# Boost.Range Documentation



## Table of Contents



Boost.Range is a collection of concepts and utilities that are particularly useful for specifying and implementing generic algorithms.

# 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 we are used to work with is standard library containers. However, one often finds it desirable to extend that code to work with 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

- correct handling of null-terminated strings

  **Warning: support for null-terminated strings is deprecated and will disappear in the next Boost release (1.34).**

- safe use of built-in arrays (for legacy code; why else would you use built-in arrays?)

Below are given a small example (the complete example can be found here ):

```
//
// example: extracting bounds in a generic algorithm
//
template< class ForwardReadableRange, class T >
inline typename boost::range_iterator< ForwardReadableRange >::type
find( ForwardReadableRange& c, const T& value )
{
   return std::find( boost::begin( c ), boost::end( c ), value );
}

template< class ForwardReadableRange, class T >
inline typename boost::range_const_iterator< ForwardReadableRange >::type
find( const ForwardReadableRange& c, const T& value )
{
   return std::find( boost::begin( c ), boost::end( c ), value );
}

//
// replace first value and return its index
//
template< class ForwardReadableWriteableRange, class T >
inline typename boost::range_size< ForwardReadableWriteableRange >::type
my_generic_replace( ForwardReadableWriteableRange& c, const T& value, const T& replacement )
{
   typename boost::range_iterator< ForwardReadableWriteableRange >::type found = find( c, value );

   if( found != boost::end( c ) )
       *found = replacement;
   return std::distance( boost::begin( c ), found );
}

//
// usage
//
const int N = 5;
std::vector<int> my_vector;
int values[] = { 1,2,3,4,5,6,7,8,9 };

my_vector.assign( values, boost::end( values ) );
typedef std::vector<int>::iterator iterator;
std::pair<iterator,iterator>       my_view( boost::begin( my_vector ),
                                            boost::begin( my_vector ) + N );
char  str_val[] = "a string";
char* str       = str_val;
```

```
std::cout << my_generic_replace( my_vector, 4, 2 );
std::cout << my_generic_replace( my_view, 4, 2 );
std::cout << my_generic_replace( str, 'a', 'b' );

// prints '3', '5' and '0'
```

By using the free-standing functions and metafunctions, the code automatically works for all the types supported by this library; now and in the future. Notice that we have to provide two version of `find()` since we cannot forward a non-const rvalue with reference arguments (see this article about The Forwarding Problem ).

# 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

| | | |
|---|---|---|
| Value type | `boost::range_value<X>::type` | The type of the object stored in a Range. |
| 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_const_iterator<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_const_iterator<X>::type` otherwise |
| End of range | `boost::end(a)` | `boost::range_iterator<X>::type` if a is mutable, `boost::range_const_iterator<X>::type` otherwise |
| Is range empty? | boost::empty(a) | Convertible to bool |

### 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::size(a) == 0`. |
| `boost::end(a)` | Returns an iterator pointing one past the last element in the Range. | `boost::end(a)` is past-the-end. |
| `boost::empty(a)` | Equivalent to `boost::begin(a) == boost::end(a)`. (But possibly faster.) | - |

### Complexity guarantees

All three functions are at most amortized linear time. For most practical purposes, one can expect `boost::begin(a)`, `boost::end(a)` and `boost::empty(a)` 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

Container

implementation of metafunctions

implementation of functions

# 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. |

## 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)` is at most amortized linear time.

## Invariants

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

## 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_const_reverse_iterator<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_const_reverse_iterator<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_const_reverse_iterator<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

# 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 insure 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.

```
function_requires<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.

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

Four types of objects are currently supported by the library:

- standard-like containers
- `std::pair<iterator,iterator>`
- null terminated strings (this includes `char[]`,`wchar_t[]`, `char*`, and `wchar_t*`)

  **Warning: support for null-terminated strings is deprecated and will disappear in the next Boost release (1.34).**

- 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.

# Synopsis

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

    template< class T >
    struct range_value;

    template< class T >
    struct range_iterator;

    template< class T >
    struct range_const_iterator;

    //
    // Forward Range metafunctions
    //

    template< class T >
    struct range_difference;

    template< class T >
    struct range_size;

    //
    // Bidirectional Range metafunctions
    //

    template< class T >
    struct range_reverse_iterator;

    template< class T >
    struct range_const_reverse_iterator;

    //
    // Special metafunctions
    //

    template< class T >
    struct range_result_iterator;

    template< class T >
    struct range_reverse_result_iterator;

    //
    // Single Pass Range functions
    //

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

    template< class T >
```

```
typename range_const_iterator<T>::type
begin( const T& c );

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

template< class T >
typename range_const_iterator<T>::type
end( const T& c );

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

//
// Forward Range functions
//

template< class T >
typename range_size<T>::type
size( const T& c );

//
// Bidirectional Range functions
//

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

template< class T >
typename range_const_reverse_iterator<T>::type
rbegin( const T& c );

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

template< class T >
typename range_const_reverse_iterator<T>::type
rend( const T& c );

//
// Special const Range functions
//

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

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

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

```
    template< class T >
    typename range_const_reverse_iterator<T>::type
    const_rend( 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*` |

Please notice in tables below that when four lines appear in a cell, the first line will describe the primary template, the second line pairs of iterators, the third line arrays and the last line null-terminated strings.

## Metafunctions

| Expression | Return type | Complexity |
|---|---|---|
| `range_value<X>::type` | `T::value_type`<br>`boost::iterator_value<P::first_type>::type`<br>`A`<br>`Char` | compile time |
| `range_iterator<X>::type` | `T::iterator`<br>`P::first_type`<br>`A*`<br>`Char*` | compile time |
| `range_const_iterator<X>::type` | `T::const_iterator`<br>`P::first_type`<br>`const A*`<br>`const Char*` | compile time |
| `range_difference<X>::type` | `T::difference_type`<br>`boost::iterator_difference<P::first_type>::type`<br>`std::ptrdiff_t`<br>`std::ptrdiff_t` | compile time |
| `range_size<X>::type` | `T::size_type`<br>`std::size_t`<br>`std::size_t`<br>`std::size_t` | compile time |
| `range_result_iterator<X>::type` | `range_const_iterator<X>::type` if `X` is `const`<br>`range_iterator<X>::type` otherwise | compile time |
| `range_reverse_iterator<X>::type` | `boost::reverse_iterator< typename range_iterator<T>::type >` | compile time |
| `range_const_reverse_iterator<X>::type` | `boost::reverse_iterator< typename range_const_iterator<T>::type >` | compile time |
| `range_reverse_result_iterator<X>::type` | `boost::reverse_iterator< typename range_result_iterator<T>::type >` | compile time |

The special metafunctions `range_result_iterator` and `range_reverse_result_iterator` are not part of any Range concept, but they are very useful when implementing certain Range classes like `sub_range` because of their ability to select iterators based on constness.

## Functions

| Expression | Return type | Returns | Complexity |
|---|---|---|---|
| `begin(x)` | `range_result_iterator<X>::type` | `p.first` if `p` is of type `std::pair<T>`<br>`a` if `a` is an array<br>`s` if `s` is a string literal<br>`boost_range_begin(x)` if that expression would invoke a function found by ADL<br>`t.begin()` otherwise | constant time |
| `end(x)` | `range_result_iterator<X>::type` | `p.second` if `p` is of type `std::pair<T>`<br>`a + sz` if `a` is an array of size `sz`<br>`s + std::char_traits<X>::length( s )` if `s` is a `Char*`<br>`s + sz - 1` if `s` is a string literal of size `sz`<br>`boost_range_end(x)` if that expression would invoke a function found by ADL<br>`t.end()` otherwise | linear if `X` is `Char*` constant time otherwise |
| `empty(x)` | `bool` | `begin(x) == end( x )` | linear if `X` is `Char*` constant time otherwise |
| `size(x)` | `range_size<X>::type` | `std::distance(p.first,p.second)` if `p` is of type `std::pair<T>`<br>`sz` if `a` is an array of size `sz`<br>`end(s) - s` if `s` is a string literal or a `Char*`<br>`boost_range_size(x)` if that expression would invoke a function found by ADL<br>`t.size()` otherwise | linear if `X` is `Char*` or if `std::distance()` is linear<br>constant time otherwise |
| `rbegin(x)` | `range_reverse_result_iterator<X>::type` | `range_reverse_result_iterator<X>::type( end(x) )` | same as `end(x)` |
| `rend(x)` | `range_reverse_result_iterator<X>::type` | `range_reverse_result_iterator<X>::type( begin(x) )` | same as `begin(x)` |
| `const_begin(x)` | `range_const_iterator<X>::type` | `range_const_iterator<X>::type( begin(x) )` | same as `begin(x)` |
| `const_end(x)` | `range_const_iterator<X>::type` | `range_const_iterator<X>::type( end(x) )` | same as `end(x)` |
| `const_rbegin(x)` | `range_const_reverse_iterator<X>::type` | `range_const_reverse_iterator<X>::type( rbegin(x) )` | same as `rbegin(x)` |
| `const_rend(x)` | `range_const_reverse_iterator<X>::type` | `range_const_reverse_iterator<X>::type( rend(x) )` | same as `rend(x)` |

The special const functions are not part of any Range concept, but are very useful when you want to document clearly that your code is read-only.

# 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 |
| `size()` | Forward 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 |
| `size_type` | Forward 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 |
|---|---|
| `boost_range_begin(x)` | Single Pass Range |
| `boost_range_end(x)` | Single Pass Range |
| `boost_range_size(x)` | Forward Range |

`boost_range_begin()` and `boost_range_end()` must be overloaded for both `const` and `mutable` reference arguments.

You must also specialize 3 metafunctions for your type `X`:

| Metafunction | Related concept |
|---|---|
| `boost::range_iterator` | Single Pass Range |
| `boost::range_const_iterator` | Single Pass Range |
| `boost::range_size` | Forward 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_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;
 };

 template< class T >
 struct range_size< Foo::Pair<T> >
 {

  typedef std::size_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 boost_range_begin( Pair<T>& x )
 {
  return x.first;
 }

 template< class T >
 inline T boost_range_begin( const Pair<T>& x )
 {
  return x.first;
 }

 template< class T >
 inline T boost_range_end( Pair<T>& x )
 {
  return x.last;
 }

 template< class T >
 inline T boost_range_end( const Pair<T>& x )
 {
  return x.last;
 }

 template< class T >
 inline typename boost::range_size< Pair<T> >::type
 boost_range_size( const Pair<T>& x )
 {
  return std::distance(x.first,x.last);
 }

} // namespace 'Foo'

#include <vector>

int main()
{
 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_size< Foo::Pair<iter> >::type s = boost::size( pair );
 s      = boost::size( cpair );
 boost::range_const_reverse_iterator< Foo::Pair<iter> >::type
 ri     = boost::rbegin( cpair ),
 re     = boost::rend( cpair );
}
```

# 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`

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 Forward 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 means that the `iterator_range` will still be usable in many contexts even though the iterators underneath are not.

## Synopsis

```
namespace boost
{
    template< class ForwardTraversalIterator >
    class iterator_range
    {
    public: // Forward Range types
        typedef ...                         value_type;
        typedef ...                         difference_type;
        typedef ...                         size_type;
        typedef ForwardTraversalIterator    iterator;
        typedef ForwardTraversalIterator    const_iterator;

    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;
        size_type size() const;
        bool      empty() const;

    public: // convenience
        operator    unspecified_bool_type() const;
        bool        equal( const iterator_range& ) const;
        value_type& front() const;
        value_type& back() const;
        // for Random Access Range only:
        value_type& operator[]( size_type at ) 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 iterator_of<ForwardRange>::type >
    make_iterator_range( ForwardRange& r );

    template< class ForwardRange >
    iterator_range< typename const_iterator_of<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_const_iterator<Range>::type >
    make_iterator_range( const Range& r,
                         typename range_difference<Range>::type advance_begin,
                         typename range_difference<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_result_iterator<ForwardRange>::type >
    {
    public:
        typedef typename range_result_iterator<ForwardRange>::type iterator;
        typedef typename range_const_iterator<ForwardRange>::type  const_iterator;

    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" );
```

# 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

| Header | Includes | Related Concept |
|---|---|---|
| <boost/range.hpp> | everything | - |
| <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/const_iterator.hpp> | range_const_iterator | Single Pass Range |
| <boost/range/difference_type.hpp> | range_difference | Forward Range |
| <boost/range/size_type.hpp> | range_size | Forward Range |
| <boost/range/result_iterator.hpp> | range_result_iterator | - |
| <boost/range/reverse_iterator.hpp> | range_reverse_iterator | Bidirectional Range |
| <boost/range/const_reverse_iterator.hpp> | range_const_reverse_iterator | _bidirectionalrange_ |
| <boost/range/reverse_result_iterator.hpp> | range_reverse_result_iterator | - |
| <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/size.hpp> | size | Forward Range |
| <boost/range/rbegin.hpp> | rbegin and const_rbegin | Bidirectional Range |
| <boost/range/rend.hpp> | rend and const_rend | Bidirectional Range |
| <boost/range/iterator_range.hpp> | iterator_range | - |
| <boost/range/sub_range.hpp> | sub_range | - |
| <boost/range/concepts.hpp> | concept checks | - |

# 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*.

**Warning: support for null-terminated strings is deprecated and will disappear in the next Boost release (1.34).**

- 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

# 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_result_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

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 sa possible) to vc6 and vc7.

The concept checks and their documentation was provided by Daniel Walker.