
Fusion 2.0

Joel de Guzman

Dan Marsden

Tobias Schwinger

Copyright © 2001-2007 Joel de Guzman, Dan Marsden, Tobias Schwinger

Distributed under the Boost Software License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)

Table of Contents

Preface	3
Introduction	4
Quick Start	5
Organization	8
Support	9
is_sequence	10
is_view	10
tag_of	11
category_of	12
deduce	14
deduce_sequence	15
pair	16
Iterator	18
Concepts	19
Forward Iterator	19
Bidirectional Iterator	21
Random Access Iterator	22
Associative Iterator	23
Functions	24
deref	24
next	25
prior	26
distance	27
advance	28
advance_c	29
deref_data	30
Operator	31
Operator *	31
Operator ==	32
Operator !=	32
Metafunctions	33
value_of	33
deref	34
next	35
prior	36
equal_to	37
distance	38
advance	39
advance_c	40
key_of	41

value_of_data	42
deref_data	43
Sequence	44
Concepts	44
Forward Sequence	44
Bidirectional Sequence	46
Random Access Sequence	47
Associative Sequence	49
Intrinsic	50
Functions	50
Metafunctions	60
Operator	73
I/O	73
Comparison	75
Container	81
vector	81
cons	83
list	84
set	86
map	88
Generation	89
Functions	89
MetaFunctions	97
Conversion	104
Functions	104
Metafunctions	107
View	110
single_view	110
filter_view	111
iterator_range	112
joint_view	113
zip_view	115
transform_view	116
reverse_view	118
nview	119
Adapted	121
std::pair	121
mpl sequence	122
boost::array	122
boost::tuple	123
BOOST_FUSION_ADAPT_STRUCT	123
BOOST_FUSION_ADAPT_ASSOC_STRUCT	124
Algorithm	125
Iteration	126
Functions	126
Metafunctions	130
Query	132
Functions	133
Metafunctions	140
Transformation	146
Functions	147
Metafunctions	167
Tuple	184
Class template tuple	184
Construction	185
Tuple creation functions	186
Tuple helper classes	186
Element access	186

Relational operators	187
Pairs	188
Extension	189
The Full Extension Mechanism	189
Sequence Facade	195
Iterator Facade	197
Functional	199
Concepts	200
Callable Object	200
Regular Callable Object	200
Deferred Callable Object	201
Polymorphic Function Object	202
Invocation	203
Functions	203
Metafunctions	207
Limits	209
Adapters	209
fused	209
fused_procedure	211
fused_function_object	212
unfused	214
unfused_typed	216
Limits	219
Generation	219
Functions	219
Metafunctions	223
Notes	225
Change log	228
Acknowledgements	228
References	228

Preface

“Algorithms + Data Structures = Programs.”

--Niklaus Wirth

Description

Fusion is a library for working with heterogenous collections of data, commonly referred to as tuples. A set of containers (vector, list, set and map) is provided, along with views that provide a transformed presentation of their underlying data. Collectively the containers and views are referred to as sequences, and Fusion has a suite of algorithms that operate upon the various sequence types, using an iterator concept that binds everything together.

The architecture is modeled after [MPL](#) which in turn is modeled after [STL](#). It is named "fusion" because the library is a "fusion" of compile time metaprogramming with runtime programming.

Motivation

Tuples are powerful beasts. After having developed two significant projects ([Spirit](#) and [Phoenix](#)) that relied heavily metaprogramming, it became apparent that tuples are a powerful means to simplify otherwise tricky tasks; especially those that require a combination of metaprogramming and manipulation of heterogenous data types with values. While [MPL](#) is an extremely powerful metaprogramming tool, [MPL](#) focuses on type manipulation only. Ultimately, you'll have to map these types to real values to make them useful in the runtime world where all the real action takes place.





As [Spirit](#) and [Phoenix](#) evolved, patterns and idioms related to tuple manipulation emerged. Soon, it became clear that those patterns and idioms were best assembled in a tuples algorithms library. [David Abrahams](#) outlined such a scheme in 2002. At that time, it just

so happened that [Spirit](#) and [Phoenix](#) had an adhoc collection of tuple manipulation and traversal routines. It was an instant *AHA!* moment.

How to use this manual

Some icons are used to mark certain topics indicative of their relevance. These icons precede some text to indicate:

Table 1. Icons

Icon	Name	Meaning
	Note	Information provided is auxiliary but will give the reader a deeper insight into a specific topic. May be skipped.
	Alert	Information provided is of utmost importance.
	Caution	A mild warning.
	Tip	A potentially useful and helpful piece of information.

This documentation is automatically generated by Boost QuickBook documentation tool. QuickBook can be found in the [Boost Tools](#).

Support

Please direct all questions to Spirit's mailing list. You can subscribe to the [Spirit Mailing List](#). The mailing list has a searchable archive. A search link to this archive is provided in [Spirit's](#) home page. You may also read and post messages to the mailing list through [Spirit General NNTP news portal](#) (thanks to [Gmane](#)). The news group mirrors the mailing list. Here is a link to the archives: <http://news.gmane.org/gmane.comp.parsers.spirit.general>.

Introduction

An advantage other languages such as Python and Lisp/ Scheme, ML and Haskell, etc., over C++ is the ability to have heterogeneous containers that can hold arbitrary element types. All the containers in the standard library can only hold a specific type. A `vector<int>` can only hold ints. A `list<X>` can only hold elements of type `X`, and so on.

True, you can use inheritance to make the containers hold different types, related through subclassing. However, you have to hold the objects through a pointer or smart reference of some sort. Doing this, you'll have to rely on virtual functions to provide polymorphic behavior since the actual type is erased as soon as you store a pointer to a derived class to a pointer to its base. The held objects must be related: you cannot hold objects of unrelated types such as `char`, `int`, `class X`, `float`, etc. Oh sure you can use something like [Boost.Any](#) to hold arbitrary types, but then you pay more in terms of runtime costs and due to the fact that you practically erased all type information, you'll have to perform dangerous casts to get back the original type.

The [Boost.Tuple](#) library written by [Jaakko Jarvi](#) provides heterogeneous containers in C++. The `tuple` is a basic data structure that can hold heterogeneous types. It's a good first step, but it's not complete. What's missing are the algorithms. It's nice that we can store and retrieve data to and from tuples, pass them around as arguments and return types. As it is, the [Boost.Tuple](#) facility is already very useful. Yet, as soon as you use it more often, usage patterns emerge. Eventually, you collect these patterns into algorithm libraries.

Hmmm, kinda reminds us of STL right? Right! Can you imagine how it would be like if you used STL without the algorithms? Everyone will have to reinvent their own *algorithm* wheels.

Fusion is a library and a framework similar to both [STL](#) and the boost [MPL](#). The structure is modeled after [MPL](#), which is modeled after [STL](#). It is named "fusion" because the library is reminiscent of the "fusion" of compile time meta-programming with runtime programming. The library inherently has some interesting flavors and characteristics of both [MPL](#) and [STL](#). It lives in the twilight zone between compile time meta-programming and run time programming. [STL](#) containers work on values. [MPL](#) containers work on types. Fusion containers work on both types and values.

Unlike [MPL](#), Fusion algorithms are lazy and non sequence-type preserving. What does that mean? It means that when you operate on a sequence through a Fusion algorithm that returns a sequence, the sequence returned may not be of the same class as the original. This is by design. Runtime efficiency is given a high priority. Like [MPL](#), and unlike [STL](#), fusion algorithms are functional in nature such that algorithms are non mutating (no side effects). However, due to the high cost of returning full sequences such as vectors and lists, *Views* are returned from Fusion algorithms instead. For example, the [transform](#) algorithm does not actually return a transformed version of the original sequence. [transform](#) returns a [transform_view](#). This view holds a reference to the original sequence plus the transform function. Iteration over the [transform_view](#) will apply the transform function over the sequence elements on demand. This *lazy* evaluation scheme allows us to chain as many algorithms as we want without incurring a high runtime penalty.

The *lazy* evaluation scheme where algorithms return views allows operations such as [push_back](#) to be totally generic. In Fusion, [push_back](#) is actually a generic algorithm that works on all sequences. Given an input sequence *s* and a value *x*, Fusion's [push_back](#) algorithm simply returns a [joint_view](#): a view that holds a reference to the original sequence *s* and the value *x*. Functions that were once sequence specific and need to be implemented N times over N different sequences are now implemented only once.

Fusion provides full round compatibility with [MPL](#). Fusion sequences are fully conforming [MPL](#) sequences and [MPL](#) sequences are fully compatible with Fusion. You can work with Fusion sequences on [MPL](#) if you wish to work solely on types ¹. In [MPL](#), Fusion sequences follow [MPL](#)'s sequence-type preserving semantics (i.e. algorithms preserve the original sequence type. e.g. transforming a vector returns a vector). You can also convert from an [MPL](#) sequence to a Fusion sequence. For example, there are times when it is convenient to work solely on [MPL](#) using pure [MPL](#) sequences, then, convert them to Fusion sequences as a final step before actual instantiation of real runtime objects with data. You have the best of both worlds.

Quick Start

I assume the reader is already familiar with tuples ([Boost.Tuple](#)) and its ancestor `std::pair`. The tuple is a generalization of `std::pair` for multiple heterogeneous elements (triples, quadruples, etc.). The tuple is more or less a synonym for fusion's [vector](#).

For starters, we shall include all of Fusion's [Sequence\(s\)](#) ²:

```
#include <boost/fusion/sequence.hpp>
#include <boost/fusion/include/sequence.hpp>
```

Let's begin with a [vector](#) ³:

```
vector<int, char, std::string> stuff(1, 'x', "howdy");
int i = at_c<0>(stuff);
char ch = at_c<1>(stuff);
std::string s = at_c<2>(stuff);
```

Just replace tuple for [vector](#) and get for `at_c` and this is exactly like [Boost.Tuple](#). Actually, either names can be used interchangeably. Yet, the similarity ends there. You can do a lot more with Fusion [vector](#) or tuple. Let's see some examples.

Print the vector as XML

First, let's include the algorithms:

¹ Choose [MPL](#) over fusion when doing pure type calculations. Once the static type calculation is finished, you can instantiate a fusion sequence (see [Conversion](#)) for the runtime part.

² There are finer grained header files available if you wish to have more control over which components to include (see section [Organization](#) for details).

³ Unless otherwise noted, components are in namespace `boost::fusion`. For the sake of simplicity, code in this quick start implies `using` directives for the fusion components we will be using.

```
#include <boost/fusion/algorithm.hpp>
#include <boost/fusion/include/algorithm.hpp>
```

Now, let's write a function object that prints XML of the form `<type>data</type>` for each member in the tuple.

```
struct print_xml
{
    template <typename T>
    void operator()(T const& x) const
    {
        std::cout
            << '<' << typeid(x).name() << '>'
            << x
            << "</" << typeid(x).name() << '>'
            ;
    }
};
```

Now, finally:

```
for_each(stuff, print_xml());
```

That's it! `for_each` is a fusion algorithm. It is a generic algorithm similar to STL's. It iterates over the sequence and calls a user supplied function. In our case, it calls `print_xml`'s `operator()` for each element in `stuff`.



Caution

The result of `typeid(x).name()` is platform specific. The code here is just for exposition. Of course you already know that :-)

`for_each` is generic. With `print_xml`, you can use it to print just about any Fusion [Sequence](#).

Print only pointers

Let's get a little cleverer. Say we wish to write a *generic* function that takes in an arbitrary sequence and XML prints only those elements which are pointers. Ah, easy. First, let's include the `is_pointer` boost type trait:

```
#include <boost/type_traits/is_pointer.hpp>
```

Then, simply:

```
template <typename Sequence>
void xml_print_pointers(Sequence const& seq)
{
    for_each(filter_if<boost::is_pointer<_> >(seq), print_xml());
}
```

`filter_if` is another Fusion algorithm. It returns a `filter_view`, a conforming Fusion sequence. This view reflects only those elements that pass the given predicate. In this case, the predicate is `boost::is_pointer<_>`. This "filtered view" is then passed to the `for_each` algorithm, which then prints the "filtered view" as XML.

Easy, right?

Associative tuples

Ok, moving on...

Apart from `vector`, fusion has a couple of other sequence types to choose from. Each sequence has its own characteristics. We have `list`, `set`, `map`, plus a multitude of views that provide various ways to present the sequences.

Fusion's `map` associate types with elements. It can be used as a cleverer replacement of the `struct`. Example:

```
namespace fields
{
    struct name;
    struct age;
}

typedef map<
    fusion::pair<fields::name, std::string>
    , fusion::pair<fields::age, int> >
person;
```

`map` is an associative sequence. Its elements are Fusion pairs which differ somewhat from `std::pair`. Fusion pairs only contain one member, with the type of their second template parameter. The first type parameter of the pair is used as an index to the associated element in the sequence. For example, given a `a_person` of type `person`, you can do:

```
using namespace fields;
std::string person_name = at_key<name>(a_person);
int person_age = at_key<age>(a_person);
```

Why go through all this trouble, you say? Well, for one, unlike the `struct`, we are dealing with a generic data structure. There are a multitude of facilities available at your disposal provided out of the box with fusion or written by others. With these facilities, introspection comes for free, for example. We can write one serialization function (well, two, if you consider loading and saving) that will work for all your fusion `maps`. Example:

```
struct saver
{
    template <typename Pair>
    void operator()(Pair const& data) const
    {
        some_archive << data.second;
    }
};

template <typename Stuff>
void save(Stuff const& stuff)
{
    for_each(stuff, saver());
}
```

The `save` function is generic and will work for all types of `stuff` regardless if it is a person, a dog or a whole `alternate_universe`.

Tip of the Iceberg

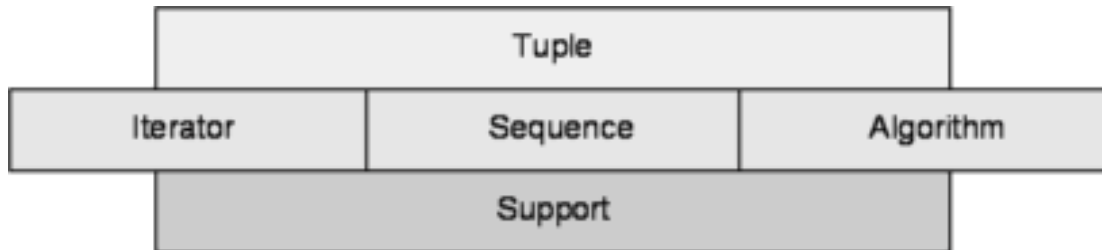
And... we've barely scratched the surface! You can compose and expand the data structures, remove elements from the structures, find specific data types, query the elements, filter out types for inspection, transform data structures, etc. What you've seen is just the tip of the iceberg.

Organization

The library is organized into layers of modules, with each module addressing a particular area of responsibility. A module may not depend on modules in higher layers.

The library is organized in three layers:

Layers



The entire library is found in the "boost/fusion" directory. Modules are organized in directories. Each module has its own header file placed in the same directory with the actual module-directory. For example, there exists "boost/fusion/support.hpp" in the same directory as "boost/fusion/support". Everything, except those found inside "detail" directories, is public.

There is also a "boost/fusion/include/" directory that contains all the headers to all the components and modules. If you are unsure where to find a specific component or module, or don't want to fuss with hierarchy and nesting, use this.

The library is header-only. There is no need to build object files to link against.

Directory

- tuple
- algorithm
 - iteration
 - query
 - transformation
- adapted
 - array
 - mpl
 - boost::tuple
 - std_pair
 - struct
 - variant
- view
 - filter_view
 - iterator_range
 - joint_view

- `reverse_view`
- `single_view`
- `transform_view`
- `zip_view`
- `container`
 - `deque`
 - `list`
 - `map`
 - `set`
 - `vector`
 - `generation`
- `mpl`
- `functional`
- `sequence`
 - `comparison`
 - `intrinsic`
 - `io`
- `iterator`
- `support`

Example

If, for example, you want to use `list`, depending on the granularity that you desire, you may do so by including one of

```
#include <boost/fusion/container.hpp>
#include <boost/fusion/include/container.hpp>
#include <boost/fusion/container/list.hpp>
#include <boost/fusion/include/list.hpp>
```

The first includes all containers The second includes only `list` ⁴.

Support

A couple of classes and metafunctions provide basic support for Fusion.

⁴ Modules may contain smaller components. Header file information for each component will be provided as part of the component's documentation.

is_sequence

Description

Metafunction that evaluates to `mpl::true_` if a certain type `T` is a conforming Fusion [Sequence](#), `mpl::false_` otherwise. This may be specialized to accomodate clients which provide Fusion conforming sequences.

Synopsis

```
namespace traits
{
    template <typename T>
    struct is_sequence
    {
        typedef unspecified type;
    };
}
```

Parameters

Parameter	Requirement	Description
<code>T</code>	Any type	The type to query.

Expression Semantics

```
typedef traits::is_sequence<T>::type c;
```

Return type: An [MPL Boolean Constant](#).

Semantics: Metafunction that evaluates to `mpl::true_` if a certain type `T` is a conforming Fusion sequence, `mpl::false_` otherwise.

Header

```
#include <boost/fusion/support/is_sequence.hpp>
#include <boost/fusion/include/is_sequence.hpp>
```

Example

```
BOOST_MPL_ASSERT_NOT(( traits::is_sequence< std::vector<int> > ));
BOOST_MPL_ASSERT_NOT(( is_sequence< int > ));
BOOST_MPL_ASSERT(( traits::is_sequence<list<> > ));
BOOST_MPL_ASSERT(( traits::is_sequence<list<int> > ));
BOOST_MPL_ASSERT(( traits::is_sequence<vector<> > ));
BOOST_MPL_ASSERT(( traits::is_sequence<vector<int> > ));
```

is_view

Description

Metafunction that evaluates to `mpl::true_` if a certain type `T` is a conforming Fusion [View](#), `mpl::false_` otherwise. A view is a specialized sequence that does not actually contain data. Views hold sequences which may be other views. In general, views are held by other views by value, while non-views are held by other views by reference. `is_view` may be specialized to accomodate clients providing Fusion conforming views.

Synopsis

```
namespace traits
{
    template <typename T>
    struct is_view
    {
        typedef unspecified type;
    };
}
```

Parameters

Parameter	Requirement	Description
T	Any type	The type to query.

Expression Semantics

```
typedef traits::is_view<T>::type c;
```

Return type: An [MPL Boolean Constant](#).

Semantics: Metafunction that evaluates to `mpl::true_` if a certain type T is a conforming Fusion view, `mpl::false_` otherwise.

Header

```
#include <boost/fusion/support/is_view.hpp>
#include <boost/fusion/include/is_view.hpp>
```

Example

```
BOOST_MPL_ASSERT_NOT(( traits::is_view<std::vector<int> > ));
BOOST_MPL_ASSERT_NOT(( traits::is_view<int> ));

using boost::mpl::_;
using boost::is_pointer;
typedef vector<int*, char, long*, bool, double> vector_type;
typedef filter_view<vector_type, is_pointer<_> > filter_view_type;
BOOST_MPL_ASSERT(( traits::is_view<filter_view_type> ));
```

tag_of

Description

All conforming Fusion sequences and iterators have an associated tag type. The purpose of the tag is to enable *tag dispatching* from [Intrinsic](#) functions to implementations appropriate for the type.

This metafunction may be specialized to accomodate clients providing Fusion conforming sequences.

Synopsis

```
namespace traits
{
    template<typename Sequence>
    struct tag_of
    {
        typedef unspecified type;
    };
}
```

Parameters

Parameter	Requirement	Description
T	Any type	The type to query.

Expression Semantics

```
typedef traits::tag_of<T>::type tag;
```

Return type: Any type.

Semantics: Returns the tag type associated with T.

Header

```
#include <boost/fusion/support/tag_of.hpp>
#include <boost/fusion/include/tag_of.hpp>
```

Example

```
typedef traits::tag_of<list<> >::type tag1;
typedef traits::tag_of<list<int> >::type tag2;
typedef traits::tag_of<vector<> >::type tag3;
typedef traits::tag_of<vector<int> >::type tag4;

BOOST_MPL_ASSERT((boost::is_same<tag1, tag2>));
BOOST_MPL_ASSERT((boost::is_same<tag3, tag4>));
```

category_of

Description

A metafunction that establishes the conceptual classification of a particular [Sequence](#) or [Iterator](#) (see [Iterator Concepts](#) and [Sequence Concepts](#)).

Synopsis

```
namespace traits
{
    template <typename T>
    struct category_of
    {
        typedef unspecified type;
    };
}
```

Parameters

Parameter	Requirement	Description
T	Any type	The type to query.

Expression Semantics

```
typedef traits::category_of<T>::type category;
```

Return type:

The return type is derived from one of:

```
namespace boost { namespace fusion
{
    struct incrementable_traversal_tag {};

    struct single_pass_traversal_tag
        : incrementable_traversal_tag {};

    struct forward_traversal_tag
        : single_pass_traversal_tag {};

    struct bidirectional_traversal_tag
        : forward_traversal_tag {};

    struct random_access_traversal_tag
        : bidirectional_traversal_tag {};
}}
```

And optionally from:

```
namespace boost { namespace fusion
{
    struct associative_tag {};
}}
```

Semantics: Establishes the conceptual classification of a particular [Sequence](#) or [Iterator](#).

Header

```
#include <boost/fusion/support/category_of.hpp>
#include <boost/fusion/include/category_of.hpp>
```

Example

```
using boost::is_base_of;
typedef traits::category_of<list<> >::type list_category;
typedef traits::category_of<vector<> >::type vector_category;
BOOST_MPL_ASSERT(( is_base_of<forward_traversal_tag, list_category> ));
BOOST_MPL_ASSERT(( is_base_of<random_access_traversal_tag, vector_category> ));
```

deduce

Description

Metafunction to apply *element conversion* to the full argument type.

It removes references to `const`, references to array types are kept, even if the array is `const`. Reference wrappers are removed (see `boost::ref`).

Header

```
#include <boost/fusion/support/deduce.hpp>
#include <boost/fusion/include/deduce.hpp>
```

Synopsis

```
namespace traits
{
    template <typename T>
    struct deduce
    {
        typedef unspecified type;
    };
}
```

Example

```
template <typename T>
struct holder
{
    typename traits::deduce<T const &>::type element;

    holder(T const & a)
        : element(a)
    { }
};

template <typename T>
holder<T> make_holder(T const & a)
{
    return holder<T>(a);
}
```

See also

- [deduce_sequence](#)

deduce_sequence

Description

Applies *element conversion* to each element in a [Forward Sequence](#). The resulting type is a [Random Access Sequence](#) that provides a converting constructor accepting the original type as its argument.

Header

```
#include <boost/fusion/support/deduce_sequence.hpp>
#include <boost/fusion/include/deduce_sequence.hpp>
```

Synopsis

```
namespace traits
{
    template <class Sequence>
    struct deduce_sequence
    {
        typedef unspecified type;
    };
}
```

Example

```
template <class Seq>
struct holder
{
    typename traits::deduce_sequence<Seq>::type element;

    holder(Seq const & a)
        : element(a)
    { }
};

template <typename T0, typename T1>
holder< vector<T0 const &, T1 const &> >
make_holder(T0 const & a0, T1 const & a1)
{
    typedef vector<T0 const &, T1 const &> arg_vec_t;
    return holder<arg_vec_t>( arg_vec_t(a0,a1) );
}
```

See also

- [deduce](#)

pair

Description

Fusion `pair` type is a half runtime pair. A half runtime pair is similar to a `std::pair`, but, unlike `std::pair`, the first type does not have data. It is used as elements in `maps`, for example.

Synopsis

```
template <typename First, typename Second>
struct pair;

namespace result_of
{
    template <typename Pair>
    struct first;

    template <typename Pair>
    struct second;

    template <typename First, typename Second>
    struct make_pair;
}

template <typename First, typename Second>
typename result_of::make_pair<First,Second>::type
make_pair(Second const &);
```

Template parameters

Parameter	Description
First	The first type. This is purely a type. No data is held.
Second	The second type. This contains data.

Notation

P	Fusion pair type
p, p2	Fusion pairs
F, S	Arbitrary types
s	Value of type S
o	Output stream
i	Input stream

Expression Semantics

Expression	Semantics
<code>P::first_type</code>	The type of the first template parameter, <code>F</code> , equivalent to <code>result_of::first<P>::type</code> .
<code>P::second_type</code>	The type of the second template parameter, <code>S</code> , equivalent to <code>result_of::second<P>::type</code> .
<code>P()</code>	Default construction.
<code>P(s)</code>	Construct a pair given value for the second type, <code>s</code> .
<code>P(p2)</code>	Copy constructs a pair from another pair, <code>p2</code> .
<code>p.second</code>	Get the data from <code>p1</code> .
<code>p = p2</code>	Assigns a pair, <code>p1</code> , from another pair, <code>p2</code> .
<code>make_pair<F>(s)</code>	Make a pair given the first type, <code>F</code> , and a value for the second type, <code>s</code> . The second type assumes the type of <code>s</code> .
<code>o << p</code>	Output <code>p</code> to output stream, <code>o</code> .
<code>i >> p</code>	Input <code>p</code> from input stream, <code>i</code> .
<code>p == p2</code>	Tests two pairs for equality.
<code>p != p2</code>	Tests two pairs for inequality.

Header

```
#include <boost/fusion/support/pair.hpp>
#include <boost/fusion/include/pair.hpp>
```

Example

```
pair<int, char> p('X');
std::cout << p << std::endl;
std::cout << make_pair<int>('X') << std::endl;
assert((p == make_pair<int>('X')));
```

Iterator

Like [MPL](#) and [STL](#), iterators are a fundamental concept in Fusion. As with [MPL](#) and [STL](#) iterators describe positions, and provide access to data within an underlying [Sequence](#).

Header

```
#include <boost/fusion/iterator.hpp>
#include <boost/fusion/include/iterator.hpp>
```

Concepts

Fusion iterators are divided into different traversal categories. [Forward Iterator](#) is the most basic concept. [Bidirectional Iterator](#) is a refinement of [Forward Iterator](#). [Random Access Iterator](#) is a refinement of [Bidirectional Iterator](#). [Associative Iterator](#) is a refinement of [Forward Iterator](#), [Bidirectional Iterator](#) or [Random Access Iterator](#).

Forward Iterator

Description

A Forward Iterator traverses a [Sequence](#) allowing movement in only one direction through it's elements, one element at a time.

Notation

i, j	Forward Iterators
I, J	Forward Iterator types
M	An MPL integral constant
N	An integral constant

Expression requirements

A type models Forward Iterator if, in addition to being CopyConstructable, the following expressions are valid:

Expression	Return type	Runtime Complexity
<code>next(i)</code>	Forward Iterator	Constant
<code>i == j</code>	Convertible to bool	Constant
<code>i != j</code>	Convertible to bool	Constant
<code>advance_c<N>(i)</code>	Forward Iterator	Constant
<code>advance<M>(i)</code>	Forward Iterator	Constant
<code>distance(i, j)</code>	<code>result_of::distance<I, J>::type</code>	Constant
<code>deref(i)</code>	<code>result_of::deref<I>::type</code>	Constant
<code>*i</code>	<code>result_of::deref<I>::type</code>	Constant

Meta Expressions

Expression	Compile Time Complexity
<code>result_of::next<I>::type</code>	Amortized constant time
<code>result_of::equal_to<I, J>::type</code>	Amortized constant time
<code>result_of::advance_c<I, N>::type</code>	Linear
<code>result_of::advance<I, M>::type</code>	Linear
<code>result_of::distance<I, J>::type</code>	Linear
<code>result_of::deref<I>::type</code>	Amortized constant time
<code>result_of::value_of<I>::type</code>	Amortized constant time

Expression Semantics

Expression	Semantics
<code>next(i)</code>	An iterator to the element following <code>i</code>
<code>i == j</code>	Iterator equality comparison
<code>i != j</code>	Iterator inequality comparison
<code>advance_c<N>(i)</code>	An iterator <code>n</code> elements after <code>i</code> in the sequence
<code>advance<M>(i)</code>	Equivalent to <code>advance_c<M::value>(i)</code>
<code>distance(i, j)</code>	The number of elements between <code>i</code> and <code>j</code>
<code>deref(i)</code>	The element at position <code>i</code>
<code>*i</code>	Equivalent to <code>deref(i)</code>

Invariants

The following invariants always hold:

- `!(i == j) == (i != j)`
- `next(i) == advance_c<1>(i)`
- `distance(i, advance_c<N>(i)) == N`
- Using `next` to traverse the sequence will never return to a previously seen position
- `deref(i)` is equivalent to `*i`
- If `i == j` then `*i` is equivalent to `*j`

Models

- `std::pair` iterator
- `boost::array` iterator

- [vector](#) iterator
- [cons](#) iterator
- [list](#) iterator
- [set](#) iterator
- [map](#) iterator
- [single_view](#) iterator
- [filter_view](#) iterator
- [iterator_range](#) iterator
- [joint_view](#) iterator
- [transform_view](#) iterator
- [reverse_view](#) iterator

Bidirectional Iterator

Description

A Bidirectional Iterator traverses a [Sequence](#) allowing movement in either direction one element at a time.

Notation

- i* A Bidirectional Iterator
- I* A Bidirectional Iterator type
- M* An [MPL](#) integral constant
- N* An integral constant

Refinement of

[Forward Iterator](#)

Expression requirements

In addition to the requirements defined in [Forward Iterator](#), the following expressions must be valid:

Expression	Return type	Runtime Complexity
<code>next(i)</code>	Bidirectional Iterator	Constant
<code>prior(i)</code>	Bidirectional Iterator	Constant
<code>advance_c<N>(i)</code>	Bidirectional Iterator	Constant
<code>advance<M>(i)</code>	Bidirectional Iterator	Constant

Meta Expressions

Expression	Compile Time Complexity
<code>result_of::prior<I>::type</code>	Amortized constant time

Expression Semantics

The semantics of an expression are defined only where they differ from, or are not defined in [Forward Iterator](#)

Expression	Semantics
<code>prior(i)</code>	An iterator to the element preceding <code>i</code>

Invariants

In addition to the invariants of [Forward Iterator](#), the following invariants always hold:

- `prior(next(i)) == i` && `prior(next(i)) == next(prior(i))`
- `prior(i) == advance_c<-1>(i)`
- Using `prior` to traverse a sequence will never return a previously seen position

Models

- `std::pair` iterator
- `boost::array` iterator
- `vector` iterator
- `iterator_range` (where adapted sequence is a [Bidirectional Sequence](#))
- `transform_view` (where adapted sequence is a [Bidirectional Sequence](#))
- `reverse_view`

Random Access Iterator

Description

A Random Access Iterator traverses a [Sequence](#) moving in either direction, permitting efficient arbitrary distance movements back and forward through the sequence.

Notation

<code>i, j</code>	Random Access Iterators
<code>I, J</code>	Random Access Iterator types
<code>M</code>	An MPL integral constant
<code>N</code>	An integral constant

Refinement of

[Bidirectional Iterator](#)

Expression requirements

In addition to the requirements defined in [Bidirectional Iterator](#), the following expressions must be valid:

Expression	Return type	Runtime Complexity
<code>next(i)</code>	Random Access Iterator	Constant
<code>prior(i)</code>	Random Access Iterator	Constant
<code>advance_c<N>(i)</code>	Random Access Iterator	Constant
<code>advance<M>(i)</code>	Random Access Iterator	Constant

Meta Expressions

Expression	Compile Time Complexity
<code>result_of::advance_c<I, N>::type</code>	Amortized constant time
<code>result_of::advance<I, M>::type</code>	Amortized constant time
<code>result_of::distance<I, J>::type</code>	Amortized constant time

Models

- `vector` iterator
- `std::pair` iterator
- `boost::array` iterator
- `iterator_range` iterator (where adapted sequence is a [Random Access Sequence](#))
- `transform_view` iterator (where adapted sequence is a [Random Access Sequence](#))
- `reverse_view` iterator (where adapted sequence is a [Random Access Sequence](#))

Associative Iterator

Description

An Associative Iterator provides additional semantics to obtain the properties of the element of an associative forward, bidirectional or random access sequence.

Notation

- `i` Associative Iterator
- `I` Associative Iterator type

Refinement of

[Forward Iterator](#), [Bidirectional Iterator](#) or [Random Access Iterator](#)

Expression requirements

In addition to the requirements defined in [Forward Iterator](#), [Bidirectional Iterator](#) or [Random Access Iterator](#) the following expressions must be valid:

Expression	Return type	Runtime Complexity
<code>deref_data(i)</code>	<code>result_of::deref_data<I>::type</code>	Constant

Meta Expressions

Expression	Compile Time Complexity
<code>result_of::key_of<I>::type</code>	Amortized constant time
<code>result_of::value_of_data<I>::type</code>	Amortized constant time
<code>result_of::deref_data<I>::type</code>	Amortized constant time

Models

- `map` iterator
- `set` iterator
- `filter_view` iterator (where adapted sequence is an [Associative Sequence](#) and a [Forward Sequence](#))
- `iterator_range` iterator (where adapted iterators are [Associative Iterators](#))
- `joint_view` iterator (where adapted sequences are [Associative Sequences](#) and [Forward Sequences](#))
- `reverse_view` iterator (where adapted sequence is an [Associative Sequence](#) and a [Bidirectional Sequence](#))

Functions

Fusion provides functions for manipulating iterators, analogous to the similar functions from the [MPL](#) library.

deref

Description

Deferencs an iterator.

Synopsis

```
template<
    typename I
>
typename result_of::deref<I>::type deref(I const& i);
```

Table 2. Parameters

Parameter	Requirement	Description
<code>i</code>	Model of Forward Iterator	Operation's argument

Expression Semantics

```
deref(i);
```

Return type: `result_of::deref<I>::type`

Semantics: Dereferences the iterator `i`.

Header

```
#include <boost/fusion/iterator/deref.hpp>
#include <boost/fusion/include/deref.hpp>
```

Example

```
typedef vector<int,int&> vec;

int i(0);
vec v(1,i);
assert(deref(begin(v)) == 1);
assert(deref(next(begin(v))) == 0);
assert(&(deref(next(begin(v)))) == &i);
```

next

Description

Moves an iterator 1 position forwards.

Synopsis

```
template<
    typename I
>
typename result_of::next<I>::type next(I const& i);
```

Table 3. Parameters

Parameter	Requirement	Description
<code>i</code>	Model of Forward Iterator	Operation's argument

Expression Semantics

```
next(i);
```

Return type: A model of the same iterator concept as `i`.

Semantics: Returns an iterator to the next element after `i`.

Header

```
#include <boost/fusion/iterator/next.hpp>
#include <boost/fusion/include/next.hpp>
```

Example

```
typedef vector<int,int,int> vec;

vec v(1,2,3);
assert(deref(begin(v)) == 1);
assert(deref(next(begin(v))) == 2);
assert(deref(next(next(begin(v)))) == 3);
```

prior

Description

Moves an iterator 1 position backwards.

Synopsis

```
template<
    typename I
>
typename result_of::prior<I>::type prior(I const& i);
```

Table 4. Parameters

Parameter	Requirement	Description
i	Model of Bidirectional Iterator	Operation's argument

Expression Semantics

```
prior(i);
```

Return type: A model of the same iterator concept as i.

Semantics: Returns an iterator to the element prior to i.

Header

```
#include <boost/fusion/iterator/prior.hpp>
#include <boost/fusion/include/prior.hpp>
```

Example

```
typedef vector<int,int> vec;

vec v(1,2);
assert(deref(next(begin(v))) == 2);
assert(deref(prior(next(begin(v)))) == 1);
```

distance

Description

Returns the distance between 2 iterators.

Synopsis

```
template<
    typename I,
    typename J
>
typename result_of::distance<I, J>::type distance(I const& i, J const& j);
```

Table 5. Parameters

Parameter	Requirement	Description
i, j	Models of Forward Iterator into the same sequence	The start and end points of the distance to be measured

Expression Semantics

```
distance(i, j);
```

Return type: int

Semantics: Returns the distance between iterators i and j.

Header

```
#include <boost/fusion/iterator/distance.hpp>
#include <boost/fusion/include/distance.hpp>
```

Example

```
typedef vector<int,int,int> vec;

vec v(1,2,3);
assert(distance(begin(v), next(next(begin(v)))) == 2);
```

advance

Description

Moves an iterator by a specified distance.

Synopsis

```
template<
    typename I,
    typename M
>
typename result_of::advance<I, M>::type advance(I const& i);
```

Table 6. Parameters

Parameter	Requirement	Description
<i>i</i>	Model of Forward Iterator	Iterator to move relative to
<i>N</i>	An MPL Integral Constant	Number of positions to move

Expression Semantics

```
advance<M>(i);
```

Return type: A model of the same iterator concept as *i*.

Semantics: Returns an iterator to the element *M* positions from *i*. If *i* is a [Bidirectional Iterator](#) then *M* may be negative.

Header

```
#include <boost/fusion/iterator/advance.hpp>
#include <boost/fusion/include/advance.hpp>
```

Example

```
typedef vector<int,int,int> vec;

vec v(1,2,3);
assert(deref(advance<mpl::int_<2>>(begin(v))) == 3);
```

advance_c

Description

Moves an iterator by a specified distance.

Synopsis

```
template<
    typename I,
    int N
>
typename result_of::advance_c<I, N>::type advance_c(I const& i);
```

Table 7. Parameters

Parameter	Requirement	Description
i	Model of Forward Iterator	Iterator to move relative to
N	Integer constant	Number of positions to move

Expression Semantics

```
advance_c<N>(i);
```

Return type: A model of the same iterator concept as i.

Semantics: Returns an iterator to the element N positions from i. If i is a [Bidirectional Iterator](#) then N may be negative.

Header

```
#include <boost/fusion/iterator/advance.hpp>
#include <boost/fusion/include/advance.hpp>
```

Example

```
typedef vector<int,int,int> vec;

vec v(1,2,3);
assert(deref(advance_c<2>(begin(v))) == 3);
```

deref_data

Description

Deferencs the data property associated with the element referenced by an associative iterator.

Synopsis

```
template<
    typename I
>
typename result_of::deref_data<I>::type deref(I const& i);
```

Table 8. Parameters

Parameter	Requirement	Description
i	Model of Associative Iterator	Operation's argument

Expression Semantics

```
deref_data(i);
```

Return type: `result_of::deref_data<I>::type`

Semantics: Dereferences the data property associated with the element referenced by an associative iterator i.

Header

```
#include <boost/fusion/iterator/deref_data.hpp>
#include <boost/fusion/include/deref_data.hpp>
```

Example

```
typedef map<pair<float,int&> > map;

int i(0);
map m(1.0f,i);
assert(deref_data(begin(m)) == 0);
assert(&(deref_data(begin(m))) == &i);
```

Operator

Overloaded operators are provided to provide a more natural syntax for dereferencing iterators, and comparing them for equality.

Operator *

Description

Dereferences an iterator.

Synopsis

```
template<
    typename I
>
typename result_of::deref<I>::type operator*(unspecified<I> const& i);
```

Table 9. Parameters

Parameter	Requirement	Description
i	Model of Forward Iterator	Operation's argument

Expression Semantics

```
*i
```

Return type: Equivalent to the return type of `deref(i)`.

Semantics: Equivalent to `deref(i)`.

Header

```
#include <boost/fusion/iterator/deref.hpp>
#include <boost/fusion/include/deref.hpp>
```

Example

```
typedef vector<int,int&> vec;

int i(0);
vec v(1,i);
assert(*begin(v) == 1);
assert(*next(begin(v)) == 0);
assert(&(*next(begin(v))) == &i);
```

Operator ==

Description

Compares 2 iterators for equality.

Synopsis

```
template<
    typename I,
    typename J
>
unspecified operator==(I const& i, J const& j);
```

Table 10. Parameters

Parameter	Requirement	Description
i, j	Any fusion iterators	Operation's arguments

Expression Semantics

```
i == j
```

Return type: Convertible to bool.

Semantics: Equivalent to `result_of::equal_to<I,J>::value` where I and J are the types of i and j respectively.

Header

```
#include <boost/fusion/iterator/equal_to.hpp>
#include <boost/fusion/include/equal_to.hpp>
```

Operator !=

Description

Compares 2 iterators for inequality.

Synopsis

```
template<
    typename I,
    typename J
>
unspecified operator==(I const& i, J const& j);
```

Table 11. Parameters

Parameter	Requirement	Description
i, j	Any fusion iterators	Operation's arguments

Expression Semantics

Return type: Convertible to `bool`.

Semantics: Equivalent to `!result_of::equal_to<I,J>::value` where `I` and `J` are the types of `i` and `j` respectively.

Header

```
#include <boost/fusion/iterator/equal_to.hpp>
#include <boost/fusion/include/equal_to.hpp>
```

Metafunctions

value_of

Description

Returns the type stored at the position of an iterator.

Synopsis

```
template<
    typename I
>
struct value_of
{
    typedef unspecified type;
};
```

Table 12. Parameters

Parameter	Requirement	Description
I	Model of Forward Iterator	Operation's argument

Expression Semantics

```
result_of::value_of<I>::type
```

Return type: Any type

Semantics: Returns the type stored in a sequence at iterator position `I`.

Header

```
#include <boost/fusion/iterator/value_of.hpp>
#include <boost/fusion/include/value_of.hpp>
```

Example

```
typedef vector<int,int&,const int&> vec;
typedef result_of::begin<vec>::type first;
typedef result_of::next<first>::type second;
typedef result_of::next<second>::type third;

BOOST_MPL_ASSERT((boost::is_same<result_of::value_of<first>::type, int>));
BOOST_MPL_ASSERT((boost::is_same<result_of::value_of<second>::type, int&>));
BOOST_MPL_ASSERT((boost::is_same<result_of::value_of<third>::type, const int&>));
```

deref

Description

Returns the type that will be returned by dereferencing an iterator.

Synopsis

```
template<
    typename I
>
struct deref
{
    typedef unspecified type;
};
```

Table 13. Parameters

Parameter	Requirement	Description
<code>I</code>	Model of Forward Iterator	Operation's argument

Expression Semantics

```
result_of::deref<I>::type
```

Return type: Any type

Semantics: Returns the result of dereferencing an iterator of type `I`.

Header

```
#include <boost/fusion/iterator/deref.hpp>
#include <boost/fusion/include/deref.hpp>
```

Example

```
typedef vector<int,int&> vec;
typedef const vec const_vec;
typedef result_of::begin<vec>::type first;
typedef result_of::next<first>::type second;

typedef result_of::begin<const_vec>::type const_first;
typedef result_of::next<const_first>::type const_second;

BOOST_MPL_ASSERT((boost::is_same<result_of::deref<first>::type, int&>));
BOOST_MPL_ASSERT((boost::is_same<result_of::deref<second>::type, int&>));
```

next

Description

Returns the type of the next iterator in a sequence.

Synopsis

```
template<
    typename I
>
struct next
{
    typedef unspecified type;
};
```

Table 14. Parameters

Parameter	Requirement	Description
I	Model of Forward Iterator	Operation's argument

Expression Semantics

```
result_of::next<I>::type
```

Return type: A model of the same iterator concept as I.

Semantics: Returns an iterator to the next element in the sequence after I.

Header

```
#include <boost/fusion/iterator/next.hpp>
#include <boost/fusion/include/next.hpp>
```

Example

```
typedef vector<int,double> vec;
typedef result_of::next<result_of::begin<vec>::type>::type second;

BOOST_MPL_ASSERT((boost::is_same<result_of::value_of<second>::type, double>));
```

prior

Description

Returns the type of the previous iterator in a sequence.

Synopsis

```
template<
    typename I
>
struct prior
{
    typedef unspecified type;
};
```

Table 15. Parameters

Parameter	Requirement	Description
I	Model of Bidirectional Iterator	Operation's argument

Expression Semantics

```
result_of::prior<I>::type
```

Return type: A model of the same iterator concept as I.

Semantics: Returns an iterator to the previous element in the sequence before I.

Header

```
#include <boost/fusion/iterator/prior.hpp>
#include <boost/fusion/include/prior.hpp>
```

Example

```
typedef vector<int,double> vec;
typedef result_of::next<result_of::begin<vec>::type>::type second;

BOOST_MPL_ASSERT((boost::is_same<result_of::value_of<second>::type, double>));

typedef result_of::prior<second>::type first;
BOOST_MPL_ASSERT((boost::is_same<result_of::value_of<first>::type, int>));
```

equal_to

Description

Returns a true-valued [MPL Integral Constant](#) if *I* and *J* are equal.

Synopsis

```
template<
    typename I,
    typename J
>
struct equal_to
{
    typedef unspecified type;
};
```

Table 16. Parameters

Parameter	Requirement	Description
<i>I</i> , <i>J</i>	Any fusion iterators	Operation's arguments

Expression Semantics

```
result_of::equal_to<I, J>::type
```

Return type: A model of [MPL Integral Constant](#).

Semantics: Returns `boost::mpl::true_` if *I* and *J* are iterators to the same position. Returns `boost::mpl::false_` otherwise.

Header

```
#include <boost/fusion/iterator/equal_to.hpp>
#include <boost/fusion/include/equal_to.hpp>
```

Example

```
typedef vector<int,double> vec;
typedef result_of::begin<vec>::type first;
typedef result_of::end<vec>::type last;
BOOST_MPL_ASSERT((result_of::equal_to<first, first>));
BOOST_MPL_ASSERT_NOT((result_of::equal_to<first,last>));
```

distance

Description

Returns the distance between two iterators.

Synopsis

```
template<
    typename I,
    typename J
>
struct distance
{
    typedef unspecified type;
};
```

Table 17. Parameters

Parameter	Requirement	Description
I, J	Models of Forward Iterator into the same sequence	The start and end points of the distance to be measured

Expression Semantics

```
result_of::distance<I, J>::type
```

Return type: A model of [MPL Integral Constant](#).

Semantics: Returns the distance between iterators of types I and J.

Header

```
#include <boost/fusion/iterator/distance.hpp>
#include <boost/fusion/include/distance.hpp>
```

Example

```
typedef vector<int,double,char> vec;
typedef result_of::begin<vec>::type first;
typedef result_of::next<first>::type second;
typedef result_of::next<second>::type third;
typedef result_of::distance<first,third>::type dist;

BOOST_MPL_ASSERT_RELATION(dist::value, ==, 2);
```

advance

Description

Moves an iterator a specified distance.

Synopsis

```
template<
    typename I,
    typename M
>
struct advance
{
    typedef unspecified type;
};
```

Table 18. Parameters

Parameter	Requirement	Description
I	Model of Forward Iterator	Iterator to move relative to
M	Model of MPL Integral Constant	Number of positions to move

Expression Semantics

```
result_of::advance<I,M>::type
```

Return type: A model of the same iterator concept as I.

Semantics: Returns an iterator a distance M from I. If I is a [Bidirectional Iterator](#) then M may be negative.

Header

```
#include <boost/fusion/iterator/advance.hpp>
#include <boost/fusion/include/advance.hpp>
```

Example

```
typedef vector<int,double,char> vec;
typedef result_of::begin<vec>::type first;
typedef result_of::next<first>::type second;
typedef result_of::next<second>::type third;

BOOST_MPL_ASSERT((result_of::equal_to<result_of::advance<first, boost::mpl::int_<2> >::type, third>));
```

advance_c

Description

Moves an iterator by a specified distance.

Synopsis

```
template<
    typename I,
    int N
>
struct advance_c
{
    typedef unspecified type;
};
```

Table 19. Parameters

Parameter	Requirement	Description
I	Model of Forward Iterator	Iterator to move relative to
N	Integer constant	Number of positions to move

Expression Semantics

```
result_of::advance_c<I, N>::type
```

Return type: A model of the same iterator concept as I.

Semantics: Returns an iterator a distance N from I. If I is a [Bidirectional Iterator](#) then N may be negative. Equivalent to `result_of::advance<I, boost::mpl::int_<N> >::type`.

Header

```
#include <boost/fusion/iterator/advance.hpp>
#include <boost/fusion/include/advance.hpp>
```

Example

```
typedef vector<int,double,char> vec;
typedef result_of::begin<vec>::type first;
typedef result_of::next<first>::type second;
typedef result_of::next<second>::type third;

BOOST_MPL_ASSERT((result_of::equal_to<result_of::advance_c<first, 2>::type, third>));
```

key_of

Description

Returns the key type associated with the element referenced by an associative iterator.

Synopsis

```
template<
    typename I
>
struct key_of
{
    typedef unspecified type;
};
```

Table 20. Parameters

Parameter	Requirement	Description
I	Model of Associative Iterator	Operation's argument

Expression Semantics

```
result_of::key_of<I>::type
```

Return type: Any type

Semantics: Returns the key type associated with the element referenced by an associative iterator I.

Header

```
#include <boost/fusion/iterator/key_of.hpp>
#include <boost/fusion/include/key_of.hpp>
```

Example

```
typedef map<pair<float,int> > vec;
typedef result_of::begin<vec>::type first;

BOOST_MPL_ASSERT((boost::is_same<result_of::key_of<first>::type, float>));
```

value_of_data

Description

Returns the type of the data property associated with the element referenced by an associative iterator references.

Synopsis

```
template<
    typename I
>
struct value_of_data
{
    typedef unspecified type;
};
```

Table 21. Parameters

Parameter	Requirement	Description
I	Model of Associative Iterator	Operation's argument

Expression Semantics

```
result_of::value_of_data<I>::type
```

Return type: Any type

Semantics: Returns the type of the data property associated with the element referenced by an associative iterator I.

Header

```
#include <boost/fusion/iterator/value_of_data.hpp>
#include <boost/fusion/include/value_of_data.hpp>
```

Example

```
typedef map<pair<float,int> > vec;
typedef result_of::begin<vec>::type first;

BOOST_MPL_ASSERT((boost::is_same<result_of::value_of_data<first>::type, int>));
```

deref_data

Description

Returns the type that will be returned by dereferencing the data property referenced by an associative iterator.

Synopsis

```
template<
    typename I
>
struct deref_data
{
    typedef unspecified type;
};
```

Table 22. Parameters

Parameter	Requirement	Description
I	Model of Associative Iterator	Operation's argument

Expression Semantics

```
result_of::deref_data<I>::type
```

Return type: Any type

Semantics: Returns the result of dereferencing the data property referenced by an associative iterator of type I.

Header

```
#include <boost/fusion/iterator/deref_data.hpp>
#include <boost/fusion/include/deref_data.hpp>
```

Example

```
typedef map<pair<float,int> > map;
typedef result_of::begin<vec>::type first;

BOOST_MPL_ASSERT((boost::is_same<result_of::deref_data<first>::type, int&>));
```

Sequence

Like [MPL](#), the Sequence is a fundamental concept in Fusion. A Sequence may or may not actually store or contain data. [Container](#) are sequences that hold data. [Views](#), on the other hand, are sequences that do not store any data. Instead, they are proxies that impart an alternative presentation over another sequence. All models of Sequence have an associated [Iterator](#) type that can be used to iterate through the Sequence's elements.

Header

```
#include <boost/fusion/sequence.hpp>
#include <boost/fusion/include/sequence.hpp>
```

Concepts

Fusion Sequences are organized into a hierarchy of concepts.

Traversal

Fusion's sequence traversal related concepts parallel Fusion's [Iterator Concepts](#). [Forward Sequence](#) is the most basic concept. [Bidirectional Sequence](#) is a refinement of [Forward Sequence](#). [Random Access Sequence](#) is a refinement of [Bidirectional Sequence](#). These concepts pertain to sequence traversal.

Associativity

The [Associative Sequence](#) concept is orthogonal to traversal. An Associative Sequence allows efficient retrieval of elements based on keys.

Forward Sequence

Description

A Forward Sequence is a Sequence whose elements are arranged in a definite order. The ordering is guaranteed not to change from iteration to iteration. The requirement of a definite ordering allows the definition of element-by-element equality (if the container's element type is Equality Comparable) and of lexicographical ordering (if the container's element type is LessThan Comparable).

Notation

- s A Forward Sequence
- S A Forward Sequence type
- o An arbitrary object
- e A Sequence element

Valid Expressions

For any Forward Sequence the following expressions must be valid:

Expression	Return type	Type Requirements	Runtime Complexity
<code>begin(s)</code>	Forward Iterator		Constant
<code>end(s)</code>	Forward Iterator		Constant
<code>size(s)</code>	MPL Integral Constant. Convertible to int.		Constant
<code>empty(s)</code>	MPL Boolean Constant. Convertible to bool.		Constant
<code>front(s)</code>	Any type		Constant
<code>front(s) = o</code>	Any type	<code>s</code> is mutable and <code>e = o</code> , where <code>e</code> is the first element in the sequence, is a valid expression.	Constant

Result Type Expressions

Expression	Compile Time Complexity
<code>result_of::begin<S>::type</code>	Amortized constant time
<code>result_of::end<S>::type</code>	Amortized constant time
<code>result_of::size<S>::type</code>	Unspecified
<code>result_of::empty<S>::type</code>	Constant time
<code>result_of::front<S>::type</code>	Amortized constant time

Expression Semantics

Expression	Semantics
<code>begin(s)</code>	An iterator to the first element of the sequence; see <code>begin</code> .
<code>end(s)</code>	A past-the-end iterator to the sequence; see <code>end</code> .
<code>size(s)</code>	The size of the sequence; see <code>size</code> .
<code>empty(s)</code>	A boolean Integral Constant <code>c</code> such that <code>c::value == true</code> if and only if the sequence is empty; see <code>empty</code> .
<code>front(s)</code>	The first element in the sequence; see <code>front</code> .

Invariants

For any Forward Sequence `s` the following invariants always hold:

- `[begin(s), end(s))` is always a valid range.
- An Algorithm that iterates through the range `[begin(s), end(s))` will pass through every element of `s` exactly once.

- `begin(s)` is identical to `end(s)` if and only if `s` is empty.
- Two different iterations through `s` will access its elements in the same order.

Models

- `std::pair`
- `boost::array`
- `vector`
- `cons`
- `list`
- `set`
- `map`
- `single_view`
- `filter_view`
- `iterator_range`
- `joint_view`
- `transform_view`
- `reverse_view`
- `zip_view`

Bidirectional Sequence

Description

A Bidirectional Sequence is a [Forward Sequence](#) whose iterators model [Bidirectional Iterator](#).

Refinement of

[Forward Sequence](#)

Notation

- `s` A Forward Sequence
- `S` A Forward Sequence type
- `o` An arbitrary object
- `e` A Sequence element

Valid Expressions

In addition to the requirements defined in [Forward Sequence](#), for any Bidirectional Sequence the following must be met:

Expression	Return type	Type Requirements	Runtime Complexity
<code>begin(s)</code>	Bidirectional Iterator		Constant
<code>end(s)</code>	Bidirectional Iterator		Constant
<code>back(s)</code>	Any type		Constant
<code>back(s) = o</code>	Any type	<code>s</code> is mutable and <code>e = o</code> , where <code>e</code> is the first element in the sequence, is a valid expression.	Constant

Result Type Expressions

Expression	Compile Time Complexity
<code>result_of::begin<S>::type</code>	Amortized constant time
<code>result_of::end<S>::type</code>	Amortized constant time
<code>result_of::back<S>::type</code>	Amortized constant time

Expression Semantics

The semantics of an expression are defined only where they differ from, or are not defined in [Forward Sequence](#).

Expression	Semantics
<code>back(s)</code>	The last element in the sequence; see back .

Models

- `std::pair`
- `boost::array`
- `vector`
- `reverse_view`
- `iterator_range` (where adapted sequence is a Bidirectional Sequence)
- `transform_view` (where adapted sequence is a Bidirectional Sequence)
- `zip_view` (where adapted sequences are models of Bidirectional Sequence)

Random Access Sequence

Description

A Random Access Sequence is a [Bidirectional Sequence](#) whose iterators model [Random Access Iterator](#). It guarantees constant time access to arbitrary sequence elements.

Refinement of

[Bidirectional Sequence](#)

Notation

- s A Random Access Sequence
- S A Random Access Sequence type
- N An [MPL Integral Constant](#)
- o An arbitrary object
- e A Sequence element

Valid Expressions

In addition to the requirements defined in [Bidirectional Sequence](#), for any Random Access Sequence the following must be met:

Expression	Return type	Type Requirements	Runtime Complexity
<code>begin(s)</code>	Random Access Iterator		Constant
<code>end(s)</code>	Random Access Iterator		Constant
<code>at<N>(s)</code>	Any type		Constant
<code>at<N>(s) = o</code>	Any type	s is mutable and $e = o$, where e is the first element in the sequence, is a valid expression.	Constant

Result Type Expressions

Expression	Compile Time Complexity
<code>result_of::begin<S>::type</code>	Amortized constant time
<code>result_of::end<S>::type</code>	Amortized constant time
<code>result_of::at<S, N>::type</code>	Amortized constant time
<code>result_of::value_at<S, N>::type</code>	Amortized constant time



`result_of::at<S, N>` returns the actual type returned by `at<N>(s)`. In most cases, this is a reference. Hence, there is no way to know the exact element type using `result_of::at<S, N>`. The element at N may actually be a reference to begin with. For this purpose, you can use `result_of::value_at<S, N>`.

Expression Semantics

The semantics of an expression are defined only where they differ from, or are not defined in [Bidirectional Sequence](#).

Expression	Semantics
<code>at<N>(s)</code>	The N th element from the beginning of the sequence; see at .

Models

- `std::pair`

- `boost::array`
- `vector`
- `reverse_view`
- `iterator_range` (where adapted sequence is a Random Access Sequence)
- `transform_view` (where adapted sequence is a Random Access Sequence)
- `zip_view` (where adapted sequences are models of Random Access Sequence)

Associative Sequence

Description

An Associative Sequence allows efficient retrieval of elements based on keys. Like associative sequences in [MPL](#), and unlike associative containers in [STL](#), Fusion associative sequences have no implied ordering relation. Instead, type identity is used to impose an equivalence relation on keys.

Notation

- s An Associative Sequence
- S An Associative Sequence type
- K An arbitrary *key* type
- o An arbitrary object
- e A Sequence element

Valid Expressions

For any Associative Sequence the following expressions must be valid:

Expression	Return type	Type Requirements	Runtime Complexity
<code>has_key<K>(s)</code>	MPL Boolean Constant . Convertible to bool.		Constant
<code>at_key<K>(s)</code>	Any type		Constant
<code>at_key<K>(s) = o</code>	Any type	s is mutable and $e = o$, where e is the first element in the sequence, is a valid expression.	Constant

Result Type Expressions

Expression	Compile Time Complexity
<code>result_of::has_key<S, K>::type</code>	Amortized constant time
<code>result_of::at_key<S, K>::type</code>	Amortized constant time
<code>result_of::value_at_key<S, K>::type</code>	Amortized constant time



`result_of::at_key<S, K>` returns the actual type returned by `at_key<K>(s)`. In most cases, this is a reference. Hence, there is no way to know the exact element type using `result_of::at_key<S, K>`. The element at `K` may actually be a reference to begin with. For this purpose, you can use `result_of::value_at_key<S, N>`.

Expression Semantics

Expression	Semantics
<code>has_key<K>(s)</code>	A boolean Integral Constant <code>c</code> such that <code>c::value == true</code> if and only if there is one or more elements with the key <code>k</code> in <code>s</code> ; see has_key .
<code>at_key<K>(s)</code>	The element associated with the key <code>K</code> in the sequence <code>s</code> ; see at .

Models

- [set](#)
- [map](#)
- [filter_view](#) (where adapted sequence is an [Associative Sequence](#) and a [Forward Sequence](#))
- [iterator_range](#) (where adapted iterators are [Associative Iterators](#))
- [joint_view](#) (where adapted sequences are [Associative Sequences](#) and [Forward Sequences](#))
- [reverse_view](#) (where adapted sequence is an [Associative Sequence](#) and a [Bidirectional Sequence](#))

Intrinsic

Intrinsic form the essential interface of every Fusion [Sequence](#). STL counterparts of these functions are usually implemented as member functions. Intrinsic functions, unlike [Algorithms](#), are not generic across the full [Sequence](#) repertoire. They need to be implemented for each Fusion [Sequence](#) ⁵.

Header

```
#include <boost/fusion/sequence/intrinsic.hpp>
#include <boost/fusion/include/intrinsic.hpp>
```

Functions

begin

Description

Returns an iterator pointing to the first element in the sequence.

⁵ In practice, many of intrinsic functions have default implementations that will work in majority of cases

Synopsis

```
template <typename Sequence>
typename result_of::begin<Sequence>::type
begin(Sequence& seq);

template <typename Sequence>
typename result_of::begin<Sequence const>::type
begin(Sequence const& seq);
```

Parameters

Parameter	Requirement	Description
seq	Model of Forward Sequence	The sequence we wish to get an iterator from.

Expression Semantics

```
begin(seq);
```

Return type:

- A model of [Forward Iterator](#) if seq is a [Forward Sequence](#) else, [Bidirectional Iterator](#) if seq is a [Bidirectional Sequence](#) else, [Random Access Iterator](#) if seq is a [Random Access Sequence](#).
- A model of [Associative Iterator](#) if seq is an [Associative Sequence](#).

Semantics: Returns an iterator pointing to the first element in the sequence.

Header

```
#include <boost/fusion/sequence/intrinsic/begin.hpp>
#include <boost/fusion/include/begin.hpp>
```

Example

```
vector<int, int, int> v(1, 2, 3);
assert(deref(begin(v)) == 1);
```

end

Description

Returns an iterator pointing to one element past the end of the sequence.

Synopsis

```
template <typename Sequence>
typename result_of::end<Sequence>::type
end(Sequence& seq);

template <typename Sequence>
typename result_of::end<Sequence const>::type
end(Sequence const& seq);
```

Parameters

Parameter	Requirement	Description
seq	Model of Forward Sequence	The sequence we wish to get an iterator from.

Expression Semantics

```
end(seq);
```

Return type:

- A model of [Forward Iterator](#) if seq is a [Forward Sequence](#) else, [Bidirectional Iterator](#) if seq is a [Bidirectional Sequence](#) else, [Random Access Iterator](#) if seq is a [Random Access Sequence](#).
- A model of [Associative Iterator](#) if seq is an [Associative Sequence](#).

Semantics: Returns an iterator pointing to one element past the end of the sequence.

Header

```
#include <boost/fusion/sequence/intrinsic/end.hpp>
#include <boost/fusion/include/end.hpp>
```

Example

```
vector<int, int, int> v(1, 2, 3);
assert(deref(prior(end(v))) == 3);
```

empty

Description

Returns a type convertible to `bool` that evaluates to `true` if the sequence is empty, else, evaluates to `false`.

Synopsis

```
template <typename Sequence>
typename result_of::empty<Sequence>::type
empty(Sequence const& seq);
```

Parameters

Parameter	Requirement	Description
seq	Model of Forward Sequence	The sequence we wish to investigate.

Expression Semantics

```
empty(seq);
```

Return type: Convertible to `bool`.

Semantics: Evaluates to `true` if the sequence is empty, else, evaluates to `false`.

Header

```
#include <boost/fusion/sequence/intrinsic/empty.hpp>
#include <boost/fusion/include/empty.hpp>
```

Example

```
vector<int, int, int> v(1, 2, 3);
assert(empty(v) == false);
```

front

Description

Returns the first element in the sequence.

Synopsis

```
template <typename Sequence>
typename result_of::front<Sequence>::type
front(Sequence& seq);

template <typename Sequence>
typename result_of::front<Sequence const>::type
front(Sequence const& seq);
```

Parameters

Parameter	Requirement	Description
<code>seq</code>	Model of Forward Sequence	The sequence we wish to investigate.

Expression Semantics

```
front(seq);
```

Return type: Returns a reference to the first element in the sequence `seq` if `seq` is mutable and `e = o`, where `e` is the first element in the sequence, is a valid expression. Else, returns a type convertible to the first element in the sequence.

Precondition: `empty(seq) == false`

Semantics: Returns the first element in the sequence.

Header

```
#include <boost/fusion/sequence/intrinsic/front.hpp>
#include <boost/fusion/include/front.hpp>
```

Example

```
vector<int, int, int> v(1, 2, 3);
assert(front(v) == 1);
```

back

Description

Returns the last element in the sequence.

Synopsis

```
template <typename Sequence>
typename result_of::back<Sequence>::type
back(Sequence& seq);

template <typename Sequence>
typename result_of::back<Sequence const>::type
back(Sequence const& seq);
```

Parameters

Parameter	Requirement	Description
seq	Model of Bidirectional Sequence	The sequence we wish to investigate.

Expression Semantics

```
back(seq);
```

Return type: Returns a reference to the last element in the sequence `seq` if `seq` is mutable and `e = o`, where `e` is the last element in the sequence, is a valid expression. Else, returns a type convertible to the last element in the sequence.

Precondition: `empty(seq) == false`

Semantics: Returns the last element in the sequence.

Header

```
#include <boost/fusion/sequence/intrinsic/back.hpp>
#include <boost/fusion/include/back.hpp>
```

Example

```
vector<int, int, int> v(1, 2, 3);
assert(back(v) == 3);
```

size

Description

Returns a type convertible to `int` that evaluates the number of elements in the sequence.

Synopsis

```
template <typename Sequence>
typename result_of::size<Sequence>::type
size(Sequence const& seq);
```

Parameters

Parameter	Requirement	Description
<code>seq</code>	Model of Forward Sequence	The sequence we wish to investigate.

Expression Semantics

```
size(seq);
```

Return type: Convertible to `int`.

Semantics: Returns the number of elements in the sequence.

Header

```
#include <boost/fusion/sequence/intrinsic/size.hpp>
#include <boost/fusion/include/size.hpp>
```

Example

```
vector<int, int, int> v(1, 2, 3);
assert(size(v) == 3);
```

at

Description

Returns the N-th element from the beginning of the sequence.

Synopsis

```
template <typename N, typename Sequence>
typename result_of::at<Sequence, N>::type
at(Sequence& seq);

template <typename N, typename Sequence>
typename result_of::at<Sequence const, N>::type
at(Sequence const& seq);
```

Parameters

Parameter	Requirement	Description
seq	Model of Random Access Sequence	The sequence we wish to investigate.
N	An MPL Integral Constant	An index from the beginning of the sequence.

Expression Semantics

```
at<N>(seq);
```

Return type: Returns a reference to the N-th element from the beginning of the sequence `seq` if `seq` is mutable and `e = o`, where `e` is the N-th element from the beginning of the sequence, is a valid expression. Else, returns a type convertible to the N-th element from the beginning of the sequence.

Precondition: `0 <= N::value < size(s)`

Semantics: Equivalent to

```
deref(advance<N>(begin(s)))
```

Header

```
#include <boost/fusion/sequence/intrinsic/at.hpp>
#include <boost/fusion/include/at.hpp>
```

Example

```
vector<int, int, int> v(1, 2, 3);
assert(at<mpl::int_<1>>(v) == 2);
```

at_c

Description

Returns the N-th element from the beginning of the sequence.

Synopsis

```
template <int N, typename Sequence>
typename result_of::at_c<Sequence, N>::type
at_c(Sequence& seq);

template <int N, typename Sequence>
typename result_of::at_c<Sequence const, N>::type
at_c(Sequence const& seq);
```

Parameters

Parameter	Requirement	Description
seq	Model of Random Access Sequence	The sequence we wish to investigate.
N	An integral constant	An index from the beginning of the sequence.

Expression Semantics

```
at_c<N>(seq);
```

Return type: Returns a reference to the N-th element from the beginning of the sequence `seq` if `seq` is mutable and $e = o$, where e is the N-th element from the beginning of the sequence, is a valid expression. Else, returns a type convertible to the N-th element from the beginning of the sequence.

Precondition: $0 \leq N < \text{size}(s)$

Semantics: Equivalent to

```
deref(advance<N>(begin(s)))
```

Header

```
#include <boost/fusion/sequence/intrinsic/at_c.hpp>
#include <boost/fusion/include/at_c.hpp>
```

Example

```
vector<int, int, int> v(1, 2, 3);
assert(at_c<1>(v) == 2);
```

has_key

Description

Returns a type convertible to `bool` that evaluates to `true` if the sequence contains an element associated with a `Key`, else, evaluates to `false`.

Synopsis

```
template <typename Key, typename Sequence>
typename result_of::has_key<Sequence, Key>::type
has_key(Sequence const& seq);
```

Parameters

Parameter	Requirement	Description
seq	Model of Associative Sequence	The sequence we wish to investigate.
Key	Any type	The queried key.

Expression Semantics

```
has_key<Key>(seq);
```

Return type: Convertible to `bool`.

Semantics: Evaluates to `true` if the sequence contains an element associated with `Key`, else, evaluates to `false`.

Header

```
#include <boost/fusion/sequence/intrinsic/has_key.hpp>
#include <boost/fusion/include/has_key.hpp>
```

Example

```
set<int, char, bool> s(1, 'x', true);
assert(has_key<char>(s) == true);
```

at_key

Description

Returns the element associated with a `Key` from the sequence.

Synopsis

```
template <typename Key, typename Sequence>
typename result_of::at_key<Sequence, Key>::type
at_key(Sequence& seq);

template <typename Key, typename Sequence>
typename result_of::at_key<Sequence const, Key>::type
at_key(Sequence const& seq);
```

Parameters

Parameter	Requirement	Description
seq	Model of Associative Sequence	The sequence we wish to investigate.
Key	Any type	The queried key.

Expression Semantics

```
at_key<Key>(seq);
```

Return type: Returns a reference to the element associated with `Key` from the sequence `seq` if `seq` is mutable and `e = o`, where `e` is the element associated with `Key`, is a valid expression. Else, returns a type convertible to the element associated with `Key`.

Precondition: `has_key<Key>(seq) == true`

Semantics: Returns the element associated with `Key`.

Header

```
#include <boost/fusion/sequence/intrinsic/at_key.hpp>
#include <boost/fusion/include/at_key.hpp>
```

Example

```
set<int, char, bool> s(1, 'x', true);
assert(at_key<char>(s) == 'x');
```

swap

Description

Performs an element by element swap of the elements in 2 sequences.

Synopsis

```
template<typename Seq1, typename Seq2>
void swap(Seq1& seq1, Seq2& seq2);
```

Parameters

Parameters	Requirement	Description
<code>seq1, seq2</code>	Models of Forward Sequence	The sequences whos elements we wish to swap.

Expression Semantics

```
swap(seq1, seq2);
```

Return type: `void`

Precondition: `size(seq1) == size(seq2)`

Semantics: Calls `swap(a1, b1)` for corresponding elements in `seq1` and `seq2`.

`/sequence/intrinsic/swap.hpp>`

Example

```
vector<int, std::string> v1(1, "hello"), v2(2, "world");
swap(v1, v2);
assert(v1 == make_vector(2, "world"));
assert(v2 == make_vector(1, "hello"));
```

Metafunctions

begin

Description

Returns the result type of `begin`.

Synopsis

```
template<typename Seq>
struct begin
{
    typedef unspecified type;
};
```

Table 23. Parameters

Parameter	Requirement	Description
Seq	A model of Forward Sequence	Argument sequence

Expression Semantics

```
result_of::begin<Seq>::type
```

Return type:

- A model of [Forward Iterator](#) if `seq` is a [Forward Sequence](#) else, [Bidirectional Iterator](#) if `seq` is a [Bidirectional Sequence](#) else, [Random Access Iterator](#) if `seq` is a [Random Access Sequence](#).
- A model of [Associative Iterator](#) if `seq` is an [Associative Sequence](#).

Semantics: Returns the type of an iterator to the first element of `Seq`.

Header

```
#include <boost/fusion/sequence/intrinsic/begin.hpp>
#include <boost/fusion/include/begin.hpp>
```

Example

```
typedef vector<int> vec;
typedef result_of::begin<vec>::type it;
BOOST_MPL_ASSERT((boost::is_same<result_of::deref<it>::type, int&>))
```

end

Description

Returns the result type of [end](#).

Synopsis

```
template<typename Seq>
struct end
{
    typedef unspecified type;
};
```

Table 24. Parameters

Parameter	Requirement	Description
Seq	A model of Forward Sequence	Argument sequence

Expression Semantics

```
result_of::end<Seq>::type
```

Return type:

- A model of [Forward Iterator](#) if seq is a [Forward Sequence](#) else, [Bidirectional Iterator](#) if seq is a [Bidirectional Sequence](#) else, [Random Access Iterator](#) if seq is a [Random Access Sequence](#).
- A model of [Associative Iterator](#) if seq is an [Associative Sequence](#).

Semantics: Returns the type of an iterator one past the end of Seq.

Header

```
#include <boost/fusion/sequence/intrinsic/end.hpp>
#include <boost/fusion/include/end.hpp>
```

Example

```
typedef vector<int> vec;
typedef result_of::prior<result_of::end<vec>::type>::type first;
BOOST_MPL_ASSERT((result_of::equal_to<first, result_of::begin<vec>::type>))
```

empty

Description

Returns the result type of `empty`.

Synopsis

```
template<typename Seq>
struct empty
{
    typedef unspecified type;
};
```

Table 25. Parameters

Parameter	Requirement	Description
Seq	A model of Forward Sequence	Argument sequence

Expression Semantics

```
result_of::empty<Seq>::type
```

Return type: An [MPL Integral Constant](#)

Semantics: Returns `mpl::true_` if Seq has zero elements, `mpl::false_` otherwise.

Header

```
#include <boost/fusion/sequence/intrinsic/empty.hpp>
#include <boost/fusion/include/empty.hpp>
```

Example

```
typedef vector<> empty_vec;
typedef vector<int,float,char> vec;

BOOST_MPL_ASSERT((result_of::empty<empty_vec>));
BOOST_MPL_ASSERT_NOT((result_of::empty<vec>));
```

front

Description

Returns the result type of `front`.

Synopsis

```
template<typename Seq>
struct front
{
    typedef unspecified type;
};
```

Table 26. Parameters

Parameter	Requirement	Description
Seq	A model of Forward Sequence	Argument sequence

Expression Semantics

```
result_of::front<Seq>::type
```

Return type: Any type

Semantics: The type returned by dereferencing an iterator to the first element in Seq. Equivalent to `result_of::deref<result_of::begin<Seq>::type>::type`.

Header

```
#include <boost/fusion/sequence/intrinsic/front.hpp>
#include <boost/fusion/include/front.hpp>
```

Example

```
typedef vector<int, char> vec;
BOOST_MPL_ASSERT((boost::is_same<result_of::front<vec>::type, int&>));
```

back

Description

Returns the result type of `back`.

Synopsis

```
template<typename Seq>
struct back
{
    typedef unspecified type;
};
```

Table 27. Parameters

Parameter	Requirement	Description
Seq	A model of Forward Sequence	Argument sequence

Expression Semantics

```
result_of::back<Seq>::type
```

Return type: Any type

Semantics: The type returned by dereferencing an iterator to the last element in the sequence. Equivalent to `result_of::deref<result_of::prior<result_of::end<Seq>::type>::type>::type`.

Header

```
#include <boost/fusion/sequence/intrinsic/back.hpp>
#include <boost/fusion/include/back.hpp>
```

Example

```
typedef vector<int, char> vec;
BOOST_MPL_ASSERT((boost::is_same<result_of::back<vec>::type, char&>));
```

size

Description

Returns the result type of `size`.

Synopsis

```
template<typename Seq>
struct size
{
    typedef unspecified type;
};
```

Table 28. Parameters

Parameter	Requirement	Description
Seq	A model of Forward Sequence	Argument sequence

Expression Semantics

```
result_of::size<Seq>::type
```

Return type: An [MPL Integral Constant](#).

Semantics: Returns the number of elements in Seq.

Header

```
#include <boost/fusion/sequence/intrinsic/size.hpp>
#include <boost/fusion/include/size.hpp>
```

Example

```
typedef vector<int,float,char> vec;
typedef result_of::size<vec>::type size_mpl_integral_constant;
BOOST_MPL_ASSERT_RELATION(size_mpl_integral_constant::value, ==, 3);
```

at

Description

Returns the result type of [at](#)⁶.

⁶ [result_of::at](#) reflects the actual return type of the function [at](#). [Sequence\(s\)](#) typically return references to its elements via the [at](#) function. If you want to get the actual element type, use [result_of::value_at](#)

Synopsis

```
template<
    typename Seq,
    typename N>
struct at
{
    typedef unspecified type;
};
```

Table 29. Parameters

Parameter	Requirement	Description
Seq	A model of Forward Sequence	Argument sequence
N	An MPL Integral Constant	Index of element

Expression Semantics

```
result_of::at<Seq, N>::type
```

Return type: Any type.

Semantics: Returns the result type of using `at` to access the `N`th element of `Seq`.

Header

```
#include <boost/fusion/sequence/intrinsic/at.hpp>
#include <boost/fusion/include/at.hpp>
```

Example

```
typedef vector<int,float,char> vec;
BOOST_MPL_ASSERT((boost::is_same<result_of::at<vec, boost::mpl::int_<1> >::type, float&>));
```

at_c

Description

Returns the result type of `at_c`⁷.

⁷ `result_of::at_c` reflects the actual return type of the function `at_c`. Sequence(s) typically return references to its elements via the `at_c` function. If you want to get the actual element type, use `result_of::value_at_c`

Synopsis

```
template<
    typename Seq,
    int M>
struct at_c
{
    typedef unspecified type;
};
```

Table 30. Parameters

Parameter	Requirement	Description
Seq	A model of Forward Sequence	Argument sequence
M	Positive integer index	Index of element

Expression Semantics

```
result_of::at_c<Seq, M>::type
```

Return type: Any type

Semantics: Returns the result type of using `at_c` to access the `M`th element of `Seq`.

Header

```
#include <boost/fusion/sequence/intrinsic/at.hpp>
#include <boost/fusion/include/at.hpp>
```

Example

```
typedef vector<int,float,char> vec;
BOOST_MPL_ASSERT((boost::is_same<result_of::at_c<vec, 1>::type, float&>));
```

value_at

Description

Returns the actual type at a given index from the [Sequence](#).

Synopsis

```
template<
    typename Seq,
    typename N>
struct value_at
{
    typedef unspecified type;
};
```

Table 31. Parameters

Parameter	Requirement	Description
Seq	A model of Forward Sequence	Argument sequence
N	An MPL Integral Constant	Index of element

Expression Semantics

```
result_of::value_at<Seq, N>::type
```

Return type: Any type.

Semantics: Returns the actual type at the Nth element of Seq.

Header

```
#include <boost/fusion/sequence/intrinsic/value_at.hpp>
#include <boost/fusion/include/value_at.hpp>
```

Example

```
typedef vector<int,float,char> vec;
BOOST_MPL_ASSERT((boost::is_same<result_of::value_at<vec, boost::mpl::int_<1> >::type, float>));
```

value_at_c

Description

Returns the actual type at a given index from the [Sequence](#).

Synopsis

```
template<
    typename Seq,
    int M>
struct value_at_c
{
    typedef unspecified type;
};
```

Table 32. Parameters

Parameter	Requirement	Description
Seq	A model of Forward Sequence	Argument sequence
M	Positive integer index	Index of element

Expression Semantics

```
result_of::value_at_c<Seq, M>::type
```

Return type: Any type

Semantics: Returns the actual type at the Mth element of Seq.

Header

```
#include <boost/fusion/sequence/intrinsic/value_at.hpp>
#include <boost/fusion/include/value_at.hpp>
```

Example

```
typedef vector<int,float,char> vec;
BOOST_MPL_ASSERT((boost::is_same<result_of::value_at_c<vec, 1>::type, float>));
```

has_key

Description

Returns the result type of [has_key](#).

Synopsis

```
template<
    typename Seq,
    typename Key>
struct has_key
{
    typedef unspecified type;
};
```

Table 33. Parameters

Parameter	Requirement	Description
Seq	A model of Forward Sequence	Argument sequence
Key	Any type	Key type

Expression Semantics

```
result_of::has_key<Seq, Key>::type
```

Return type: An [MPL Integral Constant](#).

Semantics: Returns `mpl::true_` if Seq contains an element with key type Key, returns `mpl::false_` otherwise.

Header

```
#include <boost/fusion/sequence/intrinsic/has_key.hpp>
#include <boost/fusion/include/has_key.hpp>
```

Example

```
typedef map<pair<int, char>, pair<char, char>, pair<double, char> > mymap;
BOOST_MPL_ASSERT((result_of::has_key<mymap, int>));
BOOST_MPL_ASSERT_NOT((result_of::has_key<mymap, void*>));
```

at_key

Description

Returns the result type of [at_key](#)⁸.

⁸ `result_of::at_key` reflects the actual return type of the function `at_key`. `_sequence_s` typically return references to its elements via the `at_key` function. If you want to get the actual element type, use `result_of::value_at_key`

Synopsis

```
template<
    typename Seq,
    typename Key>
struct at_key
{
    typedef unspecified type;
};
```

Table 34. Parameters

Parameter	Requirement	Description
Seq	A model of Forward Sequence	Argument sequence
Key	Any type	Key type

Expression Semantics

```
result_of::at_key<Seq, Key>::type
```

Return type: Any type.

Semantics: Returns the result of using [at_key](#) to access the element with key type Key in Seq.

Header

```
#include <boost/fusion/sequence/intrinsic/at_key.hpp>
#include <boost/fusion/include/at_key.hpp>
```

Example

```
typedef map<pair<int, char>, pair<char, char>, pair<double, char> > mymap;
BOOST_MPL_ASSERT((boost::is_same<result_of::at_key<mymap, int>::type, char&>));
```

value_at_key

Description

Returns the actual element type associated with a Key from the [Sequence](#).

Synopsis

```
template<
    typename Seq,
    typename Key>
struct value_at_key
{
    typedef unspecified type;
};
```

Table 35. Parameters

Parameter	Requirement	Description
Seq	A model of Forward Sequence	Argument sequence
Key	Any type	Key type

Expression Semantics

```
result_of::value_at_key<Seq, Key>::type
```

Return type: Any type.

Semantics: Returns the actual element type associated with key type `Key` in `Seq`.

Header

```
#include <boost/fusion/sequence/intrinsic/value_at_key.hpp>
#include <boost/fusion/include/value_at_key.hpp>
```

Example

```
typedef map<pair<int, char>, pair<char, char>, pair<double, char> > mymap;
BOOST_MPL_ASSERT((boost::is_same<result_of::at_key<mymap, int>::type, char>));
```

swap

Description

Returns the return type of `swap`.

Synopsis

```
template<typename Seq1, typename Seq2>
struct swap
{
    typedef void type;
};
```

Table 36. Parameters

Parameters	Requirement	Description
Seq1, Seq2	Models of Forward Sequence	The sequences being swapped

Expression Semantics

```
result_of::swap<Seq1, Seq2>::type
```

Return type: void.

Semantics: Always returns void.

Header

```
#include <boost/fusion/sequence/intrinsic/swap.hpp>
#include <boost/fusion/include/swap.hpp>
```

Operator

These operators, like the [Algorithms](#), work generically on all Fusion sequences. All conforming Fusion sequences automatically get these operators for free.

I/O

The I/O operators: << and >> work generically on all Fusion sequences. The global operator<< has been overloaded for generic output streams such that [Sequence](#)(s) are output by recursively calling operator<< for each element. Analogously, the global operator>> has been overloaded to extract [Sequence](#)(s) from generic input streams by recursively calling operator>> for each element.

The default delimiter between the elements is space, and the [Sequence](#) is enclosed in parenthesis. For Example:

```
vector<float, int, std::string> a(1.0f, 2, std::string("Howdy folks!"));
cout << a;
```

outputs the [vector](#) as: (1.0 2 Howdy folks!)

The library defines three manipulators for changing the default behavior:

Manipulators

tuple_open(arg)	Defines the character that is output before the first element.
tuple_close(arg)	Defines the character that is output after the last element.
tuple_delimiter(arg)	Defines the delimiter character between elements.

The argument to tuple_open, tuple_close and tuple_delimiter may be a char, wchar_t, a C-string, or a wide C-string.

Example:

```
std::cout << tuple_open('[') << tuple_close(']') << tuple_delimiter(", ") << a;
```

outputs the same [vector](#), a as: [1.0, 2, Howdy folks!]

The same manipulators work with operator>> and istream as well. Suppose the std::cin stream contains the following data:

```
( 1 2 3 ) [ 4:5 ]
```

The code:

```
vector<int, int, int> i;
vector<int, int> j;

std::cin >> i;
std::cin >> set_open('[') >> set_close(']') >> set_delimiter(':');
std::cin >> j;
```

reads the data into the `vector`(s) `i` and `j`.

Note that extracting `Sequence`(s) with `std::string` or C-style string elements does not generally work, since the streamed `Sequence` representation may not be unambiguously parseable.

Header

```
#include <boost/fusion/sequence/io.hpp>
#include <boost/fusion/include/io.hpp>
```

in

Description

Read a `Sequence` from an input stream.

Synopsis

```
template <typename IStream, typename Sequence>
IStream&
operator>>(IStream& is, Sequence& seq);
```

Parameters

Parameter	Requirement	Description
<code>is</code>	An input stream.	Stream to extract information from.
<code>seq</code>	A <code>Sequence</code> .	The sequence to read.

Expression Semantics

```
is >> seq
```

Return type: `IStream&`

Semantics: For each element, `e`, in sequence, `seq`, call `is >> e`.

Header

```
#include <boost/fusion/sequence/io/in.hpp>
#include <boost/fusion/include/in.hpp>
```

Example

```
vector<int, std::string, char> v;
std::cin >> v;
```

out

Description

Write a [Sequence](#) to an output stream.

Synopsis

```
template <typename OStream, typename Sequence>
OStream&
operator<<(OStream& os, Sequence& seq);
```

Parameters

Parameter	Requirement	Description
os	An output stream.	Stream to write information to.
seq	A Sequence .	The sequence to write.

Expression Semantics

```
os << seq
```

Return type: OStream&

Semantics: For each element, e, in sequence, seq, call os << e.

Header

```
#include <boost/fusion/sequence/io/out.hpp>
#include <boost/fusion/include/out.hpp>
```

Example

```
std::cout << make_vector(123, "Hello", 'x') << std::endl;
```

Comparison

The Comparison operators: ==, !=, <, <=, >= and >= work generically on all Fusion sequences. Comparison operators are "short-circuited": elementary comparisons start from the first elements and are performed only until the result is clear.

Header

```
#include <boost/fusion/sequence/comparison.hpp>
#include <boost/fusion/include/comparison.hpp>
```

equal

Description

Compare two sequences for equality.

Synopsis

```
template <typename Seq1, typename Seq2>
bool
operator==(Seq1 const& a, Seq2 const& b);
```

Parameters

Parameter	Requirement	Description
a, b	Instances of Sequence	Sequence (s) to compare

Expression Semantics

```
a == b
```

Return type: bool

Requirements:

For each element, e1, in sequence a, and for each element, e2, in sequence b, a == b is a valid expression returning a type that is convertible to bool.

An attempt to compare two Sequences of different lengths results in a compile time error.

Semantics:

For each element, e1, in sequence a, and for each element, e2, in sequence b, e1 == e2 returns true. For any 2 zero length sequence_s, e and f, e == f returns true.

Header

```
#include <boost/fusion/sequence/comparison/equal_to.hpp>
#include <boost/fusion/include/equal_to.hpp>
```

Example

```
vector<int, char> v1(5, 'a');
vector<int, char> v2(5, 'a');
assert(v1 == v2);
```

not equal

Compare two sequences for inequality.

Synopsis

```
template <typename Seq1, typename Seq2>
bool
operator!=(Seq1 const& a, Seq2 const& b);
```

Parameters

Parameter	Requirement	Description
a, b	Instances of Sequence	Sequence (s) to compare

Expression Semantics

```
a != b
```

Return type: bool

Requirements:

For each element, e1, in sequence a, and for each element, e2, in sequence b, a == b is a valid expression returning a type that is convertible to bool.

An attempt to compare two Sequences of different lengths results in a compile time error.

Semantics:

Returns !(a == b).

Header

```
#include <boost/fusion/sequence/comparison/not_equal_to.hpp>
#include <boost/fusion/include/not_equal_to.hpp>
```

Example

```
vector<int, char> v3(5, 'b');
vector<int, char> t4(2, 'a');
assert(v1 != v3);
assert(v1 != t4);
assert(!(v1 != v2));
```

less than

Lexicographically compare two sequences.

Synopsis

```
template <typename Seq1, typename Seq2>
bool
operator<(Seq1 const& a, Seq2 const& b);
```

Parameters

Parameter	Requirement	Description
a, b	Instances of Sequence	Sequence (s) to compare

Expression Semantics

```
a < b
```

Return type: bool

Requirements:

For each element, e1, in sequence a, and for each element, e2, in sequence b, a < b is a valid expression returning a type that is convertible to bool.

An attempt to compare two Sequences of different lengths results in a compile time error.

Semantics: Returns the lexicographical comparison of between a and b.

Header

```
#include <boost/fusion/sequence/comparison/less.hpp>
#include <boost/fusion/include/less.hpp>
```

Example

```
vector<int, float> v1(4, 3.3f);
vector<short, float> v2(5, 3.3f);
vector<long, double> v3(5, 4.4);
assert(v1 < v2);
assert(v2 < v3);
```

less than equal

Lexicographically compare two sequences.

Synopsis

```
template <typename Seq1, typename Seq2>
bool
operator<=(Seq1 const& a, Seq2 const& b);
```

Parameters

Parameter	Requirement	Description
a, b	Instances of Sequence	Sequence (s) to compare

Expression Semantics

```
a <= b
```

Return type: `bool`

Requirements:

For each element, `e1`, in sequence `a`, and for each element, `e2`, in sequence `b`, `a < b` is a valid expression returning a type that is convertible to `bool`.

An attempt to compare two Sequences of different lengths results in a compile time error.

Semantics: Returns `!(b < a)`.

Header

```
#include <boost/fusion/sequence/comparison/less_equal.hpp>
#include <boost/fusion/include/less_equal.hpp>
```

Example

```
vector<int, float> v1(4, 3.3f);
vector<short, float> v2(5, 3.3f);
vector<long, double> v3(5, 4.4);
assert(v1 <= v2);
assert(v2 <= v3);
```

greater than

Lexicographically compare two sequences.

Synopsis

```
template <typename Seq1, typename Seq2>
bool
operator>(Seq1 const& a, Seq2 const& b);
```

Parameters

Parameter	Requirement	Description
<code>a, b</code>	Instances of Sequence	Sequence (s) to compare

Expression Semantics

```
a > b
```

Return type: `bool`

Requirements:

For each element, `e1`, in sequence `a`, and for each element, `e2`, in sequence `b`, `a < b` is a valid expression returning a type that is convertible to `bool`.

An attempt to compare two Sequences of different lengths results in a compile time error.

Semantics: Returns $b < a$.

Header

```
#include <boost/fusion/sequence/comparison/less_equal.hpp>
#include <boost/fusion/include/less_equal.hpp>
```

Example

```
vector<int, float> v1(4, 3.3f);
vector<short, float> v2(5, 3.3f);
vector<long, double> v3(5, 4.4);
assert(v2 > v1);
assert(v3 > v2);
```

greater than equal

Lexicographically compare two sequences.

Synopsis

```
template <typename Seq1, typename Seq2>
bool
operator>=(Seq1 const& a, Seq2 const& b);
```

Parameters

Parameter	Requirement	Description
a, b	Instances of Sequence	Sequence (s) to compare

Expression Semantics

```
a >= b
```

Return type: bool

Requirements:

For each element, e_1 , in sequence a, and for each element, e_2 , in sequence b, $a < b$ is a valid expression returning a type that is convertible to bool.

An attempt to compare two Sequences of different lengths results in a compile time error.

Semantics: Returns $!(a < b)$.

Header

```
#include <boost/fusion/sequence/comparison/greater_equal.hpp>
#include <boost/fusion/include/greater_equal.hpp>
```

Example

```
vector<int, float> v1(4, 3.3f);
vector<short, float> v2(5, 3.3f);
vector<long, double> v3(5, 4.4);
assert(v2 >= v1);
assert(v3 >= v2);
```

Container

Fusion provides a few predefined sequences out of the box. These *containers* actually hold heterogenously typed data; unlike [Views](#). These containers are more or less counterparts of those in [STL](#).

Header

```
#include <boost/fusion/container.hpp>
#include <boost/fusion/include/container.hpp>
```

vector

Description

`vector` is a [Random Access Sequence](#) of heterogenous typed data structured as a simple `struct` where each element is held as a member variable. `vector` is the simplest of the Fusion sequence container, and in many cases the most efficient.

Header

```
#include <boost/fusion/container/vector.hpp>
#include <boost/fusion/include/vector.hpp>
#include <boost/fusion/container/vector/vector_fwd.hpp>
#include <boost/fusion/include/vector_fwd.hpp>

// numbered forms
#include <boost/fusion/container/vector/vector10.hpp>
#include <boost/fusion/include/vector10.hpp>
#include <boost/fusion/container/vector/vector20.hpp>
#include <boost/fusion/include/vector20.hpp>
#include <boost/fusion/container/vector/vector30.hpp>
#include <boost/fusion/include/vector30.hpp>
#include <boost/fusion/container/vector/vector40.hpp>
#include <boost/fusion/include/vector40.hpp>
#include <boost/fusion/container/vector/vector50.hpp>
#include <boost/fusion/include/vector50.hpp>
```

Synopsis

Numbered forms

```

struct vector0;

template <typename T0>
struct vector1;

template <typename T0, typename T1>
struct vector2;

template <typename T0, typename T1, typename T2>
struct vector3;

...

template <typename T0, typename T1, typename T2..., typename TN>
struct vectorN;

```

Variadic form

```

template <
    typename T0 = unspecified
    , typename T1 = unspecified
    , typename T2 = unspecified
    ...
    , typename TN = unspecified
>
struct vector;

```

The numbered form accepts the exact number of elements. Example:

```
vector3<int, char, double>
```

The variadic form accepts 0 to FUSION_MAX_VECTOR_SIZE elements, where FUSION_MAX_VECTOR_SIZE is a user definable pre-defined maximum that defaults to 10. Example:

```
vector<int, char, double>
```

You may define the preprocessor constant FUSION_MAX_VECTOR_SIZE before including any Fusion header to change the default. Example:

```
#define FUSION_MAX_VECTOR_SIZE 20
```

Template parameters

Parameter	Description	Default
T0...TN	Element types	<i>unspecified</i>

Model of

- [Random Access Sequence](#)

Notation

v	Instance of vector
V	A vector type

`e0...en` Heterogeneous values

`s` A [Forward Sequence](#)

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in [Random Access Sequence](#).

Expression	Semantics
<code>V()</code>	Creates a vector with default constructed elements.
<code>V(e0, e1, ... en)</code>	Creates a vector with elements <code>e0...en</code> .
<code>V(s)</code>	Copy constructs a vector from a Forward Sequence , <code>s</code> .
<code>v = s</code>	Assigns to a vector, <code>v</code> , from a Forward Sequence , <code>s</code> .

Example

```
vector<int, float> v(12, 5.5f);
std::cout << at_c<0>(v) << std::endl;
std::cout << at_c<1>(v) << std::endl;
```

cons

Description

`cons` is a simple [Forward Sequence](#). It is a lisp style recursive list structure where `car` is the *head* and `cdr` is the *tail*: usually another `cons` structure or `nil`: the empty list. Fusion's `list` is built on top of this more primitive data structure. It is more efficient than `vector` when the target sequence is constructed piecemeal (a data at a time). The runtime cost of access to each element is peculiarly constant (see [Recursive Inlined Functions](#)).

Header

```
#include <boost/fusion/container/list/cons.hpp>
#include <boost/fusion/include/cons.hpp>
```

Synopsis

```
template <typename Car, typename Cdr = nil>
struct cons;
```

Template parameters

Parameter	Description	Default
<code>Car</code>	Head type	
<code>Cdr</code>	Tail type	<code>nil</code>

Model of

- [Forward Sequence](#)

Notation

<code>nil</code>	An empty <code>cons</code>
<code>C</code>	A <code>cons</code> type
<code>l, l2</code>	Instances of <code>cons</code>
<code>car</code>	An arbitrary data
<code>cdr</code>	Another <code>cons</code> list
<code>s</code>	A Forward Sequence
<code>N</code>	An MPL Integral Constant

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in [Forward Sequence](#).

Expression	Semantics
<code>nil()</code>	Creates an empty list.
<code>C()</code>	Creates a <code>cons</code> with default constructed elements.
<code>C(car)</code>	Creates a <code>cons</code> with <code>car</code> head and default constructed tail.
<code>C(car, cdr)</code>	Creates a <code>cons</code> with <code>car</code> head and <code>cdr</code> tail.
<code>C(s)</code>	Copy constructs a <code>cons</code> from a Forward Sequence , <code>s</code> .
<code>l = s</code>	Assigns to a <code>cons</code> , <code>l</code> , from a Forward Sequence , <code>s</code> .
<code>at<N>(l)</code>	The <code>N</code> th element from the beginning of the sequence; see at .



`at<N>(l)` is provided for convenience and compatibility with the original [Boost.Tuple](#) library, despite `cons` being a [Forward Sequence](#) only (`at` is supposed to be a [Random Access Sequence](#) requirement). The runtime complexity of `at` is constant (see [Recursive Inlined Functions](#)).

Example

```
cons<int, cons<float> > l(12, cons<float>(5.5f));
std::cout << at_c<0>(l) << std::endl;
std::cout << at_c<1>(l) << std::endl;
```

list

Description

`list` is a [Forward Sequence](#) of heterogenous typed data built on top of `cons`. It is more efficient than `vector` when the target sequence is constructed piecemeal (a data at a time). The runtime cost of access to each element is peculiarly constant (see [Recursive Inlined Functions](#)).

Header

```
#include <boost/fusion/container/list.hpp>
#include <boost/fusion/include/list.hpp>
#include <boost/fusion/container/list/list_fwd.hpp>
#include <boost/fusion/include/list_fwd.hpp>
```

Synopsis

```
template <
    typename T0 = unspecified
    , typename T1 = unspecified
    , typename T2 = unspecified
    ...
    , typename TN = unspecified
>
struct list;
```

The variadic class interface accepts 0 to FUSION_MAX_LIST_SIZE elements, where FUSION_MAX_LIST_SIZE is a user definable predefined maximum that defaults to 10. Example:

```
list<int, char, double>
```

You may define the preprocessor constant FUSION_MAX_LIST_SIZE before including any Fusion header to change the default. Example:

```
#define FUSION_MAX_LIST_SIZE 20
```

Template parameters

Parameter	Description	Default
T0...TN	Element types	<i>unspecified-type</i>

Model of

- [Forward Sequence](#)

Notation

L	A list type
l	An instance of list
e0...en	Heterogeneous values
s	A Forward Sequence
N	An MPL Integral Constant

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in [Forward Sequence](#).

Expression	Semantics
<code>L()</code>	Creates a list with default constructed elements.
<code>L(e0, e1, ... en)</code>	Creates a list with elements <code>e0...en</code> .
<code>L(s)</code>	Copy constructs a list from a Forward Sequence , <code>s</code> .
<code>l = s</code>	Assigns to a list, <code>l</code> , from a Forward Sequence , <code>s</code> .
<code>at<N>(l)</code>	The Nth element from the beginning of the sequence; see at .



`at<n>(l)` is provided for convenience and compatibility with the original [Boost.Tuple](#) library, despite `list` being a [Forward Sequence](#) only (`at` is supposed to be a [Random Access Sequence](#) requirement). The runtime complexity of `at` is constant (see [Recursive Inlined Functions](#)).

Example

```
list<int, float> l(12, 5.5f);
std::cout << at_c<0>(l) << std::endl;
std::cout << at_c<1>(l) << std::endl;
```

set

Description

`set` is an [Associative Sequence](#) of heterogeneous typed data elements. Type identity is used to impose an equivalence relation on keys. The element's type is its key. A set may contain at most one element for each key. Membership testing and element key lookup has constant runtime complexity (see [Overloaded Functions](#)).

Header

```
#include <boost/fusion/container/set.hpp>
#include <boost/fusion/include/set.hpp>
#include <boost/fusion/container/set_fwd.hpp>
#include <boost/fusion/include/set_fwd.hpp>
```

Synopsis

```
template <
    typename T0 = unspecified
    , typename T1 = unspecified
    , typename T2 = unspecified
    ...
    , typename TN = unspecified
>
struct set;
```

The variadic class interface accepts 0 to `FUSION_MAX_SET_SIZE` elements, where `FUSION_MAX_SET_SIZE` is a user definable predefined maximum that defaults to 10. Example:

```
set<int, char, double>
```

You may define the preprocessor constant `FUSION_MAX_SET_SIZE` before including any Fusion header to change the default. Example:

```
#define FUSION_MAX_SET_SIZE 20
```

Template parameters

Parameter	Description	Default
T0...TN	Element types	<i>unspecified-type</i>

Model of

- [Associative Sequence](#)
- [Forward Sequence](#)

Notation

<code>S</code>	A <code>set</code> type
<code>s</code>	An instance of <code>set</code>
<code>e0...en</code>	Heterogeneous values
<code>fs</code>	A Forward Sequence

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in [Random Access Sequence](#) and [Associative Sequence](#).

Expression	Semantics
<code>S()</code>	Creates a set with default constructed elements.
<code>S(e0, e1, ... en)</code>	Creates a set with elements <code>e0...en</code> .
<code>S(fs)</code>	Copy constructs a set from a Forward Sequence <code>fs</code> .
<code>s = fs</code>	Assigns to a set, <code>s</code> , from a Forward Sequence <code>fs</code> .

Example

```
typedef set<int, float> S;
S s(12, 5.5f);
std::cout << at_key<int>(s) << std::endl;
std::cout << at_key<float>(s) << std::endl;
std::cout << result_of::has_key<S, double>::value << std::endl;
```

map

Description

map is an [Associative Sequence](#) of heterogeneous typed data elements. Each element is a key/data pair (see [fusion::pair](#)) where the key has no data (type only). Type identity is used to impose an equivalence relation on keys. A map may contain at most one element for each key. Membership testing and element key lookup has constant runtime complexity (see [Overloaded Functions](#)).

Header

```
#include <boost/fusion/container/map.hpp>
#include <boost/fusion/include/map.hpp>
#include <boost/fusion/container/map_fwd.hpp>
#include <boost/fusion/include/map_fwd.hpp>
```

Synopsis

```
template <
    typename T0 = unspecified
    , typename T1 = unspecified
    , typename T2 = unspecified
    ...
    , typename TN = unspecified
>
struct map;
```

The variadic class interface accepts 0 to FUSION_MAX_MAP_SIZE elements, where FUSION_MAX_MAP_SIZE is a user definable predefined maximum that defaults to 10. Example:

```
map<pair<int, char>, pair<char, char>, pair<double, char> >
```

You may define the preprocessor constant FUSION_MAX_MAP_SIZE before including any Fusion header to change the default. Example:

```
#define FUSION_MAX_MAP_SIZE 20
```

Template parameters

Parameter	Description	Default
T0...TN	Element types	<i>unspecified-type</i>

Model of

- [Associative Sequence](#)
- [Forward Sequence](#)

Notation

<code>M</code>	A map type
<code>m</code>	An instance of map
<code>e0...en</code>	Heterogeneous key/value pairs (see fusion::pair)
<code>s</code>	A Forward Sequence

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in [Random Access Sequence](#) and [Associative Sequence](#).

Expression	Semantics
<code>M()</code>	Creates a map with default constructed elements.
<code>M(e0, e1, ... en)</code>	Creates a map with element pairs <code>e0...en</code> .
<code>M(s)</code>	Copy constructs a map from a Forward Sequence <code>s</code> .
<code>m = s</code>	Assigns to a map, <code>m</code> , from a Forward Sequence <code>s</code> .

Example

```
typedef map<
    pair<int, char>
    , pair<double, std::string> >
map_type;

map_type m(
    make_pair<int>( 'X' )
    , make_pair<double>( "Men" ) );

std::cout << at_key<int>(m) << std::endl;
std::cout << at_key<double>(m) << std::endl;
```

Generation

These are the functions that you can use to generate various forms of [Container](#) from elemental values.

Header

```
#include <boost/fusion/container/generation.hpp>
#include <boost/fusion/include/generation.hpp>
```

Functions

make_list

Description

Create a [list](#) from one or more values.

Synopsis

```
template <typename T0, typename T1, ... typename TN>
typename result_of::make_list<T0, T1, ... TN>::type
make_list(T0 const& x0, T1 const& x1... TN const& xN);
```

The variadic function accepts 0 to `FUSION_MAX_LIST_SIZE` elements, where `FUSION_MAX_LIST_SIZE` is a user definable predefined maximum that defaults to 10. You may define the preprocessor constant `FUSION_MAX_LIST_SIZE` before including any Fusion header to change the default. Example:

```
#define FUSION_MAX_LIST_SIZE 20
```

Parameters

Parameter	Requirement	Description
<code>x0, x1, ... xN</code>	Instances of <code>T0, T1, ... TN</code>	The arguments to <code>make_list</code>

Expression Semantics

```
make_list(x0, x1, ... xN);
```

Return type: `result_of::make_list<T0, T1, ... TN>::type`

Semantics: Create a `list` from `x0, x1, ... xN`.

Header

```
#include <boost/fusion/container/generation/make_list.hpp>
#include <boost/fusion/include/make_list.hpp>
```

Example

```
make_list(123, "hello", 12.5)
```

See also

[boost::ref](#)

make_cons

Description

Create a `cons` from `car` (*head*) and optional `cdr` (*tail*).

Synopsis

```
template <typename Car>
typename result_of::make_cons<Car>::type
make_cons(Car const& car);

template <typename Car, typename Cdr>
typename result_of::make_cons<Car, Cdr>::type
make_cons(Car const& car, Cdr const& cdr);
```

Parameters

Parameter	Requirement	Description
car	Instance of Car	The list's head
cdr	Instance of Cdr	The list's tail (optional)

Expression Semantics

```
make_cons(car, cdr);
```

Return type: `result_of::make_cons<Car, Cdr>::type` or `result_of::make_cons<Car>::type`

Semantics: Create a `cons` from `car` (*head*) and optional `cdr` (*tail*).

Header

```
#include <boost/fusion/container/generation/make_cons.hpp>
#include <boost/fusion/include/make_cons.hpp>
```

Example

```
make_cons('x', make_cons(123))
```

See also

[boost::ref](#)

make_vector

Description

Create a `vector` from one or more values.

Synopsis

```
template <typename T0, typename T1, ... typename TN>
typename result_of::make_vector<T0, T1, ... TN>::type
make_vector(T0 const& x0, T1 const& x1... TN const& xN);
```

The variadic function accepts 0 to `FUSION_MAX_VECTOR_SIZE` elements, where `FUSION_MAX_VECTOR_SIZE` is a user definable predefined maximum that defaults to 10. You may define the preprocessor constant `FUSION_MAX_VECTOR_SIZE` before including any Fusion header to change the default. Example:

```
#define FUSION_MAX_VECTOR_SIZE 20
```

Parameters

Parameter	Requirement	Description
<code>x0, x1, ... xN</code>	Instances of <code>T0, T1, ... TN</code>	The arguments to <code>make_vector</code>

Expression Semantics

```
make_vector(x0, x1, ... xN);
```

Return type: `result_of::make_vector<T0, T1, ... TN>::type`

Semantics: Create a `vector` from `x0, x1, ... xN`.

Header

```
#include <boost/fusion/container/generation/make_vector.hpp>
#include <boost/fusion/include/make_vector.hpp>
```

Example

```
make_vector(123, "hello", 12.5)
```

See also

`boost::ref`

make_set

Description

Create a `set` from one or more values.

Synopsis

```
template <typename T0, typename T1, ... typename TN>
typename result_of::make_set<T0, T1, ... TN>::type
make_set(T0 const& x0, T1 const& x1... TN const& xN);
```

The variadic function accepts 0 to `FUSION_MAX_SET_SIZE` elements, where `FUSION_MAX_SET_SIZE` is a user definable predefined maximum that defaults to 10. You may define the preprocessor constant `FUSION_MAX_SET_SIZE` before including any Fusion header to change the default. Example:

```
#define FUSION_MAX_SET_SIZE 20
```

Parameters

Parameter	Requirement	Description
<code>x0, x1, ... xN</code>	Instances of <code>T0, T1, ... TN</code>	The arguments to <code>make_set</code>

Expression Semantics

```
make_set(x0, x1, ... xN);
```

Return type: `result_of::make_set<T0, T1, ... TN>::type`

Semantics: Create a `set` from `x0, x1, ... xN`.

Precondition: There may be no duplicate key types.

Header

```
#include <boost/fusion/container/generation/make_set.hpp>
#include <boost/fusion/include/make_set.hpp>
```

Example

```
make_set(123, "hello", 12.5)
```

See also

[boost::ref](#)

make_map

Description

Create a `map` from one or more key/data pairs.

Synopsis

```
template <
    typename K0, typename K1, ... typename KN
    , typename T0, typename T1, ... typename TN>
typename result_of::make_map<K0, K0, ... KN, T0, T1, ... TN>::type
make_map(T0 const& x0, T1 const& x1... TN const& xN);
```

The variadic function accepts 0 to `FUSION_MAX_MAP_SIZE` elements, where `FUSION_MAX_MAP_SIZE` is a user definable predefined maximum that defaults to 10. You may define the preprocessor constant `FUSION_MAX_MAP_SIZE` before including any Fusion header to change the default. Example:

```
#define FUSION_MAX_MAP_SIZE 20
```

Parameters

Parameter	Requirement	Description
<code>K0, K1, ... KN</code>	The key types	Keys associated with <code>x0, x1, ... xN</code>
<code>x0, x1, ... xN</code>	Instances of <code>T0, T1, ... TN</code>	The arguments to <code>make_map</code>

Expression Semantics

```
make_map<K0, K1, ... KN>(x0, x1, ... xN);
```

Return type: `result_of::make_map<K0, K0, ... KN, T0, T1, ... TN>::type`

Semantics: Create a `map` from `K0, K1, ... KN` keys and `x0, x1, ... xN` data.

Precondition: There may be no duplicate key types.

Header

```
#include <boost/fusion/container/generation/make_map.hpp>
#include <boost/fusion/include/make_map.hpp>
```

Example

```
make_map<int, double>('X', "Men")
```

See also

`boost::ref`, `fusion::pair`

Tiers

Tiers are sequences, where all elements are non-const reference types. They are constructed with a call to a couple of *tie* function templates. The succeeding sections document the various *tier* flavors.

- `list_tie`
- `vector_tie`
- `map_tie`

Example:

```
int i; char c; double d;
...
vector_tie(i, c, a);
```

The `vector_tie` function creates a `vector` of type `vector<int&, char&, double&>`. The same result could be achieved with the call `make_vector(ref(i), ref(c), ref(a))`⁹.

A *tie* can be used to 'unpack' another tuple into variables. E.g.:

```
int i; char c; double d;
vector_tie(i, c, d) = make_vector(1, 'a', 5.5);
std::cout << i << " " << c << " " << d;
```

This code prints 1 a 5.5 to the standard output stream. A sequence unpacking operation like this is found for example in ML and Python. It is convenient when calling functions which return sequences.

Ignore

There is also an object called *ignore* which allows you to ignore an element assigned by a sequence. The idea is that a function may return a sequence, only part of which you are interested in. For example:

⁹ see [Boost.Ref](#) for details about `ref`

```
char c;
vector_tie(ignore, c) = make_vector(1, 'a');
```

list_tie

Description

Constructs a tie using a `list` sequence.

Synopsis

```
template <typename T0, typename T1, ... typename TN>
list<T0&, T1&, ... TN&>
list_tie(T0& x0, T1& x1... TN& xN);
```

The variadic function accepts 0 to `FUSION_MAX_LIST_SIZE` elements, where `FUSION_MAX_LIST_SIZE` is a user definable predefined maximum that defaults to 10. You may define the preprocessor constant `FUSION_MAX_LIST_SIZE` before including any Fusion header to change the default. Example:

```
#define FUSION_MAX_LIST_SIZE 20
```

Parameters

Parameter	Requirement	Description
<code>x0, x1, ... xN</code>	Instances of <code>T0, T1, ... TN</code>	The arguments to <code>list_tie</code>

Expression Semantics

```
list_tie(x0, x1, ... xN);
```

Return type: `list<T0&, T1&, ... TN&>`

Semantics: Create a `list` of references from `x0, x1, ... xN`.

Header

```
#include <boost/fusion/container/generation/list_tie.hpp>
#include <boost/fusion/include/list_tie.hpp>
```

Example

```
int i = 123;
double d = 123.456;
list_tie(i, d)
```

vector_tie

Description

Constructs a tie using a `vector` sequence.

Synopsis

```
template <typename T0, typename T1, ... typename TN>
vector<T0&, T1&, ... TN&>
vector_tie(T0& x0, T1& x1... TN& xN);
```

The variadic function accepts 0 to `FUSION_MAX_VECTOR_SIZE` elements, where `FUSION_MAX_VECTOR_SIZE` is a user definable predefined maximum that defaults to 10. You may define the preprocessor constant `FUSION_MAX_VECTOR_SIZE` before including any Fusion header to change the default. Example:

```
#define FUSION_MAX_VECTOR_SIZE 20
```

Parameters

Parameter	Requirement	Description
<code>x0, x1, ... xN</code>	Instances of <code>T0, T1, ... TN</code>	The arguments to <code>vector_tie</code>

Expression Semantics

```
vector_tie(x0, x1, ... xN);
```

Return type: `vector<T0&, T1&, ... TN&>`

Semantics: Create a `vector` of references from `x0, x1, ... xN`.

Header

```
#include <boost/fusion/container/generation/vector_tie.hpp>
#include <boost/fusion/include/vector_tie.hpp>
```

Example

```
int i = 123;
double d = 123.456;
vector_tie(i, d)
```

map_tie

Description

Constructs a tie using a `map` sequence.

Synopsis

```
template <typename K0, typename K1, ... typename KN, typename D0, typename D1, ... typename DN>
map<pair<K0, D0&>, pair<K1, D1&>, ... pair<KN, DN&> >
map_tie(D0& d0, D1& d1... DN& dN);
```

The variadic function accepts 0 to `FUSION_MAX_MAP_SIZE` elements, where `FUSION_MAX_MAP_SIZE` is a user definable predefined maximum that defaults to 10, and a corresponding number of key types. You may define the preprocessor constant `FUSION_MAX_MAP_SIZE` before including any Fusion header to change the default. Example:


```
#define FUSION_MAX_MAP_SIZE 20
```

Parameters

Parameter	Requirement	Description
K_0, K_1, \dots, K_N	Any type	The key types associated with each of the x_1, x_2, \dots, x_N values
x_0, x_1, \dots, x_N	Instances of T_0, T_1, \dots, T_N	The arguments to <code>map_tie</code>

Expression Semantics

```
map_tie<K0, K1, ... KN>(x0, x1, ... xN);
```

Return type: `map<pair<K0, D0&>, pair<K1, D1&>, ... pair<KN, DN&>>`

Semantics: Create a `map` of references from x_0, x_1, \dots, x_N with keys K_0, K_1, \dots, K_N

Header

```
#include <boost/fusion/container/generation/map_tie.hpp>
#include <boost/fusion/include/map_tie.hpp>
```

Example

```
struct int_key;
struct double_key;
...
int i = 123;
double d = 123.456;
map_tie<int_key, double_key>(i, d)
```

MetaFunctions

make_list

Description

Returns the result type of `make_list`.

Synopsis

```
template <typename T0, typename T1, ... typename TN>
struct make_list;
```

The variadic function accepts 0 to `FUSION_MAX_LIST_SIZE` elements, where `FUSION_MAX_LIST_SIZE` is a user definable predefined maximum that defaults to 10. You may define the preprocessor constant `FUSION_MAX_LIST_SIZE` before including any Fusion header to change the default. Example:

```
#define FUSION_MAX_LIST_SIZE 20
```

Parameters

Parameter	Requirement	Description
T0, T1, ... TN	Any type	Template arguments to <code>make_list</code>

Expression Semantics

```
result_of::make_list<T0, T1, ... TN>::type
```

Return type: A `list` with elements of types converted following the rules for *element conversion*.

Semantics: Create a `list` from T0, T1, ... TN.

Header

```
#include <boost/fusion/container/generation/make_list.hpp>
#include <boost/fusion/include/make_list.hpp>
```

Example

```
result_of::make_list<int, const char(&)[7], double>::type
```

make_cons

Description

Returns the result type of `make_cons`.

Synopsis

```
template <typename Car, typename Cdr = nil>
struct make_cons;
```

Parameters

Parameter	Requirement	Description
Car	Any type	The list's head type
Cdr	A cons	The list's tail type (optional)

Expression Semantics

```
result_of::make_cons<Car, Cdr>::type
```

Return type: A `cons` with head element, Car, of type converted following the rules for *element conversion*, and tail, Cdr.

Semantics: Create a `cons` from Car (*head*) and optional Cdr (*tail*).

Header

```
#include <boost/fusion/container/generation/make_cons.hpp>
#include <boost/fusion/include/make_cons.hpp>
```

Example

```
result_of::make_cons<char, result_of::make_cons<int>::type>::type
```

make_vector

Description

Returns the result type of `make_vector`.

Synopsis

```
template <typename T0, typename T1, ... typename TN>
struct make_vector;
```

The variadic function accepts 0 to `FUSION_MAX_VECTOR_SIZE` elements, where `FUSION_MAX_VECTOR_SIZE` is a user definable predefined maximum that defaults to 10. You may define the preprocessor constant `FUSION_MAX_VECTOR_SIZE` before including any Fusion header to change the default. Example:

```
#define FUSION_MAX_VECTOR_SIZE 20
```

Parameters

Parameter	Requirement	Description
<code>T0, T1, ... TN</code>	Any type	Template arguments to <code>make_vector</code>

Expression Semantics

```
result_of::make_vector<T0, T1, ... TN>::type
```

Return type: A `vector` with elements of types converted following the rules for *element conversion*.

Semantics: Create a `vector` from `T0, T1, ... TN`.

Header

```
#include <boost/fusion/container/generation/make_list.hpp>
#include <boost/fusion/include/make_list.hpp>
```

Example

```
result_of::make_vector<int, const char(&)[7], double>::type
```

make_set

Description

Returns the result type of `make_set`.

Synopsis

```
template <typename T0, typename T1, ... typename TN>
struct make_set;
```

The variadic function accepts 0 to `FUSION_MAX_SET_SIZE` elements, where `FUSION_MAX_SET_SIZE` is a user definable predefined maximum that defaults to 10. You may define the preprocessor constant `FUSION_MAX_SET_SIZE` before including any Fusion header to change the default. Example:

```
#define FUSION_MAX_SET_SIZE 20
```

Parameters

Parameter	Requirement	Description
<code>T0, T1, ... TN</code>	Any type	The arguments to <code>make_set</code>

Expression Semantics

```
result_of::make_set<T0, T1, ... TN>::type
```

Return type: A [set](#) with elements of types converted following the rules for *element conversion*.

Semantics: Create a [set](#) from `T0, T1, ... TN`.

Precondition: There may be no duplicate key types.

Header

```
#include <boost/fusion/container/generation/make_set.hpp>
#include <boost/fusion/include/make_set.hpp>
```

Example

```
result_of::make_set<int, char, double>::type
```

make_map

Description

Returns the result type of [make_map](#).

Synopsis

```
template <
    typename K0, typename K1, ... typename KN
    , typename T0, typename T1, ... typename TN>
struct make_map;
```

The variadic function accepts 0 to `FUSION_MAX_MAP_SIZE` elements, where `FUSION_MAX_MAP_SIZE` is a user definable predefined maximum that defaults to 10. You may define the preprocessor constant `FUSION_MAX_MAP_SIZE` before including any Fusion header to change the default. Example:

```
#define FUSION_MAX_MAP_SIZE 20
```

Parameters

Parameter	Requirement	Description
$K_0, K_1, \dots K_N$	Any type	Keys associated with $T_0, T_1, \dots T_N$
$T_0, T_1, \dots T_N$	Any type	Data associated with keys $K_0, K_1, \dots K_N$

Expression Semantics

```
result_of::make_map<K0, K1, ... KN, T0, T1, ... TN>::type;
```

Return type: `result_of::make_map<K0, K0, ... KN, T0, T1, ... TN>::type`

Semantics: A `map` with `fusion::pair` elements where the `second_type` is converted following the rules for *element conversion*.

Precondition: There may be no duplicate key types.

Header

```
#include <boost/fusion/container/generation/make_map.hpp>
#include <boost/fusion/include/make_map.hpp>
```

Example

```
result_of::make_map<int, double, char, double>::type
```

See also

`fusion::pair`

list_tie

Description

Returns the result type of `list_tie`.

Synopsis

```
template <typename T0, typename T1, ... typename TN>
struct list_tie;
```

The variadic function accepts 0 to `FUSION_MAX_LIST_SIZE` elements, where `FUSION_MAX_LIST_SIZE` is a user definable predefined maximum that defaults to 10. You may define the preprocessor constant `FUSION_MAX_LIST_SIZE` before including any Fusion header to change the default. Example:

```
#define FUSION_MAX_LIST_SIZE 20
```

Parameters

Parameter	Requirement	Description
$T_0, T_1, \dots T_N$	Any type	The arguments to <code>list_tie</code>

Expression Semantics

```
result_of::list_tie<T0, T1, ... TN>::type;
```

Return type: `list<T0&, T1&, ... TN&>`

Semantics: Create a `list` of references from $T_0, T_1, \dots T_N$.

Header

```
#include <boost/fusion/container/generation/list_tie.hpp>
#include <boost/fusion/include/list_tie.hpp>
```

Example

```
result_of::list_tie<int, double>::type
```

vector_tie

Description

Returns the result type of `vector_tie`.

Synopsis

```
template <typename T0, typename T1, ... typename TN>
struct vector_tie;
```

The variadic function accepts 0 to `FUSION_MAX_VECTOR_SIZE` elements, where `FUSION_MAX_VECTOR_SIZE` is a user definable predefined maximum that defaults to 10. You may define the preprocessor constant `FUSION_MAX_VECTOR_SIZE` before including any Fusion header to change the default. Example:

```
#define FUSION_MAX_VECTOR_SIZE 20
```

Parameters

Parameter	Requirement	Description
$T_0, T_1, \dots T_N$	Any type	The arguments to <code>vector_tie</code>

Expression Semantics

```
result_of::vector_tie<T0, T1, ... TN>::type;
```

Return type: `vector<T0&, T1&, ... TN&>`

Semantics: Create a [vector](#) of references from T_0, T_1, \dots, T_N .

Header

```
#include <boost/fusion/container/generation/vector_tie.hpp>
#include <boost/fusion/include/vector_tie.hpp>
```

Example

```
result_of::vector_tie<int, double>::type
```

map_tie

Description

Returns the result type of [map_tie](#).

Synopsis

```
template <typename K0, typename K1, ... typename KN, typename D0, typename D1, ... typename DN>
struct map_tie;
```

The variadic function accepts 0 to `FUSION_MAX_MAP_SIZE` elements, where `FUSION_MAX_MAP_SIZE` is a user definable predefined maximum that defaults to 10. You may define the preprocessor constant `FUSION_MAX_MAP_SIZE` before including any Fusion header to change the default. Example:

```
#define FUSION_MAX_MAP_SIZE 20
```

Parameters

Parameter	Requirement	Description
K_0, K_1, \dots, K_N	Any type	The key types for map_tie
D_0, D_1, \dots, D_N	Any type	The arguments types for map_tie

Expression Semantics

```
result_of::map_tie<K0, K1, ... KN, D0, D1, ... DN>::type;
```

Return type: `map<pair<K0, D0&>, pair<K1, D1&>, ... pair<KN, DN&>>`

Semantics: Create a [map](#) of references from D_0, D_1, \dots, D_N with keys K_0, K_1, \dots, K_N

Header

```
#include <boost/fusion/container/generation/map_tie.hpp>
#include <boost/fusion/include/map_tie.hpp>
```

Example

```
struct int_key;
struct double_key;
...
result_of::map_tie<int_key, double_key, int, double>::type
```

Conversion

All fusion sequences can be converted to one of the [Container](#) types using one of these conversion functions.

Header

```
#include <boost/fusion/include/convert.hpp>
```

Functions

as_list

Description

Convert a fusion sequence to a [list](#).

Synopsis

```
template <typename Sequence>
typename result_of::as_list<Sequence>::type
as_list(Sequence& seq);

template <typename Sequence>
typename result_of::as_list<Sequence const>::type
as_list(Sequence const& seq);
```

Parameters

Parameter	Requirement	Description
seq	An instance of Sequence	The sequence to convert.

Expression Semantics

```
as_list(seq);
```

Return type: `result_of::as_list<Sequence>::type`

Semantics: Convert a fusion sequence, `seq`, to a [list](#).

Header

```
#include <boost/fusion/container/list/convert.hpp>
#include <boost/fusion/include/as_list.hpp>
```

Example

```
as_list(make_vector('x', 123, "hello"))
```

as_vector

Description

Convert a fusion sequence to a [vector](#).

Synopsis

```
template <typename Sequence>
typename result_of::as_vector<Sequence>::type
as_vector(Sequence& seq);

template <typename Sequence>
typename result_of::as_vector<Sequence const>::type
as_vector(Sequence const& seq);
```

Parameters

Parameter	Requirement	Description
seq	An instance of Sequence	The sequence to convert.

Expression Semantics

```
as_vector(seq);
```

Return type: `result_of::as_vector<Sequence>::type`

Semantics: Convert a fusion sequence, `seq`, to a [vector](#).

Header

```
#include <boost/fusion/container/vector/convert.hpp>
#include <boost/fusion/include/as_vector.hpp>
```

Example

```
as_vector(make_list('x', 123, "hello"))
```

as_set

Description

Convert a fusion sequence to a [set](#).

Synopsis

```
template <typename Sequence>
typename result_of::as_set<Sequence>::type
as_set(Sequence& seq);

template <typename Sequence>
typename result_of::as_set<Sequence const>::type
as_set(Sequence const& seq);
```

Parameters

Parameter	Requirement	Description
seq	An instance of Sequence	The sequence to convert.

Expression Semantics

```
as_set(seq);
```

Return type: `result_of::as_set<Sequence>::type`

Semantics: Convert a fusion sequence, `seq`, to a `set`.

Precondition: There may be no duplicate key types.

Header

```
#include <boost/fusion/container/set/convert.hpp>
#include <boost/fusion/include/as_set.hpp>
```

Example

```
as_set(make_vector('x', 123, "hello"))
```

as_map

Description

Convert a fusion sequence to a `map`.

Synopsis

```
template <typename Sequence>
typename result_of::as_map<Sequence>::type
as_map(Sequence& seq);

template <typename Sequence>
typename result_of::as_map<Sequence const>::type
as_map(Sequence const& seq);
```

Parameters

Parameter	Requirement	Description
seq	An instance of Sequence	The sequence to convert.

Expression Semantics

```
as_map(seq);
```

Return type: `result_of::as_map<Sequence>::type`

Semantics: Convert a fusion sequence, `seq`, to a `map`.

Precondition: The elements of the sequence are assumed to be `__fusionpair`s. There may be no duplicate `fusion::pair` key types.

Header

```
#include <boost/fusion/container/map/convert.hpp>
#include <boost/fusion/include/as_map.hpp>
```

Example

```
as_map(make_vector(
    make_pair<int>('X')
    , make_pair<double>("Men")))
```

Metafunctions

as_list

Description

Returns the result type of `as_list`.

Synopsis

```
template <typename Sequence>
struct as_list;
```

Parameters

Parameter	Requirement	Description
Sequence	A fusion Sequence	The sequence type to convert.

Expression Semantics

```
result_of::as_list<Sequence>::type;
```

Return type: A `list` with same elements as the input sequence, `Sequence`.

Semantics: Convert a fusion sequence, `Sequence`, to a `list`.

Header

```
#include <boost/fusion/container/list/convert.hpp>
#include <boost/fusion/include/as_list.hpp>
```

Example

```
result_of::as_list<vector<char, int> >::type
```

as_vector

Description

Returns the result type of `as_vector`.

Synopsis

```
template <typename Sequence>
struct as_vector;
```

Parameters

Parameter	Requirement	Description
Sequence	A fusion Sequence	The sequence to convert.

Expression Semantics

```
result_of::as_vector<Sequence>::type;
```

Return type: A [vector](#) with same elements as the input sequence, `Sequence`.

Semantics: Convert a fusion sequence, `Sequence`, to a [vector](#).

Header

```
#include <boost/fusion/container/vector/convert.hpp>
#include <boost/fusion/include/as_vector.hpp>
```

Example

```
result_of::as_vector<list<char, int> >::type
```

as_set

Description

Returns the result type of `as_set`.

Synopsis

```
template <typename Sequence>
struct as_set;
```

Parameters

Parameter	Requirement	Description
Sequence	A fusion Sequence	The sequence to convert.

Expression Semantics

```
result_of::as_set<Sequence>::type;
```

Return type: A [set](#) with same elements as the input sequence, Sequence.

Semantics: Convert a fusion sequence, Sequence, to a [set](#).

Precondition: There may be no duplicate key types.

Header

```
#include <boost/fusion/container/set/convert.hpp>
#include <boost/fusion/include/as_set.hpp>
```

Example

```
result_of::as_set<vector<char, int> >::type
```

as_map

Description

Returns the result type of [as_map](#).

Synopsis

```
template <typename Sequence>
struct as_map;
```

Parameters

Parameter	Requirement	Description
Sequence	A fusion Sequence	The sequence to convert.

Expression Semantics

```
result_of::as_map<Sequence>::type;
```

Return type: A [map](#) with same elements as the input sequence, Sequence.

Semantics: Convert a fusion sequence, Sequence, to a [map](#).

Precondition: The elements of the sequence are assumed to be `__fusionpair`s. There may be no duplicate `fusion::pair` key types.

Header

```
#include <boost/fusion/container/map/convert.hpp>
#include <boost/fusion/include/as_map.hpp>
```

Example

```
result_of::as_map<vector<
    fusion::pair<int, char>
    , fusion::pair<double, std::string> > >::type
```

View

Views are sequences that do not actually contain data, but instead impart an alternative presentation over the data from one or more underlying sequences. Views are proxies. They provide an efficient yet purely functional way to work on potentially expensive sequence operations. Views are inherently lazy. Their elements are only computed on demand only when the elements of the underlying sequence(s) are actually accessed. Views' lazy nature make them very cheap to copy and be passed around by value.

Header

```
#include <boost/fusion/view.hpp>
#include <boost/fusion/include/view.hpp>
```

single_view

`single_view` is a view into a value as a single element sequence.

Header

```
#include <boost/fusion/view/single_view.hpp>
#include <boost/fusion/include/single_view.hpp>
```

Synopsis

```
template <typename T>
struct single_view;
```

Template parameters

Parameter	Description	Default
T	Any type	

Model of

- [Forward Sequence](#)

Notation

S A `single_view` type

<code>s, s2</code>	Instances of <code>single_view</code>
<code>x</code>	An instance of <code>T</code>

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in [Forward Sequence](#).

Expression	Semantics
<code>S(x)</code>	Creates a <code>single_view</code> from <code>x</code> .
<code>S(s)</code>	Copy constructs a <code>single_view</code> from another <code>single_view</code> , <code>s</code> .
<code>s = s2</code>	Assigns to a <code>single_view</code> , <code>s</code> , from another <code>single_view</code> , <code>s2</code> .

Example

```
single_view<int> view(3);
std::cout << view << std::endl;
```

filter_view

Description

`filter_view` is a view into a subset of its underlying sequence's elements satisfying a given predicate (an [MPL](#) metafunction). The `filter_view` presents only those elements for which its predicate evaluates to `mpl::true_`.

Header

```
#include <boost/fusion/view/filter_view.hpp>
#include <boost/fusion/include/filter_view.hpp>
```

Synopsis

```
template <typename Sequence, typename Pred>
struct filter_view;
```

Template parameters

Parameter	Description	Default
Sequence	A Forward Sequence	
Pred	Unary Metafunction returning an <code>mpl::bool_</code>	

Model of

- [Forward Sequence](#)
- [Associative Sequence](#) if Sequence implements the [Associative Sequence](#) model.

Notation

F	A <code>filter_view</code> type
f, f2	Instances of <code>filter_view</code>
s	A Forward Sequence

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in the implemented models.

Expression	Semantics
<code>F(s)</code>	Creates a <code>filter_view</code> given a sequence, <code>s</code> .
<code>F(f)</code>	Copy constructs a <code>filter_view</code> from another <code>filter_view</code> , <code>f</code> .
<code>f = f2</code>	Assigns to a <code>filter_view</code> , <code>f</code> , from another <code>filter_view</code> , <code>f2</code> .

Example

```
using boost::mpl::_;
using boost::mpl::not_;
using boost::is_class;

typedef vector<std::string, char, long, bool, double> vector_type;

vector_type v("a-string", '@', 987654, true, 6.6);
filter_view<vector_type const, not_<is_class<_> > > view(v);
std::cout << view << std::endl;
```

iterator_range

Description

`iterator_range` presents a sub-range of its underlying sequence delimited by a pair of iterators.

Header

```
#include <boost/fusion/view/iterator_range.hpp>
#include <boost/fusion/include/iterator_range.hpp>
```

Synopsis

```
template <typename First, typename Last>
struct iterator_range;
```

Template parameters

Parameter	Description	Default
First	A fusion Iterator	
Last	A fusion Iterator	

Model of

- [Forward Sequence](#), [Bidirectional Sequence](#) or [Random Access Sequence](#) depending on the traversal characteristics (see [Sequence Traversal Concept](#)) of its underlying sequence.
- [Associative Sequence](#) if `First` and `Last` implement the [Associative Iterator](#) model.

Notation

<code>IR</code>	An <code>iterator_range</code> type
<code>f</code>	An instance of <code>First</code>
<code>l</code>	An instance of <code>Last</code>
<code>ir, ir2</code>	Instances of <code>iterator_range</code>

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in the implemented models.

Expression	Semantics
<code>IR(f, l)</code>	Creates an <code>iterator_range</code> given iterators, <code>f</code> and <code>l</code> .
<code>IR(ir)</code>	Copy constructs an <code>iterator_range</code> from another <code>iterator_range</code> , <code>ir</code> .
<code>ir = ir2</code>	Assigns to a <code>iterator_range</code> , <code>ir</code> , from another <code>iterator_range</code> , <code>ir2</code> .

Example

```
char const* s = "Ruby";
typedef vector<int, char, double, char const*> vector_type;
vector_type vec(1, 'x', 3.3, s);

typedef result_of::begin<vector_type>::type A;
typedef result_of::end<vector_type>::type B;
typedef result_of::next<A>::type C;
typedef result_of::prior<B>::type D;

C c(vec);
D d(vec);

iterator_range<C, D> range(c, d);
std::cout << range << std::endl;
```

joint_view

Description

`joint_view` presents a view which is a concatenation of two sequences.

Header

```
#include <boost/fusion/view/joint_view.hpp>
#include <boost/fusion/include/joint_view.hpp>
```

Synopsis

```
template <typename Sequence1, typename Sequence2>
struct joint_view;
```

Template parameters

Parameter	Description	Default
Sequence1	A Forward Sequence	
Sequence2	A Forward Sequence	

Model of

- [Forward Sequence](#)
- [Associative Sequence](#) if Sequence1 and Sequence2 implement the [Associative Sequence](#) model.

Notation

JV	A <code>joint_view</code> type
s1	An instance of Sequence1
s2	An instance of Sequence2
jv, jv2	Instances of <code>joint_view</code>

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in the implemented models.

Expression	Semantics
JV(s1, s2)	Creates a <code>joint_view</code> given sequences, s1 and s2.
JV(jv)	Copy constructs a <code>joint_view</code> from another <code>joint_view</code> , jv.
jv = jv2	Assigns to a <code>joint_view</code> , jv, from another <code>joint_view</code> , jv2.

Example

```
vector<int, char> v1(3, 'x');
vector<std::string, int> v2("hello", 123);
joint_view<
    vector<int, char>
    , vector<std::string, int>
> view(v1, v2);
std::cout << view << std::endl;
```

zip_view

Description

`zip_view` presents a view which iterates over a collection of [Sequence\(s\)](#) in parallel. A `zip_view` is constructed from a [Sequence](#) of references to the component `_sequence_s`.

Header

```
#include <boost/fusion/view/zip_view.hpp>
#include <boost/fusion/include/zip_view.hpp>
```

Synopsis

```
template <typename Sequences>
struct zip_view;
```

Template parameters

Parameter	Description	Default
Sequences	A Forward Sequence of references to other Fusion <code>_sequence_s</code>	

Model of

- [Forward Sequence](#), [Bidirectional Sequence](#) or [Random Access Sequence](#) depending on the traversal characteristics (see [Sequence Traversal Concept](#)) of its underlying sequence.

Notation

<code>zV</code>	A <code>zip_view</code> type
<code>s</code>	An instance of Sequences
<code>zv1, zv2</code>	Instances of <code>zV</code>

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in [Forward Sequence](#).

Expression	Semantics
<code>ZV(s)</code>	Creates a <code>zip_view</code> given a sequence of references to the component <code>_sequence_s</code> .
<code>ZV(zv1)</code>	Copy constructs a <code>zip_view</code> from another <code>zip_view</code> , <code>zv</code> .
<code>zv1 = zv2</code>	Assigns to a <code>zip_view</code> , <code>zv</code> , from another <code>zip_view</code> , <code>zv2</code> .

Example

```
typedef vector<int,int> vec1;
typedef vector<char,char> vec2;
vec1 v1(1,2);
vec2 v2('a','b');
typedef vector<vec1&, vec2&> sequences;
std::cout << zip_view<sequences>(sequences(v1, v2)) << std::endl; // ((1 a) (2 b))
```

transform_view

The unary version of `transform_view` presents a view of its underlying sequence given a unary function object or function pointer. The binary version of `transform_view` presents a view of 2 underlying sequences, given a binary function object or function pointer. The `transform_view` inherits the traversal characteristics (see [Sequence Traversal Concept](#)) of its underlying sequence or sequences.

Header

```
#include <boost/fusion/view/transform_view.hpp>
#include <boost/fusion/include/transform_view.hpp>
```

Synopsis

Unary Version

```
template <typename Sequence, typename F1>
struct transform_view;
```

Binary Version

```
template <typename Sequence1, typename Sequence2, typename F2>
struct transform_view;
```

Template parameters

Parameter	Description	Default
Sequence	A Forward Sequence	
Sequence1	A Forward Sequence	
Sequence2	A Forward Sequence	
F1	A unary function object or function pointer. <code>boost::result_of<F1(E)>::type</code> is the return type of an instance of F1 when called with a value of each element type E in the input sequence.	
F2	A binary function object or function pointer. <code>boost::result_of<F2(E1, E2)>::type</code> is the return type of an instance of F2 when called with a value of each corresponding pair of element type E1 and E2 in the input sequences.	

Model of

- [Forward Sequence](#), [Bidirectional Sequence](#) or [Random Access Sequence](#) depending on the traversal characteristics (see [Sequence Traversal Concept](#)) of its underlying sequence.

Notation

TV	A <code>transform_view</code> type
BTv	A binary <code>transform_view</code> type
UTv	A unary <code>transform_view</code> type
f1	An instance of F1
f2	An instance of F2
s	An instance of Sequence
s1	An instance of Sequence1
s2	An instance of Sequence2
tv, tv2	Instances of <code>transform_view</code>

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in [Forward Sequence](#), [Bidirectional Sequence](#) or [Random Access Sequence](#) depending on the traversal characteristics (see [Sequence Traversal Concept](#)) of its underlying sequence or sequences.

Expression	Semantics
UTV(<i>s</i> , <i>f1</i>)	Creates a unary transform_view given sequence, <i>s</i> and unary function object or function pointer, <i>f1</i> .
BTV(<i>s1</i> , <i>s2</i> , <i>f2</i>)	Creates a binary transform_view given sequences, <i>s1</i> and <i>s2</i> and binary function object or function pointer, <i>f2</i> .
TV(<i>tv</i>)	Copy constructs a transform_view from another transform_view, <i>tv</i> .
<i>tv</i> = <i>tv2</i>	Assigns to a transform_view, <i>tv</i> , from another transform_view, <i>tv2</i> .

Example

```
struct square
{
    template<typename Sig>
    struct result;

    template<typename U>
    struct result<square(U)>
    : remove_reference<U>
    {};

    template <typename T>
    T operator()(T x) const
    {
        return x * x;
    }
};

typedef vector<int, short, double> vector_type;
vector_type vec(2, 5, 3.3);

transform_view<vector_type, square> transform(vec, square());
std::cout << transform << std::endl;
```

reverse_view

reverse_view presents a reversed view of underlying sequence. The first element will be its last and the last element will be its first.

Header

```
#include <boost/fusion/view/reverse_view.hpp>
#include <boost/fusion/include/reverse_view.hpp>
```

Synopsis

```
template <typename Sequence>
struct reverse_view;
```

Template parameters

Parameter	Description	Default
Sequence	A Bidirectional Sequence	

Model of

- A model of [Bidirectional Sequence](#) if Sequence is a [Bidirectional Sequence](#) else, [Random Access Sequence](#) if Sequence is a [Random Access Sequence](#).
- [Associative Sequence](#) if Sequence implements the [Associative Sequence](#) model.

Notation

RV	A reverse_view type
s	An instance of Sequence
rv, rv2	Instances of reverse_view

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in the implemented models.

Expression	Semantics
RV(s)	Creates a unary reverse_view given sequence, s.
RV(rv)	Copy constructs a reverse_view from another reverse_view, rv.
rv = rv2	Assigns to a reverse_view, rv, from another reverse_view, rv2.

Example

```
typedef vector<int, short, double> vector_type;
vector_type vec(2, 5, 3.3);

reverse_view<vector_type> reverse(vec);
std::cout << reverse << std::endl;
```

nview

Description

nview presents a view which iterates over a given [Sequence](#) in a specified order. An nview is constructed from an arbitrary [Sequence](#) and a list of indices specifying the elements to iterate over.

Header

```
#include <boost/fusion/view/nview.hpp>
#include <boost/fusion/include/nview.hpp>
```

Synopsis

```
template <typename Sequence, typename Indices>
struct nview;

template <typename Sequence, int I1, int I2 = -1, ...>
typename result_of::nview<Sequence, I1, I2, ...>::type
as_nview(Sequence& s);
```

Template parameters

Parameter	Description	Default
Sequence	An arbitrary Fusion Forward Sequence	
Indices	A <code>mpl::vector_c<int, ...></code> holding the indices defining the required iteration order.	
I1, I2, I3...	A list of integers specifying the required iteration order.	INT_MAX for I2, I3...

Model of

- [Random Access Sequence](#) (see [Sequence Traversal Concept](#))

Notation

NV	A <code>nview</code> type
s	An instance of Sequences
nv1, nv2	Instances of NV

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in [Random Access Sequence](#).

Expression	Semantics
<code>NV(s)</code>	Creates an <code>nview</code> given a sequence and a list of indices.
<code>NV(nv1)</code>	Copy constructs an <code>nview</code> from another <code>nview</code> , <code>nv1</code> .
<code>nv1 = nv2</code>	Assigns to an <code>nview</code> , <code>nv1</code> , from another <code>nview</code> , <code>nv2</code> .

The `nview` internally stores a Fusion [vector](#) of references to the elements of the original Fusion [Sequence](#)

Example

```
typedef vector<int, char, double> vec;
typedef mpl::vector_c<int, 2, 1, 0, 2, 0> indicies;

vec v1(1, 'c', 2.0);

std::cout << nview<vec, indicies>(v1) << std::endl; // (2.0 c 1 2.0 1)
std::cout << as_nview<2, 1, 1, 0>(v1) << std::endl; // (2.0 c c 1)
```

Adapted

Fusion provides a couple of adapters for other sequences such as `std::pair`, [MPL](#) sequences, and `boost::array`. These adapters are written using Fusion's non-intrusive [Extension](#) mechanism. If you wish to use these sequences with fusion, simply include the necessary files and they will be regarded as first-class, fully conforming fusion sequences.

Fusion also provides various schemes to make it easy for the user to adapt various data structures, non-intrusively, as full fledged Fusion sequences.

Header

```
#include <boost/fusion/adapted.hpp>
#include <boost/fusion/include/adapted.hpp>
```

Fusion sequences may also be adapted as fully conforming [MPL](#) sequences (see [Intrinsics](#)). That way, we can have 2-way adaptation to and from [MPL](#) and Fusion. To make Fusion sequences fully conforming [MPL](#) sequences, include:

```
#include <boost/fusion/mpl.hpp>
```

If you want bi-directional adaptation to and from [MPL](#) and Fusion, simply include:

```
#include <boost/fusion/include/mpl.hpp>
```

The header includes all the necessary headers.

std::pair

This module provides adapters for `std::pair`. Including the module header makes `std::pair` a fully conforming [Random Access Sequence](#).

Header

```
#include <boost/fusion/adapted/std_pair.hpp>
#include <boost/fusion/include/std_pair.hpp>
```

Model of

- [Random Access Sequence](#)

Example

```
std::pair<int, std::string> p(123, "Hola!!!");
std::cout << at_c<0>(p) << std::endl;
std::cout << at_c<1>(p) << std::endl;
std::cout << p << std::endl;
```

See also

`std::pair`, `TR1` and `std::pair`

mpl sequence

This module provides adapters for [MPL](#) sequences. Including the module header makes all [MPL](#) sequences fully conforming fusion sequences.

Header

```
#include <boost/fusion/adapted/mpl.hpp>
#include <boost/fusion/include/mpl.hpp>
```

Model of

- [Forward Sequence](#) (If the [MPL](#) sequence is a forward sequence.)
- [Bidirectional Sequence](#) (If the [MPL](#) sequence is a bidirectional sequence.)
- [Random Access Sequence](#) (If the [MPL](#) sequence is a random access sequence.)

Example

```
mpl::vector_c<int, 123, 456> vec_c;
fusion::vector2<int, long> v(vec_c);
std::cout << at_c<0>(v) << std::endl;
std::cout << at_c<1>(v) << std::endl;

v = mpl::vector_c<int, 456, 789>();
std::cout << at_c<0>(v) << std::endl;
std::cout << at_c<1>(v) << std::endl;
```

See also

[MPL](#)

boost::array

This module provides adapters for `boost::array`. Including the module header makes `boost::array` a fully conforming [Random Access Sequence](#).

Header

```
#include <boost/fusion/adapted/array.hpp>
#include <boost/fusion/include/array.hpp>
```

Model of

- [Random Access Sequence](#)

Example

```
boost::array<int,3> arr = {{1,2,3}};

std::cout << *begin(arr) << std::endl;
std::cout << *next(begin(arr)) << std::endl;
std::cout << *advance_c<2>(begin(arr)) << std::endl;
std::cout << *prior(end(arr)) << std::endl;
std::cout << at_c<2>(arr) << std::endl;
```

See also

[Boost.Array Library](#)

boost::tuple

This module provides adapters for `boost::tuple`. Including the module header makes `boost::tuple` a fully conforming [Forward Sequence](#).

Header

```
#include <boost/fusion/adapted/boost_tuple.hpp>
#include <boost/fusion/include/boost_tuple.hpp>
```

Model of

- [Forward Sequence](#)

Example

```
boost::tuple<int,std::string> example_tuple(101, "hello");
std::cout << *boost::fusion::begin(example_tuple) << '\n';
std::cout << *boost::fusion::next(boost::fusion::begin(example_tuple)) << '\n';
```

See also

[Boost.Tuple Library](#)

BOOST_FUSION_ADAPT_STRUCT

Description

`BOOST_FUSION_ADAPT_STRUCT` is a macro that can be used to generate all the necessary boilerplate to make an arbitrary struct into a [Random Access Sequence](#).

Synopsis

```
BOOST_FUSION_ADAPT_STRUCT(  
    struct_name,  
    (member_type0, member_name0)  
    (member_type1, member_name1)  
    ...  
)
```

Semantics

The above macro generates the necessary code to adapt `struct_name` as a model of [Random Access Sequence](#). The sequence of `(member_typeN, member_nameN)` pairs declare the type and names of each of the struct members that will be part of the sequence.

The macro should be used at global scope, and `struct_name` should be the fully namespace qualified name of the struct to be converted.

Header

```
#include <boost/fusion/adapted/struct/adapt_struct.hpp>  
#include <boost/fusion/include/adapt_struct.hpp>
```

Example

```
namespace demo  
{  
    struct employee  
    {  
        std::string name;  
        int age;  
    };  
}  
  
// demo::employee is now a Fusion sequence  
BOOST_FUSION_ADAPT_STRUCT(  
    demo::employee,  
    (std::string, name)  
    (int, age))
```

BOOST_FUSION_ADAPT_ASSOC_STRUCT

Description

`BOOST_FUSION_ADAPT_ASSOC_STRUCT` is a macro that can be used to generate all the necessary boilerplate to make an arbitrary struct into a model of [Random Access Sequence](#) and [Associative Sequence](#).

Synopsis

```
BOOST_FUSION_ADAPT_ASSOC_STRUCT(
    struct_name,
    (member_type0, member_name0, key_type0)
    (member_type1, member_name1, key_type1)
    ...
)
```

Semantics

The above macro generates the necessary code to adapt `struct_name` as a model of [Random Access Sequence](#) and [Associative Sequence](#). The sequence of `(member_typeN, member_nameN, key_typeN)` triples declare the type, name and key type of each of the struct members that will be part of the sequence.

The macro should be used at global scope, and `struct_name` should be the fully namespace qualified name of the struct to be converted.

Header

```
#include <boost/fusion/adapted/struct/adapt_assoc_struct.hpp>
#include <boost/fusion/include/adapt_assoc_struct.hpp>
```

Example

```
namespace demo
{
    struct employee
    {
        std::string name;
        int age;
    };
}

namespace keys
{
    struct name;
    struct age;
}

// demo::employee is now a Fusion sequence
// It is also an associative sequence with
// keys keys::name and keys::age present.
BOOST_FUSION_ADAPT_ASSOC_STRUCT(
    demo::employee,
    (std::string, name, keys::name)
    (int, age, keys::age))
```

Algorithm

Lazy Evaluation

Unlike [MPL](#), Fusion algorithms are lazy and non sequence-type preserving. What does that mean? It means that when you operate on a sequence through a Fusion algorithm that returns a sequence, the sequence returned may not be of the same class as the original. This is by design. Runtime efficiency is given a high priority. Like [MPL](#), and unlike [STL](#), fusion algorithms are functional in nature such that algorithms are non mutating (no side effects). However, due to the high cost of returning full sequences such as vectors and lists, *Views* are returned from Fusion algorithms instead. For example, the [transform](#) algorithm does not actually return a

transformed version of the original sequence. `transform` returns a `transform_view`. This view holds a reference to the original sequence plus the transform function. Iteration over the `transform_view` will apply the transform function over the sequence elements on demand. This *lazy* evaluation scheme allows us to chain as many algorithms as we want without incurring a high runtime penalty.

Sequence Extension

The *lazy* evaluation scheme where [Algorithms](#) return [Views](#) also allows operations such as `push_back` to be totally generic. In Fusion, `push_back` is actually a generic algorithm that works on all sequences. Given an input sequence `s` and a value `x`, Fusion's `push_back` algorithm simply returns a `joint_view`: a view that holds a reference to the original sequence `s` and the value `x`. Functions that were once sequence specific and need to be implemented N times over N different sequences are now implemented only once. That is to say that Fusion sequences are cheaply extensible. However, an important caveat is that the result of a sequence extending operation like `push_back` does not retain the properties of the original sequence such as associativity of `set(s)`. To regain the original sequence, [Conversion](#) functions are provided. You may use one of the [Conversion](#) functions to convert back to the original sequence type.

Header

```
#include <boost/fusion/algorithm.hpp>
#include <boost/fusion/include/algorithm.hpp>
```

Iteration

The iteration algorithms provide the fundamental algorithms for traversing a sequence repeatedly applying an operation to its elements.

Header

```
#include <boost/fusion/algorithm/iteration.hpp>
#include <boost/fusion/include/iteration.hpp>
```

Functions

fold

Description

For a sequence `seq`, initial state `initial_state`, and binary function object or function pointer `f`, `fold` returns the result of the repeated application of binary `f` to the result of the previous `f` invocation (`initial_state` if it is the first call) and each element of `seq`.

Synopsis

```
template<
    typename Sequence,
    typename State,
    typename F
>
typename result_of::fold<Sequence, State, F>::type fold(
    Sequence& seq, State const& initial_state, F const& f);
```

Table 37. Parameters

Parameter	Requirement	Description
seq	A model of Forward Sequence , $f(s, e)$ must be a valid expression for current state s , and each element e in seq	Operation's argument
initial_state	Any type	Initial state
f	<code>boost::result_of<F(S, E)>::type</code> is the return type of $f(s, e)$ current state s of type S , and for each element e of type E in seq	Operation's argument

Expression Semantics

```
fold(seq, initial_state, f);
```

Return type: Any type

Semantics: Equivalent to $f(\dots f(f(\text{initial_state}, e_1), e_2) \dots e_N)$ where $e_1 \dots e_N$ are the elements of seq .

Complexity

Linear, exactly `result_of::size<Sequence>::value` applications of f .

Header

```
#include <boost/fusion/algorithm/iteration/fold.hpp>
#include <boost/fusion/include/fold.hpp>
```

Example

```
struct make_string
{
    typedef std::string result_type;

    template<typename T>
    std::string operator()(const std::string& str, const T& t) const
    {
        return str + boost::lexical_cast<std::string>(t);
    }
};

...
const vector<int,int> vec(1,2);
assert(fold(vec,std::string(""), make_string()) == "12");
```

accumulate

Description

For a sequence `seq`, initial state `initial_state`, and binary function object or function pointer `f`, `accumulate` returns the result of the repeated application of binary `f` to the result of the previous `f` invocation (`initial_state` if it is the first call) and each element of `seq`.

Synopsis

```
template<
    typename Sequence,
    typename State,
    typename F
>
typename result_of::accumulate<Sequence, State, F>::type accumulate(
    Sequence& seq, State const& initial_state, F const& f);
```

Table 38. Parameters

Parameter	Requirement	Description
<code>seq</code>	A model of Forward Sequence , <code>f(s,e)</code> must be a valid expression for current state <code>s</code> , and each element <code>e</code> in <code>seq</code>	Operation's argument
<code>initial_state</code>	Any type	Initial state
<code>f</code>	<code>boost::result_of<F(S,E)>::type</code> is the return type of <code>f(s,e)</code> current state <code>s</code> of type <code>S</code> , and for each element <code>e</code> of type <code>E</code> in <code>seq</code>	Operation's argument

Expression Semantics

```
accumulate(seq, initial_state, f);
```

Return type: Any type

Semantics: Equivalent to `f(... f(f(initial_state,e1),e2) ...eN)` where `e1 ...eN` are the elements of `seq`.

Complexity

Linear, exactly `result_of::size<Sequence>::value` applications of `f`.

Header

```
#include <boost/fusion/algorithm/iteration/accumulate.hpp>
#include <boost/fusion/include/accumulate.hpp>
```

Example

```
struct make_string
{
    typedef std::string result_type;

    template<typename T>
    std::string operator()(const std::string& str, const T& t) const
    {
        return str + boost::lexical_cast<std::string>(t);
    }
};

...
const vector<int,int> vec(1,2);
assert(accumulate(vec,std::string(""), make_string()) == "12");
```

for_each

Description

Applies a unary function object to each element of a sequence.

Synopsis

```
template<
    typename Sequence,
    typename F
>
typename result_of::for_each<Sequence, F>::type for_each(
    Sequence& seq, F const& f);
```

Table 39. Parameters

Parameter	Requirement	Description
<code>seq</code>	A model of Forward Sequence , <code>f(e)</code> must be a valid expression for each element <code>e</code> in <code>seq</code>	Operation's argument
<code>f</code>	A unary Regular Callable Object	Operation's argument

Expression Semantics

```
for_each(seq, f);
```

Return type: `void`

Semantics: Calls `f(e)` for each element `e` in `seq`.

Complexity

Linear, exactly `result_of::size<Sequence>::value` applications of `f`.

Header

```
#include <boost/fusion/algorithm/iteration/for_each.hpp>
#include <boost/fusion/include/for_each.hpp>
```

Example

```
struct increment
{
    template<typename T>
    void operator()(T& t) const
    {
        ++t;
    }
};

...
vector<int,int> vec(1,2);
for_each(vec, increment());
assert(vec == make_vector(2,3));
```

Metafunctions

fold

Description

Returns the result type of `fold`.

Synopsis

```
template<
    typename Sequence,
    typename State,
    typename F>
struct fold
{
    typedef unspecified type;
};
```

Table 40. Parameters

Parameter	Requirement	Description
Sequence	A model of Forward Sequence	The sequence to iterate
State	Any type	The initial state for the first application of <code>F</code>
<code>F</code>	<code>boost::result_of<F(S,E)>::type</code> is the return type of <code>f(s,e)</code> for current state <code>s</code> of type <code>S</code> , and for each element <code>e</code> of type <code>E</code> in <code>seq</code>	The operation to be applied on forward traversal

Expression Semantics

```
result_of::fold<Sequence, State, F>::type
```

Return type: Any type

Semantics: Returns the result of applying `fold` to a sequence of type `Sequence`, with an initial state of type `State` and binary function object or function pointer of type `F`.

Complexity

Linear, exactly `result_of::size<Sequence>::value` applications of `F`.

Header

```
#include <boost/fusion/algorithm/iteration/fold.hpp>
#include <boost/fusion/include/fold.hpp>
```

accumulate

Description

Returns the result type of `accumulate`.

Synopsis

```
template<
    typename Sequence,
    typename State,
    typename F>
struct accumulate
{
    typedef unspecified type;
};
```

Table 41. Parameters

Parameter	Requirement	Description
Sequence	A model of Forward Sequence	The sequence to iterate
State	Any type	The initial state for the first application of <code>F</code>
<code>F</code>	<code>boost::result_of<F(S, E)>::type</code> is the return type of <code>f(s, e)</code> for current state <code>s</code> of type <code>S</code> , and for each element <code>e</code> of type <code>E</code> in <code>seq</code>	The operation to be applied on forward traversal

Expression Semantics

```
result_of::accumulate<Sequence, State, F>::type
```

Return type: Any type

Semantics: Returns the result of applying `accumulate` to a sequence of type `Sequence`, with an initial state of type `State` and binary function object or function pointer of type `F`.

Complexity

Linear, exactly `result_of::size<Sequence>::value` applications of `F`.

Header

```
#include <boost/fusion/algorithm/iteration/accumulate.hpp>
#include <boost/fusion/include/accumulate.hpp>
```

for_each

A metafunction returning the result type of applying `for_each` to a sequence. The return type of `for_each` is always `void`.

Description

Synopsis

```
template<
    typename Sequence,
    typename F
>
struct for_each
{
    typedef void type;
};
```

Table 42. Parameters

Parameter	Requirement	Description
Sequence	A model of Forward Sequence	Operation's argument
F	Any type	Operation's argument

Expression Semantics

```
result_of::for_each<Sequence, F>::type
```

Return type: `void`.

Semantics: Returns the return type of `for_each` for a sequence of type `Sequence` and a unary function object `F`. The return type is always `void`.

Complexity

Constant.

Header

```
#include <boost/fusion/algorithm/iteration/for_each.hpp>
#include <boost/fusion/include/for_each.hpp>
```

Query

The query algorithms provide support for searching and analyzing sequences.

Header

```
#include <boost/fusion/algorithm/query.hpp>
#include <boost/fusion/include/query.hpp>
```

Functions

any

Description

For a sequence `seq` and unary function object `f`, `any` returns true if `f` returns true for at least one element of `seq`.

Synopsis

```
template<
    typename Sequence,
    typename F
>
typename result_of::any<Sequence,F>::type any(
    Sequence const& seq, F f);
```

Table 43. Parameters

Parameter	Requirement	Description
<code>seq</code>	A model of Forward Sequence , <code>f(e)</code> must be a valid expression, convertible to <code>bool</code> , for each element <code>e</code> in <code>seq</code>	The sequence to search
<code>f</code>	A unary function object	The search predicate

Expression semantics

```
any(seq, f);
```

Return type: `bool`

Semantics: Returns true if and only if `f(e)` evaluates to true for some element `e` in `seq`.

Complexity

Linear. At most `result_of::size<Sequence>::value` comparisons.

Header

```
#include <boost/fusion/algorithm/query/any.hpp>
#include <boost/fusion/include/any.hpp>
```

Example

```
struct odd
{
    template<typename T>
    bool operator()(T t) const
    {
        return t % 2;
    }
};

...
assert(any(make_vector(1,2), odd()));
assert(!any(make_vector(2,4), odd()));
```

all

Description

For a sequence `seq` and unary function object `f`, `all` returns true if `f` returns true for every element of `seq`.

Synopsis

```
template<
    typename Sequence,
    typename F
>
typename result_of::all<Sequence,F>::type all(
    Sequence const& seq, F f);
```

Table 44. Parameters

Parameter	Requirement	Description
<code>seq</code>	A model of Forward Sequence , <code>f(e)</code> is a valid expression, convertible to <code>bool</code> , for every element <code>e</code> in <code>seq</code>	The sequence to search
<code>f</code>	A unary function object	The search predicate

Expression Semantics

```
all(seq, f);
```

Return type: `bool`

Semantics: Returns true if and only if `f(e)` evaluates to true for every element `e` in `seq`.

Complexity

Linear. At most `result_of::size<Sequence>::value` comparisons.

Header

```
#include <boost/fusion/algorithm/query/all.hpp>
#include <boost/fusion/include/all.hpp>
```

Example

```
struct odd
{
    template<typename T>
    bool operator()(T t) const
    {
        return t % 2;
    }
};

...
assert(all(make_vector(1,3), odd()));
assert(!all(make_vector(1,2), odd()));
```

none

Description

For a sequence `seq` and unary function object `f`, `none` returns true if `f` returns false for every element of `seq`.

Synopsis

```
template<
    typename Sequence,
    typename F
>
typename result_of::none<Sequence,F>::type none(
    Sequence const& seq, F f);
```

Table 45. Parameters

Parameter	Requirement	Description
<code>seq</code>	A model of Forward Sequence , <code>f(e)</code> is a valid expression, convertible to <code>bool</code> , for every element <code>e</code> in <code>seq</code>	The sequence to search
<code>f</code>	A unary function object	The search predicate

Expression Semantics

```
none(seq, f);
```

Return type: `bool`

Semantics: Returns true if and only if `f(e)` evaluates to false for every element `e` in `seq`. Result equivalent to `!any(seq, f)`.

Complexity

Linear. At most `result_of::size<Sequence>::value` comparisons.

Header

```
#include <boost/fusion/algorithm/query/none.hpp>
#include <boost/fusion/include/none.hpp>
```

Example

```
struct odd
{
    template<typename T>
    bool operator()(T t) const
    {
        return t % 2;
    }
};

...
assert(none(make_vector(2,4), odd()));
assert(!none(make_vector(1,2), odd()));
```

find

Description

Finds the first element of a given type within a sequence.

Synopsis

```
template<
    typename T,
    typename Sequence
>
unspecified find(Sequence const& seq);

template<
    typename T,
    typename Sequence
>
unspecified find(Sequence& seq);
```

Table 46. Parameters

Parameter	Requirement	Description
seq	A model of Forward Sequence	The sequence to search
T	Any type	The type to search for

Expression Semantics

```
find<T>(seq)
```

Return type: A model of the same iterator category as the iterators of `seq`.

Semantics: Returns an iterator to the first element of `seq` of type `T`, or `end(seq)` if there is no such element. Equivalent to `find_if<boost::is_same<_, T>>(seq)`

Complexity

Linear. At most `result_of::size<Sequence>::value` comparisons.

Header

```
#include <boost/fusion/algorithm/query/find.hpp>
#include <boost/fusion/include/find.hpp>
```

Example

```
const vector<char,int> vec('a','0');
assert(*find<int>(vec) == '0');
assert(find<double>(vec) == end(vec));
```

find_if

Finds the first element within a sequence with a type for which a given [MPL Lambda Expression](#) evaluates to `boost::mpl::true_`.

Description

Synopsis

```
template<
    typename F,
    typename Sequence
>
unspecified find_if(Sequence const& seq);

template<
    typename F,
    typename Sequence
>
unspecified find_if(Sequence& seq);
```

Table 47. Parameters

Parameter	Requirement	Description
<code>seq</code>	A model of Forward Sequence	The sequence to search
<code>F</code>	A unary MPL Lambda Expression	The search predicate

Expression Semantics

```
find_if<F>(seq)
```

Return type: An iterator of the same iterator category as the iterators of `seq`.

Semantics: Returns the first element of `seq` for which [MPL Lambda Expression](#) `F` evaluates to `boost::mpl::true_`, or `end(seq)` if there is no such element.

Complexity

Linear. At most `result_of::size<Sequence>::value` comparisons.

1. include `<boost/fusion/algorithm/query/find_if.hpp>`

2. include <boost/fusion/include/find_if.hpp>

Example

```
const vector<double,int> vec(1.0,2);
assert(*find_if<is_integral<mpl::_> >(vec) == 2);
assert(find_if<is_class<mpl::_> >(vec) == end(vec));
```

count

Description

Returns the number of elements of a given type within a sequence.

Synopsis

```
template<
    typename Sequence,
    typename T
>
typename result_of::count<Sequence, T>::type count(
    Sequence const& seq, T const& t);
```

Table 48. Parameters

Parameter	Requirement	Description
seq	A model of Forward Sequence , <code>e == t</code> must be a valid expression, convertible to <code>bool</code> , for each element <code>e</code> in <code>seq</code>	The sequence to search
T	Any type	The type to count

Expression Semantics

```
count(seq, t);
```

Return type: `int`

Semantics: Returns the number of elements of type `T` and equal to `t` in `seq`.

Complexity

Linear. At most `result_of::size<Sequence>::value` comparisons.

Header

```
#include <boost/fusion/algorithm/query/count.hpp>
#include <boost/fusion/include/count.hpp>
```

Example

```
const vector<double,int,int> vec(1.0,2,3);
assert(count(vec,2) == 1);
```

count_if

Description

Returns the number of elements within a sequence with a type for which a given unary function object evaluates to true.

Synopsis

```
template<
    typename Sequence,
    typename F
>
typename result_of::count_if<Sequence, F>::type count_if(
    Sequence const& seq, F f);
```

Table 49. Parameters

Parameter	Requirement	Description
seq	A model of Forward Sequence , $f(e)$ is a valid expression, convertible to <code>bool</code> , for each element e in <code>seq</code>	The sequence to search
f	A unary function object	The search predicate

Expression Semantics

```
count_if(seq, f)
```

Return type: `int`

Semantics: Returns the number of elements in `seq` where `f` evaluates to true.

Complexity

Linear. At most `result_of::size<Sequence>::value` comparisons.

Header

```
#include <boost/fusion/algorithm/query/count_if.hpp>
#include <boost/fusion/include/count_if.hpp>
```

Example

```
const vector<int,int,int> vec(1,2,3);
assert(count_if(vec,odd()) == 2);
```

Metafunctions

any

Description

A metafunction returning the result type of [any](#).

Synopsis

```
template<
    typename Sequence,
    typename F
>
struct any
{
    typedef bool type;
};
```

Table 50. Parameters

Parameter	Requirement	Description
Sequence	A model of Forward Sequence	Operation's argument
F	A model of unary Polymorphic Function Object	Operation's argument

Expression Semantics

```
result_of::any<Sequence, F>::type
```

Return type: `bool`.

Semantics: Returns the return type of [any](#) given a sequence of type `Sequence` and a unary [Polymorphic Function Object](#) of type `F`. The return type is always `bool`.

Complexity

Constant.

Header

```
#include <boost/fusion/algorithm/query/any.hpp>
#include <boost/fusion/include/any.hpp>
```

all

Description

A metafunction returning the result type of [all](#).

Synopsis

```
template<
    typename Sequence,
    typename F
>
struct all
{
    typedef bool type;
};
```

Table 51. Parameters

Parameter	Requirement	Description
Sequence	A model of Forward Sequence	Operation's argument
F	A model of unary Polymorphic Function Object	Operation's argument

Expression Semantics

```
result_of::all<Sequence, F>::type
```

Return type: `bool`.

Semantics: Returns the return type of [all](#) given a sequence of type `Sequence` and a unary [Polymorphic Function Object](#) of type `F`. The return type is always `bool`.

Complexity

Constant.

Header

```
#include <boost/fusion/algorithm/query/all.hpp>
#include <boost/fusion/include/all.hpp>
```

none

Description

A metafunction returning the result type of [none](#).

Synopsis

```
template<
    typename Sequence,
    typename F
>
struct none
{
    typedef bool type;
};
```

Table 52. Parameters

Parameter	Requirement	Description
Sequence	A model of Forward Sequence	Operation's argument
F	A model of unary Polymorphic Function Object	Operation's argument

Expression Semantics

```
result_of::none<Sequence, F>::type
```

Return type: `bool`.

Semantics: Returns the return type of `none` given a sequence of type `Sequence` and a unary [Polymorphic Function Object](#) of type `F`. The return type is always `bool`.

Complexity

Constant.

Header

```
#include <boost/fusion/algorithm/query/none.hpp>
#include <boost/fusion/include/none.hpp>
```

find

Description

Returns the result type of `find`, given the sequence and search types.

Synopsis

```
template<
    typename Sequence,
    typename T
>
struct find
{
    typedef unspecified type;
};
```

Table 53. Parameters

Parameter	Requirement	Description
Sequence	Model of Forward Sequence	Operation's argument
T	Any type	Operation's argument

Expression Semantics

```
result_of::find<Sequence, T>::type
```

Return type: A model of the same iterator category as the iterators of Sequence.

Semantics: Returns an iterator to the first element of type T in Sequence, or `result_of::end<Sequence>::type` if there is no such element.

Complexity

Linear, at most `result_of::size<Sequence>::value` comparisons.

Header

```
#include <boost/fusion/algorithm/query/find.hpp>
#include <boost/fusion/include/find.hpp>
```

find_if

Description

Returns the result type of `find_if` given the sequence and predicate types.

Synopsis

```
template<
    typename Sequence,
    typename Pred
>
struct find_if
{
    typedef unspecified type;
};
```

Table 54. Parameters

Parameter	Requirement	Description
Sequence	A model of Forward Sequence	Operation's argument
Pred	A model of MPL Lambda Expression	Operation's arguments

Expression Semantics

```
result_of::find_if<Sequence, Pred>::type
```

Return type: A model of the same iterator category as the iterators of Sequence.

Semantics: Returns an iterator to the first element in Sequence for which Pred evaluates to true. Returns `result_of::end<Sequence>::type` if there is no such element.

Complexity

Linear. At most `result_of::size<Sequence>::value` comparisons.

Header

```
#include <boost/fusion/algorithm/query/find_if.hpp>
#include <boost/fusion/include/find_if.hpp>
```

count

Description

A metafunction that returns the result type of `count` given the sequence and search types.

Synopsis

```
template<
    typename Sequence,
    typename T
>
struct count
{
    typedef int type;
};
```

Table 55. Parameters

Parameter	Requirement	heading Description
Sequence	A model of Forward Sequence	Operation's argument
T	Any type	Operation's argument

Expression Semantics

```
result_of::count<T>::type
```

Return type: `int`.

Semantics: Returns the return type of `count`. The return type is always `int`.

Complexity

Constant.

Header

```
#include <boost/fusion/algorithm/query/count.hpp>
#include <boost/fusion/include/count.hpp>
```

count_if

Description

A metafunction that returns the result type of `count_if` given the sequence and predicate types.

Synopsis

```
template<
    typename Sequence,
    typename Pred
>
struct count_if
{
    typedef int type;
};
```

Table 56. Parameters

Parameter	Requirement	Description
Sequence	A model of Forward Sequence	Operation's argument
Pred	A unary function object	Operation's argument

Expression Semantics

```
result_of::count_if<Sequence, Pred>::type
```

Return type: int.

Semantics: Returns the return type of `count_if`. The return type is always int.

Complexity

Constant.

Header

```
#include <boost/fusion/algorithm/query/count_if.hpp>
#include <boost/fusion/include/count_if.hpp>
```

Transformation

The transformation algorithms create new sequences out of existing sequences by performing some sort of transformation. In reality the new sequences are views onto the data in the original sequences.



Note

As the transformation algorithms return views onto their input arguments, it is important that the lifetime of the input arguments is greater than the period during which you wish to use the results.

Header

```
#include <boost/fusion/algorithm/transformation.hpp>
#include <boost/fusion/include/transformation.hpp>
```

Functions

filter

Description

For a given sequence, filter returns a new sequences containing only the elements of a specified type.

Synopsis

```
template<
    typename T,
    typename Sequence
>
typename result_of::filter<Sequence const, T>::type filter(Sequence const& seq);
```

Table 57. Parameters

Parameter	Requirement	Description
seq	A model of Forward Sequence	Operation's argument
T	Any type	The type to retain

Expression Semantics

```
filter<T>(seq);
```

Return type:

- A model of [Forward Sequence](#).
- A model of [Associative Sequence](#) if seq implements the [Associative Sequence](#) model.

Semantics: Returns a sequence containing all the elements of seq of type T. Equivalent to `filter_if<boost::same_type<_, T>>(seq)`.

Complexity

Constant. Returns a view which is lazily evaluated.

Header

```
#include <boost/fusion/algorithm/transformation/filter.hpp>
#include <boost/fusion/include/filter.hpp>
```

Example

```
const vector<int,int,long,long> vec(1,2,3,4);
assert(filter<int>(vec) == make_vector(1,2));
```

filter_if

Description

For a given sequence, `filter_if` returns a new sequences containing only the elements with types for which a given [MPL Lambda Expression](#) evaluates to `boost::mpl::true_`.

Synopsis

```
template<
    typename Pred,
    typename Sequence
>
typename result_of::filter_if<Sequence const, Pred>::type filter_if(Sequence const& seq);
```

Table 58. Parameters

Parameter	Requirement	Description
seq	A model of Forward Sequence	Operation's argument
Pred	A unary MPL Lambda Expression	The predicate to filter by

Expression Semantics

```
filter_if<Pred>(seq);
```

Return type:

- A model of [Forward Sequence](#).
- A model of [Associative Sequence](#) if `seq` implements the [Associative Sequence](#) model.

Semantics: Returns a sequence containing all the elements of `seq` with types for which `Pred` evaluates to `boost::mpl::true_`. The order of the retained elements is the same as in the original sequence.

Complexity

Constant. Returns a view which is lazily evaluated.

Header

```
#include <boost/fusion/algorithm/transformation/filter_if.hpp>
#include <boost/fusion/include/filter_if.hpp>
```

Example

```
const vector<int,int,double,double> vec(1,2,3.0,4.0);
assert(filter_if<is_integral<mpl::_> >(vec) == make_vector(1,2));
```

transform

Description

For a sequence `seq` and function object or function pointer `f`, `transform` returns a new sequence with elements created by applying `f(e)` to each element `e` of `seq`.

Unary version synopsis

```
template<
    typename Sequence,
    typename F
>
typename result_of::transform<Sequence const, F>::type transform(
    Sequence const& seq, F f);
```

Table 59. Parameters

Parameter	Requirement	Description
<code>seq</code>	A model of Forward Sequence	Operation's argument
<code>f</code>	<code>f(e)</code> is a valid expression for each element <code>e</code> of <code>seq</code> . <code>boost::result_of<F(E)>::type</code> is the return type of <code>f</code> when called with a value of each element type <code>E</code> .	Transformation function

Expression Semantics

```
transform(seq, f);
```

Return type: A model of [Forward Sequence](#)

Semantics: Returns a new sequence, containing the return values of `f(e)` for each element `e` within `seq`.

Binary version synopsis

```
template<
    typename Sequence1,
    typename Sequence2,
    typename F
>
typename result_of::transform<Sequence1 const, Sequence2 const, F>::type transform(
    Sequence1 const& seq1, Sequence2 const& seq2, F f);
```

Table 60. Parameters

Parameter	Requirement	Description
seq1	A model of Forward Sequence	Operation's argument
seq2	A model of Forward Sequence	Operation's argument
f	$f(e1, e2)$ is a valid expression for each pair of elements $e1$ of <code>seq1</code> and $e2$ of <code>seq2</code> . <code>boost::result_of<F(E1, E2)>::type</code> is the return type of <code>f</code> when called with elements of type <code>E1</code> and <code>E2</code>	Transformation function

Return type: A model of [Forward Sequence](#).

Semantics: Returns a new sequence, containing the return values of $f(e1, e2)$ for each pair of elements $e1$ and $e2$ within `seq1` and `seq2` respectively.

Complexity

Constant. Returns a view which is lazily evaluated.

Header

```
#include <boost/fusion/algorithm/transformation/transform.hpp>
#include <boost/fusion/include/transform.hpp>
```

Example

```
struct triple
{
    typedef int result_type;

    int operator()(int t) const
    {
        return t * 3;
    };
};

...
assert(transform(make_vector(1,2,3), triple()) == make_vector(3,6,9));
```

replace

Description

Replaces each value within a sequence of a given type and value with a new value.

Synopsis

```
template<
    typename Sequence,
    typename T
>
typename result_of::replace<Sequence const, T>::type replace(
    Sequence const& seq, T const& old_value, T const& new_value);
```

Table 61. Parameters

Parameter	Requirement	Description
seq	A model of Forward Sequence , <code>e == old_value</code> is a valid expression, convertible to <code>bool</code> , for each element <code>e</code> in <code>seq</code> with type convertible to <code>T</code>	Operation's argument
old_value	Any type	Value to replace
new_value	Any type	Replacement value

Expression Semantics

```
replace(seq, old_value, new_value);
```

Return type: A model of [Forward Sequence](#).

Semantics: Returns a new sequence with all the values of `seq` with `new_value` assigned to elements with the same type and equal to `old_value`.

Complexity

Constant. Returns a view which is lazily evaluated.

Header

```
#include <boost/fusion/algorithm/transformation/replace.hpp>
#include <boost/fusion/include/replace.hpp>
```

Example

```
assert(replace(make_vector(1,2), 2, 3) == make_vector(1,3));
```

replace_if

Description

Replaces each element of a given sequence for which an unary function object evaluates to `true` replaced with a new value.

Synopsis

```
template<
    typename Sequence,
    typename F,
    typename T>
typename result_of::replace_if<Sequence const, F, T>::type replace_if(
    Sequence const& seq, F f, T const& new_value);
```

Table 62. Parameters

Parameter	Requirement	Description
seq	A model of Forward Sequence	Operation's argument
f	A function object for which <code>f(e)</code> is a valid expression, convertible to <code>bool</code> , for each element <code>e</code> in <code>seq</code>	Operation's argument
new_value	Any type	Replacement value

Expression Semantics

```
replace_if(seq, f, new_value);
```

Return type: A model of [Forward Sequence](#).

Semantics: Returns a new sequence with all the elements of `seq`, with `new_value` assigned to each element for which `f` evaluates to `true`.

Complexity

Constant. Returns a view which is lazily evaluated.

Header

```
#include <boost/fusion/algorithm/transformation/replace_if.hpp>
#include <boost/fusion/include/replace_if.hpp>
```

Example

```
struct odd
{
    template<typename T>
    bool operator()(T t) const
    {
        return t % 2;
    }
};
...
assert(replace_if(make_vector(1,2), odd(), 3) == make_vector(3,2));
```

remove

Description

Returns a new sequence, with all the elements of the original sequence, except those of a given type.

Synopsis

```
template<
    typename T,
    typename Sequence
>
typename result_of::remove<Sequence const, T>::type replace(Sequence const& seq);
```

Table 63. Parameters

Parameter	Requirement	Description
seq	A model of Forward Sequence	Operation's argument
T	Any type	Type to remove

Expression Semantics

```
remove<T>(seq);
```

Return type:

- A model of [Forward Sequence](#).
- A model of [Associative Sequence](#) if Sequence implements the [Associative Sequence](#) model.

Semantics: Returns a new sequence, containing all the elements of seq, in their original order, except those of type T. Equivalent to `remove_if<boost::is_same<_, T>>(seq)`.

Complexity

Constant. Returns a view which is lazily evaluated.

Header

```
#include <boost/fusion/algorithm/transformation/remove.hpp>
#include <boost/fusion/include/remove.hpp>
```

Example

```
const vector<int,double> vec(1,2.0);
assert(remove<double>(vec) == make_vector(1));
```

remove_if

Description

Returns a new sequence, containing all the elements of the original except those where a given unary function object evaluates to true.

Synopsis

```
template<
    typename Pred,
    typename Sequence
>
typename result_of::remove_if<Sequence const, Pred>::type remove_if(Sequence const& seq);
```

Table 64. Parameters

Parameter	Requirement	Description
seq	A model of Forward Sequence	Operation's argument
Pred	A model of unary MPL Lambda Expression	Removal predicate

Expression Semantics

```
remove_if<Pred>(seq);
```

Return type:

- A model of [Forward Sequence](#).
- A model of [Associative Sequence](#) if seq implements the [Associative Sequence](#) model.

Semantics: Returns a new sequence, containing all the elements of seq, in their original order, except those elements with types for which Pred evaluates to `boost::mpl::true_`. Equivalent to `filter<boost::mpl::not_<Pred> >(seq)`.

Complexity

Constant. Returns a view which is lazily evaluated.

Header

```
#include <boost/fusion/algorithm/transformation/remove_if.hpp>
#include <boost/fusion/include/remove_if.hpp>
```

Example

```
const vector<int,double> vec(1,2.0);
assert(remove_if<is_floating_point<mpl::_> >(vec) == make_vector(1));
```

reverse

Description

Returns a new sequence with the elements of the original in reverse order.

Synopsis

```
template<
    typename Sequence
>
typename result_of::reverse<Sequence const>::type reverse(Sequence const& seq);
```

Table 65. Parameters

Parameter	Requirement	Description
seq	A model of Bidirectional Sequence	Operation's argument

Expression Semantics

```
reverse(seq);
```

Return type:

- A model of [Bidirectional Sequence](#) if seq is a [Bidirectional Sequence](#) else, [Random Access Sequence](#) if seq is a [Random Access Sequence](#).
- A model of [Associative Sequence](#) if seq implements the [Associative Sequence](#) model.

Semantics: Returns a new sequence containing all the elements of seq in reverse order.

Complexity

Constant. Returns a view which is lazily evaluated.

Header

```
#include <boost/fusion/algorithm/transformation/reverse.hpp>
#include <boost/fusion/include/reverse.hpp>
```

Example

```
assert(reverse(make_vector(1,2,3)) == make_vector(3,2,1));
```

clear

Description

`clear` returns an empty sequence.

Synopsis

```
template<
    typename Sequence
>
typename result_of::clear<Sequence const>::type clear(Sequence const& seq);
```

Table 66. Parameters

Parameter	Requirement	Description
seq	A model of Forward Sequence	Operation's argument

Expression Semantics

```
clear(seq);
```

Return type: A model of [Forward Sequence](#).

Expression Semantics: Returns a sequence with no elements.

Complexity

Constant.

Header

```
#include <boost/fusion/algorithm/transformation/clear.hpp>
#include <boost/fusion/include/clear.hpp>
```

Example

```
assert(clear(make_vector(1,2,3)) == make_vector());
```

erase

Description

Returns a new sequence, containing all the elements of the original except those at a specified iterator, or between two iterators.

Synopsis

```
template<
    typename Sequence,
    typename First
>
typename result_of::erase<Sequence const, First>::type erase(
    Sequence const& seq, First const& it1);

template<
    typename Sequence,
    typename First,
    typename Last
>
typename result_of::erase<Sequence const, First, Last>::type erase(
    Sequence const& seq, First const& it1, Last const& it2);
```

Table 67. Parameters

Parameters	Requirement	Description
seq	A model of Forward Sequence	Operation's argument
it1	A model of Forward Iterator	Iterator into seq
it2	A model of Forward Iterator	Iterator into seq after it1

Expression Semantics

```
erase(seq, pos);
```

Return type:

- A model of [Forward Sequence](#).
- A model of [Associative Sequence](#) if seq implements the [Associative Sequence](#) model.

Semantics: Returns a new sequence, containing all the elements of seq except the element at pos.

```
erase(seq, first, last);
```

Return type:

- A model of [Forward Sequence](#).
- A model of [Associative Sequence](#) if seq implements the [Associative Sequence](#) model.

Semantics: Returns a new sequence, with all the elements of seq, in their original order, except those in the range [first,last).

Complexity

Constant. Returns a view which is lazily evaluated.

Header

```
#include <boost/fusion/algorithm/transformation/erase.hpp>
#include <boost/fusion/include/erase.hpp>
```

Example

```
const vector<int, double, char> vec(1, 2.0, 'c');
assert(erase(vec, next(begin(vec))) == make_vector(1, 'c'));
assert(erase(vec, next(begin(vec)), end(vec)) == make_vector(1));
```

erase_key

Description

For an [associative](#) [Forward Sequence](#) `seq`, returns a [associative](#) [Forward Sequence](#) containing all the elements of the original except those with a given key.

Synopsis

```
template<
    typename Key,
    typename Sequence
>
typename result_of::erase_key<Sequence const, Key>::type erase_key(Sequence const& seq);
```

Table 68. Parameters

Parameter	Requirement	Description
<code>seq</code>	A model of Forward Sequence and Associative Sequence	Operation's argument
<code>Key</code>	Any type	Key to erase

Expression Semantics

```
erase_key<Key>(seq);
```

Return type: A model of [Forward Sequence](#) and [Associative Sequence](#).

Semantics: Returns a new sequence, containing all the elements of `seq`, except those with key `Key`.

Complexity

Constant. Returns a view which is lazily evaluated.

Header

```
#include <boost/fusion/algorithm/transformation/erase_key.hpp>
#include <boost/fusion/include/erase_key.hpp>
```

Example

```
assert(erase_key<int>(make_map<int, long>('a', 'b')) == make_map<long>('b'));
```

insert

Description

Returns a new sequence with all the elements of the original, an a new element inserted the position described by a given iterator.

Synopsis

```
template<
    typename Sequence,
    typename Pos,
    typename T
>
typename result_of::insert<Sequence const, Pos, T>::type insert(
    Sequence const& seq, Pos const& pos, T const& t);
```

Table 69. Parameters

Parameter	Requirement	Description
seq	A model of Forward Sequence	Operation's argument
pos	A model of Forward Iterator	The position to insert at
t	Any type	The value to insert

Expression Semantics

```
insert(seq, p, t);
```

Return type:

- A model of [Forward Sequence](#).
- A model of [Associative Sequence](#) if seq implements the [Associative Sequence](#) model.

Semantics: Returns a new sequence, containing all the elements of seq, in their original order, and a new element with the type and value of t inserted at iterator pos.

Complexity

Constant. Returns a view which is lazily evaluated.

Header

```
#include <boost/fusion/algorithm/transformation/insert.hpp>
#include <boost/fusion/include/insert.hpp>
```

Example

```
const vector<int,int> vec(1,2);
assert(insert(vec, next(begin(vec)), 3) == make_vector(1,3,2));
```

insert_range

Description

Returns a new sequence with another sequence inserted at a specified iterator.

Synopsis

```
template<
    typename Sequence,
    typename Pos,
    typename Range
>
typename result_of::insert_range<Sequence const, Pos, Range>::type insert_range(
    Sequence const& seq, Pos const& pos, Range const& range);
```

Table 70. Parameters

Parameter	Requirement	Description
seq	A model of Forward Sequence	Operation's argument
pos	A model of Forward Iterator	The position to insert at
range	A model of Forward Sequence	Range to insert

Expression Semantics

```
insert(seq, pos, range);
```

Return type:

- A model of [Forward Sequence](#).
- A model of [Associative Sequence](#) if `seq` implements the [Associative Sequence](#) model.

Semantics: Returns a new sequence, containing all the elements of `seq`, and the elements of `range` inserted at iterator `pos`. All elements retaining their ordering from the original sequences.

Complexity

Constant. Returns a view which is lazily evaluated.

Header

```
#include <boost/fusion/algorithm/transformation/insert_range.hpp>
#include <boost/fusion/include/insert_range.hpp>
```

Example

```
const vector<int,int> vec(1,2);
assert(insert_range(vec, next(begin(vec)), make_vector(3,4)) == make_vector(1,3,4,2));
```

join

Description

Takes 2 sequences and returns a sequence containing the elements of the first followed by the elements of the second.

Synopsis

```
template<
    typename LhSequence,
    typename RhSequence>
typename result_of::join<LhSequence, RhSequence>::type join(LhSequence const& lhs, RhSequence const& rhs);
```

Table 71. Parameters

Parameter	Requirement	Description
lhs	A model of Forward Sequence	Operation's argument
rhs	A model of Forward Sequence	Operation's argument

Expression Semantics

```
join(lhs, rhs);
```

Return type:

- A model of [Forward Sequence](#).
- A model of [Associative Sequence](#) if lhs and rhs implement the [Associative Sequence](#) model.

Semantics: Returns a sequence containing all the elements of lhs followed by all the elements of rhs. The order of the elements is preserved.

Complexity

Constant. Returns a view which is lazily evaluated.

Header

```
#include <boost/fusion/algorithm/transformation/join.hpp>
#include <boost/fusion/include/join.hpp>
```

Example

```
vector<int, char> v1(1, 'a');
vector<int, char> v2(2, 'b');
assert(join(v1, v2) == make_vector(1, 'a', 2, 'b'));
```

zip

Description

Zips sequences together to form a single sequence, whos members are tuples of the members of the component sequences.

Synopsis

```
template<
    typename Sequence1,
    typename Sequence2,
    ...
    typename SequenceN
>
typename result_of::zip<Sequence1, Sequence2, ... SequenceN>::type
zip(Sequence1 const& seq1, Sequence2 const& seq2, ... SequenceN const& seqN);
```

Table 72. Parameters

Parameter	Requirement	Description
seq1 to seqN	Each sequence is a model of Forward Sequence .	Operation's argument

Expression Semantics

```
zip(seq1, seq2, ... seqN);
```

Return type: A model of [Forward Sequence](#).

Semantics: Returns a sequence containing tuples of elements from sequences seq1 to seqN. For example, applying zip to tuples (1, 2, 3) and ('a', 'b', 'c') would return ((1, 'a'), (2, 'b'), (3, 'c'))

Complexity

Constant. Returns a view which is lazily evaluated.

Header

```
#include <boost/fusion/algorithm/transformation/zip.hpp>
#include <boost/fusion/include/zip.hpp>
```

Example

```
vector<int, char> v1(1, 'a');
vector<int, char> v2(2, 'b');
assert(zip(v1, v2) == make_vector(make_vector(1, 2), make_vector('a', 'b')));
```

pop_back

Description

Returns a new sequence, with the last element of the original removed.

Synopsis

```
template<
    typename Sequence
>
typename result_of::pop_back<Sequence const>::type pop_back(Sequence const& seq);
```

Table 73. Parameters

Parameter	Requirement	Description
seq	A model of Forward Sequence	Operation's argument

Expression Semantics

```
pop_back(seq);
```

Return type:

- A model of [Forward Sequence](#).
- A model of [Associative Sequence](#) if seq implements the [Associative Sequence](#) model.

Semantics: Returns a new sequence containing all the elements of seq, except the last element. The elements in the new sequence are in the same order as they were in seq.

Complexity

Constant. Returns a view which is lazily evaluated.

Header

```
#include <boost/fusion/algorithm/transformation/pop_back.hpp>
#include <boost/fusion/include/pop_back.hpp>
```

Example

```
assert(____pop_back____(make_vector(1,2,3)) == make_vector(1,2));
```

pop_front

Description

Returns a new sequence, with the first element of the original removed.

Synopsis

```
template<
    typename Sequence
>
typename result_of::pop_front<Sequence const>::type pop_front(Sequence const& seq);
```

Table 74. Parameters

Parameter	Requirement	Description
seq	A model of Forward Sequence	Operation's argument

Expression Semantics

```
pop_front(seq);
```

Return type:

- A model of [Forward Sequence](#).
- A model of [Associative Sequence](#) if `seq` implements the [Associative Sequence](#) model.

Semantics: Returns a new sequence containing all the elements of `seq`, except the first element. The elements in the new sequence are in the same order as they were in `seq`.

Complexity

Constant. Returns a view which is lazily evaluated.

Header

```
#include <boost/fusion/algorithm/transformation/pop_front.hpp>
#include <boost/fusion/include/pop_front.hpp>
```

Example

```
assert(pop_front(make_vector(1,2,3)) == make_vector(2,3));
```

push_back

Description

Returns a new sequence with an element added at the end.

Synopsis

```
template<
    typename Sequence,
    typename T
>
typename result_of::push_back<Sequence, T>::type push_back(
    Sequence const& seq, T const& t);
```

Table 75. Parameters

Parameter	Requirement	Description
seq	A model of Forward Sequence	Operation's argument
t	Any type	The value to add to the end

Expression Semantics

```
push_back(seq, t);
```

Return type:

- A model of [Forward Sequence](#).
- A model of [Associative Sequence](#) if seq implements the [Associative Sequence](#) model.

Semantics: Returns a new sequence, containing all the elements of seq, and new element t appended to the end. The elements are in the same order as they were in seq.

Complexity

Constant. Returns a view which is lazily evaluated.

Header

```
#include <boost/fusion/algorithm/transformation/push_back.hpp>
#include <boost/fusion/include/push_back.hpp>
```

Example

```
assert(push_back(make_vector(1,2,3),4) == make_vector(1,2,3,4));
```

push_front

Description

Returns a new sequence with an element added at the beginning.

Synopsis

```
template<
    typename Sequence,
    typename T
>
typename result_of::push_front<Sequence, T>::type push_front(
    Sequence const& seq, T const& t);
```

Table 76. Parameters

Parameter	Requirement	Description
seq	A model of Forward Sequence	Operation's argument
t	Any type	The value to add to the beginning

Expression Semantics

```
push_back(seq, t);
```

Return type:

- A model of [Forward Sequence](#).
- A model of [Associative Sequence](#) if seq implements the [Associative Sequence](#) model.

Semantics: Returns a new sequence, containing all the elements of seq, and new element t appended to the beginning. The elements are in the same order as they were in seq.

Complexity

Constant. Returns a view which is lazily evaluated.

Header

```
#include <boost/fusion/algorithm/transformation/push_front.hpp>
#include <boost/fusion/include/push_front.hpp>
```

Example

```
assert(push_front(make_vector(1,2,3),0) == make_vector(0,1,2,3));
```

Metafunctions

filter

Description

Returns the result type of `filter` given the sequence type and type to retain.

Synopsis

```
template<
    typename Sequence,
    typename T
>
struct filter
{
    typedef unspecified type;
};
```

Table 77. Parameter

Parameter	Requirement	Description
Sequence	A model of Forward Sequence	Operation's argument
T	Any type	Type to retain

Expression Semantics

```
result_of::filter<Sequence, T>::type
```

Return type:

- A model of [Forward Sequence](#).
- A model of [Associative Sequence](#) if Sequence implements the [Associative Sequence](#) model.

Semantics: Returns a sequence containing the elements of Sequence that are of type T. Equivalent to `result_of::filter_if<Sequence, boost::is_same<mpl::__, T> >::type`.

Complexity

Constant.

Header

```
#include <boost/fusion/algorithm/transformation/filter.hpp>
#include <boost/fusion/include/filter.hpp>
```

filter_if

Description

Returns the result type of `filter_if` given the sequence and unary [MPL Lambda Expression](#) predicate type.

Synopsis

```
template<
    typename Sequence,
    typename Pred
>
struct filter_if
{
    typedef unspecified type;
};
```

Table 78. Parameter

Parameter	Requirement	Description
Sequence	A model of Forward Sequence	Operation's argument
Pred	A unary MPL Lambda Expression	Type to retain

Expression Semantics

```
result_of::filter_if<Sequence, Pred>::type
```

Return type:

- A model of [Forward Sequence](#).
- A model of [Associative Sequence](#) if Sequence implements the [Associative Sequence](#) model.

Semantics: Returns a sequence containing the elements of Sequence for which Pred evaluates to `boost::mpl::true_`.

Complexity

Constant.

Header

```
#include <boost/fusion/algorithm/transformation/filter_if.hpp>
#include <boost/fusion/include/filter_if.hpp>
```

transform

Description

For a sequence `seq` and function object or function pointer `f`, `transform` returns a new sequence with elements created by applying `f(e)` to each element of `e` of `seq`.

Unary version synopsis

```
template<
    typename Sequence,
    typename F
>
typename result_of::transform<Sequence const, F>::type transform(
    Sequence const& seq, F f);
```

Table 79. Parameters

Parameter	Requirement	Description
seq	A model of Forward Sequence	Operation's argument
f	$f(e)$ is a valid expression for each element e of seq. <code>boost::result_of<F(E)>::type</code> is the return type of f when called with a value of each element type E .	Transformation function

Expression Semantics

```
transform(seq, f);
```

Return type:

- A model of [Forward Sequence](#)
- A model of [Associative Sequence](#) if `Sequence` implements the [Associative Sequence](#) model.

Semantics: Returns a new sequence, containing the return values of $f(e)$ for each element e within `seq`.

Binary version synopsis

```
template<
    typename Sequence1,
    typename Sequence2,
    typename F
>
typename result_of::transform<Sequence1 const, Sequence2 const, F>::type transform(
    Sequence1 const& seq1, Sequence2 const& seq2, F f);
```

Table 80. Parameters

Parameter	Requirement	Description
seq1	A model of Forward Sequence	Operation's argument
seq2	A model of Forward Sequence	Operation's argument
f	$f(e1, e2)$ is a valid expression for each pair of elements $e1$ of <code>seq1</code> and $e2$ of <code>seq2</code> . <code>boost::result_of<F(E1, E2)>::type</code> is the return type of f when called with elements of type $E1$ and $E2$	Transformation function

Return type: A model of [Forward Sequence](#).

Semantics: Returns a new sequence, containing the return values of $f(e1, e2)$ for each pair of elements $e1$ and $e2$ within `seq1` and `seq2` respectively.

Complexity

Constant. Returns a view which is lazily evaluated.

Header

```
#include <boost/fusion/algorithm/transformation/transform.hpp>
#include <boost/fusion/include/transform.hpp>
```

Example

```
struct triple
{
    typedef int result_type;

    int operator()(int t) const
    {
        return t * 3;
    };
};
...
assert(transform(make_vector(1,2,3), triple()) == make_vector(3,6,9));
```

replace

Description

Returns the result type of [replace](#), given the types of the input sequence and element to replace.

Synopsis

```
template<
    typename Sequence,
    typename T
>
struct replace
{
    typedef unspecified type;
};
```

Table 81. Parameters

Parameter	Requirement	Description
Sequence	A model of Forward Sequence	Operation's argument
T	Any type	The type of the search and replacement objects

Expression Semantics

```
result_of::replace<Sequence,T>::type
```

Return type: A model of [Forward Sequence](#).

Semantics: Returns the return type of [replace](#).

Complexity

Constant.

Header

```
#include <boost/fusion/algorithm/transformation/replace.hpp>
#include <boost/fusion/include/replace.hpp>
```

replace_if

Description

Returns the result type of `replace_if`, given the types of the sequence, [Polymorphic Function Object](#) predicate and replacement object.

Synopsis

```
template<
    typename Sequence,
    typename F,
    typename T>
struct replace_if
{
    typedef unspecified type;
};
```

Table 82. Parameters

Parameter	Requirement	Description
Sequence	A model of Forward Sequence	Operation's argument
F	A model of unary Polymorphic Function Object	Replacement predicate
T	Any type	The type of the replacement object

Expression Semantics

```
result_of::replace_if<Sequence, F, T>::type
```

Return type: A model of [Forward Sequence](#).

Semantics: Returns the return type of `replace_if`.

Complexity

Constant.

Header

```
#include <boost/fusion/algorithm/transformation/replace_if.hpp>
#include <boost/fusion/include/replace_if.hpp>
```

remove

Description

Returns the result type of `remove`, given the sequence and removal types.

Synopsis

```
template<
    typename Sequence,
    typename T
>
struct remove
{
    typedef unspecified type;
};
```

Table 83. Parameters

Parameter	Requirement	Description
Sequence	A model of Forward Sequence	Operation's argument
T	Any type	Remove elements of this type

Expression Semantics

```
result_of::remove<Sequence, T>::type
```

Return type:

- A model of [Forward Sequence](#).
- A model of [Associative Sequence](#) if Sequence implements the [Associative Sequence](#) model.

Semantics: Returns a sequence containing the elements of Sequence not of type T. Equivalent to `result_of::replace_if<Sequence, boost::is_same<mpl::_1, T> >::type`.

Complexity

Constant.

Header

```
#include <boost/fusion/algorithm/transformation/remove.hpp>
#include <boost/fusion/include/remove.hpp>
```

remove_if

Description

Returns the result type of `remove_if`, given the input sequence and unary [MPL Lambda Expression](#) predicate types.

Synopsis

```
template<
    typename Sequence,
    typename Pred
>
struct remove_if
{
    typedef unspecified type;
};
```

Table 84. Parameters

Parameter	Requirement	Description
Sequence	A model of Forward Sequence	Operation's argument
Pred	A model of unary MPL Lambda Expression	Remove elements which evaluate to <code>boost::mpl::true_</code>

Expression Semantics

```
result_of::remove_if<Sequence, Pred>::type
```

Return type:

- A model of [Forward Sequence](#).
- A model of [Associative Sequence](#) if Sequence implements the [Associative Sequence](#) model.

Semantics: Returns a sequence containing the elements of Sequence for which Pred evaluates to `boost::mpl::false_`.

Complexity

Constant.

Header

```
#include <boost/fusion/algorithm/transformation/remove_if.hpp>
#include <boost/fusion/include/remove_if.hpp>
```

reverse

Description

Returns the result type of [reverse](#), given the input sequence type.

Synopsis

```
template<
    typename Sequence
>
struct reverse
{
    typedef unspecified type;
};
```

Table 85. Parameters

Parameter	Requirement	Description
Sequence	A model of Bidirectional Sequence	Operation's argument

Expression Semantics

```
result_of::reverse<Sequence>::type
```

Return type:

- A model of [Bidirectional Sequence](#) if Sequence is a [Bidirectional Sequence](#) else, [Random Access Sequence](#) if Sequence is a [Random Access Sequence](#).
- A model of [Associative Sequence](#) if Sequence implements the [Associative Sequence](#) model.

Semantics: Returns a sequence with the elements in the reverse order to Sequence.

Complexity

Constant.

Header

```
#include <boost/fusion/algorithm/transformation/reverse.hpp>
#include <boost/fusion/include/reverse.hpp>
```

clear

Description

Returns the result type of [clear](#), given the input sequence type.

Synopsis

```
template<
    typename Sequence
>
struct clear
{
    typedef unspecified type;
};
```

Table 86. Parameters

Parameter	Requirement	Description
Sequence	Any type	Operation's argument

Expression Semantics

```
result_of::clear<Sequence>::type
```

Return type: A model of [Forward Sequence](#).

Semantics: Returns an empty sequence.

Complexity

Constant.

Header

```
#include <boost/fusion/algorithm/transformation/clear.hpp>
#include <boost/fusion/include/clear.hpp>
```

erase

Returns the result type of [erase](#), given the input sequence and range delimiting iterator types.

Description

Synopsis

```
template<
    typename Sequence,
    typename It1,
    typename It2 = unspecified>
struct erase
{
    typedef unspecified type;
};
```

Table 87. Parameters

Parameter	Requirement	Description
Sequence	A model of Forward Sequence	Operation's argument
It1	A model of Forward Iterator	Operation's argument
It2	A model of Forward Iterator	Operation's argument

Expression Semantics

```
result_of::erase<Sequence, It1>::type
```

Return type:

- A model of [Forward Sequence](#).
- A model of [Associative Sequence](#) if Sequence implements the [Associative Sequence](#) model.

Semantics: Returns a new sequence with the element at `It1` removed.

```
result_of::erase<Sequence, It1, It2>::type
```

Return type: A model of [Forward Sequence](#).

Semantics: Returns a new sequence with the elements between `It1` and `It2` removed.

Complexity

Constant.

Header

```
#include <boost/fusion/algorithm/transformation/erase.hpp>
#include <boost/fusion/include/erase.hpp>
```

erase_key

Description

Returns the result type of [erase_key](#), given the sequence and key types.

Synopsis

```
template<
    typename Sequence,
    typename Key
>
struct erase_key
{
    typedef unspecified type;
};
```

Table 88. Parameters

Parameter	Requirement	Description
Sequence	A model of Forward Sequence and Associative Sequence	Operation's argument
Key	Any type	Key type

Expression Semantics

```
result_of::erase_key<Sequence, Key>::type
```

Return type: A model of [Forward Sequence](#) and [Associative Sequence](#).

Semantics: Returns a sequence with the elements of *Sequence*, except those with key *Key*.

Complexity

Constant.

Header

```
#include <boost/fusion/algorithm/transformation/erase_key.hpp>
#include <boost/fusion/include/erase_key.hpp>
```

insert

Description

Returns the result type of [insert](#), given the sequence, position iterator and insertion types.

Synopsis

```
template<
    typename Sequence,
    typename Position,
    typename T
>
struct insert
{
    typedef unspecified type;
};
```

Table 89. Parameters

Parameter	Requirement	Description
Sequence	A model of Forward Sequence	Operation's argument
Position	A model of Forward Iterator	Operation's argument
T	Any type	Operation's argument

Expression Semantics

```
result_of::insert<Sequence, Position, T>::type
```

Return type:

- A model of [Forward Sequence](#).
- A model of [Associative Sequence](#) if Sequence implements the [Associative Sequence](#) model.

Semantics: Returns a sequence with an element of type T inserted at position Position in Sequence.

Complexity

Constant.

Header

```
#include <boost/fusion/algorithm/transformation/insert.hpp>
#include <boost/fusion/include/insert.hpp>
```

insert_range

Description

Returns the result type of [insert_range](#), given the input sequence, position iterator and insertion range types.

Synopsis

```
template<
    typename Sequence,
    typename Position,
    typename Range
>
struct insert_range
{
    typedef unspecified type;
};
```

Table 90. Parameters

Parameter	Requirement	Description
Sequence	A model of Forward Sequence	Operation's argument
Position	A model of Forward Iterator	Operation's argument
Range	A model of Forward Sequence	Operation's argument

Expression Semantics

```
result_of::insert_range<Sequence, Position, Range>::type
```

Return type:

- A model of [Forward Sequence](#).
- A model of [Associative Sequence](#) if Sequence implements the [Associative Sequence](#) model.

Semantics: Returns a sequence with the elements of Range inserted at position Position into Sequence.

Complexity

Constant.

Header

```
#include <boost/fusion/algorithm/transformation/insert_range.hpp>
#include <boost/fusion/include/insert_range.hpp>
```

join

Description

Returns the result of joining 2 sequences, given the sequence types.

Synopsis

```
template<
    typename LhSequence,
    typename RhSequence
>
struct join
{
    typedef unspecified type;
};
```

Expression Semantics

```
result_of::join<LhSequence, RhSequence>::type
```

Return type:

- A model of [Forward Sequence](#).
- A model of [Associative Sequence](#) if LhSequence and RhSequence implement the [Associative Sequence](#) model.

Semantics: Returns a sequence containing the elements of LhSequence followed by the elements of RhSequence. The order of the elements in the 2 sequences is preserved.

Complexity

Constant.

Header

```
#include <boost/fusion/algorithm/transformation/join.hpp>
#include <boost/fusion/include/join.hpp>
```

zip

Description

Zips sequences together to form a single sequence, whose members are tuples of the members of the component sequences.

Synopsis

```
template<
    typename Sequence1,
    typename Sequence2,
    ...
    typename SequenceN
>
struct zip
{
    typedef unspecified type;
};
```

Expression Semantics

```
result_of::zip<Sequence1, Sequence2, ... SequenceN>::type
```

Return type: A model of the most restrictive traversal category of sequences Sequence1 to SequenceN.

Semantics: Return a sequence containing tuples of elements from each sequence. For example, applying `zip` to tuples `(1, 2, 3)` and `('a', 'b', 'c')` would return `((1, 'a'), (2, 'b'), (3, 'c'))`

Complexity

Constant.

Header

```
#include <boost/fusion/algorithm/transformation/zip.hpp>
#include <boost/fusion/include/zip.hpp>
```

pop_back

Description

Returns the result type of `pop_back`, given the input sequence type.

Synopsis

```
template<
    typename Sequence
>
struct pop_back
{
    typedef unspecified type;
};
```

Table 91. Parameters

Parameter	Requirement	Description
Sequence	A model of Forward Sequence	Operation's argument

Expression Semantics

```
result_of::pop_back<Sequence>::type
```

Return type:

- A model of [Forward Sequence](#).
- A model of [Associative Sequence](#) if Sequence implements the [Associative Sequence](#) model.

Semantics: Returns a sequence with all the elements of Sequence except the last element.

Complexity

Constant.

Header

```
#include <boost/fusion/algorithm/transformation/pop_back.hpp>
#include <boost/fusion/include/pop_back.hpp>
```

pop_front

Description

Returns the result type of `pop_front`, given the input sequence type.

Synopsis

```
template<
    typename Sequence
>
struct pop_front
{
    typedef unspecified type;
};
```

Table 92. Parameters

Parameter	Requirement	Description
Sequence	A model of Forward Sequence	Operation's argument

Expression Semantics

```
result_of::pop_front<Sequence>::type
```

Return type:

- A model of [Forward Sequence](#).
- A model of [Associative Sequence](#) if Sequence implements the [Associative Sequence](#) model.

Semantics: Returns a sequence with all the elements of Sequence except the first element.

Complexity

Constant.

Header

```
#include <boost/fusion/algorithm/transformation/pop_front.hpp>
#include <boost/fusion/include/pop_front.hpp>
```

push_back

Description

Returns the result type of `push_back`, given the types of the input sequence and element to push.

Synopsis

```
template<
    typename Sequence,
    typename T
>
struct push_back
{
    typedef unspecified type;
};
```

Table 93. Parameters

Parameter	Requirement	Description
Sequence	A model of Forward Sequence	Operation's argument
T	Any type	Operation's argument

Expression Semantics

```
result_of::push_back<Sequence, T>::type
```

Return type:

- A model of [Forward Sequence](#).
- A model of [Associative Sequence](#) if Sequence implements the [Associative Sequence](#) model.

Semantics: Returns a sequence with the elements of Sequence and an element of type T added to the end.

Complexity

Constant.

Header

```
#include <boost/fusion/algorithm/transformation/push_back.hpp>
#include <boost/fusion/include/push_back.hpp>
```

push_front

Description

Returns the result type of [push_front](#), given the types of the input sequence and element to push.

Synopsis

```
template<
    typename Sequence,
    typename T
>
struct push_front
{
    typedef unspecified type;
};
```

Table 94. Parameters

Parameter	Requirement	Description
Sequence	A model of Forward Sequence	Operation's argument
T	Any type	Operation's argument

Expression Semantics

```
result_of::push_front<Sequence, T>::type
```

Return type:

- A model of [Forward Sequence](#).
- A model of [Associative Sequence](#) if Sequence implements the [Associative Sequence](#) model.

Semantics: Returns a sequence with the elements of Sequence and an element of type T added to the beginning.

Complexity

Constant.

Header

```
#include <boost/fusion/algorithm/transformation/push_front.hpp>
#include <boost/fusion/include/push_front.hpp>
```

Tuple

The TR1 technical report describes extensions to the C++ standard library. Many of these extensions will be considered for the next iteration of the C++ standard. TR1 describes a tuple type, and support for treating `std::pair` as a type of tuple.

Fusion provides full support for the [TR1 Tuple](#) interface, and the extended uses of `std::pair` described in the TR1 document.

Class template tuple

Fusion's implementation of the [TR1 Tuple](#) is also a fusion [Forward Sequence](#). As such the fusion tuple type provides a lot of functionality beyond that required by TR1.

Currently tuple is basically a synonym for [vector](#), although this may be changed in future releases of fusion.

Synopsis

```
template<
    typename T1 = unspecified,
    typename T2 = unspecified,
    ...
    typename TN = unspecified>
class tuple;
```

/tuple.hpp>

Construction

Description

The **TR1 Tuple** type provides a default constructor, a constructor that takes initializers for all of its elements, a copy constructor, and a converting copy constructor. The details of the various constructors are described in this section.

Specification

Notation

$T_1 \dots T_N, U_1 \dots U_N$	Tuple element types
$P_1 \dots P_N$	Parameter types
T_i, U_i	The type of the i th element of a tuple
P_i	The type of the i th parameter

```
tuple();
```

Requirements: Each T_i is default constructable.

Semantics: Default initializes each element of the tuple.

```
tuple(P1, P2, ..., PN);
```

Requirements: Each P_i is T_i if T_i is a reference type, `const T_i&` otherwise.

Semantics: Copy initializes each element with the corresponding parameter.

```
tuple(const tuple& t);
```

Requirements: Each T_i should be copy constructable.

Semantics: Copy constructs each element of `*this` with the corresponding element of `t`.

```
template<typename U1, typename U2, ..., typename UN>
tuple(const tuple<U1, U2, ..., UN>& t);
```

Requirements: Each T_i shall be constructible from the corresponding U_i .

Semantics: Constructs each element of `*this` with the corresponding element of `t`.

Tuple creation functions

Description

TR1 describes 2 utility functions for creating `_tr1tuple_s`. `make_tuple` builds a tuple out of it's argument list, and `tie` builds a tuple of references to it's arguments. The details of these creation functions are described in this section.

Specification

```
template<typename T1, typename T2, ..., typename TN>
tuple<V1, V2, ..., VN> make_tuple(const T1& t1, const T2& t2, ..., const TN& tn);
```

Where V_i is `X&` if the cv-unqualified type T_i is `reference_wrapper<X>`, otherwise V_i is T_i .

Returns: `tuple<V1, V2, ..., VN>(t1, t2, ..., tN)`

```
template<typename T1, typename T2, ..., typename TN>
tuple<T1&, T2&, ..., TN&> tie(T1& t1, T2& t2, ..., TN& tn);
```

Returns: `tuple<T1&, T2&, ..., TN&>(t1, t2, ..., tN)`. When argument `ti` is `ignore`, assigning any value to the corresponding tuple element has no effect.

Tuple helper classes

Description

The [TR1 Tuple](#) provides 2 helper traits, for compile time access to the tuple size, and the element types.

Specification

```
tuple_size<T>::value
```

Requires: T is any fusion sequence type, including `tuple`.

Type: [MPL Integral Constant](#)

Value: The number of elements in the sequence. Equivalent to `result_of::size<T>::type`.

```
tuple_element<I, T>::type
```

Requires: T is any fusion sequence type, including `tuple`. $0 \leq I < N$ or the program is ill formed.

Value: The type of the i th element of T . Equivalent to `result_of::value_at<I, T>::type`.

Element access

Description

The [TR1 Tuple](#) provides the `get` function to provide access to it's elements by zero based numeric index.

Specification

```
template<int I, T>
RJ get(T& t);
```

Requires: $0 \leq I \leq N$. The program is ill formed if I is out of bounds. T is any fusion sequence type, including `tuple`.

Return type: `RJ` is equivalent to `result_of::at_c<I,T>::type`.

Returns: A reference to the `I`th element of `T`.

```
template<int I, typename T>
PJ get(T const& t);
```

Requires: $0 < I \leq N$. The program is ill formed if `I` is out of bounds. `T` is any fusion sequence type, including `tuple`.

Return type: `PJ` is equivalent to `result_of::at_c<I,T>::type`.

Returns: A const reference to the `I`th element of `T`.

Relational operators

Description

The [TR1 Tuple](#) provides the standard boolean relational operators.

Specification

Notation

<code>T1 ... TN, U1 ... UN</code>	Tuple element types
<code>P1 ... PN</code>	Parameter types
<code>Ti, Ui</code>	The type of the <code>i</code> th element of a tuple
<code>Pi</code>	The type of the <code>i</code> th parameter

```
template<typename T1, typename T2, ..., typename TN,
        typename U1, typename U2, ..., typename UN>
bool operator==(
    const tuple<T1, T2, ..., TN>& lhs,
    const tuple<U1, U2, ..., UN>& rhs);
```

Requirements: For all `i`, $1 \leq i \leq N$, `get<i>(lhs) == get<i>(rhs)` is a valid expression returning a type that is convertible to `bool`.

Semantics: Returns `true` if and only if `get<i>(lhs) == get<i>(rhs)` for all `i`. For any 2 zero length tuples `e` and `f`, `e == f` returns `true`.

```
template<typename T1, typename T2, ..., typename TN,
        typename U1, typename U2, ..., typename UN>
bool operator<(
    const tuple<T1, T2, ..., TN>& lhs,
    const tuple<U1, U2, ..., UN>& rhs);
```

Requirements: For all `i`, $1 \leq i \leq N$, `get<i>(lhs) < get<i>(rhs)` is a valid expression returning a type that is convertible to `bool`.

Semantics: Returns the lexicographical comparison of between `lhs` and `rhs`.

```
template<typename T1, typename T2, ..., typename TN,
        typename U1, typename U2, ..., typename UN>
bool operator!=(
    const tuple<T1, T2, ..., TN>& lhs,
    const tuple<U1, U2, ..., UN>& rhs);
```

Requirements: For all $i, 1 \leq i < N$, `get< i >(lhs) == get< i >(rhs)` is a valid expression returning a type that is convertible to `bool`.

Semantics: Returns `!(lhs == rhs)`.

```
template<typename T1, typename T2, ..., typename TN,
        typename U1, typename U2, ..., typename UN>
bool operator<=(
    const tuple<T1, T2, ..., TN>& lhs,
    const tuple<U1, U2, ..., UN>& rhs);
```

Requirements: For all $i, 1 \leq i < N$, `get< i >(rhs) < get< i >(lhs)` is a valid expression returning a type that is convertible to `bool`.

Semantics: Returns `!(rhs < lhs)`

```
template<typename T1, typename T2, ..., typename TN,
        typename U1, typename U2, ..., typename UN>
bool operator>(
    const tuple<T1, T2, ..., TN>& lhs,
    const tuple<U1, U2, ..., UN>& rhs);
```

Requirements: For all $i, 1 \leq i < N$, `get< i >(rhs) < get< i >(lhs)` is a valid expression returning a type that is convertible to `bool`.

Semantics: Returns `rhs < lhs`.

```
template<typename T1, typename T2, ..., typename TN,
        typename U1, typename U2, ..., typename UN>
bool operator>=(
    const tuple<T1, T2, ..., TN>& lhs,
    const tuple<U1, U2, ..., UN>& rhs);
```

Requirements: For all $i, 1 \leq i < N$, `get< i >(lhs) < get< i >(rhs)` is a valid expression returning a type that is convertible to `bool`.

Semantics: Returns `!(lhs < rhs)`.

Pairs

Description

The [TR1 Tuple](#) interface is specified to provide uniform access to `std::pair` as if it were a 2 element tuple.

Specification

```
tuple_size<std::pair<T1, T2> >::value
```

Type: An [MPL Integral Constant](#)

Value: Returns 2, the number of elements in a pair.

```
tuple_element<0, std::pair<T1, T2> >::type
```

Type: T1

Value: Returns the type of the first element of the pair

```
tuple_element<1, std::pair<T1, T2> >::type
```

Type: T2

Value: Returns the type of the second element of the pair

```
template<int I, typename T1, typename T2>
P& get(std::pair<T1, T2>& pr);

template<int I, typename T1, typename T2>
const P& get(const std::pair<T1, T2>& pr);
```

Type: If `I == 0` `P` is `T1`, else if `I == 1` `P` is `T2` else the program is ill-formed.

Returns: `pr.first` if `I == 0` else `pr.second`. [*Returns: `pr.first` if `I == 0` else `pr.second`.]

Extension

The Full Extension Mechanism

The Fusion library is designed to be extensible, new sequence types can easily be added. In fact, the library support for `std::pair`, `boost::array` and [MPL](#) sequences is entirely provided using the extension mechanism.

The process for adding a new sequence type to Fusion is:

1. Enable the [tag dispatching](#) mechanism used by Fusion for your sequence type
2. Design an iterator type for the sequence
3. Provide specialized behaviour for the intrinsic operations of the new Fusion sequence

Our example

In order to illustrate enabling a new sequence type for use with Fusion, we are going to use the type:

```
namespace example
{
    struct example_struct
    {
        std::string name;
        int age;
        example_struct(
            const std::string& n,
            int a)
            : name(n), age(a)
        {}
    };
}
```

We are going to pretend that this type has been provided by a 3rd party library, and therefore cannot be modified. We shall work through all the necessary steps to enable `example_struct` to serve as an [Associative Sequence](#) as described in the [Quick Start](#) guide.

Enabling Tag Dispatching

The Fusion extensibility mechanism uses *tag dispatching* to call the correct code for a given sequence type. In order to exploit the tag dispatching mechanism we must first declare a new tag type for the mechanism to use. For example:

```
namespace example {
    struct example_sequence_tag; // Only definition needed
}
```

Next we need to enable the `traits::tag_of` metafunction to return our newly chosen tag type for operations involving our sequence. This is done by specializing `traits::tag_of` for our sequence type.

```
#include <boost/fusion/support/tag_of_fwd.hpp>
#include <boost/fusion/include/tag_of_fwd.hpp>

namespace boost { namespace fusion { namespace traits {
    template<>
    struct tag_of<example_struct>
    {
        typedef example::example_sequence_tag type;
    };
}}}
```

`traits::tag_of` also has a second template argument, that can be used in conjunction with `boost::enable_if` to provide tag support for groups of related types. This feature is not necessary for our sequence, but for an example see the code in:

```
#include <boost/fusion/adapted/array/tag_of.hpp>
#include <boost/fusion/include/tag_of.hpp>
```

Designing a suitable iterator

We need an iterator to describe positions, and provide access to the data within our sequence. As it is straightforward to do, we are going to provide a random access iterator in our example.

We will use a simple design, in which the 2 members of `example_struct` are given numbered indices, 0 for name and 1 for age respectively.

```

template<typename Struct, int Pos>
struct example_struct_iterator
    : boost::fusion::iterator_base<example_struct_iterator<Struct, Pos> >
{
    BOOST_STATIC_ASSERT(Pos >=0 && Pos < 3);
    typedef Struct struct_type;
    typedef boost::mpl::int_<Pos> index;
    typedef boost::fusion::random_access_traversal_tag category;

    example_struct_iterator(Struct& str)
        : struct_(str) {}

    Struct& struct_;
};

```

A quick summary of the details of our iterator:

1. The iterator is parameterized by the type it is iterating over, and the index of the current element.
2. The typedefs `struct_type` and `index` provide convenient access to information we will need later in the implementation.
3. The typedef `category` allows the `traits::category_of` metafunction to establish the traversal category of the iterator.
4. The constructor stores a reference to the `example_struct` being iterated over.

We also need to enable *tag dispatching* for our iterator type, with another specialization of `traits::tag_of`.

In isolation, the iterator implementation is pretty dry. Things should become clearer as we add features to our implementation.

A first couple of instructive features

To start with, we will get the `result_of::value_of` metafunction working. To do this, we provide a specialization of the `boost::fusion::extension::value_of_impl` template for our iterator's tag type.

```

template<>
struct value_of_impl<example::example_struct_iterator_tag>
{
    template<typename Iterator>
    struct apply;

    template<typename Struct>
    struct apply<example::example_struct_iterator<Struct, 0> >
    {
        typedef std::string type;
    };

    template<typename Struct>
    struct apply<example::example_struct_iterator<Struct, 1> >
    {
        typedef int type;
    };
};

```

The implementation itself is pretty simple, it just uses 2 partial specializations to provide the type of the 2 different members of `example_struct`, based on the index of the iterator.

To understand how `value_of_impl` is used by the library we will look at the implementation of `result_of::value_of`:

```
template <typename Iterator>
struct value_of
: extension::value_of_impl<typename detail::tag_of<Iterator>::type>::
    template apply<Iterator>
{};
```

So `result_of::value_of` uses *tag dispatching* to select an [MPL Metafunction Class](#) to provide its functionality. You will notice this pattern throughout the implementation of Fusion.

Ok, lets enable dereferencing of our iterator. In this case we must provide a suitable specialization of `deref_impl`.

```
template<>
struct deref_impl<example::example_struct_iterator_tag>
{
    template<typename Iterator>
    struct apply;

    template<typename Struct>
    struct apply<example::example_struct_iterator<Struct, 0> >
    {
        typedef typename mpl::if_<
            is_const<Struct>, std::string const&, std::string&>::type type;

        static type
        call(example::example_struct_iterator<Struct, 0> const& it)
        {
            return it.struct_.name;
        }
    };

    template<typename Struct>
    struct apply<example::example_struct_iterator<Struct, 1> >
    {
        typedef typename mpl::if_<
            is_const<Struct>, int const&, int&>::type type;

        static type
        call(example::example_struct_iterator<Struct, 1> const& it)
        {
            return it.struct_.age;
        }
    };
};
```

The use of `deref_impl` is very similar to that of `value_of_impl`, but it also provides some runtime functionality this time via the `call` static member function. To see how `deref_impl` is used, lets have a look at the implementation of `deref`:


```

namespace result_of
{
    template <typename Iterator>
    struct deref
    : extension::deref_impl<typename detail::tag_of<Iterator>::type>::
        template apply<Iterator>
    {};
}

template <typename Iterator>
typename result_of::deref<Iterator>::type
deref(Iterator const& i)
{
    typedef result_of::deref<Iterator> deref_meta;
    return deref_meta::call(i);
}

```

So again `result_of::deref` uses *tag dispatching* in exactly the same way as the `result_of::value_of` implementation. The runtime functionality used by `deref` is provided by the `call` static function of the selected [MPL Metafunction Class](#).

The actual implementation of `deref_impl` is slightly more complex than that of `value_of_impl`. We also need to implement the `call` function, which returns a reference to the appropriate member of the underlying sequence. We also require a little bit of metaprogramming to return `const` references if the underlying sequence is `const`.



Note

Although there is a fair amount of left to do to produce a fully fledged Fusion sequence, `result_of::value_of` and `deref` illustrate all the significant concepts required. The remainder of the process is very repetitive, simply requiring implementation of a suitable `xxxx_impl` for each feature `xxxx`.

Implementing the remaining iterator functionality

Ok, now we have seen the way `result_of::value_of` and `deref` work, everything else will work in pretty much the same way. Lets start with forward iteration, by providing a `next_impl`:

```

template<>
struct next_impl<example::example_struct_iterator_tag>
{
    template<typename Iterator>
    struct apply
    {
        typedef typename Iterator::struct_type struct_type;
        typedef typename Iterator::index index;
        typedef example::example_struct_iterator<struct_type, index::value + 1> type;

        static type
        call(Iterator const& i)
        {
            return type(i.struct_);
        }
    };
};

```

This should be very familiar from our `deref_impl` implementation, we will be using this approach again and again now. Our design is simply to increment the `index` counter to move on to the next element. The various other iterator manipulations we need to perform will all just involve simple calculations with the `index` variables.

We also need to provide a suitable `equal_to_impl` so that iterators can be correctly compared. A [Bidirectional Iterator](#) will also need an implementation of `prior_impl`. For a [Random Access Iterator](#) `distance_impl` and `advance_impl` also need to be

provided in order to satisfy the necessary complexity guarantees. As our iterator is a [Random Access Iterator](#) we will have to implement all of these functions.

Full implementations of `prior_impl`, `advance_impl`, `distance_impl` and `equal_to_impl` are provided in the example code.

Implementing the intrinsic functions of the sequence

In order that Fusion can correctly identify our sequence as a Fusion sequence, we need to enable `is_sequence` for our sequence type. As usual we just create an `impl` type specialized for our sequence tag:

```
template<>
struct is_sequence_impl<example::example_sequence_tag>
{
    template<typename T>
    struct apply : mpl::true_ {};
};
```

We've some similar formalities to complete, providing `category_of_impl` so Fusion can correctly identify our sequence type, and `is_view_impl` so Fusion can correctly identify our sequence as not being a [View](#) type. Implementations are provide in the example code.

Now we've completed some formalities, on to more interesting features. Lets get [begin](#) working so that we can get an iterator to start accessing the data in our sequence.

```
template<>
struct begin_impl<example::example_sequence_tag>
{
    template<typename Sequence>
    struct apply
    {
        typedef example::example_struct_iterator<Sequence, 0> type;

        static type
        call(Sequence& seq)
        {
            return type(seq);
        }
    };
};
```

The implementation uses the same ideas we have applied throughout, in this case we are just creating one of the iterators we developed earlier, pointing to the first element in the sequence. The implementation of [end](#) is very similar, and is provided in the example code.

For our [Random Access Sequence](#) we will also need to implement `size_impl`, `value_at_impl` and `at_impl`.

Enabling our type as an associative sequence

In order for `example_struct` to serve as an associative forward sequence, we need to adapt the traversal category of our sequence and our iterator accordingly and enable 3 intrinsic sequence lookup features, `at_key`, `value_at_key` and `has_key`. We also need to enable 3 iterator lookup features, `result_of::key_of`, `result_of::value_of_data` and `deref_data`.

To implement `at_key_impl` we need to associate the `fields::name` and `fields::age` types described in the [Quick Start](#) guide with the appropriate members of `example_struct`. Our implementation is as follows:

```

template<>
struct at_key_impl<example::example_sequence_tag>
{
    template<typename Sequence, typename Key>
    struct apply;

    template<typename Sequence>
    struct apply<Sequence, fields::name>
    {
        typedef typename mpl::if_<
            is_const<Sequence>,
            std::string const&,
            std::string&>::type type;

        static type
        call(Sequence& seq)
        {
            return seq.name;
        };
    };

    template<typename Sequence>
    struct apply<Sequence, fields::age>
    {
        typedef typename mpl::if_<
            is_const<Sequence>,
            int const&,
            int&>::type type;

        static type
        call(Sequence& seq)
        {
            return seq.age;
        };
    };
};

```

Its all very similar to the implementations we've seen previously, such as `deref_impl` and `value_of_impl`. Instead of identifying the members by index or position, we are now selecting them using the types `fields::name` and `fields::age`. The implementations of the other functions are equally straightforward, and are provided in the example code.

Summary

We've now worked through the entire process for adding a new random access sequence and we've also enabled our type to serve as an associative sequence. The implementation was slightly longwinded, but followed a simple repeating pattern.

The support for `std::pair`, [MPL](#) sequences, and `boost::array` all use the same approach, and provide additional examples of the approach for a variety of types.

Sequence Facade

Description

The `sequence_facade` template provides an intrusive mechanism for producing a conforming Fusion iterator.

Synopsis

```
template<typename Derived, typename TraversalTag, typename IsView = mpl::false_>
struct sequence_facade;
```

Usage

The user of `sequence_facade` derives his sequence type from a specialization of `sequence_facade` and passes the derived sequence type as the first template parameter. The second template parameter should be the traversal category of the sequence being implemented. The 3rd parameter should be set to `mpl::true_` if the sequence is a view.

The user must implement the key expressions required by their sequence type.

Table 95. Parameters

Name	Description
sequence, Seq	A type derived from <code>sequence_facade</code>
N	An MPL Integral Constant

Table 96. Key Expressions

Expression	Result
<code>sequence::template begin<Seq>::type</code>	The type of an iterator to the beginning of a sequence of type Seq
<code>sequence::template begin<Seq>::call(seq)</code>	An iterator to the beginning of sequence seq
<code>sequence::template end<Seq>::type</code>	The type of an iterator to the end of a sequence of type Seq
<code>sequence::template end<Seq>::call(seq)</code>	An iterator to the end of sequence seq
<code>sequence::template size<Seq>::type</code>	The size of a sequence of type Seq as an MPL Integral Constant
<code>sequence::template size<Seq>::call(seq)</code>	The size of sequence seq
<code>sequence::template at<Seq, N>::type</code>	The type of element N in a sequence of type Seq
<code>sequence::template at<Seq, N>::call(seq)</code>	Element N in sequence seq
<code>sequence::template value_at<Sequence, N>::type</code>	The type of the Nth element in a sequence of type Seq

Include

```
#include <boost/fusion/sequence/sequence_facade.hpp>
#include <boost/fusion/include/sequence_facade.hpp>
```

Example

A full working example using `sequence_facade` is provided in `triple.cpp` in the extension examples.

Iterator Facade

Description

The `iterator_facade` template provides an intrusive mechanism for producing a conforming Fusion iterator.

Synopsis

```
template<typename Derived, typename TraversalTag>
struct iterator_facade;
```

Usage

The user of `iterator_facade` derives his iterator type from a specialization of `iterator_facade` and passes the derived iterator type as the first template parameter. The second template parameter should be the traversal category of the iterator being implemented.

The user must implement the key expressions required by their iterator type.

Table 97. Parameters

Name	Description
<code>iterator</code> , <code>It</code> , <code>It1</code> , <code>It2</code>	A type derived from <code>iterator_facade</code>
<code>N</code>	An MPL Integral Constant

Table 98. Key Expressions

Expression	Result	Default
<code>iterator::template value_of<It>::type</code>	The element stored at iterator position <code>It</code>	None
<code>iterator::template deref<It>::type</code>	The type returned when dereferencing an iterator of type <code>It</code>	None
<code>iterator::template deref<It>::call(it)</code>	Dereferences iterator <code>it</code>	None
<code>iterator::template next<It>::type</code>	The type of the next element from <code>It</code>	None
<code>iterator::template next<It>::call(it)</code>	The next iterator after <code>it</code>	None
<code>iterator::template prior<It>::type</code>	The type of the next element from <code>It</code>	None
<code>iterator::template prior<It>::call(it)</code>	The next iterator after <code>it</code>	None
<code>iterator::template advance<It, N>::type</code>	The type of an iterator advanced <code>N</code> elements from <code>It</code>	Implemented in terms of <code>next</code> and <code>prior</code>
<code>iterator::template advance<It, N>::call(it)</code>	An iterator advanced <code>N</code> elements from <code>it</code>	Implemented in terms of <code>next</code> and <code>prior</code>
<code>iterator::template distance<It1, It2>::type</code>	The distance between iterators of type <code>It1</code> and <code>It2</code> as an MPL Integral Constant	None
<code>iterator::template distance<It1, It2>::call(it1, it2)</code>	The distance between iterator <code>it1</code> and <code>it2</code>	None
<code>iterator::template equal_to<It1, It2>::type</code>	The distance between iterators of type <code>It1</code> and <code>It2</code>	<code>boost::same_type<It1, It2>::type</code>
<code>iterator::template equal_to<It1, It2>::call(it1, it2)</code>	The distance between iterators <code>it1</code> and <code>it2</code>	<code>boost::same_type<It1, It2>::type()</code>
<code>iterator::template key_of<It>::type</code>	The key type associated with the element from <code>It</code>	None
<code>iterator::template value_of_data<It>::type</code>	The type of the data property associated with the element from <code>It</code>	None
<code>iterator::template deref_data<It>::type</code>	The type that will be returned by dereferencing the data property of the element from <code>It</code>	None
<code>iterator::template deref_data<It>::call(it)</code>	Dereferences the data property associated with the element referenced by <code>it</code>	None

Header

```
#include <boost/fusion/iterator/iterator_facade.hpp>
#include <boost/fusion/include/iterator_facade.hpp>
```

Example

A full working example using `iterator_facade` is provided in `triple.cpp` in the extension examples.

Functional

Components to call functions and function objects and to make Fusion code callable through a function object interface.

Header

```
#include <boost/fusion/functional.hpp>
```

Fused and unfused forms

What is a function call?

```
f (a, b, c)
```

It is a name and a tuple written next to each other, left-to-right.

Although the C++ syntax does not allow to replace `(a, b, c)` with some Fusion [Sequence](#), introducing yet another function provides a solution:

```
invoke(f, my_sequence)
```

Alternatively it is possible to apply a simple transformation to `f` in order to achieve the same effect:

```
f tuple <=> f' (tuple)
```

Now, `f'` is an unary function that takes the arguments to `f` as a tuple; `f'` is the *fused* form of `f`. Reading the above equivalence right-to-left to get the inverse transformation, `f` is the *unfused* form of `f'`.

Calling functions and function objects

Having generic C++ code call back arbitrary functions provided by the client used to be a heavily repetitive task, as different functions can differ in arity, invocation syntax and other properties that might be part of the type. Transporting arguments as Fusion sequences and factoring out the invocation makes Fusion algorithms applicable to function arguments and also reduces the problem to one invocation syntax and a fixed arity (instead of an arbitrary number of arbitrary arguments times several syntactic variants times additional properties).

Transforming an unfused function into its fused counterpart allows n-ary calls from an algorithm that invokes an unary [Polymorphic Function Object](#) with [Sequence](#) arguments.

The library provides several function templates to invoke different kinds of functions and adapters to transform them into fused form, respectively. Every variant has a corresponding generator function template that returns an adapter instance for the given argument.

Constructors can be called applying [Boost.Functional/Factory](#).

Making Fusion code callable through a function object interface

Transforming a fused function into its unfused counterpart allows to create function objects to accept arbitrary calls. In other words, an unary function object can be implemented instead of (maybe heavily overloaded) function templates or function call operators.

The library provides both a strictly typed and a generic variant for this transformation. The latter should be used in combination with [Boost.Functional/Forward](#) to attack [The Forwarding Problem](#).

Both variants have a corresponding generator function template that returns an adapter instance for the given argument.

Concepts

Callable Object

Description

A pointer to a function, a pointer to member function, a pointer to member data, or a class type whose objects can appear immediately to the left of a function call operator.

Models

- function pointer types
- member (function or data) pointer types
- all kinds of function objects

Examples

```
& a_free_function
& a_class::a_static_member_function
& a_class::a_nonstatic_data_member
& a_class::a_nonstatic_member_function
std::less<int>()
// using namespace boost;
bind(std::less<int>(), _1, 5)
lambda::_1 += lambda::_2;
fusion::make_fused_function_object(std::less<int>())
```

Regular Callable Object

Description

A non-member-pointer [Callable Object](#) type: A pointer to a function or a class type whose objects can appear immediately to the left of a function call operator.

Refinement of

- [Callable Object](#)

Notation

F	A possibly const qualified Deferred Callable Object type
f	An object or reference to an object of type F
$A1 \dots AN$	Argument types
$a1 \dots aN$	Objects or references to objects with types $A1 \dots AN$

Expression requirements

Expression	Return Type	Runtime Complexity
<code>f(a1, ...aN)</code>	Unspecified	Unspecified

Models

- function pointer types
- all kinds of function objects

Examples

```
& a_free_function
& a_class::a_static_member_function
std::less<int>()
// using namespace boost;
bind(std::less<int>(), _1, 5)
lambda::_1 += lambda::_2;
fusion::make_fused_function_object(std::less<int>())
```

Deferred Callable Object

Description

Callable Object types that work with `Boost.ResultOf` to determine the result of a call.

Refinement of

- Callable Object

note Once C++ supports the `decltype` keyword, all models of [Callable Object](#) will also be models of [Deferred Callable Object](#), because function objects won't need client-side support for `result_of`.

Notation

<code>F</code>	A possibly const qualified Deferred Callable Object type
<code>A1 ... AN</code>	Argument types
<code>a1 ... aN</code>	Objects or references to objects with types <code>A1 ... AN</code>
<code>T1 ... TN</code>	<code>Ti</code> is <code>Ai</code> & if <code>ai</code> is an LValue, same as <code>Ai</code> , otherwise

Expression requirements

Expression	Type
<code>boost::result_of< F(T1 ... TN) >::type</code>	Result of a call with <code>A1 ... AN</code> -typed arguments

Models

- Polymorphic Function Object types
- member (function or data) pointer types

Examples

```
& a_free_function
& a_class::a_static_member_function
& a_class::a_nonstatic_data_member
& a_class::a_nonstatic_member_function
std::less<int>()
// using namespace boost;
bind(std::less<int>(), _1, 5)
// Note: Boost.Lambda expressions don't work with __boost_result_of__
fusion::make_fused_function_object(std::less<int>())
```

Polymorphic Function Object

Description

A non-member-pointer [Deferred Callable Object](#) type.

Refinement of

- [Regular Callable Object](#)
- [Deferred Callable Object](#)

Notation

F	A possibly const-qualified Polymorphic Function Object type
f	An object or reference to an object of type F
$A1 \dots AN$	Argument types
$a1 \dots aN$	Objects or references to objects with types $A1 \dots AN$
$T1 \dots TN$	T_i is A_i & if a_i is an LValue, same as A_i , otherwise

Expression requirements

Expression	Return Type	Runtime Complexity
$f(a1, \dots aN)$	<code>result_of< F(T1, ...TN) >::type</code>	Unspecified

Models

- function pointers
- function objects of the Standard Library
- all Fusion [functional adapters](#)

Examples

```
& a_free_function
& a_class::a_static_member_function
std::less<int>()
// using namespace boost;
bind(std::less<int>(), _1, 5)
// Note: Boost.Lambda expressions don't work with __boost_result_of__
fusion::make_fused_function_object(std::less<int>())
```

Invocation

Functions

invoke

Description

Calls a [Deferred Callable Object](#) with the arguments from a [Sequence](#).

The first template parameter can be specialized explicitly to avoid copying and/or to control the const qualification of a function object.

If the target function is a pointer to a class members, the corresponding object can be specified as a reference, pointer, or smart pointer. In case of the latter, a freestanding `get_pointer` function must be defined (Boost provides this function for `std::auto_ptr` and `boost::shared_ptr`).

Constructors can be called applying [Boost.Functional/Factory](#).

Synopsis

```
template<
    typename Function,
    class Sequence
>
typename result_of::invoke<Function, Sequence>::type
invoke(Function f, Sequence & s);

template<
    typename Function,
    class Sequence
>
typename result_of::invoke<Function, Sequence const>::type
invoke(Function f, Sequence const & s);
```

Parameters

Parameter	Requirement	Description
<code>f</code>	A Deferred Callable Object	The function to call.
<code>s</code>	A Forward Sequence	The arguments.

Expression Semantics

```
invoke(f,s);
```

Return type: Return type of `f` when invoked with the elements in `s` as its arguments.

Semantics: Invokes `f` with the elements in `s` as arguments and returns the result of the call expression.

Header

```
#include <boost/fusion/functional/invoke/invoke.hpp>
```

Example

```
std::plus<int> add;
assert(invoke(add,make_vector(1,1)) == 2);
```

See also

- [invoke_procedure](#)
- [invoke_function_object](#)
- [result_of::invoke](#)
- [fused](#)
- [make_fused](#)

invoke_procedure

Description

Calls a [Callable Object](#) with the arguments from a [Sequence](#). The result of the call is ignored.

The first template parameter can be specialized explicitly to avoid copying and/or to control the const qualification of a function object.

For pointers to class members corresponding object can be specified as a reference, pointer, or smart pointer. In case of the latter, a freestanding `get_pointer` function must be defined (Boost provides this function for `std::auto_ptr` and [boost::shared_ptr](#)).

The target function must not be a pointer to a member object (dereferencing such a pointer without returning anything does not make sense, so it isn't implemented).

Synopsis

```
template<
    typename Function,
    class Sequence
>
typename result_of::invoke_procedure<Function, Sequence>::type
invoke_procedure(Function f, Sequence & s);

template<
    typename Function,
    class Sequence
>
typename result_of::invoke_procedure<Function, Sequence const>::type
invoke_procedure(Function f, Sequence const & s);
```

Parameters

Parameter	Requirement	Description
f	Model of Callable Object	The function to call.
s	Model of Forward Sequence	The arguments.

Expression Semantics

```
invoke_procedure(f,s);
```

Return type: void

Semantics: Invokes f with the elements in s as arguments.

Header

```
#include <boost/fusion/functional/invoke/invoke_procedure.hpp>
```

Example

```
vector<int,int> v(1,2);
using namespace boost::lambda;
invoke_procedure(_1 += _2, v);
assert(front(v) == 3);
```

See also

- [invoke](#)
- [invoke_function_object](#)
- [result_of::invoke_procedure](#)
- [fused_procedure](#)
- [make_fused_procedure](#)

invoke_function_object

Description

Calls a [Polymorphic Function Object](#) with the arguments from a [Sequence](#).

The first template parameter can be specialized explicitly to avoid copying and/or to control the const qualification of a function object.

Constructors can be called applying [Boost.Functional/Factory](#).

Synopsis

```
template<
    typename Function,
    class Sequence
>
typename result_of::invoke_function_object<Function, Sequence>::type
invoke_function_object(Function f, Sequence & s);

template<
    typename Function,
    class Sequence
>
typename result_of::invoke_function_object<Function, Sequence const>::type
invoke_function_object(Function f, Sequence const & s);
```

Parameters

Parameter	Requirement	Description
<code>f</code>	Model of Polymorphic Function Object	The function object to call.
<code>s</code>	Model of Forward Sequence	The arguments.

Expression Semantics

```
invoke_function_object(f,s);
```

Return type: Return type of `f` when invoked with the elements in `s` as its arguments.

Semantics: Invokes `f` with the elements in `s` as arguments and returns the result of the call expression.

Header

```
#include <boost/fusion/functional/invoke/invoke_function_object.hpp>
```

Example

```
struct sub
{
    template <typename Sig>
    struct result;

    template <class Self, typename T>
    struct result< Self(T,T) >
    { typedef typename remove_reference<T>::type type; };

    template<typename T>
    T operator()(T lhs, T rhs) const
    {
        return lhs - rhs;
    }
};

void try_it()
{
    sub f;
    assert(f(2,1) == invoke_function_object(f,make_vector(2,1)));
}
```

See also

- [invoke](#)
- [invoke_procedure](#)
- [result_of::invoke_function_object](#)
- [fused_function_object](#)
- [make_fused_function_object](#)

Metafunctions

invoke

Description

Returns the result type of [invoke](#).

Synopsis

```
namespace result_of
{
    template<
        typename Function,
        class Sequence
    >
    struct invoke
    {
        typedef unspecified type;
    };
}
```

See also

- [invoke](#)
- [fused](#)

invoke_procedure

Description

Returns the result type of [invoke_procedure](#).

Synopsis

```
namespace result_of
{
    template<
        typename Function,
        class Sequence
    >
    struct invoke_procedure
    {
        typedef unspecified type;
    };
}
```

See also

- [invoke_procedure](#)
- [fused_procedure](#)

invoke_function_object

Description

Returns the result type of [invoke_function_object](#).

Synopsis

```
namespace result_of
{
    template<
        class Function,
        class Sequence
    >
    struct invoke_function_object
    {
        typedef unspecified type;
    };
}
```

See also

- [invoke_function_object](#)
- [fused_function_object](#)

Limits

Header

```
#include <boost/fusion/functional/invoke/limits.hpp>
```

Macros

The following macros can be defined to change the maximum arity. The default is 6.

- `BOOST_FUSION_INVOKE_MAX_ARITY`
- `BOOST_FUSION_INVOKE_PROCEDURE_MAX_ARITY`
- `BOOST_FUSION_INVOKE_FUNCTION_OBJECT_MAX_ARITY`

Adapters

Function object templates to transform a particular target function.

fused

Description

An unary [Polymorphic Function Object](#) adapter template for [Deferred Callable Object](#) target functions. It takes a [Forward Sequence](#) that contains the arguments for the target function.

The type of the target function is allowed to be const qualified or a reference. Const qualification is preserved and propagated appropriately (in other words, only const versions of `operator()` can be used for a target function object that is const or, if the target function object is held by value, the adapter is const - these semantics have nothing to do with the const qualification of a member function, which is referring to the type of object pointed to by `this` which is specified with the first element in the sequence passed to the adapter).

If the target function is a pointer to a class members, the corresponding object can be specified as a reference, pointer, or smart pointer. In case of the latter, a freestanding `get_pointer` function must be defined (Boost provides this function for `std::auto_ptr` and `boost::shared_ptr`).

Header

```
#include <boost/fusion/functional/adapter/fused.hpp>
```

Synopsis

```
template <typename Function>
class fused;
```

Template parameters

Parameter	Description	Default
Function	A Deferred Callable Object	

Model of

- [Polymorphic Function Object](#)
- [Deferred Callable Object](#)

Notation

\mathbb{R} A possibly const qualified [Deferred Callable Object](#) type or reference type thereof

r An object convertible to \mathbb{R}

s A [Sequence](#) of arguments that are accepted by r

f An instance of `fused< \mathbb{R} >`

Expression Semantics

Expression	Semantics
<code>fused<\mathbb{R}>(r)</code>	Creates a fused function as described above, initializes the target function with r .
<code>fused<\mathbb{R}>()</code>	Creates a fused function as described above, attempts to use \mathbb{R} 's default constructor.
<code>f(s)</code>	Calls r with the elements in s as its arguments.

Example

```
fused< std::plus<long> > > f;
assert(f(make_vector(1,21)) == 31);
```

See also

- [fused_procedure](#)
- [fused_function_object](#)
- [invoke](#)
- [make_fused](#)
- [deduce](#)

fused_procedure

Description

An unary [Polymorphic Function Object](#) adapter template for [Callable Object](#) target functions. It takes a [Forward Sequence](#) that contains the arguments for the target function.

The result is discarded and the adapter's return type is `void`.

The type of the target function is allowed to be const qualified or a reference. Const qualification is preserved and propagated appropriately (in other words, only const versions of `operator()` can be used for a target function object that is const or, if the target function object is held by value, the adapter is const - these semantics have nothing to do with the const qualification of a member function, which is referring to the type of object pointed to by `this` which is specified with the first element in the sequence passed to the adapter).

If the target function is a pointer to a members function, the corresponding object can be specified as a reference, pointer, or smart pointer. In case of the latter, a freestanding `get_pointer` function must be defined (Boost provides this function for `std::auto_ptr` and `boost::shared_ptr`).

The target function must not be a pointer to a member object (dereferencing such a pointer without returning anything does not make sense, so this case is not implemented).

Header

```
#include <boost/fusion/functional/adapter/fused_procedure.hpp>
```

Synopsis

```
template <typename Function>
class fused_procedure;
```

Template parameters

Parameter	Description	Default
Function	Callable Object type	

Model of

- [Polymorphic Function Object](#)
- [Deferred Callable Object](#)

Notation

- R A possibly const qualified [Callable Object](#) type or reference type thereof
- r An object convertible to R
- s A [Sequence](#) of arguments that are accepted by r
- f An instance of `fused<R>`

Expression Semantics

Expression	Semantics
<code>fused_procedure<R>(r)</code>	Creates a fused function as described above, initializes the target function with <code>r</code> .
<code>fused_procedure<R>()</code>	Creates a fused function as described above, attempts to use <code>R</code> 's default constructor.
<code>f(s)</code>	Calls <code>r</code> with the elements in <code>s</code> as its arguments.

Example

```
template<class SequenceOfSequences, class Func>
void n_ary_for_each(SequenceOfSequences const & s, Func const & f)
{
    for_each(zip_view<SequenceOfSequences>(s),
             fused_procedure<Func const &>(f));
}

void try_it()
{
    vector<int,float> a(2,2.0f);
    vector<int,float> b(1,1.5f);
    using namespace boost::lambda;
    n_ary_for_each(vector_tie(a,b), _1 == _2);
    assert(a == make_vector(1,0.5f));
}
```

See also

- [fused](#)
- [fused_function_object](#)
- [invoke_procedure](#)
- [make_fused_procedure](#)

fused_function_object

Description

An unary [Polymorphic Function Object](#) adapter template for a [Polymorphic Function Object](#) target function. It takes a [Forward Sequence](#) that contains the arguments for the target function.

The type of the target function is allowed to be const qualified or a reference. Const qualification is preserved and propagated appropriately (in other words, only const versions of `operator()` can be used for an target function object that is const or, if the target function object is held by value, the adapter is const).

Header

```
#include <boost/fusion/functional/adapters/fused_function_object.hpp>
```

Synopsis

```
template <class Function>
class fused_function_object;
```

Template parameters

Parameter	Description	Default
Function	Polymorphic Function Object type	

Model of

- [Polymorphic Function Object](#)
- [Deferred Callable Object](#)

Notation

R A possibly const qualified [Polymorphic Function Object](#) type or reference type thereof

r An object convertible to R

s A [Sequence](#) of arguments that are accepted by r

f An instance of `fused< R >`

Expression Semantics

Expression	Semantics
<code>fused_function_object<R>(r)</code>	Creates a fused function as described above, initializes the target function with r .
<code>fused_function_object<R>()</code>	Creates a fused function as described above, attempts to use R 's default constructor.
<code>f(s)</code>	Calls r with the elements in s as its arguments.

Example

```
template<class SeqOfSeqs, class Func>
typename result_of::transform< zip_view<SeqOfSeqs> const,
    fused_function_object<Func const &> >::type
n_ary_transform(SeqOfSeqs const & s, Func const & f)
{
    return transform(zip_view<SeqOfSeqs>(s),
        fused_function_object<Func const &>(f));
}

struct sub
{
    template <typename Sig>
    struct result;

    template <class Self, typename T>
    struct result< Self(T,T) >
    { typedef typename remove_reference<T>::type type; };

    template<typename T>
    T operator()(T lhs, T rhs) const
    {
        return lhs - rhs;
    }
};

void try_it()
{
    vector<int,float> a(2,2.0f);
    vector<int,float> b(1,1.5f);
    vector<int,float> c(1,0.5f);
    assert(c == n_ary_transform(vector_tie(a,b), sub()));
}
```

See also

- [fused](#)
- [fused_procedure](#)
- [invoke_function_object](#)
- [make_fused_function_object](#)
- [deduce](#)

unfused

Description

An n-ary [Polymorphic Function Object](#) adapter template for an unary [Polymorphic Function Object](#) target function. When called, its arguments are bundled to a [Random Access Sequence](#) of references that is passed to the target function object.

The nullary overload of the call operator can be removed by setting the second template parameter to `false`, which is very useful if the result type computation would result in a compile error, otherwise (nullary call operator's prototypes can't be templates and thus are instantiated as early as the class template).

Only LValue arguments are accepted. To overcome this limitation, apply [Boost.Functional/Forward](#).

The type of the target function is allowed to be const qualified or a reference. Const qualification is preserved and propagated appropriately. In other words, only const versions of `operator()` can be used if the target function object is const - or, in case the target function object is held by value, the adapter is const.

Header

```
#include <boost/fusion/functional/adapter/unfused.hpp>
```

Synopsis

```
template <class Function, bool AllowNullary = true>
class unfused;
```

Template parameters

Parameter	Description	Default
Function	A unary Polymorphic Function Object	
AllowNullary	Boolean constant	true

Model of

- [Polymorphic Function Object](#)
- [Deferred Callable Object](#)

Notation

F	A possibly const qualified, unary Polymorphic Function Object type or reference type thereof
f	An object convertible to F
UL	The type <code>unfused<F></code>
ul	An instance of UL , initialized with f
$a0...aN$	Arguments to ul

Expression Semantics

Expression	Semantics
<code>UL(f)</code>	Creates a fused function as described above, initializes the target function with f .
<code>UL()</code>	Creates a fused function as described above, attempts to use F 's default constructor.
<code>ul(a0...aN)</code>	Calls f with a Sequence that contains references to the arguments $a0...aN$.

Example

```
struct fused_incremter
{
    template <class Seq>
    struct result
    {
        typedef void type;
    };

    template <class Seq>
    void operator()(Seq const & s) const
    {
        for_each(s, ++boost::lambda::_1);
    }
};

void try_it()
{
    unfused<fused_incremter> increment;
    int a = 2; char b = 'X';
    increment(a,b);
    assert(a == 3 && b == 'Y');
}
```

See also

- [unfused_typed](#)
- [make_unfused](#)

unfused_typed

Description

An n-ary [Polymorphic Function Object](#) adapter template for an unary [Polymorphic Function Object](#) target function. When called, its arguments are bundled to a [Random Access Sequence](#) that is passed to the target function object.

The call operators of esulting function objects are strictly typed (in other words, non-templized) with the types from a [Sequence](#).

The type of the target function is allowed to be const qualified or a reference. Const qualification is preserved and propagated appropriately (in other words, only const versions of `operator()` can be used if the target function object is const - or, in case the target function object is held by value, the adapter is const).



For Microsoft Visual C++ 7.1 (Visual Studio 2003) the detection of the Function Object's const qualification easily causes an internal error. Therefore the adapter is always treated as if it was const.



If the type sequence passed to this template contains non-reference elements, the element is copied only once - the call operator's signature is optimized automatically to avoid by-value parameters.

Header

```
#include <boost/fusion/functional/adaptor/unfused_typed.hpp>
```

Synopsis

```
template <class Function, class Sequence>
class unfused_typed;
```

Template parameters

Parameter	Description	Default
Function	A unary Polymorphic Function Object	
Sequence	A Sequence	

Model of

- [Polymorphic Function Object](#)
- [Deferred Callable Object](#)

Notation

F	A possibly const qualified, unary Polymorphic Function Object type or reference type thereof
f	An object convertible to F
S	A Sequence of parameter types
UT	The type <code>unfused_typed<F, S></code>
ut	An instance of UT , initialized with f
$a0...aN$	Arguments to ut , convertible to the types in S

Expression Semantics

Expression	Semantics
<code>UT(f)</code>	Creates a fused function as described above, initializes the target function with f .
<code>UT()</code>	Creates a fused function as described above, attempts to use F 's default constructor.
<code>ut($a0...aN$)</code>	Calls f with an instance of S (or a subsequence of S starting at the first element, if fewer arguments are given and the overload hasn't been disabled) initialized with $a0...aN$.

Example

```

struct add_assign // applies operator+=
{
    typedef void result_type; // for simplicity

    template <typename T>
    void operator()(T & lhs, T const & rhs) const
    {
        lhs += rhs;
    }
};

template <class Tie>
class fused_parallel_adder
{
    Tie tie_dest;
public:
    explicit fused_parallel_adder(Tie const & dest)
        : tie_dest(dest)
    { }

    typedef void result_type;

    template <class Seq>
    void operator()(Seq const & s) const
    {
        for_each( zip(tie_dest,s), fused<add_assign>() );
    }
};

// accepts a tie and creates a typed function object from it
struct fused_parallel_adder_maker
{
    template <typename Sig>
    struct result;

    template <class Self, class Seq>
    struct result< Self(Seq) >
    {
        typedef typename remove_reference<Seq>::type seq;

        typedef unfused_typed< fused_parallel_adder<seq>,
            typename mpl::transform<seq, remove_reference<_> >::type > type;
    };

    template <class Seq>
    typename result< void(Seq) >::type operator()(Seq const & tie)
    {
        return typename result< void(Seq) >::type(
            fused_parallel_adder<Seq>(tie) );
    }
};

unfused<fused_parallel_adder_maker> parallel_add;

void try_it()
{
    int a = 2; char b = 'X';
    // the second call is strictly typed with the types deduced from the

```

```
// first call
parallel_add(a,b)(3,2);
parallel_add(a,b)(3);
parallel_add(a,b)();
assert(a == 8 && b == 'Z');
```

See also

- [unfused](#)
- [deduce](#)
- [deduce_sequence](#)

Limits

Header

```
#include <boost/fusion/functional/adapter/limits.hpp>
```

Macros

The following macros can be defined to change the maximum arity. The value used for these macros must not exceed `FUSION_MAX_VECTOR_SIZE`. The default is 6.

- `BOOST_FUSION_UNFUSED_MAX_ARITY`
- `BOOST_FUSION_UNFUSED_TYPE_MAX_ARITY`

Generation

Functions

`make_fused`

Description

Creates a `fused` adapter for a given [Deferred Callable Object](#). The usual *element conversion* is applied to the target function.

Synopsis

```
template <typename F>
inline typename make_fused<F>::type
make_fused(F const & f);
```

Parameters

Parameter	Requirement	Description
<code>f</code>	Model of Deferred Callable Object	The function to transform.

Expression Semantics

```
make_fused(f);
```

Return type: A specialization of [fused](#).

Semantics: Returns a [fused](#) adapter for `f`.

Header

```
#include <boost/fusion/functional/generation/make_fused.hpp>
#include <boost/fusion/include/make_fused.hpp>
```

Example

```
float sub(float a, float b) { return a - b; }

void try_it()
{
    vector<int,float> a(2,2.0f);
    vector<int,float> b(1,1.5f);
    vector<float,float> c(1.0f,0.5f);
    assert(c == transform(zip(a,b), make_fused(& sub)));
    assert(c == transform(zip(a,b), make_fused(std::minus<float>())));
}
```

See also

- [fused](#)
- [deduce](#)
- [make_fused](#)

make_fused_procedure

Description

Creates a [fused_procedure](#) adapter for a given [Deferred Callable Object](#). The usual *element conversion* applied to the target function.

Synopsis

```
template <typename F>
inline typename make_fused_procedure<F>::type
make_fused_procedure(F const & f);
```

Parameters

Parameter	Requirement	Description
<code>f</code>	Model of Callable Object	The function to transform.

Expression Semantics

```
make_fused_procedure(f);
```

Return type: A specialization of `fused_procedure`.

Semantics: Returns a `fused_procedure` adapter for `f`.

Header

```
#include <boost/fusion/functional/generation/make_fused_procedure.hpp>
#include <boost/fusion/include/make_fused_procedure.hpp>
```

Example

```
vector<int,int,int> v(1,2,3);
using namespace boost::lambda;
make_fused_procedure(_1 += _2 - _3)(v);
assert(front(v) == 0);
```

See also

- `fused_procedure`
- `deduce`
- `make_fused_procedure`

make_fused_function_object

Description

Creates a `fused_function_object` adapter for a given [Deferred Callable Object](#). The usual *element conversion* is applied to the target function.

Synopsis

```
template <typename F>
inline typename make_fused_function_object<F>::type
make_fused_function_object(F const & f);
```

Parameters

Parameter	Requirement	Description
<code>f</code>	Model of Polymorphic Function Object	The function to transform.

Expression Semantics

```
make_fused_function_object(f);
```

Return type: A specialization of `fused_function_object`.

Semantics: Returns a `fused_function_object` adapter for `f`.

Header

```
#include <boost/fusion/functional/generation/make_fused_function_object.hpp>
#include <boost/fusion/include/make_fused_function_object.hpp>
```

Example

```
struct sub
{
    template <typename Sig>
    struct result;

    template <class Self, typename T>
    struct result< Self(T,T) >
    { typedef typename remove_reference<T>::type type; };

    template<typename T>
    T operator()(T lhs, T rhs) const
    {
        return lhs - rhs;
    }
};

void try_it()
{
    vector<int,float> a(2,2.0f);
    vector<int,float> b(1,1.5f);
    vector<int,float> c(1,0.5f);
    assert(c == transform(zip(a,b), make_fused_function_object(sub())));
}
```

See also

- [fused_function_object](#)
- [deduce](#)
- [make_fused_function_object](#)

make_unfused

Description

Creates a `unfused` adapter for a given, unary [Polymorphic Function Object](#). The usual *element conversion* is applied to the target function.

Synopsis

```
template <typename F>
inline typename make_unfused<F>::type
make_unfused(F const & f);
```

Parameters

Parameter	Requirement	Description
<code>f</code>	Model of Polymorphic Function Object	The function to transform.

Expression Semantics

```
make_unfused(f);
```

Return type: A specialization of [unfused](#).

Semantics: Returns a [unfused](#) adapter for `f`.

Header

```
#include <boost/fusion/functional/generation/make_unfused.hpp>
#include <boost/fusion/include/make_unfused.hpp>
```

Example

```
struct fused_incrementer
{
    template <class Seq>
    struct result
    {
        typedef void type;
    };

    template <class Seq>
    void operator()(Seq const & s) const
    {
        for_each(s, ++boost::lambda::_1);
    }
};

void try_it()
{
    int a = 2; char b = 'X';
    make_unfused(fused_incrementer())(a,b);
    assert(a == 3 && b == 'Y');
}
```

See also

- [unfused](#)
- [deduce](#)
- [make_unfused](#)

Metafunctions

make_fused

Description

Returns the result type of [make_fused](#).

Header

```
#include <boost/fusion/functional/generation/make_fused.hpp>
#include <boost/fusion/include/make_fused.hpp>
```

Synopsis

```
namespace result_of
{
    template<typename Function>
    struct make_fused
    {
        typedef unspecified type;
    };
}
```

See also

- [make_fused](#)

make_fused_procedure

Description

Returns the result type of [make_fused_procedure](#).

Header

```
#include <boost/fusion/functional/generation/make_fused_procedure.hpp>
#include <boost/fusion/include/make_fused_procedure.hpp>
```

Synopsis

```
namespace result_of
{
    template<typename Function>
    struct make_fused_procedure
    {
        typedef unspecified type;
    };
}
```

See also

- [make_fused_procedure](#)

make_fused_function_object

Description

Returns the result type of [make_fused_function_object](#).

Header

```
#include <boost/fusion/functional/generation/make_fused_function_object.hpp>
#include <boost/fusion/include/make_fused_function_object.hpp>
```

Synopsis

```
namespace result_of
{
    template<typename Function>
    struct make_fused_function_object
    {
        typedef unspecified type;
    };
}
```

See also

- [make_fused_function_object](#)

make_unfused

Description

Returns the result type of [make_unfused](#).

Header

```
#include <boost/fusion/functional/generation/make_unfused.hpp>
#include <boost/fusion/include/make_unfused.hpp>
```

Synopsis

```
namespace result_of
{
    template<typename Function>
    struct make_unfused
    {
        typedef unspecified type;
    };
}
```

See also

- [make_unfused](#)

Notes

Recursive Inlined Functions

An interesting peculiarity of functions like [at](#) when applied to a [Forward Sequence](#) like [list](#) is that what could have been linear runtime complexity effectively becomes constant $O(1)$ due to compiler optimization of C++ inlined functions, however deeply recursive (up to a certain compiler limit of course). Compile time complexity remains linear.

Overloaded Functions

Associative sequences use function overloading to implement membership testing and type associated key lookup. This amounts to constant runtime and amortized constant compile time complexities. There is an overloaded function, $f(k)$, for each key *type* k . The compiler chooses the appropriate function given a key, k .

Tag Dispatching

Tag dispatching is a generic programming technique for selecting template specializations. There are typically 3 components involved in the tag dispatching mechanism:

1. A type for which an appropriate template specialization is required
2. A metafunction that associates the type with a tag type
3. A template that is specialized for the tag type

For example, the fusion `result_of::begin` metafunction is implemented as follows:

```
template <typename Sequence>
struct begin
{
    typedef typename
        result_of::begin_impl<typename traits::tag_of<Sequence>::type>::
        template apply<Sequence>::type
        type;
};
```

In the case:

1. `Sequence` is the type for which a suitable implementation of `result_of::begin_impl` is required
2. `traits::tag_of` is the metafunction that associates `Sequence` with an appropriate tag
3. `result_of::begin_impl` is the template which is specialized to provide an implementation for each tag type

Extensibility

Unlike [MPL](#), there is no extensible sequence concept in fusion. This does not mean that Fusion sequences are not extensible. In fact, all Fusion sequences are inherently extensible. It is just that the manner of sequence extension in Fusion is different from both [STL](#) and [MPL](#) on account of the lazy nature of fusion [Algorithms](#). [STL](#) containers extend themselves in place through member functions such as `push_back` and `insert`. [MPL](#) sequences, on the other hand, are extended through "intrinsic" functions that actually return whole sequences. [MPL](#) is purely functional and can not have side effects. For example, [MPL](#)'s `push_back` does not actually mutate an `mpl::vector`. It can't do that. Instead, it returns an extended `mpl::vector`.

Like [MPL](#), Fusion too is purely functional and can not have side effects. With runtime efficiency in mind, Fusion sequences are extended through generic functions that return [Views](#). [Views](#) are sequences that do not actually contain data, but instead impart an alternative presentation over the data from one or more underlying sequences. [Views](#) are proxies. They provide an efficient yet purely functional way to work on potentially expensive sequence operations. For example, given a `vector`, Fusion's `push_back` returns a `joint_view`, instead of an actual extended `vector`. A `joint_view` holds a reference to the original sequence plus the appended data --making it very cheap to pass around.

Element Conversion

Functions that take in elemental values to form sequences (e.g. `make_list`) convert their arguments to something suitable to be stored as a sequence element. In general, the element types are stored as plain values. Example:

```
make_list(1, 'x')
```

returns a `list<int, char>`.

There are a few exceptions, however.

Arrays:

Array arguments are deduced to reference to const types. For example ¹⁰:

```
make_list("Donald", "Daisy")
```

creates a `list` of type

```
list<const char (&)[7], const char (&)[6]>
```

Function pointers:

Function pointers are deduced to the plain non-reference type (i.e. to plain function pointer). Example:

```
void f(int i);
...
make_list(&f);
```

creates a `list` of type

```
list<void (*)(int)>
```

boost::ref

Fusion's generation functions (e.g. `make_list`) by default stores the element types as plain non-reference types. Example:

```
void foo(const A& a, B& b) {
    ...
    make_list(a, b)
```

creates a `list` of type

```
list<A, B>
```

Sometimes the plain non-reference type is not desired. You can use `boost::ref` and `boost::cref` to store references or const references (respectively) instead. The mechanism does not compromise const correctness since a const object wrapped with `ref` results in a tuple element with const reference type (see the fifth code line below). Examples:

For example:

¹⁰ Note that the type of a string literal is an array of const characters, not `const char*`. To get `make_list` to create a `list` with an element of a non-const array type one must use the `ref` wrapper (see `boost::ref`).

```
A a; B b; const A ca = a;
make_list(cref(a), b);           // creates list<const A&, B>
make_list(ref(a), b);            // creates list<A&, B>
make_list(ref(a), cref(b));      // creates list<A&, const B&>
make_list(cref(ca));             // creates list<const A&>
make_list(ref(ca));              // creates list<const A&>
```

See [Boost.Ref](#) for details.

Change log

This section summarizes significant changes to the Fusion library.

- Sep 27, 2006: Added `boost::tuple` support. (Joel de Guzman)
- Nov 17, 2006: Added `boost::variant` support. (Joel de Guzman)
- Feb 15, 2007: Added functional module. (Tobias Schwinger)
- April 2, 2007: Added struct adapter. (Joel de Guzman)
- May 8, 2007: Added associative struct adapter. (Dan Marsden)
- Dec 20, 2007: Removed `boost::variant` support. After thorough investigation, I think now that the move to make variant a fusion sequence is rather quirky. A variant will always have a `size==1` regardless of the number of types it can contain and there's no way to know at compile time what it contains. Iterating over its types is simply wrong. All these imply that the variant is **not** a fusion sequence. (Joel de Guzman)
- Oct 12, 2009: The accumulator is the first argument to the functor of `fold` and `accumulate`. (Christopher Schmidt)
- Oct 30, 2009: Added support for associative iterators and views. (Christopher Schmidt)

Acknowledgements

Special thanks to David Abrahams, Douglas Gregor, Hartmut Kaiser, Aleksey Gurtovoy, Peder Holt, Daniel Wallin, Jaakko Jarvi, Jeremiah Willcock, Dan Marsden, Eric Niebler, Joao Abecasis and Andy Little. These people are instrumental in the design and development of Fusion.

Special thanks to Ronald Garcia, the review manager and to all the people in the boost community who participated in the review: Andreas Pokorny, Andreas Huber, Jeff Flinn, David Abrahams, Pedro Lamarao, Larry Evans, Ryan Gallagher, Andy Little, Gennadiy Rozental, Tobias Schwinger, Joao Abecasis, Eric Niebler, Oleg Abrosimov, Gary Powell, Eric Friedman, Darren Cook, Martin Bonner and Douglas Gregor.

References

1. [New Iterator Concepts](#), David Abrahams, Jeremy Siek, Thomas Witt, 2004-11-01.
2. [The Boost Tuple Library](#), Jaakko Jarvi, 2001.
3. [Spirit Parser Library](#), Joel de Guzman, 2001-2006.
4. [The Boost MPL Library](#), Aleksey Gurtovoy and David Abrahams, 2002-2004.
5. [Boost Array](#), Nicolai Josuttis, 2002-2004.
6. [Standard Template Library Programmer's Guide](#), Hewlett-Packard Company, 1994.
7. [Boost.Ref](#), Jaakko Jarvi, Peter Dimov, Douglas Gregor, Dave Abrahams, 1999-2002.