The MPL Reference Manual
# Contents

## 1 Sequences

1.1 Concepts

1.1.1 Forward Sequence

1.1.2 Bidirectional Sequence

1.1.3 Random Access Sequence

1.1.4 Extensible Sequence

1.1.5 Front Extensible Sequence

1.1.6 Back Extensible Sequence

1.1.7 Associative Sequence

1.1.8 Extensible Associative Sequence

1.1.9 Integral Sequence Wrapper

1.1.10 Variadic Sequence

1.2 Classes

1.2.1 vector

1.2.2 list

1.2.3 deque

1.2.4 set

1.2.5 map

1.2.6 range_c

1.2.7 vector_c

1.2.8 list_c

1.2.9 set_c

1.3 Views

1.3.1 empty_sequence

1.3.2 filter_view

1.3.3 iterator_range

1.3.4 joint_view

1.3.5 single_view

1.3.6 transform_view

1.3.7 zip_view

1.4 Intrinsic Metafunctions

1.4.1 at

1.4.2 at_c

1.4.3 back

1.4.4 begin

1.4.5 clear

1.4.6 empty

1.4.7 end
2 Iterators
2.1 Concepts .................................................. 69
  2.1.1 Forward Iterator ................................. 69
  2.1.2 Bidirectional Iterator ......................... 70
  2.1.3 Random Access Iterator ...................... 71
2.2 Iterator Metafunctions ................................. 72
  2.2.1 advance ............................................. 72
  2.2.2 distance ............................................ 74
  2.2.3 next ............................................... 75
  2.2.4 prior .............................................. 76
  2.2.5 deref ............................................. 77
  2.2.6 iterator_category .............................. 78

3 Algorithms ................................................. 81
3.1 Concepts ................................................ 81
  3.1.1 Inserter .......................................... 81
  3.1.2 Reversible Algorithm ......................... 82
3.2 Inserters ................................................. 84
  3.2.1 back_inserter ................................... 84
  3.2.2 front_inserter .................................. 85
  3.2.3 inserter .......................................... 86
3.3 Iteration Algorithms .................................. 87
  3.3.1 fold ............................................... 88
  3.3.2 iter_fold ....................................... 89
  3.3.3 reverse_fold .................................... 90
  3.3.4 reverse_iter_fold ................................ 92
  3.3.5 accumulate ...................................... 94
3.4 Querying Algorithms .................................. 95
  3.4.1 find .............................................. 95
  3.4.2 find_if .......................................... 96
  3.4.3 contains ........................................ 97
  3.4.4 count ............................................ 98
  3.4.5 count_if ........................................ 99
  3.4.6 lower_bound .................................... 100
  3.4.7 upper_bound .................................... 101
  3.4.8 min_element ................................... 103

Revision Date: 15th November 2004
4 Metafunctions

4.1 Concepts ........................................ 141
  4.1.1 Metafunction ................................ 141
  4.1.2 Metafunction Class .......................... 142
  4.1.3 Lambda Expression ............................ 143
  4.1.4 Placeholder Expression ...................... 144
  4.1.5 Tag Dispatched Metafunction ............... 144
  4.1.6 Numeric Metafunction ....................... 147
  4.1.7 Trivial Metafunction ....................... 148

4.2 Type Selection .................................. 149
  4.2.1 if_ ........................................... 149
  4.2.2 if_c .......................................... 151
  4.2.3 eval_if ...................................... 152
  4.2.4 eval_if_c .................................... 153

4.3 Invocation ....................................... 154
  4.3.1 apply ........................................ 154
  4.3.2 apply_wrap .................................. 156
  4.3.3 unpack_args .................................. 158

4.4 Composition and Argument Binding ............. 159
  4.4.1 Placeholders ................................. 159
  4.4.2 lambda ...................................... 160
  4.4.3 bind ......................................... 161
  4.4.4 quote ........................................ 164
  4.4.5 arg .......................................... 166
  4.4.6 protect ...................................... 167

4.5 Arithmetic Operations ............................ 169
  4.5.1 plus ......................................... 169

Revision Date: 15th November 2004
### CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5.2 minus</td>
<td>170</td>
</tr>
<tr>
<td>4.5.3 times</td>
<td>172</td>
</tr>
<tr>
<td>4.5.4 divides</td>
<td>173</td>
</tr>
<tr>
<td>4.5.5 modulus</td>
<td>175</td>
</tr>
<tr>
<td>4.5.6 negate</td>
<td>176</td>
</tr>
<tr>
<td>4.6 Comparisons</td>
<td>177</td>
</tr>
<tr>
<td>4.6.1 less</td>
<td>177</td>
</tr>
<tr>
<td>4.6.2 less_equal</td>
<td>179</td>
</tr>
<tr>
<td>4.6.3 greater</td>
<td>180</td>
</tr>
<tr>
<td>4.6.4 greater_equal</td>
<td>181</td>
</tr>
<tr>
<td>4.6.5 equal_to</td>
<td>183</td>
</tr>
<tr>
<td>4.6.6 not_equal_to</td>
<td>184</td>
</tr>
<tr>
<td>4.7 Logical Operations</td>
<td>185</td>
</tr>
<tr>
<td>4.7.1 and_</td>
<td>185</td>
</tr>
<tr>
<td>4.7.2 or_</td>
<td>186</td>
</tr>
<tr>
<td>4.7.3 not_</td>
<td>188</td>
</tr>
<tr>
<td>4.8 Bitwise Operations</td>
<td>189</td>
</tr>
<tr>
<td>4.8.1 bitand</td>
<td>189</td>
</tr>
<tr>
<td>4.8.2 bitor</td>
<td>190</td>
</tr>
<tr>
<td>4.8.3 bitxor</td>
<td>192</td>
</tr>
<tr>
<td>4.8.4 shift_left</td>
<td>194</td>
</tr>
<tr>
<td>4.8.5 shift_right</td>
<td>195</td>
</tr>
<tr>
<td>4.9 Trivial</td>
<td>197</td>
</tr>
<tr>
<td>4.9.1 Trivial Metafunctions Summary</td>
<td>197</td>
</tr>
<tr>
<td>4.10 Miscellaneous</td>
<td>197</td>
</tr>
<tr>
<td>4.10.1 identity</td>
<td>197</td>
</tr>
<tr>
<td>4.10.2 always</td>
<td>198</td>
</tr>
<tr>
<td>4.10.3 inherit</td>
<td>199</td>
</tr>
<tr>
<td>4.10.4 inherit_linearly</td>
<td>202</td>
</tr>
<tr>
<td>4.10.5 numeric_cast</td>
<td>203</td>
</tr>
<tr>
<td>4.10.6 min</td>
<td>205</td>
</tr>
<tr>
<td>4.10.7 max</td>
<td>206</td>
</tr>
<tr>
<td>4.10.8 sizeof_</td>
<td>207</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 Data Types</td>
<td>209</td>
</tr>
<tr>
<td>5.1 Concepts</td>
<td>209</td>
</tr>
<tr>
<td>5.1.1 Integral Constant</td>
<td>209</td>
</tr>
<tr>
<td>5.2 Numeric</td>
<td>210</td>
</tr>
<tr>
<td>5.2.1 bool_</td>
<td>210</td>
</tr>
<tr>
<td>5.2.2 int_</td>
<td>211</td>
</tr>
<tr>
<td>5.2.3 long_</td>
<td>212</td>
</tr>
<tr>
<td>5.2.4 size_t</td>
<td>213</td>
</tr>
<tr>
<td>5.2.5 integral_c</td>
<td>214</td>
</tr>
<tr>
<td>5.3 Miscellaneous</td>
<td>216</td>
</tr>
<tr>
<td>5.3.1 pair</td>
<td>216</td>
</tr>
<tr>
<td>5.3.2 empty_base</td>
<td>217</td>
</tr>
<tr>
<td>5.3.3 void_</td>
<td>218</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 Macros</td>
<td>219</td>
</tr>
<tr>
<td>6.1 Asserts</td>
<td>219</td>
</tr>
<tr>
<td>6.1.1 BOOST_MPL_ASSERT</td>
<td>219</td>
</tr>
<tr>
<td>6.1.2 BOOST_MPL_ASSERT_MSG</td>
<td>220</td>
</tr>
<tr>
<td>6.1.3 BOOST_MPL_ASSERT_NOT</td>
<td>222</td>
</tr>
</tbody>
</table>

Revision Date: 15th November 2004
6.1.4  BOOST_MPL_ASSERT_RELATION ........................................ 223
6.2  Introspection ................................................................. 224
  6.2.1  BOOST_MPL_HAS_XXX_TRAIT_DEF .................................... 224
  6.2.2  BOOST_MPL_HAS_XXX_TRAIT_NAMED_DEF ............................ 225
6.3  Configuration ................................................................. 227
  6.3.1  BOOST_MPL_CFG_NO_PREPROCESSED_HEADERS ........................ 227
  6.3.2  BOOST_MPL_CFG_NO_HAS_XXX ........................................ 228
  6.3.3  BOOST_MPL_LIMIT_METAFUNCTION_ARITY ............................ 228
  6.3.4  BOOST_MPL_LIMIT_VECTOR_SIZE ..................................... 229
  6.3.5  BOOST_MPL_LIMIT_LIST_SIZE ....................................... 229
  6.3.6  BOOST_MPL_LIMIT_SET_SIZE ........................................ 230
  6.3.7  BOOST_MPL_LIMIT_MAP_SIZE ....................................... 231
  6.3.8  BOOST_MPL_LIMIT_UNROLLING ....................................... 232
6.4  Broken Compiler Workarounds ............................................ 232
  6.4.1  BOOST_MPL_AUX_LAMBDA_SUPPORT .................................. 232

7  Terminology ................................................................. 235

8  Categorized Index ........................................................... 237
  8.1  Concepts ................................................................. 237
  8.2  Components .............................................................. 237

9  Acknowledgements ........................................................ 243

Bibliography ................................................................. 245
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 capabilities.

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 a the following expressions must be valid:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin&lt;a&gt;::type</td>
<td>Forward Iterator</td>
<td>Amortized constant time</td>
</tr>
<tr>
<td>end&lt;a&gt;::type</td>
<td>Forward Iterator</td>
<td>Amortized constant time</td>
</tr>
<tr>
<td>size&lt;a&gt;::type</td>
<td>Integral Constant</td>
<td>Unspecified</td>
</tr>
<tr>
<td>empty&lt;a&gt;::type</td>
<td>Boolean Integral Constant</td>
<td>Constant time</td>
</tr>
<tr>
<td>front&lt;a&gt;::type</td>
<td>Any type</td>
<td>Amortized constant time</td>
</tr>
</tbody>
</table>
1.1 Concepts

Expression semantics

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin&lt;s&gt;::type</td>
<td>An iterator to the first element of the sequence; see begin.</td>
</tr>
<tr>
<td>end&lt;s&gt;::type</td>
<td>A past-the-end iterator to the sequence; see end.</td>
</tr>
<tr>
<td>size&lt;s&gt;::type</td>
<td>The size of the sequence; see size.</td>
</tr>
<tr>
<td>empty&lt;s&gt;::type</td>
<td>A boolean Integral Constant c such that c::value == true if and only if the sequence is empty; see empty.</td>
</tr>
<tr>
<td>front&lt;s&gt;::type</td>
<td>The first element in the sequence; see front.</td>
</tr>
</tbody>
</table>

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:

Revision Date: 15th November 2004
### Expression semantics

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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>back&lt;s&gt;::type</code></td>
<td>The last element in the sequence; see <code>back</code>.</td>
</tr>
</tbody>
</table>

### 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:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>begin&lt;s&gt;::type</code></td>
<td>Random Access Iterator</td>
<td>Amortized constant time</td>
</tr>
<tr>
<td><code>end&lt;s&gt;::type</code></td>
<td>Random Access Iterator</td>
<td>Amortized constant time</td>
</tr>
<tr>
<td><code>at&lt;s,n&gt;::type</code></td>
<td>Any type</td>
<td>Amortized constant time</td>
</tr>
</tbody>
</table>

**Expression semantics**

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

---

Revision Date: 15th November 2004
1.1 Concepts

Expression | Semantics
---|---
\(\text{at} \langle s, n \rangle::\text{type} \) | The \(n\)th element from the beginning of the sequence; see \(\text{at}\).

Models

- \text{vector}
- \text{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 \(\text{pos}\) and \(\text{last}\), Forward Sequence \(r\), and any type \(x\), the following expressions must be valid:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{insert} \langle s, \text{pos}, x \rangle::\text{type} )</td>
<td>Extensible Sequence</td>
<td>Unspecified</td>
</tr>
<tr>
<td>(\text{insert_range} \langle s, \text{pos}, r \rangle::\text{type} )</td>
<td>Extensible Sequence</td>
<td>Unspecified</td>
</tr>
<tr>
<td>(\text{erase} \langle s, \text{pos} \rangle::\text{type} )</td>
<td>Extensible Sequence</td>
<td>Unspecified</td>
</tr>
<tr>
<td>(\text{erase} \langle s, \text{pos}, \text{last} \rangle::\text{type} )</td>
<td>Extensible Sequence</td>
<td>Unspecified</td>
</tr>
<tr>
<td>(\text{clear} \langle s \rangle::\text{type} )</td>
<td>Extensible Sequence</td>
<td>Constant time</td>
</tr>
</tbody>
</table>

Expression semantics

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{insert} \langle s, \text{pos}, x \rangle::\text{type} )</td>
<td>A new sequence, concept-identical to (s), of the following elements: ([\text{begin} \langle s \rangle::\text{type}, \text{pos}, x, \text{pos}, \text{end} \langle s \rangle::\text{type}]); see (\text{insert}).</td>
</tr>
<tr>
<td>(\text{insert_range} \langle s, \text{pos}, r \rangle::\text{type} )</td>
<td>A new sequence, concept-identical to (s), of the following elements: ([\text{begin} \langle s \rangle::\text{type}, \text{pos}, \text{begin} \langle r \rangle::\text{type}, \text{end} \langle r \rangle::\text{type}, \text{pos}, \text{end} \langle s \rangle::\text{type}]); see (\text{insert_range}).</td>
</tr>
<tr>
<td>(\text{erase} \langle s, \text{pos} \rangle::\text{type} )</td>
<td>A new sequence, concept-identical to (s), of the following elements: ([\text{begin} \langle s \rangle::\text{type}, \text{pos}, \text{next} \langle \text{pos} \rangle::\text{type}, \text{end} \langle s \rangle::\text{type}]); see (\text{erase}).</td>
</tr>
<tr>
<td>(\text{erase} \langle s, \text{pos}, \text{last} \rangle::\text{type} )</td>
<td>A new sequence, concept-identical to (s), of the following elements: ([\text{begin} \langle s \rangle::\text{type}, \text{pos}, \text{last}, \text{end} \langle s \rangle::\text{type}]); see (\text{erase}).</td>
</tr>
<tr>
<td>(\text{clear} \langle s \rangle::\text{type} )</td>
<td>An empty sequence concept-identical to (s); see (\text{clear}).</td>
</tr>
</tbody>
</table>

Revision Date: 15th November 2004
1.1 Concepts

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:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>push_front&lt;s,x&gt;::type</td>
<td>Front Extensible Sequence</td>
<td>Amortized constant time</td>
</tr>
<tr>
<td>pop_front&lt;s&gt;::type</td>
<td>Front Extensible Sequence</td>
<td>Amortized constant time</td>
</tr>
<tr>
<td>front&lt;s&gt;::type</td>
<td>Any type</td>
<td>Amortized constant time</td>
</tr>
</tbody>
</table>

Expression semantics

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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>push_front&lt;s,x&gt;::type</td>
<td>Equivalent to insert&lt;s,begin&lt;s&gt;::type,x&gt;::type; see push_-front.</td>
</tr>
<tr>
<td>pop_front&lt;v&gt;::type</td>
<td>Equivalent to erase&lt;s,begin&lt;s&gt;::type&gt;::type; see pop_front.</td>
</tr>
<tr>
<td>front&lt;s&gt;::type</td>
<td>The first element in the sequence; see front.</td>
</tr>
</tbody>
</table>

Models

- vector
- list

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

Revision Date: 15th November 2004
1.1 Concepts

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 the following must be met:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>push_back&lt;s,x&gt;::type</td>
<td>Back Extensible Sequence</td>
<td>Amortized constant time</td>
</tr>
<tr>
<td>pop_back&lt;s&gt;::type</td>
<td>Back Extensible Sequence</td>
<td>Amortized constant time</td>
</tr>
<tr>
<td>back&lt;s&gt;::type</td>
<td>Any type</td>
<td>Amortized constant time</td>
</tr>
</tbody>
</table>

Expression semantics

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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>push_back&lt;s,x&gt;::type</td>
<td>Equivalent to insert&lt;s,end&lt;s&gt;::type,x&gt;::type; see push_back.</td>
</tr>
<tr>
<td>pop_back&lt;v&gt;::type</td>
<td>Equivalent to erase&lt;s,end&lt;s&gt;::type&gt;::type; see pop_back.</td>
</tr>
<tr>
<td>back&lt;s&gt;::type</td>
<td>The last element in the sequence; see back.</td>
</tr>
</tbody>
</table>

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.

Revision Date: 15th November 2004
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 retrieved 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:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>has_key&lt;s,k&gt;::type</td>
<td>Boolean</td>
<td>Amortized constant time</td>
</tr>
<tr>
<td>count&lt;s,k&gt;::type</td>
<td>Integral Constant</td>
<td>Amortized constant time</td>
</tr>
<tr>
<td>order&lt;s,k&gt;::type</td>
<td>Integral Constant or void_</td>
<td>Amortized constant time</td>
</tr>
<tr>
<td>at&lt;s,k&gt;::type</td>
<td>Any type</td>
<td>Amortized constant time</td>
</tr>
<tr>
<td>at&lt;s,k,def&gt;::type</td>
<td>Any type</td>
<td>Amortized constant time</td>
</tr>
<tr>
<td>key_type&lt;s,x&gt;::type</td>
<td>Any type</td>
<td>Amortized constant time</td>
</tr>
<tr>
<td>value_type&lt;s,x&gt;::type</td>
<td>Any type</td>
<td>Amortized constant time</td>
</tr>
</tbody>
</table>

Expression semantics

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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>has_key&lt;s,k&gt;::type</td>
<td>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.</td>
</tr>
<tr>
<td>count&lt;s,k&gt;::type</td>
<td>The number of elements with the key k in s; see count.</td>
</tr>
<tr>
<td>order&lt;s,k&gt;::type</td>
<td>A unique unsigned Integral Constant associated with the key k in the sequence s; see order.</td>
</tr>
<tr>
<td>at&lt;s,k&gt;::type</td>
<td>The first element associated with the key k in the sequence s; see at.</td>
</tr>
<tr>
<td>at&lt;s,k,def&gt;::type</td>
<td>The key part of the element x that would be used to identify x in s; see key_type.</td>
</tr>
<tr>
<td>value_type&lt;s,x&gt;::type</td>
<td>The value part of the element x that would be used for x in s; see value_type.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert&lt;s,x&gt;::type</td>
<td>Extensible Associative Sequence</td>
<td>Amortized constant time</td>
</tr>
<tr>
<td>insert&lt;s,pos,x&gt;::type</td>
<td>Extensible Associative Sequence</td>
<td>Amortized constant time</td>
</tr>
<tr>
<td>erase_key&lt;s,k&gt;::type</td>
<td>Extensible Associative Sequence</td>
<td>Amortized constant time</td>
</tr>
<tr>
<td>erase&lt;s,pos&gt;::type</td>
<td>Extensible Associative Sequence</td>
<td>Amortized constant time</td>
</tr>
<tr>
<td>clear&lt;s&gt;::type</td>
<td>Extensible Associative Sequence</td>
<td>Amortized constant time</td>
</tr>
</tbody>
</table>

Expression semantics

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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert&lt;s,x&gt;::type</td>
<td>Inserts x into s; the resulting sequence r is equivalent to s except that at&lt;r, key_type&lt;s,x&gt;::type&gt;::type is identical to value_type&lt;s,x&gt;::type; see insert.</td>
</tr>
<tr>
<td>insert&lt;s,pos,x&gt;::type</td>
<td>Equivalent to insert&lt;s,x&gt;::type; pos is ignored; see insert.</td>
</tr>
<tr>
<td>erase_key&lt;s,k&gt;::type</td>
<td>Erases elements in s associated with the key k; the resulting sequence r is equivalent to s except that has_key&lt;r,k&gt;::value == false; see erase_key.</td>
</tr>
<tr>
<td>erase&lt;s,pos&gt;::type</td>
<td>Erases the element at a specific position; equivalent to erase_key&lt;s, deref(pos)::type&gt;::type; see erase.</td>
</tr>
<tr>
<td>clear&lt;s&gt;::type</td>
<td>An empty sequence concept-identical to s; see clear.</td>
</tr>
</tbody>
</table>

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 $c_1, c_2, \ldots, c_n$ are integral constants of an integral type T to be stored in the sequence, the wrapper provides us with the following notation:

$$\text{seq}_c\langle T, c_1, c_2, \ldots, c_n \rangle$$

If seq is a Variadic Sequence, numbered wrapper forms are also available:

$$\text{seq}_n\langle T, c_1, c_2, \ldots, c_n \rangle$$

Expression requirements

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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq_c\langle T, c_1, c_2, \ldots, c_n \rangle</td>
<td>Forward Sequence</td>
<td>Amortized constant time.</td>
</tr>
<tr>
<td>seq_c\langle T, c_1, c_2, \ldots, c_n \rangle::type</td>
<td>Forward Sequence</td>
<td>Amortized constant time.</td>
</tr>
<tr>
<td>seq_c\langle T, c_1, c_2, \ldots, c_n \rangle::value_type</td>
<td>An integral type</td>
<td>Amortized constant time.</td>
</tr>
<tr>
<td>seq_n\langle T, c_1, c_2, \ldots, c_n \rangle</td>
<td>Forward Sequence</td>
<td>Amortized constant time.</td>
</tr>
<tr>
<td>seq_n\langle T, c_1, c_2, \ldots, c_n \rangle::type</td>
<td>Forward Sequence</td>
<td>Amortized constant time.</td>
</tr>
<tr>
<td>seq_n\langle T, c_1, c_2, \ldots, c_n \rangle::value_type</td>
<td>An integral type</td>
<td>Amortized constant time.</td>
</tr>
</tbody>
</table>

Expression semantics

```cpp
typedef seq\_c\langle T, c_1, c_2, \ldots, c_n \rangle s;
typedef seq\_n\langle T, c_1, c_2, \ldots, c_n \rangle s;

Semantics: s is a sequence seq of integral constant wrappers integral\_c\langle T, c_1 \rangle, integral\_c\langle T, c_2 \rangle,
\ldots integral\_c\langle T, c_n \rangle.

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

```cpp
typedef seq\_c\langle T, c_1, c_2, \ldots, c_n \rangle::type s;
typedef seq\_n\langle T, c_1, c_2, \ldots, c_n \rangle::type s;

Semantics: s is identical to seqn\langle integral\_c\langle T, c_1 \rangle, integral\_c\langle T, c_2 \rangle, \ldots integral\_c\langle T, c_n \rangle \rangle.
```

```cpp
typedef seq\_c\langle T, c_1, c_2, \ldots, c_n \rangle::value\_type t;
typedef seq\_n\langle T, c_1, c_2, \ldots, c_n \rangle::value\_type t;

Semantics: is\_same\langle t, T \rangle::value == true.
```

Models

- vector\_c
- list\_c
- set\_c

Revision Date: 15th November 2004
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 \( t_1, t_2, \ldots, t_n \), for
any n from 0 up to a preprocessor-configurable limit BOOST_MPL_LIMIT_seq_SIZE, using the following notation:

\[
\text{seq}\langle t_1, t_2, \ldots, t_n \rangle
\]

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

\[
\text{seqn}\langle t_1, t_2, \ldots, t_n \rangle
\]

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.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq\langle t_1, t_2, \ldots, t_n \rangle</td>
<td>Forward Sequence</td>
<td>Amortized constant time</td>
</tr>
<tr>
<td>seq\langle t_1, t_2, \ldots, t_n \rangle::type</td>
<td>Forward Sequence</td>
<td>Amortized constant time</td>
</tr>
<tr>
<td>seqn\langle t_1, t_2, \ldots, t_n \rangle</td>
<td>Forward Sequence</td>
<td>Amortized constant time</td>
</tr>
<tr>
<td>seqn\langle t_1, t_2, \ldots, t_n \rangle::type</td>
<td>Forward Sequence</td>
<td>Amortized constant time</td>
</tr>
</tbody>
</table>

Expression semantics

\[
\text{typedef seq}\langle t_1, t_2, \ldots, t_n \rangle \ s; \\
\text{typedef seqn}\langle t_1, t_2, \ldots, t_n \rangle \ s;
\]

Semantics: \( s \) is a sequence of elements \( t_1, t_2, \ldots, t_n \).

Postcondition: \( \text{size}\langle s \rangle::\text{value} = n \).

\[
\text{typedef seq}\langle t_1, t_2, \ldots, t_n \rangle::\text{type} \ s; \\
\text{typedef seqn}\langle t_1, t_2, \ldots, t_n \rangle::\text{type} \ s;
\]

Semantics: \( s \) is identical to seqn\langle t_1, t_2, \ldots, t_n \rangle.

Postcondition: \( \text{size}\langle s \rangle::\text{value} = n \).

Models

- vector
- list
- map

Revision Date: 15th November 2004
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

<table>
<thead>
<tr>
<th>Sequence form</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variadic</td>
<td><code>#include &lt;boost/mpl/vector.hpp&gt;</code></td>
</tr>
<tr>
<td>Numbered</td>
<td><code>#include &lt;boost/mpl/vector/vector&lt;n&gt;.hpp&gt;</code></td>
</tr>
</tbody>
</table>

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 t₁, t₂,..., tₙ are arbitrary types.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vector&lt;t₁,t₂,...,tₙ&gt;</code></td>
<td>vector of elements t₁,t₂,...,tₙ; see Variadic Sequence.</td>
</tr>
<tr>
<td><code>vector&lt;t₁,t₂,...,tₙ&gt;::type</code></td>
<td>Identical to <code>vector&lt;n&lt;t₁,t₂,...,tₙ&gt;::type</code>; see Variadic Sequence.</td>
</tr>
<tr>
<td><code>begin&lt;v&gt;::type</code></td>
<td>An iterator pointing to the beginning of v; see Random Access Sequence.</td>
</tr>
<tr>
<td><code>end&lt;v&gt;::type</code></td>
<td>An iterator pointing to the end of v; see Random Access Sequence.</td>
</tr>
<tr>
<td><code>size&lt;v&gt;::type</code></td>
<td>The size of v; see Random Access Sequence.</td>
</tr>
<tr>
<td><code>empty&lt;v&gt;::type</code></td>
<td>A boolean Integral Constant c such that c::value == true if and only if the sequence is empty; see Random Access Sequence.</td>
</tr>
</tbody>
</table>

Revision Date: 15th November 2004
1.2 Classes

## Sequences

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

### Example

```cpp
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

<table>
<thead>
<tr>
<th>Sequnce form</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variadic</td>
<td>&quot;#include &lt;boost/mpl/list.hpp&gt;&quot;</td>
</tr>
<tr>
<td>Numbered</td>
<td>&quot;#include &lt;boost/mpl/list/listn.hpp&gt;&quot;</td>
</tr>
</tbody>
</table>

Revision Date: 15th November 2004
21 Sequences 1.2 Classes

Model of

— Variadic Sequence
— Forward Sequence
— Extensible Sequence
— Front Extensible Sequence

Expression semantics

In the following table, \( l \) is a list, \( \text{pos} \) and \( \text{last} \) are iterators into \( l \), \( r \) is a Forward Sequence, and \( t_1, t_2, \ldots, t_n \) and \( x \) are arbitrary types.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>list&lt;( t_1, t_2, \ldots, t_n )&gt;</td>
<td>list of elements ( t_1, t_2, \ldots, t_n ); see Variadic Sequence.</td>
</tr>
<tr>
<td>listn&lt;( t_1, t_2, \ldots, t_n )&gt;</td>
<td>Identical to list&lt;( t_1, t_2, \ldots, t_n )&gt;; see Variadic Sequence.</td>
</tr>
<tr>
<td>list&lt;( t_1, t_2, \ldots, t_n &gt;::\text{type}</td>
<td>Identical to list&lt;( t_1, t_2, \ldots, t_n )&gt;; see Variadic Sequence.</td>
</tr>
<tr>
<td>begin&lt;( l )::\text{type}</td>
<td>An iterator to the beginning of ( l ); see Forward Sequence.</td>
</tr>
<tr>
<td>end&lt;( l )::\text{type}</td>
<td>An iterator to the end of ( l ); see Forward Sequence.</td>
</tr>
<tr>
<td>size&lt;( l )::\text{type}</td>
<td>The size of ( l ); see Forward Sequence.</td>
</tr>
<tr>
<td>empty&lt;( l )::\text{type}</td>
<td>A boolean Integral Constant ( c ) such that ( c::\text{value} == \text{true} ) if and only if ( l ) is empty; see Forward Sequence.</td>
</tr>
<tr>
<td>front&lt;( l )::\text{type}</td>
<td>The first element in ( l ); see Forward Sequence.</td>
</tr>
<tr>
<td>insert&lt;( l, \text{pos}, x )::\text{type}</td>
<td>A new list of following elements: ([\begin{array}{l} \text{begin}&lt;( l )::\text{type}, \text{pos}, x, \text{next}&lt;( \text{pos} )::\text{type} \end{array} ); see Extensible Sequence.</td>
</tr>
<tr>
<td>insert_range&lt;( l, \text{pos}, r )::\text{type}</td>
<td>A new list of following elements: ([\begin{array}{l} \text{begin}&lt;( l )::\text{type}, \text{pos}, \text{begin}&lt;( r )::\text{type}, \text{end}&lt;( r )::\text{type} \end{array} ); see Extensible Sequence.</td>
</tr>
<tr>
<td>erase&lt;( l, \text{pos} )::\text{type}</td>
<td>A new list of following elements: ([\begin{array}{l} \text{begin}&lt;( l )::\text{type}, \text{pos}, \text{next}&lt;( \text{pos} )::\text{type} \end{array} ); see Extensible Sequence.</td>
</tr>
<tr>
<td>erase&lt;( l, \text{pos}, \text{last} )::\text{type}</td>
<td>A new list of following elements: ([\begin{array}{l} \text{begin}&lt;( l )::\text{type}, \text{pos}, \text{last}, \text{end}&lt;( l )::\text{type} \end{array} ); see Extensible Sequence.</td>
</tr>
<tr>
<td>clear&lt;( l )::\text{type}</td>
<td>An empty list; see Extensible Sequence.</td>
</tr>
<tr>
<td>push_front&lt;( l, x )::\text{type}</td>
<td>A new list containing ( x ) as its first element; see Front Extensible Sequence.</td>
</tr>
<tr>
<td>pop_front&lt;( l )::\text{type}</td>
<td>A new list containing all but the first elements of ( l ) in the same order; see Front Extensible Sequence.</td>
</tr>
</tbody>
</table>

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 > ));
1.2 Classes Sequences 22

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

Revision Date: 15th November 2004
Model of

— Variadic Sequence
— Associative Sequence
— Extensible Associative Sequence

Expression semantics

In the following table, \( s \) is an instance of \( \text{set} \), \( \text{pos} \) is an iterator into \( s \), and \( x, k, t_1, t_2, \ldots, t_n \) are arbitrary types.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{set}&lt;t_1, t_2, \ldots, t_n&gt; )</td>
<td>set of elements ( t_1, t_2, \ldots, t_n ); see Variadic Sequence.</td>
</tr>
<tr>
<td>( \text{setn}&lt;t_1, t_2, \ldots, t_n&gt; )</td>
<td>Identical to ( \text{set}&lt;t_1, t_2, \ldots, t_n&gt; ); see Variadic Sequence.</td>
</tr>
<tr>
<td>( \text{begin}&lt;s&gt;::\text{type} )</td>
<td>An iterator pointing to the beginning of ( s ); see Associative Sequence.</td>
</tr>
<tr>
<td>( \text{end}&lt;s&gt;::\text{type} )</td>
<td>An iterator pointing to the end of ( s ); see Associative Sequence.</td>
</tr>
<tr>
<td>( \text{empty}&lt;s&gt;::\text{type} )</td>
<td>A boolean Integral Constant ( c ) such that ( c::\text{value} == \text{true} ) if and only if ( s ) is empty; see Associative Sequence.</td>
</tr>
<tr>
<td>( \text{front}&lt;s&gt;::\text{type} )</td>
<td>The first element in ( s ); see Associative Sequence.</td>
</tr>
<tr>
<td>( \text{has_key}&lt;s,k&gt;::\text{type} )</td>
<td>A boolean Integral Constant ( c ) such that ( c::\text{value} == \text{true} ) if and only if there is one or more elements with the key ( k ) in ( s ); see Associative Sequence.</td>
</tr>
<tr>
<td>( \text{count}&lt;s,k&gt;::\text{type} )</td>
<td>The number of elements with the key ( k ) in ( s ); see Associative Sequence.</td>
</tr>
<tr>
<td>( \text{order}&lt;s,k&gt;::\text{type} )</td>
<td>A unique unsigned Integral Constant associated with the key ( k ) in ( s ); see Associative Sequence.</td>
</tr>
<tr>
<td>( \text{at}&lt;s,k&gt;::\text{type} )</td>
<td>The element associated with the key ( k ) in ( s ); see Associative Sequence.</td>
</tr>
<tr>
<td>( \text{key_type}&lt;s,x&gt;::\text{type} )</td>
<td>Identical to ( x ); see Associative Sequence.</td>
</tr>
<tr>
<td>( \text{value_type}&lt;s,x&gt;::\text{type} )</td>
<td>Identical to ( x ); see Associative Sequence.</td>
</tr>
<tr>
<td>( \text{insert}&lt;s,x&gt;::\text{type} )</td>
<td>A new ( \text{set} ) equivalent to ( s ) except that ( \text{at}&lt;t, \text{key_type}&lt;s,x&gt;::\text{type}&gt;::\text{type} ) is identical to ( \text{value_type}&lt;s,x&gt;::\text{type} ).</td>
</tr>
<tr>
<td>( \text{insert}&lt;s,\text{pos},x&gt;::\text{type} )</td>
<td>Equivalent to ( \text{insert}&lt;s,x&gt;::\text{type} ); ( \text{pos} ) is ignored.</td>
</tr>
<tr>
<td>( \text{erase_key}&lt;s,k&gt;::\text{type} )</td>
<td>A new ( \text{set} ) equivalent to ( s ) except that ( \text{has_key}&lt;t, k&gt;::\text{value} == \text{false} ).</td>
</tr>
<tr>
<td>( \text{erase}&lt;s,\text{pos}&gt;::\text{type} )</td>
<td>Equivalent to ( \text{erase}&lt;s, \text{deref}&lt;\text{pos}&gt;::\text{type}&gt;::\text{type} ).</td>
</tr>
<tr>
<td>( \text{clear}&lt;&gt;::\text{type} )</td>
<td>An empty ( \text{set} ); see ( \text{clear} ).</td>
</tr>
</tbody>
</table>
Example

```cpp
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

<table>
<thead>
<tr>
<th>Sequence form</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variadic</td>
<td>#include &lt;boost/mpl/map.hpp&gt;</td>
</tr>
<tr>
<td>Numbered</td>
<td>#include &lt;boost/mpl/mapn.hpp&gt;</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>map&lt;( p_1,p_2,...,p_n )&gt;</td>
<td>map of elements ( p_1,p_2,...,p_n ); see Variadic Sequence.</td>
</tr>
<tr>
<td>mapn&lt;( p_1,p_2,...,p_n )&gt;</td>
<td>Identical to map&lt;( p_1,p_2,...,p_n )&gt;; see Variadic Sequence.</td>
</tr>
<tr>
<td>begin&lt;m&gt;::type</td>
<td>An iterator pointing to the beginning of m; see Associative Sequence.</td>
</tr>
<tr>
<td>end&lt;m&gt;::type</td>
<td>An iterator pointing to the end of m; see Associative Sequence.</td>
</tr>
<tr>
<td>size&lt;m&gt;::type</td>
<td>The size of m; see Associative Sequence.</td>
</tr>
</tbody>
</table>

Revision Date: 15th November 2004
<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty(&lt;m&gt;&gt;::\text{type}</td>
<td>A boolean Integral Constant c such that c::value == true if and only if m is empty; see Associative Sequence.</td>
</tr>
<tr>
<td>front(&lt;m&gt;&gt;::\text{type}</td>
<td>The first element in m; see Associative Sequence.</td>
</tr>
<tr>
<td>has_key(&lt;m,k&gt;)::\text{type}</td>
<td>Queries the presence of elements with the key k in m; see Associative Sequence.</td>
</tr>
<tr>
<td>count(&lt;m,k&gt;)::\text{type}</td>
<td>The number of elements with the key k in m; see Associative Sequence.</td>
</tr>
<tr>
<td>order(&lt;m,k&gt;)::\text{type}</td>
<td>A unique unsigned Integral Constant associated with the key k in m; see Associative Sequence.</td>
</tr>
<tr>
<td>at(&lt;m,k&gt;)::\text{type}</td>
<td>The element associated with the key k in m; see Associative Sequence.</td>
</tr>
<tr>
<td>at(&lt;m,k,default&gt;)::\text{type}</td>
<td>Identical to x::first; see Associative Sequence.</td>
</tr>
<tr>
<td>key_type(&lt;m,x&gt;)::\text{type}</td>
<td>Identical to x::first; see Associative Sequence.</td>
</tr>
<tr>
<td>value_type(&lt;m,x&gt;)::\text{type}</td>
<td>Identical to x::second; see Associative Sequence.</td>
</tr>
<tr>
<td>insert(&lt;m,x&gt;)::\text{type}</td>
<td>A new map equivalent to m except that at&lt; t, key_type(&lt;m,x&gt;)::\text{type} &gt;::\text{type} is identical to value_type(&lt;m,x&gt;)::\text{type}.</td>
</tr>
<tr>
<td>insert(&lt;m,pos,x&gt;)::\text{type}</td>
<td>Equivalent to insert(&lt;m,x&gt;)::\text{type}; pos is ignored.</td>
</tr>
<tr>
<td>erase_key(&lt;m,k&gt;)::\text{type}</td>
<td>A new map equivalent to m except that has_key(&lt;t,k&gt;)::value == false.</td>
</tr>
<tr>
<td>erase(&lt;pos&gt;)::\text{type}</td>
<td>Equivalent to erase(&lt;m,\ deref(pos)&gt;)::\text{type}.</td>
</tr>
<tr>
<td>clear(&lt;m&gt;)::\text{type}</td>
<td>An empty map; see clear.</td>
</tr>
</tbody>
</table>

Example

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

See also

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

1.2.6 range_c

Synopsis

```cpp
template<
```
typedef 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.

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

Example

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

Revision Date: 15th November 2004
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

<table>
<thead>
<tr>
<th>Sequence form</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variadic</td>
<td>#include &lt;boost/mpl/vector_c.hpp&gt;</td>
</tr>
<tr>
<td>Numbered</td>
<td>#include &lt;boost/mpl/vector/vectorn_c.hpp&gt;</td>
</tr>
</tbody>
</table>

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.


### 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**

<table>
<thead>
<tr>
<th>Sequence form</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variadic</td>
<td>#include &lt;boost/mpl/list_c.hpp&gt;</td>
</tr>
<tr>
<td>Numbered</td>
<td>#include &lt;boost/mpl/list/listn_c.hpp&gt;</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>list_c&lt;T, c_1, c_2, ..., c_n&gt;</code></td>
<td>A list of integral constant wrappers <code>integral_c&lt;T, c_1&gt;</code>, <code>integral_c&lt;T, c_2&gt;</code>, ..., <code>integral_c&lt;T, c_n&gt;</code>; see Integral Sequence Wrapper.</td>
</tr>
<tr>
<td><code>listn_c&lt;T, c_1, c_2, ..., c_n&gt;</code></td>
<td>Identical to <code>listn&lt;integral_c&lt;T, c_1&gt;, integral_c&lt;T, c_2&gt;, ..., integral_c&lt;T, c_n&gt;&gt;</code>; see Integral Sequence Wrapper.</td>
</tr>
<tr>
<td><code>list_c&lt;T, c_1, c_2, ..., c_n&gt;::type</code></td>
<td>Identical to <code>T</code>; see Integral Sequence Wrapper.</td>
</tr>
</tbody>
</table>

### Example

```cpp
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

<table>
<thead>
<tr>
<th>Sequence form</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variadic</td>
<td>#include <code>&lt;boost/mpl/set_c.hpp&gt;</code></td>
</tr>
<tr>
<td>Numbered</td>
<td>#include <code>&lt;boost/mpl/set/setn_c.hpp&gt;</code></td>
</tr>
</tbody>
</table>

#### 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`.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>set_c&lt;T, c_1, c_2, ..., c_n&gt;</code></td>
<td>A set of integral constant wrappers <code>integral_c&lt;T, c_1&gt;</code>, <code>integral_c&lt;T, c_2&gt;</code>, ..., <code>integral_c&lt;T, c_n&gt;</code>; see Integral Sequence Wrapper.</td>
</tr>
<tr>
<td><code>setn_c&lt;T, c_1, c_2, ..., c_n&gt;</code></td>
<td></td>
</tr>
</tbody>
</table>
1.3 Views

Sequences

1.3.1 empty_sequence

**Synopsis**

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

**Description**

Represents a sequence containing no elements.

**Header**

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

---

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**

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

**Description**

Represents a sequence containing no elements.

**Header**

```cpp
#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.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty_sequence</td>
<td>An empty Random Access Sequence.</td>
</tr>
<tr>
<td>size$s$::type</td>
<td>size$s$::value == 0; see Random Access Sequence.</td>
</tr>
</tbody>
</table>

Example

```cpp
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

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

Description

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

Header

```cpp
#include <boost/mpl/filter_view.hpp>
```
Model of
— Forward Sequence

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Forward Sequence</td>
<td>A sequence to wrap.</td>
</tr>
<tr>
<td>Pred</td>
<td>Unary Lambda Expression</td>
<td>A filtering predicate.</td>
</tr>
</tbody>
</table>

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, \( \text{pred} \) is an unary Lambda Expression.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{filter} _\text{view}\langle s, \text{pred} \rangle )</td>
<td>A lazy Forward Sequence sequence of all the elements in the range ([\text{begin}\langle s \rangle::\text{type}, \text{end}\langle s \rangle::\text{type}]) that satisfy the predicate ( \text{pred} ).</td>
</tr>
<tr>
<td>( \text{size} \langle v \rangle::\text{type} )</td>
<td>The size of ( v ); ( \text{size} \langle v \rangle::\text{value} == \text{count}_\langle s, \text{pred} \rangle::\text{value} ); linear complexity; see Forward Sequence.</td>
</tr>
</tbody>
</table>

Example

Find the largest floating type in a sequence.

```cpp
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;
:numel<
    transform_view< filter_view< types, boost::is_float<_> >, size_of<_> > >
>
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

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

Revision Date: 15th November 2004
Description
A view into subset of sequence elements identified by a pair of iterators.

Header

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

Model of

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

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>First, Last</td>
<td>Forward Iterator</td>
<td>Iterators identifying the view’s boundaries.</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>iterator_range&lt;first,last&gt;</td>
<td>A lazy sequence all the elements in the range <code>[first, last)</code>.</td>
</tr>
<tr>
<td>iterator_range&lt;first,last&gt;::type</td>
<td></td>
</tr>
</tbody>
</table>

Example

```cpp
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

```cpp
    template<
```
```cpp
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**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence1, Sequence2</td>
<td>Forward Sequence</td>
<td>Sequences to create a view on.</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>joint_view&lt;s1,s2&gt;</td>
<td>A lazy Forward Sequence of all the elements in the ranges [begin&lt;s1&gt;::type, end&lt;s1&gt;::type), [begin&lt;s2&gt;::type, end&lt;s2&gt;::type).</td>
</tr>
<tr>
<td>joint_view&lt;s1,s2&gt;::type</td>
<td>The size of v; size&lt;v&gt;::value == size&lt;s1&gt;::value + size&lt;s2&gt;::value; linear complexity; see Forward Sequence.</td>
</tr>
<tr>
<td>size&lt;v&gt;::type</td>
<td>The size of v; size&lt;v&gt;::value == size&lt;s1&gt;::value + size&lt;s2&gt;::value; linear complexity; see Forward Sequence.</td>
</tr>
</tbody>
</table>

**Example**

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

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

**Revision Date:** 15th November 2004
See also
Sequences, Views, filter_view, transform_view, zip_view, iterator_range

1.3.5 single_view

Synopsis

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

Description

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

Header

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

Model of

— Random Access Sequence

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>Any type</td>
<td>The type to be wrapped in a sequence.</td>
</tr>
</tbody>
</table>

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 \(\text{single_view}\), \(x\) is an arbitrary type.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{single_view}&lt;x&gt;)</td>
<td>A single-element Random Access Sequence (v) such that (\text{front}&lt;v&gt;::\text{type}) is identical to (x).</td>
</tr>
<tr>
<td>(\text{single_view}&lt;v&gt;::\text{type})</td>
<td>The size of (v); (\text{size}&lt;v&gt;::\text{value} == 1); see Random Access Sequence.</td>
</tr>
</tbody>
</table>

Example

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

Revision Date: 15th November 2004
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

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

Description

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

Header

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

Model of

— Forward Sequence

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Forward Sequence</td>
<td>A sequence to wrap.</td>
</tr>
<tr>
<td>F</td>
<td>Unary Lambda Expression</td>
<td>A transformation.</td>
</tr>
</tbody>
</table>

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.
37 Sequences 1.3 Views

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>transform_view&lt;s,f&gt;</td>
<td>A lazy Forward Sequence such that for each ( i ) in the range ([\text{begin} v::\text{type}, \text{end} v::\text{type}]) and each ( j ) in the range ([\text{begin} s::\text{type}, \text{end} s::\text{type}]) ( \text{deref} i::\text{type} ) is identical to ( \text{apply} f, \text{deref} j::\text{type} ).</td>
</tr>
<tr>
<td>transform_view&lt;s,f&gt;::type</td>
<td>The size of ( v ); ( \text{size} v::\text{value} == \text{size} s::\text{value} ); linear complexity; see Forward Sequence.</td>
</tr>
</tbody>
</table>

Example

Find the largest type in a sequence.

```cpp
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

```cpp
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

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

Model of

— Forward Sequence

Parameters

Revision Date: 15th November 2004
### 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.

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

---

Revision Date: 15th November 2004
### 1.4.1 at

**Synopsis**

```cpp
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**

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

**Model of**

*Tag Dispatched Metafunction*

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td><em>Forward Sequence</em></td>
<td>A sequence to be examined.</td>
</tr>
<tr>
<td>AssocSeq</td>
<td><em>Associative Sequence</em></td>
<td>A sequence to be examined.</td>
</tr>
<tr>
<td>N</td>
<td><em>Integral Constant</em></td>
<td>An offset from the beginning of the sequence specifying the element to be retrieved.</td>
</tr>
<tr>
<td>Key</td>
<td>Any type</td>
<td>A key for the element to be retrieved.</td>
</tr>
<tr>
<td>Default</td>
<td>Any type</td>
<td>A default value to return if the element is not found.</td>
</tr>
</tbody>
</table>

Revision Date: 15th November 2004
Expression semantics

For any Forward Sequence s, and Integral Constant n:

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

**Return type:** A type.

**Precondition:** $0 \leq n\cdot\text{value} < \text{size}\cdot\text{value}$.

**Semantics:** Equivalent to

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

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

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

**Return type:** A type.

**Semantics:** If $\text{has\_key}<s,key>::\text{value} == \text{true}$, t is the value type associated with key; otherwise t is identical to x.

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

**Return type:** A type.

**Semantics:** Equivalent to

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

### Complexity

<table>
<thead>
<tr>
<th>Sequence archetype</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward Sequence</td>
<td>Linear.</td>
</tr>
<tr>
<td>Random Access Sequence</td>
<td>Amortized constant time.</td>
</tr>
<tr>
<td>Associative Sequence</td>
<td>Amortized constant time.</td>
</tr>
</tbody>
</table>

### Example

```cpp
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

Revision Date: 15th November 2004
1.4.2 at_c

Synopsis

```cpp
template<
    typename Sequence,
    long n>

struct at_c
{
    typedef unspecified type;
};
```

Description

Returns a type identical to the n\text{th} element from the beginning of the sequence. `at\_c<Sequence, n>::type` is a shortcut notation for `at< Sequence, long_<n> >::type`.

Header

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

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Forward Sequence</td>
<td>A sequence to be examined.</td>
</tr>
<tr>
<td>n</td>
<td>A compile-time integral constant</td>
<td>An offset from the beginning of the sequence specifying the element to be retrieved.</td>
</tr>
</tbody>
</table>

Expression semantics

```cpp
typedef at\_c<Sequence, n>::type t;
```

Return type: A type

Precondition: \(0 \leq n < \text{size<Sequence>::value}\)

Semantics: Equivalent to

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

Complexity

<table>
<thead>
<tr>
<th>Sequence archetype</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward Sequence</td>
<td>Linear.</td>
</tr>
<tr>
<td>Random Access Sequence</td>
<td>Amortized constant time.</td>
</tr>
</tbody>
</table>

Example

```cpp
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);
```

Revision Date: 15th November 2004
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

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

Description

Returns the last element in the sequence.

Header

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

Model of

Tag Dispatched Metafunction

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Bidirectional Sequence</td>
<td>A sequence to be examined.</td>
</tr>
</tbody>
</table>

Expression semantics

For any Bidirectional Sequence s:

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

Return type: A type.

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

Semantics: Equivalent to

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

Complexity

Amortized constant time.

Revision Date: 15th November 2004
Example

```cpp
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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Any type</td>
<td>A type whose begin iterator, if any, will be returned.</td>
</tr>
</tbody>
</table>

Expression semantics
For any arbitrary type x:
```
typedef begin<x>::type first;
```

Return type: Forward Iterator or `void_`.

Revision Date: 15th November 2004
1.4 Intrinsic Metafunctions

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 \( \text{size}<x>::\text{value} == 0 \).

Complexity
Amortized constant time.

Example

```cpp
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

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

Description
Returns an empty sequence concept-identical to Sequence.

Header

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

Model of
Tag Dispatched Metafunction

Parameters

Revision Date: 15th November 2004
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Extensible Sequence or Extensible Associative Sequence</td>
<td>A sequence to get an empty “copy” of.</td>
</tr>
</tbody>
</table>

**Expression semantics**

For any Extensible Sequence or Extensible Associative Sequence `s`:

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

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

**Semantics:** Equivalent to

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

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

**Complexity**

Amortized constant time.

**Example**

```cpp
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**

```cpp
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**

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

Revision Date: 15th November 2004
Model of
Tag Dispatched Metafunction

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Forward Sequence</td>
<td>A sequence to test.</td>
</tr>
</tbody>
</table>

Expression semantics

For any Forward Sequence \( s \):

\[
\text{typedef empty}<s>::\text{type} \ c;
\]

**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

\[
\text{typedef range_c}\langle\text{int,0,0}\rangle \ \text{empty\_range} ;
\text{typedef vector}\langle\text{long,\text{float,\text{double}}\rangle \ \text{types} ;
\]

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

\[
\text{template}<
\text{\hspace{1cm} typename X }
\n\text{\hspace{1cm} >
}\]

**Description**

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

Revision Date: 15th November 2004
Sequence 1.4 Intrinsic Metafunctions

Header

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

Model of
Tag Dispatched Metafunction

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Any type</td>
<td>A type whose end iterator, if any, will be returned.</td>
</tr>
</tbody>
</table>

Expression semantics

For any arbitrary type x:

```cpp
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

```cpp
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

```cpp
template<
  typename Sequence,
  typename First,
  typename Last = unspecified
>
struct erase
```
1.4 Intrinsic Metafunctions

Sequences

```
{  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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Extensible Sequence or Extensible Associative Sequence</td>
<td>A sequence to erase from.</td>
</tr>
<tr>
<td>First</td>
<td>Forward Iterator</td>
<td>An iterator to the beginning of the range to be erased.</td>
</tr>
<tr>
<td>Last</td>
<td>Forward Iterator</td>
<td>An iterator past-the-end of the range to be erased.</td>
</tr>
</tbody>
</table>

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: \([\text{begin}<s>::\text{type}, pos), [last, end<s>::\text{type})\).

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>::\text{type}>::\text{type} r;
```

For any Extensible Associative Sequence \( s \), and iterator \( pos \) into \( s \):
```
typedef erase<s,pos>::type r;
```

Return type: Extensible Sequence.

Revision Date: 15th November 2004
49   Sequences

1.4   Intrinsic Metafunctions

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

<table>
<thead>
<tr>
<th>Sequence archetype</th>
<th>Complexity (the range form)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extensible Associative Sequence</td>
<td>Amortized constant time.</td>
</tr>
<tr>
<td>Extensible Sequence</td>
<td>Quadratic in the worst case, linear at best.</td>
</tr>
</tbody>
</table>

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>

Revision Date: 15th November 2004
Intrinsic Metafunctions

Model of

Tag Dispatched Metafunction

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AssocSeq</td>
<td>Extensible Associative Sequence</td>
<td>A sequence to erase elements from.</td>
</tr>
<tr>
<td>Key</td>
<td>Any type</td>
<td>A key for the elements to be removed.</td>
</tr>
</tbody>
</table>

Expression semantics

For any Extensible Associative Sequence \( s \), and arbitrary type \( \text{key} \):

```cpp
typedef erase_key<s,\text{key}>::\text{type} \; r;
```

Return type: Extensible Associative Sequence.

Semantics: \( r \) is concept-identical and equivalent to \( s \) except that \( \text{has_key}\langle r,\text{key}\rangle::\text{value} == \text{false} \).

Postcondition: \( \text{size}\langle r\rangle::\text{value} == \text{size}\langle s\rangle::\text{value} - 1 \).

Complexity

Amortized constant time.

Example

```cpp
typedef map< pair<int,\text{unsigned}>, pair<char,\text{long}>> m;
typedef erase_key<m,\text{char}>::\text{type} m1;
BOOST_MPL_ASSERT_RELATION( \text{size}\langle m1\rangle::\text{type}::\text{value}, ==, 1 );
BOOST_MPL_ASSERT(( \text{is_same}\langle \text{at}\langle m1,\text{char}\rangle::\text{type},\text{void} > ));
BOOST_MPL_ASSERT(( \text{is_same}\langle \text{at}\langle m1,\text{int}\rangle::\text{type},\text{unsigned} > ));
```

See also

Extensible Associative Sequence, erase, has_key, insert

1.4.10 front

Synopsis

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

Revision Date: 15th November 2004
Description
Returns the first element in the sequence.

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

Model of
Tag Dispatched Metafunction

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Forward Sequence</td>
<td>A sequence to be examined.</td>
</tr>
</tbody>
</table>

Expression semantics
For any Forward Sequence \( s \):
```
typedef front<s>::type t;
```
**Return type:** A type.
**Precondition:** \( \text{empty}<s>::\text{value} == \text{false} \).
**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

Revision Date: 15th November 2004
1.4 Intrinsic Metafunctions

1.4.11 has_key

Synopsis

```cpp
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

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

Model of

Tag Dispatched Metafunction

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Associative Sequence</td>
<td>A sequence to query.</td>
</tr>
<tr>
<td>Key</td>
<td>Any type</td>
<td>The queried key.</td>
</tr>
</tbody>
</table>

Expression semantics

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

```cpp
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

```cpp
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> ));
```

Revision Date: 15th November 2004
See also

Associative Sequence, count, insert, erase_key

1.4.12 insert

Synopsis

```cpp
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

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

Model of

Tag Dispatched Metafunction

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Extensible Sequence or Extensible Associative Sequence</td>
<td>A sequence to insert into.</td>
</tr>
</tbody>
</table>
1.4 Intrinsic Metafunctions

<table>
<thead>
<tr>
<th><strong>Parameter</strong></th>
<th><strong>Requirement</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Pos</td>
<td>Forward Iterator</td>
<td>An iterator in Sequence specifying the insertion position.</td>
</tr>
<tr>
<td>T</td>
<td>Any type</td>
<td>The element to be inserted.</td>
</tr>
</tbody>
</table>

**Expression semantics**

For any Extensible Sequence \( s \), iterator \( \text{pos} \) in \( s \), and arbitrary type \( x \):

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

**Return type:** Extensible Sequence

**Precondition:** \( \text{pos} \) is an iterator in \( s \).

**Semantics:** \( r \) is a sequence, concept-identical to \( s \), of the following elements: \([\text{begin}\langle s\rangle::type, \text{pos}\rangle, x, [\text{pos}, \text{end}\langle s\rangle::type]).\)

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

\[
\text{at}< r, \text{distance}< \text{begin}\langle s\rangle::type, \text{pos} >::type >::type \\
is \text{identical to } x;
\]

\[
\text{size}<r>::\text{value} == \text{size}<s>::\text{value} + 1;
\]

For any Extensible Associative Sequence \( s \), iterator \( \text{pos} \) in \( s \), and arbitrary type \( x \):

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

**Return type:** Extensible Associative Sequence

**Semantics:** \( r \) is concept-identical and equivalent to \( s \), except that \( \text{at}< r, \text{key_type}<s,x>::type >::type \) is identical to \( \text{value_type}<s,x>::type$.\)

**Postcondition:** \( \text{size}<r>::\text{value} == \text{size}<s>::\text{value} + 1.\)

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

**Return type:** Extensible Associative Sequence

**Precondition:** \( \text{pos} \) is an iterator in \( s \).

**Semantics:** Equivalent to \( \text{typedef insert}<s,x>::type r;\) \( \text{pos} \) is ignored.

**Complexity**

<table>
<thead>
<tr>
<th><strong>Sequence archetype</strong></th>
<th><strong>Complexity</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Extensible Associative Sequence</td>
<td>Amortized constant time.</td>
</tr>
<tr>
<td>Extensible Sequence</td>
<td>Linear in the worst case, or amortized constant time.</td>
</tr>
</tbody>
</table>

**Example**

```cpp
typedef vector_c<int,0,1,3,4,5,6,7,8,9> numbers;
 typdef find< numbers,integral_c<int,3> >::type pos;
 typdef 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> > ));
```

Revision Date: 15th November 2004
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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Extensible Sequence or Extensible Associative Sequence</td>
<td>A sequence to insert into.</td>
</tr>
<tr>
<td>Pos</td>
<td>Forward Iterator</td>
<td>An iterator in Sequence specifying the insertion position.</td>
</tr>
<tr>
<td>Range</td>
<td>Forward Sequence</td>
<td>The range of elements to be inserted.</td>
</tr>
</tbody>
</table>

Expression semantics

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

Revision Date: 15th November 2004
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;
\[ \text{size}<r>::\text{value} == \text{size}<s>::\text{value} + \text{size}<\text{range}>::\text{value} \]

**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 );
BOOST_MPL_ASSERT(( equal< range,range_c<int,0,10> > ));
```
```
typedef insert_range<
    list0<
        , end< list0<> >::type
    , list<int>
    >::type result2;
BOOST_MPL_ASSERT_RELATION( size<result2>::value, ==, 1 );
```

**See also**
Extensible Sequence, insert, push_front, push_back, erase

### 1.4.14 is_sequence

**Synopsis**
```
template<
    typename X
>
struct is_sequence
{
    typedef unspecified type;
};
```

**Description**
Returns a boolean Integral Constant c such that c::value == true if and only if X is a model of Forward Sequence.

Revision Date: 15th November 2004
Header

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

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Any type</td>
<td>The type to query.</td>
</tr>
</tbody>
</table>

Expression semantics

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

**Return type**: Boolean Integral Constant.

**Semantics**: Equivalent to

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

Complexity

Amortized constant time.

Example

```cpp
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

```cpp
template<
    typename Sequence,
    typename X
>
struct key_type
{
```
1.4 Intrinsic Metafunctions

```cpp
typedef unspecified type;
```

**Description**

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

**Header**

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

**Model of**

Tag Dispatched Metafunction

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Associative Sequence</td>
<td>A sequence to query.</td>
</tr>
<tr>
<td>X</td>
<td>Any type</td>
<td>The type to get the key for.</td>
</tr>
</tbody>
</table>

**Expression semantics**

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

```cpp
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**

```cpp
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

Revision Date: 15th November 2004
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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Associative Sequence</td>
<td>A sequence to query.</td>
</tr>
<tr>
<td>Key</td>
<td>Any type</td>
<td>The queried key.</td>
</tr>
</tbody>
</table>

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_ > ));

Revision Date: 15th November 2004
1.4 Intrinsic Metafunctions

See also

Associative Sequence, has_key, count, map

1.4.17 pop_back

Synopsis

```cpp
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

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

Model of

Tag Dispatched Metafunction

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Back Extensible Sequence</td>
<td>A sequence to erase the last element from.</td>
</tr>
</tbody>
</table>

Expression semantics

For any Back Extensible Sequence $s$:

```cpp
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.

Revision Date: 15th November 2004
Example

```cpp
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

```cpp
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

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

Model of

Tag Dispatched Metafunction

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Front Extensible Sequence</td>
<td>A sequence to erase the first element from.</td>
</tr>
</tbody>
</table>
1.4 Intrinsic Metafunctions

Expression semantics

For any Front Extensible Sequence \( s \):

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

Return type: Front Extensible Sequence.

Precondition: \( \text{empty}<s>::\text{value} == \text{false} \).

Semantics: Equivalent to \( \text{erase}<s,\text{begin}<s>::\text{type}>::\text{type}; \).

Postcondition: \( \text{size}<r>::\text{value} == \text{size}<s>::\text{value} - 1 \).

Complexity

Amortized constant time.

Example

```cpp
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, \text{erase}, \text{push_front}, \text{front}, \text{pop_back}

1.4.19 push_back

Synopsis

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

Revision Date: 15th November 2004
Description

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

Header

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

Model of

Tag Dispatched Metafunction

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Back Extensible Sequence</td>
<td>A sequence to insert into.</td>
</tr>
<tr>
<td>T</td>
<td>Any type</td>
<td>The element to be inserted.</td>
</tr>
</tbody>
</table>

Expression semantics

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

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

**Return type:** Back Extensible Sequence.

**Semantics:** Equivalent to

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

**Postcondition:** back$r>::type is identical to $x$:

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

Complexity

Amortized constant time.

Example

```cpp
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`

Revision Date: 15th November 2004
1.4 Intrinsic Metafunctions

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Front Extensible</td>
<td>A sequence to insert into.</td>
</tr>
<tr>
<td>T</td>
<td>Any type</td>
<td>The element to be inserted.</td>
</tr>
</tbody>
</table>

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 );

Revision Date: 15th November 2004
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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Any type</td>
<td>A type to obtain a sequence tag for.</td>
</tr>
</tbody>
</table>

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.
1.4 Intrinsic Metafunctions

See also
Intrinsic Metafunctions, Tag Dispatched Metafunction

1.4.22 size

Synopsis

```cpp
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 \([\text{begin}<\text{Sequence}>::\text{type}, \text{end}<\text{Sequence}>::\text{type})\).

Header

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

Model of

Tag Dispatched Metafunction

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Forward Sequence</td>
<td>A sequence to query.</td>
</tr>
</tbody>
</table>

Expression semantics

For any Forward Sequence s:

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

Return type: Integral Constant.

Semantics: Equivalent to

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

Postcondition: \(n::\text{value} \geq 0\).

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.

Revision Date: 15th November 2004
If the \( s \) is a Random Access Sequence, size<\( s \>::\text{type} \) 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**

```cpp
typedef list<> 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**

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

**Description**

Returns the \text{value} that would be used for element \( X \) in Sequence.

**Header**

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

**Model of**

Tag Dispatched Metafunction

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Associative Sequence</td>
<td>A sequence to query.</td>
</tr>
<tr>
<td>( X )</td>
<td>Any type</td>
<td>The type to get the value for.</td>
</tr>
</tbody>
</table>
Expression semantics

For any **Associative Sequence** \( s \) and an arbitrary type \( x \):

```cpp
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

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

then

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

is identical to \( value_type<s,x>::type \).

**Complexity**

Amortized constant time.

**Example**

```cpp
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 interface 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 \(\text{deref}<i>::\text{type}\) 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 \(\text{next}<i>::\text{type}\) 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 \(\text{next}\) 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
### 2.1 Concepts

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>deref&lt;i&gt;::type</td>
<td>Any type</td>
<td>Amortized constant time</td>
</tr>
<tr>
<td>next&lt;i&gt;::type</td>
<td>Forward Iterator</td>
<td>Amortized constant time</td>
</tr>
<tr>
<td>i::category</td>
<td>Integral Constant, convertible to forward_iterator_tag</td>
<td>Constant time</td>
</tr>
</tbody>
</table>

**Expression semantics**

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

**Precondition:** `i` is dereferenceable

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

```cpp
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

```cpp
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 \( \text{prior}<i>::\text{type} \) 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.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>next&lt;i&gt;::type</td>
<td>Bidirectional Iterator</td>
<td>Amortized constant time</td>
</tr>
<tr>
<td>prior&lt;i&gt;::type</td>
<td>Bidirectional Iterator</td>
<td>Amortized constant time</td>
</tr>
<tr>
<td>i::category</td>
<td>Integral Constant, convertible to bidirectional_iterator_tag</td>
<td>Constant time</td>
</tr>
</tbody>
</table>

Expression semantics

\[
\text{typedef prior}<i>::\text{type} \ j;
\]

**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 \( \text{prior}< \text{next}<i>::\text{type} >::\text{type} \) is a null operation; similarly, if \( i \) is decrementable, \( \text{next}< \text{prior}<i>::\text{type} >::\text{type} \) 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.

Revision Date: 15th November 2004
### Expression Metafunctions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>next&lt;i&gt;::type</td>
<td>Random Access Iterator</td>
<td>Amortized constant time</td>
</tr>
<tr>
<td>prior&lt;i&gt;::type</td>
<td>Random Access Iterator</td>
<td>Amortized constant time</td>
</tr>
<tr>
<td>i::category</td>
<td>Integral Constant, convertible to random_access_iterator_tag</td>
<td>Constant time</td>
</tr>
<tr>
<td>advance&lt;i,n&gt;::type</td>
<td>Random Access Iterator</td>
<td>Amortized constant time</td>
</tr>
<tr>
<td>distance&lt;i,j&gt;::type</td>
<td>Integral Constant</td>
<td>Amortized constant time</td>
</tr>
</tbody>
</table>

#### Expression semantics

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

**Semantics:** See advance specification

```cpp
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**

```cpp
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**

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

Revision Date: 15th November 2004
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterator</td>
<td>Forward Iterator</td>
<td>An iterator to advance.</td>
</tr>
<tr>
<td>N</td>
<td>Integral Constant</td>
<td>A distance.</td>
</tr>
</tbody>
</table>

Model Of

Tag Dispatched Metafunction

Expression semantics

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

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

**Return type:** Forward Iterator.

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

**Semantics:** Equivalent to:

```cpp
typedef iter i0;
typedef next<i0>::type i1;
...
typedef next<i0-n>::type j;
```

if `n::value > 0`, and

```cpp
typedef iter i0;
typedef prior<i0>::type i1;
...
typedef prior<i0-n>::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

```cpp
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> ));
```

Revision Date: 15th November 2004
See also
Iterators, Tag Dispatched Metafunction, distance, next

### 2.2.2 distance

**Synopsis**

```cpp
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**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>First, Last</td>
<td>Forward Iterator</td>
<td>Iterators to compute a distance between.</td>
</tr>
</tbody>
</table>

**Model Of**

Tag Dispatched Metafunction

**Expression semantics**

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

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

**Return type:** Integral Constant.

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

**Semantics:** Equivalent to

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

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

Revision Date: 15th November 2004
Complexity
Amortized constant time if `first` and `last` are Random Access Iterators, otherwise linear time.

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

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterator</td>
<td>Forward Iterator.</td>
<td>An iterator to increment.</td>
</tr>
</tbody>
</table>

Expression semantics

For any Forward Iterators `iter`:
```cpp
typedef next<iter>::type j;
```

Return type: Forward Iterator.

Revision Date: 15th November 2004
**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

```cpp
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, \(\text{prior}<X>\) 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**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterator</td>
<td>Forward Iterator.</td>
<td>An iterator to decrement.</td>
</tr>
</tbody>
</table>

Revision Date: 15th November 2004
Expression semantics

For any Forward Iterators \texttt{iter}:

\begin{verbatim}
    typedef prior<iter>::type j;
\end{verbatim}

\textbf{Return type:} Forward Iterator.

\textbf{Precondition:} \texttt{iter} is decrementable.

\textbf{Semantics:} \texttt{j} is an iterator pointing to the previous element in the sequence. If \texttt{iter} is a user-defined iterator, the library-provided default implementation is equivalent to

\begin{verbatim}
    typedef iter::prior j;
\end{verbatim}

\textbf{Complexity}

Amortized constant time.

\textbf{Example}

\begin{verbatim}
    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 > ));
\end{verbatim}

\textbf{See also}

Iterators, \texttt{begin} / \texttt{end}, \texttt{next}, \texttt{derefer}

\subsection{2.2.5 deref}

\textbf{Synopsis}

\begin{verbatim}
    template<
        typename Iterator
    >
    struct deref
    {
        typedef unspecified type;
    };
\end{verbatim}

\textbf{Description}

Dereferences an iterator.

\textbf{Header}

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

\textbf{Parameters}

\textbf{Revision Date:} 15th November 2004
2.2 Iterator Metafunctions

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterator</td>
<td>Forward Iterator</td>
<td>The iterator to dereference.</td>
</tr>
</tbody>
</table>

Expression semantics

For any Forward Iterator iter:

```cpp
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

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

Complexity

Amortized constant time.

Example

```cpp
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**

```cpp
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`

Revision Date: 15th November 2004
Header

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

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterator</td>
<td>Forward Iterator</td>
<td>The iterator to obtain a category for.</td>
</tr>
</tbody>
</table>

Expression semantics

For any Forward Iterator `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
    >
{
};
```

Revision Date: 15th November 2004
See also

Iterators, begin / end, advance, distance, next
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 implicit *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*.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>in::state</td>
<td>Any type</td>
</tr>
<tr>
<td>in::operation</td>
<td>Binary Lambda Expression</td>
</tr>
</tbody>
</table>
Expression semantics

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>in::state</td>
<td>The inserter’s initial state.</td>
</tr>
<tr>
<td>in::operation</td>
<td>The inserter’s “output” operation.</td>
</tr>
</tbody>
</table>

Example

```cpp
typedef transform<
    range_c<int,0,10>
    , plus<_1, _1>
    , back_inserter<vector<>>
>::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**.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x&lt;s_1, s_2, ..., s_n, ...&gt;::type</code></td>
<td>Forward Sequence</td>
<td>Unspecified.</td>
</tr>
<tr>
<td><code>x&lt;s_1, s_2, ..., s_n, ... in&gt;::type</code></td>
<td>Any type</td>
<td>Unspecified.</td>
</tr>
<tr>
<td><code>reverse_x&lt;s_1, s_2, ..., s_n, ...&gt;::type</code></td>
<td>Forward Sequence</td>
<td>Unspecified.</td>
</tr>
<tr>
<td><code>reverse_x&lt;s_1, s_2, ..., s_n, ... in&gt;::type</code></td>
<td>Any type</td>
<td>Unspecified.</td>
</tr>
</tbody>
</table>

Expression semantics

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

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

Revision Date: 15th November 2004
Semantics: t is equivalent to
\[ x < \]
\[ s_1, s_2, \ldots, s_n, \ldots, \]
\[ \text{back_inserter< clear<s_i>::type >} \]
\[ ::\text{type} \]
if has_push_back<s_i>::value == true and
\[ \text{reverse_x<} \]
\[ s_1, s_2, \ldots, s_n, \ldots, \]
\[ \text{front_inserter< clear<s_i>::type >} \]
\[ ::\text{type} \]
otherwise.

typedef x<s_1, s_2, \ldots, s_n, \ldots, \text{in}>::\text{type} t;

Semantics: t is the result of an \text{x} invocation with arguments \( s_1, s_2, \ldots, s_n, \ldots, \text{in} \).

typedef reverse_x<s_1, s_2, \ldots, s_n, \ldots, \text{in}>::\text{type} t;

Precondition: \( s_1 \) is an Extensible Sequence.

Semantics: t is equivalent to
\[ x < \]
\[ s_1, s_2, \ldots, s_n, \ldots, \]
\[ \text{front_inserter< clear<s_i>::type >} \]
\[ ::\text{type} \]
if has_push_front<s_i>::value == true and
\[ \text{reverse_x<} \]
\[ s_1, s_2, \ldots, s_n, \ldots, \]
\[ \text{back_inserter< clear<s_i>::type >} \]
\[ ::\text{type} \]
otherwise.

typedef reverse_x<s_1, s_2, \ldots, s_n, \ldots, \text{in}>::\text{type} t;

Semantics: t is the result of a \text{reverse_x} invocation with arguments \( s_1, s_2, \ldots, s_n, \ldots, \text{in} \).

Example

typedef transform<
    range_c<int,0,10>,
    plus<_1,int_<7>>,
    \text{back_inserter< vector0<> >} >::\text{type} r1;

typedef transform< r1, minus<_1,int_2>, \text{back_inserter< vector0<> >} >::\text{type} r2;
typedef reverse_transform<
    r2,
    minus<_1,5>,
    \text{front_inserter< vector0<> >} >::\text{type} r3;

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

Revision Date: 15th November 2004
3.2 Inserters

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq</td>
<td>Back Extensible Sequence</td>
<td>A sequence to bind the inserter to.</td>
</tr>
</tbody>
</table>

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 `a`:
<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>back_inserter&lt;s&gt;</td>
<td>An Inserter in, equivalent to</td>
</tr>
<tr>
<td></td>
<td>struct in : inserter&lt;s,push_back&lt;_1,_2&gt; &gt; {};</td>
</tr>
</tbody>
</table>

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
3.2 Inserters

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:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>front_inserter&lt;s&gt;</code></td>
<td>An Inserter in, equivalent to struct in : inserter&lt;s,push_front&lt;_1,_2&gt; &gt; {};</td>
</tr>
</tbody>
</table>

Complexity

Amortized constant time.

Example

```cpp
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

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

Description

A general-purpose model of the Inserter concept.

Revision Date: 15th November 2004
Header

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

Model of

Inserter

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>Any type</td>
<td>A initial state.</td>
</tr>
<tr>
<td>Operation</td>
<td>Binary Lambda Expression</td>
<td>An output operation.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>inserter&lt;op,state&gt;</td>
<td>An Inserter.</td>
</tr>
</tbody>
</table>

Complexity

Amortized constant time.

Example

```cpp
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 Iteration Algorithms

3.3.1 fold

Synopsis

\[
\text{template<}
\text{typename Sequence, typename State, typename ForwardOp >}
\text{struct fold}
\{
\text{typedef unspecified type;}
\};
\]

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 \([\text{begin<Sequence>::type}, \text{end<Sequence>::type})\) in order.

Header

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

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Forward Sequence</td>
<td>A sequence to iterate.</td>
</tr>
<tr>
<td>State</td>
<td>Any type</td>
<td>The initial state for the first ForwardOp application.</td>
</tr>
<tr>
<td>ForwardOp</td>
<td>Binary Lambda Expression</td>
<td>The operation to be executed on forward traversal.</td>
</tr>
</tbody>
</table>

Expression semantics

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

\[
\text{typedef fold<s, state, op>::type t;}
\]

Return type: A type.

Semantics: Equivalent to

\[
\text{typedef iter_fold< s, state, apply<op, _1, deref<_2> > >::type t;}
\]

Complexity

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

Example

\[
\text{typedef vector<long, float, short, double, float, long, long double> types;}
\]

\[
\text{typedef fold<}
\text{types}
\]

Revision Date: 15th November 2004
See also

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

3.3.2 iter_fold

Synopsis

```cpp
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 \([\text{begin<Sequence>::type}, \text{end<Sequence>::type})\) in order.

Header

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

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Forward Sequence</td>
<td>A sequence to iterate.</td>
</tr>
<tr>
<td>State</td>
<td>Any type</td>
<td>The initial state for the first ForwardOp application.</td>
</tr>
<tr>
<td>ForwardOp</td>
<td>Binary Lambda Expression</td>
<td>The operation to be executed on forward traversal.</td>
</tr>
</tbody>
</table>

Expression semantics

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

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

Return type: A type.

Semantics: Equivalent to

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

Revision Date: 15th November 2004
typedef next<i>::type i2;
typedef apply<op,state1,i2>::type state2;
...
typedef apply<op,state_{n-1},i_n>::type state_n;
typedef next<i_n>::type last;
typedef state_{n} 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**

```cpp
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.

Revision Date: 15th November 2004
#include <boost/mpl/reverse_fold.hpp>

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Forward Sequence</td>
<td>A sequence to iterate.</td>
</tr>
<tr>
<td>State</td>
<td>Any type</td>
<td>The initial state for the first BackwardOp / ForwardOp application.</td>
</tr>
<tr>
<td>BackwardOp</td>
<td>Binary Lambda Expression</td>
<td>The operation to be executed on backward traversal.</td>
</tr>
<tr>
<td>ForwardOp</td>
<td>Binary Lambda Expression</td>
<td>The operation to be executed on forward traversal.</td>
</tr>
</tbody>
</table>

**Expression semantics**

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

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

**Return type:** A type

**Semantics:** Equivalent to

```cpp
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

```cpp
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:

```cpp
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
```
3.3 Iteration Algorithms

, list_c<int>, if<_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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Forward Sequence</td>
<td>A sequence to iterate.</td>
</tr>
<tr>
<td>State</td>
<td>Any type</td>
<td>The initial state for the first BackwardOp / ForwardOp application.</td>
</tr>
<tr>
<td>BackwardOp</td>
<td>Binary Lambda Expression</td>
<td>The operation to be executed on backward traversal.</td>
</tr>
<tr>
<td>ForwardOp</td>
<td>Binary Lambda Expression</td>
<td>The operation to be executed on forward traversal.</td>
</tr>
</tbody>
</table>

Expression semantics

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

---

Revision Date: 15th November 2004

\[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

```cpp
typedef begin<s>::type i;
typedef next<i>::type i1;
...
typedef next<i_n>::type last;
typedef apply<backward_op, state, i_n>::type state_n;
typedef apply<backward_op, state_n, i_{n-1}>::type state_{n-1};
...
typedef apply<backward_op, state_2, i_1>::type state_1;
typedef state_1 t;
```

where \( n = \text{size}<s>::\text{value} \) 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

```cpp
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.

```cpp
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

Revision Date: 15th November 2004
3.3.5 accumulate

Synopsis

```cpp
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

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

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Forward Sequence</td>
<td>A sequence to iterate.</td>
</tr>
<tr>
<td>State</td>
<td>Any type</td>
<td>The initial state for the first <code>ForwardOp</code> application.</td>
</tr>
<tr>
<td>ForwardOp</td>
<td>Binary Lambda Expression</td>
<td>The operation to be executed on forward traversal.</td>
</tr>
</tbody>
</table>

Expression semantics

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

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

**Return type:** A type.

**Semantics:** Equivalent to

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

Complexity

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

Example

```cpp
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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Forward Sequence</td>
<td>A sequence to search in.</td>
</tr>
<tr>
<td>T</td>
<td>Any type</td>
<td>A type to search for.</td>
</tr>
</tbody>
</table>

Expression semantics

For any Forward Sequence s and arbitrary type t:

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

Return type:  Forward Iterator.

Semantics:  Equivalent to

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

Revision Date: 15th November 2004
Complexity
Linear. At most $\text{size}::\text{value}$ comparisons for identity.

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

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Forward Sequence</td>
<td>A sequence to search in.</td>
</tr>
<tr>
<td>Pred</td>
<td>Unary Lambda Expression</td>
<td>A search condition.</td>
</tr>
</tbody>
</table>

Expression semantics
For any Forward Sequence $s$ and unary Lambda Expression $\text{pred}$:
```cpp
typedef find_if<s, pred>::type i;
```

Return type: Forward Iterator.

Semantics: $i$ is the first iterator in the range $[\text{begin}s::\text{type}, \text{end}s::\text{type})$ 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

```cpp
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**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Forward Sequence</td>
<td>A sequence to be examined.</td>
</tr>
<tr>
<td>T</td>
<td>Any type</td>
<td>A type to search for.</td>
</tr>
</tbody>
</table>

**Expression semantics**

For any Forward Sequence s and arbitrary type t:
typedef contains<s,t>::type r;

**Return type:** Integral Constant.

**Semantics:** Equivalent to

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

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

**Example**
```cpp
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**
```cpp
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**
```cpp
#include <boost/mpl/count.hpp>
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Forward Sequence</td>
<td>A sequence to be examined.</td>
</tr>
<tr>
<td>T</td>
<td>Any type</td>
<td>A type to search for.</td>
</tr>
</tbody>
</table>

Revision Date: 15th November 2004
Expression semantics

For any Forward Sequence s and arbitrary type t:

```cpp
typedef count<s,t>::type n;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```cpp
typedef count_if< s,is_same<_,T> >::type n;
```

Complexity

Linear. Exactly size<s>::value comparisons for identity.

Example

```cpp
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**

```cpp
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**

```cpp
#include <boost/mpl/count_if.hpp>
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Forward Sequence</td>
<td>A sequence to be examined.</td>
</tr>
<tr>
<td>Pred</td>
<td>Unary Lambda Expression</td>
<td>A count condition.</td>
</tr>
</tbody>
</table>

Revision Date: 15th November 2004
### Expression semantics

For any Forward Sequence s and unary Lambda Expression pred:

```cpp
typedef count_if<s,pred>::type n;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```cpp
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**

```cpp
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**

```cpp
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

```cpp
#include <boost/mpl/lower_bound.hpp>
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Forward Sequence</td>
<td>A sorted sequence to search in.</td>
</tr>
<tr>
<td>T</td>
<td>Any type</td>
<td>A type to search a position for.</td>
</tr>
<tr>
<td>Pred</td>
<td>Binary Lambda Expression</td>
<td>A search criteria.</td>
</tr>
</tbody>
</table>

Expression semantics

For any sorted Forward Sequence $s$, binary Lambda Expression $\text{pred}$, and arbitrary type $x$:

```cpp
typedef lower_bound< s,x,pred >::type i;
```

**Return type:** Forward Iterator.

**Semantics:** $i$ is the furthermost iterator in $[\begin{eqnarray} \text{begin}<s>::\text{type} \end{eqnarray}, \text{end}<s>::\text{type}]$ such that, for every iterator $j$ in $[\begin{eqnarray} \text{begin}<s>::\text{type} \end{eqnarray}, i)$,

```cpp
apply< \text{pred}, \text{deref}<j>::\text{type}, x >::\text{value} == \text{true}
```

Complexity

The number of comparisons is logarithmic: at most \(\log_2(\text{size}<s>::\text{value}) + 1\). 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 $\text{size}<s>::\text{value}$.

Example

```cpp
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

```cpp
template<
    typename Sequence,
    typename T
```
3.4 Querying Algorithms

`, 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**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Forward Sequence</td>
<td>A sorted sequence to search in.</td>
</tr>
<tr>
<td>T</td>
<td>Any type</td>
<td>A type to search a position for.</td>
</tr>
<tr>
<td>Pred</td>
<td>Binary Lambda Expression</td>
<td>A search criteria.</td>
</tr>
</tbody>
</table>

**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 furthest 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 `log2(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( defer<iter>::type::value, ==, 5 );
```

**Revision Date:** 15th November 2004
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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Forward Sequence</td>
<td>A sequence to be searched.</td>
</tr>
<tr>
<td>Pred</td>
<td>Binary Lambda Expression</td>
<td>A comparison criteria.</td>
</tr>
</tbody>
</table>

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<
3.4 Querying Algorithms

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Forward Sequence</td>
<td>A sequence to be searched.</td>
</tr>
<tr>
<td>Pred</td>
<td>Binary Lambda Expression</td>
<td>A comparison criteria.</td>
</tr>
</tbody>
</table>

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 >::value == false

Complexity

Linear. Zero comparisons if s is empty, otherwise exactly size<s>::value - 1 comparisons.

Revision Date: 15th November 2004
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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq1,Seq2</td>
<td>Forward Sequence</td>
<td>Sequences to compare.</td>
</tr>
<tr>
<td>Pred</td>
<td>Binary Lambda Expression</td>
<td>A comparison criterion.</td>
</tr>
</tbody>
</table>

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
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 algorithm 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

```cpp
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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Forward Sequence</td>
<td>A sequence to copy.</td>
</tr>
<tr>
<td>In</td>
<td>Inserter</td>
<td>An inserter.</td>
</tr>
</tbody>
</table>

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
```

Revision Date: 15th November 2004
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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Forward Sequence</td>
<td>A sequence to copy.</td>
</tr>
<tr>
<td>Pred</td>
<td>Unary Lambda Expression</td>
<td>A copying condition.</td>
</tr>
<tr>
<td>In</td>
<td>Inserter</td>
<td>An inserter.</td>
</tr>
</tbody>
</table>

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>
>
>

Revision Date: 15th November 2004
Complexity

Linear. Exactly \( \text{size}<s>::\text{value} \) applications of \text{pred}, and at most \( \text{size}<s>::\text{value} \) applications of \text{in}::\text{operation}.

Example

```cpp
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

```cpp
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`, `e2`) 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**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Sequence, Seq1, Seq2</code></td>
<td>Forward Sequence</td>
<td>Sequences to transform.</td>
</tr>
<tr>
<td><code>Op, BinaryOp</code></td>
<td>Lambda Expression</td>
<td>A transformation.</td>
</tr>
<tr>
<td><code>In</code></td>
<td>Inserter</td>
<td>An inserter.</td>
</tr>
</tbody>
</table>

**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`:

```cpp
typedef transform<s,op,in>::type r;
```

**Return type:** A type.

**Postcondition:** Equivalent to

```cpp
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;
```

```cpp
typedef transform<s1,s2,op,in>::type r;
```

**Return type:** A type.

**Postcondition:** Equivalent to

```cpp
typedef lambda<op2>::type f;
typedef lambda<in::operation>::type in_op;

typedef fold<
```

Revision Date: 15th November 2004
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>

Revision Date: 15th November 2004
3.5 Transformation Algorithms

Model of
Reversible Algorithm

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Forward Sequence</td>
<td>A original sequence.</td>
</tr>
<tr>
<td>OldType</td>
<td>Any type</td>
<td>A type to be replaced.</td>
</tr>
<tr>
<td>NewType</td>
<td>Any type</td>
<td>A type to replace with.</td>
</tr>
<tr>
<td>In</td>
<td>Inserter</td>
<td>An inserter.</td>
</tr>
</tbody>
</table>

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 $y$:

```cpp
typedef replace<s,x,y,in>::type r;
```

**Return type:** A type.

**Semantics:** Equivalent to

```cpp
typedef replace_if< s,y,is_same<_,x>,in >::type r;
```

Complexity

Linear. Performs exactly $\text{size}<s>::\text{value}$ comparisons for identity / insertions.

Example

```cpp
typedef vector<int,float,char,float,float,double> types;
typedef vector<int,double,char,double,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

```cpp
template<
    typename Sequence,
    typename Pred
    , typename In = unspecified
>
struct replace_if
{
```

Revision Date: 15th November 2004
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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Forward Sequence</td>
<td>An original sequence.</td>
</tr>
<tr>
<td>Pred</td>
<td>Unary Lambda Expression</td>
<td>A replacement condition.</td>
</tr>
<tr>
<td>NewType</td>
<td>Any type</td>
<td>A type to replace with.</td>
</tr>
<tr>
<td>In</td>
<td>Inserter</td>
<td>An inserter.</td>
</tr>
</tbody>
</table>

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;

Revision Date: 15th November 2004
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

```cpp
template<
    typename Sequence,
    typename T
    , typename In = unspecified
>
struct remove
{
    typedef unspecified type;
};
```

#### Description

Returns a new sequence that contains all elements from \[ \text{begin<Sequence>::type}, \text{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

```cpp
#include <boost/mpl/remove.hpp>
```

#### Model of

Reversible Algorithm

#### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Forward Sequence</td>
<td>An original sequence.</td>
</tr>
<tr>
<td>T</td>
<td>Any type</td>
<td>A type to be removed.</td>
</tr>
<tr>
<td>In</td>
<td>Inserter</td>
<td>An inserter.</td>
</tr>
</tbody>
</table>

#### 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 \( \text{in} \), and arbitrary type \( x \):

```cpp
typedef remove<s,x,\text{in}>::type r;
```

Return type: A type.
Semantics: Equivalent to

```cpp
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

```cpp
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

```cpp
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

```cpp
#include <boost/mpl/remove_if.hpp>
```

Model of

Reversible Algorithm

Parameters

Revision Date: 15th November 2004
### 3.5 Transformation Algorithms

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Forward Sequence</td>
<td>An original sequence.</td>
</tr>
<tr>
<td>Pred</td>
<td>Unary Lambda Expression</td>
<td>A removal condition.</td>
</tr>
<tr>
<td>In</td>
<td>Inserter</td>
<td>An inserter.</td>
</tr>
</tbody>
</table>

#### 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`:

```cpp
typedef remove_if<s,pred,in>::type r;
```

**Return type:** A type.

**Semantics:** Equivalent to

```cpp
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

```cpp
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

```cpp
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**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Forward Sequence</td>
<td>An original sequence.</td>
</tr>
<tr>
<td>Pred</td>
<td>Binary Lambda Expression</td>
<td>An equivalence relation.</td>
</tr>
<tr>
<td>In</td>
<td>Inserter</td>
<td>An inserter.</td>
</tr>
</tbody>
</table>

**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:

```cpp
typedef unique<s,pred,in>::type r;
```

**Return type:** A type.

**Semantics:** If size<s>::value <= 1, then equivalent to

```cpp
typedef copy<s,in>::type r;
```

otherwise equivalent to

```cpp
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;
```

```cpp
typedef fold<
```

Revision Date: 15th November 2004
3.5 Transformation Algorithms

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, 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>

Revision Date: 15th November 2004
Model of
Reversible Algorithm

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq</td>
<td>Forward Sequence</td>
<td>An original sequence.</td>
</tr>
<tr>
<td>Pred</td>
<td>Unary Lambda Expression</td>
<td>A partitioning predicate.</td>
</tr>
<tr>
<td>In1, In2</td>
<td>Inserter</td>
<td>Output inserters.</td>
</tr>
</tbody>
</table>

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 Inserter s in1 and in2:

```cpp
typedef partition<s, pred, in1, in2>::type r;
```

Return type: A pair.

Semantics: Equivalent to

```cpp
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

```cpp
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

```cpp
template<
```
3.5 Transformation Algorithms

```cpp
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**

```cpp
#include <boost/mpl/stable_partition.hpp>
```

**Model of**

Reversible Algorithm

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq</td>
<td>Forward Sequence</td>
<td>An original sequence.</td>
</tr>
<tr>
<td>Pred</td>
<td>Unary Lambda Expression</td>
<td>A partitioning predicate.</td>
</tr>
<tr>
<td>In1, In2</td>
<td>Inserter</td>
<td>Output inserters.</td>
</tr>
</tbody>
</table>

**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`:

```cpp
typedef stable_partition<s,pred,in1,in2>::type r;
```

*Return type: A pair.*

*Semantics: Equivalent to*

```cpp
typedef lambda<pred>::type p;
typedef lambda<in1::operation>::type in1_op;
typedef lambda<in2::operation>::type in2_op;
```

```cpp
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 \texttt{size}<s>::value applications of \texttt{pred}, and \texttt{size}<s>::value of summarized \texttt{in1::operation} / \texttt{in2::operation} applications.

**Example**

```cpp
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;
```

```cpp
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, \texttt{reverse_stable_partition}, \texttt{partition}, \texttt{sort}, \texttt{transform}

### 3.5.11 sort

**Synopsis**

```cpp
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 \texttt{[begin<Seq>::type, end<Seq>::type)} sorted according to the ordering relation \texttt{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*
3.5 Transformation Algorithms

Header

```cpp
#include <boost/mpl/sort.hpp>
```

Model of

Reversible Algorithm

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq</td>
<td>Forward Sequence</td>
<td>An original sequence.</td>
</tr>
<tr>
<td>Pred</td>
<td>Binary Lambda Expression</td>
<td>An ordering relation.</td>
</tr>
<tr>
<td>In</td>
<td>Inserter</td>
<td>An inserter.</td>
</tr>
</tbody>
</table>

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`:

```cpp
typedef sort<s,pred,in>::type r;
```

**Return type**: A type.

**Semantics:** If `size<s>::value <= 1`, equivalent to

```cpp
typedef copy<s,in>::type r;
```

otherwise equivalent to

```cpp
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;
```

Revision Date: 15th November 2004
Complexity

Average $O(n \log(n))$ where $n \equiv \text{size<s>::value}$, quadratic at worst.

Example

```cpp
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

```cpp
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

```cpp
#include <boost/mpl/reverse.hpp>
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Forward Sequence</td>
<td>A sequence to reverse.</td>
</tr>
<tr>
<td>In</td>
<td>Inserter</td>
<td>An inserter.</td>
</tr>
</tbody>
</table>

Expression semantics

For any `Forward Sequence` `a`, and an `Inserter` `in`:

```cpp
typedef reverse<s,in>::type r;
```

Revision Date: 15th November 2004
Return type: A type.

Semantics: Equivalent to

```cpp
typedef reverse_copy<s,in>::type r;
```

Complexity
Linear.

Example

```cpp
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

```cpp
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

```cpp
#include <boost/mpl/copy.hpp>
```

Model of

Reversible Algorithm

Parameters
### 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`:

```cpp
typedef reverse_copy<s,in>::type r;
```

**Return type:** A type.

**Semantics:** Equivalent to

```cpp
typedef reverse_fold<s,in::state,in::operation>::type r;
```

### Complexity

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

### Example

```cpp
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

```cpp
template<
  typename Sequence,
  typename Pred,
  typename In = unspecified
>
struct reverse_copy_if
{
  typedef unspecified type;
};
```

Revision Date: 15th November 2004
Description

Returns a reversed, filtered copy of the original sequence containing the elements that satisfy the predicate \texttt{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

```cpp
#include <boost/mpl/copy_if.hpp>
```

Model of

Reversible Algorithm

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Forward Sequence</td>
<td>A sequence to copy.</td>
</tr>
<tr>
<td>Pred</td>
<td>Unary Lambda Expression</td>
<td>A copying condition.</td>
</tr>
<tr>
<td>In</td>
<td>Inserter</td>
<td>An inserter.</td>
</tr>
</tbody>
</table>

Expression semantics

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

For any \texttt{Forward Sequence} \texttt{s}, an unary \texttt{Lambda Expression} \texttt{pred}, and an \texttt{Inserter} \texttt{in}:

```cpp
typedef reverse_copy_if<s,pred,in>::type r;
```

**Return type:** A type

**Semantics:** Equivalent to

```cpp
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 \texttt{size<s>::value} applications of \texttt{pred}, and at most \texttt{size<s>::value} applications of \texttt{in::operation}.

Revision Date: 15th November 2004
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, e2) 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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence, Seq1, Seq2</td>
<td>Forward Sequence</td>
<td>Sequences to transform.</td>
</tr>
<tr>
<td>Op, BinaryOp</td>
<td>Lambda Expression</td>
<td>A transformation.</td>
</tr>
<tr>
<td>In</td>
<td>Inserter</td>
<td>An inserter.</td>
</tr>
</tbody>
</table>

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 $s, s_1$ and $s_2$, Lambda Expressions $op$ and $op_2$, and an Inserter $in$:

```cpp
typedef reverse_transform<s,op,in>::type r;
```

**Return type:** A type.

**Postcondition:** Equivalent to

```cpp
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

```cpp
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;
```

Revision Date: 15th November 2004
Complexity

Linear. Exactly \( \frac{\text{size}_s::\text{value}}{\text{size}_s1::\text{value}} \) applications of \( \text{op} / \text{op2} \) and \( \text{in}::\text{operation} \).

Example

```cpp
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

```cpp
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

```cpp
#include <boost/mpl/replace.hpp>
```

Model of

Reversible Algorithm

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Forward Sequence</td>
<td>A original sequence.</td>
</tr>
</tbody>
</table>
### 3.5 Transformation Algorithms

#### 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 \( y \):

\[
\text{typedef reverse_replace}< s,x,y,in >::\text{type } r;
\]

**Return type:** A type.

**Semantics:** Equivalent to

\[
\text{typedef reverse_replace_if}< s,y,\text{is_same}<_,x>,in >::\text{type } r;
\]

### Complexity

Linear. Performs exactly \( \text{size}<s>::\text{value} \) comparisons for identity / insertions.

### Example

```cpp
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**

```cpp
template<
    typename Sequence
, typename Pred
, typename In = unspecified
>
struct reverse_replace_if
{
    typedef unspecified type;
};
```

Revision Date: 15th November 2004
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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Forward Sequence</td>
<td>An original sequence.</td>
</tr>
<tr>
<td>Pred</td>
<td>Unary Lambda Expression</td>
<td>A replacement condition.</td>
</tr>
<tr>
<td>NewType</td>
<td>Any type</td>
<td>A type to replace with.</td>
</tr>
<tr>
<td>In</td>
<td>Inserter</td>
<td>An inserter.</td>
</tr>
</tbody>
</table>

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 reverse_replace_if<s,pred,x,in>::type r;
```

Return type: A type.

Semantics: Equivalent to

```
typedef lambda<pred>::type p;
typedef reverse_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 reverse_replace_if<
    numbers
    , greater< _, int_<4> >
    , int_<0>
    , front_inserter< vector<> >
```

Revision Date: 15th November 2004
3.5 Transformation Algorithms

>::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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Forward Sequence</td>
<td>An original sequence.</td>
</tr>
<tr>
<td>T</td>
<td>Any type</td>
<td>A type to be removed.</td>
</tr>
<tr>
<td>In</td>
<td>Inserter</td>
<td>An inserter.</td>
</tr>
</tbody>
</table>

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 a, 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

Revision Date: 15th November 2004
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Forward Sequence</td>
<td>An original sequence.</td>
</tr>
<tr>
<td>Pred</td>
<td>Unary Lambda Expression</td>
<td>A removal condition.</td>
</tr>
<tr>
<td>In</td>
<td>Inserter</td>
<td>An inserter.</td>
</tr>
</tbody>
</table>

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:

```cpp
typedef reverse_remove_if<s,pred,in>::type r;
```

**Return type:** A type.

**Semantics:** Equivalent to

```cpp
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**

```cpp
typedef vector_c<int,1,4,5,2,7,5,3,5>::type numbers;
typedef reverse_remove_if< numbers, greater<_1,int::_4> > r::type result;

BOOST_MPL_ASSERT((equal< result,vector_c<int,3,2,4,1>,equal_to<_1,_4> > ));
```

**See also**

Transformation Algorithms, Reversible Algorithm, remove_if, reverse_remove, reverse_copy_if, replace_if

### 3.5.20 reverse_unique

**Synopsis**

```cpp
    template<
        typename Seq
```
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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Forward Sequence</td>
<td>An original sequence.</td>
</tr>
<tr>
<td>Pred</td>
<td>Binary Lambda Expression</td>
<td>An equivalence relation.</td>
</tr>
<tr>
<td>In</td>
<td>Inserter</td>
<td>An inserter.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq</td>
<td>Forward Sequence</td>
<td>An original sequence.</td>
</tr>
<tr>
<td>Pred</td>
<td>Unary Lambda Expression</td>
<td>A partitioning predicate.</td>
</tr>
<tr>
<td>In1, In2</td>
<td>Inserter</td>
<td>Output inserters.</td>
</tr>
</tbody>
</table>

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 Inserter s 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

Revision Date: 15th November 2004
3.5 Transformation Algorithms

3.5.22 reverse_stable_partition

Synopsis

```cpp
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

```cpp
#include <boost/mpl/stable_partition.hpp>
```

Model of

Reversible Algorithm

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq</td>
<td>Forward Sequence</td>
<td>An original sequence.</td>
</tr>
<tr>
<td>Pred</td>
<td>Unary Lambda Expression</td>
<td>A partitioning predicate.</td>
</tr>
<tr>
<td>In1, In2</td>
<td>Inserter</td>
<td>Output inserters.</td>
</tr>
</tbody>
</table>

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`:

```cpp
typedef reverse_stable_partition<s, pred, in1, in2>::type r;
```

Return type: A pair.

Semantics: Equivalent to

```cpp
typedef lambda<pred>::type p;
typedef lambda<in1::operation>::type in1_op;
typedef lambda<in2::operation>::type in2_op;
```

Revision Date: 15th November 2004
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) {};  

type def 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

Revision Date: 15th November 2004
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, encapsulating 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 central role in making programming with MPL seemingly effortless. In particular, there are at least two contexts where built-in language facilities fall short:

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 \texttt{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.

\footnote{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 advised to use them in the place of the conventional operators, even at the price of slightly decreased readability.}
4.1 Concepts

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>f::type</td>
<td>Any type</td>
<td>Unspecified.</td>
</tr>
<tr>
<td>f&lt;&gt;::type</td>
<td>Any type</td>
<td>Unspecified.</td>
</tr>
<tr>
<td>f&lt;a1,..,an&gt;::type</td>
<td>Any type</td>
<td>Unspecified.</td>
</tr>
</tbody>
</table>

Expression semantics

```cpp
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.

```cpp
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.

```cpp
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.

Revision Date: 15th November 2004
### Expression

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>f::apply::type</code></td>
<td>Any type</td>
<td>Unspecified.</td>
</tr>
<tr>
<td><code>f::apply&lt;&gt;::type</code></td>
<td>Any type</td>
<td>Unspecified.</td>
</tr>
<tr>
<td><code>f::apply&lt;\ldots an&gt;::type</code></td>
<td>Any type</td>
<td>Unspecified.</td>
</tr>
</tbody>
</table>

### Expression semantics

```cpp
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.

```cpp
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.

```cpp
typedef f::apply<\ldots 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 \( a_1, \ldots, a_n \).

### 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 concise way to describe these requirements.

#### Expression requirements

See corresponding Metafunction Class and Placeholder Expression specifications.
4.1 Concepts

— 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 \( a_1,\ldots,a_n \) are arbitrary types, then \( X<a_1,\ldots,a_n> \) is a Placeholder Expression if and only if all of the following conditions hold:

— At least one of the template arguments \( a_1,\ldots,a_n \) 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_FUNCTION_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/extension of the metafunction's behavior.

Notation
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Legend</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>A placeholder token for the specific metafunction’s name.</td>
</tr>
<tr>
<td>tag-metafunction</td>
<td>A placeholder token for the tag metafunction’s name.</td>
</tr>
<tr>
<td>tag</td>
<td>A placeholder token for one of possible tag types returned by the tag metafunction.</td>
</tr>
</tbody>
</table>

**Synopsis**

```cpp
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 \( \text{user}_\text{type} \) that would have the required semantics.

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 (\text{list0}, \text{list1}, \text{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

```cpp
#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

4Usualy such user-defined specialization is still required to preserve the f’s original invariants and complexity requirements.

Revision Date: 15th November 2004
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 x₁, x₂,..., xₙ are arbitrary numeric types.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>op_tag&lt;x&gt;::type</td>
<td>Integral Constant</td>
<td>Amortized constant time.</td>
</tr>
<tr>
<td>op_impl&lt;op_tag&lt;x&gt;::type, op_tag&lt;y&gt;::type&gt;::apply&lt;x,y&gt;::type</td>
<td>Any type</td>
<td>Unspecified.</td>
</tr>
<tr>
<td>op&lt;x₁,x₂,...,xₙ&gt;::type</td>
<td>Any type</td>
<td>Unspecified.</td>
</tr>
</tbody>
</table>

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<x₁,x₂,...,xₙ>::type r;

**Semantics:** r is the result of op application on arguments x₁,x₂,...,xₙ.

Example

```cpp
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 {
```

Revision Date: 15th November 2004
4.1 Concepts

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 access 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.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>name&lt;x&gt;::type</td>
<td>Any type</td>
<td>Constant time.</td>
</tr>
</tbody>
</table>

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
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Integral Constant</td>
<td>A selection condition.</td>
</tr>
<tr>
<td>T1, T2</td>
<td>Any type</td>
<td>Types to select from.</td>
</tr>
</tbody>
</table>

### Expression semantics

For any Integral Constant c and arbitrary types t1, t2:

```cpp
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

```cpp
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

```cpp
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 shortcut notation for if_< bool_<c>,t1,t2 >::type.

#### Header

```cpp
#include <boost/mpl/if.hpp>
```

#### Parameters
4.2 Type Selection

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>An integral constant</td>
<td>A selection condition.</td>
</tr>
<tr>
<td>T1, T2</td>
<td>Any type</td>
<td>Types to select from.</td>
</tr>
</tbody>
</table>

Expression semantics

For any integral constant c and arbitrary types t1, t2:

```cpp
typedef if_c<c, t1, t2>::type t;
```

**Return type:** Any type.

**Semantics:** Equivalent to `typedef if_<bool_<c>, t1, t2 >::type t;`

**Example**

```cpp
typedef if_c<true, char, long>::type t1;
typedef if_c,false, char, long>::type t2;
```

```cpp
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**

```cpp
#include <boost/mpl/eval_if.hpp>
```

**Description**

Evaluates one of its two nullary-metafunction arguments, F1 or F2, depending on the value C.

**Header**

```cpp
#include <boost/mpl/eval_if.hpp>
```

**Parameters**
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Integral Constant</td>
<td>An evaluation condition.</td>
</tr>
<tr>
<td>F1, F2</td>
<td>Nullary Metafunction</td>
<td>Metafunctions to select for evaluation from.</td>
</tr>
</tbody>
</table>

**Expression semantics**

For any Integral Constant \( c \) and nullary Metafunctions \( f_1, f_2 \):

```cpp
typedef eval_if<c,f1,f2>::type t;
```

**Return type:** Any type.

**Semantics:** If \( c::\text{value} == \text{true} \), \( t \) is identical to \( f_1::\text{type} \); otherwise \( t \) is identical to \( f_2::\text{type} \).

**Example**

```cpp
typedef eval_if< true_, identity<char>, identity<long> >::type t1;
typedef eval_if< false_, identity<char>, identity<long> >::type t2;
```

```cpp
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**

```cpp
template<
    bool c
 , typename F1
 , typename F2
 >
struct eval_if_c
{
    typedef unspecified type;
};
```

**Description**

Evaluates one of its two nullary-metafunction arguments, \( F_1 \) or \( F_2 \), depending on the value of integral constant \( c \). 
\( \text{eval\_if\_c} <c,f1,f2>::\text{type} \) is a shortcut notation for \( \text{eval\_if} < \text{bool} <c>, \text{f1,f2} >::\text{type} \).

**Header**

```cpp
#include <boost/mpl/eval_if.hpp>
```

**Parameters**

Revision Date: 15th November 2004
4.3 Invocation Metafunctions

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>An integral constant</td>
<td>An evaluation condition.</td>
</tr>
<tr>
<td>F1, F2</td>
<td>Nullary Metafunction</td>
<td>Metafunctions to select for evaluation from.</td>
</tr>
</tbody>
</table>

Expression semantics

For any integral constant \( c \) and nullary Metafunctions \( f_1, f_2 \):

```cpp
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

```cpp
typedef eval_if_c< true, identity<char>, identity<long> >::type t1;
typedef eval_if_c< false, identity<char>, identity<long> >::type t2;
```

```cpp
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**

```cpp
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
>
```

Revision Date: 15th November 2004
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**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>Lambda Expression</td>
<td>An expression to invoke.</td>
</tr>
<tr>
<td>A1,... An</td>
<td>Any type</td>
<td>Invocation arguments.</td>
</tr>
</tbody>
</table>

**Expression semantics**

For any Lambda Expression f and arbitrary types a1,... an:

```cpp
typedef applyn<f,a1,...an>::type t;
typedef apply<f,a1,...an>::type t;
```

**Return type:** Any type.

**Semantics:** Equivalent to `typedef apply_wrap<f::type,a1,... an>::type t;`.

**Example**

```cpp
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 );
```

Revision Date: 15th November 2004
BOOST_MPL_ASSERT_RELATION( r2::value, ==, 5 );

See also
Metafunctions, apply_wrap, lambda, quote, bind

4.3.2 apply_wrap

Synopsis

```cpp
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

Revision Date: 15th November 2004
Expression semantics

For any Metafunction Class \( f \) and arbitrary types \( a_1, \ldots, a_n \):

\[
\text{typedef apply\_wrap}<f,a_1,\ldots,a_n>::\text{type} \ t;
\]

**Return type:** Any type.

**Semantics:** If \( n > 0 \), equivalent to \( \text{typedef } f::\text{apply}<a_1,\ldots,a_n>::\text{type} \ t; \), otherwise equivalent to either \( \text{typedef } f::\text{apply}::\text{type} \ t; \) or \( \text{typedef } f::\text{apply}<>::\text{type} \ t; \) depending on whether \( f::\text{apply} \) is a class or a class template.

Example

```cpp
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;
    };
};
```

```cpp
typedef apply\_wrap0< f0 >::\text{type} \ r1;
typedef apply\_wrap0< g0 >::\text{type} \ r2;
typedef apply\_wrap2< f2,int,char >::\text{type} \ r3;
```

```cpp
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

```cpp
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

```cpp
#include <boost/mpl/unpack_args.hpp>
```

Model of

Metafunction Class

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F$</td>
<td>Lambda Expression</td>
<td>A lambda expression to adopt.</td>
</tr>
</tbody>
</table>

Expression semantics

For an arbitrary Lambda Expression $f$, and arbitrary types $a_1, \ldots, a_n$:

```cpp
typedef unpack_args<f> g;
```

**Return type:** Metafunction Class.

**Semantics:** $g$ is a unary Metafunction Class such that

```cpp
apply<\text{apply_wrap}_n\< g, \text{vector}<a_1,\ldots,a_n>\>,::\text{type}
```

is identical to

```cpp
apply<F,a_1,\ldots,a_n,::\text{type}
```

Example

```cpp
BOOST_MPL_ASSERT(( apply<
    unpack_args< is_same<_1,_2> >,
    vector<int,int>
    > ));
```

Revision Date: 15th November 2004
4.4 Composition and Argument Binding

4.4.1 Placeholders

Synopsis

```cpp
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

```cpp
#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:

```cpp
typedef apply_wrap<n, a1, ..., an>::type x;
```

Return type: A type.

Semantics: Equivalent to

Revision Date: 15th November 2004
typedef apply_wrap\narg<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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Any type</td>
<td>An expression to transform.</td>
</tr>
<tr>
<td>Tag</td>
<td>Any type</td>
<td>A tag determining transform semantics.</td>
</tr>
</tbody>
</table>

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

```cpp
typedef protect< bind<
    quote<n<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

```cpp
typedef protect< bind<
    quote<n<X,tag>
    , lambda<a1,tag>::type,... lambda<an,tag>::type
  > > f;
```

otherwise, f is identical to x.

**Example**

```cpp
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**

```cpp
template<
  typename F
>
struct bind0
{
  // unspecified
  // ...
};
```

Revision Date: 15th November 2004
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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>Metafunction Class</td>
<td>An metafunction class to perform binding on.</td>
</tr>
<tr>
<td>A1,... An</td>
<td>Any type</td>
<td>Arguments to bind.</td>
</tr>
</tbody>
</table>
Expression semantics

For any Metafunction Class $f$ and arbitrary types $a_1, \ldots, a_n$:

\[
\text{typedef } \text{bind}\langle f, a_1, \ldots, a_n \rangle \text{ g;} \\
\text{typedef } \text{bindn}\langle f, a_1, \ldots, a_n \rangle \text{ g;} \\
\]

**Return type:** Metafunction Class

**Semantics:** Equivalent to

```
struct g {
    template<
        typename U1 = unspecified
        ...
    , typename Un = unspecified
    >
    struct apply :
    : apply_wrap<
        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 h_k :
    : apply_wrap<X,U1,...Un>
    {
    };
```

if $f$ or $a_k$ is a bind expression or a placeholder, and

```
template< typename X, typename U1,
    ... typename Un > struct h_k {
    typedef X type;
};
```

otherwise. [Note: Every $n$th 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 {
```

Revision Date: 15th November 2004
Composition and Argument Binding Metafunctions

{  
    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  
    // ...  
};

Revision Date: 15th November 2004
...  

```cpp
template<
    template< typename P1,...  typename Pn > class F
  , typename Tag = unspecified
>
struct quote
{
  // unspecified
  // ...
};
```

**Description**

`quote` is a higher-order primitive that wraps an \textit{n}-ary \texttt{Metafunction} to create a corresponding \texttt{Metafunction Class}.

**Header**

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

**Model of**

\texttt{Metafunction Class}

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{F}</td>
<td>\textit{Metafunction}</td>
<td>A metafunction to wrap.</td>
</tr>
<tr>
<td>\textit{Tag}</td>
<td>Any type</td>
<td>A tag determining wrap semantics.</td>
</tr>
</tbody>
</table>

**Expression semantics**

For any \textit{n}-ary \texttt{Metafunction} \textit{f} and arbitrary type \textit{tag}:

```
typedef quote<\textit{f}> g;
typedef quote<\textit{f},\textit{tag}> g;
```

**Return type:** \texttt{Metafunction Class}

**Semantics:** Equivalent to

```
struct g
{
    template< typename A1,...  typename An > struct apply
    :
      \textit{f}\langle A1,... An\rangle
    
    
};
```

if \textit{f}\langle A1,... An\rangle has a nested type member \texttt{::type}, and to

```
struct g
{
```

Revision Date: 15th November 2004
4.4 Composition and Argument Binding

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 A n = 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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>An integral constant</td>
<td>A number of argument to return.</td>
</tr>
</tbody>
</table>

Expression semantics
For any integral constant n in the range [1, BOOST_MPL_LIMIT_METAFUNCTION_ARITY] and arbitrary types a1,... an:

typedef apply_wrap<n< 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
{

Description

Protect is an identity wrapper for a Metafunction Class that prevents its argument from being recognized as a bind expression.

Header

```cpp
#include <boost/mpl/protect.hpp>
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>Metafunction Class</td>
<td>A metafunction class to wrap.</td>
</tr>
</tbody>
</table>

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_wrap<f,U1,...,Un>
    {
        <
    }
};
```

otherwise equivalent to `typedef f g;`.

Example

```
FIXME

struct f
{
    template< typename T1, typename T2 > struct apply
    {
        // ...
    };
};
```

Revision Date: 15th November 2004
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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1, T2,... Tn</td>
<td>Integral Constant</td>
<td>Operation’s arguments.</td>
</tr>
</tbody>
</table>
Expression semantics

For any Integral Constants \( c_1, c_2, \ldots, c_n \):

\[
\text{typedef plus\langle c_1, \ldots, c_n \rangle \cdot \text{type} r;}
\]

**Return type:** Integral Constant.

**Semantics:** Equivalent to

\[
\text{typedef integral_c<}
\]
\[
\text{typeof(c1::value + c2::value)}
\]
\[
, (c1::value + c2::value)
\]
\[
> c;
\]
\[
\text{typedef plus\langle c, c_3, \ldots, c_n \rangle \cdot \text{type} r;}
\]
\[
\text{typedef plus\langle c_1, \ldots, c_n \rangle \cdot r;}
\]

**Return type:** Integral Constant.

**Semantics:** Equivalent to

\[
\text{struct r} : \text{plus\langle c_1, \ldots, c_n \rangle \cdot \text{type} {}};
\]

Complexity

Amortized constant time.

Example

\[
\text{typedef plus\langle \text{int}_{<-10}, \text{int}_{<3}, \text{long}_{<1} \rangle \cdot \text{type} r;}
\]
\[
\text{BOOST_MPL_ASSERT\_RELATION( r::value, \text{==}, -6 \};}
\]
\[
\text{BOOST\_MPL\_ASSERT(( is\_same\langle r::value\_type, \text{long} > ))};
\]

See also

Arithmetic Operations, Numeric Metafunction, numeric\_cast, minus, negate, times

4.5.2 minus

Synopsis

\[
\text{template<}
\]
\[
\text{typename T1}
\]
\[
, \text{typename T2}
\]
\[
, \text{typename T3} = \text{unspecified}
\]
\[
\ldots
\]
\[
, \text{typename Tn} = \text{unspecified}
\]
\[
> \text{struct minus}
\]

Revision Date: 15th November 2004
### Description

Returns the difference of its arguments.

### Header

```cpp
#include <boost/mpl/minus.hpp>
#include <boost/mpl/arithmetic.hpp>
```

### Model of

Numeric Metafunction

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1, T2,..., Tn</td>
<td>Integral Constant</td>
<td>Operation’s arguments.</td>
</tr>
</tbody>
</table>

[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 $c_1, c_2, ..., c_n$:

```cpp
typedef minus<c1, ... cn>::type r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```cpp
typedef integral_c<
    typedef(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

```cpp
struct r : minus<c1,...cn>::type {};
```

### Complexity

Amortized constant time.

---

Revision Date: 15th November 2004
Example

```cpp
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

```cpp
template<
    typename T1,
    typename T2,
    typename T3 = unspecified ...
>
    struct times
    {
        typedef unspecified type;
    };
```

Description

Returns the product of its arguments.

Header

```cpp
#include <boost/mpl/times.hpp>
#include <boost/mpl/arithmetic.hpp>
```

Model of

Numeric Metafunction

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1, T2,... Tn</td>
<td>Integral Constant</td>
<td>Operation’s arguments.</td>
</tr>
</tbody>
</table>

[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 \( c_1, c_2, \ldots, c_n \):

```cpp
typedef times<c1,...cn>::type r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```cpp
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

```cpp
struct r : times<c1,...cn>::type {};
```

Complexity

Amortized constant time.

Example

```cpp
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

```cpp
template<
    typename T1
    , typename T2
    , typename T3 = unspecified
    ...
    , typename Tn = unspecified
>
struct divides
{
    typedef unspecified type;
};
```

Revision Date: 15th November 2004
4.5 Arithmetic Operations

Description

Returns the quotient of its arguments.

Header

```cpp
#include <boost/mpl/divides.hpp>
#include <boost/mpl/arithmetic.hpp>
```

Model of

Numeric Metafunction

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1, T2,... Tn</td>
<td>Integral Constant</td>
<td>Operation’s arguments.</td>
</tr>
</tbody>
</table>

[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 \(c_1, c_2, \ldots, c_n\):

```cpp
typedef divides<c1,...cn>::type r;
```

Return type: Integral Constant.

Precondition: \(c_2::value \neq 0, \ldots, c_n::value \neq 0\).

Semantics: Equivalent to

```cpp
typedef integral_c<
    typeof(c1::value / c2::value)
, ( c1::value / c2::value )
> c;
```

```cpp
typedef divides<c,c3,...cn>::type r;
```

```cpp
typedef divides<c1,...cn> r;
```

Complexity

Amortized constant time.

Revision Date: 15th November 2004
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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1, T2</td>
<td>Integral Constant</td>
<td>Operation’s arguments.</td>
</tr>
</tbody>
</table>

[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

```cpp
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

```cpp
struct r : modulus<c1, c2>::type {};
```

Complexity
Amortized constant time.

Example
```cpp
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
```cpp
template<
    typename T
>
struct negate
{
    typedef unspecified type;
};
```

Description
Returns the negative (additive inverse) of its argument.

Header
```cpp
#include <boost/mpl/negate.hpp>
#include <boost/mpl/arithmetic.hpp>
```
Model of
Numeric Metafunction

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>Integral Constant</td>
<td>Operation’s argument.</td>
</tr>
</tbody>
</table>

[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 Constant c:

```cpp
typedef negate<c>::type r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```cpp
typedef integral_c< c::value_type, ( -c::value ) > r;

typedef negate<c> r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```cpp
struct r : negate<c>::type {};
```

Complexity

Amortized constant time.

Example

```cpp
typedef negate< int_<-10> >::type r;
BOOST_MPL_ASSERT_RELATION( r::value, ==, 10 );
BOOST_MPL_ASSERT(( is_same< r::value_type, int > ));
```

See also

Arithmetic Operations, Numeric Metafunction, numeric_cast, plus, minus, times

4.6 Comparisons

4.6.1 less

Synopsis

```cpp
template<
    typename T1
```
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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1, T2</td>
<td>Integral Constant</td>
<td>Operation’s arguments.</td>
</tr>
</tbody>
</table>

[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

```cpp
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

```cpp
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

```cpp
#include <boost/mpl/less_equal.hpp>
#include <boost/mpl/comparison.hpp>
```

Model of

Numeric Metafunction

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1, T2</td>
<td>Integral Constant</td>
<td>Operation’s arguments.</td>
</tr>
</tbody>
</table>

[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:

```cpp
typedef less_equal<c1,c2>::type r;
```
4.6 Comparisons

Return type: Integral Constant.

Semantics: Equivalent to

```cpp
typedef bool_<(c1::value <= c2::value)> r;
typedef less_equal<c1,c2> r;
```

Return type: Integral Constant.

Semantics: Equivalent to

```cpp
struct r : less_equal<c1,c2>::type {};
```

Complexity

Amortized constant time.

Example

```cpp
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

```cpp
template<
    typename T1,
    typename T2>
struct greater
{
    typedef unspecified type;
};
```

Description

Returns a true-valued Integral Constant if T1 is greater than T2.

Header

```cpp
#include <boost/mpl/greater.hpp>
#include <boost/mpl/comparison.hpp>
```

Model of

Numeric Metafunction

Revision Date: 15th November 2004
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1, T2</td>
<td>Integral Constant</td>
<td>Operation’s arguments.</td>
</tr>
</tbody>
</table>

[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:

```c
typedef greater<c1,c2>::type r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```c
typedef bool_< (c1::value < c2::value) > r;
```

```c
typedef greater<c1,c2> r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```c
struct r : greater<c1,c2>::type {};
```

Complexity

Amortized constant time.

Example

```c
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

```c
template<
    typename T1,
    typename T2
>
struct greater_equal
{
    typedef unspecified type;
};
```

Revision Date: 15th November 2004
4.6 Comparisons

Description

Returns a true-valued Integral Constant if T1 is greater than or equal to T2.

Header

```cpp
#include <boost/mpl/greater_equal.hpp>
#include <boost/mpl/comparison.hpp>
```

Model of

Numeric Metafunction

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1, T2</td>
<td>Integral Constant</td>
<td>Operation’s arguments.</td>
</tr>
</tbody>
</table>

[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:

```cpp
typedef greater_equal<c1,c2>::type r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```cpp
typedef bool_< (c1::value < c2::value) > r;
typedef greater_equal<c1,c2> r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```cpp
struct r : greater_equal<c1,c2>::type {};
```

Complexity

Amortized constant time.

Example

```cpp
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> > ));
```

Revision Date: 15th November 2004
See also

Comparisons, Numeric Metafunction, numeric_cast, greater, less, equal_to

4.6.5 equal_to

Synopsis

```cpp
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

```cpp
#include <boost/mpl/equal_to.hpp>
#include <boost/mpl/comparison.hpp>
```

Model of

Numeric Metafunction

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1, T2</td>
<td>Integral Constant</td>
<td>Operation’s arguments.</td>
</tr>
</tbody>
</table>

[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:

```cpp
typedef equal_to<c1,c2>::type r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```cpp
typedef bool_< (c1::value == c2::value) > r;
typedef equal_to<c1,c2> r;
```

Revision Date: 15th November 2004
Return type: Integral Constant.

Semantics: Equivalent to

```cpp
struct r : equal_to<c1,c2>::type {};
```

Complexity

Amortized constant time.

Example

```cpp
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

```cpp
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

```cpp
#include <boost/mpl/not_equal_to.hpp>
#include <boost/mpl/comparison.hpp>
```

Model of

Numeric Metafunction

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1, T2</td>
<td>Integral Constant</td>
<td>Operation’s arguments.</td>
</tr>
</tbody>
</table>
Expression semantics
For any Integral Constants c1 and c2:
```cpp
typedef not_equal_to<c1,c2>::type r;
```
**Return type:** Integral Constant.
**Semantics:** Equivalent to
```cpp
typedef bool_< (c1::value != c2::value) > r;
```
```cpp
typedef not_equal_to<c1,c2> r;
```
**Return type:** Integral Constant.
**Semantics:** Equivalent to
```cpp
struct r : not_equal_to<c1,c2>::type {};
```

Complexity
Amortized constant time.

Example
```cpp
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
```cpp
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

```cpp
#include <boost/mpl/and.hpp>
#include <boost/mpl/logical.hpp>
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1, F2,.., Fn</td>
<td>Nullary Metafunction</td>
<td>Operation’s arguments.</td>
</tr>
</tbody>
</table>

Expression semantics

For arbitrary nullary Metafunctions f1, f2,.. fn:

```cpp
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 fi metafunction that evaluates to `false` are not evaluated.

```cpp
typedef and_<f1,f2,...,fn> r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```cpp
struct r : and_<f1,f2,...,fn>::type {};
```

Example

```cpp
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

```cpp
template<
typedef 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

```cpp
#include <boost/mpl/or.hpp>
#include <boost/mpl/logical.hpp>
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1, F2,..., Fn</td>
<td>Nullary Metafunction</td>
<td>Operation’s arguments.</td>
</tr>
</tbody>
</table>

Expression semantics

For arbitrary nullary Metafunctions f1, f2,... fn:

```cpp
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 fi metafunction that evaluates to true are not evaluated.

```cpp
typedef or_<f1,f2,...,fn> r;
```

Return type: Integral Constant.

Semantics: Equivalent to

```cpp
struct r : or_<f1,f2,...,fn>::type {};
```

Example

```cpp
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
```

Revision Date: 15th November 2004
See also
Metafunctions, Logical Operations, and_, not_

4.7.3 not_

Synopsis

```cpp
template<
    typename F
>
struct not_
{
    typedef unspecified type;
};
```

Description

Returns the result of logical not (!) operation on its argument.

Header

```cpp
#include <boost/mpl/not.hpp>
#include <boost/mpl/logical.hpp>
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>Nullary Metafunction</td>
<td>Operation’s argument.</td>
</tr>
</tbody>
</table>

Expression semantics

For arbitrary nullary Metafunction f:

```cpp
typef not_<f>::type r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```cpp
typef bool_< (!f::type::value) > r;
```

```cpp
typef not_<f> r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```cpp
struct r : not_<f>::type {};
```

Example

```cpp
BOOST_MPL_ASSERT_NOT(( not_< true_ > ));
BOOST_MPL_ASSERT(( not_< false_ > ));
```

Revision Date: 15th November 2004
4.8 Bitwise Operations

4.8.1 bitand_

Synopsis

```cpp
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

```cpp
#include <boost/mpl/bitand.hpp>
#include <boost/mpl/bitwise.hpp>
```

Model of

Numeric Metafunction

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1, T2, ..., Tn</td>
<td>Integral Constant</td>
<td>Operation’s arguments.</td>
</tr>
</tbody>
</table>

[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 $c_1, c_2, ..., c_n$:

```cpp
typedef bitand_<c1, ... cn>::type r;
```

Return type: Integral Constant.

Revision Date: 15th November 2004
4.8 Bitwise Operations

**Semantics:** Equivalent to

```cpp
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

```cpp
struct r : bitand_<c1,...cn>::type {};
```

**Complexity**

Amortized constant time.

**Example**

```cpp
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( (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,uffffffff>::value), ==, 0 );
BOOST_MPL_ASSERT_RELATION( (bitand_<u1,uffffffff>::value), ==, 1 );
BOOST_MPL_ASSERT_RELATION( (bitand_<u8,uffffffff>::value), ==, 8 );
```

**See also**

Bitwise Operations, Numeric Metafunction, numeric_cast, bitor_, bitxor_, shift_left

4.8.2 bitor_

**Synopsis**

```cpp
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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1, T2,... Tn</td>
<td>Integral Constant</td>
<td>Operation’s arguments.</td>
</tr>
</tbody>
</table>

[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_<
    typedef(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.

Revision Date: 15th November 2004
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

Revision Date: 15th November 2004
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T_1, T_2, \ldots, T_n)</td>
<td>Integral Constant</td>
<td>Operation’s arguments.</td>
</tr>
</tbody>
</table>

[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 \(c_1, c_2, \ldots, c_n\):

```cpp
typedef bitxor_<c1, ..., cn>::type r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```cpp
typedef integral_c<
    typeof(c1::value ^ c2::value)
, ( c1::value ^ c2::value )
> c;
```

```cpp
typedef bitxor_<c,c3,...,cn>::type r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```cpp
struct r : bitxor_<c1, ..., cn>::type {};
```

### Complexity

Amortized constant time.

### Example

```cpp
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`

Revision Date: 15th November 2004
4.8.4 shift_left

Synopsis

```cpp
template<
    typename T,
    typename Shift
>
struct shift_left
{
    typedef unspecified type;
};
```

Description

Returns the result of bitwise shift left (<<) operation on T.

Header

```cpp
#include <boost/mpl/shift_left.hpp>
#include <boost/mpl/bitwise.hpp>
```

Model of

Numeric Metafunction

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>Integral Constant</td>
<td>A value to shift.</td>
</tr>
<tr>
<td>Shift</td>
<td>Unsigned Integral Constant</td>
<td>A shift distance.</td>
</tr>
</tbody>
</table>

[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:

```cpp
typedef shift_left<c,shift>::type r;
```

Return type: Integral Constant.

Semantics: Equivalent to

```cpp
typedef integral_c<
    c::value_type
  , ( c::value << shift::value )
  > r;

typedef shift_left<c,shift> r;
```

Revision Date: 15th November 2004
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

Revision Date: 15th November 2004
4.8 Bitwise Operations

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>Integral Constant</td>
<td>A value to shift.</td>
</tr>
<tr>
<td>Shift</td>
<td>Unsigned Integral Constant</td>
<td>A shift distance.</td>
</tr>
</tbody>
</table>

[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:

```cpp
typedef shift_right<c,shift>::type r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```cpp
typedef integral_c<
    c::value_type,
    ( c::value >> shift::value )
> r;
```

```cpp
typedef shift_right<c,shift> r;
```

**Return type:** Integral Constant.

**Semantics:** Equivalent to

```cpp
struct r : shift_right<c,shift>::type {};
```

Complexity

Amortized constant time.

Example

```cpp
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_

Revision Date: 15th November 2004
4.9 Trivial

The MPL provides a number of Trivial Metafunctions that are 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 components 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.

<table>
<thead>
<tr>
<th>Metafunction</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{first&lt;x&gt;::type}</td>
<td>#include &lt;boost/mpl/pair.hpp&gt;</td>
</tr>
<tr>
<td>\texttt{second&lt;x&gt;::type}</td>
<td>#include &lt;boost/mpl/pair.hpp&gt;</td>
</tr>
<tr>
<td>\texttt{base&lt;x&gt;::type}</td>
<td>#include &lt;boost/mpl/base.hpp&gt;</td>
</tr>
</tbody>
</table>

See Also

Metafunctions, Trivial Metafunction

4.10 Miscellaneous

4.10.1 identity

Synopsis

```cpp
template<
    typename X
>
struct identity
{
    typedef X type;
};
```

Description

The \texttt{identity} metafunction. Returns \( X \) unchanged.

Header

\#include <boost/mpl/identity.hpp>

Model of

Metafunction

Parameters

Revision Date: 15th November 2004
4.10 Miscellaneous Metafunctions

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Any type</td>
<td>An argument to be returned.</td>
</tr>
</tbody>
</table>

Expression semantics

For an arbitrary type x:

```cpp
typedef identity<x>::type r;
```

**Return type:** A type.

**Semantics:** Equivalent to

```cpp
typedef x r;
```

**Postcondition:** `is_same<r,x>::value == true`.

Example

```cpp
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**

```cpp
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**

```cpp
#include <boost/mpl/always.hpp>
```

Revision Date: 15th November 2004
Model of
Metafunction Class

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Any type</td>
<td>A type to be returned.</td>
</tr>
</tbody>
</table>

Expression semantics

For an arbitrary type \( x \):

```cpp
typedef always<x> f;
```

**Return type**: Metafunction Class.

**Semantics**: Equivalent to

```cpp
struct f : bind< identity<_1>, x > {};
```

Example

```cpp
typedef always<true_> always_true;
```

```cpp
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**

```cpp
template<
    typename T1, typename T2
>
struct inherit2
{
    typedef unspecified type;
};
```

```cpp
...
```

```cpp
template<
    typename T1, typename T2,... typename Tn
>
struct inheritn
{
    typedef unspecified type;
};
```

Revision Date: 15th November 2004
};

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1, T2,... Tn</td>
<td>A class type</td>
<td>Classes to derived from.</td>
</tr>
</tbody>
</table>

Expression semantics

For arbitrary class types t1,t2,... tn:

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 {};

Revision Date: 15th November 2004
typedef inherit\langle t_1,t_2,...,t_n\rangle::type r;

**Return type:** A class type.

**Precondition:** \( t_1, t_2, ..., t_n \) are complete types.

**Semantics:** Equivalent to

```cpp
struct r
    : inherit2<
        inherit\langle t_1,t_2,...,t_{n-1}\rangle::type
    , t_n
    
    
};
```

typedef inherit\langle t_1,t_2,...,t_n\rangle::type r;

**Precondition:** \( t_1, t_2, ..., t_n \) are complete types.

**Return type:** A class type.

**Semantics:** Equivalent to

```cpp
typedef inherit\langle t_1,t_2,...,t_n\rangle::type r;
```

**Complexity**

Amortized constant time.

**Example**

```cpp
struct udt1 { int n; }
struct udt2 {};

typedef inherit\langle udt1, udt2\rangle::type r1;
typedef inherit\langle empty_base, udt1\rangle::type r2;
typedef inherit\langle empty_base, udt1, empty_base, empty_base\rangle::type r3;
typedef inherit\langle udt1, empty_base, udt2\rangle::type r4;
typedef inherit\langle empty_base, empty_base\rangle::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

Revision Date: 15th November 2004
4.10 Miscellaneous

4.10.4 inherit_linearly

Synopsis

```cpp
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

```cpp
#include <boost/mpl/inherit_linearly.hpp>
```

Model of

Metafunction

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Types</td>
<td>Forward Sequence</td>
<td>Types to inherit from.</td>
</tr>
<tr>
<td>Node</td>
<td>Binary Lambda Expression</td>
<td>A derivation metafunction.</td>
</tr>
<tr>
<td>Root</td>
<td>A class type</td>
<td>A type to be placed at the root of the class hierarchy.</td>
</tr>
</tbody>
</table>

Expression semantics

For any Forward Sequence `types`, binary Lambda Expression `node`, and arbitrary class type `root`:

```cpp
typedef inherit_linearly<Types,node,root>::type r;
```

**Return type:** A class type.

**Semantics:** Equivalent to

```cpp
typedef fold<Types,root,node>::type r;
```

Complexity

Linear. Exactly `size<Types>::value` applications of `node`.

Revision Date: 15th November 2004
Example

```cpp
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

```cpp
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.

Revision Date: 15th November 2004
4.10 Miscellaneous Metafunctions

Header

```cpp
#include <boost/mpl/numeric_cast.hpp>
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SourceTag</td>
<td>Integral Constant</td>
<td>A tag for the conversion’s source type.</td>
</tr>
<tr>
<td>TargetTag</td>
<td>Integral Constant</td>
<td>A tag for the conversion’s destination type.</td>
</tr>
</tbody>
</table>

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:

```cpp
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

```cpp
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

Revision Date: 15th November 2004
4.10 Miscellaneous Metafunctions

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N1, N2</td>
<td>Any type</td>
<td>Types to compare.</td>
</tr>
</tbody>
</table>

Expression semantics

For arbitrary types x and y:

\[
\text{typedef min}\langle x, y\rangle::\text{type } r; 
\]

Return type: A type.

Precondition: \( \text{less}\langle x, y\rangle::\text{value} \) is a well-formed integral constant expression.

Semantics: Equivalent to

\[
\text{typedef if}_< \text{less}\langle x, y\rangle, x, y >::\text{type } r; 
\]

Complexity

Constant time.

Example

\[
\text{typedef fold<} \\
\quad \text{vector}\_c\langle \text{int}, 1, 7, 0, -2, 5, -1\rangle \\
\quad , \text{int}_<-10> \\
\quad , \text{min}_<1, 2> \\
\quad >::\text{type } r; \\
\]

\[
\text{BOOST}_\text{MPL}\_\text{ASSERT}(( \text{is}_\text{same}\langle r, \text{int}_<-10> > )); 
\]

See also

Metafunctions, comparison, max, less, min_element

4.10.7 max

Synopsis

\[
\text{template}<> \\
\quad \text{typename N1} \\
\quad , \text{typename N2} \\
\quad > \\
\quad \text{struct max} \\
\quad \{ \\
\quad \quad \text{typedef unspecified type;} \\
\quad \}; 
\]

Description

Returns the larger of its two arguments.
Header

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

Model of

Metafunction

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N1, N2</td>
<td>Any type</td>
<td>Types to compare.</td>
</tr>
</tbody>
</table>

Expression semantics

For arbitrary types \( x \) and \( y \):

```
typedef max<x,y>::type r;
```

Return type: A type.

Precondition: \( \text{less}<x,y>::\text{value} \) is a well-formed integral constant expression.

Semantics: Equivalent to

```
typedef if_< \text{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_
```
```cpp
{    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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Any type</td>
<td>A type to compute the sizeof for.</td>
</tr>
</tbody>
</table>

Expression semantics

For an arbitrary type x:

```cpp
typedef sizeof_<x>::type n;
```  

Return type: Integral Constant.

Precondition: x is a complete type.

Semantics: Equivalent to

```cpp
typedef size_t< sizeof(x) > n;
```  

Complexity

Constant time.

Example

```cpp
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

Revision Date: 15th November 2004
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.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>n::value_type</td>
<td>An integral type</td>
<td>Constant time.</td>
</tr>
<tr>
<td>n::value</td>
<td>An integral constant expression</td>
<td>Constant time.</td>
</tr>
<tr>
<td>n::type</td>
<td>Integral Constant</td>
<td>Constant time.</td>
</tr>
<tr>
<td>next&lt;n&gt;::type</td>
<td>Integral Constant</td>
<td>Constant time.</td>
</tr>
<tr>
<td>prior&lt;n&gt;::type</td>
<td>Integral Constant</td>
<td>Constant time.</td>
</tr>
<tr>
<td>n::value_type const c = n()</td>
<td></td>
<td>Constant time.</td>
</tr>
</tbody>
</table>

Expression semantics

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>n::value_type</td>
<td>A cv-unqualified type of n::value.</td>
</tr>
<tr>
<td>n::value</td>
<td>The value of the wrapped integral constant.</td>
</tr>
<tr>
<td>n::type</td>
<td>is_same&lt;n::type,n&gt;::value == true.</td>
</tr>
<tr>
<td>next&lt;n&gt;::type</td>
<td>An Integral Constant c of type n::value_type such that c::value == n::value + 1.</td>
</tr>
<tr>
<td>prior&lt;n&gt;::type</td>
<td>An Integral Constant c of type n::value_type such that c::value == n::value - 1.</td>
</tr>
<tr>
<td>n::value_type const c = n()</td>
<td>c == n::value.</td>
</tr>
</tbody>
</table>

Models

- bool
- int
5.2 Numeric Data Types

— `long_`
— `integral_c`

See also

Data Types, Integral Sequence Wrapper, `integral_c`

5.2 Numeric

5.2.1 bool_

Synopsis

```cpp
template<
    bool C
>
struct bool_
{
    // unspecified
    // ...
};

typedef bool_<true> true_;
typedef bool_<false> false_;
```

Description

A boolean Integral Constant wrapper.

Header

```cpp
#include <boost/mpl/bool.hpp>
```

Model of

Integral Constant

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>A boolean integral constant</td>
<td>A value to wrap.</td>
</tr>
</tbody>
</table>

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`:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bool_&lt;c&gt;</code></td>
<td>An Integral Constant <code>x</code> such that <code>x::value</code> == <code>c</code> and <code>x::value_type</code> is identical to <code>bool</code>.</td>
</tr>
</tbody>
</table>

Revision Date: 15th November 2004
Example

```cpp
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

```cpp
template<
    int N
>
struct int_
{
    // unspecified
    // ...
};
```

Description

An Integral Constant wrapper for int.

Header

```cpp
#include <boost/mpl/int.hpp>
```

Model of

Integral Constant

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>An integral constant</td>
<td>A value to wrap.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>int_&lt;c&gt;</td>
<td>An Integral Constant x such that x::value == c and x::value_type is identical to int.</td>
</tr>
</tbody>
</table>

Revision Date: 15th November 2004
Example

```cpp
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

```cpp
template<
    long N
>
struct long_
{
    // unspecified
    // ...
};
```

Description

An Integral Constant wrapper for long.

Header

```cpp
#include <boost/mpl/long.hpp>
```

Model of

Integral Constant

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>An integral constant</td>
<td>A value to wrap.</td>
</tr>
</tbody>
</table>

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:

Revision Date: 15th November 2004
<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>long_&lt;c&gt;</td>
<td>An Integral Constant $x$ such that $x::\text{value} == c$ and $x::\text{value_type}$ is identical to long.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>An integral constant</td>
<td>A value to wrap.</td>
</tr>
</tbody>
</table>

Revision Date: 15th November 2004
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:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>size_t&lt;c&gt;</td>
<td>An Integral Constant x such that x::value == c and x::value_type is identical to std::size_t.</td>
</tr>
</tbody>
</table>

Example

```cpp
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

```cpp
template<
   typename T, T N
>
struct integral_c
{
   // unspecified
   // ...
};
```

Description

A generic Integral Constant wrapper.

Header

```cpp
#include <boost/mpl/integral_c.hpp>
```

Model of

Integral Constant

Revision Date: 15th November 2004
Parameters
### 5.3 Miscellaneous Data Types

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>An integral type</td>
<td>Wrapper’s value type.</td>
</tr>
<tr>
<td>N</td>
<td>An integral constant</td>
<td>A value to wrap.</td>
</tr>
</tbody>
</table>

#### 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 \( n \):

<table>
<thead>
<tr>
<th>Expression</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{integral}_c&lt;t,c&gt; )</td>
<td>An Integral Constant ( x ) such that ( x::\text{value} == c ) and ( x::\text{value_type} ) is identical to ( t ).</td>
</tr>
</tbody>
</table>

#### Example

```cpp
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

### 5.3.1 pair

#### Synopsis

```cpp
template<
    typename T1,
    typename T2
>
struct pair
{
    typedef pair type;
    typedef T1 first;
    typedef T2 second;
};
```

#### Description

A transparent holder for two arbitrary types.

Revision Date: 15th November 2004
Header

```cpp
#include <boost/mpl/pair.hpp>
```

Example

Count a number of elements in the sequence together with a number of negative elements among these.

```cpp
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

```cpp
struct empty_base {};
```

Description

An empty base class. Inheritance from `empty_base` through the `inherit` metafunction is a no-op.

Header

```cpp
#include <boost/mpl/empty_base.hpp>
```

See also

Data Types, inherit, inherit_linearly, void_

Revision Date: 15th November 2004
5.3.3 **void_**

**Synopsis**

```cpp
struct void_
{
    typedef void_ type;
};

template< typename T > struct is_void;
```

**Description**

void_ is a generic type placeholder representing “nothing”.

**Header**

```cpp
#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

```cpp
#define BOOST_MPL_ASSERT( pred )
    unspecified token sequence /**/
```

Description

Generates a compilation error when the predicate `pred` holds false.

Header

```cpp
#include <boost/mpl/assert.hpp>
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pred</code></td>
<td>Boolean nullary Metafunction</td>
<td>A predicate to be asserted.</td>
</tr>
</tbody>
</table>

Expression semantics

For any boolean nullary Metafunction `pred`:

```cpp
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

```cpp
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

```cpp
#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

```cpp
#include <boost/mpl/assert.hpp>
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>condition</td>
<td>An integral constant expression</td>
<td>A condition to be asserted.</td>
</tr>
<tr>
<td>message</td>
<td>A legal identifier token</td>
<td>A custom message in a form of a legal C++ identifier token.</td>
</tr>
</tbody>
</table>
Parameter | Requirement | Description
--- | --- | ---
types | A legal function parameter list | A parenthesized 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`:

```cpp
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 parenthesized list of `t1, t2,... tn` types, and have a general form of:

```cpp
... ***********(...::message)***********)(t1, t2,... tn)... 
```

**Example**

```cpp
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

Revision Date: 15th November 2004
6.1 Asserts Macros

6.1.3 BOOST_MPL_ASSERT_NOT

Synopsis

```c
#define BOOST_MPL_ASSERT_NOT( pred )
    unspecified token sequence
/**/
```

Description

Generates a compilation error when predicate holds true.

Header

```c
#include <boost/mpl/assert.hpp>
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pred</td>
<td>Boolean nullary Metafunction</td>
<td>A predicate to be asserted to be false.</td>
</tr>
</tbody>
</table>

Expression semantics

For any boolean nullary Metafunction `pred`:

```c
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:

```c
... ************boost::mpl::not_< pred >::************ ...
```

Example

```c
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

Revision Date: 15th November 2004
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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>An integral constant</td>
<td>Left operand of the checked relation.</td>
</tr>
<tr>
<td>y</td>
<td>An integral constant</td>
<td>Right operand of the checked relation.</td>
</tr>
<tr>
<td>relation</td>
<td>A C++ operator token</td>
<td>An operator token for the relation being checked.</td>
</tr>
</tbody>
</table>

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;  

Revision Date: 15th November 2004
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 member x::name.

On the deficient compilers not capable of performing the detection, has_name<x>::value always returns false. A boolean configuration 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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>A legal identifier token</td>
<td>A name of the member being detected.</td>
</tr>
</tbody>
</table>

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

Revision Date: 15th November 2004
Example

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

// Defines a trait that checks if a type has a certain member
#define BOOST_MPL_HAS_XXX_TRAIT_DEF(trait, name, default_) \
    unspecified token sequence \
/**/

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> ));

#ifdef !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**

```cpp
#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`.

Revision Date: 15th November 2004
On the deficient compilers not capable of performing the detection, \texttt{trait\langle x\rangle::value} always returns a fallback value \texttt{default\_.} A boolean configuration macro, \texttt{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

\begin{verbatim}
#include <boost/mpl/has_xxx.hpp>
\end{verbatim}

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{trait}</td>
<td>A legal identifier token</td>
<td>A name of the metafunction to be generated.</td>
</tr>
<tr>
<td>\texttt{name}</td>
<td>A legal identifier token</td>
<td>A name of the member being detected.</td>
</tr>
<tr>
<td>\texttt{default_}</td>
<td>An boolean constant</td>
<td>A fallback value for the deficient compilers.</td>
</tr>
</tbody>
</table>

Expression semantics

For any legal C++ identifiers \texttt{trait} and \texttt{name}, boolean constant expression \texttt{c1}, boolean \texttt{Integral Constant} \texttt{c2}, and arbitrary type \texttt{x}:

\begin{verbatim}
BOOST_MPL_HAS_XXX_TRAIT_NAMED_DEF(trait, name, c1)
\end{verbatim}

**Precondition:** Appears at namespace scope.

**Return type:** None.

**Semantics:** Expands into an equivalent of the following class template definition
\begin{verbatim}
template< typename X, typename fallback = boost::mpl::bool_<c1> >
struct trait
{
    // unspecified
    // ...
};
\end{verbatim}

where \texttt{trait} is a boolean \texttt{Metafunction} with the following semantics:

\begin{verbatim}
typedef trait\langle x\rangle::type r;
\end{verbatim}

**Return type:** \texttt{Integral Constant}.

**Semantics:** If \texttt{BOOST\_MPL\_CFG\_NO\_HAS\_XXX} is defined, \texttt{r::value} == \texttt{c1}; otherwise, \texttt{r::value} == \texttt{true} if and only if \texttt{x} is a class type that has a nested type member \texttt{x::name}.

\begin{verbatim}
typedef trait\langle x, c2 \rangle::type r;
\end{verbatim}

**Return type:** \texttt{Integral Constant}.

**Semantics:** If \texttt{BOOST\_MPL\_CFG\_NO\_HAS\_XXX} is defined, \texttt{r::value} == \texttt{c2::value}; otherwise, equivalent to

\begin{verbatim}
typedef trait\langle x\rangle::type r;
\end{verbatim}

Revision Date: 15th November 2004
Example

```cpp
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

```cpp
// #define BOOST_MPL_CFG_NO_PREPROCESSED_HEADERS
```

Description

BOOST_MPL_CFG_NO_PREPROCESSED_HEADERS is a 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.
6.3 Configuration Macros

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

#ifndef !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

Revision Date: 15th November 2004


```cpp
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**

```cpp
#if !defined(BOOST_MPL_LIMIT_VECTOR_SIZE)
#define BOOST_MPL_LIMIT_VECTOR_SIZE
   implementation-defined integral constant
/**/
#elif !defined(BOOST_MPL_LIMIT_VECTOR_SIZE)
#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**

```cpp
#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**

```cpp
#if !defined(BOOST_MPL_LIMIT_LIST_SIZE)
```

Revision Date: 15th November 2004
6.3 Configuration Macros

# 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]

Revision Date: 15th November 2004
Example

```cpp
#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_SET_SIZE

### 6.3.7 BOOST_MPL_LIMIT_MAP_SIZE

**Synopsis**

```cpp
#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

```cpp
#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!
```

Revision Date: 15th November 2004
6.3.8 **BOOST_MPL_LIMIT_UNROLLING**

**Synopsis**

```c
#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 performance, 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**

```c
#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**

```c
#include <boost/mpl/aux_/lambda_support.hpp>
```

**Parameters**

---

Revision Date: 15th November 2004
### Expression semantics

For any integral constant \( n \), a Metafunction \( \text{fun} \), and arbitrary types \( A_1, \ldots, A_n \):

\[
\text{template< typename } A_1, \ldots, \text{ typename } A_n > \text{ struct } \text{fun} \\
\{ \\
    \text{// ...} \\
    \text{BOOST\_MPL\_AUX\_LAMBDA\_SUPPORT(} n, \text{fun, (} A_1, \ldots, A_n ) \text{)} \\
\};
\]

**Precondition:** Appears in \( \text{fun} \)’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 \( \text{fun} \) to participate in Lambda Expressions with the semantics described in this manual.

**Example**

\[
\text{template< typename } T, \text{ typename } U = \text{ int } > \text{ struct } f \\
\{ \\
    \text{typedef } T \text{ type}[\text{sizeof(} U\text{)}]; \\
    \text{BOOST\_MPL\_AUX\_LAMBDA\_SUPPORT(2, } f, (T, U)) \\
\};
\]

\[
\text{typedef apply1< } f<\text{char,}_1>, \text{long } >\text{::type } r; \\
\text{BOOST\_MPL\_ASSERT(} (\text{ is\_same< } r, \text{char[sizeof(} long\text{)] } > )));
\]

**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:

\[
\begin{align*}
\text{bind< quote3<if_>, _1,int,long >} \\
\text{bind< _1, bind< plus<> , int_<5>, _2 >} \\
\text{bind< times<> , int_<2>, int_<2> >}
\end{align*}
\]

and these are not:

\[
\begin{align*}
\text{if_< _1, bind< plus<> , int_<5>, _2>, _2 >} \\
\text{protect< bind< quote3<if_>, _1,int,long > >} \\
\text{_2}
\end{align*}
\]
Chapter 8  Categorized Index

8.1  Concepts

- Associative Sequence
- Back Extensible Sequence
- Bidirectional Iterator
- Bidirectional Sequence
- Extensible Associative Sequence
- Extensible Sequence
- Forward Iterator
- Forward Sequence
- Front Extensible Sequence
- Inserter
- Integral Constant
- Integral Sequence Wrapper
- Lambda Expression
- Metafunction
- Metafunction Class
- Numeric Metafunction
- Placeholder Expression
- Random Access Iterator
- Random Access Sequence
- Reversible Algorithm
- Tag Dispatched Metafunction
- Trivial Metafunction
- Variadic Sequence

8.2  Components

- BOOST_MPL_ASSERT
- BOOST_MPL_ASSERT_MSG
- BOOST_MPL_ASSERT_NOT
— BOOST_MPL_ASSERT_RELATION
— BOOST_MPL_AUX_LAMBDA_SUPPORT
— BOOST_MPL_CFG_NO_HAS_XXX
— BOOST_MPL_CFG_NO_PREPROCESSED_HEADERS
— BOOST_MPL_HAS_XXX_TRAIT_DEF
— BOOST_MPL_HAS_XXX_TRAIT_NAMED_DEF
— BOOST_MPL_LIMIT_LIST_SIZE
— BOOST_MPL_LIMIT_MAP_SIZE
— BOOST_MPL_LIMIT_METAFUNCTION_ARITY
— BOOST_MPL_LIMIT_SET_SIZE
— BOOST_MPL_LIMIT_UNROLLING
— BOOST_MPL_LIMIT_VECTOR_SIZE
— _1, _2, _3,...
— accumulate
— advance
— always
— and_
— apply
— apply_wrap
— arg
— at
— at_c
— back
— back_inserter
— begin
— bind
— bitand_
— bitor_
— bitxor_
— bool_
— clear
— contains
— copy
— copy_if
— count

Revision Date: 15th November 2004
— count_if
— deque
— deref
— distance
— divides
— empty
— empty_base
— empty_sequence
— end
— equal
— equal_to
— erase
— erase_key
— eval_if
— eval_if_c
— filter_view
— find
— find_if
— fold
— front
— front_inserter
— greater
— greater_equal
— has_key
— identity
— if_
— if_c
— inherit
— inherit_linearly
— insert
— insert_range
— inserter
— int_
— integral_c
— is_sequence
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
— quote
— range_c
— remove
— remove_if
— replace
— replace_if
— reverse
— reverse_copy
— reverse_copy_if
— reverse_fold
— reverse_iter_fold
— reverse_partition
— reverse_remove
— reverse_remove_if
— reverse_replace
— reverse_replace_if
— reverse_stable_partition
— reverse_transform
— reverse_unique
— sequence_tag
— set
— set_c
— shift_left
— shift_right
— single_view
— size
— size_t
— sizeof_
— sort
— stable_partition
— times
— transform
— transform_view
— unique
— unpack_args
— 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


[PRE] Vesa Karvonen, Paul Mensonides, The Boost Preprocessor Metaprogramming library