Boost Exam Reference

Felix Morgner

January 12, 2015

Contents

Chapter 1

Boost.Operators

in `boost/operators.hpp`

1.1 boost::less_than_comparable

1.1.1 Synopsis

```
template<typename OperandType>
struct less_than_comparable;
```

1.1.2 Description

The class type `boost::less_than_comparable<PriOp>` simplifies creating comparable types by providing the following `operator` functions:

- `operator>(PriOp const &lhs, PriOp const &rhs)` - greater than
- `operator>=(PriOp const &lhs, PriOp const &rhs)` - greater than or equal
- `operator<=(PriOp const &lhs, PriOp const &rhs)` - less than or equal

These functions are implemented via `friend` functions and are not part of the user defined type. The type that inherits from `boost::less_than_comparable<PriOp>` only needs to supply one of these:

- `operator<(PriOp const &rhs) const`
- `operator<(PriOp const &lhs, PriOp const &rhs)`

functions.

1.1.3 Example

```
#include "boost/operators.hpp"
#include <iostream>

struct Apple : boost::less_than_comparable<Apple>
  {
  explicit Apple(int size = int{}) : m_nSize{size} {}

  bool operator <(Apple const &rhs) const { return m_nSize < rhs.m_nSize; }

  private:
    int m_nSize{};
  };

int main()
  {
  auto a1 = Apple{5};
  auto a2 = Apple{0};
  auto a3 = Apple{1};
  auto a4 = Apple{1};

  std::cout << std::boolalpha;

  std::cout << "a1 < a2:  " << (a1 < a2) << "\n";
  std::cout << "a2 > a3:  " << (a2 > a3) << "\n";
  std::cout << "a3 <= a4: " << (a3 <= a4) << "\n";
  std::cout << "a4 >= a1: " << (a4 >= a1) << "\n";
  }
```

1.1.4 Example output

```
a1 < a2:  false
a2 > a3:  false
a3 <= a4: true
a4 >= a1: false
```

1.2 boost::equality_comparable

1.2.1 Synopsis

```
template<typename OperandType>
struct equality_comparable;
```

1.2.2 Description

The class type `boost::equality_comparable<PriOp>` simplifies creating comparable types by providing the following `operator` function:

- `operator!=(PriOp const &lhs, PriOp const &rhs)` - not equal

These functions are implemented via `friend` functions and are not part of the user defined type. The type that inherits from `boost::equality_comparable<PriOp>` only needs to supply one of these:

- `operator==(PriOp const &rhs) const`
- `operator==(PriOp const &lhs, PriOp const &rhs)`

functions.

1.2.3 Example

```
#include "boost/operators.hpp"
#include <iostream>

struct Banana : boost::equality_comparable<Banana>
  {
  explicit Banana(int size = int{}) : m_nSize{size} {}

  bool operator ==(Banana const &rhs) const { return m_nSize == rhs.m_nSize; }

  private:
    int m_nSize{};
  };

int main()
  {
  auto b1 = Banana{5};
  auto b2 = Banana{5};

  std::cout << std::boolalpha;

  std::cout << "b1 == b2:  " << (b1 == b2) << "\n";
  std::cout << "b2 != b2:  " << (b1 != b2) << "\n";
  }
```

1.2.4 Example output

```
b1 == b2:  true
b2 != b2:  false
```

1.3 boost::incrementable

1.3.1 Synopsis

```
template<typename OperandType>
struct incrementable;
```

1.3.2 Description

The class type `boost::incrementable<PriOp>` simplifies creating incrementable types by providing the following `operator` function:

- `operator++(PriOp &lhs, int)` - postfix increment

This function is implemented via a `friend` function and is not part of the user defined type. The type that inherits from `boost::incrementable<PriOp>` only needs to supply one of these:

- `operator++()`
- `operator++(PriOp &lhs)`

functions.

1.3.3 Example

```
#include "boost/operators.hpp"
#include <iostream>

struct Cranberry : boost::incrementable<Cranberry>
  {
  explicit Cranberry(int size = int{}) : size{size} {}

  Cranberry operator ++() { size++; return *this; }

  int size{};
  };

int main()
  {
  auto c1 = Cranberry{5};

  std::cout << std::boolalpha;

  std::cout << "c1++ : " << (c1++).size << "\n";
  std::cout << "++c1 : " << (++c1).size << "\n";
  }
```

1.3.4 Example output

```
c1++ : 5
++c1 : 7
```

1.4 boost::decrementable

1.4.1 Synopsis

```
template<typename OperandType>
struct decrementable;
```

1.4.2 Description

The class type `boost::decrementable<PriOp>` simplifies creating decrementable types by providing the following `operator` function:

- `operator--(PriOp &lhs, int)` - postfix decrement

This function is implemented via a `friend` function and is not part of the user defined type. The type that inherits from `boost::decrementable<PriOp>` only needs to supply one of these:

- `operator--()`
- `operator--(PriOp &lhs)`

functions.

1.4.3 Example

```
#include "boost/operators.hpp"
#include <iostream>

struct Date : boost::decrementable<Date>
  {
  explicit Date(int size = int{}) : size{size} {}

  Date operator --() { size--; return *this; }

  int size{};
  };

int main()
  {
  auto d1 = Date{5};

  std::cout << std::boolalpha;

  std::cout << "d1-- : " << (d1--).size << "\n";
  std::cout << "--d1 : " << (--d1).size << "\n";
  }
```

1.4.4 Example output

```
d1-- : 5
--d1 : 3
```

Chapter 2

Boost.IteratorHelpers

in `boost/operators.hpp`

2.1 boost::output_iterator_helper

2.1.1 Synopsis

```
template<typename OperandType>
struct output_iterator_helper;
```

2.1.2 Description

The class type `boost::output_iterator_helper<PriOp>` simplifies creating output iterators by providing the following `operator` functions:

- `operator*()` - dereference
- `operator++()` - prefix increment

The `operator*()` function is implemented as a `member` function, while `operator++` is implemented as a `friend` function. `operator*()` is inherited by the user defined type. The type that inherits from `boost::output_iterator_helper<PriOp>` only needs to supply the:

- `operator=(ValueType const &)`

function. `ValueType` may be a template parameter of the user defined type, or be replaced with some other type.

2.1.3 Example

```
#include "boost/operators.hpp"
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

struct Elderberry : boost::output_iterator_helper<Elderberry>
  {
  explicit Elderberry(std::ostream &out = std::cout) : m_roOut(out) {}

  Elderberry & operator=(std::string const &elem)
    {
    m_roOut << elem << '\n';
    return *this;
    }

  private:
    std::ostream &m_roOut;
  };

int main()
  {
  auto v = std::vector<std::string>{"Spam", "Eggs", "Knights"};

  copy(v.cbegin(), v.cend(), Elderberry{});
  }
```

2.1.4 Example output

```
Spam
Eggs
Knights
```

2.2 boost::input_iterator_helper

2.2.1 Synopsis

```
template<typename OperandType, typename ValueType>
struct input_iterator_helper;
```

2.2.2 Description

The class type `boost::input_iterator_helper<PriOp, ValueType>` simplifies creating input iterators by providing the following `operator` functions:

- `operator->()` - dereference
- `operator++(PriOp &, int)` - postfix increment
- `operator!=(PriOp & const, PriOp & const)` - not equal

`operator->()` is implemented via a `member` function while the others are implemented via `friend` functions. The `member` function is inherited by the user defined type. The type that inherits from `boost::input_iterator_helper<PriOp, ValueType>` only needs to supply these:

- `operator==(PriOp const &)` or `operator==(PriOp const & lhs, PriOp const & rhs)`
- `operator++()` or `operator++(PriOp & lhs)`
- `operator*()`

functions.

2.2.3 Example

```
#include "boost/operators.hpp"
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <iterator>

struct Feijoa : boost::input_iterator_helper<Feijoa, int>
  {
  explicit Feijoa(value_type count = value_type{}) : m_nCount(count) {}

  value_type operator*() { return m_nCount; }
  Feijoa operator++() { ++m_nCount; return *this; }
  bool operator==(Feijoa const &rhs) const { return m_nCount == rhs.m_nCount; }

  private:
    value_type m_nCount;
  };

int main()
  {
  copy(Feijoa{}, Feijoa{4}, std::ostream_iterator<int>{std::cout, "\n"});
  }
```

2.2.4 Example output

```
0
1
2
3
```

2.3 boost::forward_iterator_helper

2.3.1 Synopsis

```
template<typename OperandType, typename ValueType>
struct forward_iterator_helper;
```

2.3.2 Description

The class type `boost::forward_iterator_helper<PriOp, ValueType>` simplifies creating forward iterators by providing the following `operator` functions:

- `operator->()` - dereference
- `operator++(PriOp &, int)` - postfix increment
- `operator!=(PriOp & const, PriOp & const)` - not equal

`operator->()` is implemented via a `member` function while the others are implemented via `friend` functions. The `member` function is inherited by the user defined type. The type that inherits from `boost::forward_iterator_helper<PriOp, ValueType>` only needs to supply these:

- `operator==(PriOp const &)` or `operator==(PriOp const & lhs, PriOp const & rhs)`
- `operator++()` or `operator++(PriOp & lhs)`
- `operator*()`

functions.
Forward iterators guarantee validity when used in multipass algorithms.

2.3.3 Example

```
#include "boost/operators.hpp"
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <iterator>

struct Grape : boost::forward_iterator_helper<Grape, int>
  {
  explicit Grape(value_type count = value_type{}) : m_nCount(count) {}

  value_type operator*() { return m_nCount; }
  Grape operator++() { ++m_nCount; return *this; }
  bool operator==(Grape const &rhs) const { return m_nCount == rhs.m_nCount; }

  private:
    value_type m_nCount;
  };

int main()
  {
  copy(Grape{}, Grape{4}, std::ostream_iterator<int>{std::cout, "\n"});
  }
```

2.3.4 Example output

```
0
1
2
3
```

2.4 boost::bidirectional_iterator_helper

2.4.1 Synopsis

```
template<typename OperandType, typename ValueType>
struct bidirectional_iterator_helper;
```

2.4.2 Description

The class type `boost::bidirectional_iterator_helper<PriOp, ValueType>` simplifies creating bidirectional iterators by providing the following `operator` functions:

- `operator->()` - dereference
- `operator++(PriOp &, int)` - postfix increment
- `operator--(PriOp &, int)` - postfix decrement
- `operator!=(PriOp & const, PriOp & const)` - not equal

`operator->()` is implemented via a `member` function while the others are implemented via `friend` functions. The `member` function is inherited by the user defined type. The type that inherits from `boost::bidirectional_iterator_helper<PriOp, ValueType>` only needs to supply these:

- `operator==(PriOp const &)` or `operator==(PriOp const & lhs, PriOp const & rhs)`
- `operator++()` or `operator++(PriOp & lhs)`
- `operator--()` or `operator--(PriOp & lhs)`
- `operator*()`

functions.
Bidirectional iterators give the same guarantees as forward iterators, while additionally providing the ability to iterate in reverse.

2.4.3 Example

```
#include "boost/operators.hpp"
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <iterator>

struct HoneydewMelon : boost::bidirectional_iterator_helper<HoneydewMelon, int>
  {
  explicit HoneydewMelon(value_type count = value_type{}) : m_nCount(count) {}

  value_type operator*() { return m_nCount; }
  HoneydewMelon operator++() { ++m_nCount; return *this; }
  HoneydewMelon operator--() { --m_nCount; return *this; }
  bool operator==(HoneydewMelon const &rhs) const { return m_nCount == rhs.m_nCount; }

  private:
    value_type m_nCount;
  };

int main()
  {
  std::cout << "forward: ";
  copy(HoneydewMelon{}, HoneydewMelon{4}, std::ostream_iterator<int>{std::cout, " "});
  std::cout << "\nreverse: ";
  reverse_copy(HoneydewMelon{}, HoneydewMelon{4}, std::ostream_iterator<int>{std::cout, " "});
  }
```

2.4.4 Example output

```
forward: 0 1 2 3
reverse: 3 2 1 0
```

Chapter 3

Boost.Iterators

in `boost/iterator/filter_iterator.hpp`
or `boost/iterator/counting_iterator.hpp`
or `boost/iterator/transform_iterator.hpp`

3.1 boost::filter_iterator

3.1.1 Synopsis

```
template<typename Predicate, typename Iterator>
struct filter_iterator;
```

3.1.2 Description

The class type `boost::filter_iterator<Predicate, Iterator>` creates a filtered view of an iterator range. Values for which the `Predicate` returns **false** are skipped. An instance of `boost::filter_iterator<Predicate, Iterator>` is most often constructed using of these constructors:

- `filter_iterator(Predicate f, Iterator begin, Iterator end = Iterator{})`
- `filter_iterator(Iterator begin, Iterator end = Iterator{})`

To be able to use the first ctor, Predicate needs to be **DefaultConstructible** .
An alternative way to construct a `boost::filter_iterator<Predicate, Iterator>` is to use
`boost::make_filter_iterator(Predicate f, Iterator begin, Iterator end = Iterator{})`
or
`boost::make_filter_iterator(Iterator begin, Iterator end = Iterator{})` .
When using the second version, one has to make sure that Predicate is **DefaultConstructible** and the resulting type has to be specified.

3.1.3 Example

```
#include "boost/iterator/filter_iterator.hpp"
#include <iostream>
#include <vector>
#include <iterator>
#include <algorithm>

struct Grape
  {
  bool operator()(int num) { return num % 2; }
  };

int main()
  {
  auto v = std::vector<int>{1, 2, 3, 4, 3, 6, 7};

  using Filt = boost::filter_iterator<Grape, std::vector<int>::const_iterator>;

  std::cout << "via constructor: ";
  auto beg1 = Filt{Grape{}, v.cbegin(), v.cend()};
  auto end1 = Filt{Grape{}, v.cend(), v.cend()};

  copy(beg1, end1, std::ostream_iterator<int>{std::cout});
  std::cout << "\n";

  std::cout << "via factory:     ";
  auto beg2 = boost::make_filter_iterator(Grape{}, v.cbegin(), v.cend());
  auto end2 = boost::make_filter_iterator(Grape{}, v.cend(), v.cend());

  copy(beg2, end2, std::ostream_iterator<int>{std::cout});
  std::cout << "\n";
  }
```

3.1.4 Example output

```
via constructor: 1337
via factory:     1337
```

3.2 boost::counting_iterator

3.2.1 Synopsis

```
template<typename Incrementable>
struct counting_iterator;
```

3.2.2 Description

The class type `boost::counting_iterator<Incrementable>` creates a "lazy sequence" - an interator that produces sequencial values - over some type `Incrementable`. `Incrementable` needs to be **CopyConstructible** and **Assignable** . `boost::counting_iterator<Incrementable>` is usualy constructed via one of these ctors:

- `counting_iterator(Incrementable x)`
- `counting_iterator(counting_iterator const &rhs)`

In order to use the second ctor, the template argument `Incrementable` of the supplied object needs to be implicitly convertible to `Incrementable` of the requested instatiation of `boost::counting_iterator<Incrementable>` .
An alternative way to construct a `boost::counting_iterator<Incrementable>` is to use `boost::make_counting_iterator(Incrementable x)` .

3.2.3 Example

```
#include "boost/iterator/counting_iterator.hpp"
#include <iostream>
#include <vector>
#include <iterator>
#include <algorithm>

int main()
  {
  using Count = boost::counting_iterator<int>;

  std::cout << "via constructor: ";
  auto beg1 = Count{0};
  auto end1 = Count{7};

  copy(beg1, end1, std::ostream_iterator<int>{std::cout, " "});
  std::cout << "\n";

  std::cout << "via factory:     ";
  auto beg2 = boost::make_counting_iterator(7);
  auto end2 = boost::make_counting_iterator(14);

  copy(beg2, end2, std::ostream_iterator<int>{std::cout, " "});
  std::cout << "\n";
  }
```

3.2.4 Example output

```
via constructor: 0 1 2 3 4 5 6
via factory:     7 8 9 10 11 12 13
```

3.3 boost::transform_iterator

3.3.1 Synopsis

```
template<typename Function, typename Iterator>
struct transform_iterator;
```

3.3.2 Description

The class type `boost::transform_iterator<Function, Iterator>` creates an iterator that, when dereferenced, derefences the adapted iterator and applies the user provided function to that value.
`boost::transform_iterator<Function, Iterator>` is constructed via
`transform_iterator(Iterator const &it, Function func)`. The function must conform to **UnaryFunction** and the expression `func(*it)` must be valid, where:

- `func` is a `const` object of type `Function`
- `it` is an object of type `Iterator`
- the return value of `func(*it)` is of type `iterator_traits<Iterator>::reference`

An alternative way to construct a `boost::transform_iterator<Function, Iterator>` is to use
`boost::make_transform_iterator(Iterator it, Function func)`
or
`boost::make_transform_iterator(Iterator it)` .
The latter version needs `Function` to be **DefaultConstructible** .

3.3.3 Example

```
#include "boost/iterator/transform_iterator.hpp"
#include <iostream>
#include <vector>
#include <iterator>
#include <algorithm>

struct HoneydewMelon
  {
  int operator()(int const &num) const { return num * 3; }
  };

int main()
  {
  auto v = std::vector<int>{1, 2, 3, 4, 5, 6, 7};

  using Trans = boost::transform_iterator<HoneydewMelon, std::vector<int>::const_iterator>;

  std::cout << "via constructor: ";
  auto beg1 = Trans{v.cbegin(), HoneydewMelon{}};
  auto end1 = Trans{v.cend(), HoneydewMelon{}};

  copy(beg1, end1, std::ostream_iterator<int>{std::cout, " "});
  std::cout << "\n";

  std::cout << "via factory:     ";
  auto beg2 = boost::make_transform_iterator(v.cbegin(), HoneydewMelon{});
  auto end2 = boost::make_transform_iterator(v.cend(), HoneydewMelon{});

  copy(beg2, end2, std::ostream_iterator<int>{std::cout, " "});
  std::cout << "\n";
  }
```

3.3.4 Example output

```
via constructor: 3 6 9 12 15 18 21
via factory:     3 6 9 12 15 18 21
```