace boost::assign;
    using namespace boost::adaptors;

    std::vector<int> input;
    input += 1,1,2,2,2,3,4,5,6;

    boost::copy(
        input | uniqued,
        std::ostream_iterator<int>(std::cout, ","));

    return 0;
}
```

This would produce the output:

```
1,2,3,4,5,6,
```

Range Algorithms

Introduction and motivation

In its most simple form a **Range Algorithm** (or range-based algorithm) is simply an iterator-based algorithm where the *two* iterator arguments have been replaced by *one* range argument. For example, we may write

```
#include <boost/range/algorithm.hpp>
#include <vector>

std::vector<int> vec = ...;
boost::sort(vec);
```

instead of

```
std::sort(vec.begin(), vec.end());
```

However, the return type of range algorithms is almost always different from that of existing iterator-based algorithms.

One group of algorithms, like `boost::sort()`, will simply return the same range so that we can continue to pass the range around and/or further modify it. Because of this we may write

```
boost:unique(boost::sort(vec));
```

to first sort the range and then run `unique()` on the sorted range.

Algorithms like `boost::unique()` fall into another group of algorithms that return (potentially) narrowed views of the original range. By default `boost::unique(rng)` returns the range `[boost::begin(rng), found)` where `found` denotes the iterator returned by `std::unique(boost::begin(rng), boost::end(rng))`

Therefore exactly the unique values can be copied by writing

```
boost::copy(boost::unique(boost::sort(vec)),
            std::ostream_iterator<int>(std::cout));
```

Algorithms like `boost::unique` usually return the same range: `[boost::begin(rng), found)`. However, this behaviour may be changed by supplying the algorithms with a template argument:

Expression	**Return**
`boost::unique<boost::return_found>(rng)`	returns a single iterator like `std::unique`
`boost::unique<boost::return_begin_found>(rng)`	returns the range `[boost::begin(rng), found)` (this is the default)
`boost::unique<boost::return_begin_next>(rng)`	returns the range `[boost::begin(rng), boost::next(found))`
`boost::unique<boost::return_found_end>(rng)`	returns the range `[found, boost::end(rng))`
`boost::unique<boost::return_next_end>(rng)`	returns the range `[boost::next(found),boost::end(rng))`
`boost::unique<boost::return_begin_end>(rng)`	returns the entire original range.

This functionality has the following advantages:

1. it allows for ***seamless functional-style programming*** where you do not need to use named local variables to store intermediate results
2. it is very ***safe*** because the algorithm can verify out-of-bounds conditions and handle tricky conditions that lead to empty ranges

For example, consider how easy we may erase the duplicates in a sorted container:

```
std::vector<int> vec = ...;
boost::erase(vec, boost::unique<boost::return_found_end>(boost::sort(vec)));
```

Notice the use of `boost::return_found_end`. What if we wanted to erase all the duplicates except one of them? In old-fashined STL-programming we might write

```
// assume 'vec' is already sorted
std::vector<int>::iterator i = std::unique(vec.begin(), vec.end());

// remember this check or you get into problems
if (i != vec.end())
    ++i;

vec.erase(i, vec.end());
```

The same task may be accomplished simply with

```
boost::erase(vec, boost::unique<boost::return_next_end>(vec));
```

and there is no need to worry about generating an invalid range. Furthermore, if the container is complex, calling `vec.end()` several times will be more expensive than using a range algorithm.

Mutating algorithms

copy

Prototype

```
template<class SinglePassRange, class OutputIterator>
OutputIterator copy(const SinglePassRange& source_rng, OutputIterator out_it);
```

Description

`copy` copies all elements from `source_rng` to the range `[out_it, out_it + distance(source_rng))`. The return value is `out_it + distance(source_rng)`

Definition

Defined in the header file `boost/range/algorithm/copy.hpp`

Requirements

- `SinglePassRange` is a model of the Single Pass Range Concept.
- `OutputIterator` is a model of the `OutputIteratorConcept`.
- The `value_type` of Single Pass Range Concept is convertible to a type in `OutputIterator`'s set of value types.

Precondition:

- `out_it` is not an iterator within the `source_rng`.
- `[out_it, out_it + distance(source_rng))` is a valid range.

Complexity

Linear. Exactly `distance(source_rng)` assignments are performed.

copy_backward

Prototype

```
template<class BidirectionalRange, class BidirectionalOutputIterator>
    BidirectionalOutputIterator
        copy_backward(const BidirectionalRange& source_rng,
                      BidirectionalOutputIterator out_it);
```

Description

`copy_backward` copies all elements from `source_rng` to the range `[out_it - distance(source_rng), out_it)`.

The values are copied in reverse order. The return value is `out_it - distance(source_rng)`.

Note well that unlike all other standard algorithms `out_it` denotes the **end** of the output sequence.

Definition

Defined in the header file `boost/range/algorithm/copy_backward.hpp`

Requirements

- `BidirectionalRange` is a model of Bidirectional Range Concept.
- `OutputIterator` is a model of the `OutputIteratorConcept`.
- The `value_type` of Bidirectional Range Concept is convertible to a type in `OutputIterator`'s set of value types.

Precondition:

- `out_it` is not an iterator within the `source_rng`.
- `[out_it, out_it + distance(source_rng))` is a valid range.

Complexity

Linear. Exactly `distance(source_rng)` assignments are performed.

fill

Prototype

```
template<class ForwardRange, class Value>
ForwardRange& fill( ForwardRange& rng, const Value& val );
```

Description

`fill` assigns the value `val` to every element in the range `rng`.

Definition

Defined in the header file `boost/range/algorithm/fill.hpp`

Requirements

- `ForwardRange` is a model of the Forward Range Concept.
- `ForwardRange` is mutable.

- `Value` is a model of the `AssignableConcept`.
- `Value` is convertible to `ForwardRange`'s value type.

Complexity

Linear. Exactly `distance(rng)` assignments are performed.

fill_n

Prototype

```
template<class ForwardRange, class Size, class Value>
ForwardRange& fill( ForwardRange& rng, Size n, const Value& val );
```

Description

`fill_n` assigns the value `val` to `n` elements in the range `rng` beginning with `boost::begin(rng)`.

Definition

Defined in the header file `boost/range/algorithm/fill_n.hpp`

Requirements

- `ForwardRange` is a model of the Forward Range Concept.
- `ForwardRange` is mutable.
- `Value` is a model of the `AssignableConcept`.
- `Value` is convertible to `ForwardRange`'s value type.

Complexity

Linear. Exactly `n` assignments are performed.

generate

Prototype

```
template<class ForwardRange, class Generator>
ForwardRange& generate( ForwardRange& rng, Generator gen );

template<class ForwardRange, class Generator>
const ForwardRange& generate( const ForwardRange& rng, Generator gen );
```

Description

`generate` assigns the result of `gen()` to each element in range `rng`. Returns the resultant range.

Definition

Defined in the header file `boost/range/algorithm/generate.hpp`

Requirements

- `ForwardRange` is a model of the Forward Range Concept.
- `ForwardRange` is mutable.

- `Generator` is a model of the `GeneratorConcept`.
- The `value_type` of `SinglePassRange` is convertible to a type in `OutputIterator`'s set of value types.

Precondition:

- `out_it` is not an iterator within `rng`.
- `[out_it, out_it + distance(rng))` is a valid range.

Complexity

Linear. Exactly `distance(rng)` assignments are performed.

inplace_merge

Prototype

```
template<class BidirectionalRange>
BidirectionalRange&
inplace_merge( BidirectionalRange& rng,
               typename range_iterator<BidirectionalRange>::type middle );

template<class BidirectionalRange>
const BidirectionalRange&
inplace_merge( const BidirectionalRange& rng,
               typename range_iterator<const BidirectionalRange>::type middle );

template<class BidirectionalRange, class BinaryPredicate>
BidirectionalRange&
inplace_merge( BidirectionalRange& rng,
               typename range_iterator<BidirectionalRange>::type middle,
               BinaryPredicate pred );

template<class BidirectionalRange, class BinaryPredicate>
const BidirectionalRange&
inplace_merge( const BidirectionalRange& rng,
               typename range_iterator<const BidirectionalRange>::type middle,
               BinaryPredicate pred );
```

Description

`inplace_merge` combines two consecutive sorted ranges `[begin(rng), middle)` and `[middle, end(rng))` into a single sorted range `[begin(rng), end(rng))`. That is, it starts with a range `[begin(rng), end(rng))` that consists of two pieces each of which is in ascending order, and rearranges it so that the entire range is in ascending order. `inplace_merge` is stable, meaning both that the relative order of elements within each input range is preserved.

Definition

Defined in the header file `boost/range/algorithm/inplace_merge.hpp`

Requirements

For the non-predicate version:

- `BidirectionalRange` is a model of the Bidirectional Range Concept.
- `BidirectionalRange` is mutable.
- `range_value<BidirectionalRange>::type` is a model of `LessThanComparableConcept`

- The ordering on objects of `range_type<BidirectionalRange>::type` is a **strict weak ordering**, as defined in the `LessThanComparableConcept` requirements.

For the predicate version: * `BidirectionalRange` is a model of the Bidirectional Range Concept. * `BidirectionalRange` is mutable. * `BinaryPredicate` is a model of the `StrictWeakOrderingConcept`. * `BidirectionalRange`'s value type is convertible to both `BinaryPredicate`'s argument types.

Precondition:

For the non-predicate version:

- `middle` is in the range `rng`.
- `[begin(rng), middle)` is in ascending order. That is for each pair of adjacent elements `[x,y]`, `y < x` is `false`.
- `[middle, end(rng))` is in ascending order. That is for each pair of adjacent elements `[x,y]`, `y < x` is `false`.

For the predicate version:

- `middle` is in the range `rng`.
- `[begin(rng), middle)` is in ascending order. That is for each pair of adjacent elements `[x,y]`, `pred(y,x) == false`.
- `[middle, end(rng))` is in ascending order. That is for each pair of adjacent elements `[x,y]`, `pred(y,x) == false`.

Complexity

Worst case: `O(N log(N))`

merge

Prototype

```
template<
    class SinglePassRange1,
    class SinglePassRange2,
    class OutputIterator
    >
OutputIterator merge(const SinglePassRange1& rng1,
                     const SinglePassRange2& rng2,
                     OutputIterator          out);

template<
    class SinglePassRange1,
    class SinglePassRange2,
    class OutputIterator,
    class BinaryPredicate
    >
OutputIterator merge(const SinglePassRange1& rng1,
                     const SinglePassRange2& rng2,
                     OutputIterator          out,
                     BinaryPredicate         pred);
```

Description

`merge` combines two sorted ranges `rng1` and `rng2` into a single sorted range by copying elements. `merge` is stable. The return value is `out + distance(rng1) + distance(rng2)`.

The two versions of `merge` differ by how they compare the elements.

The non-predicate version uses the `operator<()` for the range value type. The predicate version uses the predicate instead of `operator<()`.

Definition

Defined in the header file `boost/range/algorithm/merge.hpp`

Requirements

For the non-predicate version:

- `SinglePassRange1` is a model of the Single Pass Range Concept.
- `SinglePassRange2` is a model of the Single Pass Range Concept.
- `range_value<SinglePassRange1>::type` is the same as `range_value<SinglePassRange2>::type`.
- `range_value<SinglePassRange1>::type` is a model of the `LessThanComparableConcept`.
- The ordering on objects of `range_value<SinglePassRange1>::type` is a **strict weak ordering**, as defined in the `LessThanComparableConcept` requirements.
- `range_value<SinglePassRange1>::type` is convertible to a type in `OutputIterator`'s set of value types.

For the predicate version:

- `SinglePassRange1` is a model of the Single Pass Range Concept.
- `SinglePassRange2` is a model of the Single Pass Range Concept.
- `range_value<SinglePassRange1>::type` is the same as `range_value<SinglePassRange2>::type`.
- `BinaryPredicate` is a model of the `StrictWeakOrderingConcept`.
- `SinglePassRange1`'s value type is convertible to both `BinaryPredicate`'s argument types.
- `range_value<SinglePassRange1>::type` is convertible to a type in `OutputIterator`'s set of value types.

Precondition:

For the non-predicate version:

- The elements of `rng1` are in ascending order. That is, for each adjacent element pair `[x,y]` of `rng1`, `y < x == false`.
- The elements of `rng2` are in ascending order. That is, for each adjacent element pair `[x,y]` of `rng2`, `y < x == false`.
- The ranges `rng1` and `[out, out + distance(rng1) + distance(rng2))` do not overlap.
- The ranges `rng2` and `[out, out + distance(rng1) + distance(rng2))` do not overlap.
- `[out, out + distance(rng1) + distance(rng2))` is a valid range.

For the predicate version:

- The elements of `rng1` are in ascending order. That is, for each adjacent element pair `[x,y]`, of `rng1`, `pred(y, x) == false`.
- The elements of `rng2` are in ascending order. That is, for each adjacent element pair `[x,y]`, of `rng2`, `pred(y, x) == false`.
- The ranges `rng1` and `[out, out + distance(rng1) + distance(rng2))` do not overlap.
- The ranges `rng2` and `[out, out + distance(rng1) + distance(rng2))` do not overlap.
- `[out, out + distance(rng1) + distance(rng2))` is a valid range.

Complexity

Linear. There are no comparisons if both `rng1` and `rng2` are empty, otherwise at most `distance(rng1) + distance(rng2) - 1` comparisons.

nth_element

Prototype

```
template<class RandomAccessRange>
RandomAccessRange& nth_element(
    RandomAccessRange& rng,
    typename range_iterator<RandomAccessRange>::type nth);

template<class RandomAccessRange>
const RandomAccessRange& nth_element(
    const RandomAccessRange& rng,
    typename range_iterator<const RandomAccessRange>::type nth);

template<class RandomAccessRange>
RandomAccessRange& nth_element(
    RandomAccessRange& rng,
    typename range_iterator<RandomAccessRange>::type nth,
    BinaryPredicate sort_pred);

template<class RandomAccessRange>
const RandomAccessRange& nth_element(
    const RandomAccessRange& rng,
    typename range_iterator<const RandomAccessRange>::type nth,
    BinaryPredicate sort_pred);
```

Description

`nth_element` partially orders a range of elements. `nth_element` arranges the range `rng` such that the element corresponding with the iterator `nth` is the same as the element that would be in that position if `rng` has been sorted.

Definition

Defined in the header file `boost/range/algorithm/nth_element.hpp`

Requirements

For the non-predicate version:

- `RandomAccessRange` is a model of the Random Access Range Concept.
- `RandomAccessRange` is mutable.
- `RandomAccessRange`'s value type is a model of the `LessThanComparableConcept`.
- The ordering relation on `RandomAccessRange`'s value type is a **strict weak ordering**, as defined in the `LessThanComparableConcept` requirements.

For the predicate version:

- `RandomAccessRange` is a model of the Random Access Range Concept.
- `RandomAccessRange` is mutable.
- `BinaryPredicate` is a model of the `StrictWeakOrderingConcept`.
- `RandomAccessRange`'s value type is convertible to both of `BinaryPredicate`'s argument types.

Complexity

On average, linear in `distance(rng)`.

partial_sort

Prototype

```
template<class RandomAccessRange>
RandomAccessRange& partial_sort(
    RandomAccessRange& rng,
    typename range_iterator<RandomAccessRange>::type middle);

template<class RandomAccessRange>
const RandomAccessRange& partial_sort(
    const RandomAccessRange& rng,
    typename range_iterator<const RandomAccessRange>::type middle);

template<class RandomAccessRange>
RandomAccessRange& partial_sort(
    RandomAccessRange& rng,
    typename range_iterator<RandomAccessRange>::type middle,
    BinaryPredicate sort_pred);

template<class RandomAccessRange>
const RandomAccessRange& partial_sort(
    const RandomAccessRange& rng,
    typename range_iterator<const RandomAccessRange>::type middle,
    BinaryPredicate sort_pred);
```

Description

`partial_sort` rearranges the elements in `rng`. It places the smallest `distance(begin(rng), middle)` elements, sorted in ascending order, into the range `[begin(rng), middle)`. The remaining elements are placed in an unspecified order into `[middle, last)`.

The non-predicative versions of this function specify that one element is less than another by using `operator<()`. The predicate versions use the predicate instead.

Definition

Defined in the header file `boost/range/algorithm/partial_sort.hpp`

Requirements

For the non-predicate version:

- `RandomAccessRange` is a model of the Random Access Range Concept.
- `RandomAccessRange` is mutable.
- `RandomAccessRange`'s value type is a model of the `LessThanComparableConcept`.
- The ordering relation on `RandomAccessRange`'s value type is a **strict weak ordering**, as defined in the `LessThanComparableConcept` requirements.

For the predicate version:

- `RandomAccessRange` is a model of the Random Access Range Concept.
- `RandomAccessRange` is mutable.

- `BinaryPredicate` is a model of the `StrictWeakOrderingConcept`.
- `RandomAccessRange`'s value type is convertible to both of `BinaryPredicate`'s argument types.

Complexity

Approximately `distance(rng) * log(distance(begin(rng), middle))` comparisons.

partition

Prototype

```
template<
    class ForwardRange,
    class UnaryPredicate
    >
typename range_iterator<ForwardRange>::type
partition(ForwardRange& rng, UnaryPredicate pred);

template<
    class ForwardRange,
    class UnaryPredicate
    >
typename range_iterator<const ForwardRange>::type
partition(const ForwardRange& rng, UnaryPredicate pred);

template<
    range_return_value re,
    class ForwardRange,
    class UnaryPredicate
    >
typename range_return<ForwardRange, re>::type
partition(ForwardRange& rng, UnaryPredicate pred);

template<
    range_return_value re,
    class ForwardRange,
    class UnaryPredicate
    >
typename range_return<const ForwardRange, re>::type
partition(const ForwardRange& rng, UnaryPredicate pred);
```

Description

`partition` orders the elements in `rng` based on `pred`, such that the elements that satisfy `pred` precede the elements that do not. In the versions that return a single iterator, the return value is the middle iterator. In the versions that have a configurable range_return, `found` corresponds to the middle iterator.

Definition

Defined in the header file `boost/range/algorithm/partition.hpp`

Requirements

- `ForwardRange` is a model of the Forward Range Concept.
- `UnaryPredicate` is a model of the `PredicateConcept`.
- `ForwardRange`'s value type is convertible to `UnaryPredicate`'s argument type.

Complexity

Linear. Exactly `distance(rng)` applications of `pred`, and at most `distance(rng) / 2` swaps.

random_shuffle

Prototype

```
template<class RandomAccessRange>
RandomAccessRange& random_shuffle(RandomAccessRange& rng);

template<class RandomAccessRange>
const RandomAccessRange& random_shuffle(const RandomAccessRange& rng);

template<class RandomAccessRange, class Generator>
RandomAccessRange& random_shuffle(RandomAccessRange& rng, Generator& gen);

template<class RandomAccessRange, class Generator>
const RandomAccessRange& random_shuffle(const RandomAccessRange& rng, Generator& gen);
```

Description

`random_shuffle` randomly rearranges the elements in `rng`. The versions of `random_shuffle` that do not specify a `Generator` use an internal random number generator. The versions of `random_shuffle` that do specify a `Generator` use this instead. Returns the shuffles range.

Definition

Defined in the header file `boost/range/algorithm/random_shuffle.hpp`

Requirements

For the version without a Generator:

- `RandomAccessRange` is a model of the Random Access Range Concept.

For the version with a Generator:

- `RandomAccessRange` is a model of the Random Access Range Concept.
- `Generator` is a model of the `RandomNumberGeneratorConcept`.
- `RandomAccessRange`'s distance type is convertible to `Generator`'s argument type.

Precondition:

- `distance(rng)` is less than `gen`'s maximum value.

Complexity

Linear. If `!empty(rng)`, exactly `distance(rng) - 1` swaps are performed.

remove

Prototype

```
template<
    class ForwardRange,
    class Value
    >
typename range_iterator<ForwardRange>::type
remove(ForwardRange& rng, const Value& val);

template<
    class ForwardRange,
    class Value
    >
typename range_iterator<const ForwardRange>::type
remove(const ForwardRange& rng, const Value& val);

template<
    range_return_value re,
    class ForwardRange,
    class Value
    >
typename range_return<ForwardRange,re>::type
remove(ForwardRange& rng, const Value& val);

template<
    range_return_value re,
    class ForwardRange,
    class Value
    >
typename range_return<const ForwardRange,re>::type
remove(const ForwardRange& rng, const Value& val);
```

Description

`remove` removes from `rng` all of the elements `x` for which `x == val` is `true`. The versions of `remove` that return an iterator, return an iterator `new_last` such that the range `[begin(rng), new_last)` contains no elements equal to `val`. The `range_return` versions of `remove` defines `found` as the new last element. The iterators in the range `[new_last, end(rng))` are dereferenceable, but the elements are unspecified.

Definition

Defined in the header file `boost/range/algorithm/remove.hpp`

Requirements

- `ForwardRange` is a model of the Forward Range Concept.
- `ForwardRange` is mutable.
- `Value` is a model of the `EqualityComparableConcept`.
- Objects of type `Value` can be compared for equality with objects of `ForwardRange`'s value type.

Complexity

Linear. `remove` performs exactly `distance(rng)` comparisons for equality.

remove_copy

Prototype

```
template<class ForwardRange, class Outputiterator, class Value>
OutputIterator
remove_copy(ForwardRange& rng, OutputIterator out, const Value& val);

template<class ForwardRange, class OutputIterator, class Value>
OutputIterator
remove_copy(const ForwardRange& rng, OutputIterator out, const Value& val);
```

Description

`remove_copy` copied all of the elements `x` from `rng` for which `x == val` is `false`.

Definition

Defined in the header file `boost/range/algorithm/remove_copy.hpp`

Requirements

- `ForwardRange` is a model of the Forward Range Concept.
- `ForwardRange` is mutable.
- `Value` is a model of the `EqualityComparableConcept`.
- Objects of type `Value` can be compared for equality with objects of `ForwardRange`'s value type.

Complexity

Linear. `remove_copy` performs exactly `distance(rng)` comparisons for equality.

remove_copy_if

Prototype

```
template<class ForwardRange, class Outputiterator, class UnaryPred>
OutputIterator
remove_copy_if(ForwardRange& rng, OutputIterator out, UnaryPred pred);

template<class ForwardRange, class OutputIterator, class UnaryPred>
OutputIterator
remove_copy_if(const ForwardRange& rng, OutputIterator out, UnaryPred pred);
```

Description

`remove_copy_if` copied all of the elements `x` from `rng` for which `pred(x)` is `false`.

Definition

Defined in the header file `boost/range/algorithm/remove_copy_if.hpp`

Requirements

- `ForwardRange` is a model of the Forward Range Concept.
- `ForwardRange` is mutable.
- `UnaryPred` is a model of the `UnaryPredicateConcept`.

Complexity

Linear. `remove_copy_if` performs exactly `distance(rng)` comparisons with UnaryPred.

remove_if

Prototype

```
template<
    class ForwardRange,
    class UnaryPredicate
    >
typename range_iterator<ForwardRange>::type
remove(ForwardRange& rng, UnaryPredicate pred);

template<
    class ForwardRange,
    class UnaryPredicate
    >
typename range_iterator<const ForwardRange>::type
remove(const ForwardRange& rng, UnaryPredicate pred);

template<
    range_return_value re,
    class ForwardRange,
    class UnaryPredicate
    >
typename range_return<ForwardRange,re>::type
remove(ForwardRange& rng, UnaryPredicate pred);

template<
    range_return_value re,
    class ForwardRange,
    class UnaryPredicate
    >
typename range_return<const ForwardRange,re>::type
remove(const ForwardRange& rng, UnaryPredicate pred);
```

Description

`remove_if` removes from `rng` all of the elements `x` for which `pred(x)` is `true`. The versions of `remove_if` that return an iterator, return an iterator `new_last` such that the range `[begin(rng), new_last)` contains no elements where `pred(x)` is `true`. The iterators in the range `[new_last, end(rng))` are dereferenceable, but the elements are unspecified.

Definition

Defined in the header file `boost/range/algorithm/remove_if.hpp`

Requirements

- `ForwardRange` is a model of the Forward Range Concept.
- `ForwardRange` is mutable.
- `UnaryPredicate` is a model of the `PredicateConcept`.
- `ForwardRange`'s value type is convertible to `UnaryPredicate`'s argument type.

Complexity

Linear. `remove_if` performs exactly `distance(rng)` applications of `pred`.

replace

Prototype

```
template<
    class ForwardRange,
    class Value
    >
ForwardRange& replace(ForwardRange& rng, const Value& what, const Value& with_what);

template<
    class ForwardRange,
    class UnaryPredicate
    >
const ForwardRange& replace(const ForwardRange& rng, const Value& what, const Value& with_what);
```

Description

`replace` every element in `rng` equal to `what` with `with_what`. Return a reference to `rng`.

Definition

Defined in the header file `boost/range/algorithm/replace.hpp`

Requirements

- `ForwardRange` is a model of the Forward Range Concept.
- `ForwardRange` is mutable.
- `Value` is convertible to `ForwardRange`'s value type.
- `Value` is a model of the `AssignableConcept`.
- `Value` is a model of the `EqualityComparableConcept`, and may be compared for equality with objects of `ForwardRange`'s value type.

Complexity

Linear. `replace` performs exactly `distance(rng)` comparisons for equality and at most `distance(rng)` assignments.

replace_copy

Prototype

```
template<class ForwardRange, class OutputIterator, class Value>
OutputIterator replace_copy(const ForwardRange& rng, OutputIterator out,
                            const Value& what, const Value& with_what);
```

Description

`replace_copy` copy every element `x` in `rng` such that the corresponding element in the output range `y` is `x == what ? with_what : x`.

Definition

Defined in the header file `boost/range/algorithm/replace_copy.hpp`

Requirements

- `ForwardRange` is a model of the Forward Range Concept.
- `ForwardRange` is mutable.
- `Value` is convertible to `ForwardRange`'s value type.
- `Value` is a model of the `AssignableConcept`.
- `OutputIterator` is a model of the `OutputIteratorConcept`.

Complexity

Linear. `replace_copy` performs exactly `distance(rng)`.

replace_copy_if

Prototype

```
template<class ForwardRange, class OutputIterator, class UnaryPredicate, class Value>
OutputIterator replace_copy_if(const ForwardRange& rng, OutputIterator out,
                               UnaryPredicate pred, const Value& with_what);
```

Description

`replace_copy_if` copy every element `x` in `rng` such that the corresponding element in the output range `y` is `pred(x) ? with_what : x`.

Definition

Defined in the header file `boost/range/algorithm/replace_copy_if.hpp`

Requirements

- `ForwardRange` is a model of the Forward Range Concept.
- `ForwardRange` is mutable.
- `Value` is convertible to `ForwardRange`'s value type.
- `Value` is a model of the `AssignableConcept`.
- `OutputIterator` is a model of the `OutputIteratorConcept`.
- `UnaryPredicate` is a model of the `UnaryPredicateConcept`.

Complexity

Linear. `replace_copy_if` performs exactly `distance(rng)` evaluations of `pred`.

replace_if

Prototype

```
template<class ForwardRange, class UnaryPredicate, class Value>
ForwardRange& replace_if(ForwardRange& rng, UnaryPredicate pred, const Value& with_what);

template<class ForwardRange, class UnaryPredicate, class Value>
const ForwardRange& replace_if(const ForwardRange& rng, UnaryPredic↵
ate pred, const Value& with_what);
```

Description

`replace_if` replaces every element `x` in `rng` for which `pred(x) == true` with `with_what`. Returns a reference to `rng`.

Definition

Defined in the header file `boost/range/algorithm/replace_if.hpp`

Requirements

- `ForwardRange` is a model of the Forward Range Concept.
- `ForwardRange` is mutable.
- `UnaryPredicate` is a model of the `PredicateConcept`
- `ForwardRange`'s value type is convertible to `UnaryPredicate`'s argument type.
- `Value` is convertible to `ForwardRange`'s value type.
- `Value` is a model of the `AssignableConcept`.

Complexity

Linear. `replace_if` performs exactly `distance(rng)` applications of `pred`, and at most `distance(rng)` assignments.

reverse

Prototype

```
template<class BidirectionalRange>
BidirectionalRange& reverse(BidirectionalRange& rng);

template<class BidirectionalRange>
const BidirectionalRange& reverse(const BidirectionalRange& rng);
```

Description

`reverse` reverses a range. Returns a reference to the reversed range.

Definition

Defined in the header file `boost/range/algorithm/reverse.hpp`

Requirements

- `BidirectionalRange` is a model of the Bidirectional Range Concept.
- `BidirectionalRange` is mutable.

Complexity

Linear. `reverse` makes `distance(rng)/2` calls to `iter_swap`.

reverse_copy

Prototype

```
template<class BidirectionalRange, class OutputIterator>
OutputIterator reverse_copy(const BidirectionalRange& rng, OutputIterator out);
```

Description

`reverse_copy` copies the elements from `rng` in reverse order to `out`. Returns the output iterator one passed the last copied element.

Definition

Defined in the header file `boost/range/algorithm/reverse_copy.hpp`

Requirements

- `BidirectionalRange` is a model of the Bidirectional Range Concept.
- `BidirectionalRange` is mutable.
- `OutputIterator` is a model of the `OutputIteratorConcept`.

Complexity

Linear. `reverse_copy` makes `distance(rng)` copies.

rotate

Prototype

```
template<class ForwardRange>
ForwardRange& rotate(ForwardRange& rng,
                     typename range_iterator<ForwardRange>::type middle);

template<class ForwardRange>
const ForwardRange& rotate(const ForwardRange& rng,
                           typename range_iterator<const ForwardRange>::type middle);
```

Description

`rotate` rotates the elements in a range. It exchanges the two ranges `[begin(rng), middle)` and `[middle, end(rng))`. Returns a reference to `rng`.

Definition

Defined in the header file `boost/range/algorithm/rotate.hpp`

Requirements

- `ForwardRange` is a model of the Forward Range Concept.
- `ForwardRange` is mutable.

Precondition:

- `[begin(rng), middle)` is a valid range.
- `[middle, end(rng))` is a valid range.

Complexity

Linear. At most `distance(rng)` swaps are performed.

rotate_copy

Prototype

```
template<class ForwardRange, class OutputIterator>
OutputIterator rotate_copy(
    const ForwardRange& rng,
    typename range_iterator<ForwardRange>::type middle,
    OutputIterator out);
```

Description

`rotate_copy` rotates the elements in a range. It copies the two ranges `[begin(rng), middle)` and `[middle, end(rng))` to `out`.

Definition

Defined in the header file `boost/range/algorithm/rotate_copy.hpp`

Requirements

- `ForwardRange` is a model of the Forward Range Concept.
- `ForwardRange` is mutable.
- `OutputIterator` is a model of the `OutputIteratorConcept`.

Precondition:

- `[begin(rng), middle)` is a valid range.
- `[middle, end(rng))` is a valid range.

Complexity

Linear. Exactly `distance(rng)` elements are copied.

sort

Prototype

```
template<class RandomAccessRange>
RandomAccessRange& sort(RandomAccessRange& rng);

template<class RandomAccessRange>
const RandomAccessRange& sort(const RandomAccessRange& rng);

template<class RandomAccessRange, class BinaryPredicate>
RandomAccessRange& sort(RandomAccessRange& rng, BinaryPredicate pred);

template<class RandomAccessRange, class BinaryPredicate>
const RandomAccessRange& sort(const RandomAccessRange& rng, BinaryPredicate pred);
```

Description

`sort` sorts the elements in `rng` into ascending order. `sort` is not guaranteed to be stable. Returns the sorted range.

For versions of the `sort` function without a predicate, ascending order is defined by `operator<()` such that for all adjacent elements `[x,y]`, `y < x == false`.

For versions of the `sort` function with a predicate, ascending order is defined by `pred` such that for all adjacent elements `[x,y]`, `pred(y, x) == false`.

Definition

Defined in the header file `boost/range/algorithm/sort.hpp`

Requirements

For versions of sort without a predicate:

- `RandomAccessRange` is a model of the Random Access Range Concept.
- `RandomAccessRange` is mutable.
- `RandomAccessRange`'s value type is a model of the `LessThanComparableConcept`.
- The ordering relation on `RandomAccessRange`'s value type is a **strict weak ordering**, as defined in the `LessThanComparableConcept` requirements.

For versions of sort with a predicate

- `RandomAccessRange` is a model of the Random Access Range Concept.
- `RandomAccessRange` is mutable.
- `BinaryPredicate` is a model of the `StrictWeakOrderingConcept`.
- `RandomAccessRange`'s value type is convertible to both of `BinaryPredicate`'s argument types.

Complexity

`O(N log(N))` comparisons (both average and worst-case), where `N` is `distance(rng)`.

stable_partition

Prototype

```
template<class ForwardRange, class UnaryPredicate>
typename range_iterator<ForwardRange>::type
stable_partition(ForwardRange& rng, UnaryPredicate pred);

template<class ForwardRange, class UnaryPredicate>
typename range_iterator<const ForwardRange>::type
stable_partition(const ForwardRange& rng, UnaryPredicate pred);

template<
    range_return_value re,
    class ForwardRange,
    class UnaryPredicate
>
typename range_return<ForwardRange, re>::type
stable_partition(ForwardRange& rng, UnaryPredicate pred);

template<
    range_return_value re,
    class ForwardRange,
    class UnaryPredicate
>
typename range_return<const ForwardRange, re>::type
stable_partition(const ForwardRange& rng, UnaryPredicate pred);
```

Description

`stable_partition` reorders the elements in the range `rng` base on the function object `pred`. Once this function has completed all of the elements that satisfy `pred` appear before all of the elements that fail to satisfy it. `stable_partition` differs from `partition` because it preserves relative order. It is stable.

For the versions that return an iterator, the return value is the iterator to the first element that fails to satisfy `pred`.

For versions that return a `range_return`, the `found` iterator is the iterator to the first element that fails to satisfy `pred`.

Definition

Defined in the header file `boost/range/algorithm/stable_partition.hpp`

Requirements

- `ForwardRange` is a model of the Forward Range Concept.
- `ForwardRange` is mutable.
- `UnaryPredicate` is a model of the `PredicateConcept`.

Complexity

Best case: `O(N)` where `N` is `distance(rng)`. Worst case: `N * log(N)` swaps, where `N` is `distance(rng)`.

stable_sort

Prototype

```
template<class RandomAccessRange>
RandomAccessRange& stable_sort(RandomAccessRange& rng);

template<class RandomAccessRange>
const RandomAccessRange& stable_sort(const RandomAccessRange& rng);

template<class RandomAccessRange, class BinaryPredicate>
RandomAccessRange& stable_sort(RandomAccessRange& rng, BinaryPredicate pred);

template<class RandomAccessRange, class BinaryPredicate>
const RandomAccessRange& stable_sort(const RandomAccessRange& rng, BinaryPredicate pred);
```

Description

`stable_sort` sorts the elements in `rng` into ascending order. `stable_sort` is guaranteed to be stable. The order is preserved for equivalent elements.

For versions of the `stable_sort` function without a predicate ascending order is defined by `operator<()` such that for all adjacent elements `[x,y]`, `y < x == false`.

For versions of the `stable_sort` function with a predicate, ascending order is designed by `pred` such that for all adjacent elements `[x,y]`, `pred(y,x) == false`.

Definition

Defined in the header file `boost/range/algorithm/stable_sort.hpp`

Requirements

For versions of stable_sort without a predicate

- `RandomAccessRange` is a model of the Random Access Range Concept.
- `RandomAccessRange` is mutable.
- `RandomAccessRange`'s value type is a model of the `LessThanComparableConcept`.
- The ordering relation on `RandomAccessRange`'s value type is a **strict weak ordering**, as defined in the `LessThanComparableConcept` requirements.

For versions of stable_sort with a predicate:

- `RandomAccessRange` is a model of the Random Access Range Concept.
- `RandomAccessRange` is mutable.
- `BinaryPredicate` is a model of the `StrictWeakOrderingConcept`.
- `RandomAccessRange`'s value type is convertible to both of `BinaryPredicate`'s argument types.

Complexity

Best case: `O(N)` where `N` is `distance(rng)`. Worst case: `O(N log(N)^2)` comparisons, where `N` is `distance(rng)`.

swap_ranges

Prototype

```
template<class SinglePassRange1, class SinglePassRange2>
SinglePassRange2& swap_ranges(SinglePassRange1& rng1, SinglePassRange& rng2);
```

Description

`swap_ranges` swaps each element `x` in `rng1` with the corresponding element `y` in `rng2`. Returns a reference to `rng2`.

Definition

Defined in the header file `boost/range/algorithm/swap_ranges.hpp`

Requirements

- `SinglePassRange1` is a model of the Single Pass Range Concept.
- `SinglePassRange1` is mutable.
- `SinglePassRange2` is a model of the Single Pass Range Concept.
- `SinglePassRange2` is mutable.

Complexity

Linear. Exactly `distance(rng1)` elements are swapped.

transform

Prototype

```
template<
    class SinglePassRange1,
    class OutputIterator,
    class UnaryOperation
>
OutputIterator transform(const SinglePassRange1& rng,
                         OutputIterator out,
                         UnaryOperation fun);

template<
    class SinglePassRange1,
    class SinglePassRange2,
    class OutputIterator,
    class BinaryOperation
>
OutputIterator transform(const SinglePassRange1& rng1,
                         const SinglePassRange2& rng2,
                         OutputIterator out,
                         BinaryOperation fun);
```

Description

UnaryOperation version:

`transform` assigns the value `y` to each element `[out, out + distance(rng)), y = fun(x)` where `x` is the corresponding value to `y` in `rng1`. The return value is `out + distance(rng)`.

BinaryOperation version:

`transform` assigns the value `z` to each element `[out, out + min(distance(rng1), distance(rng2))), z = fun(x,y)` where `x` is the corresponding value in `rng1` and `y` is the corresponding value in `rng2`. This version of `transform` stops upon reaching either the end of `rng1`, or the end of `rng2`. Hence there isn't a requirement for `distance(rng1) == distance(rng2)` since there is a safe guaranteed behaviour, unlike with the iterator counterpart in the standard library.

The return value is `out + min(distance(rng1), distance(rng2))`.

Definition

Defined in the header file `boost/range/algorithm/transform.hpp`

Requirements

For the unary versions of transform:

- `SinglePassRange1` is a model of the Single Pass Range Concept.
- `OutputIterator` is a model of the `OutputIteratorConcept`.
- `UnaryOperation` is a model of the `UnaryFunctionConcept`.
- `SinglePassRange1`'s value type must be convertible to `UnaryFunction`'s argument type.
- `UnaryFunction`'s result type must be convertible to a type in `OutputIterator`'s set of value types.

For the binary versions of transform:

- `SinglePassRange1` is a model of the Single Pass Range Concept.
- `SinglePassRange2` is a model of the Single Pass Range Concept.
- `OutputIterator` is a model of the `OutputIteratorConcept`.
- `BinaryOperation` is a model of the `BinaryFunctionConcept`.
- `SinglePassRange1`'s value type must be convertible to `BinaryFunction`'s first argument type.
- `SinglePassRange2`'s value type must be convertible to `BinaryFunction`'s second argument type.
- `BinaryOperation`'s result type must be convertible to a type in `OutputIterator`'s set of value types.

Precondition:

For the unary version of transform:

- `out` is not an iterator within the range `[begin(rng1) + 1, end(rng1))`.
- `[out, out + distance(rng1))` is a valid range.

For the binary version of transform:

- `out` is not an iterator within the range `[begin(rng1) + 1, end(rng1))`.
- `out` is not an iterator within the range `[begin(rng2) + 1, end(rng2))`.
- `[out, out + min(distance(rng1), distance(rng2)))` is a valid range.

Complexity

Linear. The operation is applied exactly `distance(rng1)` for the unary version and `min(distance(rng1), distance(rng2))` for the binary version.

unique

Prototype

```
template<class ForwardRange>
typename range_return<ForwardRange, return_begin_found>::type
unique(ForwardRange& rng);

template<class ForwardRange>
typename range_return<const ForwardRange, return_begin_found>::type
unique(const ForwardRange& rng);

template<class ForwardRange, class BinaryPredicate>
typename range_return<ForwardRange, return_begin_found>::type
unique(ForwardRange& rng, BinaryPredicate pred);

template<class ForwardRange, class BinaryPredicate>
typename range_return<const ForwardRange, return_begin_found>::type
unique(const ForwardRange& rng, BinaryPredicate pred);

template<range_return_value re, class ForwardRange>
typename range_return<ForwardRange, re>::type
unique(ForwardRange& rng);

template<range_return_value re, class ForwardRange>
typename range_return<const ForwardRange, re>::type
unique(const ForwardRange& rng);

template<range_return_value re, class ForwardRange, class BinaryPredicate>
typename range_return<ForwardRange, re>::type
unique(ForwardRange& rng, BinaryPredicate pred);

template<range_return_value re, class ForwardRange, class BinaryPredicate>
typename range_return<const ForwardRange, re>::type
unique(const ForwardRange& rng, BinaryPredicate pred);
```

Description

`unique` removes all but the first element of each sequence of duplicate encountered in `rng`.

Elements in the range `[new_last, end(rng))` are dereferenceable but undefined.

Equality is determined by the predicate if one is supplied, or by `operator==()` for `ForwardRange`'s value type.

Definition

Defined in the header file `boost/range/algorithm/unique.hpp`

Requirements

For the non-predicate versions of unique:

- `ForwardRange` is a model of the Forward Range Concept.
- `ForwardRange` is mutable.
- `ForwardRange`'s value type is a model of the `EqualityComparableConcept`.

For the predicate versions of unique:

- `ForwardRange` is a model of the Forward Range Concept.

- `ForwardRange` is mutable.
- `BinaryPredicate` is a model of the `BinaryPredicateConcept`.
- `ForwardRange`'s value type is convertible to `BinaryPredicate`'s first argument type and to `BinaryPredicate`'s second argument type.

Complexity

Linear. `O(N)` where `N` is `distance(rng)`. Exactly `distance(rng)` comparisons are performed.

unique_copy

Prototype

```
template<class SinglePassRange, class OutputIterator>
OutputIterator unique_copy(const SinglePassRange& rng, OutputIterator out);

template<class SinglePassRange, class OutputIterator, class BinaryPredicate>
OutputIterator unique_copy(const SinglePassRange& rng, OutputIterator out, BinaryPredicate pred);
```

Description

`unique_copy` copies the first element of each sequence of duplicates encountered in `rng` to `out`.

Equality is determined by the predicate if one is supplied, or by `operator==()` for `SinglePassRange`'s value type.

Definition

Defined in the header file `boost/range/algorithm/unique_copy.hpp`

Requirements

For the non-predicate versions of unique:

- `SinglePassRange` is a model of the Single Pass Range Concept.
- `SinglePassRange` is mutable.
- `SinglePassRange`'s value type is a model of the `EqualityComparableConcept`.
- `OutputIterator` is a model of the `OutputIteratorConcept`.

For the predicate versions of unique:

- `SinglePassRange` is a model of the Single Pass Range Concept.
- `SinglePassRange` is mutable.
- `BinaryPredicate` is a model of the `BinaryPredicateConcept`.
- `SinglePassRange`'s value type is convertible to `BinaryPredicate`'s first argument type and to `BinaryPredicate`'s second argument type.
- `OutputIterator` is a model of the `OutputIteratorConcept`.

Complexity

Linear. `O(N)` where `N` is `distance(rng)`. Exactly `distance(rng)` comparisons are performed.

Non-mutating algorithms

adjacent_find

Prototype

```
template<class ForwardRange>
typename range_iterator<ForwardRange>::type
adjacent_find(ForwardRange& rng);

template<class ForwardRange>
typename range_iterator<const ForwardRange>::type
adjacent_find(const ForwardRange& rng);

template<class ForwardRange, class BinaryPredicate>
typename range_iterator<ForwardRange>::type
adjacent_find(ForwardRange& rng, BinaryPred pred);

template<class ForwardRange, class BinaryPredicate>
typename range_iterator<const ForwardRange>::type
adjacent_find(const ForwardRange& rng, BinaryPred pred);

template<range_return_value_re, class ForwardRange>
typename range_return<ForwardRange, re>::type
adjacent_find(ForwardRange& rng);

template<range_return_value_re, class ForwardRange>
typename range_return<const ForwardRange, re>::type
adjacent_find(const ForwardRange& rng);

template<
    range_return_value re,
    class ForwardRange,
    class BinaryPredicate
    >
typename range_return<ForwardRange, re>::type
adjacent_find(ForwardRange& rng, BinaryPredicate pred);

template<
    range_return_value re,
    class ForwardRange,
    class BinaryPredicate
    >
typename range_return<const ForwardRange, re>::type
adjacent_find(const ForwardRange& rng, BinaryPredicate pred);
```

Description

Non-predicate versions:

`adjacent_find` finds the first adjacent elements `[x,y]` in `rng` where `x == y`

Predicate versions:

`adjacent_find` finds the first adjacent elements `[x,y]` in `rng` where `pred(x,y)` is `true`.

Definition

Defined in the header file `boost/range/algorithm/adjacent_find.hpp`

Requirements

For the non-predicate versions of adjacent_find:

- `ForwardRange` is a model of the Forward Range Concept.
- `ForwardRange`'s value type is a model of the `EqualityComparableConcept`.

For the predicate versions of adjacent_find:

- `ForwardRange` is a model of the Forward Range Concept.
- `BinaryPredicate` is a model of the `BinaryPredicateConcept`.
- `ForwardRange`'s value type is convertible to `BinaryPredicate`'s first argument type and to `BinaryPredicate`'s second argument type.

Complexity

Linear. If `empty(rng)` then no comparisons are performed; otherwise, at most `distance(rng) - 1` comparisons.

binary_search

Prototype

```
template<class ForwardRange, class Value>
bool binary_search(const ForwardRange& rng, const Value& val);

template<class ForwardRange, class Value, class BinaryPredicate>
bool binary_search(const ForwardRange& rng, const Value& val, BinaryPredicate pred);
```

Description

`binary_search` returns `true` if and only if the value `val` exists in the range `rng`.

Definition

Defined in the header file `boost/range/algorithm/binary_search.hpp`

Requirements

For the non-predicate versions of binary_search:

- `ForwardRange` is a model of the Forward Range Concept.
- `Value` is a model of the `LessThanComparableConcept`.
- The ordering of objects of type `Value` is a **strict weak ordering**, as defined in the `LessThanComparableConcept` requirements.
- `ForwardRange`'s value type is the same type as `Value`.

For the predicate versions of binary_search:

- `ForwardRange` is a model of the Forward Range Concept.
- `BinaryPredicate` is a model of the `StrictWeakOrderingConcept`.
- `ForwardRange`'s value type is the same type as `Value`.
- `ForwardRange`'s value type is convertible to `BinaryPredicate`'s argument type.

Precondition:

For the non-predicate version:

`rng` is ordered in ascending order according to `operator<`.

For the predicate version:

`rng` is ordered in ascending order according to the function object `pred`.

Complexity

For non-random-access ranges, the complexity is `O(N)` where `N` is `distance(rng)`.

For random-access ranges, the complexity is `O(log N)` where `N` is `distance(rng)`.

count

Prototype

```
template<class SinglePassRange, class Value>
typename range_difference<SinglePassRange>::type
count(SinglePassRange& rng, const Value& val);

template<class SinglePassRange, class Value>
typename range_difference<const SinglePassRange>::type
count(const SinglePassRange& rng, const Value& val);
```

Description

`count` returns the number of elements `x` in `rng` where `x == val` is `true`.

Definition

Defined in the header file `boost/range/algorithm/count.hpp`

Requirements

- `SinglePassRange` is a model of the Single Pass Range Concept.
- `Value` is a model of the `EqualityComparableConcept`.
- `SinglePassRange`'s value type is a model of the `EqualityComparableConcept`.
- An object of `SinglePassRange`'s value type can be compared for equality with an object of type `Value`.

Complexity

Linear. Exactly `distance(rng)` comparisons.

count_if

Prototype

```
template<class SinglePassRange, class UnaryPredicate>
typename range_difference<const SinglePassRange>::type
count_if(const SinglePassRange& rng, UnaryPredicate pred);
```

Description

`count_if` returns the number of elements `x` in `rng` where `pred(x)` is `true`.

Definition

Defined in the header file `boost/range/algorithm/count_if.hpp`

Requirements

- `SinglePassRange` is a model of the Single Pass Range Concept.
- `UnaryPredicate` is a model of the `UnaryPredicateConcept`.
- `SinglePassRange`'s value type is a model of the `EqualityComparableConcept`.
- The value type of `SinglePassRange` is convertible to the argument type of `UnaryPredicate`.

Complexity

Linear. Exactly `distance(rng)` invocations of `pred`.

equal

Prototype

```
template<
    class SinglePassRange1,
    class SinglePassRange2
>
bool equal(const SinglePassRange1& rng1,
           const SinglePassRange2& rng2);

template<
    class SinglePassRange1,
    class SinglePassRange2,
    class BinaryPredicate
>
bool equal(const SinglePassRange1& rng1,
           const SinglePassRange2& rng2,
           BinaryPredicate         pred);
```

Description

`equal` returns `true` if `distance(rng1)` is equal to the `distance(rng2)` and for each element `x` in `rng1`, the corresponding element `y` in `rng2` is equal. Otherwise `false` is returned.

In this range version of `equal` it is perfectly acceptable to pass in two ranges of unequal lengths.

Elements are considered equal in the non-predicate version if `operator==` returns `true`. Elements are considered equal in the predicate version if `pred(x,y)` is `true`.

Definition

Defined in the header file `boost/range/algorithm/equal.hpp`

Requirements

For the non-predicate versions:

- `SinglePassRange1` is a model of the Single Pass Range Concept.
- `SinglePassRange2` is a model of the Single Pass Range Concept.
- `SinglePassRange1`'s value type is a model of the `EqualityComparableConcept`.

- `SinglePassRange2`'s value type is a model of the `EqualityComparableConcept`.
- `SinglePassRange1`'s value type can be compared for equality with `SinglePassRange2`'s value type.

For the predicate versions:

- `SinglePassRange1` is a model of the Single Pass Range Concept.
- `SinglePassRange2` is a model of the Single Pass Range Concept.
- `BinaryPredicate` is a model of the `BinaryPredicateConcept`.
- `SinglePassRange1`'s value type is convertible to `BinaryPredicate`'s first argument type.
- `SinglePassRange2`'s value type is convertible to `BinaryPredicate`'s second argument type.

Complexity

Linear. At most `min(distance(rng1), distance(rng2))` comparisons.

equal_range

Prototype

```
template<
    class ForwardRange,
    class Value
    >
std::pair<typename range_iterator<ForwardRange>::type,
          typename range_iterator<ForwardRange>::type>
equal_range(ForwardRange& rng, const Value& val);

template<
    class ForwardRange,
    class Value
    >
std::pair<typename range_iterator<const ForwardRange>::type,
          typename range_iterator<const ForwardRange>::type>
equal_range(const ForwardRange& rng, const Value& val);

template<
    class ForwardRange,
    class Value,
    class SortPredicate
    >
std::pair<typename range_iterator<ForwardRange>::type,
          typename range_iterator<ForwardRange>::type>
equal_range(ForwardRange& rng, const Value& val, SortPredicate pred);

template<
    class ForwardRange,
    class Value,
    class SortPredicate
    >
std::pair<typename range_iterator<const ForwardRange>::type,
          typename range_iterator<const ForwardRange>::type>
equal_range(const ForwardRange& rng, const Value& val, SortPredicate pred);
```

Description

`equal_range` returns a range in the form of a pair of iterators where all of the elements are equal to `val`. If no values are found that are equal to `val`, then an empty range is returned, hence `result.first == result.second`. For the non-predicate versions of equal_range the equality of elements is determined by operator<. For the predicate versions of equal_range the equality of elements is determined by pred.

Definition

Defined in the header file boost/range/algorithm/equal_range.hpp

Requirements

For the non-predicate versions:

- ForwardRange is a model of the Forward Range Concept.
- Value is a model of the LessThanComparableConcept.
- The ordering of objects of type Value is a **strict weak ordering**, as defined in the LessThanComparableConcept requirements.
- ForwardRange's value type is the same type as Value.

For the predicate versions:

- ForwardRange is a model of the Forward Range Concept.
- SortPredicate is a model of the StrictWeakOrderingConcept.
- ForwardRange's value type is the same as Value.
- ForwardRange's value type is convertible to both of SortPredicate's argument types.

Precondition:

For the non-predicate versions: rng is ordered in ascending order according to operator<.

For the predicate versions: rng is ordered in ascending order according to pred.

Complexity

For random-access ranges, the complexity is O(log N), otherwise the complexity is O(N).

for_each

Prototype

```
template<
    class SinglePassRange,
    class UnaryFunction
    >
UnaryFunction for_each(SinglePassRange& rng, UnaryFunction fun);

template<
    class SinglePassRange,
    class UnaryFunction
    >
UnaryFunction for_each(const SinglePassRange& rng, UnaryFunction fun);
```

Description

for_each traverses forward through rng and for each element x it invokes fun(x).

Definition

Defined in the header file boost/range/algorithm/for_each.hpp

Requirements

- `SinglePassRange` is a model of the Single Pass Range Concept.
- `UnaryFunction` is a model of the `UnaryFunctionConcept`.
- `UnaryFunction` does not apply any non-constant operation through its argument.
- `SinglePassRange`'s value type is convertible to `UnaryFunction`'s argument type.

Complexity

Linear. Exactly `distance(rng)` applications of `UnaryFunction`.

find

Prototype

```
template<class SinglePassRange, class Value>
typename range_iterator<SinglePassRange>::type
find(SinglePassRange& rng, Value val);

template<
    range_return_value re,
    class SinglePassRange,
    class Value
    >
typename range_return<SinglePassRange, re>::type
find(SinglePassRange& rng, Value val);
```

Description

The versions of `find` that return an iterator, returns the first iterator in the range `rng` such that `*i == value`. `end(rng)` is returned if no such iterator exists. The versions of find that return a `range_return`, defines `found` in the same manner as the returned iterator described above.

Definition

Defined in the header file `boost/range/algorithm/find.hpp`

Requirements

- `SinglePassRange` is a model of the Single Pass Range Concept.
- `Value` is a model of the `EqualityComparableConcept`.
- The `operator==` is defined for type `Value` to be compared with the `SinglePassRange`'s value type.

Complexity

Linear. At most `distance(rng)` comparisons for equality.

find_end

Prototype

```
template<class ForwardRange1, class ForwardRange2>
typename range_iterator<ForwardRange1>::type
find_end(ForwardRange1& rng1, const ForwardRange2& rng2);

template<
    class ForwardRange1,
    class ForwardRange2,
    class BinaryPredicate
    >
typename range_iterator<ForwardRange1>::type
find_end(ForwardRange1& rng1, const ForwardRange2& rng2, BinaryPredicate pred);

template<
    range_return_value re,
    class ForwardRange1,
    class ForwardRange2
    >
typename range_return<ForwardRange1, re>::type
find_end(ForwardRange1& rng1, const ForwardRange2& rng2);

template<
    range_return_value re,
    class ForwardRange1,
    class ForwardRange2,
    class BinaryPredicate
    >
typename range_return<ForwardRange1, re>::type
find_end(ForwardRange1& rng1, const ForwardRange2& rng2, BinaryPredicate pred);
```

Description

The versions of `find_end` that return an iterator, return an iterator to the beginning of the last sub-sequence equal to `rng2` within `rng1`. Equality is determined by `operator==` for non-predicate versions of `find_end`, and by satisfying `pred` in the predicate versions. The versions of `find_end` that return a `range_return`, defines `found` in the same manner as the returned iterator described above.

Definition

Defined in the header file `boost/range/algorithm/find_end.hpp`

Requirements

For the non-predicate versions:

- `ForwardRange1` is a model of the Forward Range Concept.
- `ForwardRange2` is a model of the Forward Range Concept.
- `ForwardRange1`'s value type is a model of the `EqualityComparableConcept`.
- `ForwardRange2`'s value type is a model of the `EqualityComparableConcept`.
- Objects of `ForwardRange1`'s value type can be compared for equality with objects of `ForwardRange2`'s value type.

For the predicate versions:

- `ForwardRange1` is a model of the Forward Range Concept.

- `ForwardRange2` is a model of the Forward Range Concept.
- `BinaryPredicate` is a model of the `BinaryPredicateConcept`.
- `ForwardRange1`'s value type is convertible to `BinaryPredicate`'s first argument type.
- `ForwardRange2`'s value type is convertible to `BinaryPredicate`'s second argument type.

Complexity

The number of comparisons is proportional to `distance(rng1) * distance(rng2)`. If both `ForwardRange1` and `ForwardRange2` are models of `BidirectionalRangeConcept` then the average complexity is linear and the worst case is `distance(rng1) * distance(rng2)`.

find_first_of

Prototype

```
template<class SinglePassRange1, class ForwardRange2>
typename range_iterator<SinglePassRange1>::type
find_first_of(SinglePassRange1& rng1, const ForwardRange2& rng2);

template<
    class SinglePassRange1,
    class ForwardRange2,
    class BinaryPredicate
    >
typename range_iterator<SinglePassRange1>::type
find_first_of(SinglePassRange1& rng1, const ForwardRange2& rng2, BinaryPredicate pred);

template<
    range_return_value re,
    class SinglePassRange1,
    class ForwardRange2
    >
typename range_return<SinglePassRange1, re>::type
find_first_of(SinglePassRange1& rng1, const ForwardRange2& rng2);

template<
    range_return_value re,
    class SinglePassRange1,
    class ForwardRange2,
    class BinaryPredicate
    >
typename range_return<SinglePassRange1, re>::type
find_first_of(SinglePassRange1& rng1, const ForwardRange2& rng2, BinaryPredicate pred);
```

Description

The versions of `find_first_of` that return an iterator, return an iterator to the first occurrence in `rng1` of any of the elements in `rng2`. Equality is determined by `operator==` for non-predicate versions of `find_first_of`, and by satisfying `pred` in the predicate versions.

The versions of `find_first_of` that return a `range_return`, defines `found` in the same manner as the returned iterator described above.

Definition

Defined in the header file `boost/range/algorithm/find_first_of.hpp`

Requirements

For the non-predicate versions:

- `SinglePassRange1` is a model of the Single Pass Range Concept.
- `ForwardRange2` is a model of the Forward Range Concept.
- `SinglePassRange1`'s value type is a model of the `EqualityComparableConcept`, and can be compared for equality with `ForwardRange2`'s value type.

For the predicate versions:

- `SinglePassRange1` is a model of the Single Pass Range Concept.
- `ForwardRange2` is a model of the Forward Range Concept.
- `BinaryPredicate` is a model of the `BinaryPredicateConcept`.
- `SinglePassRange1`'s value type is convertible to `BinaryPredicate`'s first argument type.
- `ForwardRange2`'s value type is convertible to `BinaryPredicate`'s second argument type.

Complexity

At most `distance(rng1) * distance(rng2)` comparisons.

find_if

Prototype

```
template<class SinglePassRange, class UnaryPredicate>
typename range_iterator<SinglePassRange>::type
find_if(SinglePassRange& rng, UnaryPredicate pred);

template<
    range_return_value re,
    class SinglePassRange,
    class UnaryPredicate
    >
typename range_return<SinglePassRange, re>::type
find_if(SinglePassRange& rng, UnaryPredicate pred);
```

Description

The versions of `find_if` that return an iterator, returns the first iterator in the range `rng` such that `pred(*i)` is `true`. `end(rng)` is returned if no such iterator exists.

The versions of `find_if` that return a `range_return`, defines found in the same manner as the returned iterator described above.

Definition

Defined in the header file `boost/range/algorithm/find_if.hpp`

Requirements

- `SinglePassRange` is a model of the Single Pass Range Concept.
- `UnaryPredicate` is a model of the `PredicateConcept`.
- The value type of `SinglePassRange` is convertible to the argument type of `UnaryPredicate`.

Precondition:

For each iterator `i` in `rng`, `*i` is in the domain of `UnaryPredicate`.

Complexity

Linear. At most `distance(rng)` invocations of `pred`.

lexicographical_compare

Prototype

```
template<
    class SinglePassRange1,
    class SinglePassRange2
    >
bool lexicographical_compare(const SinglePassRange1& rng1,
                             const SinglePassRange2& rng2);

template<
    class SinglePassRange1,
    class SinglePassRange2,
    class BinaryPredicate
    >
bool lexicographical_compare(const SinglePassRange1& rng1,
                             const SinglePassRange2& rng2,
                             BinaryPredicate pred);
```

Description

`lexicographical_compare` compares element by element `rng1` against `rng2`. If the element from `rng1` is less than the element from `rng2` then `true` is returned. If the end of `rng1` without reaching the end of `rng2` this also causes the return value to be `true`. The return value is `false` in all other circumstances. The elements are compared using `operator<` in the non-predicate versions of `lexicographical_compare` and using `pred` in the predicate versions.

Definition

Defined in the header file `boost/range/algorithm/lexicographical_compare.hpp`

Requirements

For the non-predicate versions of lexicographical_compare:

- `SinglePassRange1` is a model of the Single Pass Range Concept.
- `SinglePassRange2` is a model of the Single Pass Range Concept.
- `SinglePassRange1`'s value type is a model of the `LessThanComparableConcept`.
- `SinglePassRange2`'s value type is a model of the `LessThanComparableConcept`.
- Let `x` be an object of `SinglePassRange1`'s value type. Let `y` be an object of `SinglePassRange2`'s value type. `x < y` must be valid. `y < x` must be valid.

For the predicate versions of lexicographical_compare:

- `SinglePassRange1` is a model of the Single Pass Range Concept.
- `SinglePassRange2` is a model of the Single Pass Range Concept.
- `BinaryPredicate` is a model of the `BinaryPredicateConcept`.

- `SinglePassRange1`'s value type is convertible to `BinaryPredicate`'s first argument type.
- `SinglePassRange2`'s value type is convertible to `BinaryPredicate`'s second argument type.

Complexity

Linear. At most `2 * min(distance(rng1), distance(rng2))` comparisons.

lower_bound

Prototype

```
template<
    class ForwardRange,
    class Value
    >
typename range_iterator<ForwardRange>::type
lower_bound(ForwardRange& rng, Value val);

template<
    range_return_value re,
    class ForwardRange,
    class Value
    >
typename range_return<ForwardRange, re>::type
lower_bound(ForwardRange& rng, Value val);

template<
    class ForwardRange,
    class Value,
    class SortPredicate
    >
typename range_iterator<ForwardRange>::type
lower_bound(ForwardRange& rng, Value val, SortPredicate pred);

template<
    range_return_value re,
    class ForwardRange,
    class Value,
    class SortPredicate
    >
typename range_return<ForwardRange,re>::type
lower_bound(ForwardRange& rng, Value val, SortPredicate pred);
```

Description

The versions of `lower_bound` that return an iterator, returns the first iterator in the range `rng` such that: without predicate - `*i < value` is `false`, with predicate - `pred(*i, value)` is `false`.

`end(rng)` is returned if no such iterator exists.

The versions of `lower_bound` that return a `range_return`, defines `found` in the same manner as the returned iterator described above.

Definition

Defined in the header file `boost/range/algorithm/lower_bound.hpp`

Requirements

For the non-predicate versions:

- `ForwardRange` is a model of the Forward Range Concept.
- `Value` is a model of the `LessThanComparableConcept`.
- The ordering of objects of type `Value` is a **strict weak ordering**, as defined in the `LessThanComparableConcept` requirements.
- `ForwardRange`'s value type is the same type as `Value`.

For the predicate versions:

- `ForwardRange` is a model of the Forward Range Concept.
- `BinaryPredicate` is a model of the `StrictWeakOrderingConcept`.
- `ForwardRange`'s value type is the same type as `Value`.
- `ForwardRange`'s value type is convertible to both of `BinaryPredicate`'s argument types.

Precondition:

For the non-predicate versions:

`rng` is sorted in ascending order according to `operator<`.

For the predicate versions:

`rng` is sorted in ascending order according to `pred`.

Complexity

For ranges that model the Random Access Range concept the complexity is `O(log N)`, where `N` is `distance(rng)`.

For all other range types the complexity is `O(N)`.

max_element

Prototype

```
template<class ForwardRange>
typename range_iterator<ForwardRange>::type
max_element(ForwardRange& rng);

template<class ForwardRange>
typename range_iterator<const ForwardRange>::type
max_element(const ForwardRange& rng);

template<class ForwardRange, class BinaryPredicate>
typename range_iterator<ForwardRange>::type
max_element(ForwardRange& rng, BinaryPredicate pred);

template<class ForwardRange, class BinaryPredicate>
typename range_iterator<const ForwardRange>::type
max_element(const ForwardRange& rng, BinaryPredicate pred);


template<
    range_return_value re,
    class ForwardRange
    >
typename range_return<ForwardRange, re>::type
max_element(ForwardRange& rng);

template<
    range_return_value_re,
    class ForwardRange
    >
typename range_return<const ForwardRange, re>::type
max_element(const ForwardRange& rng);

template<
    range_return_value re,
    class ForwardRange,
    class BinaryPredicate
    >
typename range_return<ForwardRange, re>::type
max_element(ForwardRange& rng, BinaryPredicate pred);

template<
    range_return_value re,
    class ForwardRange,
    class BinaryPredicate
    >
typename range_return<const ForwardRange, re>::type
max_element(const ForwardRange& rng, BinaryPredicate pred);
```

Description

The versions of `max_element` that return an iterator, return the iterator to the maximum value as determined by using `operator<` if a predicate is not supplied. Otherwise the predicate `pred` is used to determine the maximum value. The versions of `max_element` that return a `range_return`, defines `found` in the same manner as the returned iterator described above.

Definition

Defined in the header file `boost/range/algorithm/max_element.hpp`

Requirements

For the non-predicate versions:

- `ForwardRange` is a model of the Forward Range Concept.
- `ForwardRange`'s value type is a model of the `LessThanComparableConcept`.

For the predicate versions:

- `ForwardRange` is a model of the Forward Range Concept.
- `BinaryPredicate` is a model of the `BinaryPredicateConcept`.
- `ForwardRange`'s value type is convertible to both of `BinaryPredicate`'s argument types.

Complexity

Linear. Zero comparisons if `empty(rng)`, otherwise `distance(rng) - 1` comparisons.

min_element

Prototype

```
template<class ForwardRange>
typename range_iterator<ForwardRange>::type
min_element(ForwardRange& rng);

template<class ForwardRange>
typename range_iterator<const ForwardRange>::type
min_element(const ForwardRange& rng);

template<class ForwardRange, class BinaryPredicate>
typename range_iterator<ForwardRange>::type
min_element(ForwardRange& rng, BinaryPredicate pred);

template<class ForwardRange, class BinaryPredicate>
typename range_iterator<const ForwardRange>::type
min_element(const ForwardRange& rng, BinaryPredicate pred);


template<
    range_return_value re,
    class ForwardRange
    >
typename range_return<ForwardRange, re>::type
min_element(ForwardRange& rng);

template<
    range_return_value_re,
    class ForwardRange
    >
typename range_return<const ForwardRange, re>::type
min_element(const ForwardRange& rng);

template<
    range_return_value re,
    class ForwardRange,
    class BinaryPredicate
    >
typename range_return<ForwardRange, re>::type
min_element(ForwardRange& rng, BinaryPredicate pred);

template<
    range_return_value re,
    class ForwardRange,
    class BinaryPredicate
    >
typename range_return<const ForwardRange, re>::type
min_element(const ForwardRange& rng, BinaryPredicate pred);
```

Description

The versions of `min_element` that return an iterator, return the iterator to the minimum value as determined by using `operator<` if a predicate is not supplied. Otherwise the predicate `pred` is used to determine the minimum value. The versions of `min_element` that return a `range_return`, defines `found` in the same manner as the returned iterator described above.

Definition

Defined in the header file `boost/range/algorithm/min_element.hpp`

Requirements

For the non-predicate versions:

- `ForwardRange` is a model of the Forward Range Concept.
- `ForwardRange`'s value type is a model of the `LessThanComparableConcept`.

For the predicate versions:

- `ForwardRange` is a model of the Forward Range Concept.
- `BinaryPredicate` is a model of the `BinaryPredicateConcept`.
- `ForwardRange`'s value type is convertible to both of `BinaryPredicate`'s argument types.

Complexity

Linear. Zero comparisons if `empty(rng)`, otherwise `distance(rng) - 1` comparisons.

mismatch

Prototype

```
template<class SinglePassRange1, class SinglePassRange2>
std::pair<
    typename range_iterator<SinglePassRange1>::type,
    typename range_iterator<const SinglePassRange2>::type >
mismatch(SinglePassRange1& rng1, const SinglePassRange2& rng2);

template<class SinglePassRange1, class SinglePassRange2>
std::pair<
    typename range_iterator<const SinglePassRange1>::type,
    typename range_iterator<const SinglePassRange2>::type >
mismatch(const SinglePassRange1& rng1, const SinglePassRange2& rng2);

template<class SinglePassRange1, class SinglePassRange2>
std::pair<
    typename range_iterator<SinglePassRange1>::type,
    typename range_iterator<SinglePassRange2>::type >
mismatch(SinglePassRange1& rng1, SinglePassRange2& rng2);

template<class SinglePassRange1, class SinglePassRange2>
std::pair<
    typename range_iterator<const SinglePassRange1>::type,
    typename range_iterator<SinglePassRange2>::type >
mismatch(const SinglePassRange1& rng1, SinglePassRange2& rng2);


template<
    class SinglePassRange1,
    class SinglePassRange2,
    class BinaryPredicate
    >
std::pair<
    typename range_iterator<SinglePassRange1>::type,
    typename range_iterator<const SinglePassRange2>::type >
mismatch(SinglePassRange1& rng1, const SinglePassRange2& rng2,
         BinaryPredicate pred);

template<
    class SinglePassRange1,
    class SinglePassRange2,
    class BinaryPredicate
    >
std::pair<
    typename range_iterator<const SinglePassRange1>::type,
    typename range_iterator<const SinglePassRange2>::type >
mismatch(const SinglePassRange1& rng1, const SinglePassRange2& rng2,
         BinaryPredicate pred);

template<
    class SinglePassRange1,
    class SinglePassRange2,
    class BinaryPredicate
    >
std::pair<
    typename range_iterator<SinglePassRange1>::type,
    typename range_iterator<SinglePassRange2>::type >
mismatch(SinglePassRange1& rng1, SinglePassRange2& rng2,
         BinaryPredicate pred);

template<
```

```
    class SinglePassRange1,
    class SinglePassRange2,
    class BinaryPredicate
    >
std::pair<
    typename range_iterator<const SinglePassRange1>::type,
    typename range_iterator<SinglePassRange2>::type >
mismatch(const SinglePassRange1& rng1, SinglePassRange2& rng2,
         BinaryPredicate pred);
```

Description

`mismatch` finds the first position where the corresponding elements from the two ranges `rng1` and `rng2` are not equal.

Equality is determined by `operator==` for non-predicate versions of `mismatch`, and by satisfying `pred` in the predicate versions.

Definition

Defined in the header file `boost/range/algorithm/mismatch.hpp`

Requirements

For the non-predicate versions:

- `SinglePassRange1` is a model of the Single Pass Range Concept.
- `SinglePassRange2` is a model of the Single Pass Range Concept.
- `SinglePassRange1`'s value type is a model of the `EqualityComparableConcept`.
- `SinglePassRange2`'s value type is a model of the `EqualityComparableConcept`.
- `SinglePassRange1`s value type can be compared for equality with `SinglePassRange2`'s value type.

For the predicate versions:

- `SinglePassRange1` is a model of the Single Pass Range Concept.
- `SinglePassRange2` is a model of the Single Pass Range Concept.
- `BinaryPredicate` is a model of the `BinaryPredicateConcept`.
- `SinglePassRange1`'s value type is convertible to `BinaryPredicate`'s first argument type.
- `SinglePassRange2`'s value type is convertible to `BinaryPredicate`'s second argument type.

Precondition:

`distance(rng2) >= distance(rng1)`

Complexity

Linear. At most `distance(rng1)` comparisons.

search

Prototype

```
template<class ForwardRange1, class ForwardRange2>
typename range_iterator<ForwardRange1>::type
search(ForwardRange1& rng1, const ForwardRange2& rng2);

template<class ForwardRange1, class ForwardRange2>
typename range_iterator<const ForwardRange1>::type
search(const ForwardRange1& rng1, const ForwardRange2& rng2);

template<
    class ForwardRange1,
    class ForwardRange2,
    class BinaryPredicate
    >
typename range_iterator<ForwardRange1>::type,
search(ForwardRange1& rng1, const ForwardRange2& rng2, BinaryPredicate pred);

template<
    class ForwardRange1,
    class ForwardRange2,
    class BinaryPredicate
    >
typename range_iterator<const ForwardRange1>::type
search(const ForwardRange1& rng1, ForwardRange2& rng2, BinaryPredicate pred);


template<
    range_return_value re,
    class ForwardRange1,
    class ForwardRange2
    >
typename range_return<ForwardRange1, re>::type
search(ForwardRange1& rng1, const ForwardRange2& rng2);

template<
    range_return_value re,
    class ForwardRange1,
    class ForwardRange2
    >
typename range_return<const ForwardRange1, re>::type
search(const ForwardRange1& rng1, const ForwardRange2& rng2);

template<
    range_return_value re,
    class ForwardRange1,
    class ForwardRange2,
    class BinaryPredicate
    >
typename range_return<ForwardRange1, re>::type,
search(ForwardRange1& rng1, const ForwardRange2& rng2, BinaryPredicate pred);

template<
    range_return_value re,
    class ForwardRange1,
    class ForwardRange2,
    class BinaryPredicate
    >
typename range_return<const ForwardRange1, re>::type
search(const ForwardRange1& rng1, const ForwardRange2& rng2, BinaryPredicate pred);
```

Description

The versions of `search` that return an iterator, return an iterator to the start of the first subsequence in `rng1` that is equal to the subsequence `rng2`. The `end(rng1)` is returned if no such subsequence exists in `rng1`. Equality is determined by `operator==` for non-predicate versions of `search`, and by satisfying `pred` in the predicate versions.

The versions of `search` that return a `range_return`, defines `found` in the same manner as the returned iterator described above.

Definition

Defined in the header file `boost/range/algorithm/search.hpp`

Requirements

For the non-predicate versions:

- `ForwardRange1` is a model of the Forward Range Concept.
- `ForwardRange2` is a model of the Forward Range Concept.
- `ForwardRange1`'s value type is a model of the `EqualityComparableConcept`.
- `ForwardRange2`'s value type is a model of the `EqualityComparableConcept`.
- `ForwardRange1`s value type can be compared for equality with `ForwardRange2`'s value type.

For the predicate versions:

- `ForwardRange1` is a model of the Forward Range Concept.
- `ForwardRange2` is a model of the Forward Range Concept.
- `BinaryPredicate` is a model of the `BinaryPredicateConcept`.
- `ForwardRange1`'s value type is convertible to `BinaryPredicate`'s first argument type.
- `ForwardRange2`'s value type is convertible to `BinaryPredicate`'s second argument type.

Complexity

Average complexity is Linear. Worst-case complexity is quadratic.

search_n

Prototype

```
template<class ForwardRange, class Integer, class Value>
typename range_iterator<ForwardRange>::type
search_n(ForwardRange& rng, Integer n, const Value& value);

template<class ForwardRange, class Integer, class Value>
typename range_iterator<const ForwardRange>::type
search_n(const ForwardRange& rng, Integer n, const Value& value);

template<class ForwardRange, class Integer, class Value, class BinaryPredicate>
typename range_iterator<ForwardRange>::type
search_n(ForwardRange& rng, Integer n, const Value& value,
         BinaryPredicate binary_pred);

template<class ForwardRange, class Integer, class Value, class BinaryPredicate>
typename range_iterator<const ForwardRange>::type
search_n(const ForwardRange& rng, Integer n, const Value& value,
         BinaryPredicate binary_pred);
```

Description

`search_n` searches `rng` for a sequence of length `n` equal to `value` where equality is determined by operator== in the non-predicate case, and by a predicate when one is supplied.

Definition

Defined in the header file `boost/range/algorithm/search_n.hpp`

Requirements

For the non-predicate versions:

- `ForwardRange` is a model of the Forward Range Concept.
- `ForwardRange`'s value type is a model of the `EqualityComparableConcept`.
- `ForwardRange`s value type can be compared for equality with `Value`.
- `Integer` is a model of the `IntegerConcept`.

For the predicate versions:

- `ForwardRange` is a model of the Forward Range Concept.
- `BinaryPredicate` is a model of the `BinaryPredicateConcept`.
- `ForwardRange`'s value type is convertible to `BinaryPredicate`'s first argument type.
- `Value` is convertible to `BinaryPredicate`'s second argument type.
- `Integer` is a model of the `IntegerConcept`.

Complexity

Average complexity is Linear. Worst-case complexity is quadratic.

upper_bound

Prototype

```
template<
    class ForwardRange,
    class Value
    >
typename range_iterator<ForwardRange>::type
upper_bound(ForwardRange& rng, Value val);

template<
    range_return_value re,
    class ForwardRange,
    class Value
    >
typename range_return<ForwardRange, re>::type
upper_bound(ForwardRange& rng, Value val);

template<
    class ForwardRange,
    class Value,
    class SortPredicate
    >
typename range_iterator<ForwardRange>::type
upper_bound(ForwardRange& rng, Value val, SortPredicate pred);

template<
    range_return_value re,
    class ForwardRange,
    class Value,
    class SortPredicate
    >
typename range_return<ForwardRange,re>::type
upper_bound(ForwardRange& rng, Value val, SortPredicate pred);
```

Description

The versions of `upper_bound` that return an iterator, returns the first iterator in the range `rng` such that: without predicate - `val < *i` is `true`, with predicate - `pred(val, *i)` is `true`.

`end(rng)` is returned if no such iterator exists.

The versions of `upper_bound` that return a `range_return`, defines `found` in the same manner as the returned iterator described above.

Definition

Defined in the header file `boost/range/algorithm/upper_bound.hpp`

Requirements

For the non-predicate versions:

- `ForwardRange` is a model of the Forward Range Concept.
- `Value` is a model of the `LessThanComparableConcept`.
- The ordering of objects of type `Value` is a **strict weak ordering**, as defined in the `LessThanComparableConcept` requirements.
- `ForwardRange`'s value type is the same type as `Value`.

For the predicate versions:

- `ForwardRange` is a model of the Forward Range Concept.
- `BinaryPredicate` is a model of the `StrictWeakOrderingConcept`.
- `ForwardRange`'s value type is the same type as `Value`.
- `ForwardRange`'s value type is convertible to both of `BinaryPredicate`'s argument types.

Precondition:

For the non-predicate versions:

`rng` is sorted in ascending order according to `operator<`.

For the predicate versions:

`rng` is sorted in ascending order according to `pred`.

Complexity

For ranges that model the Random Access Range Concept the complexity is `O(log N)`, where `N` is `distance(rng)`. For all other range types the complexity is `O(N)`.

Set algorithms

includes

Prototype

```
template<class SinglePassRange1, class SinglePassRange2>
bool includes(const SinglePassRange1& rng1, const SinglePassRange2& rng2);

template<
    class SinglePassRange1,
    class SinglePassRange2,
    class BinaryPredicate
    >
bool includes(const SinglePassRange1& rng1, const SinglePassRange2& rng2,
              BinaryPredicate pred);
```

Description

`includes` returns `true` if and only if, for every element in `rng2`, an equivalent element is also present in `rng1`. The ordering relationship is determined by using `operator<` in the non-predicate versions, and by evaluating `pred` in the predicate versions.

Definition

Defined in the header file `boost/range/algorithm/set_algorithm.hpp`

Requirements

For the non-predicate versions:

- `SinglePassRange1` is a model of the Single Pass Range Concept.
- `SinglePassRange2` is a model of the Single Pass Range Concept.
- `SinglePassRange1` and `SinglePassRange2` have the same value type.

- `SinglePassRange1`'s value type is a model of the `LessThanComparableConcept`.
- `SinglePassRange2`'s value type is a model of the `LessThanComparableConcept`.
- The ordering of objects of type `SinglePassRange1`'s value type is a **strict weak ordering**, as defined in the `LessThanComparableConcept` requirements.
- The ordering of objects of type `SinglePassRange2`'s value type is a **strict weak ordering**, as defined in the `LessThanComparableConcept` requirements.

For the predicate versions:

- `SinglePassRange1` is a model of the Single Pass Range Concept.
- `SinglePassRange2` is a model of the Single Pass Range Concept.
- `SinglePassRange1` and `SinglePassRange2` have the same value type.
- `BinaryPredicate` is a model of the `StrictWeakOrderingConcept`.
- `SinglePassRange1`'s value type is convertible to `BinaryPredicate`'s first argument type.
- `SinglePassRange2`'s value type is convertible to `BinaryPredicate`'s second argument types.

Precondition:

For the non-predicate versions:

`rng1` and `rng2` are sorted in ascending order according to `operator<`.

For the predicate versions:

`rng1` and `rng2` are sorted in ascending order according to `pred`.

Complexity

Linear. `O(N)`, where `N` is `distance(rng1) + distance(rng2)`.

set_union

Prototype

```
template<
    class SinglePassRange1,
    class SinglePassRange2,
    class OutputIterator
    >
OutputIterator set_union(const SinglePassRange1& rng1,
                         const SinglePassRange2& rng2,
                         OutputIterator          out);

template<
    class SinglePassRange1,
    class SinglePassRange2,
    class OutputIterator,
    class BinaryPredicate
    >
OutputIterator set_union(const SinglePassRange1& rng1,
                         const SinglePassRange2& rng2,
                         OutputIterator          out,
                         BinaryPredicate         pred);
```

Description

`set_union` constructs a sorted range that is the union of the sorted ranges `rng1` and `rng2`. The return value is the end of the output range. The ordering relationship is determined by using `operator<` in the non-predicate versions, and by evaluating `pred` in the predicate versions.

Definition

Defined in the header file `boost/range/algorithm/set_algorithm.hpp`

Requirements

For the non-predicate versions:

- `SinglePassRange1` is a model of the Single Pass Range Concept.
- `SinglePassRange2` is a model of the Single Pass Range Concept.
- `OutputIterator` is a model of the `OutputIteratorConcept`.
- `SinglePassRange1` and `SinglePassRange2` have the same value type.
- `SinglePassRange1`'s value type is a model of the `LessThanComparableConcept`.
- `SinglePassRange2`'s value type is a model of the `LessThanComparableConcept`.
- The ordering of objects of type `SinglePassRange1`'s value type is a **strict weak ordering**, as defined in the `LessThanComparableConcept` requirements.
- The ordering of objects of type `SinglePassRange2`'s value type is a **strict weak ordering**, as defined in the `LessThanComparableConcept` requirements.

For the predicate versions:

- `SinglePassRange1` is a model of the Single Pass Range Concept.
- `SinglePassRange2` is a model of the Single Pass Range Concept.
- `OutputIterator` is a model of the `OutputIteratorConcept`.
- `SinglePassRange1` and `SinglePassRange2` have the same value type.
- `BinaryPredicate` is a model of the `StrictWeakOrderingConcept`.
- `SinglePassRange1`'s value type is convertible to `BinaryPredicate`'s first argument type.
- `SinglePassRange2`'s value type is convertible to `BinaryPredicate`'s second argument types.

Precondition:

For the non-predicate versions:

`rng1` and `rng2` are sorted in ascending order according to `operator<`.

For the predicate versions:

`rng1` and `rng2` are sorted in ascending order according to `pred`.

Complexity

Linear. `O(N)`, where `N` is `distance(rng1) + distance(rng2)`.

set_intersection

Prototype

```
template<
    class SinglePassRange1,
    class SinglePassRange2,
    class OutputIterator
    >
OutputIterator set_intersection(const SinglePassRange1& rng1,
                                const SinglePassRange2& rng2,
                                OutputIterator          out);

template<
    class SinglePassRange1,
    class SinglePassRange2,
    class OutputIterator,
    class BinaryPredicate
    >
OutputIterator set_intersection(const SinglePassRange1& rng1,
                                const SinglePassRange2& rng2,
                                OutputIterator          out,
                                BinaryPredicate         pred);
```

Description

`set_intersection` constructs a sorted range that is the intersection of the sorted ranges `rng1` and `rng2`. The return value is the end of the output range.

The ordering relationship is determined by using `operator<` in the non-predicate versions, and by evaluating `pred` in the predicate versions.

Definition

Defined in the header file `boost/range/algorithm/set_algorithm.hpp`

Requirements

For the non-predicate versions:

- `SinglePassRange1` is a model of the Single Pass Range Concept.
- `SinglePassRange2` is a model of the Single Pass Range Concept.
- `OutputIterator` is a model of the `OutputIteratorConcept`.
- `SinglePassRange1` and `SinglePassRange2` have the same value type.
- `SinglePassRange1`'s value type is a model of the `LessThanComparableConcept`.
- `SinglePassRange2`'s value type is a model of the `LessThanComparableConcept`.
- The ordering of objects of type `SinglePassRange1`'s value type is a **strict weak ordering**, as defined in the `LessThanComparableConcept` requirements.
- The ordering of objects of type `SinglePassRange2`'s value type is a **strict weak ordering**, as defined in the `LessThanComparableConcept` requirements.

For the predicate versions:

- `SinglePassRange1` is a model of the Single Pass Range Concept.

- `SinglePassRange2` is a model of the Single Pass Range Concept.
- `OutputIterator` is a model of the `OutputIteratorConcept`.
- `SinglePassRange1` and `SinglePassRange2` have the same value type.
- `BinaryPredicate` is a model of the `StrictWeakOrderingConcept`.
- `SinglePassRange1`'s value type is convertible to `BinaryPredicate`'s first argument type.
- `SinglePassRange2`'s value type is convertible to `BinaryPredicate`'s second argument types.

Precondition:

For the non-predicate versions:

`rng1` and `rng2` are sorted in ascending order according to `operator<`.

For the predicate versions:

`rng1` and `rng2` are sorted in ascending order according to `pred`.

Complexity

Linear. `O(N)`, where `N` is `distance(rng1) + distance(rng2)`.

set_difference

Prototype

```
template<
    class SinglePassRange1,
    class SinglePassRange2,
    class OutputIterator
    >
OutputIterator set_difference(const SinglePassRange1& rng1,
                              const SinglePassRange2& rng2,
                              OutputIterator          out);

template<
    class SinglePassRange1,
    class SinglePassRange2,
    class OutputIterator,
    class BinaryPredicate
    >
OutputIterator set_difference(const SinglePassRange1& rng1,
                              const SinglePassRange2& rng2,
                              OutputIterator          out,
                              BinaryPredicate         pred);
```

Description

`set_difference` constructs a sorted range that is the set difference of the sorted ranges `rng1` and `rng2`. The return value is the end of the output range.

The ordering relationship is determined by using `operator<` in the non-predicate versions, and by evaluating `pred` in the predicate versions.

Definition

Defined in the header file `boost/range/algorithm/set_algorithm.hpp`

Requirements

For the non-predicate versions:

- `SinglePassRange1` is a model of the Single Pass Range Concept.
- `SinglePassRange2` is a model of the Single Pass Range Concept.
- `OutputIterator` is a model of the `OutputIteratorConcept`.
- `SinglePassRange1` and `SinglePassRange2` have the same value type.
- `SinglePassRange1`'s value type is a model of the `LessThanComparableConcept`.
- `SinglePassRange2`'s value type is a model of the `LessThanComparableConcept`.
- The ordering of objects of type `SinglePassRange1`'s value type is a **strict weak ordering**, as defined in the `LessThanComparableConcept` requirements.
- The ordering of objects of type `SinglePassRange2`'s value type is a **strict weak ordering**, as defined in the `LessThanComparableConcept` requirements.

For the predicate versions:

- `SinglePassRange1` is a model of the Single Pass Range Concept.
- `SinglePassRange2` is a model of the Single Pass Range Concept.
- `OutputIterator` is a model of the `OutputIteratorConcept`.
- `SinglePassRange1` and `SinglePassRange2` have the same value type.
- `BinaryPredicate` is a model of the `StrictWeakOrderingConcept`.
- `SinglePassRange1`'s value type is convertible to `BinaryPredicate`'s first argument type.
- `SinglePassRange2`'s value type is convertible to `BinaryPredicate`'s second argument types.

Precondition:

For the non-predicate versions:

`rng1` and `rng2` are sorted in ascending order according to `operator<`.

For the predicate versions:

`rng1` and `rng2` are sorted in ascending order according to `pred`.

Complexity

Linear. `O(N)`, where `N` is `distance(rng1) + distance(rng2)`.

set_symmetric_difference

Prototype

```
template<
    class SinglePassRange1,
    class SinglePassRange2,
    class OutputIterator
    >
OutputIterator
set_symmetric_difference(const SinglePassRange1& rng1,
                         const SinglePassRange2& rng2,
                         OutputIterator          out);

template<
    class SinglePassRange1,
    class SinglePassRange2,
    class OutputIterator,
    class BinaryPredicate
    >
OutputIterator
set_symmetric_difference(const SinglePassRange1& rng1,
                         const SinglePassRange2& rng2,
                         OutputIterator          out,
                         BinaryPredicate         pred);
```

Description

`set_symmetric_difference` constructs a sorted range that is the set symmetric difference of the sorted ranges `rng1` and `rng2`. The return value is the end of the output range.

The ordering relationship is determined by using `operator<` in the non-predicate versions, and by evaluating `pred` in the predicate versions.

Definition

Defined in the header file `boost/range/algorithm/set_algorithm.hpp`

Requirements

For the non-predicate versions:

- `SinglePassRange1` is a model of the Single Pass Range Concept.
- `SinglePassRange2` is a model of the Single Pass Range Concept.
- `OutputIterator` is a model of the `OutputIteratorConcept`.
- `SinglePassRange1` and `SinglePassRange2` have the same value type.
- `SinglePassRange1`'s value type is a model of the `LessThanComparableConcept`.
- `SinglePassRange2`'s value type is a model of the `LessThanComparableConcept`.
- The ordering of objects of type `SinglePassRange1`'s value type is a **strict weak ordering**, as defined in the `LessThanComparableConcept` requirements.
- The ordering of objects of type `SinglePassRange2`'s value type is a **strict weak ordering**, as defined in the `LessThanComparableConcept` requirements.

For the predicate versions:

- `SinglePassRange1` is a model of the Single Pass Range Concept.
- `SinglePassRange2` is a model of the Single Pass Range Concept.
- `OutputIterator` is a model of the `OutputIteratorConcept`.
- `SinglePassRange1` and `SinglePassRange2` have the same value type.
- `BinaryPredicate` is a model of the `StrictWeakOrderingConcept`.
- `SinglePassRange1`'s value type is convertible to `BinaryPredicate`'s first argument type.
- `SinglePassRange2`'s value type is convertible to `BinaryPredicate`'s second argument types.

Precondition:

For the non-predicate versions:

`rng1` and `rng2` are sorted in ascending order according to `operator<`.

For the predicate versions:

`rng1` and `rng2` are sorted in ascending order according to `pred`.

Complexity

Linear. `O(N)`, where `N` is `distance(rng1) + distance(rng2)`.

Heap algorithms

push_heap

Prototype

```
template<class RandomAccessRange>
RandomAccessRange& push_heap(RandomAccessRange& rng);

template<class RandomAccessRange>
const RandomAccessRange& push_heap(const RandomAccessRange& rng);

template<class RandomAccessRange, class Compare>
RandomAccessRange& push_heap(RandomAccessRange& rng, Compare pred);

template<class RandomAccessRange, class Compare>
const RandomAccessRange& push_heap(const RandomAccessRange& rng, Compare pred);
```

Description

`push_heap` adds an element to a heap. It is assumed that `begin(rng)`, `prior(end(rng))` is already a heap and that the element to be added is `*prior(end(rng))`.

The ordering relationship is determined by using `operator<` in the non-predicate versions, and by evaluating `pred` in the predicate versions.

Definition

Defined in the header file `boost/range/algorithm/heap_algorithm.hpp`

Requirements

For the non-predicate versions:

- `RandomAccessRange` is a model of the Random Access Range Concept.
- `RandomAccessRange` is mutable.
- `RandomAccessRange`'s value type is a model of the `LessThanComparableConcept`.
- The ordering of objects of type `RandomAccessRange`'s value type is a **strict weak ordering**, as defined in the `LessThanComparableConcept` requirements.

For the predicate versions:

- `RandomAccessRange` is a model of the Random Access Range Concept.
- `RandomAccessRange` is mutable.
- `Compare` is a model of the `StrictWeakOrderingConcept`.
- `RandomAccessRange`'s value type is convertible to both of `Compare`'s argument types.

Precondition:

- `!empty(rng)`
- `[begin(rng), prior(end(rng)))` is a heap.

Complexity

Logarithmic. At most `log(distance(rng))` comparisons.

pop_heap

Prototype

```
template<class RandomAccessRange>
RandomAccessRange& pop_heap(RandomAccessRange& rng);

template<class RandomAccessRange>
const RandomAccessRange& pop_heap(const RandomAccessRange& rng);

template<class RandomAccessRange, class Compare>
RandomAccessRange& pop_heap(RandomAccessRange& rng, Compare pred);

template<class RandomAccessRange, class Compare>
const RandomAccessRange& pop_heap(const RandomAccessRange& rng, Compare pred);
```

Description

`pop_heap` removes the largest element from the heap. It is assumed that `begin(rng), prior(end(rng))` is already a heap (and therefore the largest element is `*begin(rng)`).

The ordering relationship is determined by using `operator<` in the non-predicate versions, and by evaluating `pred` in the predicate versions.

Definition

Defined in the header file `boost/range/algorithm/heap_algorithm.hpp`

Requirements

For the non-predicate versions:

- `RandomAccessRange` is a model of the Random Access Range Concept.

- `RandomAccessRange` is mutable.
- `RandomAccessRange`'s value type is a model of the `LessThanComparableConcept`.
- The ordering of objects of type `RandomAccessRange`'s value type is a **strict weak ordering**, as defined in the `LessThanComparableConcept` requirements.

For the predicate versions:

- `RandomAccessRange` is a model of the Random Access Range Concept.
- `RandomAccessRange` is mutable.
- `Compare` is a model of the `StrictWeakOrderingConcept`.
- `RandomAccessRange`'s value type is convertible to both of `Compare`'s argument types.

Precondition:

- `!empty(rng)`
- `rng` is a heap.

Complexity

Logarithmic. At most `2 * log(distance(rng))` comparisons.

make_heap

Prototype

```
template<class RandomAccessRange>
RandomAccessRange& make_heap(RandomAccessRange& rng);

template<class RandomAccessRange>
const RandomAccessRange& make_heap(const RandomAccessRange& rng);

template<class RandomAccessRange, class Compare>
RandomAccessRange& make_heap(RandomAccessRange& rng, Compare pred);

template<class RandomAccessRange, class Compare>
const RandomAccessRange& make_heap(const RandomAccessRange& rng, Compare pred);
```

Description

`make_heap` turns `rng` into a heap.

The ordering relationship is determined by using `operator<` in the non-predicate versions, and by evaluating `pred` in the predicate versions.

Definition

Defined in the header file `boost/range/algorithm/heap_algorithm.hpp`

Requirements

For the non-predicate versions:

- `RandomAccessRange` is a model of the Random Access Range Concept.
- `RandomAccessRange` is mutable.

- `RandomAccessRange`'s value type is a model of the `LessThanComparableConcept`.
- The ordering of objects of type `RandomAccessRange`'s value type is a **strict weak ordering**, as defined in the `LessThanComparableConcept` requirements.

For the predicate versions:

- `RandomAccessRange` is a model of the Random Access Range Concept.
- `RandomAccessRange` is mutable.
- `Compare` is a model of the `StrictWeakOrderingConcept`.
- `RandomAccessRange`'s value type is convertible to both of `Compare`'s argument types.

Complexity

Linear. At most `3 * distance(rng)` comparisons.

sort_heap

Prototype

```
template<class RandomAccessRange>
RandomAccessRange& sort_heap(RandomAccessRange& rng);

template<class RandomAccessRange>
const RandomAccessRange& sort_heap(const RandomAccessRange& rng);

template<class RandomAccessRange, class Compare>
RandomAccessRange& sort_heap(RandomAccessRange& rng, Compare pred);

template<class RandomAccessRange, class Compare>
const RandomAccessRange& sort_heap(const RandomAccessRange& rng, Compare pred);
```

Description

`sort_heap` turns a heap into a sorted range.

The ordering relationship is determined by using `operator<` in the non-predicate versions, and by evaluating `pred` in the predicate versions.

Definition

Defined in the header file `boost/range/algorithm/heap_algorithm.hpp`

Requirements

For the non-predicate versions:

- `RandomAccessRange` is a model of the Random Access Range Concept.
- `RandomAccessRange` is mutable.
- `RandomAccessRange`'s value type is a model of the `LessThanComparableConcept`.
- The ordering of objects of type `RandomAccessRange`'s value type is a **strict weak ordering**, as defined in the `LessThanComparableConcept` requirements.

For the predicate versions:

- `RandomAccessRange` is a model of the Random Access Range Concept.

- RandomAccessRange is mutable.
- Compare is a model of the StrictWeakOrderingConcept.
- RandomAccessRange's value type is convertible to both of Compare's argument types.

Precondition:

rng is a heap.

Complexity

At most N * log(N) comparisons, where N is distance(rng).

Permutation algorithms

next_permutation

Prototype

```
template<class BidirectionalRange>
bool next_permutation(BidirectionalRange& rng);

template<class BidirectionalRange>
bool next_permutation(const BidirectionalRange& rng);

template<class BidirectionalRange, class Compare>
bool next_permutation(BidirectionalRange& rng, Compare pred);

template<class BidirectionalRange, class Compare>
bool next_permutation(const BidirectionalRange& rng, Compare pred);
```

Description

next_permutation transforms the range of elements rng into the lexicographically next greater permutation of the elements if such a permutation exists. If one does not exist then the range is transformed into the lexicographically smallest permutation and false is returned. true is returned when the next greater permutation is successfully generated.

The ordering relationship is determined by using operator< in the non-predicate versions, and by evaluating pred in the predicate versions.

Definition

Defined in the header file boost/range/algorithm/permutation.hpp

Requirements

For the non-predicate versions:

- BidirectionalRange is a model of the Bidirectional Range Concept.
- BidirectionalRange is mutable.
- BidirectionalRange's value type is a model of the LessThanComparableConcept.
- The ordering of objects of type BidirectionalRange's value type is a **strict weak ordering**, as defined in the LessThanComparableConcept requirements.

For the predicate versions:

- BidirectionalRange is a model of the Bidirectional Range Concept.

- `BidirectionalRange` is mutable.
- `Compare` is a model of the `StrictWeakOrderingConcept`.
- `BidirectionalRange`'s value type is convertible to both of `Compare`'s argument types.

Complexity

Linear. At most `distance(rng) / 2` swaps.

prev_permutation

Prototype

```
template<class BidirectionalRange>
bool prev_permutation(BidirectionalRange& rng);

template<class BidirectionalRange>
bool prev_permutation(const BidirectionalRange& rng);

template<class BidirectionalRange, class Compare>
bool prev_permutation(BidirectionalRange& rng, Compare pred);

template<class BidirectionalRange, class Compare>
bool prev_permutation(const BidirectionalRange& rng, Compare pred);
```

Description

`prev_permutation` transforms the range of elements `rng` into the lexicographically next smaller permutation of the elements if such a permutation exists. If one does not exist then the range is transformed into the lexicographically largest permutation and `false` is returned. `true` is returned when the next smaller permutation is successfully generated.

The ordering relationship is determined by using `operator<` in the non-predicate versions, and by evaluating `pred` in the predicate versions.

Definition

Defined in the header file `boost/range/algorithm/permutation.hpp`

Requirements

For the non-predicate versions:

- `BidirectionalRange` is a model of the Bidirectional Range Concept.
- `BidirectionalRange` is mutable.
- `BidirectionalRange`'s value type is a model of the `LessThanComparableConcept`.
- The ordering of objects of type `BidirectionalRange`'s value type is a **strict weak ordering**, as defined in the `LessThanComparableConcept` requirements.

For the predicate versions:

- `BidirectionalRange` is a model of the Bidirectional Range Concept.
- `BidirectionalRange` is mutable.
- `Compare` is a model of the `StrictWeakOrderingConcept`.
- `BidirectionalRange`'s value type is convertible to both of `Compare`'s argument types.

Complexity

Linear. At most `distance(rng) / 2` swaps.

New algorithms

copy_n

Prototype

```
template<class SinglePassRange, class Size, class OutputIterator>
OutputIterator copy_n(const SinglePassRange& rng, Size n, OutputIterator out);
```

Description

`copy_n` is provided to completely replicate the standard algorithm header, it is preferable to use Range Adaptors and the extension functions to achieve the same result with greater safety.

`copy_n` copies elements from `[boost::begin(rng), boost::begin(rng) + n)` to the range `[out, out + n)`

Definition

Defined in the header file `boost/range/algorithm_ext/copy_n.hpp`

Requirements

1. `SinglePassRange` is a model of the Single Pass Range Concept.

2. `Size` is a model of the `Integer` Concept.

3. `OutputIterator` is a model of the `OutputIteratorConcept`.

Complexity

Linear. Exactly `n` elements are copied.

erase

Prototype

```
template<class Container>
Container& erase(
    Container& target,
    iterator_range<typename Container::iterator> to_erase);
```

Description

`erase` the iterator range `to_erase` from the container `target`.

`remove_erase` performs the frequently used combination equivalent to `target.erase(std::remove(target.begin(), target.end(), value), target.end());`

`remove_erase_if` performs the frequently used combination equivalent to `target.erase(std::remove_if(target.begin(), target.end(), pred), target.end());`

Definition

Defined in the header file `boost/range/algorithm_ext/erase.hpp`

Requirements

1. `Container` supports erase of an iterator range.

Complexity

Linear. Proprotional to `distance(to_erase)`.

for_each

Prototype

```
template<
    class SinglePassRange1,
    class SinglePassRange2,
    class BinaryFunction
    >
BinaryFunction for_each(const SinglePassRange1& rng1,
                        const SinglePassRange2& rng2,
                        BinaryFunction fn);

template<
    class SinglePassRange1,
    class SinglePassRange2,
    class BinaryFunction
    >
BinaryFunction for_each(const SinglePassRange1& rng1,
                        SinglePassRange2& rng2,
                        BinaryFunction fn);

template<
    class SinglePassRange1,
    class SinglePassRange2,
    class BinaryFunction
    >
BinaryFunction for_each(SinglePassRange1& rng1,
                        const SinglePassRange2& rng2,
                        BinaryFunction fn);

template<
    class SinglePassRange1,
    class SinglePassRange2,
    class BinaryFunction
    >
BinaryFunction for_each(SinglePassRange1& rng1,
                        SinglePassRange2& rng2,
                        BinaryFunction fn);
```

Description

`for_each` traverses forward through `rng1` and `rng2` simultaneously. For each iteration, the element `x` is used from `rng1` and the corresponding element `y` is used from `rng2` to invoke `fn(x,y)`.

Iteration is stopped upon reaching the end of the shorter of `rng1`, or `rng2`. It is safe to call this function with unequal length ranges.

Definition

Defined in the header file `boost/range/algorithm_ext/for_each.hpp`

Requirements

1. `SinglePassRange1` is a model of the Single Pass Range Concept.

2. `SinglePassRange2` is a model of the Single Pass Range Concept.

3. `BinaryFunction` is a model of the `BinaryFunctionConcept`.

4. `SinglePassRange1`'s value type is convertible to `BinaryFunction`'s first argument type.

5. `SinglepassRange2`'s value type is convertible to `BinaryFunction`'s second argument type.

Complexity

Linear. Exactly `min(distance(rng1), distance(rng2))` applications of `BinaryFunction`.

insert

Prototype

```
template<
    class Container,
    class SinglePassRange
    >
Container& insert(Container& target,
                  typename Container::iterator before,
                  const SinglePassRange& from);
```

Description

`insert` all of the elements in the range `from` before the `before` iterator into `target`.

Definition

Defined in the header file `boost/range/algorithm_ext/insert.hpp`

Requirements

1. `SinglePassRange` is a model of the Single Pass Range Concept.

2. `Container` supports insert at a specified position.

3. `SinglePassRange`'s value type is convertible to `Container`'s value type.

Complexity

Linear. `distance(from)` assignments are performed.

iota

Prototype

```
template<class ForwardRange, class Value>
ForwardRange& iota(ForwardRange& rng, Value x);
```

Description

`iota` traverses forward through `rng`, each element `y` in `rng` is assigned a value equivalent to `x + boost::distance(boost::begin(rng), it)`

Definition

Defined in the header file `boost/range/algorithm_ext/iota.hpp`

Requirements

1. `ForwardRange` is a model of the Forward Range Concept.

2. `Value` is a model of the `Incrementable` Concept.

Complexity

Linear. Exactly `distance(rng)` assignments into `rng`.

is_sorted

Prototype

```
template<class SinglePassRange>
bool is_sorted(const SinglePassRange& rng);

template<class SinglePassRange, class BinaryPredicate>
bool is_sorted(const SinglePassRange& rng, BinaryPredicate pred);
```

Description

`is_sorted` determines if a range is sorted. For the non-predicate version the return value is `true` if and only if for each adjacent elements `[x,y]` the expression `x < y` is `true`. For the predicate version the return value is `true` is and only if for each adjacent elements `[x,y]` the expression `pred(x,y)` is `true`.

Definition

Defined in the header file `boost/range/algorithm_ext/is_sorted.hpp`

Requirements

1. `SinglePassRange` is a model of the Single Pass Range Concept.

2. `BinaryPredicate` is a model of the `BinaryPredicate` Concept.

3. The value type of `SinglePassRange` is convertible to both argument types of `BinaryPredicate`.

Complexity

Linear. A maximum of `distance(rng)` comparisons are performed.

overwrite

Prototype

```
template<
    class SinglePassRange1,
    class SinglePassRange2
    >
void overwrite(const SinglePassRange1& from,
               SinglePassRange2& to);
```

Description

`overwrite` assigns the values from the range `from` into the range `to`.

Definition

Defined in the header file `boost/range/algorithm_ext/overwrite.hpp`

Requirements

1. `SinglePassRange1` is a model of the Single Pass Range Concept.
2. `SinglePassRange2` is a model of the Single Pass Range Concept.
3. `SinglePassRange2` is mutable.
4. `distance(SinglePassRange1) <= distance(SinglePassRange2)`
5. `SinglePassRange1`'s value type is convertible to `SinglePassRange2`'s value type.

Complexity

Linear. `distance(rng1)` assignments are performed.

push_back

Prototype

```
template<
    class Container,
    class SinglePassRange
    >
Container& push_back(Container& target,
                     const SinglePassRange& from);
```

Description

`push_back` all of the elements in the range `from` to the back of the container `target`.

Definition

Defined in the header file `boost/range/algorithm_ext/push_back.hpp`

Requirements

1. `SinglePassRange` is a model of the Single Pass Range Concept.
2. `Container` supports insert at `end()`.
3. `SinglePassRange`'s value type is convertible to `Container`'s value type.

Complexity

Linear. `distance(from)` assignments are performed.

push_front

Prototype

```
template<
    class Container,
    class SinglePassRange
    >
Container& push_front(Container& target,
                      const SinglePassRange& from);
```

Description

`push_front` all of the elements in the range `from` to the front of the container `target`.

Definition

Defined in the header file `boost/range/algorithm_ext/push_front.hpp`

Requirements

1. `SinglePassRange` is a model of the Single Pass Range Concept.
2. `Container` supports insert at `begin()`.
3. `SinglePassRange`'s value type is convertible to `Container`'s value type.

Complexity

Linear. `distance(from)` assignments are performed.

remove_erase

Prototype

```
template<class Container, class Value>
Container& remove_erase(Container& target,
                        const Value& value);
```

Description

`remove_erase` actually eliminates the elements equal to `value` from the container. This is in contrast to the `remove` algorithm which merely rearranges elements.

Definition

Defined in the header file `boost/range/algorithm_ext/erase.hpp`

Requirements

1. `Container` supports erase of an iterator range.

Complexity

Linear. Proportional to `distance(target)`s.

remove_erase_if

Prototype

```
template<class Container, class Pred>
Container& remove_erase_if(Container& target,
                           Pred pred);
```

Description

`remove_erase_if` removes the elements `x` that satisfy `pred(x)` from the container. This is in contrast to the `erase` algorithm which merely rearranges elements.

Definition

Defined in the header file `boost/range/algorithm_ext/erase.hpp`

Requirements

1. `Container` supports erase of an iterator range.

2. `Pred` is a model of the `Predicate` Concept.

Complexity

Linear. Proportional to `distance(target)`s.

Numeric algorithms

accumulate

Prototype

```
template<
    class SinglePassRange,
    class Value
    >
Value accumulate(const SinglePassRange& source_rng,
                 Value init);

template<
    class SinglePassRange,
    class Value,
    class BinaryOperation
    >
Value accumulate(const SinglePassRange& source_rng,
                 Value init,
                 BinaryOperation op);
```

Description

`accumulate` is a generalisation of summation. It computes a binary operation (`operator+` in the non-predicate version) of `init` and all of the elements in `rng`.

The return value is the resultant value of the above algorithm.

Definition

Defined in the header file `boost/range/numeric.hpp`

Requirements

For the first version

1. `SinglePassRange` is a model of the Single Pass Range Concept.

2. `Value` is a model of the `AssignableConcept`.

3. An `operator+` is defined for a left-hand operand of type `Value` and a right-hand operand of the `SinglePassRange` value type.

4. The return type of the above operator is convertible to `Value`.

For the second version

1. `SinglePassRange` is a model of the Single Pass Range Concept.

2. `Value` is a model of the `AssignableConcept`.

3. `BinaryOperation` is a model of the `BinaryFunctionConcept`.

4. `Value` is convertible to `BinaryOperation`'s first argument type.

5. `SinglePassRange`'s value type is convertible to `BinaryOperation`'s second argument type.

6. The return type of `BinaryOperation` is convertible to `Value`.

Complexity

Linear. Exactly `distance(source_rng)`.

adjacent_difference

Prototype

```
template<
    class SinglePassRange,
    class OutputIterator
    >
OutputIterator adjacent_difference(
    const SinglePassRange& source_rng,
    OutputIterator out_it);

template<
    class SinglePassRange,
    class OutputIterator,
    class BinaryOperation
    >
OutputIterator adjacent_difference(
    const SinglePassRange& source_rng,
    OutputIterator out_it,
    BinaryOperation op);
```

Description

`adjacent_difference` calculates the differences of adjacent_elements in `rng`.

The first version of `adjacent_difference` uses `operator-()` to calculate the differences. The second version uses `BinaryOperation` instead of `operator-()`.

Definition

Defined in the header file `boost/range/numeric.hpp`

Requirements

For the first version

1. `SinglePassRange` is a model of the Single Pass Range Concept.

2. `OutputIterator` is a model of the `OutputIteratorConcept`.

3. If `x` and `y` are objects of `SinglePassRange`'s value type, then `x - y` is defined.

4. The value type of `SinglePassRange` is convertible to a type in `OutputIterator`'s set of value types.

5. The return type of `x - y` is convertible to a type in `OutputIterator`'s set of value types.

For the second version

1. `SinglePassRange` is a model of the Single Pass Range Concept.

2. `OutputIterator` is a model of the `OutputIteratorConcept`.
3. `BinaryOperation` is a model of the `BinaryFunctionConcept`.
4. The value type of `SinglePassRange` is convertible to `BinaryOperation`'s first and second argument types.
5. The value type of `SinglePassRange` is convertible to a type in `OutputIterator`'s set of value types.
6. The result type of `BinaryOperation` is convertible to a type in `OutputIterator`'s set of value types.

Precondition:

`[result, result + distance(rng))` is a valid range.

Complexity

Linear. If `empty(rng)` then zero applications, otherwise `distance(rng) - 1` applications are performed.

inner_product

Prototype

```
template<class SinglePassRange1,
         class SinglePassRange2,
         class Value>
    Value inner_product( const SinglePassRange1& rng1,
                         const SinglePassRange2& rng2,
                         Value                   init );

template<class SinglePassRange1,
         class SinglePassRange2,
         class Value,
         class BinaryOperation1,
         class BinaryOperation2>
    Value inner_product( const SinglePassRange1& rng1,
                         const SinglePassRange2& rng2,
                         Value                   init,
                         BinaryOperation1        op1,
                         BinaryOperation2        op2 );
```

Description

`inner_product` calculates a generalised inner product of the range `rng1` and `rng2`.

For further information on the `inner_product` algorithm please see inner_product.

Definition

Defined in the header file `boost/range/numeric.hpp`

Requirements

For the first version

1. `SinglePassRange1` is a model of the Single Pass Range Concept.
2. `SinglePassRange2` is a model of the Single Pass Range Concept.
3. `Value` is a model of the `AssignableConcept`.
4. If `x` is an object of type `Value`, `y` is an object of `SinglePassRange1`'s value type, and `z` is an object of `SinglePassRange2`'s value type, then `x + y * z` is defined.

5. The result type of the expression `x + y * z` is convertible to `Value`.

For the second version

1. `SinglePassRange1` is a model of the Single Pass Range Concept.
2. `SinglePassRange2` is a model of the Single Pass Range Concept.
3. `Value` is a model of the `AssignableConcept`.
4. `BinaryOperation1` is a model of the `BinaryFunctionConcept`.
5. `BinaryOperation2` is a model of the `BinaryFunctionConcept`.
6. The value type of `SinglePassRange1` is convertible to the first argument type of `BinaryOperation2`.
7. The value type of `SinglePassRange2` is convertible to the second argument type of `BinaryOperation2`.
8. `Value` is convertible to the value type of `BinaryOperation1`.
9. The return type of `BinaryOperation2` is convertible to the second argument type of `BinaryOperation1`.
10. The return type of `BinaryOperation1` is convertible to `Value`.

Precondition:

`distance(rng2) >= distance(rng1)` is a valid range.

Complexity

Linear. Exactly `distance(rng)`.

partial_sum

Prototype

```
template<class SinglePassRange,
         class OutputIterator>
OutputIterator partial_sum(const SinglePassRange& rng,
                           OutputIterator out_it);

template<class SinglePassRange,
         class OutputIterator,
         class BinaryOperation>
OutputIterator partial_sum(const SinglePassRange& rng,
                           OutputIterator out_it,
                           BinaryOperation op);
```

Description

`partial_sum` calculates a generalised partial sum of `rng` in the same manner as `std::partial_sum(boost::begin(rng), boost::end(rng), out_it)`. See partial_sum.

Definition

Defined in the header file `boost/range/numeric.hpp`

Requirements

For the first version

1. `SinglePassRange` is a model of the Single Pass Range Concept.

2. `OutputIterator` is a model of the `OutputIteratorConcept`.

3. If `x` and `y` are objects of `SinglePassRange`'s value type, then `x + y` is defined.

4. The return type of `x + y` is convertible to the value type of `SinglePassRange`.

5. The value type of `SinglePassRange` is convertible to a type in `OutputIterator`'s set of value types.

For the second version

1. `SinglePassRange` is a model of the Single Pass Range Concept.

2. `OutputIterator` is a model of the `OutputIteratorConcept`.

3. `BinaryOperation` is a model of the `BinaryFunctionConcept`.

4. The result type of `BinaryOperation` is convertible to the value type of `SinglePassRange`.

5. The value type of `SinglePassRange` is convertible to a type in `OutputIterator`'s set of value types.

Precondition:

`[result, result + distance(rng))` is a valid range.

Complexity

Linear. If `empty(rng)` then zero applications, otherwise `distance(rng) - 1` applications are performed.

Provided Ranges

any_range

Description

`any_range` is a range that has the type information erased hence a `any_range<int, boost::forward_pass_traversal_tag, int, std::ptrdiff_t>` can be used to represent a `std::vector<int>`, a `std::list<int>` or many other types.

The type erasure article covers the motivation and goals of type erasure in this context. Clearly my implementation is building upon a lot of prior art created by others. Thomas Becker's `any_iterator` was a strong influence. Adobe also have an `any_iterator` implementation, but this has very tight coupling to other parts of the library that precluded it from use in Boost.Range. Early development versions of this Range Adaptor directly used Thomas Becker's any_iterator implementation. Subsequently I discovered that the heap allocations of this and many other implementations cause poor speed performance particularly at the tails of the distribution. To solve this required a new design that incorporated the embedded buffer optimization.

Despite the underlying `any_iterator` being the fastest available implementation, the performance overhead of `any_range` is still appreciable due to the cost of virtual function calls required to implement `increment`, `decrement`, `advance`, `equal` etc. Frequently a better design choice is to convert to a canonical form.

Please see the type_erased for a Range Adaptor that returns `any_range` instances.

Synopsis

```
template<
    class Value
  , class Traversal
  , class Reference
  , class Difference
  , class Buffer = any_iterator_default_buffer
>
class any_range
    : public iterator_range<
        range_detail::any_iterator<
            Value
          , Traversal
          , Reference
          , Difference
          , Buffer
        >
    >
{
    typedef range_detail::any_iterator<
        Value
      , Traversal
      , Reference
      , Difference
      , Buffer
    > any_iterator_type;

    typedef iterator_range<any_iterator_type> base_type;

    struct enabler {};
    struct disabler {};
public:
    typedef any_iterator_type iterator;
    typedef any_iterator_type const_iterator;

    any_range()
    {
    }

    any_range(const any_range& other)
        : base_type(other)
    {
    }

    template<class WrappedRange>
    any_range(WrappedRange& wrapped_range)
    : base_type(boost::begin(wrapped_range),
                boost::end(wrapped_range))
    {
    }

    template<class WrappedRange>
    any_range(const WrappedRange& wrapped_range)
    : base_type(boost::begin(wrapped_range),
                boost::end(wrapped_range))
    {
    }

    template<
        class OtherValue
      , class OtherTraversal
      , class OtherReference
```

```
        , class OtherDifference
    >
    any_range(const any_range<
                        OtherValue
                      , OtherTraversal
                      , OtherReference
                      , OtherDifference
                      , Buffer
                    >& other)
    : base_type(boost::begin(other), boost::end(other))
    {
    }

    template<class Iterator>
    any_range(Iterator first, Iterator last)
        : base_type(first, last)
    {
    }
};
```

Definition

Defined in header file `boost/range/any_range.hpp`

counting_range

Prototype

```
template< class Incrementable > inline
iterator_range< counting_iterator<Incrementable> >
counting_range(Incrementable first, Incrementable last);

template< class SinglePassRange > inline
iterator_range< counting_iterator<typename range_iterator<SinglePassRange>::type >
counting_range(const SinglePassRange& rng);

template< class SinglePassRange > inline
iterator_range< counting_iterator<typename range_iterator<SinglePassRange>::type >
counting_range(SinglePassRange& rng);
```

Description

`counting_range` is a function to generator that generates an `iterator_range` wrapping a `counting_iterator` (from Boost.Iterator).

Definition

Defined in header file `boost/range/counting_range.hpp`

Requirements

1. `Incrementable` is a model of the `Incrementable` Concept.

istream_range

Prototype

```
template< class Type, class Elem, class Traits > inline
iterator_range< std::istream_iterator<Type, Elem, Traits> >
istream_range(std::basic_istream<Elem, Traits>& in);
```

Description

`istream_range` is a function to generator that generates an `iterator_range` wrapping a `std::istream_iterator`.

Definition

Defined in header file `boost/range/istream_range.hpp`

irange

Prototype

```
template<class Integer>
iterator_range< range_detail::integer_iterator<Integer> >
irange(Integer first, Integer  last);

template<class Integer, class StepSize>
iterator_range< range_detail::integer_iterator_with_step<Integer, StepSize> >
irange(Integer first, Integer last, StepSize step_size);
```

Description

`irange` is a function to generate an Integer Range.

`irange` allows treating integers as a model of the Random Access Range Concept. It should be noted that the `first` and `last` parameters denoted a half-open range.

Definition

Defined in the header file `boost/range/irange.hpp`

Requirements

1. `Integer` is a model of the `Integer` Concept.
2. `StepSize` is a model of the `SignedInteger` Concept.

Complexity

Constant. Since this function generates a new range the most significant performance cost is incurred through the iteration of the generated range.

Utilities

Having an abstraction that encapsulates a pair of iterators is very useful. The standard library uses `std::pair` in some circumstances, but that class is cumbersome to use because we need to specify two template arguments, and for all range algorithm purposes we must enforce the two template arguments to be the same. Moreover, `std::pair<iterator,iterator>` is hardly self-documenting whereas more domain specific class names are. Therefore these two classes are provided:

- Class `iterator_range`

- Class sub_range
- Function join

The iterator_range class is templated on an Forward Traversal Iterator and should be used whenever fairly general code is needed. The sub_range class is templated on an Forward Range and it is less general, but a bit easier to use since its template argument is easier to specify. The biggest difference is, however, that a sub_range can propagate constness because it knows what a corresponding const_iterator is.

Both classes can be used as ranges since they implement the minimal interface required for this to work automatically.

Class iterator_range

The intention of the iterator_range class is to encapsulate two iterators so they fulfill the Forward Range concept. A few other functions are also provided for convenience.

If the template argument is not a model of Forward Traversal Iterator, one can still use a subset of the interface. In particular, size() requires Random Access Traversal Iterators whereas empty() only requires Single Pass Iterators.

Recall that many default constructed iterators are **singular** and hence can only be assigned, but not compared or incremented or anything. However, if one creates a default constructed iterator_range, then one can still call all its member functions. This design decision avoids the iterator_range imposing limitations upon ranges of iterators that are not singular. Any singularity limitation is simply propagated from the underlying iterator type.

Synopsis

```
namespace boost
{
    template< class ForwardTraversalIterator >
    class iterator_range
    {
    public: // Forward Range types
        typedef ForwardTraversalIterator   iterator;
        typedef ForwardTraversalIterator   const_iterator;
        typedef iterator_difference<iterator>::type difference_type;

    public: // construction, assignment
        template< class ForwardTraversalIterator2 >
        iterator_range( ForwardTraversalIterator2 Begin, ForwardTraversalIterator2 End );

        template< class ForwardRange >
        iterator_range( ForwardRange& r );

        template< class ForwardRange >
        iterator_range( const ForwardRange& r );

        template< class ForwardRange >
        iterator_range& operator=( ForwardRange& r );

        template< class ForwardRange >
        iterator_range& operator=( const ForwardRange& r );

    public: // Forward Range functions
        iterator  begin() const;
        iterator  end() const;

    public: // convenience
        operator    unspecified_bool_type() const;
        bool        equal( const iterator_range& ) const;
        value_type& front() const;
        value_type& back() const;
        iterator_range& advance_begin(difference_type n);
        iterator_range& advance_end(difference_type n);
        bool       empty() const;
        // for Random Access Range only:
        reference operator[]( difference_type at ) const;
        value_type operator()( difference_type at ) const;
        size_type size() const;
    };

    // stream output
    template< class ForwardTraversalIterator, class T, class Traits >
    std::basic_ostream<T,Traits>&
    operator<<( std::basic_ostream<T,Traits>& Os,
                const iterator_range<ForwardTraversalIterator>& r );

    // comparison
    template< class ForwardTraversalIterator, class ForwardTraversalIterator2 >
    bool operator==( const iterator_range<ForwardTraversalIterator>& l,
                     const iterator_range<ForwardTraversalIterator2>& r );

    template< class ForwardTraversalIterator, class ForwardRange >
    bool operator==( const iterator_range<ForwardTraversalIterator>& l,
                     const ForwardRange& r );

    template< class ForwardTraversalIterator, class ForwardRange >
    bool operator==( const ForwardRange& l,
```

```
                      const iterator_range<ForwardTraversalIterator>& r );

    template< class ForwardTraversalIterator, class ForwardTraversalIterator2 >
    bool operator!=( const iterator_range<ForwardTraversalIterator>& l,
                     const iterator_range<ForwardTraversalIterator2>& r );

    template< class ForwardTraversalIterator, class ForwardRange >
    bool operator!=( const iterator_range<ForwardTraversalIterator>& l,
                     const ForwardRange& r );

    template< class ForwardTraversalIterator, class ForwardRange >
    bool operator!=( const ForwardRange& l,
                     const iterator_range<ForwardTraversalIterator>& r );

    template< class ForwardTraversalIterator, class ForwardTraversalIterator2 >
    bool operator<( const iterator_range<ForwardTraversalIterator>& l,
                    const iterator_range<ForwardTraversalIterator2>& r );

    template< class ForwardTraversalIterator, class ForwardRange >
    bool operator<( const iterator_range<ForwardTraversalIterator>& l,
                    const ForwardRange& r );

    template< class ForwardTraversalIterator, class ForwardRange >
    bool operator<( const ForwardRange& l,
                    const iterator_range<ForwardTraversalIterator>& r );

    // external construction
    template< class ForwardTraversalIterator >
    iterator_range< ForwardTraversalIterator >
    make_iterator_range( ForwardTraversalIterator Begin,
                         ForwardTraversalIterator End );

    template< class ForwardRange >
    iterator_range< typename range_iterator<ForwardRange>::type >
    make_iterator_range( ForwardRange& r );

    template< class ForwardRange >
    iterator_range< typename range_iterator<const ForwardRange>::type >
    make_iterator_range( const ForwardRange& r );

    template< class Range >
    iterator_range< typename range_iterator<Range>::type >
    make_iterator_range( Range& r,
                         typename range_difference<Range>::type advance_begin,
                         typename range_difference<Range>::type advance_end );

    template< class Range >
    iterator_range< typename range_iterator<const Range>::type >
    make_iterator_range( const Range& r,
                         typename range_difference<const Range>::type advance_begin,
                         typename range_difference<const Range>::type advance_end );

    // convenience
    template< class Sequence, class ForwardRange >
    Sequence copy_range( const ForwardRange& r );

} // namespace 'boost'
```

If an instance of `iterator_range` is constructed by a client with two iterators, the client must ensure that the two iterators delimit a valid closed-open range [begin,end).

It is worth noticing that the templated constructors and assignment operators allow conversion from `iterator_range<iterator>` to `iterator_range<const_iterator>`. Similarly, since the comparison operators have two template arguments, we can compare ranges whenever the iterators are comparable; for example when we are dealing with const and non-const iterators from the same container.

Details member functions

`operator unspecified_bool_type() const;`

> ***Returns*** `!empty();`

`bool equal( iterator_range& r ) const;`

> ***Returns*** `begin() == r.begin() && end() == r.end();`

Details functions

`bool operator==( const ForwardRange1& l, const ForwardRange2& r );`

> ***Returns*** `size(l) != size(r) ? false : std::equal( begin(l), end(l), begin(r) );`

`bool operator!=( const ForwardRange1& l, const ForwardRange2& r );`

> ***Returns*** `!( l == r );`

`bool operator<( const ForwardRange1& l, const ForwardRange2& r );`

> ***Returns*** `std::lexicographical_compare( begin(l), end(l), begin(r), end(r) );`

```
iterator_range make_iterator_range( Range& r,
                                    typename range_difference<Range>::type advance_begin,
                                    typename range_difference<Range>::type advance_end );
```

> ***Effects:***

```
iterator new_begin = begin( r ),
iterator new_end   = end( r );
std::advance( new_begin, advance_begin );
std::advance( new_end, advance_end );
return make_iterator_range( new_begin, new_end );
```

`Sequence copy_range( const ForwardRange& r );`

> ***Returns*** `Sequence( begin(r), end(r) );`

Class `sub_range`

The `sub_range` class inherits all its functionality from the `iterator_range` class. The `sub_range` class is often easier to use because one must specify the Forward Range template argument instead of an iterator. Moreover, the `sub_range` class can propagate constness since it knows what a corresponding `const_iterator` is.

Synopsis

```
namespace boost
{
    template< class ForwardRange >
    class sub_range : public iterator_range< typename range_iterator<ForwardRange>::type >
    {
    public:
        typedef typename range_iterator<ForwardRange>::type iterator;
        typedef typename range_iterator<const ForwardRange>::type  const_iterator;
        typedef typename iterator_difference<iterator>::type difference_type;

    public: // construction, assignment
        template< class ForwardTraversalIterator >
        sub_range( ForwardTraversalIterator Begin, ForwardTraversalIterator End );

        template< class ForwardRange2 >
        sub_range( ForwardRange2& r );

        template< class ForwardRange2 >
        sub_range( const Range2& r );

        template< class ForwardRange2 >
        sub_range& operator=( ForwardRange2& r );

        template< class ForwardRange2 >
        sub_range& operator=( const ForwardRange2& r );

    public:  // Forward Range functions
        iterator        begin();
        const_iterator  begin() const;
        iterator        end();
        const_iterator  end() const;

    public: // convenience
        value_type&       front();
        const value_type& front() const;
        value_type&       back();
        const value_type& back() const;
        // for Random Access Range only:
        value_type&       operator[]( size_type at );
        const value_type& operator[]( size_type at ) const;

    public:
        // rest of interface inherited from iterator_range
    };

} // namespace 'boost'
```

The class should be trivial to use as seen below. Imagine that we have an algorithm that searches for a sub-string in a string. The result is an iterator_range, that delimits the match. We need to store the result from this algorithm. Here is an example of how we can do it with and without `sub_range`

```
std::string str("hello");
iterator_range<std::string::iterator> ir = find_first( str, "ll" );
sub_range<std::string>               sub = find_first( str, "ll" );
```

Function join

The intention of the `join` function is to join two ranges into one longer range.

The resultant range will have the lowest common traversal of the two ranges supplied as parameters.

Note that the joined range incurs a performance cost due to the need to check if the end of a range has been reached internally during traversal.

Synposis

```
template<typename SinglePassRange1, typename SinglePassRange2>
joined_range<const SinglePassRange1, const SinglePassRange2>
join(const SinglePassRange1& rng1, const SinglePassRange2& rng2)

template<typename SinglePassRange1, typename SinglePassRange2>
joined_range<SinglePassRange1, SinglePassRange2>
join(SinglePassRange1& rng1, SinglePassRange2& rng2);
```

For the const version:

- **Precondition:** The `range_value<SinglePassRange2>::type` must be convertible to `range_value<SinglePassRange1>::type`. The `range_reference<const SinglePassRange2>::type` must be convertible to `range_reference<const SinglePassRange1>::type`.
- **Range Category:** Both `rng1` and `rng2` must be a model of Single Pass Range or better.
- **Range Return Type:** `joined_range<const SinglePassRange1, const SinglePassRange2>` which is a model of the lesser of the two range concepts passed.
- **Returned Range Category:** The minimum of the range category of `rng1` and `rng2`.

For the mutable version:

- **Precondition:** The `range_value<SinglePassRange2>::type` must be convertible to `range_value<SinglePassRange1>::type`. The `range_reference<SinglePassRange2>::type` must be convertible to `range_reference<SinglePassRange1>::type`.
- **Range Category:** Both `rng1` and `rng2` must be a model of Single Pass Range or better.
- **Range Return Type:** `joined_range<SinglePassRange1, SinglePassRange2>` which is a model of the lesser of the two range concepts passed.
- **Returned Range Category:** The minimum of the range category of `rng1` and `rng2`.

Example

The expression `join(irange(0,5), irange(5,10))` would evaluate to a range representing an integer range `[0,10)`

Extending the library

Method 1: provide member functions and nested types

This procedure assumes that you have control over the types that should be made conformant to a Range concept. If not, see method 2.

The primary templates in this library are implemented such that standard containers will work automatically and so will boost::array. Below is given an overview of which member functions and member types a class must specify to be useable as a certain Range concept.

Member function	Related concept
`begin()`	Single Pass Range
`end()`	Single Pass Range

Notice that `rbegin()` and `rend()` member functions are not needed even though the container can support bidirectional iteration.

The required member types are:

Member type	Related concept
`iterator`	Single Pass Range
`const_iterator`	Single Pass Range

Again one should notice that member types `reverse_iterator` and `const_reverse_iterator` are not needed.

Method 2: provide free-standing functions and specialize metafunctions

This procedure assumes that you cannot (or do not wish to) change the types that should be made conformant to a Range concept. If this is not true, see method 1.

The primary templates in this library are implemented such that certain functions are found via argument-dependent-lookup (ADL). Below is given an overview of which free-standing functions a class must specify to be useable as a certain Range concept. Let `x` be a variable (`const` or `mutable`) of the class in question.

Function	Related concept
`range_begin(x)`	Single Pass Range
`range_end(x)`	Single Pass Range
`range_calculate_size(x)`	Optional. This can be used to specify a mechanism for constant-time computation of the size of a range. The default behaviour is to return `boost::end(x) - boost::begin(x)` for random access ranges, and to return `x.size()` for ranges with lesser traversal capability. This behaviour can be changed by implementing `range_calculate_size` in a manner that will be found via ADL. The ability to calculate size in O(1) is often possible even with ranges with traversal categories less than random access.

`range_begin()` and `range_end()` must be overloaded for both `const` and `mutable` reference arguments.

You must also specialize two metafunctions for your type `X`:

Metafunction	Related concept
`boost::range_mutable_iterator`	Single Pass Range
`boost::range_const_iterator`	Single Pass Range

A complete example is given here:

```
#include <boost/range.hpp>
#include <iterator>         // for std::iterator_traits, std::distance()

namespace Foo
{
    //
    // Our sample UDT. A 'Pair'
    // will work as a range when the stored
    // elements are iterators.
    //
    template< class T >
    struct Pair
    {
        T first, last;
    };

} // namespace 'Foo'

namespace boost
{
    //
    // Specialize metafunctions. We must include the range.hpp header.
    // We must open the 'boost' namespace.
    //

 template< class T >
 struct range_mutable_iterator< Foo::Pair<T> >
 {
  typedef T type;
 };

 template< class T >
 struct range_const_iterator< Foo::Pair<T> >
 {
  //
  // Remark: this is defined similar to 'range_iterator'
  //         because the 'Pair' type does not distinguish
  //         between an iterator and a const_iterator.
  //
  typedef T type;
 };

} // namespace 'boost'

namespace Foo
{
 //
 // The required functions. These should be defined in
 // the same namespace as 'Pair', in this case
 // in namespace 'Foo'.
 //

 template< class T >
 inline T range_begin( Pair<T>& x )
 {
  return x.first;
 }

 template< class T >
 inline T range_begin( const Pair<T>& x )
 {
  return x.first;
 }
```

```
 template< class T >
 inline T range_end( Pair<T>& x )
 {
  return x.last;
 }

 template< class T >
 inline T range_end( const Pair<T>& x )
 {
  return x.last;
 }

} // namespace 'Foo'

#include <vector>

int main(int argc, const char* argv[])
{
 typedef std::vector<int>::iterator  iter;
 std::vector<int>                    vec;
 Foo::Pair<iter>                     pair = { vec.begin(), vec.end() };
 const Foo::Pair<iter>&              cpair = pair;
 //
 // Notice that we call 'begin' etc with qualification.
 //
 iter i = boost::begin( pair );
 iter e = boost::end( pair );
 i      = boost::begin( cpair );
 e      = boost::end( cpair );
 boost::range_difference< Foo::Pair<iter> >::type s = boost::size( pair );
 s      = boost::size( cpair );
 boost::range_reverse_iterator< const Foo::Pair<iter> >::type
 ri     = boost::rbegin( cpair ),
 re     = boost::rend( cpair );

 return 0;
}
```

Method 3: provide range adaptor implementations

Method 3.1: Implement a Range Adaptor without arguments

To implement a Range Adaptor without arguments (e.g. reversed) you need to:

1. Provide a range for your return type, for example:

```
#include <boost/range/iterator_range.hpp>
#include <boost/iterator/reverse_iterator.hpp>

template< typename R >
struct reverse_range :
    boost::iterator_range<
        boost::reverse_iterator<
            typename boost::range_iterator<R>::type> >
{
private:
    typedef boost::iterator_range<
        boost::reverse_iterator<
            typename boost::range_iterator<R>::type> > base;

public:
    typedef boost::reverse_iterator<
        typename boost::range_iterator<R>::type > iterator;

    reverse_range(R& r)
        : base(iterator(boost::end(r)), iterator(boost::begin(r)))
    { }
};
```

2. Provide a tag to uniquely identify your adaptor in the `operator|` function overload set

```
namespace detail {
    struct reverse_forwarder {};
}
```

3. Implement `operator|`

```
template< class BidirectionalRng >
inline reverse_range<BidirectionalRng>
operator|( BidirectionalRng& r, detail::reverse_forwarder )
{
 return reverse_range<BidirectionalRng>( r );
}

template< class BidirectionalRng >
inline reverse_range<const BidirectionalRng>
operator|( const BidirectionalRng& r, detail::reverse_forwarder )
{
 return reverse_range<const BidirectionalRng>( r );
}
```

4. Declare the adaptor itself (it is a variable of the tag type).

```
namespace
{
    const detail::reverse_forwarder reversed = detail::reverse_forwarder();
}
```

Method 3.2: Implement a Range Adaptor with arguments

1. Provide a range for your return type, for example:

```
#include <boost/range/adaptor/argument_fwd.hpp>
#include <boost/range/iterator_range.hpp>
#include <boost/iterator/transform_iterator.hpp>

template<typename Value>
class replace_value
{
public:
    typedef const Value& result_type;
    typedef const Value& argument_type;

    replace_value(const Value& from, const Value& to)
        : m_from(from), m_to(to)
    {
    }

    const Value& operator()(const Value& x) const
    {
        return (x == m_from) ? m_to : x;
    }
private:
    Value m_from;
    Value m_to;
};

template<typename Range>
class replace_range
: public boost::iterator_range<
    boost::transform_iterator<
        replace_value<typename boost::range_value<Range>::type>,
        typename boost::range_iterator<Range>::type> >
{
private:
    typedef typename boost::range_value<Range>::type value_type;
    typedef typename boost::range_iterator<Range>::type iterator_base;
    typedef replace_value<value_type> Fn;
    typedef boost::transform_iterator<Fn, iterator_base> replaced_iterator;
    typedef boost::iterator_range<replaced_iterator> base_t;

public:
    replace_range(Range& rng, value_type from, value_type to)
        : base_t(replaced_iterator(boost::begin(rng), Fn(from,to)),
                 replaced_iterator(boost::end(rng), Fn(from,to)))
     {
     }
 };
```

2. Implement a holder class to hold the arguments required to construct the RangeAdaptor. The holder combines multiple parameters into one that can be passed as the right operand of `operator|()`.

```
template<typename T>
class replace_holder : public boost::range_detail::holder2<T>
{
public:
    replace_holder(const T& from, const T& to)
        : boost::range_detail::holder2<T>(from, to)
    { }
private:
    void operator=(const replace_holder&);
};
```

3. Define an instance of the holder with the name of the adaptor

```
static boost::range_detail::forwarder2<replace_holder>
replaced = boost::range_detail::forwarder2<replace_holder>();
```

4. Define `operator|`

```
template<typename SinglePassRange>
inline replace_range<SinglePassRange>
operator|(SinglePassRange& rng,
          const replace_holder<typename boost::range_value<SinglePassRange>::type>& f)
{
    return replace_range<SinglePassRange>(rng, f.val1, f.val2);
}

template<typename SinglePassRange>
inline replace_range<const SinglePassRange>
operator|(const SinglePassRange& rng,
          const replace_holder<typename boost::range_value<SinglePassRange>::type>& f)
{
    return replace_range<const SinglePassRange>(rng, f.val1, f.val2);
}
```

Terminology and style guidelines

The use of a consistent terminology is as important for Ranges and range-based algorithms as it is for iterators and iterator-based algorithms. If a conventional set of names are adopted, we can avoid misunderstandings and write generic function prototypes that are **self-documenting**.

Since ranges are characterized by a specific underlying iterator type, we get a type of range for each type of iterator. Hence we can speak of the following types of ranges:

- **Value access** category:
  - Readable Range
  - Writeable Range
  - Swappable Range
  - Lvalue Range
- **Traversal** category:
  - Single Pass Range
  - Forward Range
  - Bidirectional Range
  - Random Access Range

Notice how we have used the categories from the new style iterators.

Notice that an iterator (and therefore an range) has one **traversal** property and one or more properties from the **value access** category. So in reality we will mostly talk about mixtures such as

- Random Access Readable Writeable Range
- Forward Lvalue Range

By convention, we should always specify the **traversal** property first as done above. This seems reasonable since there will only be one **traversal** property, but perhaps many **value access** properties.

It might, however, be reasonable to specify only one category if the other category does not matter. For example, the `iterator_range` can be constructed from a Forward Range. This means that we do not care about what **value access** properties the Range has. Similarly, a Readable Range will be one that has the lowest possible **traversal** property (Single Pass).

As another example, consider how we specify the interface of `std::sort()`. Algorithms are usually more cumbersome to specify the interface of since both **traversal** and **value access** properties must be exactly defined. The iterator-based version looks like this:

```
template< class RandomAccessTraversalReadableWritableIterator >
void sort( RandomAccessTraversalReadableWritableIterator first,
           RandomAccessTraversalReadableWritableIterator last );
```

For ranges the interface becomes

```
template< class RandomAccessReadableWritableRange >
void sort( RandomAccessReadableWritableRange& r );
```

Library Headers

General

Header	Includes	Related Concept
`<boost/range.hpp>`	everything from Boost.Range version 1 (Boost versions 1.42 and below). Includes the core range functions and metafunctions, but excludes Range Adaptors and Range Algorithms.	-
`<boost/range/metafunctions.hpp>`	every metafunction	-
`<boost/range/functions.hpp>`	every function	-
`<boost/range/value_type.hpp>`	`range_value`	Single Pass Range
`<boost/range/iterator.hpp>`	`range_iterator`	Single Pass Range
`<boost/range/difference_type.hpp>`	`range_difference`	Forward Range
`<boost/range/pointer.hpp>`	`range_pointer`	-
`<boost/range/category.hpp>`	`range_category`	-
`<boost/range/reverse_iterator.hpp>`	`range_reverse_iterator`	Bidirectional Range
`<boost/range/begin.hpp>`	`begin` and `const_begin`	Single Pass Range
`<boost/range/end.hpp>`	`end` and `const_end`	Single Pass Range
`<boost/range/empty.hpp>`	`empty`	Single Pass Range
`<boost/range/distance.hpp>`	`distance`	Forward Range
`<boost/range/size.hpp>`	`size`	Random Access Range
`<boost/range/rbegin.hpp>`	`rbegin` and `const_rbegin`	Bidirectional Range
`<boost/range/rend.hpp>`	`rend` and `const_rend`	Bidirectional Range
`<boost/range/as_array.hpp>`	`as_array`	-
`<boost/range/as_literal.hpp>`	`as_literal`	-
`<boost/range/iterator_range.hpp>`	`iterator_range`	-
`<boost/range/sub_range.hpp>`	`sub_range`	-
`<boost/range/concepts.hpp>`	Range concepts	-
`<boost/range/adaptors.hpp>`	every range adaptor	-
`<boost/range/algorithm.hpp>`	every range equivalent of an STL algorithm	-

Header	Includes	Related Concept
<boost/range/algorithm_ext.hpp>	every range algorithm that is an extension of the STL algorithms	-
<boost/range/counting_range.hpp>	counting_range	-
<boost/range/istream_range.hpp>	istream_range	-
<boost/range/irange.hpp>	irange	-
<boost/range/join.hpp>	join	-

Adaptors

Header	Includes
<boost/range/adaptor/adjacent_filtered.hpp>	adjacent_filtered
<boost/range/adaptor/copied.hpp>	copied
<boost/range/adaptor/filtered.hpp>	filtered
<boost/range/adaptor/indexed.hpp>	indexed
<boost/range/adaptor/indirected.hpp.	indirected
<boost/range/adaptor/map.hpp>	map_keys map_values
<boost/range/adaptor/replaced.hpp>	replaced
<boost/range/adaptor/replaced_if.hpp>	replaced_if
<boost/range/adaptor/reversed.hpp>	reversed
<boost/range/adaptor/sliced.hpp>	sliced
<boost/range/adaptor/strided.hpp>	strided
<boost/range/adaptor/tokenized.hpp>	tokenized
<boost/range/adaptor/transformed.hpp>	transformed
<boost/range/adaptor/uniqued.hpp>	uniqued

Algorithm

Header	Includes
<boost/range/algorithm/adjacent_find.hpp>	adjacent_find
<boost/range/algorithm/binary_search.hpp>	binary_search
<boost/range/algorithm/copy.hpp>	copy
<boost/range/algorithm/copy_backward.hpp>	copy_backward
<boost/range/algorithm/count.hpp>	count
<boost/range/algorithm/count_if.hpp>	count_if
<boost/range/algorithm/equal.hpp>	equal
<boost/range/algorithm/equal_range.hpp>	equal_range
<boost/range/algorithm/fill.hpp>	fill
<boost/range/algorithm/fill_n.hpp>	fill_n
<boost/range/algorithm/find.hpp>	find
<boost/range/algorithm/find_end.hpp>	find_end
<boost/range/algorithm/find_first_of.hpp>	find_first_of
<boost/range/algorithm/find_if.hpp>	find_if
<boost/range/algorithm/for_each.hpp>	for_each
<boost/range/algorithm/generate.hpp>	generate
<boost/range/algorithm/heap_algorithm.hpp>	push_heap pop_heap make_heap sort_heap
<boost/range/algorithm/inplace_merge.hpp>	inplace_merge
<boost/range/algorithm/lexicographical_compare.hpp>	lexicographical_compare
<boost/range/algorithm/lower_bound.hpp>	lower_bound
<boost/range/algorithm/max_element.hpp>	max_element
<boost/range/algorithm/merge.hpp>	merge
<boost/range/algorithm/min_element.hpp>	min_element
<boost/range/algorithm/mismatch.hpp>	mismatch
<boost/range/algorithm/nth_element.hpp>	nth_element
<boost/range/algorithm/partial_sort.hpp>	partial_sort
<boost/range/algorithm/partition.hpp>	partition
<boost/range/algorithm/permutation.hpp>	next_permutation prev_permutation

Header	Includes
<boost/range/algorithm/random_shuffle.hpp>	random_shuffle
<boost/range/algorithm/remove.hpp>	remove
<boost/range/algorithm/remove_copy.hpp>	remove_copy
<boost/range/algorithm/remove_copy_if.hpp>	remove_copy_if
<boost/range/algorithm/remove_if.hpp>	remove_if
<boost/range/algorithm/replace.hpp>	replace
<boost/range/algorithm/replace_copy.hpp>	replace_copy
<boost/range/algorithm/replace_copy_if.hpp>	replace_copy_if
<boost/range/algorithm/replace_if.hpp>	replace_if
<boost/range/algorithm/reverse.hpp>	reverse
<boost/range/algorithm/reverse_copy.hpp>	reverse_copy
<boost/range/algorithm/rotate.hpp>	rotate
<boost/range/algorithm/rotate_copy.hpp>	rotate_copy
<boost/range/algorithm/search.hpp>	search
<boost/range/algorithm/search_n.hpp>	search_n
<boost/range/algorithm/set_algorithm.hpp>	includes set_union set_intersection set_difference set_symmetric_difference
<boost/range/algorithm/sort.hpp>	sort
<boost/range/algorithm/stable_partition.hpp>	stable_partition
<boost/range/algorithm/swap_ranges.hpp>	swap_ranges
<boost/range/algorithm/transform.hpp>	transform
<boost/range/algorithm/unique.hpp>	unique
<boost/range/algorithm/unique_copy.hpp>	unique_copy
<boost/range/algorithm/upper_bound.hpp>	upper_bound

Algorithm Extensions

Header	Includes
`<boost/range/algorithm_ext/copy_n.hpp>`	copy_n
`<boost/range/algorithm_ext/erase.hpp>`	erase
`<boost/range/algorithm_ext/for_each.hpp>`	for_each
`<boost/range/algorithm_ext/insert.hpp>`	insert
`<boost/range/algorithm_ext/iota.hpp>`	iota
`<boost/range/algorithm_ext/is_sorted.hpp>`	is_sorted
`<boost/range/algorithm_ext/overwrite.hpp>`	overwrite
`<boost/range/algorithm_ext/push_back.hpp>`	push_back
`<boost/range/algorithm_ext/push_front.hpp>`	push_front

Examples

Some examples are given in the accompanying test files:

- string.cpp shows how to implement a container version of `std::find()` that works with `char[]`,`wchar_t[]`,`char*`,`wchar_t*`.
- algorithm_example.cpp shows the replace example from the introduction.
- iterator_range.cpp
- sub_range.cpp
- iterator_pair.cpp
- reversible_range.cpp
- std_container.cpp
- array.cpp

MFC/ATL (courtesy of Shunsuke Sogame)

Introduction

This implementation was kindly donated by Shunsuke Sogame. This header adapts MFC and ATL containers to the appropriate Range concepts.

Author:	Shunsuke Sogame
Contact:	mb2act@yahoo.co.jp
Date:	26th of May 2006
Copyright:	Shunsuke Sogame 2005-2006. Use, modification and distribution is subject to the Boost Software License, Version 1.0

Overview

Boost.Range MFC/ATL Extension provides Boost.Range support for MFC/ATL collection and string types.

```
CTypedPtrArray<CPtrArray, CList<CString> *> myArray;
...
BOOST_FOREACH (CList<CString> *theList, myArray)
{
    BOOST_FOREACH (CString& str, *theList)
    {
        boost::to_upper(str);
        std::sort(boost::begin(str), boost::end(str));
        ...
    }
}
```

Requirements

- Boost C++ Libraries Version 1.34.0 or later (no compilation required)
- Visual C++ 7.1 or later (for MFC and ATL)

MFC Ranges

If the `<boost/range/mfc.hpp>` is included before or after Boost.Range headers, the MFC collections and strings become models of Range. The table below lists the Traversal Category and `range_reference` of MFC ranges.

Range	Traversal Category	`range_reference<Range>::type`
`CArray<T,A>`	Random Access Range	`T&`
`CList<T,A>`	Bidirectional Range	`T&`
`CMap<K,AK,M,AM>`	Forward Range	`Range::CPair&`
`CTypedPtrArray<B,T*>`	Random Access Range	`T* const`
`CTypedPtrList<B,T*>`	Bidirectional Range	`T* const`
`CTypedPtrMap<B,T*,V*>`	Forward Range	`std::pair<T*,V*> const`
`CByteArray`	Random Access Range	`BYTE&`
`CDWordArray`	Random Access Range	`DWORD&`
`CObArray`	Random Access Range	`CObject*&`
`CPtrArray`	Random Access Range	`void*&`
`CStringArray`	Random Access Range	`CString&`
`CUIntArray`	Random Access Range	`UINT&`
`CWordArray`	Random Access Range	`WORD&`
`CObList`	Bidirectional Range	`CObject*&`
`CPtrList`	Bidirectional Range	`void*&`
`CStringList`	Bidirectional Range	`CString&`
`CMapPtrToWord`	Forward Range	`std::pair<void*,WORD> const`
`CMapPtrToPtr`	Forward Range	`std::pair<void*,void*> const`
`CMapStringToOb`	Forward Range	`std::pair<String,CObject*> const`
`CMapStringToString`	Forward Range	`Range::CPair&`
`CMapWordToOb`	Forward Range	`std::pair<WORD,CObject*> const`
`CMapWordToPtr`	Forward Range	`std::pair<WORD,void*> const`

Other Boost.Range metafunctions are defined by the following. Let `Range` be any type listed above and `Ref` be the same as `range_reference<Range>::type`. `range_value<Range>::type` is the same as `remove_reference<remove_const<Ref>::type>::type`, `range_difference<Range>::type` is the same as `std::ptrdiff_t`, and `range_pointer<Range>::type` is the same as `add_pointer<remove_reference<Ref>::type>::type`. As for `const Range`, see below.

ATL Ranges

If the `<boost/range/atl.hpp>` is included before or after Boost.Range headers, the ATL collections and strings become models of Range. The table below lists the Traversal Category and `range_reference` of ATL ranges.

Range	Traversal Category	`range_reference<Range>::type`
`CAtlArray<E,ET>`	Random Access Range	`E&`
`CAutoPtrArray<E>`	Random Access Range	`E&`
`CInterfaceArray<I,pi>`	Random Access Range	`CComQIPtr<I,pi>&`
`CAtlList<E,ET>`	Bidirectional Range	`E&`
`CAutoPtrList<E>`	Bidirectional Range	`E&`
`CHeapPtrList<E,A>`	Bidirectional Range	`E&`
`CInterfaceList<I,pi>`	Bidirectional Range	`CComQIPtr<I,pi>&`
`CAtlMap<K,V,KT,VT>`	Forward Range	`Range::CPair&`
`CRBTree<K,V,KT,VT>`	Bidirectional Range	`Range::CPair&`
`CRBMap<K,V,KT,VT>`	Bidirectional Range	`Range::CPair&`
`CRBMultiMap<K,V,KT,VT>`	Bidirectional Range	`Range::CPair&`
`CSimpleStringT<B,b>`	Random Access Range	`B&`
`CStringT<B,ST>`	Random Access Range	`B&`
`CFixedStringT<S,n>`	Random Access Range	`range_reference<S>::type`
`CComBSTR`	Random Access Range	`OLECHAR&`
`CSimpleArray<T,TE>`	Random Access Range	`T&`

Other Boost.Range metafunctions are defined by the following. Let `Range` be any type listed above and `Ref` be the same as `range_reference<Range>::type`. `range_value<Range>::type` is the same as `remove_reference<Ref>::type`, `range_difference<Range>::type` is the same as `std::ptrdiff_t`, and `range_pointer<Range>::type` is the same as `add_pointer<remove_reference<Ref>::type>::type`. As for `const Range`, see below.

const Ranges

`range_reference<const Range>::type` is defined by the following algorithm. Let `Range` be any type listed above and `Ref` be the same as `range_reference<Range>::type`.

```
if (Range is CObArray || Range is CObList)
    return CObject const * &
else if (Range is CPtrArray || Range is CPtrList)
    return void const * &
else if (there is a type X such that X& is the same as Ref)
    return X const &
else if (there is a type X such that X* const is the same as Ref)
    return X const * const
else
    return Ref
```

Other Boost.Range metafunctions are defined by the following.

Range metafunction	Result
range_value<const Range>::type	range_value<Range>::type
range_difference<const Range>::type	std::ptrdiff_t
range_pointer<const Range>::type	add_pointer<remove_reference<range_reference<const Range>::type>::type>::type

References

1. Boost.Range

2. MFC Collection Classes

3. ATL Collection Classes

Upgrade version of Boost.Range

Upgrade from version 1.49

1. `size` now returns the type Rng::size_type if the range has size_type; otherwise range_size<Rng>::type is used. This is the distance type promoted to an unsigned type.

Upgrade from version 1.45

1. `size` in addition to supporting Random Access Range now also supports extensibility via calls to the unqualified `range_calculate_size(rng)` function.
2. strided now in addition to working with any RandomAccessRange additionally works for any SinglePassRange for which `boost::size(rng)` is valid.
3. strided no longer requires `distance(rng) % stride_size == 0` or `stride_size < distance(rng)`

Upgrade from version 1.42

New features:

1. Range adaptors
2. Range algorithms

Removed:

1. `iterator_range` no longer has a `is_singular` member function. The singularity restrictions have been removed from the `iterator_range` class since this added restrictions to ranges of iterators whose default constructors were not singular. Previously the `is_singular` member function always returned `false` in release build configurations, hence it is not anticipated that this interface change will produce difficulty in upgrading.

Upgrade from version 1.34

Boost version 1.35 introduced some larger refactorings of the library:

1. Direct support for character arrays was abandoned in favor of uniform treatment of all arrays. Instead string algorithms can use the new function `as_literal()`.
2. `size` now requires a Random Access Range. The old behavior is provided as `distance()`.
3. `range_size<T>::type` has been completely removed in favor of `range_difference<T>::type`
4. `boost_range_begin()` and `boost_range_end()` have been renamed `range_begin()` and `range_end()` respectively.
5. `range_result_iterator<T>::type` and `range_reverse_result_iterator<T>::type` have been renamed `range_iterator<T>::type` and `range_reverse_iterator<T>::type`.
6. The procedure that makes a custom type work with the library has been greatly simplified. See Extending the library for UDTs for details.

Portability

A huge effort has been made to port the library to as many compilers as possible.

Full support for built-in arrays require that the compiler supports class template partial specialization. For non-conforming compilers there might be a chance that it works anyway thanks to workarounds in the type traits library. Visual C++ 6/7.0 has a limited support for arrays: as long as the arrays are of built-in type it should work.

Notice also that some compilers cannot do function template ordering properly. In that case one must rely of `range_iterator` and a single function definition instead of overloaded versions for const and non-const arguments. So if one cares about old compilers, one should not pass rvalues to the functions.

For maximum portability you should follow these guidelines:

1. do not use built-in arrays,
2. do not pass rvalues to `begin()`, `end()` and `iterator_range` Range constructors and assignment operators,
3. use `const_begin()` and `const_end()` whenever your code by intention is read-only; this will also solve most rvalue problems,
4. do not rely on ADL:
   - if you overload functions, include that header before the headers in this library,
   - put all overloads in namespace boost.

FAQ

1. ***Why is there no difference between `range_iterator<C>::type` and `range_const_iterator<C>::type` for `std::pair<iterator, iterator>`?***

    In general it is not possible nor desirable to find a corresponding `const_iterator`. When it is possible to come up with one, the client might choose to construct a `std::pair<const_iterator,const_iterator>` object.

    Note that an `iterator_range` is somewhat more convenient than a `pair` and that a `sub_range` does propagate const-ness.

2. ***Why is there not supplied more types or more functions?***

    The library has been kept small because its current interface will serve most purposes. If and when a genuine need arises for more functionality, it can be implemented.

3. ***How should I implement generic algorithms for ranges?***

    One should always start with a generic algorithm that takes two iterators (or more) as input. Then use Boost.Range to build handier versions on top of the iterator based algorithm. Please notice that once the range version of the algorithm is done, it makes sense not to expose the iterator version in the public interface.

4. ***Why is there no Incrementable Range concept?***

    Even though we speak of incrementable iterators, it would not make much sense for ranges; for example, we cannot determine the size and emptiness of a range since we cannot even compare its iterators.

    Note also that incrementable iterators are derived from output iterators and so there exist no output range.

History and Acknowledgement

Version 1 - before Boost 1.43

The library have been under way for a long time. Dietmar Kühl originally intended to submit an `array_traits` class template which had most of the functionality present now, but only for arrays and standard containers.

Meanwhile work on algorithms for containers in various contexts showed the need for handling pairs of iterators, and string libraries needed special treatment of character arrays. In the end it made sense to formalize the minimal requirements of these similar concepts. And the results are the Range concepts found in this library.

The term Range was adopted because of paragraph 24.1/7 from the C++ standard:

Most of the library's algorithmic templates that operate on data structures have interfaces that use ranges. A range is a pair of iterators that designate the beginning and end of the computation. A range [i, i) is an empty range; in general, a range [i, j) refers to the elements in the data structure starting with the one pointed to by i and up to but not including the one pointed to by j. Range [i, j) is valid if and only if j is reachable from i. The result of the application of functions in the library to invalid ranges is undefined.

Special thanks goes to

- Pavol Droba for help with documentation and implementation
- Pavel Vozenilek for help with porting the library
- Jonathan Turkanis and John Torjo for help with documentation
- Hartmut Kaiser for being review manager
- Jonathan Turkanis for porting the lib (as far as possible) to vc6 and vc7.

The concept checks and their documentation was provided by Daniel Walker.

Version 2 - Boost 1.43 and beyond

This version introduced Range Adaptors and Range Algorithms. This version 2 is the result of a merge of all of the RangeEx features into Boost.Range.

There were an enormous number of very significant contributors through all stages of this library.

Prior to Boost.RangeEx there had been a number of Range library implementations, these include library implementations by Eric Niebler, Adobe, Shunsuke Sogame etc. Eric Niebler contributed the Range Adaptor idea which is arguably the single biggest innovation in this library. Inevitably a great deal of commonality evolved in each of these libraries, but a considerable amount of effort was expended to learn from all of the divergent techniques.

The people in the following list all made contributions in the form of reviews, user feedback, design suggestions, or defect detection:

- Thorsten Ottosen: review management, design advice, documentation feedback
- Eric Niebler: early implementation, and review feedback
- Joel de Guzman: review
- Mathias Gaunard: review
- David Abrahams: implementation advice
- Robert Jones: defect reports, usage feedback
- Sean Parent: contributed experience from the Adobe range library

- Arno Schoedl: implementations, and review
- Rogier van Dalen: review
- Vincente Botet: review, documentation feedback

Regardless of how I write this section it will never truly fairly capture the gratitude that I feel to all who have contributed. Thank you everyone.