# The MPL Reference Manual

Revision Date: 15th November 2004

# Contents

# Chapter 1 Sequences

Compile-time sequences of types are one of the basic concepts of C++ template metaprogramming. Differences in types of objects being manipulated is the most common point of variability of similar, but not identical designs, and these are a direct target for metaprogramming. Templates were originally designed to address this exact problem. However, without predefined mechanisms for representing and manipulating *sequences* of types as opposed to standalone template parameters, high-level template metaprogramming is severely limited in its capabitilies.

The MPL recognizes the importance of type sequences as a fundamental building block of many higher-level metaprogramming designs by providing us with a conceptual framework for formal reasoning and understanding of sequence properties, guarantees and characteristics, as well as a first-class implementation of that framework — a wealth of tools for concise, convenient, conceptually precise and efficient sequence manipulation.

## 1.1 Concepts

The taxonomy of sequence concepts in MPL parallels the taxonomy of the MPL Iterators, with two additional classification dimensions: *extensibility* and *associativeness*.

### 1.1.1 Forward Sequence

#### Description

A Forward Sequence is an MPL concept representing a compile-time sequence of elements. Sequence elements are types, and are accessible through Iterators. The `begin` and `end` metafunctions provide iterators delimiting the range of the sequence elements. A sequence guarantees that its elements are arranged in a definite, but possibly unspecified, order. Every MPL sequence is a Forward Sequence.

#### Definitions

— The *size* of a sequence is the number of elements it contains. The size is a nonnegative number.

— A sequence is *empty* if its size is zero.

#### Expression requirements

For any Forward Sequence `s` the following expressions must be valid:

| Expression | Type | Complexity |
|---|---|---|
| `begin<s>::type` | Forward Iterator | Amortized constant time |
| `end<s>::type` | Forward Iterator | Amortized constant time |
| `size<s>::type` | Integral Constant | Unspecified |
| `empty<s>::type` | Boolean Integral Constant | Constant time |
| `front<s>::type` | Any type | Amortized constant time |

#### Expression semantics

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

#### Invariants

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

— [`begin<s>::type`, `end<s>::type`) is always a valid range.

— An algorithm that iterates through the range [`begin<s>::type`, `end<s>::type`) will pass through every element of `s` exactly once.

— `begin<s>::type` is identical to `end<s>::type` if and only if `s` is empty.

— Two different iterations through `s` will access its elements in the same order.

#### Models

— `vector`

— `map`

— `range_c`

— `iterator_range`

— `filter_view`

#### See also

Sequences, Bidirectional Sequence, Forward Iterator, `begin` / `end`, `size`, `empty`, `front`

### 1.1.2 Bidirectional Sequence

#### Description

A Bidirectional Sequence is a Forward Sequence whose iterators model Bidirectional Iterator.

#### Refinement of

Forward Sequence

#### Expression requirements

In addition to the requirements defined in Forward Sequence, for any Bidirectional Sequence `s` the following must be met:

| Expression | Type | Complexity |
|---|---|---|
| `begin<s>::type` | Bidirectional Iterator | Amortized constant time |
| `end<s>::type` | Bidirectional Iterator | Amortized constant time |
| `back<s>::type` | Any type | 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 |
|---|---|
| `back<s>::type` | The last element in the sequence; see `back`. |

#### Models

— `vector`

— `range_c`

#### See also

Sequences, Forward Sequence, Random Access Sequence, Bidirectional Iterator, `begin` / `end`, `back`

### 1.1.3 Random Access Sequence

#### Description

A Random Access Sequence is a Bidirectional Sequence whose iterators model Random Access Iterator. A random access sequence guarantees amortized constant time access to an arbitrary sequence element.

#### Refinement of

Bidirectional Sequence

#### Expression requirements

In addition to the requirements defined in Bidirectional Sequence, for any Random Access Sequence `s` the following must be met:

| Expression | Type | Complexity |
|---|---|---|
| `begin<s>::type` | Random Access Iterator | Amortized constant time |
| `end<s>::type` | Random Access Iterator | Amortized constant time |
| `at<s,n>::type` | Any type | Amortized constant time |

#### Expression semantics

Semantics of an expression is defined only where it differs from, or is not defined in Bidirectional Sequence.

| Expression | Semantics |
|---|---|

| Expression | Semantics |
|---|---|
| `at<s,n>::type` | The nth element from the beginning of the sequence; see `at`. |

**Models**

— `vector`

— `range_c`

**See also**

Sequences, Bidirectional Sequence, Extensible Sequence, Random Access Iterator, `begin` / `end`, `at`

### 1.1.4 Extensible Sequence

**Description**

An Extensible Sequence is a sequence that supports insertion and removal of elements. Extensibility is orthogonal to sequence traversal characteristics.

**Expression requirements**

For any Extensible Sequence `s`, its iterators `pos` and `last`, Forward Sequence `r`, and any type `x`, the following expressions must be valid:

| Expression | Type | Complexity |
|---|---|---|
| `insert<s,pos,x>::type` | Extensible Sequence | Unspecified |
| `insert_range<s,pos,r>::type` | Extensible Sequence | Unspecified |
| `erase<s,pos>::type` | Extensible Sequence | Unspecified |
| `erase<s,pos,last>::type` | Extensible Sequence | Unspecified |
| `clear<s>::type` | Extensible Sequence | Constant time |

**Expression semantics**

| Expression | Semantics |
|---|---|
| `insert<s,pos,x>::type` | A new sequence, concept-identical to `s`, of the following elements: [`begin<s>::type`, `pos`), `x`, [`pos`, `end<s>::type`); see `insert`. |
| `insert_range<s,pos,r>::type` | A new sequence, concept-identical to `s`, of the following elements: [`begin<s>::type`, `pos`), [`begin<r>::type`, `end<r>::type`), [`pos`, `end<s>::type`); see `insert_range`. |
| `erase<s,pos>::type` | A new sequence, concept-identical to `s`, of the following elements: [`begin<s>::type`, `pos`), [`next<pos>::type`, `end<s>::type`); see `erase`. |
| `erase<s,pos,last>::type` | A new sequence, concept-identical to `s`, of the following elements: [`begin<s>::type`, `pos`), [`last`, `end<s>::type`); see `erase`. |
| `clear<s>::type` | An empty sequence concept-identical to `s`; see `clear`. |

**Models**

— vector

— list

**See also**

Sequences, Back Extensible Sequence, insert, insert_range, erase, clear

### 1.1.5 Front Extensible Sequence

**Description**

A Front Extensible Sequence is an Extensible Sequence that supports amortized constant time insertion and removal operations at the beginning.

**Refinement of**

Extensible Sequence

**Expression requirements**

In addition to the requirements defined in Extensible Sequence, for any Back Extensible Sequence s the following must be met:

| Expression | Type | Complexity |
|---|---|---|
| push_front<s,x>::type | Front Extensible Sequence | Amortized constant time |
| pop_front<s>::type | Front Extensible Sequence | Amortized constant time |
| front<s>::type | Any type | Amortized constant time |

**Expression semantics**

The semantics of an expression are defined only where they differ from, or are not defined in Extensible Sequence.

| Expression | Semantics |
|---|---|
| push_front<s,x>::type | Equivalent to insert<s,begin<s>::type,x>::type; see push_front. |
| pop_front<v>::type | Equivalent to erase<s,begin<s>::type>::type; see pop_front. |
| front<s>::type | The first element in the sequence; see front. |

**Models**

— vector

— list

**See also**

Sequences, Extensible Sequence, Back Extensible Sequence, push_front, pop_front, front

### 1.1.6 Back Extensible Sequence

#### Description

A Back Extensible Sequence is an Extensible Sequence that supports amortized constant time insertion and removal operations at the end.

#### Refinement of

Extensible Sequence

#### Expression requirements

In addition to the requirements defined in Extensible Sequence, for any Back Extensible Sequence `s` the following must be met:

| Expression | Type | Complexity |
|---|---|---|
| `push_back<s,x>::type` | Back Extensible Sequence | Amortized constant time |
| `pop_back<s>::type` | Back Extensible Sequence | Amortized constant time |
| `back<s>::type` | Any type | Amortized constant time |

#### Expression semantics

The semantics of an expression are defined only where they differ from, or are not defined in Extensible Sequence.

| Expression | Semantics |
|---|---|
| `push_back<s,x>::type` | Equivalent to `insert<s,end<s>::type,x>::type`; see `push_back`. |
| `pop_back<v>::type` | Equivalent to `erase<s,end<s>::type>::type`; see `pop_back`. |
| `back<s>::type` | The last element in the sequence; see `back`. |

#### Models

— `vector`

— `deque`

#### See also

Sequences, Extensible Sequence, Front Extensible Sequence, `push_back`, `pop_back`, `back`

### 1.1.7 Associative Sequence

#### Description

An Associative Sequence is a Forward Sequence that allows efficient retrieval of elements based on keys. Unlike associative containers in the C++ Standard Library, MPL associative sequences have no associated ordering relation. Instead, *type identity* is used to impose an equivalence relation on keys, and the order in which sequence elements are traversed during iteration is left unspecified.

### Definitions

— A *key* is a part of the element type used to identify and retrieve the element within the sequence.

— A *value* is a part of the element type retrievied from the sequence by its key.

### Expression requirements

In the following table and subsequent specifications, `s` is an Associative Sequence, `x` is a sequence element, and `k` and `def` are arbitrary types.

In addition to the requirements defined in Forward Sequence, the following must be met:

| Expression | Type | Complexity |
|---|---|---|
| `has_key<s,k>::type` | Boolean Integral Constant | Amortized constant time |
| `count<s,k>::type` | Integral Constant | Amortized constant time |
| `order<s,k>::type` | Integral Constant or `void_` | Amortized constant time |
| `at<s,k>::type` | Any type | Amortized constant time |
| `at<s,k,def>::type` | Any type | Amortized constant time |
| `key_type<s,x>::type` | Any type | Amortized constant time |
| `value_type<s,x>::type` | Any type | 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 |
|---|---|
| `has_key<s,k>::type` | A boolean Integral Constant `c` such that `c::value == true` if and only if there is one or more elements with the key `k` in `s`; see `has_key`. |
| `count<s,k>::type` | The number of elements with the key `k` in `s`; see `count`. |
| `order<s,k>::type` | A unique unsigned Integral Constant associated with the key `k` in the sequence `s`; see `order`. |
| `at<s,k>::type`<br>`at<s,k,def>::type` | The first element associated with the key `k` in the sequence `s`; see `at`. |
| `key_type<s,x>::type` | The key part of the element `x` that would be used to identify `x` in `s`; see `key_type`. |
| `value_type<s,x>::type` | The value part of the element `x` that would be used for `x` in `s`; see `value_type`. |

### Models

— `set`

— `map`

### See also

Sequences, Extensible Associative Sequence, `has_key`, `count`, `order`, `at`, `key_type`, `value_type`

### 1.1.8 Extensible Associative Sequence

#### Description

An Extensible Associative Sequence is an Associative Sequence that supports insertion and removal of elements. In contrast to Extensible Sequence, Extensible Associative Sequence does not provide a mechanism for inserting an element at a specific position.

#### Expression requirements

In the following table and subsequent specifications, `s` is an Associative Sequence, `pos` is an iterator into `s`, and `x` and `k` are arbitrary types.

In addition to the Associative Sequence requirements, the following must be met:

| Expression | Type | Complexity |
|---|---|---|
| `insert<s,x>::type` | Extensible Associative Sequence | Amortized constant time |
| `insert<s,pos,x>::type` | Extensible Associative Sequence | Amortized constant time |
| `erase_key<s,k>::type` | Extensible Associative Sequence | Amortized constant time |
| `erase<s,pos>::type` | Extensible Associative Sequence | Amortized constant time |
| `clear<s>::type` | Extensible Associative Sequence | Amortized constant time |

#### Expression semantics

The semantics of an expression are defined only where they differ from, or are not defined in Associative Sequence.

| Expression | Semantics |
|---|---|
| `insert<s,x>::type` | Inserts `x` into `s`; the resulting sequence `r` is equivalent to `s` except that `at< r, key_type<s,x>::type >::type` is identical to `value_type<s,x>::type`; see `insert`. |
| `insert<s,pos,x>::type` | Equivalent to `insert<s,x>::type`; `pos` is ignored; see `insert`. |
| `erase_key<s,k>::type` | Erases elements in `s` associated with the key `k`; the resulting sequence `r` is equivalent to `s` except that `has_key<r,k>::value == false`; see `erase_key`. |
| `erase<s,pos>::type` | Erases the element at a specific position; equivalent to `erase_key<s, deref<pos>::type >::type`; see `erase`. |
| `clear<s>::type` | An empty sequence concept-identical to `s`; see `clear`. |

#### Models

— `set`

— `map`

#### See also

Sequences, Associative Sequence, `insert`, `erase`, `clear`

### 1.1.9 Integral Sequence Wrapper

#### Description

An Integral Sequence Wrapper is a class template that provides a concise interface for creating a corresponding sequence of Integral Constants. In particular, assuming that `seq` is a name of the wrapper's underlying sequence and $c_1$,$c_2$,... $c_n$ are integral constants of an integral type `T` to be stored in the sequence, the wrapper provides us with the following notation:

`seq_c<T,`$c_1$`,`$c_2$`,... `$c_n$`>`

If `seq` is a Variadic Sequence, *numbered* wrapper forms are also avaialable:

`seq`*n*`_c<T,`$c_1$`,`$c_2$`,... `$c_n$`>`

#### Expression requirements

In the following table and subsequent specifications, `seq` is a placeholder token for the Integral Sequence Wrapper's underlying sequence's name.

| Expression | Type | Complexity |
|---|---|---|
| `seq_c<T,`$c_1$`,`$c_2$`,... `$c_n$`>` | Forward Sequence | Amortized constant time. |
| `seq_c<T,`$c_1$`,`$c_2$`,... `$c_n$`>::type` | Forward Sequence | Amortized constant time. |
| `seq_c<T,`$c_1$`,`$c_2$`,... `$c_n$`>::value_type` | An integral type | Amortized constant time. |
| `seq`*n*`_c<T,`$c_1$`,`$c_2$`,... `$c_n$`>` | Forward Sequence | Amortized constant time. |
| `seq`*n*`_c<T,`$c_1$`,`$c_2$`,... `$c_n$`>::type` | Forward Sequence | Amortized constant time. |
| `seq`*n*`_c<T,`$c_1$`,`$c_2$`,... `$c_n$`>::value_type` | An integral type | Amortized constant time. |

#### Expression semantics

```
typedef seq_c<T,c1,c2,... cn> s;
typedef seqn_c<T,c1,c2,... cn> s;
```

**Semantics:** `s` is a sequence `seq` of integral constant wrappers `integral_c<T,`$c_1$`>`, `integral_c<T,`$c_2$`>`, ... `integral_c<T,`$c_n$`>`.

**Postcondition:** `size<s>::value == n`.

```
typedef seq_c<T,c1,c2,... cn>::type s;
typedef seqn_c<T,c1,c2,... cn>::type s;
```

**Semantics:** `s` is identical to `seq`*n*`<integral_c<T,`$c_1$`>,integral_c<T,`$c_2$`>, ... integral_c<T,`$c_n$`> >`.

```
typedef seq_c<T,c1,c2,... cn>::value_type t;
typedef seqn_c<T,c1,c2,... cn>::value_type t;
```

**Semantics:** `is_same<t,T>::value == true`.

#### Models

- `vector_c`
- `list_c`
- `set_c`

**See also**

Sequences, Variadic Sequence, Integral Constant

### 1.1.10 Variadic Sequence

**Description**

A Variadic Sequence is a member of a family of sequence classes with both *variadic* and *numbered* forms. If `seq` is a generic name for some Variadic Sequence, its *variadic form* allows us to specify a sequence of *n* elements $t_1$,$t_2$,... $t_n$, for any *n* from 0 up to a preprocessor-configurable limit `BOOST_MPL_LIMIT_`*seq*`_SIZE`, using the following notation:

`seq<`$t_1$,$t_2$,... $t_n$`>`

By contrast, each *numbered* sequence form accepts the exact number of elements that is encoded in the name of the corresponding class template:

`seq`*n*`<`$t_1$,$t_2$,... $t_n$`>`

For numbered forms, there is no predefined top limit for *n*, aside from compiler limitations on the number of template parameters.

**Expression requirements**

In the following table and subsequent specifications, `seq` is a placeholder token for the actual Variadic Sequence name.

| Expression | Type | Complexity |
|---|---|---|
| `seq<`$t_1$,$t_2$,... $t_n$`>` | Forward Sequence | Amortized constant time |
| `seq<`$t_1$,$t_2$,... $t_n$`>::type` | Forward Sequence | Amortized constant time |
| `seq`*n*`<`$t_1$,$t_2$,... $t_n$`>` | Forward Sequence | Amortized constant time |
| `seq`*n*`<`$t_1$,$t_2$,... $t_n$`>::type` | Forward Sequence | Amortized constant time |

**Expression semantics**

```
typedef seq<t1,t2,... tn> s;
typedef seqn<t1,t2,... tn> s;
```

**Semantics:** `s` is a sequence of elements $t_1$,$t_2$,... $t_n$.

**Postcondition:** `size<s>::value == n`.

```
typedef seq<t1,t2,... tn>::type s;
typedef seqn<t1,t2,... tn>::type s;
```

**Semantics:** `s` is identical to `seq`*n*`<`$t_1$,$t_2$,... $t_n$ `>`.

**Postcondition:** `size<s>::value == n`.

**Models**

- `vector`
- `list`
- `map`

### See also

Sequences, Configuration, Integral Sequence Wrapper

## 1.2 Classes

The MPL provides a large number of predefined general-purpose sequence classes covering most of the typical metaprogramming needs out-of-box.

### 1.2.1 vector

#### Description

`vector` is a variadic, random access, extensible sequence of types that supports constant-time insertion and removal of elements at both ends, and linear-time insertion and removal of elements in the middle. On compilers that support the `typeof` extension, `vector` is the simplest and in many cases the most efficient sequence.

#### Header

| Sequence form | Header |
|---|---|
| Variadic | `#include <boost/mpl/vector.hpp>` |
| Numbered | `#include <boost/mpl/vector/vector`*n*`.hpp>` |

#### Model of

- Variadic Sequence
- Random Access Sequence
- Extensible Sequence
- Back Extensible Sequence
- Front Extensible Sequence

#### Expression semantics

In the following table, `v` is an instance of `vector`, `pos` and `last` are iterators into `v`, `r` is a Forward Sequence, `n` is an Integral Constant, and `x` and $t_1$,$t_2$,... $t_n$ are arbitrary types.

| Expression | Semantics |
|---|---|
| `vector<`$t_1$`,`$t_2$`,... `$t_n$`>`<br>`vector`*n*`<`$t_1$`,`$t_2$`,... `$t_n$`>` | `vector` of elements $t_1$,$t_2$,... $t_n$; see Variadic Sequence. |
| `vector<`$t_1$`,`$t_2$`,... `$t_n$`>::type`<br>`vector`*n*`<`$t_1$`,`$t_2$`,... `$t_n$`>::type` | Identical to `vector`*n*`<`$t_1$,$t_2$,... $t_n$`>`; see Variadic Sequence. |
| `begin<v>::type` | An iterator pointing to the beginning of `v`; see Random Access Sequence. |
| `end<v>::type` | An iterator pointing to the end of `v`; see Random Access Sequence. |
| `size<v>::type` | The size of `v`; see Random Access Sequence. |
| `empty<v>::type` | A boolean Integral Constant `c` such that `c::value == true` if and only if the sequence is empty; see Random Access Sequence. |

| Expression | Semantics |
|---|---|
| `front<v>::type` | The first element in `v`; see Random Access Sequence. |
| `back<v>::type` | The last element in `v`; see Random Access Sequence. |
| `at<v,n>::type` | The `n`th element from the beginning of `v`; see Random Access Sequence. |
| `insert<v,pos,x>::type` | A new `vector` of following elements: [`begin<v>::type`, `pos`), `x`, [`pos`, `end<v>::type`); see Extensible Sequence. |
| `insert_range<v,pos,r>::type` | A new `vector` of following elements: [`begin<v>::type`, `pos`), [`begin<r>::type`, `end<r>::type`) [`pos`, `end<v>::type`); see Extensible Sequence. |
| `erase<v,pos>::type` | A new `vector` of following elements: [`begin<v>::type`, `pos`), [`next<pos>::type`, `end<v>::type`); see Extensible Sequence. |
| `erase<v,pos,last>::type` | A new `vector` of following elements: [`begin<v>::type`, `pos`), [`last`, `end<v>::type`); see Extensible Sequence. |
| `clear<v>::type` | An empty `vector`; see Extensible Sequence. |
| `push_back<v,x>::type` | A new `vector` of following elements: [`begin<v>::type`, `end<v>::type`), `x`; see Back Extensible Sequence. |
| `pop_back<v>::type` | A new `vector` of following elements: [`begin<v>::type`, `prior< end<v>::type >::type`); see Back Extensible Sequence. |
| `push_front<v,x>::type` | A new `vector` of following elements: [`begin<v>::type`, `end<v>::type`), `x`; see Front Extensible Sequence. |
| `pop_front<v>::type` | A new `vector` of following elements: [`next< begin<v>::type >::type`, `end<v>::type`); see Front Extensible Sequence. |

### Example

```
typedef vector<float,double,long double> floats;
typedef push_back<floats,int>::type types;

BOOST_MPL_ASSERT(( is_same< at_c<types,3>::type, int > ));
```

### See also

Sequences, Variadic Sequence, Random Access Sequence, Extensible Sequence, `vector_c`, `list`

## 1.2.2 list

### Description

A `list` is a variadic, forward, extensible sequence of types that supports constant-time insertion and removal of elements at the beginning, and linear-time insertion and removal of elements at the end and in the middle.

### Header

| Sequence form | Header |
|---|---|
| Variadic | `#include <boost/mpl/list.hpp>` |
| Numbered | `#include <boost/mpl/list/listn.hpp>` |

### Model of

— Variadic Sequence

— Forward Sequence

— Extensible Sequence

— Front Extensible Sequence

### Expression semantics

In the following table, `l` is a `list`, `pos` and `last` are iterators into `l`, `r` is a Forward Sequence, and $t_1$,$t_2$,... $t_n$ and `x` are arbitrary types.

| Expression | Semantics |
|---|---|
| `list<`$t_1$`,`$t_2$`,... `$t_n$`>`<br>`list`$n$`<`$t_1$`,`$t_2$`,... `$t_n$`>` | `list` of elements $t_1$,$t_2$,... $t_n$; see Variadic Sequence. |
| `list<`$t_1$`,`$t_2$`,... `$t_n$`>::type`<br>`list`$n$`<`$t_1$`,`$t_2$`,... `$t_n$`>::type` | Identical to `list`$n$`<`$t_1$,$t_2$,... $t_n$`>`; see Variadic Sequence. |
| `begin<l>::type` | An iterator to the beginning of `l`; see Forward Sequence. |
| `end<l>::type` | An iterator to the end of `l`; see Forward Sequence. |
| `size<l>::type` | The size of `l`; see Forward Sequence. |
| `empty<l>::type` | A boolean Integral Constant `c` such that `c::value == true` if and only if `l` is empty; see Forward Sequence. |
| `front<l>::type` | The first element in `l`; see Forward Sequence. |
| `insert<l,pos,x>::type` | A new `list` of following elements: [`begin<l>::type`, `pos`), `x`, [`pos`, `end<l>::type`); see Extensible Sequence. |
| `insert_range<l,pos,r>::type` | A new `list` of following elements: [`begin<l>::type`, `pos`), [`begin<r>::type`, `end<r>::type`) [`pos`, `end<l>::type`); see Extensible Sequence. |
| `erase<l,pos>::type` | A new `list` of following elements: [`begin<l>::type`, `pos`), [`next<pos>::type`, `end<l>::type`); see Extensible Sequence. |
| `erase<l,pos,last>::type` | A new `list` of following elements: [`begin<l>::type`, `pos`), [`last`, `end<l>::type`); see Extensible Sequence. |
| `clear<l>::type` | An empty `list`; see Extensible Sequence. |
| `push_front<l,x>::type` | A new `list` containing `x` as its first element; see Front Extensible Sequence. |
| `pop_front<l>::type` | A new `list` containing all but the first elements of `l` in the same order; see Front Extensible Sequence. |

### Example

```
typedef list<float,double,long double> floats;
typedef push_front<floating_types,int>::type types;

BOOST_MPL_ASSERT(( is_same< front<types>::type, int > ));
```

**See also**

Sequences, Variadic Sequence, Forward Sequence, Extensible Sequence, `vector`, `list_c`

### 1.2.3 deque

**Description**

`deque` is a variadic, random access, extensible sequence of types that supports constant-time insertion and removal of elements at both ends, and linear-time insertion and removal of elements in the middle. In this implementation of the library, `deque` is a synonym for `vector`.

**Header**

```
#include <boost/mpl/deque.hpp>
```

**Model of**

- Variadic Sequence
- Random Access Sequence
- Extensible Sequence
- Back Extensible Sequence
- Front Extensible Sequence

**Expression semantics**

See `vector` specification.

**Example**

```
typedef deque<float,double,long double> floats;
typedef push_back<floats,int>::type types;

BOOST_MPL_ASSERT(( is_same< at_c<types,3>::type, int > ));
```

**See also**

Sequences, `vector`, `list`, `set`

### 1.2.4 set

**Description**

`set` is a variadic, associative, extensible sequence of types that supports constant-time insertion and removal of elements, and testing for membership. A `set` may contain at most one element for each key.

**Header**

| Sequence form | Header |
|---|---|
| Variadic | `#include <boost/mpl/set.hpp>` |
| Numbered | `#include <boost/mpl/set/setn.hpp>` |

### Model of

— Variadic Sequence

— Associative Sequence

— Extensible Associative Sequence

### Expression semantics

In the following table, `s` is an instance of `set`, `pos` is an iterator into `s`, and `x`, `k`, and $t_1$,$t_2$,... $t_n$ are arbitrary types.

| Expression | Semantics |
|---|---|
| `set<`$t_1$`,`$t_2$`,... `$t_n$`>`<br>`setn<`$t_1$`,`$t_2$`,... `$t_n$`>` | `set` of elements $t_1$,$t_2$,... $t_n$; see Variadic Sequence. |
| `set<`$t_1$`,`$t_2$`,... `$t_n$`>::type`<br>`setn<`$t_1$`,`$t_2$`,... `$t_n$`>::type` | Identical to `setn<`$t_1$,$t_2$,... $t_n$`>`; see Variadic Sequence. |
| `begin<s>::type` | An iterator pointing to the beginning of `s`; see Associative Sequence. |
| `end<s>::type` | An iterator pointing to the end of `s`; see Associative Sequence. |
| `size<s>::type` | The size of `s`; see Associative Sequence. |
| `empty<s>::type` | A boolean Integral Constant `c` such that `c::value == true` if and only if `s` is empty; see Associative Sequence. |
| `front<s>::type` | The first element in `s`; see Associative Sequence. |
| `has_key<s,k>::type` | A boolean Integral Constant `c` such that `c::value == true` if and only if there is one or more elements with the key `k` in `s`; see Associative Sequence. |
| `count<s,k>::type` | The number of elements with the key `k` in `s`; see Associative Sequence. |
| `order<s,k>::type` | A unique unsigned Integral Constant associated with the key `k` in `s`; see Associative Sequence. |
| `at<s,k>::type`<br>`at<s,k,def>::type` | The element associated with the key `k` in `s`; see Associative Sequence. |
| `key_type<s,x>::type` | Identical to `x`; see Associative Sequence. |
| `value_type<s,x>::type` | Identical to `x`; see Associative Sequence. |
| `insert<s,x>::type` | A new `set` equivalent to `s` except that `at< t, key_type<s,x>::type >::type` is identical to `value_type<s,x>::type`. |
| `insert<s,pos,x>::type` | Equivalent to `insert<s,x>::type`; `pos` is ignored. |
| `erase_key<s,k>::type` | A new `set` equivalent to `s` except that `has_key<t, k>::value == false`. |
| `erase<s,pos>::type` | Equivalent to `erase<s, deref<pos>::type >::type`. |
| `clear<s>::type` | An empty `set`; see `clear`. |

### Example

```
typedef set< int,long,double,int_<5> > s;

BOOST_MPL_ASSERT_RELATION( size<s>::value, ==, 4 );
BOOST_MPL_ASSERT_NOT(( empty<s> ));

BOOST_MPL_ASSERT(( is_same< at<s,int>::type, int > ));
BOOST_MPL_ASSERT(( is_same< at<s,long>::type, long > ));
BOOST_MPL_ASSERT(( is_same< at<s,int_<5> >::type, int_<5> > ));
BOOST_MPL_ASSERT(( is_same< at<s,char>::type, void_ > ));
```

### See also

Sequences, Variadic Sequence, Associative Sequence, Extensible Associative Sequence, `set_c`, `map`, `vector`

## 1.2.5 map

### Description

`map` is a variadic, associative, extensible sequence of type pairs that supports constant-time insertion and removal of elements, and testing for membership. A `map` may contain at most one element for each key.

### Header

| Sequence form | Header |
|---|---|
| Variadic | `#include <boost/mpl/map.hpp>` |
| Numbered | `#include <boost/mpl/map/mapn.hpp>` |

### Model of

- Variadic Sequence
- Associative Sequence
- Extensible Associative Sequence

### Expression semantics

In the following table and subsequent specifications, `m` is an instance of `map`, `pos` is an iterator into `m`, `x` and $p_1$,$p_2$,... $p_n$ are `pairs`, and `k` is an arbitrary type.

| Expression | Semantics |
|---|---|
| `map<`$p_1$`,`$p_2$`,... `$p_n$`>`<br>`mapn<`$p_1$`,`$p_2$`,... `$p_n$`>` | `map` of elements $p_1$,$p_2$,... $p_n$; see Variadic Sequence. |
| `map<`$p_1$`,`$p_2$`,... `$p_n$`>::type`<br>`mapn<`$p_1$`,`$p_2$`,... `$p_n$`>::type` | Identical to `mapn`<$p_1$,$p_2$,... $p_n$>; see Variadic Sequence. |
| `begin<m>::type` | An iterator pointing to the beginning of `m`; see Associative Sequence. |
| `end<m>::type` | An iterator pointing to the end of `m`; see Associative Sequence. |
| `size<m>::type` | The size of `m`; see Associative Sequence. |

| Expression | Semantics |
|---|---|
| `empty<m>::type` | A boolean Integral Constant `c` such that `c::value == true` if and only if `m` is empty; see Associative Sequence. |
| `front<m>::type` | The first element in `m`; see Associative Sequence. |
| `has_key<m,k>::type` | Queries the presence of elements with the key `k` in `m`; see Associative Sequence. |
| `count<m,k>::type` | The number of elements with the key `k` in `m`; see Associative Sequence. |
| `order<m,k>::type` | A unique unsigned Integral Constant associated with the key `k` in `m`; see Associative Sequence. |
| `at<m,k>::type`<br>`at<m,k,default>::type` | The element associated with the key `k` in `m`; see Associative Sequence. |
| `key_type<m,x>::type` | Identical to `x::first`; see Associative Sequence. |
| `value_type<m,x>::type` | Identical to `x::second`; see Associative Sequence. |
| `insert<m,x>::type` | A new `map` equivalent to `m` except that `at< t, key_type<m,x>::type >::type` is identical to `value_type<m,x>::type`. |
| `insert<m,pos,x>::type` | Equivalent to `insert<m,x>::type`; `pos` is ignored. |
| `erase_key<m,k>::type` | A new `map` equivalent to `m` except that `has_key<t, k>::value == false`. |
| `erase<m,pos>::type` | Equivalent to `erase<m, deref<pos>::type >::type`. |
| `clear<m>::type` | An empty `map`; see `clear`. |

**Example**

```
typedef map<
      pair<int,unsigned>
    , pair<char,unsigned char>
    , pair<long_<5>,char[17]>
    , pair<int[42],bool>
    > m;

BOOST_MPL_ASSERT_RELATION( size<m>::value, ==, 4 );
BOOST_MPL_ASSERT_NOT(( empty<m> ));

BOOST_MPL_ASSERT(( is_same< at<m,int>::type, unsigned > ));
BOOST_MPL_ASSERT(( is_same< at<m,long_<5> >::type, char[17] > ));
BOOST_MPL_ASSERT(( is_same< at<m,int[42]>::type, bool > ));
BOOST_MPL_ASSERT(( is_same< at<m,long>::type, void_ > ));
```

**See also**

Sequences, Variadic Sequence, Associative Sequence, Extensible Associative Sequence, `set`, `vector`

### 1.2.6 range_c

**Synopsis**

```
template<
```

```
      typename T
    , T Start
    , T Finish
    >
struct range_c
{
    typedef integral_c<T,Start> start;
    typedef integral_c<T,Finish> finish;
    // unspecified
    // ...
};
```

### Description

`range_c` is a sorted Random Access Sequence of Integral Constants. Note that because it is not an Extensible Sequence, sequence-building intrinsic metafunctions such as `push_front` and transformation algorithms such as `replace` are not directly applicable — to be able to use them, you'd first need to copy the content of the range into a more suitable sequence.

### Header

```
#include <boost/mpl/range_c.hpp>
```

### Model of

Random Access Sequence

### Expression semantics

In the following table, `r` is an instance of `range_c`, `n` is an Integral Constant, `T` is an arbitrary integral type, and `n` and `m` are integral constant values of type `T`.

| Expression | Semantics |
|---|---|
| `range_c<T,n,m>`<br>`range_c<T,n,m>::type` | A sorted Random Access Sequence of integral constant wrappers for the half-open range of values [`n`, `m`): `integral_c<T,n>`, `integral_c<T,n+1>`,... `integral_c<T,m-1>`. |
| `begin<r>::type` | An iterator pointing to the beginning of `r`; see Random Access Sequence. |
| `end<r>::type` | An iterator pointing to the end of `r`; see Random Access Sequence. |
| `size<r>::type` | The size of `r`; see Random Access Sequence. |
| `empty<r>::type` | A boolean Integral Constant `c` such that `c::value == true` if and only if `r` is empty; see Random Access Sequence. |
| `front<r>::type` | The first element in `r`; see Random Access Sequence. |
| `back<r>::type` | The last element in `r`; see Random Access Sequence. |
| `at<r,n>::type` | The `n`th element from the beginning of `r`; see Random Access Sequence. |

### Example

```
typedef range_c<int,0,0> range0;
typedef range_c<int,0,1> range1;
```

```
typedef range_c<int,0,10> range10;

BOOST_MPL_ASSERT_RELATION( size<range0>::value, ==, 0 );
BOOST_MPL_ASSERT_RELATION( size<range1>::value, ==, 1 );
BOOST_MPL_ASSERT_RELATION( size<range10>::value, ==, 10 );

BOOST_MPL_ASSERT(( empty<range0> ));
BOOST_MPL_ASSERT_NOT(( empty<range1> ));
BOOST_MPL_ASSERT_NOT(( empty<range10> ));

BOOST_MPL_ASSERT(( is_same< begin<range0>::type, end<range0>::type > ));
BOOST_MPL_ASSERT_NOT(( is_same< begin<range1>::type, end<range1>::type > ));
BOOST_MPL_ASSERT_NOT(( is_same< begin<range10>::type, end<range10>::type > ));

BOOST_MPL_ASSERT_RELATION( front<range1>::type::value, ==, 0 );
BOOST_MPL_ASSERT_RELATION( back<range1>::type::value, ==, 0 );
BOOST_MPL_ASSERT_RELATION( front<range10>::type::value, ==, 0 );
BOOST_MPL_ASSERT_RELATION( back<range10>::type::value, ==, 9 );
```

**See also**

Sequences, Random Access Sequence, `vector_c`, `set_c`, `list_c`

### 1.2.7 vector_c

**Description**

`vector_c` is an Integral Sequence Wrapper for `vector`. As such, it shares all `vector` characteristics and requirements, and differs only in the way the original sequence content is specified.

**Header**

| Sequence form | Header |
|---|---|
| Variadic | `#include <boost/mpl/vector_c.hpp>` |
| Numbered | `#include <boost/mpl/vector/vector`*n*`_c.hpp>` |

**Model of**

- Integral Sequence Wrapper
- Variadic Sequence
- Random Access Sequence
- Extensible Sequence
- Back Extensible Sequence
- Front Extensible Sequence

**Expression semantics**

The semantics of an expression are defined only where they differ from, or are not defined in `vector`.

| Expression | Semantics |
|---|---|
| `vector_c<T,`$c_1$`,`$c_2$`,... `$c_n$`>`<br>`vector`$n$`_c<T,`$c_1$`,`$c_2$`,... `$c_n$`>` | A `vector` of integral constant wrappers `integral_c<T,`$c_1$`>`, `integral_c<T,`$c_2$`>`, ... `integral_c<T,`$c_n$`>`; see Integral Sequence Wrapper. |
| `vector_c<T,`$c_1$`,`$c_2$`,... `$c_n$`>::type`<br>`vector`$n$`_c<T,`$c_1$`,`$c_2$`,... `$c_n$`>::type` | Identical to `vector`$n$`< integral_c<T,`$c_1$`>, integral_c<T,`$c_2$`>, ... integral_c<T,`$c_n$`> >`; see Integral Sequence Wrapper. |
| `vector_c<T,`$c_1$`,`$c_2$`,... `$c_n$`>::value_type`<br>`vector`$n$`_c<T,`$c_1$`,`$c_2$`,... `$c_n$`>::value_type` | Identical to `T`; see Integral Sequence Wrapper. |

### Example

```
typedef vector_c<int,1,2,3,5,7,12,19,31> fibonacci;
typedef push_back<fibonacci,int_<50> >::type fibonacci2;

BOOST_MPL_ASSERT_RELATION( front<fibonacci2>::type::value, ==, 1 );
BOOST_MPL_ASSERT_RELATION( back<fibonacci2>::type::value, ==, 50 );
```

### See also

Sequences, Integral Sequence Wrapper, `vector`, `integral_c`, `set_c`, `list_c`, `range_c`

## 1.2.8 list_c

### Description

`list_c` is an Integral Sequence Wrapper for `list`. As such, it shares all `list` characteristics and requirements, and differs only in the way the original sequence content is specified.

### Header

| Sequence form | Header |
|---|---|
| Variadic | `#include <boost/mpl/list_c.hpp>` |
| Numbered | `#include <boost/mpl/list/list`$n$`_c.hpp>` |

### Model of

- Integral Sequence Wrapper
- Variadic Sequence
- Forward Sequence
- Extensible Sequence
- Front Extensible Sequence

### Expression semantics

The semantics of an expression are defined only where they differ from, or are not defined in `list`.

| Expression | Semantics |
|---|---|
| `list_c<T,`$c_1$`,`$c_2$`,... `$c_n$`>`<br>`list`$n$`_c<T,`$c_1$`,`$c_2$`,... `$c_n$`>` | A `list` of integral constant wrappers `integral_c<T,`$c_1$`>`, `integral_c<T,`$c_2$`>`, ... `integral_c<T,`$c_n$`>`; see Integral Sequence Wrapper. |
| `list_c<T,`$c_1$`,`$c_2$`,... `$c_n$`>::type`<br>`list`$n$`_c<T,`$c_1$`,`$c_2$`,... `$c_n$`>::type` | Identical to `list`$n$`< integral_c<T,`$c_1$`>, integral_c<T,`$c_2$`>, ... integral_c<T,`$c_n$`> >`; see Integral Sequence Wrapper. |
| `list_c<T,`$c_1$`,`$c_2$`,... `$c_n$`>::value_type`<br>`list`$n$`_c<T,`$c_1$`,`$c_2$`,... `$c_n$`>::value_type` | Identical to `T`; see Integral Sequence Wrapper. |

### Example

```
typedef list_c<int,1,2,3,5,7,12,19,31> fibonacci;
typedef push_front<fibonacci,int_<1> >::type fibonacci2;

BOOST_MPL_ASSERT_RELATION( front<fibonacci2>::type::value, ==, 1 );
```

### See also

Sequences, Integral Sequence Wrapper, `list`, `integral_c`, `vector_c`, `set_c`, `range_c`

## 1.2.9 set_c

### Description

`set_c` is an Integral Sequence Wrapper for `set`. As such, it shares all `set` characteristics and requirements, and differs only in the way the original sequence content is specified.

### Header

| Sequence form | Header |
|---|---|
| Variadic | `#include <boost/mpl/set_c.hpp>` |
| Numbered | `#include <boost/mpl/set/set`$n$`_c.hpp>` |

### Model of

- Variadic Sequence
- Associative Sequence
- Extensible Associative Sequence

### Expression semantics

The semantics of an expression are defined only where they differ from, or are not defined in `set`.

| Expression | Semantics |
|---|---|
| `set_c<T,`$c_1$`,`$c_2$`,... `$c_n$`>`<br>`set`$n$`_c<T,`$c_1$`,`$c_2$`,... `$c_n$`>` | A `set` of integral constant wrappers `integral_c<T,`$c_1$`>`, `integral_c<T,`$c_2$`>`, ... `integral_c<T,`$c_n$`>`; see Integral Sequence Wrapper. |

| Expression | Semantics |
|---|---|
| set_c<T,$c_1$,$c_2$,... $c_n$>::type<br>set$n$_c<T,$c_1$,$c_2$,... $c_n$>::type | Identical to set$n$< integral_c<T,$c_1$>, integral_c<T,$c_2$>, ... integral_c<T,$c_n$> >; see Integral Sequence Wrapper. |
| set_c<T,$c_1$,$c_2$,... $c_n$>::value_type<br>set$n$_c<T,$c_1$,$c_2$,... $c_n$>::value_type | Identical to T; see Integral Sequence Wrapper. |

### Example

```
typedef set_c< int,1,3,5,7,9 > odds;

BOOST_MPL_ASSERT_RELATION( size<odds>::value, ==, 5 );
BOOST_MPL_ASSERT_NOT(( empty<odds> ));

BOOST_MPL_ASSERT(( has_key< odds, integral_c<int,5> > ));
BOOST_MPL_ASSERT_NOT(( has_key< odds, integral_c<int,4> > ));
BOOST_MPL_ASSERT_NOT(( has_key< odds, integral_c<int,15> > ));
```

### See also

Sequences, Integral Sequence Wrapper, set, integral_c, vector_c, list_c, range_c

## 1.3 Views

A *view* is a sequence adaptor delivering an altered presentation of one or more underlying sequences. Views are lazy, meaning that their elements are only computed on demand. Similarly to the short-circuit logical operations and eval_if, views make it possible to avoid premature errors and inefficiencies from computations whose results will never be used. When approached with views in mind, many algorithmic problems can be solved in a simpler, more conceptually precise, more expressive way.

### 1.3.1 empty_sequence

### Synopsis

```
struct empty_sequence
{
    // unspecified
    // ...
};
```

### Description

Represents a sequence containing no elements.

### Header

```
#include <boost/mpl/empty_sequence.hpp>
```

### Expression semantics

The semantics of an expression are defined only where they differ from, or are not defined in Random Access Sequence.

In the following table, `s` is an instance of `empty_sequence`.

| Expression | Semantics |
|---|---|
| `empty_sequence` | An empty Random Access Sequence. |
| `size<s>::type` | `size<s>::value == 0`; see Random Access Sequence. |

### Example

```
typedef begin<empty_sequence>::type first;
typedef end<empty_sequence>::type last;

BOOST_MPL_ASSERT(( is_same<first,last> ));
BOOST_MPL_ASSERT_RELATION( size<empty_sequence>::value, ==, 0 );

typedef transform_view<
      empty_sequence
    , add_pointer<_>
    > empty_view;

BOOST_MPL_ASSERT_RELATION( size<empty_sequence>::value, ==, 0 );
```

### See also

Sequences, Views, `vector`, `list`, `single_view`

## 1.3.2 filter_view

### Synopsis

```
template<
      typename Sequence
    , typename Pred
    >
struct filter_view
{
    // unspecified
    // ...
};
```

### Description

A view into a subset of `Sequence`'s elements satisfying the predicate `Pred`.

### Header

```
#include <boost/mpl/filter_view.hpp>
```

### Model of

— Forward Sequence

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Forward Sequence | A sequence to wrap. |
| `Pred` | Unary Lambda Expression | A filtering predicate. |

### Expression semantics

Semantics of an expression is defined only where it differs from, or is not defined in Forward Sequence.

In the following table, `v` is an instance of `filter_view`, `s` is an arbitrary Forward Sequence, `pred` is an unary Lambda Expression.

| Expression | Semantics |
|---|---|
| `filter_view<s,pred>`<br>`filter_view<s,pred>::type` | A lazy Forward Sequence sequence of all the elements in the range [`begin<s>::type`, `end<s>::type`) that satisfy the predicate `pred`. |
| `size<v>::type` | The size of `v`; `size<v>::value == count_if<s,pred>::value`; linear complexity; see Forward Sequence. |

### Example

Find the largest floating type in a sequence.

```
typedef vector<int,float,long,float,char[50],long double,char> types;
typedef max_element<
      transform_view< filter_view< types,boost::is_float<_> >, size_of<_> >
    >::type iter;

BOOST_MPL_ASSERT(( is_same< deref<iter::base>::type, long double > ));
```

### See also

Sequences, Views, `transform_view`, `joint_view`, `zip_view`, `iterator_range`

## 1.3.3 iterator_range

### Synopsis

```
template<
      typename First
    , typename Last
    >
struct iterator_range
{
    // unspecified
    // ...
```

```
};
```

### Description

A view into subset of sequence elements identified by a pair of iterators.

### Header

```
#include <boost/mpl/fold.hpp>
```

### Model of

— Forward, Bidirectional, or Random Access Sequence, depending on the category of the underlaying iterators.

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `First`, `Last` | Forward Iterator | Iterators identifying the view's boundaries. |

### Expression semantics

The semantics of an expression are defined only where they differ from, or are not defined in Forward Sequence.

In the following table, `v` is an instance of `iterator_range`, `first` and `last` are iterators into a Forward Sequence, and [`first`, `last`) form a valid range.

| Expression | Semantics |
|---|---|
| `iterator_range<first,last>`<br>`iterator_range<first,last>::type` | A lazy sequence all the elements in the range [`first`, `last`). |

### Example

```
typedef range_c<int,0,100> r;
typedef advance_c< begin<r>::type,10 >::type first;
typedef advance_c< end<r>::type,-10 >::type last;

BOOST_MPL_ASSERT(( equal<
      iterator_range<first,last>
    , range_c<int,10,90>
    > ));
```

### See also

Sequences, Views, `filter_view`, `transform_view`, `joint_view`, `zip_view`, `max_element`

## 1.3.4 joint_view

### Synopsis

```
template<
```

```
      typename Sequence1
    , typename Sequence2
    >
struct joint_view
{
    // unspecified
    // ...
};
```

### Description

A view into the sequence of elements formed by concatenating `Sequence1` and `Sequence2` elements.

### Header

```
#include <boost/mpl/joint_view.hpp>
```

### Model of

— Forward Sequence

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence1`, `Sequence2` | Forward Sequence | Sequences to create a view on. |

### Expression semantics

The semantics of an expression are defined only where they differ from, or are not defined in Forward Sequence.

In the following table, `v` is an instance of `joint_view`, `s1` and `s2` are arbitrary Forward Sequences.

| Expression | Semantics |
|---|---|
| `joint_view<s1,s2>`<br>`joint_view<s1,s2>::type` | A lazy Forward Sequence of all the elements in the ranges [`begin<s1>::type`, `end<s1>::type`), [`begin<s2>::type`, `end<s2>::type`). |
| `size<v>::type` | The size of `v`; `size<v>::value == size<s1>::value + size<s2>::value`; linear complexity; see Forward Sequence. |

### Example

```
typedef joint_view<
      range_c<int,0,10>
    , range_c<int,10,15>
    > numbers;

BOOST_MPL_ASSERT(( equal< numbers, range_c<int,0,15> > ));
```

### See also

Sequences, Views, `filter_view`, `transform_view`, `zip_view`, `iterator_range`

### 1.3.5 single_view

### Synopsis

```
template<
      typename T
    >
struct single_view
{
    // unspecified
    // ...
};
```

### Description

A view onto an arbitrary type `T` as on a single-element sequence.

### Header

```
#include <boost/mpl/single_view.hpp>
```

### Model of

— Random Access Sequence

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `T` | Any type | The type to be wrapped in a sequence. |

### Expression semantics

The semantics of an expression are defined only where they differ from, or are not defined in Random Access Sequence.

In the following table, `v` is an instance of `single_view`, `x` is an arbitrary type.

| Expression | Semantics |
|---|---|
| `single_view<x>`<br>`single_view<x>::type` | A single-element Random Access Sequence `v` such that `front<v>::type` is identical to `x`. |
| `size<v>::type` | The size of `v`; `size<v>::value == 1`; see Random Access Sequence. |

### Example

```
typedef single_view<int> view;
typedef begin<view>::type first;
typedef end<view>::type last;
```

```
BOOST_MPL_ASSERT(( is_same< deref<first>::type,int > ));
BOOST_MPL_ASSERT(( is_same< next<first>::type,last > ));
BOOST_MPL_ASSERT(( is_same< prior<last>::type,first > ));

BOOST_MPL_ASSERT_RELATION( size<view>::value, ==, 1 );
```

**See also**

Sequences, Views, iterator_range, filter_view, transform_view, joint_view, zip_view

### 1.3.6 transform_view

**Synopsis**

```
template<
      typename Sequence
    , typename F
    >
struct transform_view
{
    // unspecified
    // ...
};
```

**Description**

A view the full range of Sequence's transformed elements.

**Header**

```
#include <boost/mpl/transform_view.hpp>
```

**Model of**

— Forward Sequence

**Parameters**

| Parameter | Requirement | Description |
|---|---|---|
| Sequence | Forward Sequence | A sequence to wrap. |
| F | Unary Lambda Expression | A transformation. |

**Expression semantics**

The semantics of an expression are defined only where they differ from, or are not defined in Forward Sequence.

In the following table, v is an instance of transform_view, s is an arbitrary Forward Sequence, and f is an unary Lambda Expression.

| Expression | Semantics |
|---|---|
| `transform_view<s,f>`<br>`transform_view<s,f>::type` | A lazy Forward Sequence such that for each `i` in the range [`begin<v>::type`, `end<v>::type`) and each `j` in for in the range [`begin<s>::type`, `end<s>::type`) `deref<i>::type` is identical to `apply< f, deref<j>::type >::type`. |
| `size<v>::type` | The size of `v`; `size<v>::value == size<s>::value`; linear complexity; see Forward Sequence. |

### Example

Find the largest type in a sequence.

```
typedef vector<int,long,char,char[50],double> types;
typedef max_element<
      transform_view< types, size_of<_> >
    >::type iter;

BOOST_MPL_ASSERT_RELATION( deref<iter>::type::value, ==, 50 );
```

### See also

Sequences, Views, `filter_view`, `joint_view`, `zip_view`, `iterator_range`

## 1.3.7 zip_view

### Synopsis

```
template<
      typename Sequences
    >
struct zip_view
{
    // unspecified
    // ...
};
```

### Description

Provides a “zipped” view onto several sequences; that is, represents several sequences as a single sequence of elements each of which, in turn, is a sequence of the corresponding `Sequences`’ elements.

### Header

```
#include <boost/mpl/zip_view.hpp>
```

### Model of

— Forward Sequence

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| Sequences | A Forward Sequence of Forward Sequences | Sequences to be “zipped”. |

### Expression semantics

The semantics of an expression are defined only where they differ from, or are not defined in Forward Sequence.

In the following table, v is an instance of zip_view, seq a Forward Sequence of n Forward Sequences.

| Expression | Semantics |
|---|---|
| zip_view<seq><br>zip_view<seq>::type | A lazy Forward Sequence v such that for each i in [begin<v>::type, end<v>::type) and for each j in [begin<seq>::type, end<seq>::type) deref<i>::type is identical to transform< deref<j>::type, deref<_1> >::type. |
| size<v>::type | The size of v; size<v>::value is equal to<br>deref< min_element<br>transform_view< seq, size<_1> ><br>>::type >::type::value;<br>linear complexity; see Forward Sequence. |

### Example

Element-wise sum of three vectors.

```
typedef vector_c<int,1,2,3,4,5> v1;
typedef vector_c<int,5,4,3,2,1> v2;
typedef vector_c<int,1,1,1,1,1> v3;

typedef transform_view<
      zip_view< vector<v1,v2,v3> >
    , unpack_args< plus<_1,_2,_3> >
    > sum;

BOOST_MPL_ASSERT(( equal< sum, vector_c<int,7,7,7,7,7> > ));
```

### See also

Sequences, Views, filter_view, transform_view, joint_view, single_view, iterator_range

## 1.4 Intrinsic Metafunctions

The metafunctions that form the essential interface of sequence classes documented in the corresponding sequence concepts are known as *intrinsic sequence operations*. They differ from generic sequence algorithms in that, in general, they need to be implemented from scratch for each new sequence class[1)].

It’s worth noting that STL counterparts of these metafunctions are usually implemented as member functions.

[1)] In practice, many of intrinsic metafunctions offer a default implementation that will work in majority of cases, given that you’ve implemented the core functionality they rely on (such as begin / end).

### 1.4.1 at

#### Synopsis

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

template<
      typename AssocSeq
    , typename Key
    , typename Default = unspecified
    >
struct at
{
    typedef unspecified type;
};
```

#### Description

`at` is an overloaded name:

— `at<Sequence,N>` returns the `N`-th element from the beginning of the Forward Sequence `Sequence`.

— `at<AssocSeq,Key,Default>` returns the first element associated with `Key` in the Associative Sequence `AssocSeq`, or `Default` if no such element exists.

#### Header

```
#include <boost/mpl/at.hpp>
```

#### Model of

Tag Dispatched Metafunction

#### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Forward Sequence | A sequence to be examined. |
| `AssocSeq` | Associative Sequence | A sequence to be examined. |
| `N` | Integral Constant | An offset from the beginning of the sequence specifying the element to be retrieved. |
| `Key` | Any type | A key for the element to be retrieved. |
| `Default` | Any type | A default value to return if the element is not found. |

### Expression semantics

For any Forward Sequence `s`, and Integral Constant `n`:

```
typedef at<s,n>::type t;
```

**Return type:** A type.

**Precondition:** `0 <= n::value < size<s>::value`.

**Semantics:** Equivalent to

```
typedef deref< advance< begin<s>::type,n >::type >::type t;
```

For any Associative Sequence `s`, and arbitrary types `key` and `x`:

```
typedef at<s,key,x>::type t;
```

**Return type:** A type.

**Semantics:** If `has_key<s,key>::value == true`, `t` is the value type associated with `key`; otherwise `t` is identical to `x`.

```
typedef at<s,key>::type t;
```

**Return type:** A type.

**Semantics:** Equivalent to

```
typedef at<s,key,void_>::type t;
```

### Complexity

| Sequence archetype | Complexity |
|---|---|
| Forward Sequence | Linear. |
| Random Access Sequence | Amortized constant time. |
| Associative Sequence | Amortized constant time. |

### Example

```
typedef range_c<long,10,50> range;
BOOST_MPL_ASSERT_RELATION( (at< range, int_<0> >::value), ==, 10 );
BOOST_MPL_ASSERT_RELATION( (at< range, int_<10> >::value), ==, 20 );
BOOST_MPL_ASSERT_RELATION( (at< range, int_<40> >::value), ==, 50 );

typedef set< int const,long*,double > s;

BOOST_MPL_ASSERT(( is_same< at<s,char>::type, void_ > ));
BOOST_MPL_ASSERT(( is_same< at<s,int>::type, int > ));
```

### See also

Forward Sequence, Random Access Sequence, Associative Sequence, `at_c`, `front`, `back`

### 1.4.2 at_c

**Synopsis**

```
template<
      typename Sequence
    , long n
    >
struct at_c
{
    typedef unspecified type;
};
```

**Description**

Returns a type identical to the nth element from the beginning of the sequence. `at_c<Sequence,n>::type` is a shorcut notation for `at< Sequence, long_<n> >::type`.

**Header**

```
#include <boost/mpl/at.hpp>
```

**Parameters**

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Forward Sequence | A sequence to be examined. |
| `n` | A compile-time integral constant | An offset from the beginning of the sequence specifying the element to be retrieved. |

**Expression semantics**

```
typedef at_c<Sequence,n>::type t;
```

**Return type:** A type

**Precondition:** `0 <= n < size<Sequence>::value`

**Semantics:** Equivalent to

```
typedef at< Sequence, long_<n> >::type t;
```

**Complexity**

| Sequence archetype | Complexity |
|---|---|
| Forward Sequence | Linear. |
| Random Access Sequence | Amortized constant time. |

**Example**

```
typedef range_c<long,10,50> range;
BOOST_MPL_ASSERT_RELATION( (at_c< range,0 >::value), ==, 10 );
BOOST_MPL_ASSERT_RELATION( (at_c< range,10 >::value), ==, 20 );
```

```
BOOST_MPL_ASSERT_RELATION( (at_c< range,40 >::value), ==, 50 );
```

**See also**

Forward Sequence, Random Access Sequence, `at`, `front`, `back`

### 1.4.3 back

**Synopsis**

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

**Description**

Returns the last element in the sequence.

**Header**

```
#include <boost/mpl/back.hpp>
```

**Model of**

Tag Dispatched Metafunction

**Parameters**

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Bidirectional Sequence | A sequence to be examined. |

**Expression semantics**

For any Bidirectional Sequence `s`:

```
typedef back<s>::type t;
```

**Return type:** A type.

**Precondition:** `empty<s>::value == false`.

**Semantics:** Equivalent to

```
typedef deref< prior< end<s>::type >::type >::type t;
```

**Complexity**

Amortized constant time.

### Example

```
typedef range_c<int,0,1> range1;
typedef range_c<int,0,10> range2;
typedef range_c<int,-10,0> range3;

BOOST_MPL_ASSERT_RELATION( back<range1>::value, ==, 0 );
BOOST_MPL_ASSERT_RELATION( back<range2>::value, ==, 9 );
BOOST_MPL_ASSERT_RELATION( back<range3>::value, ==, -1 );
```

### See also

Bidirectional Sequence, `front`, `push_back`, `end`, `deref`, `at`

## 1.4.4 begin

### Synopsis

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

### Description

Returns an iterator that points to the first element of the sequence. If the argument is not a Forward Sequence, returns `void_`.

### Header

```
#include <boost/mpl/begin_end.hpp>
```

### Model of

Tag Dispatched Metafunction

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `X` | Any type | A type whose begin iterator, if any, will be returned. |

### Expression semantics

For any arbitrary type `x`:

```
typedef begin<x>::type first;
```

**Return type:** Forward Iterator or `void_`.

**Semantics:** If `x` is a Forward Sequence, `first` is an iterator pointing to the first element of `s`; otherwise `first` is `void_`.

**Postcondition:** If `first` is an iterator, it is either dereferenceable or past-the-end; it is past-the-end if and only if `size<x>::value == 0`.

#### Complexity

Amortized constant time.

#### Example

```
typedef vector< unsigned char,unsigned short,
    unsigned int,unsigned long > unsigned_types;

typedef begin<unsigned_types>::type iter;
BOOST_MPL_ASSERT(( is_same< deref<iter>::type, unsigned char > ));

BOOST_MPL_ASSERT(( is_same< begin<int>::type, void_ > ));
```

#### See also

Iterators, Forward Sequence, `end`, `size`, `empty`

### 1.4.5 clear

#### Synopsis

```
template<
      typename Sequence
    >
struct clear
{
    typedef unspecified type;
};
```

#### Description

Returns an empty sequence concept-identical to `Sequence`.

#### Header

```
#include <boost/mpl/clear.hpp>
```

#### Model of

Tag Dispatched Metafunction

#### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| Sequence | Extensible Sequence or Extensible Associative Sequence | A sequence to get an empty "copy" of. |

#### Expression semantics

For any Extensible Sequence or Extensible Associative Sequence s:

```
typedef clear<s>::type t;
```

**Return type:** Extensible Sequence or Extensible Associative Sequence.

**Semantics:** Equivalent to

```
typedef erase< s, begin<s>::type, end<s>::type >::type t;
```

**Postcondition:** `empty<s>::value == true`.

#### Complexity

Amortized constant time.

#### Example

```
typedef vector_c<int,1,3,5,7,9,11> odds;
typedef clear<odds>::type nothing;

BOOST_MPL_ASSERT(( empty<nothing> ));
```

#### See also

Extensible Sequence, Extensible Associative Sequence, `erase`, `empty`, `begin`, `end`

### 1.4.6 empty

#### Synopsis

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

#### Description

Returns an Integral Constant `c` such that `c::value == true` if and only if the sequence is empty.

#### Header

```
#include <boost/mpl/empty.hpp>
```

### Model of

Tag Dispatched Metafunction

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Forward Sequence | A sequence to test. |

### Expression semantics

For any Forward Sequence `s`:

```
typedef empty<s>::type c;
```

**Return type:** Boolean Integral Constant.

**Semantics:** Equivalent to `typedef is_same< begin<s>::type,end<s>::type >::type c;`.

**Postcondition:** `empty<s>::value == ( size<s>::value == 0 )`.

### Complexity

Amortized constant time.

### Example

```
typedef range_c<int,0,0> empty_range;
typedef vector<long,float,double> types;

BOOST_MPL_ASSERT( empty<empty_range> );
BOOST_MPL_ASSERT_NOT( empty<types> );
```

### See also

Forward Sequence, Integral Constant, `size`, `begin` / `end`

## 1.4.7 end

### Synopsis

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

### Description

Returns the sequence's past-the-end iterator. If the argument is not a Forward Sequence, returns `void_`.

### Header

```
#include <boost/mpl/begin_end.hpp>
```

### Model of

Tag Dispatched Metafunction

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `X` | Any type | A type whose end iterator, if any, will be returned. |

### Expression semantics

For any arbitrary type `x`:

```
typedef end<x>::type last;
```

**Return type:** Forward Iterator or `void_`.

**Semantics:** If `x` is Forward Sequence, `last` is an iterator pointing one past the last element in `s`; otherwise `last` is `void_`.

**Postcondition:** If `last` is an iterator, it is past-the-end.

### Complexity

Amortized constant time.

### Example

```
typedef vector<long> v;
typedef begin<v>::type first;
typedef end<v>::type last;

BOOST_MPL_ASSERT(( is_same< next<first>::type, last > ));
```

### See also

Iterators, Forward Sequence, `begin`, `end`, `next`

## 1.4.8 erase

### Synopsis

```
template<
      typename Sequence
    , typename First
    , typename Last = unspecified
    >
struct erase
```

```
{
    typedef unspecified type;
};
```

### Description

`erase` performs a removal of one or more adjacent elements in the sequence starting from an arbitrary position.

### Header

```
#include <boost/mpl/erase.hpp>
```

### Model of

Tag Dispatched Metafunction

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Extensible Sequence or Extensible Associative Sequence | A sequence to erase from. |
| `First` | Forward Iterator | An iterator to the beginning of the range to be erased. |
| `Last` | Forward Iterator | An iterator past-the-end of the range to be erased. |

### Expression semantics

For any Extensible Sequence `s`, and iterators `pos`, `first` and `last` into `s`:

`typedef erase<s,first,last>::type r;`

**Return type:** Extensible Sequence.

**Precondition:** `[first,last)` is a valid range in `s`.

**Semantics:** `r` is a new sequence, concept-identical to `s`, of the following elements: `[begin<s>::type, pos)`, `[last, end<s>::type)`.

**Postcondition:** The relative order of the elements in `r` is the same as in `s`;

```
size<r>::value == size<s>::value - distance<first,last>::value
```

`typedef erase<s,pos>::type r;`

**Return type:** Extensible Sequence.

**Precondition:** `pos` is a dereferenceable iterator in `s`.

**Semantics:** Equivalent to

```
typedef erase< s,pos,next<pos>::type >::type r;
```

For any Extensible Associative Sequence `s`, and iterator `pos` into `s`:

`typedef erase<s,pos>::type r;`

**Return type:** Extensible Sequence.

**Precondition:** `pos` is a dereferenceable iterator to `s`.

**Semantics:** Erases the element at a specific position `pos`; equivalent to `erase_key<s, deref<pos>::type >::type`.

**Postcondition:** `size<r>::value == size<s>::value - 1`.

### Complexity

| Sequence archetype | Complexity (the range form) |
|---|---|
| Extensible Associative Sequence | Amortized constant time. |
| Extensible Sequence | Quadratic in the worst case, linear at best. |

### Example

```
typedef vector_c<int,1,0,5,1,7,5,0,5> values;
typedef find< values, integral_c<int,7> >::type pos;
typedef erase<values,pos>::type result;

BOOST_MPL_ASSERT_RELATION( size<result>::value, ==, 7 );

typedef find<result, integral_c<int,7> >::type iter;
BOOST_MPL_ASSERT(( is_same< iter, end<result>::type > ));
```

### See also

Extensible Sequence, Extensible Associative Sequence, `erase_key`, `pop_front`, `pop_back`, `insert`

## 1.4.9 erase_key

### Synopsis

```
template<
      typename AssocSeq
    , typename Key
    >
struct erase_key
{
    typedef unspecified type;
};
```

### Description

Erases elements associated with the key `Key` in the Extensible Associative Sequence `AssocSeq`.

### Header

```
#include <boost/mpl/erase_key.hpp>
```

### Model of

Tag Dispatched Metafunction

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `AssocSeq` | Extensible Associative Sequence | A sequence to erase elements from. |
| `Key` | Any type | A key for the elements to be removed. |

### Expression semantics

For any Extensible Associative Sequence `s`, and arbitrary type `key`:

```
typedef erase_key<s,key>::type r;
```

**Return type:** Extensible Associative Sequence.

**Semantics:** `r` is concept-identical and equivalent to `s` except that `has_key<r,k>::value == false`.

**Postcondition:** `size<r>::value == size<s>::value - 1`.

### Complexity

Amortized constant time.

### Example

```
typedef map< pair<int,unsigned>, pair<char,long> > m;
typedef erase_key<m,char>::type m1;

BOOST_MPL_ASSERT_RELATION( size<m1>::type::value, ==, 1 );
BOOST_MPL_ASSERT(( is_same< at<m1,char>::type,void_ > ));
BOOST_MPL_ASSERT(( is_same< at<m1,int>::type,unsigned > ));
```

### See also

Extensible Associative Sequence, `erase`, `has_key`, `insert`

## 1.4.10 front

### Synopsis

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

### Description

Returns the first element in the sequence.

### Header

```
#include <boost/mpl/front.hpp>
```

### Model of

Tag Dispatched Metafunction

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Forward Sequence | A sequence to be examined. |

### Expression semantics

For any Forward Sequence `s`:

```
typedef front<s>::type t;
```

**Return type:** A type.

**Precondition:** `empty<s>::value == false`.

**Semantics:** Equivalent to

```
typedef deref< begin<s>::type >::type t;
```

### Complexity

Amortized constant time.

### Example

```
typedef list<long>::type types1;
typedef list<int,long>::type types2;
typedef list<char,int,long>::type types3;

BOOST_MPL_ASSERT(( is_same< front<types1>::type, long > ));
BOOST_MPL_ASSERT(( is_same< front<types2>::type, int> ));
BOOST_MPL_ASSERT(( is_same< front<types3>::type, char> ));
```

### See also

Forward Sequence, `back`, `push_front`, `begin`, `deref`, `at`

### 1.4.11 has_key

#### Synopsis

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

#### Description

Returns a true-valued Integral Constant if `Sequence` contains an element with key `Key`.

#### Header

```
#include <boost/mpl/has_key.hpp>
```

#### Model of

Tag Dispatched Metafunction

#### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Associative Sequence | A sequence to query. |
| `Key` | Any type | The queried key. |

#### Expression semantics

For any Associative Sequence `s`, and arbitrary type `key`:

```
typedef has_key<s,key>::type c;
```

**Return type:** Boolean Integral Constant.

**Semantics:** `c::value == true` if `key` is in `s`'s set of keys; otherwise `c::value == false`.

#### Complexity

Amortized constant time.

#### Example

```
typedef map< pair<int,unsigned>, pair<char,long> > m;
BOOST_MPL_ASSERT_NOT(( has_key<m,long> ));

typedef insert< m, pair<long,unsigned long> > m1;
BOOST_MPL_ASSERT(( has_key<m1,long> ));
```

### See also

Associative Sequence, `count`, `insert`, `erase_key`

## 1.4.12 insert

### Synopsis

```
template<
      typename Sequence
    , typename Pos
    , typename T
    >
struct insert
{
    typedef unspecified type;
};


template<
      typename Sequence
    , typename T
    >
struct insert
{
    typedef unspecified type;
};
```

### Description

`insert` is an overloaded name:

— `insert<Sequence,Pos,T>` performs an insertion of type `T` at an arbitrary position `Pos` in `Sequence`. `Pos` is ignored is `Sequence` is a model of Extensible Associative Sequence.

— `insert<Sequence,T>` is a shortcut notation for `insert<Sequence,Pos,T>` for the case when `Sequence` is a model of Extensible Associative Sequence.

### Header

```
#include <boost/mpl/insert.hpp>
```

### Model of

Tag Dispatched Metafunction

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Extensible Sequence or Extensible Associative Sequence | A sequence to insert into. |

| Parameter | Requirement | Description |
|---|---|---|
| Pos | Forward Iterator | An iterator in Sequence specifying the insertion position. |
| T | Any type | The element to be inserted. |

### Expression semantics

For any Extensible Sequence s, iterator pos in s, and arbitrary type x:

```
typedef insert<s,pos,x>::type r;
```

**Return type:** Extensible Sequence

**Precondition:** pos is an iterator in s.

**Semantics:** r is a sequence, concept-identical to s, of the following elements: [begin<s>::type, pos), x, [pos, end<s>::type).

**Postcondition:** The relative order of the elements in r is the same as in s.

```
at< r, distance< begin<s>::type,pos >::type >::type
```

is identical to x;

```
size<r>::value == size<s>::value + 1;
```

For any Extensible Associative Sequence s, iterator pos in s, and arbitrary type x:

```
typedef insert<s,x>::type r;
```

**Return type:** Extensible Associative Sequence

**Semantics:** r is concept-identical and equivalent to s, except that at< r, key_type<s,x>::type >::type is identical to value_type<s,x>::type.

**Postcondition:** size<r>::value == size<s>::value + 1.

```
typedef insert<s,pos,x>::type r;
```

**Return type:** Extensible Associative Sequence

**Precondition:** pos is an iterator in s.

**Semantics:** Equivalent to typedef insert<s,x>::type r; pos is ignored.

### Complexity

| Sequence archetype | Complexity |
|---|---|
| Extensible Associative Sequence | Amortized constant time. |
| Extensible Sequence | Linear in the worst case, or amortized constant time. |

### Example

```
typedef vector_c<int,0,1,3,4,5,6,7,8,9> numbers;
typedef find< numbers,integral_c<int,3> >::type pos;
typedef insert< numbers,pos,integral_c<int,2> >::type range;

BOOST_MPL_ASSERT_RELATION( size<range>::value, ==, 10 );
BOOST_MPL_ASSERT(( equal< range,range_c<int,0,10> > ));
```

```
typedef map< mpl::pair<int,unsigned> > m;
typedef insert<m,mpl::pair<char,long> >::type m1;

BOOST_MPL_ASSERT_RELATION( size<m1>::value, ==, 2 );
BOOST_MPL_ASSERT(( is_same< at<m1,int>::type,unsigned > ));
BOOST_MPL_ASSERT(( is_same< at<m1,char>::type,long > ));
```

**See also**

Extensible Sequence, Extensible Associative Sequence, `insert_range`, `push_front`, `push_back`, `erase`

### 1.4.13 insert_range

**Synopsis**

```
template<
      typename Sequence
    , typename Pos
    , typename Range
    >
struct insert_range
{
    typedef unspecified type;
};
```

**Description**

`insert_range` performs an insertion of a range of elements at an arbitrary position in the sequence.

**Header**

```
#include <boost/mpl/insert_range.hpp>
```

**Model of**

Tag Dispatched Metafunction

**Parameters**

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Extensible Sequence or Extensible Associative Sequence | A sequence to insert into. |
| `Pos` | Forward Iterator | An iterator in `Sequence` specifying the insertion position. |
| `Range` | Forward Sequence | The range of elements to be inserted. |

**Expression semantics**

For any Extensible Sequence `s`, iterator `pos` in `s`, and Forward Sequence `range`:

```
typedef insert<s,pos,range>::type r;
```

**Return type:** Extensible Sequence.

**Precondition:** `pos` is an iterator into `s`.

**Semantics:** `r` is a sequence, concept-identical to `s`, of the following elements: [`begin<s>::type`, `pos`), [`begin<r>::type`, `end<r>::type`), [`pos`, `end<s>::type`).

**Postcondition:** The relative order of the elements in `r` is the same as in `s`;

```
size<r>::value == size<s>::value + size<range>::value
```

### Complexity

Sequence dependent. Quadratic in the worst case, linear at best; see the particular sequence class' specification for details.

### Example

```
typedef vector_c<int,0,1,7,8,9> numbers;
typedef find< numbers,integral_c<int,7> >::type pos;
typedef insert_range< numbers,pos,range_c<int,2,7> >::type range;

BOOST_MPL_ASSERT_RELATION( size<range>::value, ==, 10 );
```

### Header

```
#include <boost/mpl/is_sequence.hpp>
```

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| X | Any type | The type to query. |

### Expression semantics

```
typedef is_sequence<X>::type c;
```

**Return type:** Boolean Integral Constant.

**Semantics:** Equivalent to

```
typedef not_< is_same< begin<T>::type,void_ > >::type c;
```

### Complexity

Amortized constant time.

### Example

```
struct UDT {};

BOOST_MPL_ASSERT_NOT(( is_sequence< std::vector<int> > ));
BOOST_MPL_ASSERT_NOT(( is_sequence< int > ));
BOOST_MPL_ASSERT_NOT(( is_sequence< int& > ));
BOOST_MPL_ASSERT_NOT(( is_sequence< UDT > ));
BOOST_MPL_ASSERT_NOT(( is_sequence< UDT* > ));
BOOST_MPL_ASSERT(( is_sequence< range_c<int,0,0> > ));
BOOST_MPL_ASSERT(( is_sequence< list<> > ));
BOOST_MPL_ASSERT(( is_sequence< list<int> > ));
BOOST_MPL_ASSERT(( is_sequence< vector<> > ));
BOOST_MPL_ASSERT(( is_sequence< vector<int> > ));
```

### See also

Forward Sequence, `begin`, `end`, `vector`, `list`, `range_c`

## 1.4.15 key_type

### Synopsis

```
template<
      typename Sequence
    , typename X
    >
struct key_type
{
```

```
    typedef unspecified type;
};
```

### Description

Returns the key that would be used to identify `X` in `Sequence`.

### Header

```
#include <boost/mpl/key_type.hpp>
```

### Model of

Tag Dispatched Metafunction

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Associative Sequence | A sequence to query. |
| `X` | Any type | The type to get the key for. |

### Expression semantics

For any Associative Sequence `s`, iterators `pos1` and `pos2` in `s`, and an artibrary type `x`:

```
typedef key_type<s,x>::type k;
```

**Return type:** A type.

**Precondition:** `x` can be put in `s`.

**Semantics:** `k` is the key that would be used to identify `x` in `s`.

**Postcondition:** If `key_type< s,deref<pos1>::type >::type` is identical to `key_type< s,deref<pos2>::type >::type` then `pos1` is identical to `pos2`.

### Complexity

Amortized constant time.

### Example

```
typedef key_type< map<>,pair<int,unsigned> >::type k1;
typedef key_type< set<>,pair<int,unsigned> >::type k2;

BOOST_MPL_ASSERT(( is_same< k1,int > ));
BOOST_MPL_ASSERT(( is_same< k2,pair<int,unsigned> > ));
```

### See also

Associative Sequence, `value_type`, `has_key`, `set`, `map`

### 1.4.16 order

#### Synopsis

```
template<
      typename Sequence
    , typename Key
    >
struct order
{
    typedef unspecified type;
};
```

#### Description

Returns a unique unsigned Integral Constant associated with the key `Key` in `Sequence`.

#### Header

```
#include <boost/mpl/order.hpp>
```

#### Model of

Tag Dispatched Metafunction

#### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Associative Sequence | A sequence to query. |
| `Key` | Any type | The queried key. |

#### Expression semantics

For any Associative Sequence `s`, and arbitrary type `key`:

```
typedef order<s,key>::type n;
```

**Return type:** Unsigned Integral Constant.

**Semantics:** If `has_key<s,key>::value == true`, `n` is a unique unsigned Integral Constant associated with `key` in `s`; otherwise, `n` is identical to `void_`.

#### Complexity

Amortized constant time.

#### Example

```
typedef map< pair<int,unsigned>, pair<char,long> > m;

BOOST_MPL_ASSERT_NOT(( is_same< order<m,int>::type, void_ > ));
BOOST_MPL_ASSERT(( is_same< order<m,long>::type,void_ > ));
```

### See also

Associative Sequence, has_key, count, map

### 1.4.17 pop_back

### Synopsis

```
template<
      typename Sequence
    >
struct pop_back
{
    typedef unspecified type;
};
```

### Description

pop_back performs a removal at the end of the sequence with guaranteed *O(1)* complexity.

### Header

```
#include <boost/mpl/pop_back.hpp>
```

### Model of

Tag Dispatched Metafunction

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| Sequence | Back Extensible Sequence | A sequence to erase the last element from. |

### Expression semantics

For any Back Extensible Sequence s:

```
typedef pop_back<s>::type r;
```

**Return type:** Back Extensible Sequence.

**Precondition:** empty<s>::value == false.

**Semantics:** Equivalent to erase<s,end<s>::type>::type;.

**Postcondition:** size<r>::value == size<s>::value - 1.

### Complexity

Amortized constant time.

### Example

```
typedef vector<long>::type types1;
typedef vector<long,int>::type types2;
typedef vector<long,int,char>::type types3;

typedef pop_back<types1>::type result1;
typedef pop_back<types2>::type result2;
typedef pop_back<types3>::type result3;

BOOST_MPL_ASSERT_RELATION( size<result1>::value, ==, 0 );
BOOST_MPL_ASSERT_RELATION( size<result2>::value, ==, 1 );
BOOST_MPL_ASSERT_RELATION( size<result3>::value, ==, 2 );

BOOST_MPL_ASSERT(( is_same< back<result2>::type, long> ));
BOOST_MPL_ASSERT(( is_same< back<result3>::type, int > ));
```

### See also

Back Extensible Sequence, `erase`, `push_back`, `back`, `pop_front`

## 1.4.18 pop_front

### Synopsis

```
template<
      typename Sequence
    >
struct pop_front
{
    typedef unspecified type;
};
```

### Description

`pop_front` performs a removal at the beginning of the sequence with guaranteed *O(1)* complexity.

### Header

```
#include <boost/mpl/pop_front.hpp>
```

### Model of

Tag Dispatched Metafunction

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Front Extensible Sequence | A sequence to erase the first element from. |

#### Expression semantics

For any Front Extensible Sequence `s`:

`typedef pop_front<s>::type r;`

**Return type:** Front Extensible Sequence.

**Precondition:** `empty<s>::value == false`.

**Semantics:** Equivalent to `erase<s,begin<s>::type>::type;`.

**Postcondition:** `size<r>::value == size<s>::value - 1`.

#### Complexity

Amortized constant time.

#### Example

```
typedef vector<long>::type types1;
typedef vector<int,long>::type types2;
typedef vector<char,int,long>::type types3;

typedef pop_front<types1>::type result1;
typedef pop_front<types2>::type result2;
typedef pop_front<types3>::type result3;

BOOST_MPL_ASSERT_RELATION( size<result1>::value, ==, 0 );
BOOST_MPL_ASSERT_RELATION( size<result2>::value, ==, 1 );
BOOST_MPL_ASSERT_RELATION( size<result3>::value, ==, 2 );

BOOST_MPL_ASSERT(( is_same< front<result2>::type, long > ));
BOOST_MPL_ASSERT(( is_same< front<result3>::type, int > ));
```

#### See also

Front Extensible Sequence, `erase`, `push_front`, `front`, `pop_back`

### 1.4.19 push_back

#### Synopsis

```
template<
      typename Sequence
    , typename T
    >
struct push_back
{
    typedef unspecified type;
};
```

### Description

`push_back` performs an insertion at the end of the sequence with guaranteed *O(1)* complexity.

### Header

```
#include <boost/mpl/push_back.hpp>
```

### Model of

Tag Dispatched Metafunction

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Back Extensible Sequence | A sequence to insert into. |
| `T` | Any type | The element to be inserted. |

### Expression semantics

For any Back Extensible Sequence `s` and arbitrary type `x`:

```
typedef push_back<s,x>::type r;
```

**Return type:** Back Extensible Sequence.

**Semantics:** Equivalent to

```
typedef insert< s,end<s>::type,x >::type r;
```

**Postcondition:** `back<r>::type` is identical to `x`;

```
size<r>::value == size<s>::value + 1
```

### Complexity

Amortized constant time.

### Example

```
typedef vector_c<bool,false,false,false,
    true,true,true,false,false> bools;

typedef push_back<bools,false_>::type message;

BOOST_MPL_ASSERT_RELATION( back<message>::type::value, ==, false );
BOOST_MPL_ASSERT_RELATION(
      ( count_if<message, equal_to<_1,false_> >::value ), ==, 6
    );
```

### See also

Back Extensible Sequence, `insert`, `pop_back`, `back`, `push_front`

### 1.4.20 push_front

#### Synopsis

```
template<
      typename Sequence
    , typename T
    >
struct push_front
{
    typedef unspecified type;
};
```

#### Description

`push_front` performs an insertion at the beginning of the sequence with guaranteed *O(1)* complexity.

#### Header

```
#include <boost/mpl/push_front.hpp>
```

#### Model of

Tag Dispatched Metafunction

#### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Front Extensible Sequence | A sequence to insert into. |
| `T` | Any type | The element to be inserted. |

#### Expression semantics

For any Front Extensible Sequence `s` and arbitrary type `x`:

```
typedef push_front<s,x>::type r;
```

**Return type:** Front Extensible Sequence.

**Semantics:** Equivalent to

```
typedef insert< s,begin<s>::type,x >::type r;
```

**Postcondition:** `size<r>::value == size<s>::value + 1`; `front<r>::type` is identical to `x`.

#### Complexity

Amortized constant time.

#### Example

```
typedef vector_c<int,1,2,3,5,8,13,21> v;
BOOST_MPL_ASSERT_RELATION( size<v>::value, ==, 7 );
```

```
typedef push_front< v,integral_c<int,1> >::type fibonacci;
BOOST_MPL_ASSERT_RELATION( size<fibonacci>::value, ==, 8 );

BOOST_MPL_ASSERT(( equal<
      fibonacci
    , vector_c<int,1,1,2,3,5,8,13,21>
    , equal_to<_,_>
    > ));
```

**See also**

Front Extensible Sequence, `insert`, `pop_front`, `front`, `push_back`

### 1.4.21 sequence_tag

**Synopsis**

```
template<
      typename X
    >
struct sequence_tag
{
    typedef unspecified type;
};
```

**Description**

`sequence_tag` is a tag metafunction for all tag dispatched intrinsic sequence operations.

**Header**

```
#include <boost/mpl/sequence_tag.hpp>
```

**Parameters**

| Parameter | Requirement | Description |
|---|---|---|
| `X` | Any type | A type to obtain a sequence tag for. |

**Expression semantics**

For any arbitrary type `x`:

```
typedef sequence_tag<x>::type tag;
```

**Return type:** A type.

**Semantics:** `tag` is an unspecified tag type for `x`.

**Complexity**

Amortized constant time.

### See also

Intrinsic Metafunctions, Tag Dispatched Metafunction

### 1.4.22 size

### Synopsis

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

### Description

`size` returns the number of elements in the sequence, that is, the number of elements in the range [`begin<Sequence>::type`, `end<Sequence>::type`).

### Header

```
#include <boost/mpl/size.hpp>
```

### Model of

Tag Dispatched Metafunction

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Forward Sequence | A sequence to query. |

### Expression semantics

For any Forward Sequence `s`:

```
typedef size<s>::type n;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```
typedef distance< begin<s>::type,end<s>::type >::type n;
```

**Postcondition:** `n::value >= 0`.

### Complexity

The complexity of the `size` metafunction directly depends on the implementation of the particular sequence it is applied to. In the worst case, `size` guarantees a linear complexity.

If the `s` is a Random Access Sequence, `size<s>::type` is an *O(1)* operation. The opposite is not necessarily true — for example, a sequence class that models Forward Sequence might still give us an *O(1)* `size` implementation.

#### Example

```
typedef list0<> empty_list;
typedef vector_c<int,0,1,2,3,4,5> numbers;
typedef range_c<int,0,100> more_numbers;

BOOST_MPL_ASSERT_RELATION( size<list>::value, ==, 0 );
BOOST_MPL_ASSERT_RELATION( size<numbers>::value, ==, 5 );
BOOST_MPL_ASSERT_RELATION( size<more_numbers>::value, ==, 100 );
```

#### See also

Forward Sequence, Random Access Sequence, `empty`, `begin`, `end`, `distance`

### 1.4.23 value_type

#### Synopsis

```
template<
      typename Sequence
    , typename X
    >
struct value_type
{
    typedef unspecified type;
};
```

#### Description

Returns the value that would be used for element `X` in `Sequence`.

#### Header

```
#include <boost/mpl/value_type.hpp>
```

#### Model of

Tag Dispatched Metafunction

#### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Associative Sequence | A sequence to query. |
| `X` | Any type | The type to get the value for. |

### Expression semantics

For any Associative Sequence `s`, and an artibrary type `x`:

```
typedef value_type<s,x>::type v;
```

**Return type:** A type.

**Precondition:** `x` can be put in `s`.

**Semantics:** `v` is the value that would be used for `x` in `s`.

**Postcondition:** If .. parsed-literal:

```
has_key< s,key_type<s,x>::type >::type
```

then .. parsed-literal:

```
at< s,key_type<s,x>::type >::type
```

is identical to `value_type<s,x>::type`.

### Complexity

Amortized constant time.

### Example

```
typedef value_type< map<>,pair<int,unsigned> >::type v1;
typedef value_type< set<>,pair<int,unsigned> >::type v2;

BOOST_MPL_ASSERT(( is_same< v1,unsigned > ));
BOOST_MPL_ASSERT(( is_same< v2,pair<int,unsigned> > ));
```

### See also

Associative Sequence, `key_type`, `at`, `set`, `map`

# Chapter 2 Iterators

Iterators are generic means of addressing a particular element or a range of sequential elements in a sequence. They are also a mechanism that makes it possible to decouple algorithms from concrete compile-time sequence implementations. Under the hood, all MPL sequence algorithms are implemented in terms of iterators. In particular, that means that they will work on any custom compile-time sequence, given that the appropriate iterator inteface is provided.

## 2.1 Concepts

All iterators in MPL are classified into three iterator concepts, or *categories*, named according to the type of traversal provided. The categories are: Forward Iterator, Bidirectional Iterator, and Random Access Iterator. The concepts are hierarchical: Random Access Iterator is a refinement of Bidirectional Iterator, which, in its turn, is a refinement of Forward Iterator.

Because of the inherently immutable nature of the value access, MPL iterators escape the problems of the traversal-only categorization discussed at length in [n1550].

### 2.1.1 Forward Iterator

**Description**

A Forward Iterator `i` is a type that represents a positional reference to an element of a Forward Sequence. It allows to access the element through a dereference operation, and provides a way to obtain an iterator to the next element in a sequence.

**Definitions**

— An iterator can be *dereferenceable*, meaning that `deref<i>::type` is a well-defined expression.

— An iterator is *past-the-end* if it points beyond the last element of a sequence; past-the-end iterators are non-dereferenceable.

— An iterator `i` is *incrementable* if there is a "next" iterator, that is, if `next<i>::type` expression is well-defined; past-the-end iterators are not incrementable.

— Two iterators into the same sequence are *equivalent* if they have the same type.

— An iterator `j` is *reachable* from an iterator `i` if , after recursive application of `next` metafunction to `i` a finite number of times, `i` is equivalent to `j`.

— The notation [`i`,`j`) refers to a *range* of iterators beginning with `i` and up to but not including `j`.

— The range [`i`,`j`) is a *valid range* if `j` is reachable from `i`.

**Expression requirements**

| Expression | Type | Complexity |
|---|---|---|
| `deref<i>::type` | Any type | Amortized constant time |
| `next<i>::type` | Forward Iterator | Amortized constant time |
| `i::category` | Integral Constant, convertible to `forward_iterator_tag` | Constant time |

**Expression semantics**

`typedef deref<i>::type j;`

**Precondition:** `i` is dereferenceable

**Semantics:** `j` is identical to the type of the pointed element

`typedef next<i>::type j;`

**Precondition:** `i` is incrementable

**Semantics:** `j` is the next iterator in a sequence

**Postcondition:** `j` is dereferenceable or past-the-end

`typedef i::category c;`

**Semantics:** `c` is identical to the iterator's category tag

**Invariants**

For any forward iterators `i` and `j` the following invariants always hold:

— `i` and `j` are equivalent if and only if they are pointing to the same element.

— If `i` is dereferenceable, and `j` is equivalent to `i`, then `j` is dereferenceable as well.

— If `i` and `j` are equivalent and dereferenceable, then `deref<i>::type` and `deref<j>::type` are identical.

— If `i` is incrementable, and `j` is equivalent to `i`, then `j` is incrementable as well.

— If `i` and `j` are equivalent and incrementable, then `next<i>::type` and `next<j>::type` are equivalent.

**See also**

Iterators, Bidirectional Iterator, Forward Sequence, `deref`, `next`

### 2.1.2 Bidirectional Iterator

**Description**

A Bidirectional Iterator is a Forward Iterator that provides a way to obtain an iterator to the previous element in a sequence.

**Refinement of**

Forward Iterator

### Definitions

— a bidirectional iterator `i` is *decrementable* if there is a "previous" iterator, that is, if `prior<i>::type` expression is well-defined; iterators pointing to the first element of the sequence are not decrementable.

### Expression requirements

In addition to the requirements defined in Forward Iterator, the following requirements must be met.

| Expression | Type | Complexity |
|---|---|---|
| `next<i>::type` | Bidirectional Iterator | Amortized constant time |
| `prior<i>::type` | Bidirectional Iterator | Amortized constant time |
| `i::category` | Integral Constant, convertible to `bidirectional_iterator_tag` | Constant time |

### Expression semantics

```
typedef prior<i>::type j;
```

**Precondition:** `i` is decrementable

**Semantics:** `j` is an iterator pointing to the previous element of the sequence

**Postcondition:** `j` is dereferenceable and incrementable

### Invariants

For any bidirectional iterators `i` and `j` the following invariants always hold:

— If `i` is incrementable, then `prior< next<i>::type >::type` is a null operation; similarly, if `i` is decrementable, `next< prior<i>::type >::type` is a null operation.

### See also

Iterators, Forward Iterator, Random Access Iterator, Bidirectional Sequence, `prior`

## 2.1.3 Random Access Iterator

### Description

A Random Access Iterator is a Bidirectional Iterator that provides constant-time guarantees on moving the iterator an arbitrary number of positions forward or backward and for measuring the distance to another iterator in the same sequence.

### Refinement of

Bidirectional Iterator

### Expression requirements

In addition to the requirements defined in Bidirectional Iterator, the following requirements must be met.

| Expression | Type | Complexity |
|---|---|---|
| `next<i>::type` | Random Access Iterator | Amortized constant time |
| `prior<i>::type` | Random Access Iterator | Amortized constant time |
| `i::category` | Integral Constant, convertible to `random_access_iterator_tag` | Constant time |
| `advance<i,n>::type` | Random Access Iterator | Amortized constant time |
| `distance<i,j>::type` | Integral Constant | Amortized constant time |

#### Expression semantics

`typedef advance<i,n>::type j;`

**Semantics:** See `advance` specification

`typedef distance<i,j>::type n;`

**Semantics:** See `distance` specification

#### Invariants

For any random access iterators `i` and `j` the following invariants always hold:

— If `advance<i,n>::type` is well-defined, then `advance< advance<i,n>::type, negate<n>::type >::type` is a null operation.

#### See also

Iterators, Bidirectional Iterator, Random Access Sequence, `advance`, `distance`

### 2.2 Iterator Metafunctions

#### 2.2.1 advance

#### Synopsis

```
template<
      typename Iterator
    , typename N
    >
struct advance
{
    typedef unspecified type;
};
```

#### Description

Moves `Iterator` by the distance `N`. For bidirectional and random access iterators, the distance may be negative.

#### Header

`#include <boost/mpl/advance.hpp>`

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Iterator` | Forward Iterator | An iterator to advance. |
| `N` | Integral Constant | A distance. |

### Model Of

Tag Dispatched Metafunction

### Expression semantics

For a Forward Iterator `iter` and arbitrary Integral Constant `n`:

```
typedef advance<iter,n>::type j;
```

**Return type:** Forward Iterator.

**Precondition:** If `Iterator` is a Forward Iterator, `n::value` must be nonnegative.

**Semantics:** Equivalent to:

```
typedef iter i0;
typedef next<i0>::type i1;
...
typedef next<in-1>::type j;
```

if `n::value > 0`, and

```
typedef iter i0;
typedef prior<i0>::type i1;
...
typedef prior<in-1>::type j;
```

otherwise.

**Postcondition:** `j` is dereferenceable or past-the-end; `distance<iter,j>::value == n::value` if `n::value > 0`, and `distance<j,iter>::value == n::value` otherwise.

### Complexity

Amortized constant time if `iter` is a model of Random Access Iterator, otherwise linear time.

### Example

```
typedef range_c<int,0,10> numbers;
typedef begin<numbers>::type first;
typedef end<numbers>::type last;

typedef advance<first,int_<10> >::type i1;
typedef advance<last,int_<-10> >::type i2;

BOOST_MPL_ASSERT(( boost::is_same<i1,last> ));
BOOST_MPL_ASSERT(( boost::is_same<i2,first> ));
```

### See also

Iterators, Tag Dispatched Metafunction, `distance`, `next`

## 2.2.2 distance

### Synopsis

```
template<
      typename First
    , typename Last
    >
struct distance
{
    typedef unspecified type;
};
```

### Description

Returns the distance between `First` and `Last` iterators, that is, an Integral Constant `n` such that `advance<First,n>::type` is identical to `Last`.

### Header

```
#include <boost/mpl/distance.hpp>
```

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `First`, `Last` | Forward Iterator | Iterators to compute a distance between. |

### Model Of

Tag Dispatched Metafunction

### Expression semantics

For any Forward Iterators `first` and `last`:

```
typedef distance<first,last>::type n;
```

**Return type:** Integral Constant.

**Precondition:** [`first`, `last`) is a valid range.

**Semantics:** Equivalent to

```
typedef iter_fold<
      iterator_range<first,last>
    , long_<0>
    , next<_1>
    >::type n;
```

**Postcondition:** `is_same< advance<first,n>::type, last >::value == true`.

### Complexity

Amortized constant time if `first` and `last` are Random Access Iterators, otherwise linear time.

### Example

```
typedef range_c<int,0,10>::type range;
typedef begin<range>::type first;
typedef end<range>::type last;

BOOST_MPL_ASSERT_RELATION( (distance<first,last>::value), ==, 10);
```

### See also

Iterators, Tag Dispatched Metafunction, `advance`, `next`, `prior`

## 2.2.3 next

### Synopsis

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

### Description

Returns the next iterator in the sequence. [*Note:* `next` has a number of overloaded meanings, depending on the type of its argument. For instance, if `X` is an Integral Constant, `next<X>` returns an incremented Integral Constant of the same type. The following specification is iterator-specific. Please refer to the corresponding concept's documentation for the details of the alternative semantics — *end note*].

### Header

```
#include <boost/mpl/next_prior.hpp>
```

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Iterator` | Forward Iterator. | An iterator to increment. |

### Expression semantics

For any Forward Iterators `iter`:

```
typedef next<iter>::type j;
```

**Return type:** Forward Iterator.

**Precondition:** `iter` is incrementable.

**Semantics:** `j` is an iterator pointing to the next element in the sequence, or is past-the-end. If `iter` is a user-defined iterator, the library-provided default implementation is equivalent to

```
typedef iter::next j;
```

### Complexity

Amortized constant time.

### Example

```
typedef vector_c<int,1> v;
typedef begin<v>::type first;
typedef end<v>::type last;

BOOST_MPL_ASSERT(( is_same< next<first>::type, last > ));
```

### See also

Iterators, `begin` / `end`, `prior`, `deref`

## 2.2.4 prior

### Synopsis

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

### Description

Returns the previous iterator in the sequence. [*Note:* `prior` has a number of overloaded meanings, depending on the type of its argument. For instance, if `X` is an Integral Constant, `prior<X>` returns an decremented Integral Constant of the same type. The following specification is iterator-specific. Please refer to the corresponding concept's documentation for the details of the alternative semantics — *end note*].

### Header

```
#include <boost/mpl/next_prior.hpp>
```

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Iterator` | Forward Iterator. | An iterator to decrement. |

#### Expression semantics

For any Forward Iterators `iter`:

```
typedef prior<iter>::type j;
```

**Return type:** Forward Iterator.

**Precondition:** `iter` is decrementable.

**Semantics:** `j` is an iterator pointing to the previous element in the sequence. If `iter` is a user-defined iterator, the library-provided default implementation is equivalent to

```
typedef iter::prior j;
```

#### Complexity

Amortized constant time.

#### Example

```
typedef vector_c<int,1> v;
typedef begin<v>::type first;
typedef end<v>::type last;

BOOST_MPL_ASSERT(( is_same< prior<last>::type, first > ));
```

#### See also

Iterators, `begin` / `end`, `next`, `deref`

### 2.2.5 deref

#### Synopsis

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

#### Description

Dereferences an iterator.

#### Header

```
#include <boost/mpl/deref.hpp>
```

#### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Iterator` | Forward Iterator | The iterator to dereference. |

#### Expression semantics

For any Forward Iterators `iter`:

```
typedef deref<iter>::type t;
```

**Return type:** A type.

**Precondition:** `iter` is dereferenceable.

**Semantics:** `t` is identical to the element referenced by `iter`. If `iter` is a user-defined iterator, the library-provided default implementation is equivalent to

```
typedef iter::type t;
```

#### Complexity

Amortized constant time.

#### Example

```
typedef vector<char,short,int,long> types;
typedef begin<types>::type iter;

BOOST_MPL_ASSERT(( is_same< deref<iter>::type, char > ));
```

#### See also

Iterators, `begin` / `end`, `next`

### 2.2.6 iterator_category

#### Synopsis

```
template<
      typename Iterator
    >
struct iterator_category
{
    typedef typename Iterator::category type;
};
```

#### Description

Returns one of the following iterator category tags:

- `forward_iterator_tag`
- `bidirectional_iterator_tag`
- `random_access_iterator_tag`

### Header

```
#include <boost/mpl/iterator_category.hpp>
#include <boost/mpl/iterator_tags.hpp>
```

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Iterator` | Forward Iterator | The iterator to obtain a category for. |

### Expression semantics

For any Forward Iterators `iter`:

```
typedef iterator_category<iter>::type tag;
```

**Return type:** Integral Constant.

**Semantics:** `tag` is `forward_iterator_tag` if `iter` is a model of Forward Iterator, `bidirectional_iterator_tag` if `iter` is a model of Bidirectional Iterator, or `random_access_iterator_tag` if `iter` is a model of Random Access Iterator;

**Postcondition:** `forward_iterator_tag::value < bidirectional_iterator_tag::value`, `bidirectional_iterator_tag::value < random_access_iterator_tag::value`.

### Complexity

Amortized constant time.

### Example

```
template< typename Tag, typename Iterator >
struct algorithm_impl
{
    // O(n) implementation
};

template< typename Iterator >
struct algorithm_impl<random_access_iterator_tag,Iterator>
{
    // O(1) implementation
};

template< typename Iterator >
struct algorithm
    : algorithm_impl<
          iterator_category<Iterator>::type
        , Iterator
        >
{
};
```

**See also**

Iterators, `begin` / `end`, `advance`, `distance`, `next`

# Chapter 3 Algorithms

The MPL provides a broad range of fundamental algorithms aimed to satisfy the majority of sequential compile-time data processing needs. The algorithms include compile-time counterparts of many of the STL algorithms, iteration algorithms borrowed from functional programming languages, and more.

Unlike the algorithms in the C++ Standard Library, which operate on implict *iterator ranges*, the majority of MPL counterparts take and return *sequences*. This derivation is not dictated by the functional nature of C++ compile-time computations per se, but rather by a desire to improve general usability of the library, making programming with compile-time data structures as enjoyable as possible.

In the spirit of the STL, MPL algorithms are *generic*, meaning that they are not tied to particular sequence class implementations, and can operate on a wide range of arguments as long as they satisfy the documented requirements. The requirements are formulated in terms of concepts. Under the hood, algorithms are decoupled from concrete sequence implementations by operating on Iterators.

All MPL algorithms can be sorted into three major categories: iteration algorithms, querying algorithms, and transformation algorithms. The transformation algorithms introduce an associated Inserter concept, a rough equivalent for the notion of Output Iterator in the Standard Library. Moreover, every transformation algorithm provides a `reverse_` counterpart, allowing for a wider range of efficient transformations — a common functionality documented by the Reversible Algorithm concept.

## 3.1 Concepts

### 3.1.1 Inserter

#### Description

An Inserter is a compile-time substitute for STL Output Iterator. Under the hood, it's simply a type holding two entities: a *state* and an *operation*. When passed to a transformation algorithm, the inserter's binary operation is invoked for every element that would normally be written into the output iterator, with the element itself (as the second argument) and the result of the previous operation's invocation — or, for the very first element, the inserter's initial state.

Technically, instead of taking a single inserter parameter, transformation algorithms could accept the state and the "output" operation separately. Grouping these in a single parameter entity, however, brings the algorithms semantically and syntactically closer to their STL counterparts, significantly simplifying many of the common use cases.

#### Valid expressions

In the following table and subsequent specifications, `in` is a model of Inserter.

| Expression | Type |
|---|---|
| `in::state` | Any type |
| `in::operation` | Binary Lambda Expression |

### Expression semantics

| Expression | Semantics |
|---|---|
| `in::state` | The inserter's initial state. |
| `in::operation` | The inserter's "output" operation. |

### Example

```
typedef transform<
      range_c<int,0,10>
    , plus<_1,_1>
    , back_inserter< vector0<> >
    >::type result;
```

### Models

— `inserter`

— `front_inserter`

— `back_inserter`

### See also

Algorithms, Transformation Algorithms, `inserter`, `front_inserter`, `back_inserter`

## 3.1.2 Reversible Algorithm

### Description

A Reversible Algorithm is a member of a pair of transformation algorithms that iterate over their input sequence(s) in opposite directions. For each reversible algorithm `x` there exists a *counterpart* algorithm `reverse_x`, that exhibits the exact semantics of `x` except that the elements of its input sequence argument(s) are processed in the reverse order.

### Expression requirements

In the following table and subsequent specifications, `x` is a placeholder token for the actual Reversible Algorithm's name, $s_1$,$s_2$,...$s_n$ are Forward Sequences, and `in` is an Inserter.

| Expression | Type | Complexity |
|---|---|---|
| `x<`$s_1$,$s_2$,...$s_n$, ...`>::type` | Forward Sequence | Unspecified. |
| `x<`$s_1$,$s_2$,...$s_n$, ... `in>::type` | Any type | Unspecified. |
| `reverse_x<`$s_1$,$s_2$,...$s_n$, ...`>::type` | Forward Sequence | Unspecified. |
| `reverse_x<`$s_1$,$s_2$,...$s_n$, ... `in>::type` | Any type | Unspecified. |

### Expression semantics

`typedef x<`$s_1$`,`$s_2$`,...`$s_n$`,...>::type t;`

**Precondition:** $s_1$ is an Extensible Sequence.

**Semantics:** `t` is equivalent to

```
x<
      s1,s2,...sn,...
    , back_inserter< clear<s1>::type >
    >::type
```

if `has_push_back<`$s_1$`>::value == true` and

```
reverse_x<
      s1,s2,...sn,...
    , front_inserter< clear<s1>::type >
    >::type
```

otherwise.

`typedef x<`$s_1$`,`$s_2$`,...`$s_n$`,...in>::type t;`

**Semantics:** `t` is the result of an `x` invocation with arguments $s_1$,$s_2$,... $s_n$,...`in`.

`typedef reverse_x<`$s_1$`,`$s_2$`,... `$s_n$`,... >::type t;`

**Precondition:** $s_1$ is an Extensible Sequence.

**Semantics:** `t` is equivalent to

```
x<
      s1,s2,...sn,...
    , front_inserter< clear<s1>::type >
    >::type
```

if `has_push_front<`$s_1$`>::value == true` and

```
reverse_x<
      s1,s2,...sn,...
    , back_inserter< clear<s1>::type >
    >::type
```

otherwise.

`typedef reverse_x<`$s_1$`,`$s_2$`,...`$s_n$`,... in>::type t;`

**Semantics:** `t` is the result of a `reverse_x` invocation with arguments $s_1$,$s_2$,...$s_n$,...`in`.

### Example

```
typedef transform<
      range_c<int,0,10>
    , plus<_1,int_<7> >
    , back_inserter< vector0<> >
    >::type r1;

typedef transform< r1, minus<_1,int_<2> > >::type r2;
typedef reverse_transform<
      r2
    , minus<_1,5>
    , front_inserter< vector0<> >
    >::type r3;

BOOST_MPL_ASSERT(( equal<r1, range_c<int,7,17> > ));
```

```
BOOST_MPL_ASSERT(( equal<r2, range_c<int,5,15> > ));
BOOST_MPL_ASSERT(( equal<r3, range_c<int,0,10> > ));
```

### Models

- transform
- remove
- replace

### See also

Transformation Algorithms, Inserter

## 3.2 Inserters

### 3.2.1 back_inserter

### Synopsis

```
template<
      typename Seq
    >
struct back_inserter
{
    // unspecified
    // ...
};
```

### Description

Inserts elements at the end of the sequence.

### Header

```
#include <boost/mpl/back_inserter.hpp>
```

### Model of

Inserter

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| Seq | Back Extensible Sequence | A sequence to bind the inserter to. |

### Expression semantics

The semantics of an expression are defined only where they differ from, or are not defined in Inserter.

For any Back Extensible Sequence s:

| Expression | Semantics |
|---|---|
| `back_inserter<s>` | An Inserter `in`, equivalent to `struct in : inserter<s,push_back<_1,_2> > {};` |

### Complexity

Amortized constant time.

### Example

```
typedef copy<
      range_c<int,5,10>
    , back_inserter< vector_c<int,0,1,2,3,4> >
    >::type range;

BOOST_MPL_ASSERT(( equal< range, range_c<int,0,10> > ));
```

### See also

Algorithms, Inserter, Reversible Algorithm, `inserter`, `front_inserter`, `push_back`

## 3.2.2 front_inserter

### Synopsis

```
template<
      typename Seq
    >
struct front_inserter
{
    // unspecified
    // ...
};
```

### Description

Inserts elements at the beginning of the sequence.

### Header

```
#include <boost/mpl/front_inserter.hpp>
```

### Model of

Inserter

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| Seq | Front Extensible Sequence | A sequence to bind the inserter to. |

#### Expression semantics

The semantics of an expression are defined only where they differ from, or are not defined in Inserter.

For any Front Extensible Sequence s:

| Expression | Semantics |
|---|---|
| front_inserter<s> | An Inserter in, equivalent to<br>struct in : inserter<s,push_front<_1,_2> > {}; |

#### Complexity

Amortized constant time.

#### Example

```
typedef reverse_copy<
      range_c<int,0,5>
    , front_inserter< vector_c<int,5,6,7,8,9> >
    >::type range;

BOOST_MPL_ASSERT(( equal< range, range_c<int,0,10> > ));
```

#### See also

Algorithms, Inserter, Reversible Algorithm, inserter, back_inserter, push_front

### 3.2.3 inserter

#### Synopsis

```
template<
      typename State
    , typename Operation
    >
struct inserter
{
    typedef State state;
    typedef Operation operation;
};
```

#### Description

A general-purpose model of the Inserter concept.

### Header

```
#include <boost/mpl/inserter.hpp>
```

### Model of

Inserter

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `State` | Any type | A initial state. |
| `Operation` | Binary Lambda Expression | An output operation. |

### Expression semantics

The semantics of an expression are defined only where they differ from, or are not defined in Inserter.

For any binary Lambda Expression `op` and arbitrary type `state`:

| Expression | Semantics |
|---|---|
| `inserter<op,state>` | An Inserter. |

### Complexity

Amortized constant time.

### Example

```
template< typename N > struct is_odd : bool_< ( N::value % 2 ) > {};

typedef copy<
      range_c<int,0,10>
    , inserter< // a filtering 'push_back' inserter
          vector<>
        , if_< is_odd<_2>, push_back<_1,_2>, _1 >
        >
    >::type odds;

BOOST_MPL_ASSERT(( equal< odds, vector_c<int,1,3,5,7,9>, equal_to<_,_> > ));
```

### See also

Algorithms, Inserter, Reversible Algorithm, `front_inserter`, `back_inserter`

## 3.3 Iteration Algorithms

Iteration algorithms are the basic building blocks behind many of the MPL's algorithms, and are usually the first place to look at when starting to build a new one. Abstracting away the details of sequence iteration and employing various optimizations such as recursion unrolling, they provide significant advantages over a hand-coded approach.

### 3.3.1 fold

#### Synopsis

```
template<
      typename Sequence
    , typename State
    , typename ForwardOp
    >
struct fold
{
    typedef unspecified type;
};
```

#### Description

Returns the result of the successive application of binary `ForwardOp` to the result of the previous `ForwardOp` invocation (`State` if it's the first call) and every element of the sequence in the range [`begin<Sequence>::type`, `end<Sequence>::type`) in order.

#### Header

```
#include <boost/mpl/fold.hpp>
```

#### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Forward Sequence | A sequence to iterate. |
| `State` | Any type | The initial state for the first `ForwardOp` application. |
| `ForwardOp` | Binary Lambda Expression | The operation to be executed on forward traversal. |

#### Expression semantics

For any Forward Sequence `s`, binary Lambda Expression `op`, and arbitrary type `state`:

```
typedef fold<s,state,op>::type t;
```

**Return type:** A type.

**Semantics:** Equivalent to

```
typedef iter_fold< s,state,apply<op,_1,deref<_2> > >::type t;
```

#### Complexity

Linear. Exactly `size<s>::value` applications of `op`.

#### Example

```
typedef vector<long,float,short,double,float,long,long double> types;
typedef fold<
      types
```

```
    , int_<0>
    , if_< is_float<_2>,next<_1>,_1 >
    >::type number_of_floats;

BOOST_MPL_ASSERT_RELATION( number_of_floats::value, ==, 4 );
```

**See also**

Algorithms, accumulate, reverse_fold, iter_fold, reverse_iter_fold, copy, copy_if

### 3.3.2 iter_fold

**Synopsis**

```
template<
      typename Sequence
    , typename State
    , typename ForwardOp
    >
struct iter_fold
{
    typedef unspecified type;
};
```

**Description**

Returns the result of the successive application of binary ForwardOp to the result of the previous ForwardOp invocation (State if it's the first call) and each iterator in the range [begin<Sequence>::type, end<Sequence>::type) in order.

**Header**

```
#include <boost/mpl/iter_fold.hpp>
```

**Parameters**

| Parameter | Requirement | Description |
|---|---|---|
| Sequence | Forward Sequence | A sequence to iterate. |
| State | Any type | The initial state for the first ForwardOp application. |
| ForwardOp | Binary Lambda Expression | The operation to be executed on forward traversal. |

**Expression semantics**

For any Forward Sequence s, binary Lambda Expression op, and an arbitrary type state:

```
typedef iter_fold<s,state,op>::type t;
```

**Return type:** A type.

**Semantics:** Equivalent to

```
typedef begin<Sequence>::type i1;
typedef apply<op,state,i1>::type state1;
```

```
typedef next<i1>::type i2;
typedef apply<op,state1,i2>::type state2;
...
typedef apply<op,staten−1,in>::type staten;
typedef next<in>::type last;
typedef staten t;
```

where `n == size<s>::value` and `last` is identical to `end<s>::type`; equivalent to `typedef state t;` if `empty<s>::value == true`.

#### Complexity

Linear. Exactly `size<s>::value` applications of `op`.

#### Example

```
typedef vector_c<int,5,-1,0,7,2,0,-5,4> numbers;
typedef iter_fold<
      numbers
    , begin<numbers>::type
    , if_< less< deref<_1>, deref<_2> >,_2,_1 >
    >::type max_element_iter;

BOOST_MPL_ASSERT_RELATION( deref<max_element_iter>::type::value, ==, 7 );
```

#### See also

Algorithms, `reverse_iter_fold`, `fold`, `reverse_fold`, `copy`

### 3.3.3 reverse_fold

#### Synopsis

```
template<
      typename Sequence
    , typename State
    , typename BackwardOp
    , typename ForwardOp = _1
    >
struct reverse_fold
{
    typedef unspecified type;
};
```

#### Description

Returns the result of the successive application of binary `BackwardOp` to the result of the previous `BackwardOp` invocation (`State` if it's the first call) and every element in the range [`begin<Sequence>::type`, `end<Sequence>::type`) in reverse order. If `ForwardOp` is provided, then it is applied on forward traversal to form the result that is passed to the first `BackwardOp` call.

### Header

```
#include <boost/mpl/reverse_fold.hpp>
```

### Parameters

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Forward Sequence | A sequence to iterate. |
| `State` | Any type | The initial state for the first `BackwardOp` / `ForwardOp` application. |
| `BackwardOp` | Binary Lambda Expression | The operation to be executed on backward traversal. |
| `ForwardOp` | Binary Lambda Expression | The operation to be executed on forward traversal. |

### Expression semantics

For any Forward Sequence `s`, binary Lambda Expression `backward_op` and `forward_op`, and arbitrary type `state`:

```
typedef reverse_fold< s,state,backward_op >::type t;
```

**Return type:** A type

**Semantics:** Equivalent to

```
typedef reverse_iter_fold<
      s
    , state
    , apply<backward_op,_1,deref<_2> >
    >::type t;
```

```
typedef reverse_fold< s,state,backward_op,forward_op >::type t;
```

**Return type:** A type.

**Semantics:** Equivalent to

```
typedef reverse_fold<
      Sequence
    , fold<s,state,forward_op>::type
    , backward_op
    >::type t;
```

### Complexity

Linear. Exactly `size<s>::value` applications of `backward_op` and `forward_op`.

### Example

Remove negative elements from a sequence[2].

```
typedef list_c<int,5,-1,0,-7,-2,0,-5,4> numbers;
typedef list_c<int,-1,-7,-2,-5> negatives;
typedef reverse_fold<
      numbers
```

```
    , list_c<int>
    , if_< less< _2,int_<0> >, push_front<_1,_2,>, _1 >
    >::type result;

BOOST_MPL_ASSERT(( equal< negatives,result > ));
```

**See also**

Algorithms, fold, reverse_iter_fold, iter_fold

### 3.3.4 reverse_iter_fold

**Synopsis**

```
template<
      typename Sequence
    , typename State
    , typename BackwardOp
    , typename ForwardOp = _1
    >
struct reverse_iter_fold
{
    typedef unspecified  type;
};
```

**Description**

Returns the result of the successive application of binary BackwardOp to the result of the previous BackwardOp invocation (State if it's the first call) and each iterator in the range [begin<Sequence>::type, end<Sequence>::type) in reverse order. If ForwardOp is provided, then it's applied on forward traversal to form the result which is passed to the first BackwardOp call.

**Header**

```
#include <boost/mpl/reverse_iter_fold.hpp>
```

**Parameters**

| Parameter | Requirement | Description |
|---|---|---|
| Sequence | Forward Sequence | A sequence to iterate. |
| State | Any type | The initial state for the first BackwardOp / ForwardOp application. |
| BackwardOp | Binary Lambda Expression | The operation to be executed on backward traversal. |
| ForwardOp | Binary Lambda Expression | The operation to be executed on forward traversal. |

**Expression semantics**

For any Forward Sequence s, binary Lambda Expression backward_op and forward_op, and arbitrary type state:

[2] See remove_if for a more compact way to do this.

```
typedef reverse_iter_fold< s,state,backward_op >::type t;
```

**Return type:** A type.

**Semantics:** Equivalent to

```
typedef begin<s>::type i1;
typedef next<i1>::type i2;
...
typedef next<in>::type last;
typedef apply<backward_op,state,in>::type staten;
typedef apply<backward_op,staten,in−1>::type staten−1;
...
typedef apply<backward_op,state2,i1>::type state1;
typedef state1 t;
```

where `n == size<s>::value` and `last` is identical to `end<s>::type`; equivalent to `typedef state t;` if `empty<s>::value == true`.

```
typedef reverse_iter_fold< s,state,backward_op,forward_op >::type t;
```

**Return type:** A type.

**Semantics:** Equivalent to

```
typedef reverse_iter_fold<
      Sequence
    , iter_fold<s,state,forward_op>::type
    , backward_op
    >::type t;
```

### Complexity

Linear. Exactly `size<s>::value` applications of `backward_op` and `forward_op`.

### Example

Build a list of iterators to the negative elements in a sequence.

```
typedef vector_c<int,5,-1,0,-7,-2,0,-5,4> numbers;
typedef list_c<int,-1,-7,-2,-5> negatives;
typedef reverse_iter_fold<
      numbers
    , list<>
    , if_< less< deref<_2>,int_<0> >, push_front<_1,_2>, _1 >
    >::type iters;

BOOST_MPL_ASSERT(( equal<
      negatives
    , transform_view< iters,deref<_1> >
    > ));
```

### See also

Algorithms, `iter_fold`, `reverse_fold`, `fold`

### 3.3.5 accumulate

#### Synopsis

```
template<
      typename Sequence
    , typename State
    , typename ForwardOp
    >
struct accumulate
{
    typedef unspecified type;
};
```

#### Description

Returns the result of the successive application of binary `ForwardOp` to the result of the previous `ForwardOp` invocation (`State` if it's the first call) and every element of the sequence in the range [`begin<Sequence>::type`, `end<Sequence>::type`) in order. [*Note:* `accumulate` is a synonym for `fold` — *end note*]

#### Header

```
#include <boost/mpl/accumulate.hpp>
```

#### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Forward Sequence | A sequence to iterate. |
| `State` | Any type | The initial state for the first `ForwardOp` application. |
| `ForwardOp` | Binary Lambda Expression | The operation to be executed on forward traversal. |

#### Expression semantics

For any Forward Sequence `s`, binary Lambda Expression `op`, and arbitrary type `state`:

```
typedef accumulate<s,state,op>::type t;
```

**Return type:** A type.

**Semantics:** Equivalent to

```
typedef fold<s,state,op>::type t;
```

#### Complexity

Linear. Exactly `size<s>::value` applications of `op`.

#### Example

```
typedef vector<long,float,short,double,float,long,long double> types;
typedef accumulate<
      types
```

```
    , int_<0>
    , if_< is_float<_2>,next<_1>,_1 >
    >::type number_of_floats;

BOOST_MPL_ASSERT_RELATION( number_of_floats::value, ==, 4 );
```

**See also**

Algorithms, fold, reverse_fold, iter_fold, reverse_iter_fold, copy, copy_if

## 3.4 Querying Algorithms

### 3.4.1 find

**Synopsis**

```
template<
      typename Sequence
    , typename T
    >
struct find
{
    typedef unspecified type;
};
```

**Description**

Returns an iterator to the first occurrence of type T in a Sequence.

**Header**

```
#include <boost/mpl/find.hpp>
```

**Parameters**

| Parameter | Requirement | Description |
|---|---|---|
| Sequence | Forward Sequence | A sequence to search in. |
| T | Any type | A type to search for. |

**Expression semantics**

For any Forward Sequence s and arbitrary type t:

```
typedef find<s,t>::type i;
```

**Return type:** Forward Iterator.

**Semantics:** Equivalent to

```
typedef find_if<s, is_same<_,t> >::type i;
```

### Complexity

Linear. At most `size<s>::value` comparisons for identity.

### Example

```
typedef vector<char,int,unsigned,long,unsigned long> types;
typedef find<types,unsigned>::type iter;

BOOST_MPL_ASSERT(( is_same< deref<iter>::type, unsigned > ));
BOOST_MPL_ASSERT_RELATION( iter::pos::value, ==, 2 );
```

### See also

Querying Algorithms, `contains`, `find_if`, `count`, `lower_bound`

## 3.4.2 find_if

### Synopsis

```
template<
      typename Sequence
    , typename Pred
    >
struct find_if
{
    typedef unspecified type;
};
```

### Description

Returns an iterator to the first element in `Sequence` that satisfies the predicate `Pred`.

### Header

```
#include <boost/mpl/find_if.hpp>
```

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Forward Sequence | A sequence to search in. |
| `Pred` | Unary Lambda Expression | A search condition. |

### Expression semantics

For any Forward Sequence `s` and unary Lambda Expression `pred`:

```
typedef find_if<s,pred>::type i;
```

**Return type:** Forward Iterator.

**Semantics:** `i` is the first iterator in the range [`begin<s>::type`, `end<s>::type`) such that

```
apply< pred,deref<i>::type >::type::value == true
```

If no such iterator exists, `i` is identical to `end<s>::type`.

#### Complexity

Linear. At most `size<s>::value` applications of `pred`.

#### Example

```
typedef vector<char,int,unsigned,long,unsigned long> types;
typedef find_if<types, is_same<_1,unsigned> >::type iter;

BOOST_MPL_ASSERT(( is_same< deref<iter>::type, unsigned > ));
BOOST_MPL_ASSERT_RELATION( iter::pos::value, ==, 2 );
```

#### See also

Querying Algorithms, `find`, `count_if`, `lower_bound`

### 3.4.3 contains

#### Synopsis

```
template<
      typename Sequence
    , typename T
    >
struct contains
{
    typedef unspecified type;
};
```

#### Description

Returns a true-valued Integral Constant if one or more elements in `Sequence` are identical to `T`.

#### Header

```
#include <boost/mpl/contains.hpp>
```

#### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Forward Sequence | A sequence to be examined. |
| `T` | Any type | A type to search for. |

#### Expression semantics

For any Forward Sequence `s` and arbitrary type `t`:

```
typedef contains<s,t>::type r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```
typedef not_< is_same<
      find<s,t>::type
    , end<s>::type
    > >::type r;
```

### Complexity

Linear. At most `size<s>::value` comparisons for identity.

### Example

```
typedef vector<char,int,unsigned,long,unsigned long> types;
BOOST_MPL_ASSERT_NOT(( contains<types,bool> ));
```

### See also

Querying Algorithms, `find`, `find_if`, `count`, `lower_bound`

## 3.4.4 count

### Synopsis

```
template<
      typename Sequence
    , typename T
    >
struct count
{
    typedef unspecified type;
};
```

### Description

Returns the number of elements in a `Sequence` that are identical to `T`.

### Header

```
#include <boost/mpl/count.hpp>
```

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Forward Sequence | A sequence to be examined. |
| `T` | Any type | A type to search for. |

#### Expression semantics

For any Forward Sequence `s` and arbitrary type `t`:

```
typedef count<s,t>::type n;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```
typedef count_if< s,is_same<_,T> >::type n;
```

#### Complexity

Linear. Exactly `size<s>::value` comparisons for identity.

#### Example

```
typedef vector<int,char,long,short,char,short,double,long> types;
typedef count<types, short>::type n;

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

#### See also

Querying Algorithms, `count_if`, `find`, `find_if`, `contains`, `lower_bound`

### 3.4.5 count_if

#### Synopsis

```
template<
      typename Sequence
    , typename Pred
    >
struct count_if
{
    typedef unspecified type;
};
```

#### Description

Returns the number of elements in `Sequence` that satisfy the predicate `Pred`.

#### Header

```
#include <boost/mpl/count_if.hpp>
```

#### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Forward Sequence | A sequence to be examined. |
| `Pred` | Unary Lambda Expression | A count condition. |

#### Expression semantics

For any Forward Sequence `s` and unary Lambda Expression `pred`:

```
typedef count_if<s,pred>::type n;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```
typedef lambda<pred>::type p;
typedef fold<
      s
    , long_<0>
    , if_< apply_wrap1<p,_2>, next<_1>, _1 >
    >::type n;
```

#### Complexity

Linear. Exactly `size<s>::value` applications of `pred`.

#### Example

```
typedef vector<int,char,long,short,char,long,double,long> types;

BOOST_MPL_ASSERT_RELATION( (count_if< types, is_float<_> >::value), ==, 1 );
BOOST_MPL_ASSERT_RELATION( (count_if< types, is_same<_,char> >::value), ==, 2 );
BOOST_MPL_ASSERT_RELATION( (count_if< types, is_same<_,void> >::value), ==, 0 );
```

#### See also

Querying Algorithms, `count`, `find`, `find_if`, `contains`

### 3.4.6 lower_bound

#### Synopsis

```
template<
      typename Sequence
    , typename T
    , typename Pred = less<_1,_2>
    >
struct lower_bound
{
    typedef unspecified type;
};
```

#### Description

Returns the first position in the sorted `Sequence` where `T` could be inserted without violating the ordering.

### Header

```
#include <boost/mpl/lower_bound.hpp>
```

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Forward Sequence | A sorted sequence to search in. |
| `T` | Any type | A type to search a position for. |
| `Pred` | Binary Lambda Expression | A search criteria. |

### Expression semantics

For any sorted Forward Sequence `s`, binary Lambda Expression `pred`, and arbitrary type `x`:

```
typedef lower_bound< s,x,pred >::type i;
```

**Return type:** Forward Iterator.

**Semantics:** `i` is the furthermost iterator in [`begin<s>::type`, `end<s>::type`) such that, for every iterator `j` in [`begin<s>::type`, `i`),

```
apply< pred, deref<j>::type, x >::type::value == true
```

### Complexity

The number of comparisons is logarithmic: at most $\log_2($ `size<s>::value` $) + 1$. If `s` is a Random Access Sequence then the number of steps through the range is also logarithmic; otherwise, the number of steps is proportional to `size<s>::value`.

### Example

```
typedef vector_c<int,1,2,3,3,3,5,8> numbers;
typedef lower_bound< numbers, int_<3> >::type iter;

BOOST_MPL_ASSERT_RELATION(
      (distance< begin<numbers>::type,iter >::value), ==, 2
    );

BOOST_MPL_ASSERT_RELATION( deref<iter>::type::value, ==, 3 );
```

### See also

Querying Algorithms, `upper_bound`, `find`, `find_if`, `min_element`

## 3.4.7 upper_bound

### Synopsis

```
template<
      typename Sequence
    , typename T
```

```
    , typename Pred = less<_1,_2>
    >
struct upper_bound
{
    typedef unspecified type;
};
```

### Description

Returns the last position in the sorted `Sequence` where `T` could be inserted without violating the ordering.

### Header

```
#include <boost/mpl/upper_bound.hpp>
```

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Forward Sequence | A sorted sequence to search in. |
| `T` | Any type | A type to search a position for. |
| `Pred` | Binary Lambda Expression | A search criteria. |

### Expression semantics

For any sorted Forward Sequence `s`, binary Lambda Expression `pred`, and arbitrary type `x`:

```
typedef upper_bound< s,x,pred >::type i;
```

**Return type:** Forward Iterator

**Semantics:** `i` is the furthermost iterator in [`begin<s>::type`, `end<s>::type`) such that, for every iterator `j` in [`begin<s>::type`, `i`),

```
apply< pred, x, deref<j>::type >::type::value == false
```

### Complexity

The number of comparisons is logarithmic: at most $\log_2($ `size<s>::value` $) + 1$. If `s` is a Random Access Sequence then the number of steps through the range is also logarithmic; otherwise, the number of steps is proportional to `size<s>::value`.

### Example

```
typedef vector_c<int,1,2,3,3,3,5,8> numbers;
typedef upper_bound< numbers, int_<3> >::type iter;

BOOST_MPL_ASSERT_RELATION(
      (distance< begin<numbers>::type,iter >::value), ==, 5
    );

BOOST_MPL_ASSERT_RELATION( deref<iter>::type::value, ==, 5 );
```

### See also

Querying Algorithms, lower_bound, find, find_if, min_element

## 3.4.8 min_element

### Synopsis

```
template<
      typename Sequence
    , typename Pred = less<_1,_2>
    >
struct min_element
{
    typedef unspecified type;
};
```

### Description

Returns an iterator to the smallest element in Sequence.

### Header

```
#include <boost/mpl/min_element.hpp>
```

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| Sequence | Forward Sequence | A sequence to be searched. |
| Pred | Binary Lambda Expression | A comparison criteria. |

### Expression semantics

For any Forward Sequence s and binary Lambda Expression pred:

```
typedef min_element<s,pred>::type i;
```

**Return type:** Forward Iterator.

**Semantics:** i is the first iterator in [begin<s>::type, end<s>::type) such that for every iterator j in [begin<s>::type, end<s>::type),

```
apply< pred, deref<j>::type, deref<i>::type >::type::value == false
```

### Complexity

Linear. Zero comparisons if s is empty, otherwise exactly size<s>::value - 1 comparisons.

### Example

```
typedef vector<bool,char[50],long,double> types;
typedef min_element<
```

```
      transform_view< types,sizeof_<_1> >
    >::type iter;

BOOST_MPL_ASSERT(( is_same< deref<iter::base>::type, bool> ));
```

**See also**

Querying Algorithms, `max_element`, `find_if`, `upper_bound`, `find`

### 3.4.9 max_element

**Synopsis**

```
template<
      typename Sequence
    , typename Pred = less<_1,_2>
    >
struct max_element
{
    typedef unspecified type;
};
```

**Description**

Returns an iterator to the largest element in `Sequence`.

**Header**

```
#include <boost/mpl/max_element.hpp>
```

**Parameters**

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Forward Sequence | A sequence to be searched. |
| `Pred` | Binary Lambda Expression | A comparison criteria. |

**Expression semantics**

For any Forward Sequence `s` and binary Lambda Expression `pred`:

```
typedef max_element<s,pred>::type i;
```

**Return type:** Forward Iterator.

**Semantics:** `i` is the first iterator in [`begin<s>::type`, `end<s>::type`) such that for every iterator `j` in [`begin<s>::type`, `end<s>::type`),

```
apply< pred, deref<i>::type, deref<j>::type >::type::value == false
```

**Complexity**

Linear. Zero comparisons if `s` is empty, otherwise exactly `size<s>::value - 1` comparisons.

### Example

```
typedef vector<bool,char[50],long,double> types;
typedef max_element<
      transform_view< types,sizeof_<_1> >
    >::type iter;

BOOST_MPL_ASSERT(( is_same< deref<iter::base>::type, char[50]> ));
```

### See also

Querying Algorithms, `min_element`, `find_if`, `upper_bound`, `find`

## 3.4.10 equal

### Synopsis

```
template<
      typename Seq1
    , typename Seq2
    , typename Pred = is_same<_1,_2>
    >
struct equal
{
    typedef unspecified type;
};
```

### Description

Returns a true-valued Integral Constant if the two sequences `Seq1` and `Seq2` are identical when compared element-by-element.

### Header

```
#include <boost/mpl/equal.hpp>
```

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Seq1`, `Seq2` | Forward Sequence | Sequences to compare. |
| `Pred` | Binary Lambda Expression | A comparison criterion. |

### Expression semantics

For any Forward Sequences `s1` and `s2` and a binary Lambda Expression `pred`:

```
typedef equal<s1,s2,pred>::type c;
```

**Return type:** Integral Constant

**Semantics:** `c::value == true` is and only if `size<s1>::value == size<s2>::value` and for every iterator `i` in [`begin<s1>::type`, `end<s1>::type`) `deref<i>::type` is identical to

```
advance< begin<s2>::type, distance< begin<s1>::type,i >::type >::type
```

**Complexity**

Linear. At most `size<s1>::value` comparisons.

**Example**

```
typedef vector<char,int,unsigned,long,unsigned long> s1;
typedef list<char,int,unsigned,long,unsigned long> s2;

BOOST_MPL_ASSERT(( equal<s1,s2> ));
```

**See also**

Querying Algorithms, `find`, `find_if`

### 3.5 Transformation Algorithms

According to their name, MPL's *transformation*, or *sequence-building algorithms* provide the tools for building new sequences from the existing ones by performing some kind of transformation. A typical transformation alogrithm takes one or more input sequences and a transformation metafunction/predicate, and returns a new sequence built according to the algorithm's semantics through the means of its Inserter argument, which plays a role similar to the role of run-time Output Iterator.

Every transformation algorithm is a Reversible Algorithm, providing an accordingly named `reverse_` counterpart carrying the transformation in the reverse order. Thus, all sequence-building algorithms come in pairs, for instance `replace` / `reverse_replace`. In presence of variability of the output sequence's properties such as front or backward extensibility, the existence of the bidirectional algorithms allows for the most efficient way to perform the required transformation.

#### 3.5.1 copy

**Synopsis**

```
template<
      typename Sequence
    , typename In = unspecified
    >
struct copy
{
    typedef unspecified type;
};
```

**Description**

Returns a copy of the original sequence.

[*Note:* This wording applies to a no-inserter version(s) of the algorithm. See the *Expression semantics* subsection for a precise specification of the algorithm's details in all cases — *end note*]

### Header

```
#include <boost/mpl/copy.hpp>
```

### Model of

Reversible Algorithm

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Forward Sequence | A sequence to copy. |
| `In` | Inserter | An inserter. |

### Expression semantics

The semantics of an expression are defined only where they differ from, or are not defined in Reversible Algorithm.

For any Forward Sequence `s`, and an Inserter `in`:

```
typedef copy<s,in>::type r;
```

**Return type:** A type.

**Semantics:** Equivalent to

```
typedef fold< s,in::state,in::operation >::type r;
```

### Complexity

Linear. Exactly `size<s>::value` applications of `in::operation`.

### Example

```
typedef vector_c<int,0,1,2,3,4,5,6,7,8,9> numbers;
typedef copy<
      range_c<int,10,20>
    , back_inserter< numbers >
    >::type result;

BOOST_MPL_ASSERT_RELATION( size<result>::value, ==, 20 );
BOOST_MPL_ASSERT(( equal< result,range_c<int,0,20> > ));
```

### See also

Transformation Algorithms, Reversible Algorithm, `reverse_copy`, `copy_if`, `transform`

## 3.5.2 copy_if

### Synopsis

```
template<
      typename Sequence
```

```
    , typename Pred
    , typename In = unspecified
    >
struct copy_if
{
    typedef unspecified type;
};
```

### Description

Returns a filtered copy of the original sequence containing the elements that satisfy the predicate `Pred`.

[*Note:* This wording applies to a no-inserter version(s) of the algorithm. See the *Expression semantics* subsection for a precise specification of the algorithm's details in all cases — *end note*]

### Header

```
#include <boost/mpl/copy_if.hpp>
```

### Model of

Reversible Algorithm

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Forward Sequence | A sequence to copy. |
| `Pred` | Unary Lambda Expression | A copying condition. |
| `In` | Inserter | An inserter. |

### Expression semantics

The semantics of an expression are defined only where they differ from, or are not defined in Reversible Algorithm.

For any Forward Sequence `s`, an unary Lambda Expression `pred`, and an Inserter `in`:

```
typedef copy_if<s,pred,in>::type r;
```

**Return type:** A type.

**Semantics:** Equivalent to

```
typedef lambda<pred>::type p;
typedef lambda<in::operation>::type op;

typedef fold<
      s
    , in::state
    , eval_if<
          apply_wrap1<p,_2>
        , apply_wrap2<op,_1,_2>
        , identity<_1>
        >
```

```
    >::type r;
```

#### Complexity

Linear. Exactly `size<s>::value` applications of `pred`, and at most `size<s>::value` applications of `in::operation`.

#### Example

```
typedef copy_if<
      range_c<int,0,10>
    , less< _1, int_<5> >
    , back_inserter< vector<> >
    >::type result;

BOOST_MPL_ASSERT_RELATION( size<result>::value, ==, 5 );
BOOST_MPL_ASSERT(( equal<result,range_c<int,0,5> > ));
```

#### See also

Transformation Algorithms, Reversible Algorithm, `reverse_copy_if`, `copy`, `remove_if`, `replace_if`

### 3.5.3 transform

#### Synopsis

```
template<
      typename Seq
    , typename Op
    , typename In = unspecified
    >
struct transform
{
    typedef unspecified type;
};

template<
      typename Seq1
    , typename Seq2
    , typename BinaryOp
    , typename In = unspecified
    >
struct transform
{
    typedef unspecified type;
};
```

#### Description

`transform` is an overloaded name:

— `transform<Seq,Op>` returns a transformed copy of the original sequence produced by applying an unary transformation `Op` to every element in the [`begin<Sequence>::type`, `end<Sequence>::type`) range.

— `transform<Seq1,Seq2,Op>` returns a new sequence produced by applying a binary transformation `BinaryOp` to a pair of elements ($e1_1$, $e2_1$) from the corresponding [`begin<Seq1>::type`, `end<Seq1>::type`) and [`begin<Seq2>::type`, `end<Seq2>::type`) ranges.

[*Note:* This wording applies to a no-inserter version(s) of the algorithm. See the *Expression semantics* subsection for a precise specification of the algorithm's details in all cases — *end note*]

### Header

```
#include <boost/mpl/transform.hpp>
```

### Model of

Reversible Algorithm

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence`, `Seq1`, `Seq2` | Forward Sequence | Sequences to transform. |
| `Op`, `BinaryOp` | Lambda Expression | A transformation. |
| `In` | Inserter | An inserter. |

### Expression semantics

The semantics of an expression are defined only where they differ from, or are not defined in Reversible Algorithm.

For any Forward Sequences `s`, `s1` and `s2`, Lambda Expressions `op` and `op2`, and an Inserter `in`:

```
typedef transform<s,op,in>::type r;
```

**Return type:** A type.

**Postcondition:** Equivalent to

```
typedef lambda<op>::type f;
typedef lambda<in::operation>::type in_op;

typedef fold<
      s
    , in::state
    , bind< in_op, _1, bind<f, _2> >
    >::type r;
```

```
typedef transform<s1,s2,op,in>::type r;
```

**Return type:** A type.

**Postcondition:** Equivalent to

```
typedef lambda<op2>::type f;
typedef lambda<in::operation>::type in_op;

typedef fold<
```

```
      pair_view<s1,s2>
    , in::state
    , bind<
          in_op
        , _1
        , bind<f, bind<first<>,_2>, bind<second<>,_2> >
        >
    >::type r;
```

### Complexity

Linear. Exactly `size<s>::value` / `size<s1>::value` applications of `op` / `op2` and `in::operation`.

### Example

```
typedef vector<char,short,int,long,float,double> types;
typedef vector<char*,short*,int*,long*,float*,double*> pointers;
typedef transform< types,boost::add_pointer<_1> >::type result;

BOOST_MPL_ASSERT(( equal<result,pointers> ));
```

### See also

Transformation Algorithms, Reversible Algorithm, `reverse_transform`, `copy`, `replace_if`

## 3.5.4 replace

### Synopsis

```
template<
      typename Sequence
    , typename OldType
    , typename NewType
    , typename In = unspecified
    >
struct replace
{
    typedef unspecified type;
};
```

### Description

Returns a copy of the original sequence where every type identical to `OldType` has been replaced with `NewType`.

[*Note:* This wording applies to a no-inserter version(s) of the algorithm. See the *Expression semantics* subsection for a precise specification of the algorithm's details in all cases — *end note*]

### Header

```
#include <boost/mpl/replace.hpp>
```

### Model of

Reversible Algorithm

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Forward Sequence | A original sequence. |
| `OldType` | Any type | A type to be replaced. |
| `NewType` | Any type | A type to replace with. |
| `In` | Inserter | An inserter. |

### Expression semantics

The semantics of an expression are defined only where they differ from, or are not defined in Reversible Algorithm.

For any Forward Sequence `s`, an Inserter `in`, and arbitrary types `x` and `y`:

```
typedef replace<s,x,y,in>::type r;
```

**Return type:** A type.

**Semantics:** Equivalent to

```
typedef replace_if< s,y,is_same<_,x>,in >::type r;
```

### Complexity

Linear. Performs exactly `size<s>::value` comparisons for identity / insertions.

### Example

```
typedef vector<int,float,char,float,float,double> types;
typedef vector<int,double,char,double,double,double> expected;
typedef replace< types,float,double >::type result;

BOOST_MPL_ASSERT(( equal< result,expected > ));
```

### See also

Transformation Algorithms, Reversible Algorithm, `reverse_replace`, `replace_if`, `remove`, `transform`

## 3.5.5 replace_if

### Synopsis

```
template<
      typename Sequence
    , typename Pred
    , typename In = unspecified
    >
struct replace_if
{
```

```
    typedef unspecified type;
};
```

### Description

Returns a copy of the original sequence where every type that satisfies the predicate `Pred` has been replaced with `NewType`.

[*Note:* This wording applies to a no-inserter version(s) of the algorithm. See the *Expression semantics* subsection for a precise specification of the algorithm's details in all cases — *end note*]

### Header

```
#include <boost/mpl/replace_if.hpp>
```

### Model of

Reversible Algorithm

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Forward Sequence | An original sequence. |
| `Pred` | Unary Lambda Expression | A replacement condition. |
| `NewType` | Any type | A type to replace with. |
| `In` | Inserter | An inserter. |

### Expression semantics

The semantics of an expression are defined only where they differ from, or are not defined in Reversible Algorithm.

For any Forward Sequence `s`, an unary Lambda Expression `pred`, an Inserter `in`, and arbitrary type `x`:

```
typedef replace_if<s,pred,x,in>::type r;
```

**Return type:** A type.

**Semantics:** Equivalent to

```
typedef lambda<pred>::type p;
typedef transform< s, if_< apply_wrap1<p,_1>,x,_1>, in >::type r;
```

### Complexity

Linear. Performs exactly `size<s>::value` applications of `pred`, and at most `size<s>::value` insertions.

### Example

```
typedef vector_c<int,1,4,5,2,7,5,3,5> numbers;
typedef vector_c<int,1,4,0,2,0,0,3,0> expected;
typedef replace_if< numbers, greater<_,int_<4> >, int_<0> >::type result;
```

```
BOOST_MPL_ASSERT(( equal< result,expected, equal_to<_,_> > ));
```

**See also**

Transformation Algorithms, Reversible Algorithm, `reverse_replace_if`, `replace`, `remove_if`, `transform`

### 3.5.6 remove

**Synopsis**

```
template<
      typename Sequence
    , typename T
    , typename In = unspecified
    >
struct remove
{
    typedef unspecified type;
};
```

**Description**

Returns a new sequence that contains all elements from [`begin<Sequence>::type`, `end<Sequence>::type`) range except those that are identical to `T`.

[*Note:* This wording applies to a no-inserter version(s) of the algorithm. See the *Expression semantics* subsection for a precise specification of the algorithm's details in all cases — *end note*]

**Header**

```
#include <boost/mpl/remove.hpp>
```

**Model of**

Reversible Algorithm

**Parameters**

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Forward Sequence | An original sequence. |
| `T` | Any type | A type to be removed. |
| `In` | Inserter | An inserter. |

**Expression semantics**

The semantics of an expression are defined only where they differ from, or are not defined in Reversible Algorithm.

For any Forward Sequence `s`, an Inserter `in`, and arbitrary type `x`:

```
typedef remove<s,x,in>::type r;
```

**Return type:** A type.

**Semantics:** Equivalent to

```
typedef remove_if< s,is_same<_,x>,in >::type r;
```

### Complexity

Linear. Performs exactly `size<s>::value` comparisons for equality, and at most `size<s>::value` insertions.

### Example

```
typedef vector<int,float,char,float,float,double>::type types;
typedef remove< types,float >::type result;

BOOST_MPL_ASSERT(( equal< result, vector<int,char,double> > ));
```

### See also

Transformation Algorithms, Reversible Algorithm, `reverse_remove`, `remove_if`, `copy`, `replace`

## 3.5.7 remove_if

### Synopsis

```
template<
      typename Sequence
    , typename Pred
    , typename In = unspecified
    >
struct remove_if
{
    typedef unspecified type;
};
```

### Description

Returns a new sequence that contains all the elements from [`begin<Sequence>::type`, `end<Sequence>::type`) range except those that satisfy the predicate `Pred`.

[*Note:* This wording applies to a no-inserter version(s) of the algorithm. See the *Expression semantics* subsection for a precise specification of the algorithm's details in all cases — *end note*]

### Header

```
#include <boost/mpl/remove_if.hpp>
```

### Model of

Reversible Algorithm

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Forward Sequence | An original sequence. |
| `Pred` | Unary Lambda Expression | A removal condition. |
| `In` | Inserter | An inserter. |

#### Expression semantics

The semantics of an expression are defined only where they differ from, or are not defined in Reversible Algorithm.

For any Forward Sequence `s`, and an Inserter `in`, and an unary Lambda Expression `pred`:

```
typedef remove_if<s,pred,in>::type r;
```

**Return type:** A type.

**Semantics:** Equivalent to

```
typedef lambda<pred>::type p;
typedef lambda<in::operation>::type op;

typedef fold<
      s
    , in::state
    , eval_if<
          apply_wrap1<p,_2>
        , identity<_1>
        , apply_wrap2<op,_1,_2>
        >
    >::type r;
```

#### Complexity

Linear. Performs exactly `size<s>::value` applications of `pred`, and at most `size<s>::value` insertions.

#### Example

```
typedef vector_c<int,1,4,5,2,7,5,3,5>::type numbers;
typedef remove_if< numbers, greater<_,int_<4> > >::type result;

BOOST_MPL_ASSERT(( equal< result,vector_c<int,1,4,2,3>,equal_to<_,_> > ));
```

#### See also

Transformation Algorithms, Reversible Algorithm, `reverse_remove_if`, `remove`, `copy_if`, `replace_if`

### 3.5.8 unique

#### Synopsis

```
template<
      typename Seq
    , typename Pred
    , typename In = unspecified
```

```
    >
struct unique
{
    typedef unspecified type;
};
```

### Description

Returns a sequence of the initial elements of every subrange of the original sequence `Seq` whose elements are all the same.

[*Note:* This wording applies to a no-inserter version(s) of the algorithm. See the *Expression semantics* subsection for a precise specification of the algorithm's details in all cases — *end note*]

### Header

```
#include <boost/mpl/unique.hpp>
```

### Model of

Reversible Algorithm

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Forward Sequence | An original sequence. |
| `Pred` | Binary Lambda Expression | An equivalence relation. |
| `In` | Inserter | An inserter. |

### Expression semantics

The semantics of an expression are defined only where they differ from, or are not defined in Reversible Algorithm.

For any Forward Sequence `s`, a binary Lambda Expression `pred`, and an Inserter `in`:

```
typedef unique<s,pred,in>::type r;
```

**Return type:** A type.

**Semantics:** If `size<s>::value <= 1`, then equivalent to

```
typedef copy<s,in>::type r;
```

otherwise equivalent to

```
typedef lambda<pred>::type p;
typedef lambda<in::operation>::type in_op;
typedef apply_wrap2<
      in_op
    , in::state
    , front<types>::type
    >::type in_state;

typedef fold<
```

```
      s
    , pair< in_state, front<s>::type >
    , eval_if<
          apply_wrap2<p, second<_1>, _2>
        , identity< first<_1> >
        , apply_wrap2<in_op, first<_1>, _2>
        >
    >::type::first r;
```

#### Complexity

Linear. Performs exactly `size<s>::value - 1` applications of `pred`, and at most `size<s>::value` insertions.

#### Example

```
typedef vector<int,float,float,char,int,int,int,double> types;
typedef vector<int,float,char,int,double> expected;
typedef unique< types, is_same<_1,_2> >::type result;

BOOST_MPL_ASSERT(( equal< result,expected > ));
```

#### See also

Transformation Algorithms, Reversible Algorithm, `reverse_unique`, `remove`, `copy_if`, `replace_if`

### 3.5.9 partition

#### Synopsis

```
template<
      typename Seq
    , typename Pred
    , typename In1 = unspecified
    , typename In2 = unspecified
    >
struct partition
{
    typedef unspecified type;
};
```

#### Description

Returns a pair of sequences together containing all elements in the range [`begin<Seq>::type`, `end<Seq>::type`) split into two groups based on the predicate `Pred`. `partition` is a synonym for `stable_partition`.

[*Note:* This wording applies to a no-inserter version(s) of the algorithm. See the *Expression semantics* subsection for a precise specification of the algorithm's details in all cases — *end note*]

#### Header

```
#include <boost/mpl/partition.hpp>
```

### Model of

Reversible Algorithm

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Seq` | Forward Sequence | An original sequence. |
| `Pred` | Unary Lambda Expression | A partitioning predicate. |
| `In1`, `In2` | Inserter | Output inserters. |

### Expression semantics

The semantics of an expression are defined only where they differ from, or are not defined in Reversible Algorithm.

For any Forward Sequence `s`, an unary Lambda Expression `pred`, and Inserters `in1` and `in2`:

```
typedef partition<s,pred,in1,in2>::type r;
```

**Return type:** A `pair`.

**Semantics:** Equivalent to

```
typedef stable_partition<s,pred,in1,in2>::type r;
```

### Complexity

Linear. Exactly `size<s>::value` applications of `pred`, and `size<s>::value` of summarized `in1::operation` / `in2::operation` applications.

### Example

```
template< typename N > struct is_odd : bool_<(N::value % 2)> {};

typedef partition<
      range_c<int,0,10>
    , is_odd<_1>
    , back_inserter< vector<> >
    , back_inserter< vector<> >
    >::type r;

BOOST_MPL_ASSERT(( equal< r::first, vector_c<int,1,3,5,7,9> > ));
BOOST_MPL_ASSERT(( equal< r::second, vector_c<int,0,2,4,6,8> > ));
```

### See also

Transformation Algorithms, Reversible Algorithm, `reverse_partition`, `stable_partition`, `sort`

## 3.5.10 stable_partition

### Synopsis

```
template<
```

```
      typename Seq
    , typename Pred
    , typename In1 = unspecified
    , typename In2 = unspecified
    >
struct stable_partition
{
    typedef unspecified type;
};
```

### Description

Returns a pair of sequences together containing all elements in the range [`begin<Seq>::type`, `end<Seq>::type`) split into two groups based on the predicate `Pred`. `stable_partition` is guaranteed to preserve the relative order of the elements in the resulting sequences.

[*Note:* This wording applies to a no-inserter version(s) of the algorithm. See the *Expression semantics* subsection for a precise specification of the algorithm's details in all cases — *end note*]

### Header

```
#include <boost/mpl/stable_partition.hpp>
```

### Model of

Reversible Algorithm

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Seq` | Forward Sequence | An original sequence. |
| `Pred` | Unary Lambda Expression | A partitioning predicate. |
| `In1`, `In2` | Inserter | Output inserters. |

### Expression semantics

The semantics of an expression are defined only where they differ from, or are not defined in Reversible Algorithm.

For any Forward Sequence `s`, an unary Lambda Expression `pred`, and Inserters `in1` and `in2`:

```
typedef stable_partition<s,pred,in1,in2>::type r;
```

**Return type:** A `pair`.

**Semantics:** Equivalent to

```
typedef lambda<pred>::type p;
typedef lambda<in1::operation>::type in1_op;
typedef lambda<in2::operation>::type in2_op;

typedef fold<
      s
    , pair< in1::state, in2::state >
```

```
            , if_<
                  apply_wrap1<p,_2>
                , pair< apply_wrap2<in1_op,first<_1>,_2>, second<_1> >
                , pair< first<_1>, apply_wrap2<in2_op,second<_1>,_2> >
                >
            >::type r;
```

### Complexity

Linear. Exactly `size<s>::value` applications of `pred`, and `size<s>::value` of summarized `in1::operation` / `in2::operation` applications.

### Example

```
template< typename N > struct is_odd : bool_<(N::value % 2)> {};

typedef stable_partition<
      range_c<int,0,10>
    , is_odd<_1>
    , back_inserter< vector<> >
    , back_inserter< vector<> >
    >::type r;

BOOST_MPL_ASSERT(( equal< r::first, vector_c<int,1,3,5,7,9> > ));
BOOST_MPL_ASSERT(( equal< r::second, vector_c<int,0,2,4,6,8> > ));
```

### See also

Transformation Algorithms, Reversible Algorithm, `reverse_stable_partition`, `partition`, `sort`, `transform`

## 3.5.11 sort

### Synopsis

```
template<
      typename Seq
    , typename Pred = less<_1,_2>
    , typename In = unspecified
    >
struct sort
{
    typedef unspecified type;
};
```

### Description

Returns a new sequence of all elements in the range [`begin<Seq>::type`, `end<Seq>::type`) sorted according to the ordering relation `Pred`.

[*Note:* This wording applies to a no-inserter version(s) of the algorithm. See the *Expression semantics* subsection for a precise specification of the algorithm's details in all cases — *end note*]

### Header

```
#include <boost/mpl/sort.hpp>
```

### Model of

Reversible Algorithm

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Seq` | Forward Sequence | An original sequence. |
| `Pred` | Binary Lambda Expression | An ordering relation. |
| `In` | Inserter | An inserter. |

### Expression semantics

The semantics of an expression are defined only where they differ from, or are not defined in Reversible Algorithm.

For any Forward Sequence `s`, a binary Lambda Expression `pred`, and an Inserter `in`:

```
typedef sort<s,pred,in>::type r;
```

**Return type:** A type.

**Semantics:** If `size<s>::value <= 1`, equivalent to

```
typedef copy<s,in>::type r;
```

otherwise equivalent to

```
typedef back_inserter< vector<> > aux_in;
typedef lambda<pred>::type p;

typedef begin<s>::type pivot;
typedef partition<
      iterator_range< next<pivot>::type, end<s>::type >
    , apply_wrap2<p,_1,deref<pivot>::type>
    , aux_in
    , aux_in
    >::type partitioned;

typedef sort<partitioned::first,p,aux_in >::type part1;
typedef sort<partitioned::second,p,aux_in >::type part2;

typedef copy<
      joint_view<
          joint_view<part1,single_view< deref<pivot>::type > >
        , part2
        >
    , in
    >::type r;
```

### Complexity

Average *O(n log(n))* where *n* == `size<s>::value`, quadratic at worst.

### Example

```
typedef vector_c<int,3,4,0,-5,8,-1,7> numbers;
typedef vector_c<int,-5,-1,0,3,4,7,8> expected;
typedef sort<numbers>::type result;

BOOST_MPL_ASSERT(( equal< result, expected, equal_to<_,_> > ));
```

### See also

Transformation Algorithms, Reversible Algorithm, `partition`

## 3.5.12 reverse

### Synopsis

```
template<
      typename Sequence
    , typename In = unspecified
    >
struct reverse
{
    typedef unspecified type;
};
```

### Description

Returns a reversed copy of the original sequence. `reverse` is a synonym for `reverse_copy`.

[*Note:* This wording applies to a no-inserter version(s) of the algorithm. See the *Expression semantics* subsection for a precise specification of the algorithm's details in all cases — *end note*]

### Header

```
#include <boost/mpl/reverse.hpp>
```

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Forward Sequence | A sequence to reverse. |
| `In` | Inserter | An inserter. |

### Expression semantics

For any Forward Sequence `s`, and an Inserter `in`:

```
typedef reverse<s,in>::type r;
```

**Return type:** A type.

**Semantics:** Equivalent to

```
typedef reverse_copy<s,in>::type r;
```

### Complexity

Linear.

### Example

```
typedef vector_c<int,9,8,7,6,5,4,3,2,1,0> numbers;
typedef reverse< numbers >::type result;

BOOST_MPL_ASSERT(( equal< result, range_c<int,0,10> > ));
```

### See also

Transformation Algorithms, Reversible Algorithm, `reverse_copy`, `copy`, `copy_if`

## 3.5.13 reverse_copy

### Synopsis

```
template<
      typename Sequence
    , typename In = unspecified
    >
struct reverse_copy
{
    typedef unspecified type;
};
```

### Description

Returns a reversed copy of the original sequence.

[*Note:* This wording applies to a no-inserter version(s) of the algorithm. See the *Expression semantics* subsection for a precise specification of the algorithm's details in all cases — *end note*]

### Header

```
#include <boost/mpl/copy.hpp>
```

### Model of

Reversible Algorithm

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Forward Sequence | A sequence to copy. |
| `In` | Inserter | An inserter. |

**Expression semantics**

The semantics of an expression are defined only where they differ from, or are not defined in Reversible Algorithm.

For any Forward Sequence `s`, and an Inserter `in`:

```
typedef reverse_copy<s,in>::type r;
```

**Return type:** A type.

**Semantics:** Equivalent to

```
typedef reverse_fold< s,in::state,in::operation >::type r;
```

**Complexity**

Linear. Exactly `size<s>::value` applications of `in::operation`.

**Example**

```
typedef list_c<int,10,11,12,13,14,15,16,17,18,19>::type numbers;
typedef reverse_copy<
      range_c<int,0,10>
    , front_inserter< numbers >
    >::type result;

BOOST_MPL_ASSERT_RELATION( size<result>::value, ==, 20 );
BOOST_MPL_ASSERT(( equal< result,range_c<int,0,20> > ));
```

**See also**

Transformation Algorithms, Reversible Algorithm, `copy`, `reverse_copy_if`, `reverse_transform`

### 3.5.14 reverse_copy_if

**Synopsis**

```
template<
      typename Sequence
    , typename Pred
    , typename In = unspecified
    >
struct reverse_copy_if
{
    typedef unspecified type;
};
```

### Description

Returns a reversed, filtered copy of the original sequence containing the elements that satisfy the predicate `Pred`.

[*Note:* This wording applies to a no-inserter version(s) of the algorithm. See the *Expression semantics* subsection for a precise specification of the algorithm's details in all cases — *end note*]

### Header

```
#include <boost/mpl/copy_if.hpp>
```

### Model of

Reversible Algorithm

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Forward Sequence | A sequence to copy. |
| `Pred` | Unary Lambda Expression | A copying condition. |
| `In` | Inserter | An inserter. |

### Expression semantics

The semantics of an expression are defined only where they differ from, or are not defined in Reversible Algorithm.

For any Forward Sequence `s`, an unary Lambda Expression `pred`, and an Inserter `in`:

```
typedef reverse_copy_if<s,pred,in>::type r;
```

**Return type:** A type

**Semantics:** Equivalent to

```
typedef lambda<pred>::type p;
typedef lambda<in::operation>::type op;

typedef reverse_fold<
      s
    , in::state
    , eval_if<
          apply_wrap1<p,_2>
        , apply_wrap2<op,_1,_2>
        , identity<_1>
        >
    >::type r;
```

### Complexity

Linear. Exactly `size<s>::value` applications of `pred`, and at most `size<s>::value` applications of `in::operation`.

**Example**

```
typedef reverse_copy_if<
      range_c<int,0,10>
    , less< _1, int_<5> >
    , front_inserter< vector<> >
    >::type result;

BOOST_MPL_ASSERT_RELATION( size<result>::value, ==, 5 );
BOOST_MPL_ASSERT(( equal<result,range_c<int,0,5> > ));
```

**See also**

Transformation Algorithms, Reversible Algorithm, `copy_if`, `reverse_copy`, `remove_if`, `replace_if`

### 3.5.15 reverse_transform

**Synopsis**

```
template<
      typename Seq
    , typename Op
    , typename In = unspecified
    >
struct reverse_transform
{
    typedef unspecified type;
};

template<
      typename Seq1
    , typename Seq2
    , typename BinaryOp
    , typename In = unspecified
    >
struct reverse_transform
{
    typedef unspecified type;
};
```

**Description**

`reverse_transform` is an overloaded name:

— `reverse_transform<Seq,Op>` returns a reversed, transformed copy of the original sequence produced by applying an unary transformation `Op` to every element in the [`begin<Sequence>::type`, `end<Sequence>::type`) range.

— `reverse_transform<Seq1,Seq2,Op>` returns a new sequence produced by applying a binary transformation `BinaryOp` to a pair of elements ($e1_1$, $e2_1$) from the corresponding [`begin<Seq1>::type`, `end<Seq1>::type`) and [`begin<Seq2>::type`, `end<Seq2>::type`) ranges in reverse order.

[*Note:* This wording applies to a no-inserter version(s) of the algorithm. See the *Expression semantics* subsection for a precise specification of the algorithm's details in all cases — *end note*]

### Header

```
#include <boost/mpl/transform.hpp>
```

### Model of

Reversible Algorithm

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence`, `Seq1`, `Seq2` | Forward Sequence | Sequences to transform. |
| `Op`, `BinaryOp` | Lambda Expression | A transformation. |
| `In` | Inserter | An inserter. |

### Expression semantics

The semantics of an expression are defined only where they differ from, or are not defined in Reversible Algorithm.

For any Forward Sequences `s`, `s1` and `s2`, Lambda Expressions `op` and `op2`, and an Inserter `in`:

```
typedef reverse_transform<s,op,in>::type r;
```

**Return type:** A type.

**Postcondition:** Equivalent to

```
typedef lambda<op>::type f;
typedef lambda<in::operation>::type in_op;

typedef reverse_fold<
      s
    , in::state
    , bind< in_op, _1, bind<f, _2> >
    >::type r;
```

```
typedef transform<s1,s2,op,in>::type r;
```

**Return type:** A type.

**Postcondition:** Equivalent to

```
typedef lambda<op2>::type f;
typedef lambda<in::operation>::type in_op;

typedef reverse_fold<
      pair_view<s1,s2>
    , in::state
    , bind<
          in_op
        , _1
        , bind<f, bind<first<>,_2>, bind<second<>,_2> >
        >
    >::type r;
```

### Complexity

Linear. Exactly `size<s>::value` / `size<s1>::value` applications of `op` / `op2` and `in::operation`.

### Example

```
typedef vector<char,short,int,long,float,double> types;
typedef vector<double*,float*,long*,int*,short*,char*> pointers;
typedef reverse_transform< types,boost::add_pointer<_1> >::type result;

BOOST_MPL_ASSERT(( equal<result,pointers> ));
```

### See also

Transformation Algorithms, Reversible Algorithm, `transform`, `reverse_copy`, `replace_if`

## 3.5.16 reverse_replace

### Synopsis

```
template<
      typename Sequence
    , typename OldType
    , typename NewType
    , typename In = unspecified
    >
struct reverse_replace
{
    typedef unspecified type;
};
```

### Description

Returns a reversed copy of the original sequence where every type identical to `OldType` has been replaced with `NewType`.

[*Note:* This wording applies to a no-inserter version(s) of the algorithm. See the *Expression semantics* subsection for a precise specification of the algorithm's details in all cases — *end note*]

### Header

```
#include <boost/mpl/replace.hpp>
```

### Model of

Reversible Algorithm

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Forward Sequence | A original sequence. |

| Parameter | Requirement | Description |
|---|---|---|
| `OldType` | Any type | A type to be replaced. |
| `NewType` | Any type | A type to replace with. |
| `In` | Inserter | An inserter. |

#### Expression semantics

The semantics of an expression are defined only where they differ from, or are not defined in Reversible Algorithm.

For any Forward Sequence `s`, an Inserter `in`, and arbitrary types `x` and `y`:

```
typedef reverse_replace<s,x,y,in>::type r;
```

**Return type:** A type.

**Semantics:** Equivalent to

```
typedef reverse_replace_if< s,y,is_same<_,x>,in >::type r;
```

#### Complexity

Linear. Performs exactly `size<s>::value` comparisons for identity / insertions.

#### Example

```
typedef vector<int,float,char,float,float,double> types;
typedef vector<double,double,double,char,double,int> expected;
typedef reverse_replace< types,float,double >::type result;

BOOST_MPL_ASSERT(( equal< result,expected > ));
```

#### See also

Transformation Algorithms, Reversible Algorithm, `replace`, `reverse_replace_if`, `remove`, `reverse_transform`

### 3.5.17 reverse_replace_if

#### Synopsis

```
template<
      typename Sequence
    , typename Pred
    , typename In = unspecified
    >
struct reverse_replace_if
{
    typedef unspecified type;
};
```

### Description

Returns a reversed copy of the original sequence where every type that satisfies the predicate `Pred` has been replaced with `NewType`.

[*Note:* This wording applies to a no-inserter version(s) of the algorithm. See the *Expression semantics* subsection for a precise specification of the algorithm's details in all cases — *end note*]

### Header

```
#include <boost/mpl/replace_if.hpp>
```

### Model of

Reversible Algorithm

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Forward Sequence | An original sequence. |
| `Pred` | Unary Lambda Expression | A replacement condition. |
| `NewType` | Any type | A type to replace with. |
| `In` | Inserter | An inserter. |

### Expression semantics

The semantics of an expression are defined only where they differ from, or are not defined in Reversible Algorithm.

For any Forward Sequence

```
    >::type result;

BOOST_MPL_ASSERT(( equal< result,expected, equal_to<_,_> > ));
```

**See also**

Transformation Algorithms, Reversible Algorithm, `replace_if`, `reverse_replace`, `remove_if`, `transform`

### 3.5.18 reverse_remove

**Synopsis**

```
template<
      typename Sequence
    , typename T
    , typename In = unspecified
    >
struct reverse_remove
{
    typedef unspecified type;
};
```

**Description**

Returns a new sequence that contains all elements from [`begin<Sequence>::type`, `end<Sequence>::type`) range in reverse order except those that are identical to `T`.

[*Note:* This wording applies to a no-inserter version(s) of the algorithm. See the *Expression semantics* subsection for a precise specification of the algorithm's details in all cases — *end note*]

**Header**

```
#include <boost/mpl/remove.hpp>
```

**Model of**

Reversible Algorithm

**Parameters**

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Forward Sequence | An original sequence. |
| `T` | Any type | A type to be removed. |
| `In` | Inserter | An inserter. |

**Expression semantics**

The semantics of an expression are defined only where they differ from, or are not defined in Reversible Algorithm.

For any Forward Sequence `s`, an Inserter `in`, and arbitrary type `x`:

```
typedef reverse_remove<s,x,in>::type r;
```

**Return type:** A type.

**Semantics:** Equivalent to

```
typedef reverse_remove_if< s,is_same<_,x>,in >::type r;
```

#### Complexity

Linear. Performs exactly `size<s>::value` comparisons for equality, and at most `size<s>::value` insertions.

#### Example

```
typedef vector<int,float,char,float,float,double>::type types;
typedef reverse_remove< types,float >::type result;

BOOST_MPL_ASSERT(( equal< result, vector<double,char,int> > ));
```

#### See also

Transformation Algorithms, Reversible Algorithm, `remove`, `reverse_remove_if`, `reverse_copy`, `transform`, `replace`

### 3.5.19 reverse_remove_if

#### Synopsis

```
template<
      typename Sequence
    , typename Pred
    , typename In = unspecified
    >
struct reverse_remove_if
{
    typedef unspecified type;
};
```

#### Description

Returns a new sequence that contains all the elements from [`begin<Sequence>::type`, `end<Sequence>::type`) range in reverse order except those that satisfy the predicate `Pred`.

[*Note:* This wording applies to a no-inserter version(s) of the algorithm. See the *Expression semantics* subsection for a precise specification of the algorithm's details in all cases — *end note*]

#### Header

```
#include <boost/mpl/remove_if.hpp>
```

#### Model of

Reversible Algorithm

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Forward Sequence | An original sequence. |
| `Pred` | Unary Lambda Expression | A removal condition. |
| `In` | Inserter | An inserter. |

### Expression semantics

The semantics of an expression are defined only where they differ from, or are not defined in Reversible Algorithm.

For any Forward Sequence `s`, and an Inserter `in`, and an unary Lambda Expression `pred`:

```
typedef reverse_remove_if<s,pred,in>::type r;
```

**Return type:** A type.

**Semantics:** Equivalent to

```
typedef lambda<pred>::type p;
typedef lambda<in::operation>::type op;

typedef reverse_fold<
      s
    , in::state
    , eval_if<
          apply_wrap1<p,_2>
        , identity<_1>
        , apply_wrap2<op,_1,_2>
        >
    >::type r;
```

### Complexity

Linear. Performs exactly `size<s>::value` applications of `pred`, and at most `size<s>::value` insertions.

### Example

```
typedef vector_c<int,1,4,5,2,7,5,3,5>::type numbers;
typedef reverse_remove_if< numbers, greater<_,int_<4> > >::type result;

BOOST_MPL_ASSERT(( equal< result,vector_c<int,3,2,4,1>,equal_to<_,_> > ));
```

### See also

Transformation Algorithms, Reversible Algorithm, `remove_if`, `reverse_remove`, `reverse_copy_if`, `replace_if`

## 3.5.20 reverse_unique

### Synopsis

```
template<
      typename Seq
```

```
    , typename Pred
    , typename In = unspecified
    >
struct reverse_unique
{
    typedef unspecified type;
};
```

### Description

Returns a sequence of the initial elements of every subrange of the reversed original sequence `Seq` whose elements are all the same.

[*Note:* This wording applies to a no-inserter version(s) of the algorithm. See the *Expression semantics* subsection for a precise specification of the algorithm's details in all cases — *end note*]

### Header

```
#include <boost/mpl/unique.hpp>
```

### Model of

Reversible Algorithm

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Sequence` | Forward Sequence | An original sequence. |
| `Pred` | Binary Lambda Expression | An equivalence relation. |
| `In` | Inserter | An inserter. |

### Expression semantics

The semantics of an expression are defined only where they differ from, or are not defined in Reversible Algorithm.

For any Forward Sequence `s`, a binary Lambda Expression `pred`, and an Inserter `in`:

```
typedef reverse_unique<s,pred,in>::type r;
```

**Return type:** A type.

**Semantics:** If `size<s>::value <= 1`, then equivalent to

```
typedef reverse_copy<s,in>::type r;
```

otherwise equivalent to

```
typedef lambda<pred>::type p;
typedef lambda<in::operation>::type in_op;
typedef apply_wrap2<
      in_op
    , in::state
    , front<types>::type
    >::type in_state;
```

```
typedef reverse_fold<
      s
    , pair< in_state, front<s>::type >
    , eval_if<
          apply_wrap2<p, second<_1>, _2>
        , identity< first<_1> >
        , apply_wrap2<in_op, first<_1>, _2>
        >
    >::type::first r;
```

### Complexity

Linear. Performs exactly `size<s>::value - 1` applications of `pred`, and at most `size<s>::value` insertions.

### Example

```
typedef vector<int,float,float,char,int,int,int,double> types;
typedef vector<double,int,char,float,int> expected;
typedef reverse_unique< types, is_same<_1,_2> >::type result;

BOOST_MPL_ASSERT(( equal< result,expected > ));
```

### See also

Transformation Algorithms, Reversible Algorithm, `unique`, `reverse_remove`, `reverse_copy_if`, `replace_if`

## 3.5.21 reverse_partition

### Synopsis

```
template<
      typename Seq
    , typename Pred
    , typename In1 = unspecified
    , typename In2 = unspecified
    >
struct reverse_partition
{
    typedef unspecified type;
};
```

### Description

Returns a pair of sequences together containing all elements in the range [`begin<Seq>::type`, `end<Seq>::type`) split into two groups based on the predicate `Pred`. `reverse_partition` is a synonym for `reverse_stable_partition`.

[*Note:* This wording applies to a no-inserter version(s) of the algorithm. See the *Expression semantics* subsection for a precise specification of the algorithm's details in all cases — *end note*]

### Header

```
#include <boost/mpl/partition.hpp>
```

### Model of

Reversible Algorithm

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Seq` | Forward Sequence | An original sequence. |
| `Pred` | Unary Lambda Expression | A partitioning predicate. |
| `In1`, `In2` | Inserter | Output inserters. |

### Expression semantics

The semantics of an expression are defined only where they differ from, or are not defined in Reversible Algorithm.

For any Forward Sequence `s`, an unary Lambda Expression `pred`, and Inserters `in1` and `in2`:

```
typedef reverse_partition<s,pred,in1,in2>::type r;
```

**Return type:** A `pair`.

**Semantics:** Equivalent to

```
typedef reverse_stable_partition<s,pred,in1,in2>::type r;
```

### Complexity

Linear. Exactly `size<s>::value` applications of `pred`, and `size<s>::value` of summarized `in1::operation` / `in2::operation` applications.

### Example

```
template< typename N > struct is_odd : bool_<(N::value % 2)> {};

typedef partition<
      range_c<int,0,10>
    , is_odd<_1>
    , back_inserter< vector<> >
    , back_inserter< vector<> >
    >::type r;

BOOST_MPL_ASSERT(( equal< r::first, vector_c<int,9,7,5,3,1> > ));
BOOST_MPL_ASSERT(( equal< r::second, vector_c<int,8,6,4,2,0> > ));
```

### See also

Transformation Algorithms, Reversible Algorithm, `partition`, `reverse_stable_partition`, `sort`

### 3.5.22 reverse_stable_partition

#### Synopsis

```
template<
      typename Seq
    , typename Pred
    , typename In1 = unspecified
    , typename In2 = unspecified
    >
struct reverse_stable_partition
{
    typedef unspecified type;
};
```

#### Description

Returns a pair of sequences together containing all elements in the range [`begin<Seq>::type`, `end<Seq>::type`) split into two groups based on the predicate `Pred`. `reverse_stable_partition` is guaranteed to preserve the reversed relative order of the elements in the resulting sequences.

[*Note:* This wording applies to a no-inserter version(s) of the algorithm. See the *Expression semantics* subsection for a precise specification of the algorithm's details in all cases — *end note*]

#### Header

```
#include <boost/mpl/stable_partition.hpp>
```

#### Model of

Reversible Algorithm

#### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Seq` | Forward Sequence | An original sequence. |
| `Pred` | Unary Lambda Expression | A partitioning predicate. |
| `In1`, `In2` | Inserter | Output inserters. |

#### Expression semantics

The semantics of an expression are defined only where they differ from, or are not defined in Reversible Algorithm.

For any Forward Sequence `s`, an unary Lambda Expression `pred`, and Inserters `in1` and `in2`:

```
typedef reverse_stable_partition<s,pred,in1,in2>::type r;
```

**Return type:** A `pair`.

**Semantics:** Equivalent to

```
typedef lambda<pred>::type p;
typedef lambda<in1::operation>::type in1_op;
typedef lambda<in2::operation>::type in2_op;
```

```
typedef reverse_fold<
      s
    , pair< in1::state, in2::state >
    , if_<
         apply_wrap1<p,_2>
       , pair< apply_wrap2<in1_op,first<_1>,_2>, second<_1> >
       , pair< first<_1>, apply_wrap2<in2_op,second<_1>,_2> >
       >
    >::type r;
```

### Complexity

Linear. Exactly `size<s>::value` applications of `pred`, and `size<s>::value` of summarized `in1::operation` / `in2::operation` applications.

### Example

```
template< typename N > struct is_odd : bool_<(N::value % 2)> {};

typedef reverse_stable_partition<
      range_c<int,0,10>
    , is_odd<_1>
    , back_inserter< vector<> >
    , back_inserter< vector<> >
    >::type r;

BOOST_MPL_ASSERT(( equal< r::first, vector_c<int,9,7,5,3,1> > ));
BOOST_MPL_ASSERT(( equal< r::second, vector_c<int,8,6,4,2,0> > ));
```

### See also

Transformation Algorithms, Reversible Algorithm, `stable_partition`, `reverse_partition`, `sort`, `transform`

# Chapter 4 Metafunctions

The MPL includes a number of predefined metafunctions that can be roughly classified in two categories: *general purpose metafunctions*, dealing with conditional type selection and higher-order metafunction invocation, composition, and argument binding, and *numeric metafunctions*, incapsulating built-in and user-defined arithmetic, comparison, logical, and bitwise operations.

Given that it is possible to perform integer numeric computations at compile time using the conventional operators notation, the need for the second category might be not obvious, but it in fact plays a cental role in making programming with MPL seemingly effortless. In particular, there are at least two contexts where built-in language facilities fall short[3)]:

1) Passing a computation to an algorithm.

2) Performing a computation on non-integer data.

The second use case deserves special attention. In contrast to the built-in, strictly integer compile-time arithmetics, the MPL numeric metafunctions are *polymorphic*, with support for *mixed-type arithmetics*. This means that they can operate on a variety of numeric types — for instance, rational, fixed-point or complex numbers, — and that, in general, you are allowed to freely intermix these types within a single expression. See Numeric Metafunction concept for more details on the MPL numeric infrastructure.

To reduce a negative syntactical impact of the metafunctions notation over the infix operator notation, all numeric metafunctions allow to pass up to N arguments, where N is defined by the value of `BOOST_MPL_LIMIT_METAFUNCTION_ARITY` configuration macro.

## 4.1 Concepts

### 4.1.1 Metafunction

**Description**

A *metafunction* is a class or a class template that represents a function invocable at compile-time. An non-nullary metafunction is invoked by instantiating the class template with particular template parameters (metafunction arguments); the result of the metafunction application is accessible through the instantiation's nested `type` typedef. All metafunction's arguments must be types (i.e. only *type template parameters* are allowed). A metafunction can have a variable number of parameters. A *nullary metafunction* is represented as a (template) class with a nested `type` typename member.

**Expression requirements**

In the following table and subsequent specifications, `f` is a Metafunction.

[3)] All other considerations aside, as of the time of this writing (early 2004), using built-in operators on integral constants still often present a portability problem — many compilers cannot handle particular forms of expressions, forcing us to use conditional compilation. Because MPL numeric metafunctions work on types and encapsulate these kind of workarounds internally, they elude these problems, so if you aim for portability, it is generally adviced to use them in the place of the conventional operators, even at the price of slightly decreased readability.

| Expression | Type | Complexity |
|---|---|---|

| Expression | Type | Complexity |
|---|---|---|
| `f::type` | Any type | Unspecified. |
| `f<>::type` | Any type | Unspecified. |
| `f<a1,..,an>::type` | Any type | Unspecified. |

**Expression semantics**

`typedef f::type x;`

**Precondition:** `f` is a nullary Metafunction; `f::type` is a *type-name*.

**Semantics:** `x` is the result of the metafunction invocation.

`typedef f<>::type x;`

**Precondition:** `f` is a nullary Metafunction; `f<>::type` is a *type-name*.

**Semantics:** `x` is the result of the metafunction invocation.

`typedef f<a1,... an>::type x;`

**Precondition:** `f` is an *n*-ary Metafunction; `a1`,... `an` are types; `f<a1,...an>::type` is a *type-name*.

**Semantics:** `x` is the result of the metafunction invocation with the actual arguments `a1`,... `an`.

**Models**

— `identity`

— `plus`

— `begin`

— `insert`

— `fold`

**See also**

Metafunctions, Metafunction Class, Lambda Expression, invocation, `apply`, `lambda`, `bind`

### 4.1.2 Metafunction Class

**Summary**

A *metafunction class* is a certain form of metafunction representation that enables higher-order metaprogramming. More precisely, it's a class with a publicly-accessible nested Metafunction called `apply`. Correspondingly, a metafunction class invocation is defined as invocation of its nested `apply` metafunction.

**Expression requirements**

In the following table and subsequent specifications, `f` is a Metafunction Class.

| Expression | Type | Complexity |
|---|---|---|
| `f::apply::type` | Any type | Unspecified. |
| `f::apply<>::type` | Any type | Unspecified. |
| `f::apply<a1,...an>::type` | Any type | Unspecified. |

**Expression semantics**

`typedef f::apply::type x;`

**Precondition:** `f` is a nullary Metafunction Class; `f::apply::type` is a *type-name*.

**Semantics:** `x` is the result of the metafunction class invocation.

`typedef f::apply<>::type x;`

**Precondition:** `f` is a nullary Metafunction Class; `f::apply<>::type` is a *type-name*.

**Semantics:** `x` is the result of the metafunction class invocation.

`typedef f::apply<a1,...an>::type x;`

**Precondition:** `f` is an *n*-ary metafunction class; `apply` is a Metafunction.

**Semantics:** `x` is the result of the metafunction class invocation with the actual arguments `a1`,... `an`.

**Models**

— `always`

— `arg`

— `quote`

— `numeric_cast`

— `unpack_args`

**See also**

Metafunctions, Metafunction, Lambda Expression, invocation, `apply_wrap`, `bind`, `quote`

### 4.1.3 Lambda Expression

**Description**

A Lambda Expression is a compile-time invocable entity in either of the following two forms:

— Metafunction Class

— Placeholder Expression

Most of the MPL components accept either of those, and the concept gives us a consice way to describe these requirements.

**Expression requirements**

See corresponding Metafunction Class and Placeholder Expression specifications.

**Models**

— always

— unpack_args

— plus<_, int_<2> >

— if_< less<_1, int_<7> >, plus<_1,_2>, _1 >

**See also**

Metafunctions, Placeholders, apply, lambda

### 4.1.4 Placeholder Expression

**Description**

A Placeholder Expression is a type that is either a placeholder or a class template specialization with at least one argument that itself is a Placeholder Expression.

**Expression requirements**

If X is a class template, and a1,... an are arbitrary types, then X<a1,...,an> is a Placeholder Expression if and only if all of the following conditions hold:

— At least one of the template arguments a1,... an is a placeholder or a Placeholder Expression.

— All of X's template parameters, including the default ones, are types.

— The number of X's template parameters, including the default ones, is less or equal to the value of BOOST_MPL_-LIMIT_METAFUNCTION_ARITY configuration macro.

**Models**

— _1

— plus<_, int_<2> >

— if_< less<_1, int_<7> >, plus<_1,_2>, _1 >

**See also**

Lambda Expression, Placeholders, Metafunctions, apply, lambda

### 4.1.5 Tag Dispatched Metafunction

**Summary**

A Tag Dispatched Metafunction is a Metafunction that employs a *tag dispatching* technique in its implementation to build an infrastructure for easy overriding/extenstion of the metafunction's behavior.

**Notation**

| Symbol | Legend |
|---|---|
| *name* | A placeholder token for the specific metafunction's name. |
| *tag-metafunction* | A placeholder token for the tag metafunction's name. |
| *tag* | A placeholder token for one of possible tag types returned by the tag metafunction. |

**Synopsis**

```
template< typename Tag > struct name_impl;

template<
      typename X
    [, ...]
    >
struct name
    : name_impl< typename tag-metafunction<X>::type >
        ::template apply<X [, ...]>
{
};

template< typename Tag > struct name_impl
{
    template< typename X [, ...] > struct apply
    {
        // default implementation
    };
};

template<> struct name_impl<tag>
{
    template< typename X [, ...] > struct apply
    {
        // tag-specific implementation
    };
};
```

**Description**

The usual mechanism for overriding a metafunction's behavior is class template specialization — given a library-defined metafunction `f`, it's possible to write a specialization of `f` for a specific type `user_type` that would have the required semantics[4)].

While this mechanism is always available, it's not always the most convenient one, especially if it is desirable to specialize a metafunction's behavior for a *family* of related types. A typical example of it is numbered forms of sequence classes in MPL itself (`list0`, ..., `list50`, et al.), and sequence classes in general.

A Tag Dispatched Metafunction is a concept name for an instance of the metafunction implementation infrastructure being employed by the library to make it easier for users and implementors to override the behavior of library's metafunctions operating on families of specific types.

The infrastructure is built on a variation of the technique commonly known as *tag dispatching* (hence the concept name), and involves three entities: a metafunction itself, an associated tag-producing tag metafunction, and the metafunction's implementation, in the form of a Metafunction Class template parametrized by a `Tag` type parameter. The metafunction

redirects to its implementation class template by invoking its specialization on a tag type produced by the tag metafunction with the original metafunction's parameters.

### Example

```
#include <boost/mpl/size.hpp>

namespace user {

struct bitset_tag;

struct bitset0
{
    typedef bitset_tag tag;
    // ...
};

template< typename B0 > struct bitset1
{
    typedef bitset_tag tag;
    // ...
};

template< typename B0, ..., typename Bn > struct bitsetn
{
    typedef bitset_tag tag;
    // ...
};

} // namespace user

namespace boost { namespace mpl {
template<> struct size_impl<user::bitset_tag>
{
    template< typename Bitset > struct apply
    {
        typedef typename Bitset::size type;
    };
};
}}
```

### Models

— sequence_tag

### See also

Metafunction, Metafunction Class, Numeric Metafunction

[4] Usually such user-defined specialization is still required to preserve the `f`'s original invariants and complexity requirements.

### 4.1.6 Numeric Metafunction

#### Description

A Numeric Metafunction is a Tag Dispatched Metafunction that provides a built-in infrastructure for easy implementation of mixed-type operations.

#### Expression requirements

In the following table and subsequent specifications, `op` is a placeholder token for the actual Numeric Metafunction's name, and `x`, `y` and $x_1$,$x_2$,... $x_n$ are arbitrary numeric types.

| Expression | Type | Complexity |
|---|---|---|
| `op_tag<x>::type` | Integral Constant | Amortized constant time. |
| `op_impl<`<br>`  op_tag<x>::type`<br>`, op_tag<y>::type`<br>`>::apply<x,y>::type` | Any type | Unspecified. |
| `op<`$x_1$,$x_2$,... $x_n$`>::type` | Any type | Unspecified. |

#### Expression semantics

```
typedef op_tag<x>::type tag;
```

**Semantics:** `tag` is a tag type for `x` for `op`. `tag::value` is `x`'s *conversion rank*.

```
typedef op_impl<
      op_tag<x>::type
    , op_tag<y>::type
    >::apply<x,y>::type r;
```

**Semantics:** `r` is the result of `op` application on arguments `x` and `y`.

typedef op<$x_1$,$x_2$,... $x_n$>::type r;

**Semantics:** `r` is the result of `op` application on arguments $x_1$,$x_2$,... $x_n$.

#### Example

```
struct complex_tag : int_<10> {};

template< typename Re, typename Im > struct complex
{
    typedef complex_tag tag;
    typedef complex type;
    typedef Re real;
    typedef Im imag;
};

template< typename C > struct real : C::real {};
template< typename C > struct imag : C::imag {};

namespace boost { namespace mpl {
```

```
template<>
struct plus_impl< complex_tag,complex_tag >
{
    template< typename N1, typename N2 > struct apply
        : complex<
              plus< typename N1::real, typename N2::real >
            , plus< typename N1::imag, typename N2::imag >
            >
    {
    };
};

}}

typedef complex< int_<5>, int_<-1> > c1;
typedef complex< int_<-5>, int_<1> > c2;

typedef plus<c1,c2> r1;
BOOST_MPL_ASSERT_RELATION( real<r1>::value, ==, 0 );
BOOST_MPL_ASSERT_RELATION( imag<r1>::value, ==, 0 );

typedef plus<c1,c1> r2;
BOOST_MPL_ASSERT_RELATION( real<r2>::value, ==, 10 );
BOOST_MPL_ASSERT_RELATION( imag<r2>::value, ==, -2 );

typedef plus<c2,c2> r3;
BOOST_MPL_ASSERT_RELATION( real<r3>::value, ==, -10 );
BOOST_MPL_ASSERT_RELATION( imag<r3>::value, ==, 2 );
```

**Models**

— plus

— minus

— times

— divides

**See also**

Tag Dispatched Metafunction, Metafunctions, numeric_cast

### 4.1.7 Trivial Metafunction

**Description**

A Trivial Metafunction accepts a single argument of a class type x and returns the x's nested type member x::name, where name is a placeholder token for the actual member's name accessed by a specific metafunction's instance. By convention, all trivial metafunctions in MPL are named after the members they provide assess to. For instance, a Trivial Metafunction named first reaches for the x's nested member ::first.

### Expression requirements

In the following table and subsequent specifications, `name` is placeholder token for the names of the Trivial Metafunction itself and the accessed member, and `x` is a class type such that `x::name` is a valid *type-name*.

| Expression | Type | Complexity |
|---|---|---|
| `name<x>::type` | Any type | Constant time. |

### Expression semantics

```
typedef name<x>::type r;
```

**Precondition:** `x::name` is a valid *type-name*.

**Semantics:** `is_same<r,x::name>::value == true`.

### Models

- `first`
- `second`
- `base`

### See also

Metafunctions, Trivial Metafunctions, `identity`

## 4.2 Type Selection

### 4.2.1 if_

### Synopsis

```
template<
      typename C
    , typename T1
    , typename T2
    >
struct if_
{
    typedef unspecified type;
};
```

### Description

Returns one of its two arguments, `T1` or `T2`, depending on the value `C`.

### Header

```
#include <boost/mpl/if.hpp>
```

**Parameters**

| Parameter | Requirement | Description |
|---|---|---|
| C | Integral Constant | A selection condition. |
| T1, T2 | Any type | Types to select from. |

#### Expression semantics

For any Integral Constant c and arbitrary types t1, t2:

```
typedef if_<c,t1,t2>::type t;
```

**Return type:** Any type.

**Semantics:** If c::value == true, t is identical to t1; otherwise t is identical to t2.

#### Example

```
typedef if_<true_,char,long>::type t1;
typedef if_<false_,char,long>::type t2;

BOOST_MPL_ASSERT(( is_same<t1, char> ));
BOOST_MPL_ASSERT(( is_same<t2, long> ));
```

#### See also

Metafunctions, Integral Constant, if_c, eval_if

### 4.2.2 if_c

#### Synopsis

```
template<
      bool c
    , typename T1
    , typename T2
    >
struct if_c
{
    typedef unspecified type;
};
```

#### Description

Returns one of its two arguments, T1 or T2, depending on the value of integral constant c. if_c<c,t1,t2>::type is a shorcut notation for if_< bool_<c>,t1,t2 >::type.

#### Header

```
#include <boost/mpl/if.hpp>
```

#### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| c | An integral constant | A selection condition. |
| T1, T2 | Any type | Types to select from. |

### Expression semantics

For any integral constant c and arbitrary types t1, t2:

```
typedef if_c<c,t1,t2>::type t;
```

**Return type:** Any type.

**Semantics:** Equivalent to `typedef if_< bool_<c>,t1,t2 >::type t;`

### Example

```
typedef if_c<true,char,long>::type t1;
typedef if_c<false,char,long>::type t2;

BOOST_MPL_ASSERT(( is_same<t1, char> ));
BOOST_MPL_ASSERT(( is_same<t2, long> ));
```

### See also

Metafunctions, Integral Constant, `if_`, `eval_if`, `bool_`

## 4.2.3 eval_if

### Synopsis

```
template<
      typename C
    , typename F1
    , typename F2
    >
struct eval_if
{
    typedef unspecified type;
};
```

### Description

Evaluates one of its two nullary-metafunction arguments, F1 or F2, depending on the value C.

### Header

```
#include <boost/mpl/eval_if.hpp>
```

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| C | Integral Constant | An evaluation condition. |
| F1, F2 | Nullary Metafunction | Metafunctions to select for evaluation from. |

#### Expression semantics

For any Integral Constant `c` and nullary Metafunctions `f1`, `f2`:

```
typedef eval_if<c,f1,f2>::type t;
```

**Return type:** Any type.

**Semantics:** If `c::value == true`, `t` is identical to `f1::type`; otherwise `t` is identical to `f2::type`.

#### Example

```
typedef eval_if< true_, identity<char>, identity<long> >::type t1;
typedef eval_if< false_, identity<char>, identity<long> >::type t2;

BOOST_MPL_ASSERT(( is_same<t1,char> ));
BOOST_MPL_ASSERT(( is_same<t2,long> ));
```

#### See also

Metafunctions, Integral Constant, `eval_if_c`, `if_`

### 4.2.4 eval_if_c

#### Synopsis

```
template<
      bool c
    , typename F1
    , typename F2
    >
struct eval_if_c
{
    typedef unspecified type;
};
```

#### Description

Evaluates one of its two nullary-metafunction arguments, `F1` or `F2`, depending on the value of integral constant `c`. `eval_if_c<c,f1,f2>::type` is a shorcut notation for `eval_if< bool_<c>,f1,f2 >::type`.

#### Header

```
#include <boost/mpl/eval_if.hpp>
```

#### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| c | An integral constant | An evaluation condition. |
| F1, F2 | Nullary Metafunction | Metafunctions to select for evaluation from. |

**Expression semantics**

For any integral constant c and nullary Metafunctions f1, f2:

```
typedef eval_if_c<c,f1,f2>::type t;
```

**Return type:** Any type.

**Semantics:** Equivalent to `typedef eval_if< bool_<c>,f1,f2 >::type t;`

**Example**

```
typedef eval_if_c< true, identity<char>, identity<long> >::type t1;
typedef eval_if_c< false, identity<char>, identity<long> >::type t2;

BOOST_MPL_ASSERT(( is_same<t1,char> ));
BOOST_MPL_ASSERT(( is_same<t2,long> ));
```

**See also**

Metafunctions, Integral Constant, `eval_if`, `if_`, `bool_`

### 4.3 Invocation

#### 4.3.1 apply

**Synopsis**

```
template<
      typename F
    >
struct apply0
{
    typedef unspecified type;
};

template<
      typename F, typename A1
    >
struct apply1
{
    typedef unspecified type;
};

...

template<
      typename F, typename A1,... typename An
    >
```

```
struct applyn
{
    typedef unspecified type;
};

template<
      typename F
    , typename A1 = unspecified
    ...
    , typename An = unspecified
    >
struct apply
{
    typedef unspecified type;
};
```

### Description

Invokes a Metafunction Class or a Lambda Expression `F` with arguments `A1`,... `An`.

### Header

```
#include <boost/mpl/apply.hpp>
```

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `F` | Lambda Expression | An expression to invoke. |
| `A1`,... `An` | Any type | Invocation arguments. |

### Expression semantics

For any Lambda Expression `f` and arbitrary types `a1`,... `an`:

```
typedef applyn<f,a1,...an>::type t;
typedef apply<f,a1,...an>::type t;
```

**Return type:** Any type.

**Semantics:** Equivalent to `typedef apply_wrapn< lambda<f>::type,a1,... an>::type t;`.

### Example

```
template< typename N1, typename N2 > struct int_plus
    : int_<( N1::value + N2::value )>
{
};

typedef apply< int_plus<_1,_2>, int_<2>, int_<3> >::type r1;
typedef apply< quote2<int_plus>, int_<2>, int_<3> >::type r2;

BOOST_MPL_ASSERT_RELATION( r1::value, ==, 5 );
```

```
BOOST_MPL_ASSERT_RELATION( r2::value, ==, 5 );
```

**See also**

Metafunctions, apply_wrap, lambda, quote, bind

### 4.3.2 apply_wrap

**Synopsis**

```
template<
      typename F
    >
struct apply_wrap0
{
    typedef unspecified type;
};

template<
      typename F, typename A1
    >
struct apply_wrap1
{
    typedef unspecified type;
};

...

template<
      typename F, typename A1,... typename An
    >
struct apply_wrapn
{
    typedef unspecified type;
};
```

**Description**

Invokes a Metafunction Class F with arguments A1,... An.

In essence, apply_wrap forms are nothing more than syntactic wrappers around F::apply<A1,... An>::type / F::apply::type expressions (hence the name). They provide a more concise notation and higher portability than their underlaying constructs at the cost of an extra template instantiation.

**Header**

```
#include <boost/mpl/apply_wrap.hpp>
```

**Parameters**

| Parameter | Requirement | Description |
|---|---|---|
| `F` | Metafunction Class | A metafunction class to invoke. |
| `A1`,... `An` | Any type | Invocation arguments. |

### Expression semantics

For any Metafunction Class `f` and arbitrary types `a1`,... `an`:

```
typedef apply_wrapn<f,a1,...an>::type t;
```

**Return type:** Any type.

**Semantics:** If `n > 0`, equivalent to `typedef f::apply<a1,... an>::type t;`, otherwise equivalent to either `typedef f::apply::type t;` or `typedef f::apply<>::type t;` depending on whether `f::apply` is a class or a class template.

### Example

```
struct f0
{
    template< typename T = int > struct apply
    {
        typedef char type;
    };
};

struct g0
{
    struct apply { typedef char type; };
};

struct f2
{
    template< typename T1, typename T2 > struct apply
    {
        typedef T2 type;
    };
};


typedef apply_wrap0< f0 >::type r1;
typedef apply_wrap0< g0 >::type r2;
typedef apply_wrap2< f2,int,char >::type r3;

BOOST_MPL_ASSERT(( is_same<r1,char> ));
BOOST_MPL_ASSERT(( is_same<r2,char> ));
BOOST_MPL_ASSERT(( is_same<r3,char> ));
```

### See also

Metafunctions, invocation, `apply`, `lambda`, `quote`, `bind`, `protect`

### 4.3.3 unpack_args

#### Synopsis

```
template<
      typename F
    >
struct unpack_args
{
    // unspecified
    // ...
};
```

#### Description

A higher-order primitive transforming an *n*-ary Lambda Expression `F` into an unary Metafunction Class `g` accepting a single sequence of *n* arguments.

#### Header

```
#include <boost/mpl/unpack_args.hpp>
```

#### Model of

Metafunction Class

#### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| F | Lambda Expression | A lambda expression to adopt. |

#### Expression semantics

For an arbitrary Lambda Expression `f`, and arbitrary types `a1`,... `an`:

```
typedef unpack_args<f> g;
```

**Return type:** Metafunction Class.

**Semantics:** `g` is a unary Metafunction Class such that

```
apply_wrapn< g, vector<a1,...an> >::type
```

is identical to

```
apply<F,a1,...an>::type
```

#### Example

```
BOOST_MPL_ASSERT(( apply<
      unpack_args< is_same<_1,_2> >
    , vector<int,int>
    > ));
```

**See also**

Metafunctions, Lambda Expression, Metafunction Class, `apply`, `apply_wrap`, `bind`

## 4.4 Composition and Argument Binding

### 4.4.1 Placeholders

**Synopsis**

```
namespace placeholders {
typedef unspecified _;
typedef arg<1>      _1;
typedef arg<2>      _2;
...
typedef arg<n>      _n;
}

using placeholders::_;
using placeholders::_1;
using placeholders::_2;
...
using placeholders::_n;
```

**Description**

A placeholder in a form *_n* is simply a synonym for the corresponding `arg<n>` specialization. The unnamed placeholder `_` (underscore) carries special meaning in bind and lambda expressions, and does not have defined semantics outside of these contexts.

Placeholder names can be made available in the user namespace through `using namespace mpl::placeholders;` directive.

**Header**

```
#include <boost/mpl/placeholders.hpp>
```

[*Note:* The include might be omitted when using placeholders to construct a Lambda Expression for passing it to MPL's own algorithm or metafunction: any library component that is documented to accept a lambda expression makes the placeholders implicitly available for the user code — *end note*]

**Parameters**

None.

**Expression semantics**

For any integral constant `n` in the range [1, `BOOST_MPL_LIMIT_METAFUNCTION_ARITY`] and arbitrary types `a1`,... `an`:

```
typedef apply_wrapn<_n,a1,...an>::type x;
```

**Return type:** A type.

**Semantics:** Equivalent to

```
typedef apply_wrapn< arg<n>,a1,...an >::type x;
```

**Example**

```
typedef apply_wrap5< _1,bool,char,short,int,long >::type t1;
typedef apply_wrap5< _3,bool,char,short,int,long >::type t3;

BOOST_MPL_ASSERT(( is_same< t1, bool > ));
BOOST_MPL_ASSERT(( is_same< t3, short > ));
```

**See also**

Composition and Argument Binding, arg, lambda, bind, apply, apply_wrap

### 4.4.2 lambda

**Synopsis**

```
template<
      typename X
    , typename Tag = unspecified
    >
struct lambda
{
    typedef unspecified type;
};
```

**Description**

If X is a Placeholder Expression, transforms X into a corresponding Metafunction Class, otherwise X is returned unchanged.

**Header**

```
#include <boost/mpl/lambda.hpp>
```

**Parameters**

| Parameter | Requirement | Description |
|---|---|---|
| X | Any type | An expression to transform. |
| Tag | Any type | A tag determining transform semantics. |

**Expression semantics**

For arbitrary types x and tag:

```
typedef lambda<x>::type f;
```

**Return type:** Metafunction Class.

**Semantics:** If x is a Placeholder Expression in a general form X<a1,...an>, where X is a class template

and a1,... an are arbitrary types, equivalent to

```
typedef protect< bind<
      quoten<X>
    , lambda<a1>::type,... lambda<an>::type
    > > f;
```

otherwise, f is identical to x.

```
typedef lambda<x,tag>::type f;
```

**Return type:** Metafunction Class.

**Semantics:** If x is a Placeholder Expression in a general form X<a1,...an>, where X is a class template and a1,... an are arbitrary types, equivalent to

```
typedef protect< bind<
      quoten<X,tag>
    , lambda<a1,tag>::type,... lambda<an,tag>::type
    > > f;
```

otherwise, f is identical to x.

### Example

```
template< typename N1, typename N2 > struct int_plus
    : int_<( N1::value + N2::value )>
{
};

typedef lambda< int_plus<_1, int_<42> > >::type f1;
typedef bind< quote2<int_plus>, _1, int_<42> > f2;

typedef f1::apply<42>::type r1;
typedef f2::apply<42>::type r2;

BOOST_MPL_ASSERT_RELATION( r1::value, ==, 84 );
BOOST_MPL_ASSERT_RELATION( r2::value, ==, 84 );
```

### See also

Composition and Argument Binding, invocation, Placeholders, bind, quote, protect, apply

## 4.4.3 bind

### Synopsis

```
template<
      typename F
    >
struct bind0
{
    // unspecified
    // ...
};
```

```
template<
      typename F, typename A1
    >
struct bind1
{
    // unspecified
    // ...
};

...

template<
      typename F, typename A1,... typename An
    >
struct bindn
{
    // unspecified
    // ...
};

template<
      typename F
    , typename A1 = unspecified
    ...
    , typename An = unspecified
    >
struct bind
{
    // unspecified
    // ...
};
```

### Description

`bind` is a higher-order primitive for Metafunction Class composition and argument binding. In essence, it's a compile-time counterpart of the similar run-time functionality provided by Boost.Bind and Boost.Lambda libraries.

### Header

```
#include <boost/mpl/bind.hpp>
```

### Model of

Metafunction Class

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `F` | Metafunction Class | An metafunction class to perform binding on. |
| `A1,... An` | Any type | Arguments to bind. |

### Expression semantics

For any Metafunction Class `f` and arbitrary types `a1`,... `an`:

```
typedef bind<f,a1,...an> g;
typedef bindn<f,a1,...an> g;
```

**Return type:** Metafunction Class

**Semantics:** Equivalent to

```
struct g
{
    template<
          typename U1 = unspecified
        ...
        , typename Un = unspecified
        >
    struct apply
        : apply_wrapn<
              typename h0<f,U1,...Un>::type
            , typename h1<a1,U1,...Un>::type
            ...
            , typename hn<an,U1,...Un>::type
            >
    {
    };
};
```

where h*k* is equivalent to

```
template< typename X, typename U1,... typename Un > struct hk
    : apply_wrapn<X,U1,...Un>
{
};
```

if `f` or a*k* is a bind expression or a placeholder, and

```
template< typename X, typename U1,... typename Un > struct hk
{
    typedef X type;
};
```

otherwise. [*Note:* Every nth appearance of the unnamed placeholder in the `bind<f,a1,...an>` specialization is replaced with the corresponding numbered placeholder _*n* — *end note*]

### Example

```
struct f1
{
    template< typename T1 > struct apply
    {
        typedef T1 type;
    };
};

struct f5
```

```
{
    template< typename T1, typename T2, typename T3, typename T4, typename T5 >
    struct apply
    {
        typedef T5 type;
    };
};

typedef apply_wrap1<
      bind1<f1,_1>
    , int
    >::type r11;

typedef apply_wrap5<
      bind1<f1,_5>
    , void,void,void,void,int
    >::type r12;

BOOST_MPL_ASSERT(( is_same<r11,int> ));
BOOST_MPL_ASSERT(( is_same<r12,int> ));

typedef apply_wrap5<
      bind5<f5,_1,_2,_3,_4,_5>
    , void,void,void,void,int
    >::type r51;

typedef apply_wrap5<
      bind5<f5,_5,_4,_3,_2,_1>
    , int,void,void,void,void
    >::type r52;

BOOST_MPL_ASSERT(( is_same<r51,int> ));
BOOST_MPL_ASSERT(( is_same<r52,int> ));
```

**See also**

Composition and Argument Binding, invocation, Placeholders, `lambda`, `quote`, `protect`, `apply`, `apply_wrap`

### 4.4.4 quote

**Synopsis**

```
template<
      template< typename P1 > class F
    , typename Tag = unspecified
    >
struct quote1
{
    // unspecified
    // ...
};
```

```
...

template<
      template< typename P1,... typename Pn > class F
    , typename Tag = unspecified
    >
struct quoten
{
    // unspecified
    // ...
};
```

### Description

quoten is a higher-order primitive that wraps an *n*-ary Metafunction to create a corresponding Metafunction Class.

### Header

```
#include <boost/mpl/quote.hpp>
```

### Model of

Metafunction Class

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| F | Metafunction | A metafunction to wrap. |
| Tag | Any type | A tag determining wrap semantics. |

### Expression semantics

For any *n*-ary Metafunction f and arbitrary type tag:

```
typedef quoten<f> g;
typedef quoten<f,tag> g;
```

**Return type:** Metafunction Class

**Semantics:** Equivalent to

```
struct g
{
    template< typename A1,... typename An > struct apply
        : f<A1,...An>
    {
    };
};
```

if f<A1,...An> has a nested type member ::type, and to

```
struct g
{
```

```
            template< typename A1,... typename An > struct apply
            {
                typedef f<A1,...An> type;
            };
        };
```

otherwise.

**Example**

```
template< typename T > struct f1
{
    typedef T type;
};

template<
    typename T1, typename T2, typename T3, typename T4, typename T5
    >
struct f5
{
    // no 'type' member!
};

typedef quote1<f1>::apply<int>::type t1;
typedef quote5<f5>::apply<char,short,int,long,float>::type t5;

BOOST_MPL_ASSERT(( is_same< t1, int > ));
BOOST_MPL_ASSERT(( is_same< t5, f5<char,short,int,long,float> > ));
```

**See also**

Composition and Argument Binding, invocation, `bind`, `lambda`, `protect`, `apply`

### 4.4.5 arg

**Synopsis**

```
template< int n > struct arg;

template<> struct arg<1>
{
    template< typename A1,... typename An = unspecified >
    struct apply
    {
        typedef A1 type;
    };
};

...

template<> struct arg<n>
{
```

```
    template< typename A1,... typename An >
    struct apply
    {
        typedef An type;
    };
};
```

### Description

`arg<n>` specialization is a Metafunction Class that return the nth of its arguments.

### Header

```
#include <boost/mpl/arg.hpp>
```

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `n` | An integral constant | A number of argument to return. |

### Expression semantics

For any integral constant `n` in the range [1, `BOOST_MPL_LIMIT_METAFUNCTION_ARITY`] and arbitrary types `a1`,... `an`:

```
typedef apply_wrapn< arg<n>,a1,... an >::type x;
```

**Return type:** A type.

**Semantics:** `x` is identical to `an`.

### Example

```
typedef apply_wrap5< arg<1>,bool,char,short,int,long >::type t1;
typedef apply_wrap5< arg<3>,bool,char,short,int,long >::type t3;

BOOST_MPL_ASSERT(( is_same< t1, bool > ));
BOOST_MPL_ASSERT(( is_same< t3, short > ));
```

### See also

Composition and Argument Binding, Placeholders, `lambda`, `bind`, `apply`, `apply_wrap`

## 4.4.6 protect

### Synopsis

```
template<
      typename F
    >
struct protect
{
```

```
    // unspecified
    // ...
};
```

### Description

`protect` is an identity wrapper for a Metafunction Class that prevents its argument from being recognized as a bind expression.

### Header

```
#include <boost/mpl/protect.hpp>
```

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `F` | Metafunction Class | A metafunction class to wrap. |

### Expression semantics

For any Metafunction Class `f`:

```
typedef protect<f> g;
```

**Return type:** Metafunction Class.

**Semantics:** If `f` is a bind expression, equivalent to

```
struct g
{
    template<
          typename U1 = unspecified,... typename Un = unspecified
        >
    struct apply
        : apply_wrapn<f,U1,...Un>
    {
    };
};
```

otherwise equivalent to `typedef f g;`.

### Example

```
FIXME

struct f
{
    template< typename T1, typename T2 > struct apply
    {
        // ...
    };
};
```

```
typedef bind<_1, protect< bind<f,_1,_2> > >

typedef apply_wrap0< f0 >::type r1;
typedef apply_wrap0< g0 >::type r2;
typedef apply_wrap2< f2,int,char >::type r3;

BOOST_MPL_ASSERT(( is_same<r1,char> ));
BOOST_MPL_ASSERT(( is_same<r2,char> ));
BOOST_MPL_ASSERT(( is_same<r3,char> ));
```

### See also

Composition and Argument Binding, invocation, `bind`, `quote`, `apply_wrap`

## 4.5 Arithmetic Operations

### 4.5.1 plus

### Synopsis

```
template<
      typename T1
    , typename T2
    , typename T3 = unspecified
    ...
    , typename Tn = unspecified
    >
struct plus
{
    typedef unspecified type;
};
```

### Description

Returns the sum of its arguments.

### Header

```
#include <boost/mpl/plus.hpp>
#include <boost/mpl/arithmetic.hpp>
```

### Model of

Numeric Metafunction

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| T1, T2,... Tn | Integral Constant | Operation's arguments. |

[*Note:* The requirements listed in this specification are the ones imposed by the default implementation. See Numeric Metafunction concept for the details on how to provide an implementation for a user-defined numeric type that does not satisfy the Integral Constant requirements. — *end note*]

#### Expression semantics

For any Integral Constants $c_1$,$c_2$,... $c_n$:

```
typedef plus<c1,... cn>::type r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```
typedef integral_c<
      typeof(c1::value + c2::value)
    , ( c1::value + c2::value )
    > c;

typedef plus<c,c3,... cn>::type r;
```

```
typedef plus<c1,... cn> r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```
struct r : plus<c1,... cn>::type {};
```

#### Complexity

Amortized constant time.

#### Example

```
typedef plus< int_<-10>, int_<3>, long_<1> >::type r;
BOOST_MPL_ASSERT_RELATION( r::value, ==, -6 );
BOOST_MPL_ASSERT(( is_same< r::value_type, long > ));
```

#### See also

Arithmetic Operations, Numeric Metafunction, `numeric_cast`, `minus`, `negate`, `times`

### 4.5.2 minus

#### Synopsis

```
template<
      typename T1
    , typename T2
    , typename T3 = unspecified
    ...
    , typename Tn = unspecified
    >
struct minus
```

```
{
    typedef unspecified type;
};
```

### Description

Returns the difference of its arguments.

### Header

```
#include <boost/mpl/minus.hpp>
#include <boost/mpl/arithmetic.hpp>
```

### Model of

Numeric Metafunction

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| T1, T2,... Tn | Integral Constant | Operation's arguments. |

[*Note:* The requirements listed in this specification are the ones imposed by the default implementation. See Numeric Metafunction concept for the details on how to provide an implementation for a user-defined numeric type that does not satisfy the Integral Constant requirements. — *end note*]

### Expression semantics

For any Integral Constants $c_1$,$c_2$,... $c_n$:

```
typedef minus<c1,...cn>::type r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```
typedef integral_c<
      typeof(c1::value - c2::value)
    , ( c1::value - c2::value )
    > c;

typedef minus<c,c3,...cn>::type r;
```

```
typedef minus<c1,...cn> r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```
struct r : minus<c1,...cn>::type {};
```

### Complexity

Amortized constant time.

### Example

```
typedef minus< int_<-10>, int_<3>, long_<1> >::type r;
BOOST_MPL_ASSERT_RELATION( r::value, ==, -14 );
BOOST_MPL_ASSERT(( is_same< r::value_type, long > ));
```

### See also

Arithmetic Operations, Numeric Metafunction, `numeric_cast`, `plus`, `negate`, `times`

## 4.5.3 times

### Synopsis

```
template<
      typename T1
    , typename T2
    , typename T3 = unspecified
    ...
    , typename Tn = unspecified
    >
struct times
{
    typedef unspecified type;
};
```

### Description

Returns the product of its arguments.

### Header

```
#include <boost/mpl/times.hpp>
#include <boost/mpl/arithmetic.hpp>
```

### Model of

Numeric Metafunction

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| T1, T2,... Tn | Integral Constant | Operation's arguments. |

[*Note:* The requirements listed in this specification are the ones imposed by the default implementation. See Numeric Metafunction concept for the details on how to provide an implementation for a user-defined numeric type that does not satisfy the Integral Constant requirements. — *end note*]

### Expression semantics

For any Integral Constants $c_1$,$c_2$,... $c_n$:

```
typedef times<c1,...cn>::type r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```
typedef integral_c<
      typeof(c1::value * c2::value)
    , ( c1::value * c2::value )
    > c;

typedef times<c,c3,...cn>::type r;
```

```
typedef times<c1,...cn> r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```
struct r : times<c1,...cn>::type {};
```

### Complexity

Amortized constant time.

### Example

```
typedef times< int_<-10>, int_<3>, long_<1> >::type r;
BOOST_MPL_ASSERT_RELATION( r::value, ==, -30 );
BOOST_MPL_ASSERT(( is_same< r::value_type, long > ));
```

### See also

Metafunctions, Numeric Metafunction, `numeric_cast`, `divides`, `modulus`, `plus`

## 4.5.4 divides

### Synopsis

```
template<
      typename T1
    , typename T2
    , typename T3 = unspecified
    ...
    , typename Tn = unspecified
    >
struct divides
{
    typedef unspecified type;
};
```

### Description

Returns the quotient of its arguments.

### Header

```
#include <boost/mpl/divides.hpp>
#include <boost/mpl/arithmetic.hpp>
```

### Model of

Numeric Metafunction

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| T1, T2,... Tn | Integral Constant | Operation's arguments. |

[*Note:* The requirements listed in this specification are the ones imposed by the default implementation. See Numeric Metafunction concept for the details on how to provide an implementation for a user-defined numeric type that does not satisfy the Integral Constant requirements. — *end note*]

### Expression semantics

For any Integral Constants $c_1$,$c_2$,... $c_n$:

```
typedef divides<c1,... cn>::type r;
```

**Return type:** Integral Constant.

**Precondition:** `c2::value != 0`, ... `cn::value != 0`.

**Semantics:** Equivalent to

```
typedef integral_c<
      typeof(c1::value / c2::value)
    , ( c1::value / c2::value )
    > c;

typedef divides<c,c3,... cn>::type r;
```

```
typedef divides<c1,... cn> r;
```

**Return type:** Integral Constant.

**Precondition:** `c2::value != 0`, ... `cn::value != 0`.

**Semantics:** Equivalent to

```
struct r : divides<c1,... cn>::type {};
```

### Complexity

Amortized constant time.

**Example**

```
typedef divides< int_<-10>, int_<3>, long_<1> >::type r;
BOOST_MPL_ASSERT_RELATION( r::value, ==, -3 );
BOOST_MPL_ASSERT(( is_same< r::value_type, long > ));
```

**See also**

Arithmetic Operations, Numeric Metafunction, `numeric_cast`, `times`, `modulus`, `plus`

### 4.5.5 modulus

**Synopsis**

```
template<
      typename T1
    , typename T2
    >
struct modulus
{
    typedef unspecified type;
};
```

**Description**

Returns the modulus of its arguments.

**Header**

```
#include <boost/mpl/modulus.hpp>
#include <boost/mpl/arithmetic.hpp>
```

**Model of**

Numeric Metafunction

**Parameters**

| Parameter | Requirement | Description |
|---|---|---|
| T1, T2 | Integral Constant | Operation's arguments. |

[*Note:* The requirements listed in this specification are the ones imposed by the default implementation. See Numeric Metafunction concept for the details on how to provide an implementation for a user-defined numeric type that does not satisfy the Integral Constant requirements. — *end note*]

**Expression semantics**

For any Integral Constants `c1` and `c2`:

```
typedef modulus<c1,c2>::type r;
```

**Return type:** Integral Constant.

**Precondition:** `c2::value != 0`

**Semantics:** Equivalent to

```
typedef integral_c<
      typeof(c1::value % c2::value)
    , ( c1::value % c2::value )
    > r;
```

`typedef modulus<c1,c2> r;`

**Return type:** Integral Constant.

**Precondition:** `c2::value != 0`

**Semantics:** Equivalent to

```
struct r : modulus<c1,c2>::type {};
```

### Complexity

Amortized constant time.

### Example

```
typedef modulus< int_<10>, long_<3> >::type r;
BOOST_MPL_ASSERT_RELATION( r::value, ==, 1 );
BOOST_MPL_ASSERT(( is_same< r::value_type, long > ));
```

### See also

Metafunctions, Numeric Metafunction, `numeric_cast`, `divides`, `times`, `plus`

## 4.5.6 negate

### Synopsis

```
template<
      typename T
    >
struct negate
{
    typedef unspecified type;
};
```

### Description

Returns the negative (additive inverse) of its argument.

### Header

```
#include <boost/mpl/negate.hpp>
#include <boost/mpl/arithmetic.hpp>
```

**Model of**

Numeric Metafunction

**Parameters**

| Parameter | Requirement | Description |
|---|---|---|
| `T` | Integral Constant | Operation's argument. |

[*Note:* The requirements listed in this specification are the ones imposed by the default implementation. See Numeric Metafunction concept for the details on how to provide an implementation for a us137 295 682.263 Tf 17.225.63t2f

```
    , typename T2
    >
struct less
{
    typedef unspecified type;
};
```

### Description

Returns a true-valued Integral Constant if `T1` is less than `T2`.

### Header

```
#include <boost/mpl/less.hpp>
#include <boost/mpl/comparison.hpp>
```

### Model of

Numeric Metafunction

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `T1`, `T2` | Integral Constant | Operation's arguments. |

[*Note:* The requirements listed in this specification are the ones imposed by the default implementation. See Numeric Metafunction concept for the details on how to provide an implementation for a user-defined numeric type that does not satisfy the Integral Constant requirements. — *end note*]

### Expression semantics

For any Integral Constants `c1` and `c2`:

```
typedef less<c1,c2>::type r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```
typedef bool_< (c1::value < c2::value) > r;
```

```
typedef less<c1,c2> r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```
struct r : less<c1,c2>::type {};
```

### Complexity

Amortized constant time.

### Example

```
BOOST_MPL_ASSERT(( less< int_<0>, int_<10> > ));
BOOST_MPL_ASSERT_NOT(( less< long_<10>, int_<0> > ));
BOOST_MPL_ASSERT_NOT(( less< long_<10>, int_<10> > ));
```

### See also

Comparisons, Numeric Metafunction, `numeric_cast`, `less_equal`, `greater`, `equal`

## 4.6.2 less_equal

### Synopsis

```
template<
      typename T1
    , typename T2
    >
struct less_equal
{
    typedef unspecified type;
};
```

### Description

Returns a true-valued Integral Constant if `T1` is less than or equal to `T2`.

### Header

```
#include <boost/mpl/less_equal.hpp>
#include <boost/mpl/comparison.hpp>
```

### Model of

Numeric Metafunction

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `T1`, `T2` | Integral Constant | Operation's arguments. |

[*Note:* The requirements listed in this specification are the ones imposed by the default implementation. See Numeric Metafunction concept for the details on how to provide an implementation for a user-defined numeric type that does not satisfy the Integral Constant requirements. — *end note*]

### Expression semantics

For any Integral Constants `c1` and `c2`:

```
typedef less_equal<c1,c2>::type r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```
typedef bool_< (c1::value <= c2::value) > r;
```

```
typedef less_equal<c1,c2> r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```
struct r : less_equal<c1,c2>::type {};
```

#### Complexity

Amortized constant time.

#### Example

```
BOOST_MPL_ASSERT(( less_equal< int_<0>, int_<10> > ));
BOOST_MPL_ASSERT_NOT(( less_equal< long_<10>, int_<0> > ));
BOOST_MPL_ASSERT(( less_equal< long_<10>, int_<10> > ));
```

#### See also

Comparisons, Numeric Metafunction, `numeric_cast`, `less`, `greater`, `equal`

### 4.6.3 greater

#### Synopsis

```
template<
      typename T1
    , typename T2
    >
struct greater
{
    typedef unspecified type;
};
```

#### Description

Returns a true-valued Integral Constant if `T1` is greater than `T2`.

#### Header

```
#include <boost/mpl/greater.hpp>
#include <boost/mpl/comparison.hpp>
```

#### Model of

Numeric Metafunction

**Parameters**

| Parameter | Requirement | Description |
|---|---|---|
| T1, T2 | Integral Constant | Operation's arguments. |

[*Note:* The requirements listed in this specification are the ones imposed by the default implementation. See Numeric Metafunction concept for the details on how to provide an implementation for a user-defined numeric type that does not satisfy the Integral Constant requirements. — *end note*]

**Expression semantics**

For any Integral Constants c1 and c2:

```
typedef greater<c1,c2>::type r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```
typedef bool_< (c1::value < c2::value) > r;
```

```
typedef greater<c1,c2> r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```
struct r : greater<c1,c2>::type {};
```

**Complexity**

Amortized constant time.

**Example**

```
BOOST_MPL_ASSERT(( greater< int_<10>, int_<0> > ));
BOOST_MPL_ASSERT_NOT(( greater< long_<0>, int_<10> > ));
BOOST_MPL_ASSERT_NOT(( greater< long_<10>, int_<10> > ));
```

**See also**

Comparisons, Numeric Metafunction, numeric_cast, greater_equal, less, equal_to

### 4.6.4 greater_equal

**Synopsis**

```
template<
      typename T1
    , typename T2
    >
struct greater_equal
{
    typedef unspecified type;
};
```

### Description

Returns a true-valued Integral Constant if `T1` is greater than or equal to `T2`.

### Header

```
#include <boost/mpl/greater_equal.hpp>
#include <boost/mpl/comparison.hpp>
```

### Model of

Numeric Metafunction

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `T1`, `T2` | Integral Constant | Operation's arguments. |

[*Note:* The requirements listed in this specification are the ones imposed by the default implementation. See Numeric Metafunction concept for the details on how to provide an implementation for a user-defined numeric type that does not satisfy the Integral Constant requirements. — *end note*]

### Expression semantics

For any Integral Constants `c1` and `c2`:

```
typedef greater_equal<c1,c2>::type r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```
typedef bool_< (c1::value < c2::value) > r;
```

```
typedef greater_equal<c1,c2> r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```
struct r : greater_equal<c1,c2>::type {};
```

### Complexity

Amortized constant time.

### Example

```
BOOST_MPL_ASSERT(( greater_equal< int_<10>, int_<0> > ));
BOOST_MPL_ASSERT_NOT(( greater_equal< long_<0>, int_<10> > ));
BOOST_MPL_ASSERT(( greater_equal< long_<10>, int_<10> > ));
```

**See also**

Comparisons, Numeric Metafunction, `numeric_cast`, `greater`, `less`, `equal_to`

### 4.6.5 equal_to

**Synopsis**

```
template<
      typename T1
    , typename T2
    >
struct equal_to
{
    typedef unspecified type;
};
```

**Description**

Returns a true-valued Integral Constant if `T1` and `T2` are equal.

**Header**

```
#include <boost/mpl/equal_to.hpp>
#include <boost/mpl/comparison.hpp>
```

**Model of**

Numeric Metafunction

**Parameters**

| Parameter | Requirement | Description |
|---|---|---|
| `T1`, `T2` | Integral Constant | Operation's arguments. |

[*Note:* The requirements listed in this specification are the ones imposed by the default implementation. See Numeric Metafunction concept for the details on how to provide an implementation for a user-defined numeric type that does not satisfy the Integral Constant requirements. — *end note*]

**Expression semantics**

For any Integral Constants `c1` and `c2`:

```
typedef equal_to<c1,c2>::type r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```
typedef bool_< (c1::value == c2::value) > r;
```

```
typedef equal_to<c1,c2> r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```
struct r : equal_to<c1,c2>::type {};
```

### Complexity

Amortized constant time.

### Example

```
BOOST_MPL_ASSERT_NOT(( equal_to< int_<0>, int_<10> > ));
BOOST_MPL_ASSERT_NOT(( equal_to< long_<10>, int_<0> > ));
BOOST_MPL_ASSERT(( equal_to< long_<10>, int_<10> > ));
```

### See also

Comparisons, Numeric Metafunction, `numeric_cast`, `not_equal_to`, `less`

## 4.6.6 not_equal_to

### Synopsis

```
template<
      typename T1
    , typename T2
    >
struct not_equal_to
{
    typedef unspecified type;
};
```

### Description

Returns a true-valued Integral Constant if `T1` and `T2` are not equal.

### Header

```
#include <boost/mpl/not_equal_to.hpp>
#include <boost/mpl/comparison.hpp>
```

### Model of

Numeric Metafunction

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `T1`, `T2` | Integral Constant | Operation's arguments. |

[*Note:* The requirements listed in this specification are the ones imposed by the default implementation. See Numeric Metafunction concept for the details on how to provide an implementation for a user-defined numeric type that does not satisfy the Integral Constant requirements. — *end note*]

#### Expression semantics

For any Integral Constants `c1` and `c2`:

```
typedef not_equal_to<c1,c2>::type r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```
typedef bool_< (c1::value != c2::value) > r;
```

```
typedef not_equal_to<c1,c2> r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```
struct r : not_equal_to<c1,c2>::type {};
```

#### Complexity

Amortized constant time.

#### Example

```
BOOST_MPL_ASSERT(( not_equal_to< int_<0>, int_<10> > ));
BOOST_MPL_ASSERT(( not_equal_to< long_<10>, int_<0> > ));
BOOST_MPL_ASSERT_NOT(( not_equal_to< long_<10>, int_<10> > ));
```

#### See also

Comparisons, Numeric Metafunction, `numeric_cast`, `equal_to`, `less`

### 4.7 Logical Operations

#### 4.7.1 and_

#### Synopsis

```
template<
      typename F1
    , typename F2
    ...
    , typename Fn = unspecified
    >
struct and_
{
    typedef unspecified type;
};
```

### Description

Returns the result of short-circuit *logical and* (`&&`) operation on its arguments.

### Header

```
#include <boost/mpl/and.hpp>
#include <boost/mpl/logical.hpp>
```

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `F1`, `F2`,... `Fn` | Nullary Metafunction | Operation's arguments. |

### Expression semantics

For arbitrary nullary Metafunctions `f1`, `f2`,... `fn`:

```
typedef and_<f1,f2,...,fn>::type r;
```

**Return type:** Integral Constant.

**Semantics:** `r` is `false_` if either of `f1::type::value`, `f2::type::value`,... `fn::type::value` expressions evaluates to `false`, and `true_` otherwise; guarantees left-to-right evaluation; the operands subsequent to the first `f`*i* metafunction that evaluates to `false` are not evaluated.

```
typedef and_<f1,f2,...,fn> r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```
struct r : and_<f1,f2,...,fn>::type {};
```

### Example

```
struct unknown;

BOOST_MPL_ASSERT(( and_< true_,true_ > ));
BOOST_MPL_ASSERT_NOT(( and_< false_,true_ > ));
BOOST_MPL_ASSERT_NOT(( and_< true_,false_ > ));
BOOST_MPL_ASSERT_NOT(( and_< false_,false_ > ));
BOOST_MPL_ASSERT_NOT(( and_< false_,unknown > )); // OK
BOOST_MPL_ASSERT_NOT(( and_< false_,unknown,unknown > )); // OK too
```

### See also

Metafunctions, Logical Operations, `or_`, `not_`

## 4.7.2 or_

### Synopsis

```
template<
```

```
      typename F1
    , typename F2
    ...
    , typename Fn = unspecified
    >
struct or_
{
    typedef unspecified type;
};
```

### Description

Returns the result of short-circuit *logical or* (||) operation on its arguments.

### Header

```
#include <boost/mpl/or.hpp>
#include <boost/mpl/logical.hpp>
```

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| F1, F2,... Fn | Nullary Metafunction | Operation's arguments. |

### Expression semantics

For arbitrary nullary Metafunctions f1, f2,... fn:

```
typedef or_<f1,f2,...,fn>::type r;
```

**Return type:** Integral Constant.

**Semantics:** r is true_ if either of f1::type::value, f2::type::value,... fn::type::value expressions evaluates to true, and false_ otherwise; guarantees left-to-right evaluation; the operands subsequent to the first f*i* metafunction that evaluates to true are not evaluated.

```
typedef or_<f1,f2,...,fn> r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```
struct r : or_<f1,f2,...,fn>::type {};
```

### Example

```
struct unknown;

BOOST_MPL_ASSERT(( or_< true_,true_ > ));
BOOST_MPL_ASSERT(( or_< false_,true_ > ));
BOOST_MPL_ASSERT(( or_< true_,false_ > ));
BOOST_MPL_ASSERT_NOT(( or_< false_,false_ > ));
BOOST_MPL_ASSERT(( or_< true_,unknown > )); // OK
BOOST_MPL_ASSERT(( or_< true_,unknown,unknown > )); // OK too
```

**See also**

Metafunctions, Logical Operations, `and_`, `not_`

### 4.7.3 not_

**Synopsis**

```
template<
      typename F
    >
struct not_
{
    typedef unspecified type;
};
```

**Description**

Returns the result of *logical not* (`!`) operation on its argument.

**Header**

```
#include <boost/mpl/not.hpp>
#include <boost/mpl/logical.hpp>
```

**Parameters**

| Parameter | Requirement | Description |
|---|---|---|
| `F` | Nullary Metafunction | Operation's argument. |

**Expression semantics**

For arbitrary nullary Metafunction `f`:

```
typedef not_<f>::type r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```
typedef bool_< (!f::type::value) > r;
```

```
typedef not_<f> r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```
struct r : not_<f>::type {};
```

**Example**

```
BOOST_MPL_ASSERT_NOT(( not_< true_ > ));
BOOST_MPL_ASSERT(( not_< false_ > ));
```

### See also

Metafunctions, Logical Operations, and_, or_

## 4.8 Bitwise Operations

### 4.8.1 bitand_

#### Synopsis

```
template<
      typename T1
    , typename T2
    , typename T3 = unspecified
    ...
    , typename Tn = unspecified
    >
struct bitand_
{
    typedef unspecified type;
};
```

#### Description

Returns the result of *bitwise and* (&) operation of its arguments.

#### Header

```
#include <boost/mpl/bitand.hpp>
#include <boost/mpl/bitwise.hpp>
```

#### Model of

Numeric Metafunction

#### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| T1, T2,... Tn | Integral Constant | Operation's arguments. |

[*Note:* The requirements listed in this specification are the ones imposed by the default implementation. See Numeric Metafunction concept for the details on how to provide an implementation for a user-defined numeric type that does not satisfy the Integral Constant requirements. — *end note*]

#### Expression semantics

For any Integral Constants $c_1$,$c_2$,... $c_n$:

```
typedef bitand_<c1,...cn>::type r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```
typedef integral_c<
      typeof(c1::value & c2::value)
    , ( c1::value & c2::value )
    > c;

typedef bitand_<c,c3,... cn>::type r;
```

```
typedef bitand_<c1,... cn> r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```
struct r : bitand_<c1,... cn>::type {};
```

### Complexity

Amortized constant time.

### Example

```
typedef integral_c<unsigned,0> u0;
typedef integral_c<unsigned,1> u1;
typedef integral_c<unsigned,2> u2;
typedef integral_c<unsigned,8> u8;
typedef integral_c<unsigned,0xffffffff> ufffffff;

BOOST_MPL_ASSERT_RELATION( (bitand_<u0,u0>::value), ==, 0 );
BOOST_MPL_ASSERT_RELATION( (bitand_<u1,u0>::value), ==, 0 );
BOOST_MPL_ASSERT_RELATION( (bitand_<u0,u1>::value), ==, 0 );
BOOST_MPL_ASSERT_RELATION( (bitand_<u0,ufffffff>::value), ==, 0 );
BOOST_MPL_ASSERT_RELATION( (bitand_<u1,ufffffff>::value), ==, 1 );
BOOST_MPL_ASSERT_RELATION( (bitand_<u8,ufffffff>::value), ==, 8 );
```

### See also

Bitwise Operations, Numeric Metafunction, `numeric_cast`, `bitor_`, `bitxor_`, `shift_left`

## 4.8.2 bitor_

### Synopsis

```
template<
      typename T1
    , typename T2
    , typename T3 = unspecified
    ...
    , typename Tn = unspecified
    >
struct bitor_
{
    typedef unspecified type;
```

```
};
```

### Description

Returns the result of *bitwise or* (|) operation of its arguments.

### Header

```
#include <boost/mpl/bitor.hpp>
#include <boost/mpl/bitwise.hpp>
```

### Model of

Numeric Metafunction

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| T1, T2,... Tn | Integral Constant | Operation's arguments. |

[*Note:* The requirements listed in this specification are the ones imposed by the default implementation. See Numeric Metafunction concept for the details on how to provide an implementation for a user-defined numeric type that does not satisfy the Integral Constant requirements. — *end note*]

### Expression semantics

For any Integral Constants $c_1$,$c_2$,... $c_n$:

```
typedef bitor_<c1,...cn>::type r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```
typedef integral_c<
      typeof(c1::value | c2::value)
    , ( c1::value | c2::value )
    > c;

typedef bitor_<c,c3,...cn>::type r;
```

```
typedef bitor_<c1,...cn> r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```
struct r : bitor_<c1,...cn>::type {};
```

### Complexity

Amortized constant time.

### Example

```
typedef integral_c<unsigned,0> u0;
typedef integral_c<unsigned,1> u1;
typedef integral_c<unsigned,2> u2;
typedef integral_c<unsigned,8> u8;
typedef integral_c<unsigned,0xffffffff> uffffffff;

BOOST_MPL_ASSERT_RELATION( (bitor_<u0,u0>::value), ==, 0 );
BOOST_MPL_ASSERT_RELATION( (bitor_<u1,u0>::value), ==, 1 );
BOOST_MPL_ASSERT_RELATION( (bitor_<u0,u1>::value), ==, 1 );
BOOST_MPL_ASSERT_RELATION( (bitor_<u0,uffffffff>::value), ==, 0xffffffff );
BOOST_MPL_ASSERT_RELATION( (bitor_<u1,uffffffff>::value), ==, 0xffffffff );
BOOST_MPL_ASSERT_RELATION( (bitor_<u8,uffffffff>::value), ==, 0xffffffff );
```

### See also

Bitwise Operations, Numeric Metafunction, `numeric_cast`, `bitand_`, `bitxor_`, `shift_left`

## 4.8.3 bitxor_

### Synopsis

```
template<
      typename T1
    , typename T2
    , typename T3 = unspecified
    ...
    , typename Tn = unspecified
    >
struct bitxor_
{
    typedef unspecified type;
};
```

### Description

Returns the result of *bitwise xor* (`^`) operation of its arguments.

### Header

```
#include <boost/mpl/bitxor.hpp>
#include <boost/mpl/bitwise.hpp>
```

### Model of

Numeric Metafunction

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| T1, T2,... Tn | Integral Constant | Operation's arguments. |

[*Note:* The requirements listed in this specification are the ones imposed by the default implementation. See Numeric Metafunction concept for the details on how to provide an implementation for a user-defined numeric type that does not satisfy the Integral Constant requirements. — *end note*]

### Expression semantics

For any Integral Constants $c_1$,$c_2$,... $c_n$:

```
typedef bitxor_<c1,... cn>::type r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```
typedef integral_c<
      typeof(c1::value ^ c2::value)
    , ( c1::value ^ c2::value )
    > c;

typedef bitxor_<c,c3,... cn>::type r;
```

```
typedef bitxor_<c1,... cn> r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```
struct r : bitxor_<c1,... cn>::type {};
```

### Complexity

Amortized constant time.

### Example

```
typedef integral_c<unsigned,0> u0;
typedef integral_c<unsigned,1> u1;
typedef integral_c<unsigned,2> u2;
typedef integral_c<unsigned,8> u8;
typedef integral_c<unsigned,0xffffffff> uffffffff;

BOOST_MPL_ASSERT_RELATION( (bitxor_<u0,u0>::value), ==, 0 );
BOOST_MPL_ASSERT_RELATION( (bitxor_<u1,u0>::value), ==, 1 );
BOOST_MPL_ASSERT_RELATION( (bitxor_<u0,u1>::value), ==, 1 );

BOOST_MPL_ASSERT_RELATION( (bitxor_<u0,uffffffff>::value), ==, 0xffffffff ^ 0 );
BOOST_MPL_ASSERT_RELATION( (bitxor_<u1,uffffffff>::value), ==, 0xffffffff ^ 1 );
BOOST_MPL_ASSERT_RELATION( (bitxor_<u8,uffffffff>::value), ==, 0xffffffff ^ 8 );
```

### See also

Bitwise Operations, Numeric Metafunction, `numeric_cast`, `bitand_`, `bitor_`, `shift_left`

### 4.8.4 shift_left

#### Synopsis

```
template<
      typename T
    , typename Shift
    >
struct shift_left
{
    typedef unspecified type;
};
```

#### Description

Returns the result of bitwise *shift left* (<<) operation on `T`.

#### Header

```
#include <boost/mpl/shift_left.hpp>
#include <boost/mpl/bitwise.hpp>
```

#### Model of

Numeric Metafunction

#### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `T` | Integral Constant | A value to shift. |
| `Shift` | Unsigned Integral Constant | A shift distance. |

[*Note:* The requirements listed in this specification are the ones imposed by the default implementation. See Numeric Metafunction concept for the details on how to provide an implementation for a user-defined numeric type that does not satisfy the Integral Constant requirements. — *end note*]

#### Expression semantics

For arbitrary Integral Constant `c` and unsigned Integral Constant `shift`:

```
typedef shift_left<c,shift>::type r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```
typedef integral_c<
      c::value_type
    , ( c::value << shift::value )
    > r;
```

```
typedef shift_left<c,shift> r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```
struct r : shift_left<c,shift>::type {};
```

### Complexity

Amortized constant time.

### Example

```
typedef integral_c<unsigned,0> u0;
typedef integral_c<unsigned,1> u1;
typedef integral_c<unsigned,2> u2;
typedef integral_c<unsigned,8> u8;

BOOST_MPL_ASSERT_RELATION( (shift_left<u0,u0>::value), ==, 0 );
BOOST_MPL_ASSERT_RELATION( (shift_left<u1,u0>::value), ==, 1 );
BOOST_MPL_ASSERT_RELATION( (shift_left<u1,u1>::value), ==, 2 );
BOOST_MPL_ASSERT_RELATION( (shift_left<u2,u1>::value), ==, 4 );
BOOST_MPL_ASSERT_RELATION( (shift_left<u8,u1>::value), ==, 16 );
```

### See also

Bitwise Operations, Numeric Metafunction, `numeric_cast`, `shift_right`, `bitand_`

## 4.8.5 shift_right

### Synopsis

```
template<
      typename T
    , typename Shift
    >
struct shift_right
{
    typedef unspecified type;
};
```

### Description

Returns the result of bitwise *shift right* (>>) operation on `T`.

### Header

```
#include <boost/mpl/shift_right.hpp>
#include <boost/mpl/bitwise.hpp>
```

### Model of

Numeric Metafunction

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `T` | Integral Constant | A value to shift. |
| `Shift` | Unsigned Integral Constant | A shift distance. |

[*Note:* The requirements listed in this specification are the ones imposed by the default implementation. See Numeric Metafunction concept for the details on how to provide an implementation for a user-defined numeric type that does not satisfy the Integral Constant requirements. — *end note*]

### Expression semantics

For arbitrary Integral Constant `c` and unsigned Integral Constant `shift`:

```
typedef shift_right<c,shift>::type r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```
typedef integral_c<
      c::value_type
    , ( c::value >> shift::value )
    > r;
```

```
typedef shift_right<c,shift> r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```
struct r : shift_right<c,shift>::type {};
```

### Complexity

Amortized constant time.

### Example

```
typedef integral_c<unsigned,0> u0;
typedef integral_c<unsigned,1> u1;
typedef integral_c<unsigned,2> u2;
typedef integral_c<unsigned,8> u8;

BOOST_MPL_ASSERT_RELATION( (shift_right<u0,u0>::value), ==, 0 );
BOOST_MPL_ASSERT_RELATION( (shift_right<u1,u0>::value), ==, 1 );
BOOST_MPL_ASSERT_RELATION( (shift_right<u1,u1>::value), ==, 0 );
BOOST_MPL_ASSERT_RELATION( (shift_right<u2,u1>::value), ==, 1 );
BOOST_MPL_ASSERT_RELATION( (shift_right<u8,u1>::value), ==, 4 );
```

### See also

Bitwise Operations, Numeric Metafunction, `numeric_cast`, `shift_left`, `bitand_`

### 4.9 Trivial

The MPL provides a number of Trivial Metafunctions that a nothing more than thin wrappers for a differently-named class nested type members. While important in the context of in-place metafunction composition, these metafunctions have so little to them that presenting them in the same format as the rest of the compoments in this manual would result in more boilerplate syntactic baggage than the actual content. To avoid this problem, we instead factor out the common metafunctions' requirements into the corresponding concept and gather all of them in a single place — this subsection — in a compact table form that is presented below.

#### 4.9.1 Trivial Metafunctions Summary

In the following table, `x` is an arbitrary class type.

| Metafunction | Header |
|---|---|
| `first<x>::type` | `#include <boost/mpl/pair.hpp>` |
| `second<x>::type` | `#include <boost/mpl/pair.hpp>` |
| `base<x>::type` | `#include <boost/mpl/base.hpp>` |

**See Also**

Metafunctions, Trivial Metafunction

### 4.10 Miscellaneous

#### 4.10.1 identity

**Synopsis**

```
template<
      typename X
    >
struct identity
{
    typedef X type;
};
```

**Description**

The identity metafunction. Returns `X` unchanged.

**Header**

```
#include <boost/mpl/identity.hpp>
```

**Model of**

Metafunction

**Parameters**

| Parameter | Requirement | Description |
|---|---|---|
| X | Any type | An argument to be returned. |

### Expression semantics

For an arbitrary type x:

```
typedef identity<x>::type r;
```

**Return type:** A type.

**Semantics:** Equivalent to

```
typedef x r;
```

**Postcondition:** `is_same<r,x>::value == true`.

### Example

```
typedef apply< identity<_1>, char >::type t1;
typedef apply< identity<_2>, char,int >::type t2;

BOOST_MPL_ASSERT(( is_same< t1, char > ));
BOOST_MPL_ASSERT(( is_same< t2, int > ));
```

### See also

Metafunctions, Placeholders, Trivial Metafunctions, `always`, `apply`

## 4.10.2 always

### Synopsis

```
template<
      typename X
    >
struct always
{
    // unspecified
    // ...
};
```

### Description

`always<X>` specialization is a variadic Metafunction Class always returning the same type, `X`, regardless of the number and types of passed arguments.

### Header

```
#include <boost/mpl/always.hpp>
```

### Model of

Metafunction Class

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `X` | Any type | A type to be returned. |

### Expression semantics

For an arbitrary type `x`:

```
typedef always<x> f;
```

**Return type:** Metafunction Class.

**Semantics:** Equivalent to

```
struct f : bind< identity<_1>, x > {};
```

### Example

```
typedef always<true_> always_true;

BOOST_MPL_ASSERT(( apply< always_true,false_> ));
BOOST_MPL_ASSERT(( apply< always_true,false_,false_ > ));
BOOST_MPL_ASSERT(( apply< always_true,false_,false_,false_ > ));
```

### See also

Metafunctions, Metafunction Class, `identity`, `bind`, `apply`

## 4.10.3 inherit

### Synopsis

```
template<
      typename T1, typename T2
    >
struct inherit2
{
    typedef unspecified type;
};

...

template<
      typename T1, typename T2,... typename Tn
    >
struct inheritn
{
    typedef unspecified type;
```

```
};

template<
      typename T1
    , typename T2
    ...
    , typename Tn = unspecified
    >
struct inherit
{
    typedef unspecified type;
};
```

### Description

Returns an unspecified class type publically derived from `T1`, `T2`,... `Tn`. Guarantees that derivation from `empty_base` is always a no-op, regardless of the position and number of `empty_base` classes in `T1`, `T2`,... `Tn`.

### Header

```
#include <boost/mpl/inherit.hpp>
```

### Model of

Metafunction

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `T1`, `T2`,... `Tn` | A class type | Classes to derived from. |

### Expression semantics

For artibrary class types $t_1$,$t_2$,... $t_n$:

```
typedef inherit2<t1,t2>::type r;
```

**Return type:** A class type.

**Precondition:** `t1` and `t2` are complete types.

**Semantics:** If both `t1` and `t2` are identical to `empty_base`, equivalent to

```
typedef empty_base r;
```

otherwise, if `t1` is identical to `empty_base`, equivalent to

```
typedef t2 r;
```

otherwise, if `t2` is identical to `empty_base`, equivalent to

```
typedef t1 r;
```

otherwise equivalent to

```
struct r : t1, t2 {};
```

```
typedef inheritn<t1,t2,...tn>::type r;
```

**Return type:** A class type.

**Precondition:** $t_1$,$t_2$,... $t_n$ are complete types.

**Semantics:** Equivalent to

```
struct r
    : inherit2<
          inheritn-1<t1,t2,...tn-1>::type
        , tn
        >
{
};
```

```
typedef inherit<t1,t2,...tn>::type r;
```

**Precondition:** $t_1$,$t_2$,... $t_n$ are complete types.

**Return type:** A class type.

**Semantics:** Equivalent to

```
typedef inheritn<t1,t2,...tn>::type r;
```

### Complexity

Amortized constant time.

### Example

```
struct udt1 { int n; };
struct udt2 {};

typedef inherit<udt1,udt2>::type r1;
typedef inherit<empty_base,udt1>::type r2;
typedef inherit<empty_base,udt1,empty_base,empty_base>::type r3;
typedef inherit<udt1,empty_base,udt2>::type r4;
typedef inherit<empty_base,empty_base>::type r5;

BOOST_MPL_ASSERT(( is_base_and_derived< udt1, r1> ));
BOOST_MPL_ASSERT(( is_base_and_derived< udt2, r1> ));
BOOST_MPL_ASSERT(( is_same< r2, udt1> ));
BOOST_MPL_ASSERT(( is_same< r3, udt1 > ));
BOOST_MPL_ASSERT(( is_base_and_derived< udt1, r4 > ));
BOOST_MPL_ASSERT(( is_base_and_derived< udt2, r4 > ));
BOOST_MPL_ASSERT(( is_same< r5, empty_base > ));
```

### See also

Metafunctions, `empty_base`, `inherit_linearly`, `identity`

### 4.10.4 inherit_linearly

#### Synopsis

```
template<
      typename Types
    , typename Node
    , typename Root = empty_base
    >
struct inherit_linearly
    : fold<Types,Root,Node>
{
};
```

#### Description

A convenience wrapper for `fold` to use in the context of sequence-driven class composition. Returns the result the successive application of binary `Node` to the result of the previous `Node` invocation (`Root` if it's the first call) and every type in the Forward Sequence `Types` in order.

#### Header

```
#include <boost/mpl/inherit_linearly.hpp>
```

#### Model of

Metafunction

#### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `Types` | Forward Sequence | Types to inherit from. |
| `Node` | Binary Lambda Expression | A derivation metafunction. |
| `Root` | A class type | A type to be placed at the root of the class hierarchy. |

#### Expression semantics

For any Forward Sequence `types`, binary Lambda Expression `node`, and arbitrary class type `root`:

```
typedef inherit_linearly<types,node,root>::type r;
```

**Return type:** A class type.

**Semantics:** Equivalent to

```
typedef fold<types,root,node>::type r;
```

#### Complexity

Linear. Exactly `size<types>::value` applications of `node`.

### Example

```
template< typename T > struct tuple_field
{
    T field;
};

template< typename T >
inline
T& field(tuple_field<T>& t)
{
    return t.field;
}

typedef inherit_linearly<
      vector<int,char const*,bool>
    , inherit< _1, tuple_field<_2> >
    >::type tuple;


int main()
{
    tuple t;

    field<int>(t) = -1;
    field<char const*>(t) = "text";
    field<bool>(t) = false;

    std::cout
        << field<int>(t) << 'n'
        << field<char const*>(t) << 'n'
        << field<bool>(t) << 'n'
        ;
}
```

### See also

Metafunctions, Algorithms, `inherit`, `empty_base`, `fold`, `reverse_fold`

## 4.10.5 numeric_cast

### Synopsis

```
template<
      typename SourceTag
    , typename TargetTag
    >
struct numeric_cast;
```

### Description

Each `numeric_cast` specialization is a user-specialized unary Metafunction Class providing a conversion between two numeric types.

### Header

```
#include <boost/mpl/numeric_cast.hpp>
```

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| SourceTag | Integral Constant | A tag for the conversion's source type. |
| TargetTag | Integral Constant | A tag for the conversion's destination type. |

### Expression semantics

If x and y are two numeric types, x is convertible to y, and x_tag and y_tag are the types' corresponding Integral Constant tags:

```
typedef apply_wrap2< numeric_cast<x_tag,y_tag>,x >::type  r;
```

**Return type:** A type.

**Semantics:** r is a value of x converted to the type of y.

### Complexity

Unspecified.

### Example

```
struct complex_tag : int_<10> {};

template< typename Re, typename Im > struct complex
{
    typedef complex_tag tag;
    typedef complex type;
    typedef Re real;
    typedef Im imag;
};

template< typename C > struct real : C::real {};
template< typename C > struct imag : C::imag {};

namespace boost { namespace mpl {

template<> struct numeric_cast< integral_c_tag,complex_tag >
{
    template< typename N > struct apply
        : complex< N, integral_c< typename N::value_type, 0 > >
    {
    };
};

template<>
struct plus_impl< complex_tag,complex_tag >
```

```
{
    template< typename N1, typename N2 > struct apply
        : complex<
              plus< typename N1::real, typename N2::real >
            , plus< typename N1::imag, typename N2::imag >
            >
    {
    };
};

}}

typedef int_<2> i;
typedef complex< int_<5>, int_<-1> > c1;
typedef complex< int_<-5>, int_<1> > c2;

typedef plus<c1,i> r4;
BOOST_MPL_ASSERT_RELATION( real<r4>::value, ==, 7 );
BOOST_MPL_ASSERT_RELATION( imag<r4>::value, ==, -1 );

typedef plus<i,c2> r5;
BOOST_MPL_ASSERT_RELATION( real<r5>::value, ==, -3 );
BOOST_MPL_ASSERT_RELATION( imag<r5>::value, ==, 1 );
```

**See also**

Metafunctions, Numeric Metafunction, `plus`, `minus`, `times`

### 4.10.6 min

**Synopsis**

```
template<
      typename N1
    , typename N2
    >
struct min
{
    typedef unspecified type;
};
```

**Description**

Returns the smaller of its two arguments.

**Header**

```
#include <boost/mpl/min_max.hpp>
```

**Model of**

Metafunction

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| N1, N2 | Any type | Types to compare. |

### Expression semantics

For arbitrary types x and y:

```
typedef min<x,y>::type r;
```

**Return type:** A type.

**Precondition:** `less<x,y>::value` is a well-formed integral constant expression.

**Semantics:** Equivalent to

```
typedef if_< less<x,y>,x,y >::type r;
```

### Complexity

Constant time.

### Example

```
typedef fold<
      vector_c<int,1,7,0,-2,5,-1>
    , int_<-10>
    , min<_1,_2>
    >::type r;

BOOST_MPL_ASSERT(( is_same< r, int_<-10> > ));
```

### See also

Metafunctions, comparison, `max`, `less`, `min_element`

## 4.10.7 max

### Synopsis

```
template<
      typename N1
    , typename N2
    >
struct max
{
    typedef unspecified type;
};
```

### Description

Returns the larger of its two arguments.

### Header

```
#include <boost/mpl/min_max.hpp>
```

### Model of

Metafunction

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| N1, N2 | Any type | Types to compare. |

### Expression semantics

For arbitrary types x and y:

```
typedef max<x,y>::type r;
```

**Return type:** A type.

**Precondition:** `less<x,y>::value` is a well-formed integral constant expression.

**Semantics:** Equivalent to

```
typedef if_< less<x,y>,y,x >::type r;
```

### Complexity

Constant time.

### Example

```
typedef fold<
      vector_c<int,1,7,0,-2,5,-1>
    , int_<10>
    , max<_1,_2>
    >::type r;

BOOST_MPL_ASSERT(( is_same< r, int_<10> > ));
```

### See also

Metafunctions, comparison, `min`, `less`, `max_element`

## 4.10.8 sizeof_

### Synopsis

```
template<
      typename X
    >
struct sizeof_
```

```
{
    typedef unspecified type;
};
```

### Description

Returns the result of a `sizeof(X)` expression wrapped into an Integral Constant of the corresponding type, `std::size_t`.

### Header

```
#include <boost/mpl/sizeof.hpp>
```

### Model of

Metafunction

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `X` | Any type | A type to compute the `sizeof` for. |

### Expression semantics

For an arbitrary type `x`:

```
typedef sizeof_<x>::type n;
```

**Return type:** Integral Constant.

**Precondition:** `x` is a complete type.

**Semantics:** Equivalent to

```
typedef size_t< sizeof(x) > n;
```

### Complexity

Constant time.

### Example

```
struct udt { char a[100]; };

BOOST_MPL_ASSERT_RELATION( sizeof_<char>::value, ==, sizeof(char) );
BOOST_MPL_ASSERT_RELATION( sizeof_<int>::value, ==, sizeof(int) );
BOOST_MPL_ASSERT_RELATION( sizeof_<double>::value, ==, sizeof(double) );
BOOST_MPL_ASSERT_RELATION( sizeof_<udt>::value, ==, sizeof(my) );
```

### See also

Metafunctions, Integral Constant, `size_t`

# Chapter 5 Data Types

## 5.1 Concepts

### 5.1.1 Integral Constant

#### Description

An Integral Constant is a holder class for a compile-time value of an integral type. Every Integral Constant is also a nullary Metafunction, returning itself. An integral constant *object* is implicitly convertible to the corresponding run-time value of the wrapped integral type.

#### Expression requirements

In the following table and subsequent specifications, `n` is a model of Integral Constant.

| Expression | Type | Complexity |
|---|---|---|
| `n::value_type` | An integral type | Constant time. |
| `n::value` | An integral constant expression | Constant time. |
| `n::type` | Integral Constant | Constant time. |
| `next<n>::type` | Integral Constant | Constant time. |
| `prior<n>::type` | Integral Constant | Constant time. |
| `n::value_type const c = n()` | | Constant time. |

#### Expression semantics

| Expression | Semantics |
|---|---|
| `n::value_type` | A cv-unqualified type of `n::value`. |
| `n::value` | The value of the wrapped integral constant. |
| `n::type` | `is_same<n::type,n>::value == true`. |
| `next<n>::type` | An Integral Constant `c` of type `n::value_type` such that `c::value == n::value + 1`. |
| `prior<n>::type` | An Integral Constant `c` of type `n::value_type` such that `c::value == n::value - 1`. |
| `n::value_type const c = n()` | `c == n::value`. |

#### Models

— `bool_`

— `int_`

— long_

— integral_c

**See also**

Data Types, Integral Sequence Wrapper, integral_c

## 5.2 Numeric

### 5.2.1 bool_

**Synopsis**

```
template<
      bool C
    >
struct bool_
{
    // unspecified
    // ...
};

typedef bool_<true>  true_;
typedef bool_<false> false_;
```

**Description**

A boolean Integral Constant wrapper.

**Header**

```
#include <boost/mpl/bool.hpp>
```

**Model of**

Integral Constant

**Parameters**

| Parameter | Requirement | Description |
|---|---|---|
| C | A boolean integral constant | A value to wrap. |

**Expression semantics**

The semantics of an expression are defined only where they differ from, or are not defined in Integral Constant.

For arbitrary integral constant c:

| Expression | Semantics |
|---|---|
| bool_<c> | An Integral Constant x such that x::value == c and x::value_type is identical to bool. |

### Example

```
BOOST_MPL_ASSERT(( is_same< bool_<true>::value_type, bool > ));
BOOST_MPL_ASSERT(( is_same< bool_<true>, true_ > )); }
BOOST_MPL_ASSERT(( is_same< bool_<true>::type, bool_<true> > ));
BOOST_MPL_ASSERT_RELATION( bool_<true>::value, ==, true );
assert( bool_<true>() == true );
```

### See also

Data Types, Integral Constant, int_, long_, integral_c

## 5.2.2 int_

### Synopsis

```
template<
      int N
    >
struct int_
{
    // unspecified
    // ...
};
```

### Description

An Integral Constant wrapper for int.

### Header

```
#include <boost/mpl/int.hpp>
```

### Model of

Integral Constant

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| N | An integral constant | A value to wrap. |

### Expression semantics

The semantics of an expression are defined only where they differ from, or are not defined in Integral Constant.

For arbitrary integral constant n:

| Expression | Semantics |
|---|---|
| int_<c> | An Integral Constant x such that x::value == c and x::value_type is identical to int. |

### Example

```
typedef int_<8> eight;

BOOST_MPL_ASSERT(( is_same< eight::value_type, int > ));
BOOST_MPL_ASSERT(( is_same< eight::type, eight > ));
BOOST_MPL_ASSERT(( is_same< next< eight >::type, int_<9> > ));
BOOST_MPL_ASSERT(( is_same< prior< eight >::type, int_<7> > ));
BOOST_MPL_ASSERT_RELATION( (eight::value), ==, 8 );
assert( eight() == 8 );
```

### See also

Data Types, Integral Constant, `long_`, `size_t`, `integral_c`

## 5.2.3 long_

### Synopsis

```
template<
      long N
    >
struct long_
{
    // unspecified
    // ...
};
```

### Description

An Integral Constant wrapper for `long`.

### Header

```
#include <boost/mpl/long.hpp>
```

### Model of

Integral Constant

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `N` | An integral constant | A value to wrap. |

### Expression semantics

The semantics of an expression are defined only where they differ from, or are not defined in Integral Constant.

For arbitrary integral constant `n`:

| Expression | Semantics |
|---|---|
| `long_<c>` | An Integral Constant `x` such that `x::value == c` and `x::value_type` is identical to `long`. |

**Example**

```
typedef long_<8> eight;

BOOST_MPL_ASSERT(( is_same< eight::value_type, long > ));
BOOST_MPL_ASSERT(( is_same< eight::type, eight > ));
BOOST_MPL_ASSERT(( is_same< next< eight >::type, long_<9> > ));
BOOST_MPL_ASSERT(( is_same< prior< eight >::type, long_<7> > ));
BOOST_MPL_ASSERT_RELATION( (eight::value), ==, 8 );
assert( eight() == 8 );
```

**See also**

Data Types, Integral Constant, `int_`, `size_t`, `integral_c`

### 5.2.4 size_t

**Synopsis**

```
template<
      std::size_t N
    >
struct size_t
{
    // unspecified
    // ...
};
```

**Description**

An Integral Constant wrapper for `std::size_t`.

**Header**

```
#include <boost/mpl/size_t.hpp>
```

**Model of**

Integral Constant

**Parameters**

| Parameter | Requirement | Description |
|---|---|---|
| `N` | An integral constant | A value to wrap. |

### Expression semantics

The semantics of an expression are defined only where they differ from, or are not defined in Integral Constant.

For arbitrary integral constant `n`:

| Expression | Semantics |
|---|---|
| `size_t<c>` | An Integral Constant `x` such that `x::value == c` and `x::value_type` is identical to `std::size_t`. |

### Example

```
typedef size_t<8> eight;

BOOST_MPL_ASSERT(( is_same< eight::value_type, std::size_t > ));
BOOST_MPL_ASSERT(( is_same< eight::type, eight > ));
BOOST_MPL_ASSERT(( is_same< next< eight >::type, size_t<9> > ));
BOOST_MPL_ASSERT(( is_same< prior< eight >::type, size_t<7> > ));
BOOST_MPL_ASSERT_RELATION( (eight::value), ==, 8 );
assert( eight() == 8 );
```

### See also

Data Types, Integral Constant, `int_`, `long_`, `integral_c`

## 5.2.5 integral_c

### Synopsis

```
template<
      typename T, T N
    >
struct integral_c
{
    // unspecified
    // ...
};
```

### Description

A generic Integral Constant wrapper.

### Header

```
#include <boost/mpl/integral_c.hpp>
```

### Model of

Integral Constant

**Parameters**

| Parameter | Requirement | Description |
|---|---|---|
| T | An integral type | Wrapper's value type. |
| N | An integral constant | A value to wrap. |

### Expression semantics

The semantics of an expression are defined only where they differ from, or are not defined in Integral Constant.

For arbitrary integral type t and integral constant n:

| Expression | Semantics |
|---|---|
| integral_c<t,c> | An Integral Constant x such that x::value == c and x::value_type is identical to t. |

### Example

```
typedef integral_c<short,8> eight;

BOOST_MPL_ASSERT(( is_same< eight::value_type, short > ));
BOOST_MPL_ASSERT(( is_same< eight::type, eight > ));
BOOST_MPL_ASSERT(( is_same< next< eight >::type, integral_c<short,9> > ));
BOOST_MPL_ASSERT(( is_same< prior< eight >::type, integral_c<short,7> > ));
BOOST_MPL_ASSERT_RELATION( (eight::value), ==, 8 );
assert( eight() == 8 );
```

### See also

Data Types, Integral Constant, bool_, int_, long_, size_t

## 5.3 Miscellaneous

### 5.3.1 pair

### Synopsis

```
template<
      typename T1
    , typename T2
    >
struct pair
{
    typedef pair type;
    typedef T1 first;
    typedef T2 second;
};
```

### Description

A transparent holder for two arbitrary types.

### Header

```
#include <boost/mpl/pair.hpp>
```

### Example

Count a number of elements in the sequence together with a number of negative elements among these.

```
typedef fold<
      vector_c<int,-1,0,5,-7,-2,4,5,7>
    , pair< int_<0>, int_<0> >
    , pair<
          next< first<_1> >
        , if_<
              less< _2, int_<0> >
            , next< second<_1> >
            , second<_1>
            >
        >
    >::type p;

BOOST_MPL_ASSERT_RELATION( p::first::value, ==, 8 );
BOOST_MPL_ASSERT_RELATION( p::second::value, ==, 3 );
```

### See also

Data Types, Sequences, `first`, `second`

## 5.3.2 empty_base

### Synopsis

```
struct empty_base {};
```

### Description

An empty base class. Inheritance from `empty_base` through the `inherit` metafunction is a no-op.

### Header

```
#include <boost/mpl/empty_base.hpp>
```

### See also

Data Types, `inherit`, `inherit_linearly`, `void_`

### 5.3.3 void_

#### Synopsis

```
struct void_
{
    typedef void_ type;
};

template< typename T > struct is_void;
```

#### Description

`void_` is a generic type placeholder representing “nothing”.

#### Header

```
#include <boost/mpl/void.hpp>
```

#### See also

Data Types, `pair`, `empty_base`, `bool_`, `int_`, `integral_c`

# Chapter 6 Macros

Being a *template* metaprogramming framework, the MPL concentrates on getting one thing done well and leaves most of the clearly preprocessor-related tasks to the corresponding specialized libraries [PRE], [Ve03]. But whether we like it or not, macros play an important role on today's C++ metaprogramming, and some of the useful MPL-level functionality cannot be implemented without leaking its preprocessor-dependent implementation nature into the library's public interface.

## 6.1 Asserts

The MPL supplies a suite of static assertion macros that are specifically designed to generate maximally useful and informative error messages within the diagnostic capabilities of each compiler.

All assert macros can be used at class, function, or namespace scope.

### 6.1.1 BOOST_MPL_ASSERT

**Synopsis**

```
#define BOOST_MPL_ASSERT( pred ) \
    unspecified token sequence \
/**/
```

**Description**

Generates a compilation error when the predicate `pred` holds false.

**Header**

```
#include <boost/mpl/assert.hpp>
```

**Parameters**

| Parameter | Requirement | Description |
|---|---|---|
| `pred` | Boolean nullary Metafunction | A predicate to be asserted. |

**Expression semantics**

For any boolean nullary Metafunction `pred`:

```
BOOST_MPL_ASSERT(( pred ));
```

**Return type:** None.

**Semantics:** Generates a compilation error if `pred::type::value != true`, otherwise has no effect. Note that double parentheses are required even if no commas appear in the condition.

When possible within the compiler's diagnostic capabilities, the error message will include the predicate's full type name, and have a general form of:

```
... ************ pred::************ ...
```

### Example

```
template< typename T, typename U > struct my
{
    // ...
    BOOST_MPL_ASSERT(( is_same< T,U > ));
};

my<void*,char*> test;

// In instantiation of 'my<void, char*>':
//   instantiated from here
// conversion from '
//   mpl_::failed************boost::is_same<void, char*>::************' to
//   non-scalar type 'mpl_::assert<false>' requested
```

### See also

Asserts, BOOST_MPL_ASSERT_NOT, BOOST_MPL_ASSERT_MSG, BOOST_MPL_ASSERT_RELATION

## 6.1.2 BOOST_MPL_ASSERT_MSG

### Synopsis

```
#define BOOST_MPL_ASSERT_MSG( condition, message, types ) \
    unspecified token sequence \
/**/
```

### Description

Generates a compilation error with an embedded custom message when the condition doesn't hold.

### Header

```
#include <boost/mpl/assert.hpp>
```

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `condition` | An integral constant expression | A condition to be asserted. |
| `message` | A legal identifier token | A custom message in a form of a legal C++ identifier token. |

| Parameter | Requirement | Description |
|---|---|---|
| types | A legal function parameter list | A parenthized list of types to be displayed in the error message. |

**Expression semantics**

For any integral constant expression expr, legal C++ identifier message, and arbitrary types t1, t2,... tn:

```
BOOST_MPL_ASSERT_MSG( expr, message, (t1, t2,... tn) );
```

**Return type:** None.

**Precondition:** t1, t2,... tn are non-void.

**Semantics:** Generates a compilation error if expr::value != true, otherwise has no effect.

When possible within the compiler's diagnostic capabilities, the error message will include the message identifier and the parenthized list of t1, t2,... tn types, and have a general form of:

```
... ************( ...::message )************)(t1, t2,... tn) ...
```

```
BOOST_MPL_ASSERT_MSG( expr, message, (types<t1, t2,... tn>) );
```

**Return type:** None.

**Precondition:** None.

**Semantics:** Generates a compilation error if expr::value != true, otherwise has no effect.

When possible within the compiler's diagnostics capabilities, the error message will include the message identifier and the list of t1, t2,... tn types, and have a general form of:

```
... ************( ...::message )************)(types<t1, t2,... tn>) ...
```

**Example**

```
template< typename T > struct my
{
    // ...
    BOOST_MPL_ASSERT_MSG(
          is_integral<T>::value
        , NON_INTEGRAL_TYPES_ARE_NOT_ALLOWED
        , (T)
        );
};

my<void*> test;

// In instantiation of ‘my<void*>’:
//   instantiated from here
// conversion from ‘
//   mpl_::failed************(my<void*>::
//   NON_INTEGRAL_TYPES_ARE_NOT_ALLOWED::************)(void*)
//   ’ to non-scalar type ‘mpl_::assert<false>’ requested
```

**See also**

Asserts, BOOST_MPL_ASSERT, BOOST_MPL_ASSERT_NOT, BOOST_MPL_ASSERT_RELATION

### 6.1.3 BOOST_MPL_ASSERT_NOT

#### Synopsis

```
#define BOOST_MPL_ASSERT_NOT( pred ) \
    unspecified token sequence \
/**/
```

#### Description

Generates a compilation error when predicate holds true.

#### Header

```
#include <boost/mpl/assert.hpp>
```

#### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `pred` | Boolean nullary Metafunction | A predicate to be asserted to be false. |

#### Expression semantics

For any boolean nullary Metafunction `pred`:

```
BOOST_MPL_ASSERT_NOT(( pred ));
```

**Return type:** None.

**Semantics:** Generates a compilation error if `pred::type::value != false`, otherwise has no effect. Note that double parentheses are required even if no commas appear in the condition.

When possible within the compiler's diagnostic capabilities, the error message will include the predicate's full type name, and have a general form of:

```
... ************boost::mpl::not_< pred >::************ ...
```

#### Example

```
template< typename T, typename U > struct my
{
    // ...
    BOOST_MPL_ASSERT_NOT(( is_same< T,U > ));
};

my<void,void> test;

// In instantiation of ‘my<void, void>’:
//   instantiated from here
// conversion from ‘
//   mpl_::failed************boost::mpl::not_<boost::is_same<void, void>
//   >::************’ to non-scalar type ‘mpl_::assert<false>’ requested
```

**See also**

Asserts, BOOST_MPL_ASSERT, BOOST_MPL_ASSERT_MSG, BOOST_MPL_ASSERT_RELATION

### 6.1.4 BOOST_MPL_ASSERT_RELATION

**Synopsis**

```
#define BOOST_MPL_ASSERT_RELATION( x, relation, y ) \
    unspecified token sequence \
/**/
```

**Description**

A specialized assertion macro for checking numerical conditions. Generates a compilation error when the condition `( x relation y )` doesn't hold.

**Header**

```
#include <boost/mpl/assert.hpp>
```

**Parameters**

| Parameter | Requirement | Description |
|---|---|---|
| `x` | An integral constant | Left operand of the checked relation. |
| `y` | An integral constant | Right operand of the checked relation. |
| `relation` | A C++ operator token | An operator token for the relation being checked. |

**Expression semantics**

For any integral constants `x`, `y` and a legal C++ operator token `op`:

```
BOOST_MPL_ASSERT_RELATION( x, op, y );
```

**Return type:** None.

**Semantics:** Generates a compilation error if `( x op y ) != true`, otherwise has no effect.

When possible within the compiler's diagnostic capabilities, the error message will include a name of the relation being checked, the actual values of both operands, and have a general form of:

```
... ************...assert_relation<op, x, y>::************) ...
```

**Example**

```
template< typename T, typename U > struct my
{
    // ...
    BOOST_MPL_ASSERT_RELATION( sizeof(T), <, sizeof(U) );
};

my<char[50],char[10]> test;
```

```
// In instantiation of ‘my<char[50], char[10]>’:
//   instantiated from here
// conversion from ‘
//   mpl_::failed************mpl_::assert_relation<less, 50, 10>::************’
//   to non-scalar type ‘mpl_::assert<false>’ requested
```

### See also

Asserts, BOOST_MPL_ASSERT, BOOST_MPL_ASSERT_NOT, BOOST_MPL_ASSERT_MSG

## 6.2 Introspection

### 6.2.1 BOOST_MPL_HAS_XXX_TRAIT_DEF

#### Synopsis

```
#define BOOST_MPL_HAS_XXX_TRAIT_DEF(name) \
    unspecified token sequence \
/**/
```

#### Description

Expands into a definition of a boolean unary Metafunction `has_name` such that for any type `x` `has_name<x>::value == true` if and only if `x` is a class type and has a nested type memeber `x::name`.

On the deficient compilers not capabale of performing the detection, `has_name<x>::value` always returns `false`. A boolean configuraion macro, BOOST_MPL_CFG_NO_HAS_XXX, is provided to signal or override the “deficient” status of a particular compiler.

[*Note:* BOOST_MPL_HAS_XXX_TRAIT_DEF is a simplified front end to the BOOST_MPL_HAS_XXX_TRAIT_NAMED_DEF introspection macro — *end note*]

#### Header

```
#include <boost/mpl/has_xxx.hpp>
```

#### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `name` | A legal identifier token | A name of the member being detected. |

#### Expression semantics

For any legal C++ identifier `name`:

`BOOST_MPL_HAS_XXX_TRAIT_DEF(name)`

**Precondition:** Appears at namespace scope.

**Return type:** None.

**Semantics:** Equivalent to

```
BOOST_MPL_HAS_XXX_TRAIT_NAMED_DEF(
      BOOST_PP_CAT(has_,name), name, false
```

```
)
```

**Example**

```
BOOST_MPL_HAS_XXX_TRAIT_DEF(has_xxx)

struct test1 {};
struct test2 { void xxx(); };
struct test3 { int xxx; };
struct test4 { static int xxx(); };
struct test5 { template< typename T > struct xxx {}; };
struct test6 { typedef int xxx; };
struct test7 { struct xxx; };
struct test8 { typedef void (*xxx)(); };
struct test9 { typedef void (xxx)(); };

BOOST_MPL_ASSERT_NOT(( has_xxx<test1> ));
BOOST_MPL_ASSERT_NOT(( has_xxx<test2> ));
BOOST_MPL_ASSERT_NOT(( has_xxx<test3> ));
BOOST_MPL_ASSERT_NOT(( has_xxx<test4> ));
BOOST_MPL_ASSERT_NOT(( has_xxx<test5> ));

#if !defined(BOOST_MPL_CFG_NO_HAS_XXX)
BOOST_MPL_ASSERT(( has_xxx<test6> ));
BOOST_MPL_ASSERT(( has_xxx<test7> ));
BOOST_MPL_ASSERT(( has_xxx<test8> ));
BOOST_MPL_ASSERT(( has_xxx<test9> ));
#endif

BOOST_MPL_ASSERT(( has_xxx<test6,true_> ));
BOOST_MPL_ASSERT(( has_xxx<test7,true_> ));
BOOST_MPL_ASSERT(( has_xxx<test8,true_> ));
BOOST_MPL_ASSERT(( has_xxx<test9,true_> ));
```

**See also**

Macros, BOOST_MPL_HAS_XXX_TRAIT_NAMED_DEF, BOOST_MPL_CFG_NO_HAS_XXX

### 6.2.2 BOOST_MPL_HAS_XXX_TRAIT_NAMED_DEF

**Synopsis**

```
#define BOOST_MPL_HAS_XXX_TRAIT_NAMED_DEF(trait, name, default_) \
    unspecified token sequence \
/**/
```

**Description**

Expands into a definition of a boolean unary Metafunction `trait` such that for any type `x` `trait<x>::value == true` if and only if `x` is a class type and has a nested type memeber `x::name`.

On the deficient compilers not capabale of performing the detection, `trait<x>::value` always returns a fallback value `default_`. A boolean configuraion macro, `BOOST_MPL_CFG_NO_HAS_XXX`, is provided to signal or override the "deficient" status of a particular compiler. [*Note:* The fallback value call also be provided at the point of the metafunction invocation; see the *Expression semantics* section for details — *end note*]

### Header

```
#include <boost/mpl/has_xxx.hpp>
```

### Parameters

| Parameter | Requirement | Description |
|---|---|---|
| `trait` | A legal identifier token | A name of the metafunction to be generated. |
| `name` | A legal identifier token | A name of the member being detected. |
| `default_` | An boolean constant | A fallback value for the deficient compilers. |

### Expression semantics

For any legal C++ identifiers `trait` and `name`, boolean constant expression `c1`, boolean Integral Constant `c2`, and arbitrary type `x`:

```
BOOST_MPL_HAS_XXX_TRAIT_NAMED_DEF(trait, name, c1)
```

**Precondition:** Appears at namespace scope.

**Return type:** None.

**Semantics:** Expands into an equivalent of the following class template definition

```
template< typename X, typename fallback = boost::mpl::bool_<c1> >
struct trait
{
    // unspecified
    // ...
};
```

where `trait` is a boolean Metafunction with the following semantics:

```
typedef trait<x>::type r;
```

**Return type:** Integral Constant.

**Semantics:** If `BOOST_MPL_CFG_NO_HAS_XXX` is defined, `r::value == c1`; otherwise, `r::value == true` if and only if `x` is a class type that has a nested type memeber `x::name`.

```
typedef trait< x,c2 >::type r;
```

**Return type:** Integral Constant.

**Semantics:** If `BOOST_MPL_CFG_NO_HAS_XXX` is defined, `r::value == c2::value`; otherwise, equivalent to

```
typedef trait<x>::type r;
```

**Example**

```
BOOST_MPL_HAS_XXX_TRAIT_NAMED_DEF(has_xxx, xxx, false)

struct test1 {};
struct test2 { void xxx(); };
struct test3 { int xxx; };
struct test4 { static int xxx(); };
struct test5 { template< typename T > struct xxx {}; };
struct test6 { typedef int xxx; };
struct test7 { struct xxx; };
struct test8 { typedef void (*xxx)(); };
struct test9 { typedef void (xxx)(); };

BOOST_MPL_ASSERT_NOT(( has_xxx<test1> ));
BOOST_MPL_ASSERT_NOT(( has_xxx<test2> ));
BOOST_MPL_ASSERT_NOT(( has_xxx<test3> ));
BOOST_MPL_ASSERT_NOT(( has_xxx<test4> ));
BOOST_MPL_ASSERT_NOT(( has_xxx<test5> ));

#if !defined(BOOST_MPL_CFG_NO_HAS_XXX)
BOOST_MPL_ASSERT(( has_xxx<test6> ));
BOOST_MPL_ASSERT(( has_xxx<test7> ));
BOOST_MPL_ASSERT(( has_xxx<test8> ));
BOOST_MPL_ASSERT(( has_xxx<test9> ));
#endif

BOOST_MPL_ASSERT(( has_xxx<test6,true_> ));
BOOST_MPL_ASSERT(( has_xxx<test7,true_> ));
BOOST_MPL_ASSERT(( has_xxx<test8,true_> ));
BOOST_MPL_ASSERT(( has_xxx<test9,true_> ));
```

**See also**

Macros, BOOST_MPL_HAS_XXX_TRAIT_DEF, BOOST_MPL_CFG_NO_HAS_XXX

## 6.3 Configuration

### 6.3.1 BOOST_MPL_CFG_NO_PREPROCESSED_HEADERS

**Synopsis**

```
// #define BOOST_MPL_CFG_NO_PREPROCESSED_HEADERS
```

**Description**

BOOST_MPL_CFG_NO_PREPROCESSED_HEADERS is an boolean configuration macro regulating library's internal use of preprocessed headers. When defined, it instructs the MPL to discard the pre-generated headers found in boost/mpl/aux_/preprocessed directory and use preprocessor metaprogramming techniques to generate the necessary versions of the library components on the fly.

In this implementation of the library, the macro is not defined by default. To change the default configuration, define BOOST_MPL_CFG_NO_PREPROCESSED_HEADERS before including any library header.

**See also**

Macros, Configuration

### 6.3.2 BOOST_MPL_CFG_NO_HAS_XXX

**Synopsis**

```
// #define BOOST_MPL_CFG_NO_HAS_XXX
```

**Description**

BOOST_MPL_CFG_NO_HAS_XXX is an boolean configuration macro signaling availability of the BOOST_MPL_HAS_XXX_TRAIT_DEF / BOOST_MPL_HAS_XXX_TRAIT_NAMED_DEF introspection macros' functionality on a particular compiler.

**See also**

Macros, Configuration, BOOST_MPL_HAS_XXX_TRAIT_DEF, BOOST_MPL_HAS_XXX_TRAIT_NAMED_DEF

### 6.3.3 BOOST_MPL_LIMIT_METAFUNCTION_ARITY

**Synopsis**

```
#if !defined(BOOST_MPL_LIMIT_METAFUNCTION_ARITY)
#   define BOOST_MPL_LIMIT_METAFUNCTION_ARITY \
        implementation-defined integral constant \
/**/
#endif
```

**Description**

BOOST_MPL_LIMIT_METAFUNCTION_ARITY is an overridable configuration macro regulating the maximum supported arity of metafunctions and metafunction classes. In this implementation of the library, BOOST_MPL_LIMIT_METAFUNCTION_ARITY has a default value of 5. To override the default limit, define BOOST_MPL_LIMIT_METAFUNCTION_ARITY to the desired maximum arity before including any library header. [*Note:* Overriding will take effect *only* if the library is configured not to use preprocessed headers. See BOOST_MPL_CFG_NO_PREPROCESSED_HEADERS for more information. — *end note*]

**Example**

```
#define BOOST_MPL_CFG_NO_PREPROCESSED_HEADERS
#define BOOST_MPL_LIMIT_METAFUNCTION_ARITY 2
#include <boost/mpl/apply.hpp>

using namespace boost::mpl;

template< typename T1, typename T2 > struct second
{
    typedef T2 type;
};

template< typename T1, typename T2, typename T3 > struct third
```

```
{
    typedef T3 type;
};

typedef apply< second<_1,_2_>,int,long >::type r1;
// typedef apply< third<_1,_2_,_3>,int,long,float >::type r2; // error!
```

**See also**

Macros, Configuration, BOOST_MPL_CFG_NO_PREPROCESSED_HEADERS

### 6.3.4 BOOST_MPL_LIMIT_VECTOR_SIZE

**Synopsis**

```
#if !defined(BOOST_MPL_LIMIT_VECTOR_SIZE)
#   define BOOST_MPL_LIMIT_VECTOR_SIZE \
        implementation-defined integral constant \
/**/
#endif
```

**Description**

BOOST_MPL_LIMIT_VECTOR_SIZE is an overridable configuration macro regulating the maximum arity of the vector's and vector_c's variadic forms. In this implementation of the library, BOOST_MPL_LIMIT_VECTOR_SIZE has a default value of 20. To override the default limit, define BOOST_MPL_LIMIT_VECTOR_SIZE to the desired maximum arity rounded up to the nearest multiple of ten before including any library header. [*Note:* Overriding will take effect *only* if the library is configured not to use preprocessed headers. See BOOST_MPL_CFG_NO_PREPROCESSED_HEADERS for more information. — *end note*]

**Example**

```
#define BOOST_MPL_CFG_NO_PREPROCESSED_HEADERS
#define BOOST_MPL_LIMIT_VECTOR_SIZE 10
#include <boost/mpl/vector.hpp>

using namespace boost::mpl;

typedef vector_c<int,1> v_1;
typedef vector_c<int,1,2,3,4,5,6,7,8,9,10> v_10;
// typedef vector_c<int,1,2,3,4,5,6,7,8,9,10,11> v_11; // error!
```

**See also**

Configuration, BOOST_MPL_CFG_NO_PREPROCESSED_HEADERS, BOOST_MPL_LIMIT_LIST_SIZE

### 6.3.5 BOOST_MPL_LIMIT_LIST_SIZE

**Synopsis**

```
#if !defined(BOOST_MPL_LIMIT_LIST_SIZE)
```

```
#   define BOOST_MPL_LIMIT_LIST_SIZE \
        implementation-defined integral constant \
/**/
#endif
```

**Description**

BOOST_MPL_LIMIT_LIST_SIZE is an overridable configuration macro regulating the maximum arity of the list's and list_c's variadic forms. In this implementation of the library, BOOST_MPL_LIMIT_LIST_SIZE has a default value of 20. To override the default limit, define BOOST_MPL_LIMIT_LIST_SIZE to the desired maximum arity rounded up to the nearest multiple of ten before including any library header. [*Note:* Overriding will take effect *only* if the library is configured not to use preprocessed headers. See BOOST_MPL_CFG_NO_PREPROCESSED_HEADERS for more information. — *end note*]

**Example**

```
#define BOOST_MPL_CFG_NO_PREPROCESSED_HEADERS
#define BOOST_MPL_LIMIT_LIST_SIZE 10
#include <boost/mpl/list.hpp>

using namespace boost::mpl;

typedef list_c<int,1> l_1;
typedef list_c<int,1,2,3,4,5,6,7,8,9,10> l_10;
// typedef list_c<int,1,2,3,4,5,6,7,8,9,10,11> l_11; // error!
```

**See also**

Configuration, BOOST_MPL_CFG_NO_PREPROCESSED_HEADERS, BOOST_MPL_LIMIT_VECTOR_SIZE

### 6.3.6 BOOST_MPL_LIMIT_SET_SIZE

**Synopsis**

```
#if !defined(BOOST_MPL_LIMIT_SET_SIZE)
#   define BOOST_MPL_LIMIT_SET_SIZE \
        implementation-defined integral constant \
/**/
#endif
```

**Description**

BOOST_MPL_LIMIT_SET_SIZE is an overridable configuration macro regulating the maximum arity of the set's and set_c's variadic forms. In this implementation of the library, BOOST_MPL_LIMIT_SET_SIZE has a default value of 20. To override the default limit, define BOOST_MPL_LIMIT_SET_SIZE to the desired maximum arity rounded up to the nearest multiple of ten before including any library header. [*Note:* Overriding will take effect *only* if the library is configured not to use preprocessed headers. See BOOST_MPL_CFG_NO_PREPROCESSED_HEADERS for more information. — *end note*]

### Example

```
#define BOOST_MPL_CFG_NO_PREPROCESSED_HEADERS
#define BOOST_MPL_LIMIT_SET_SIZE 10
#include <boost/mpl/set.hpp>

using namespace boost::mpl;

typedef set_c<int,1> s_1;
typedef set_c<int,1,2,3,4,5,6,7,8,9,10> s_10;
// typedef set_c<int,1,2,3,4,5,6,7,8,9,10,11> s_11; // error!
```

### See also

Configuration, BOOST_MPL_CFG_NO_PREPROCESSED_HEADERS, BOOST_MPL_LIMIT_MAP_SIZE

## 6.3.7 BOOST_MPL_LIMIT_MAP_SIZE

### Synopsis

```
#if !defined(BOOST_MPL_LIMIT_MAP_SIZE)
#   define BOOST_MPL_LIMIT_MAP_SIZE \
        implementation-defined integral constant \
/**/
#endif
```

### Description

BOOST_MPL_LIMIT_MAP_SIZE is an overridable configuration macro regulating the maximum arity of the map's variadic form. In this implementation of the library, BOOST_MPL_LIMIT_MAP_SIZE has a default value of 20. To override the default limit, define BOOST_MPL_LIMIT_MAP_SIZE to the desired maximum arity rounded up to the nearest multiple of ten before including any library header. [*Note:* Overriding will take effect *only* if the library is configured not to use preprocessed headers. See BOOST_MPL_CFG_NO_PREPROCESSED_HEADERS for more information. — *end note*]

### Example

```
#define BOOST_MPL_CFG_NO_PREPROCESSED_HEADERS
#define BOOST_MPL_LIMIT_MAP_SIZE 10
#include <boost/mpl/map.hpp>
#include <boost/mpl/pair.hpp>
#include <boost/mpl/int.hpp>

using namespace boost::mpl;

template< int i > struct ints : pair< int_<i>,int_<i> > {};

typedef map< ints<1> > m_1;
typedef map< ints<1>, ints<2>, ints<3>, ints<4>, ints<5>
    ints<6>, ints<7>, ints<8>, ints<9>, ints<10> > m_10;

// typedef map< ints<1>, ints<2>, ints<3>, ints<4>, ints<5>
//     ints<6>, ints<7>, ints<8>, ints<9>, ints<10>, ints<11> > m_11; // error!
```

**See also**

Configuration, BOOST_MPL_CFG_NO_PREPROCESSED_HEADERS, BOOST_MPL_LIMIT_SET_SIZE

### 6.3.8 BOOST_MPL_LIMIT_UNROLLING

**Synopsis**

```
#if !defined(BOOST_MPL_LIMIT_UNROLLING)
#   define BOOST_MPL_LIMIT_UNROLLING \
        implementation-defined integral constant \
/**/
#endif
```

**Description**

BOOST_MPL_LIMIT_UNROLLING is an overridable configuration macro regulating the unrolling depth of the library's iteration algorithms. In this implementation of the library, BOOST_MPL_LIMIT_UNROLLING has a default value of 4. To override the default, define BOOST_MPL_LIMIT_UNROLLING to the desired value before including any library header. [*Note:* Overriding will take effect *only* if the library is configured not to use preprocessed headers. See BOOST_MPL_CFG_NO_PREPROCESSED_HEADERS for more information. — *end note*]

**Example**

Except for overall library performace, overriding the BOOST_MPL_LIMIT_UNROLLING's default value has no user-observable effects.

**See also**

Configuration, BOOST_MPL_CFG_NO_PREPROCESSED_HEADERS

## 6.4 Broken Compiler Workarounds

### 6.4.1 BOOST_MPL_AUX_LAMBDA_SUPPORT

**Synopsis**

```
#define BOOST_MPL_AUX_LAMBDA_SUPPORT(arity, fun, params) \
    unspecified token sequence \
/**/
```

**Description**

Enables metafunction fun for the use in Lambda Expressions on compilers that don't support partial template specialization or/and template template parameters. Expands to nothing on conforming compilers.

**Header**

```
#include <boost/mpl/aux_/lambda_support.hpp>
```

**Parameters**

| Parameter | Requirement | Description |
|---|---|---|
| `arity` | An integral constant | The metafunction's arity, i.e. the number of its template parameters, including the defaults. |
| `fun` | A legal identifier token | The metafunction's name. |
| `params` | A PP-tuple | A tuple of the metafunction's parameter names, in their original order, including the defaults. |

### Expression semantics

For any integral constant `n`, a Metafunction `fun`, and arbitrary types `A1`,... `An`:

```
template< typename A1,... typename An > struct fun
{
    // ...

    BOOST_MPL_AUX_LAMBDA_SUPPORT(n, fun, (A1,...An))
};
```

**Precondition:** Appears in `fun`'s scope, immediately followed by the scope-closing bracket (`}`).

**Return type:** None.

**Semantics:** Expands to nothing and has no effect on conforming compilers. On compilers that don't support partial template specialization or/and template template parameters expands to an unspecified token sequence enabling `fun` to participate in Lambda Expressions with the semantics described in this manual.

### Example

```
template< typename T, typename U = int > struct f
{
    typedef T type[sizeof(U)];

    BOOST_MPL_AUX_LAMBDA_SUPPORT(2, f, (T,U))
};

typedef apply1< f<char,_1>,long >::type r;
BOOST_MPL_ASSERT(( is_same< r, char[sizeof(long)] > ));
```

### See also

Macros, Metafunctions, Lambda Expression

# Chapter 7 Terminology

**Overloaded name** Overloaded name is a term used in this reference documentation to designate a metafunction providing more than one public interface. In reality, class template overloading is nonexistent and the referenced functionality is implemented by other, unspecified, means.

**Concept-identical** A sequence `s1` is said to be concept-identical to a sequence `s2` if `s1` and `s2` model the exact same set of concepts.

**Bind expression** A bind expression is simply that — an instantiation of one of the `bind` class templates. For instance, these are all bind expressions:

```
bind< quote3<if_>, _1,int,long >
bind< _1, bind< plus<>, int_<5>, _2> >
bind< times<>, int_<2>, int_<2> >
```

and these are not:

```
if_< _1, bind< plus<>, int_<5>, _2>, _2 >
protect< bind< quote3<if_>, _1,int,long > >
_2
```

# Chapter 8 Categorized Index

## 8.1 Concepts



## 8.2 Components

— iter_fold
— iterator_category
— iterator_range
— joint_view
— key_type
— lambda
— less
— less_equal
— list
— list_c
— long_
— lower_bound
— map
— max
— max_element
— min
— min_element
— minus
— modulus
— negate
— next
— not_
— not_equal_to
— numeric_cast
— or_
— order
— pair
— partition
— plus
— pop_back
— pop_front
— prior
— protect
— push_back
— push_front

— upper_bound

— value_type

— vector

— vector_c

— void_

— zip_view

# Chapter 9 Acknowledgements

The format and language of this reference documentation has been greatly influenced by the SGI's Standard Template Library Programmer's Guide.

# Bibliography


[n1550] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1550.htm

[PRE] Vesa Karvonen, Paul Mensonides, The Boost Preprocessor Metaprogramming library

[Ve03] Vesa Karvonen, *The Order Programming Language*, 2003.