# The Boost Iterator Library

## User defined Iterators made easy

Thomas Witt
Zephyr Associates, Inc.
witt@styleadvisor.com

# Outline

- Introduction
- Motivation
- New Iterator Concepts
- Iterator Facade
- Iterator Adaptor
- Specialized Adaptors
- When you are a hammer ...

# Boost Iterator Library

- Extension to C++98
- New concepts that describe iterator requirements
- Framework for building iterators
- Set of ready made adaptors

# History

- Iterator adaptor idea first mentioned in 1998
- Iterator adaptors started by Dave Abrahams in 2000
- First Boost version submitted by Dave Abrahams and Jeremy Siek
- The initial version used a policy based design
- Complete rewrite in 2003 adding iterator facade and integrating the new iterator categories

# Outline

- Introduction
- Motivation
- New Iterator Concepts
- Iterator Facade
- Iterator Adaptor
- Specialized Adaptors
- When you are a hammer ...

# Motivation

- STL interoperability woes

```
map<int ,string> dictionary;
...
vector<int> keys(dictionary.begin(), dictionary.end()); // !! Error
    Ooops!

vector<int*> range;
int sum(accumulate(range.begin(), range.end(), 0)); // !! Ooops
```

- Iterators are hard to get right
- If you get it wrong you are likely to not notice it, yet

# Motivation

- Writing iterators is tedious
- Fat Interface
  - 5 typedef names
  - 11 member operators
  - 9 non-member operators
- Prefix/postfix operators anybody?

# Motivation

- Derivation is not an option

```
struct MyIterator :
  public vector<int>::iterator
{
  typedef vector<int>::iterator super_t;
  MyIterator(super_t i) : super_t(i) {}
};
MyIterator i1(vec.begin());
MyIterator i2(i1++);          // !! Error, Ooops!
```

# Motivation

- operator[]

```
vector<int> vec(10);
vector<int>::iterator it(vec.begin());
it[5] = 10;
```

- Iterators that hold the referenced object internally need to return a proxy

```
T& iterator::operator[](difference_type n) const {
    return *(this + n);  // !! Maybe reference to temporary
}
```

# Motivation

- The reference type of input iterators is not required to be a reference
- Nevertheless member access through operator-> must be possible
- operator-> might need a proxy as well

# Outline

- Introduction
- Motivation
- New Iterator Concepts
- Iterator Facade
- Iterator Adaptor
- Specialized Adaptors
- When you are a hammer ...

# New Iterator Concepts

- C++98 iterator categories bind together two orthogonal concepts
  - Access
  - Positioning
- Many useful iterators cannot be sufficiently categorized using these categories
  - Transform iterators
  - vector<bool> anybody?
- Iterators are mis-categorized
- Algorithm requirements are overly strict

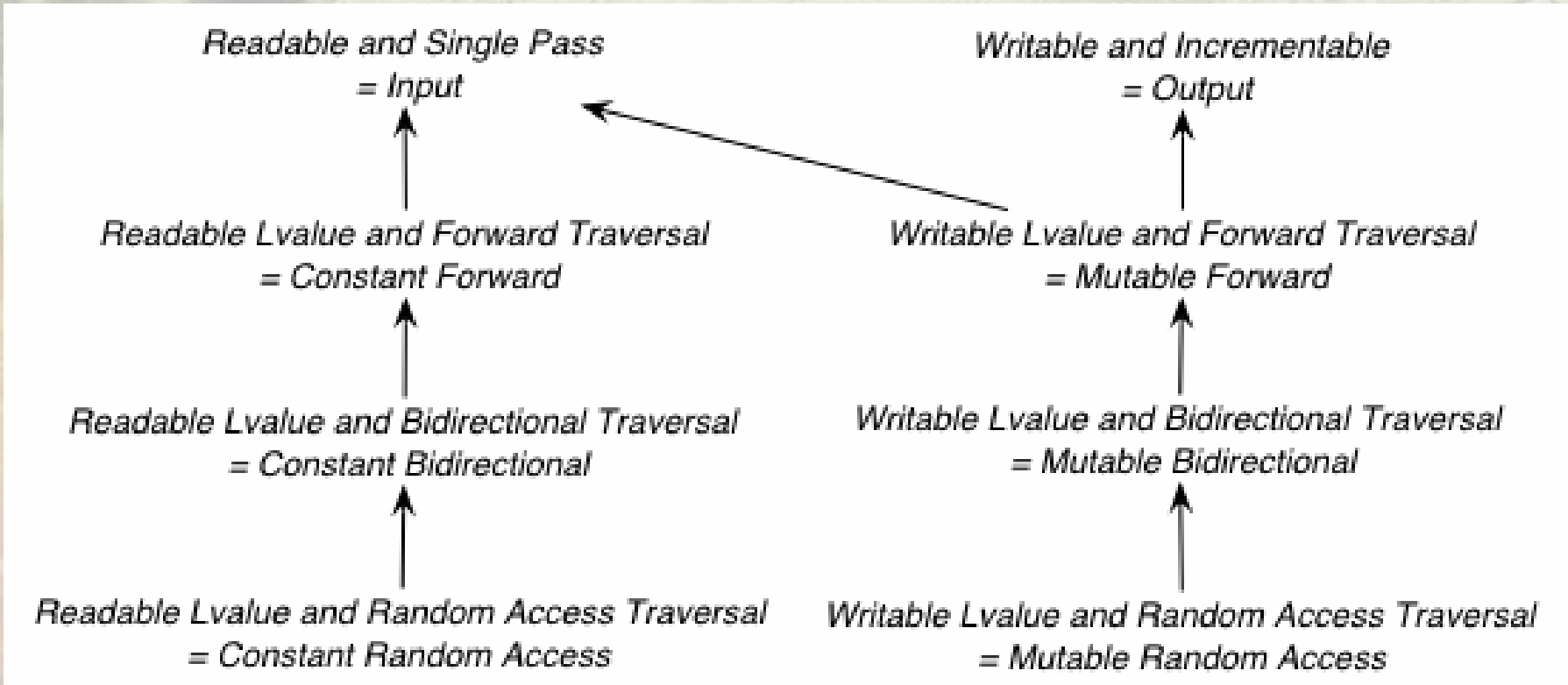
# Traversal Concepts

- Five traversal concepts
  - Incrementable Iterator
  - Single Pass Iterator
  - Forward Traversal Iterator
  - Bidirectional Traversal Iterator
  - Random Access Traversale Iterator
- Each concept is a refinement for the preceding concept
- New tags for use as iterator_category
- Backward compatible iterator_traversal trait

# Access Concepts

- Four access concepts
  - Readable
  - Writable
  - Swappable
  - Lvalue
- No refinement relationship
- No dispatch tags
- Backward compatible is_readable trait

# Relationship to C++98 Categories



# Iterator Interoperability

- Iterator types are (one-way) convertible

  iterator it;

  const_iterator cit(it);

- The available comparison operators work with mixed type arguments

  bool equal(it == cit);

- C++98 has no notion of iterator interoperability
- In some early implementations of the standard library iterator and const_iterator types were not always interoperable

# Outline

- Introduction
- Motivation
- New Iterator Concepts
- Iterator Facade
- Iterator Adaptor
- Specialized Adaptors
- When you are a hammer ...

# Iterator Facade

- Base class template that implements the full interface of C++98 iterators
- User defined iterators derive from iterator_facade, thus inheriting the iterator interface.
- iterator_facade takes care of the nitty gritty details

# Iterator Facade Example

```
class SerIterator
 : public iterator_facade< SerIterator, int, forward_traversal_tag >
{
  friend class iterator_core_access;
public:
            SerIterator()                          : m_i(0)     {}
  explicit SerIterator(int start)                  : m_i(start) {}
private:
  int  dereference()                               { return m_i; }
  void increment()                                 { ++m_i; }
  bool equal(NumberIterator const& rhs) { return m_i == rhs.m_i; }
  int m_i;
};
```

# Curiously Recurring Template Pattern

```
class SerIterator
 : public iterator_facade< SerIterator, int, forward_traversal_tag >
{
  friend class iterator_core_access;
public:
            SerIterator()                          : m_i(0)     {}
  explicit SerIterator(int start)                  : m_i(start) {}
private:
  int  dereference()                               { return m_i; }
  void increment()                                 { ++m_i; }
  bool equal(NumberIterator const& rhs) { return m_i == rhs.m_i; }
  int m_i;
};
```

# Curiously Recurring Template Pattern

- Making the derived class's type available to the base class

```
template <class Derived>
class base {
  Derived& derived() {
    return static_cast<Derived&>(*this);  }
};
```

- Providing the correct return type for unary operators
- User defined constructors
- Calling member functions of the derived class

# Iterator Core Interface

```
class SerIterator
 : public iterator_facade< SerIterator, int, forward_traversal_tag >
{
  friend class iterator_core_access;
public:
            SerIterator()                       : m_i(0)     {}
  explicit SerIterator(int start)               : m_i(start) {}
private:
  int   dereference()                           { return m_i; }
  void increment()                              { ++m_i; }
  bool equal(NumberIterator const& rhs) { return m_i == rhs.m_i; }
  int m_i;
};
```

# Iterator Core Interface

- Iterator behavior can be specified by a small number of core operations
- The iterator_facade interface implementation forwards to the core operations
- Core operations are implemented in the derived class
- No virtual function calls
- No policy class
- Completely stack based
- Full inlining possible

# Iterator Core Interface

| Core Operation | Traversal Category |
|---|---|
| reference dereference() const | Incrementable |
| void increment() | Incrementable |
| bool equal(iterator it) const | Single Pass |
| void decrement() | Bidirectional |
| void advance(difference_type n) | Random Access |
| difference_type distance_to(iterator it) const | Random Access |

# The “hidden” Interface

```
class SerIterator
 : public iterator_facade< SerIterator, int, forward_traversal_tag >
{
  friend class iterator_core_access;
public:
            SerIterator()                          : m_i(0)     {}
  explicit SerIterator(int start)                  : m_i(start) {}
private:
  int  dereference()                               { return m_i; }
  void increment()                                 { ++m_i; }
  bool equal(NumberIterator const& rhs) { return m_i == rhs.m_i; }
  int m_i;
};
```

# The “hidden” Interface

```
struct jekyll {
  friend class hyde;
private:
  bool equal(jekyll const& rhs);
};

struct hyde {
  static bool equal(jekyll const& lhs, jekyll const& rhs) {
    return lhs.equal(rhs);
  }
};

Bool operator==(jekyll const& lhs, jekyll const& rhs) {
  return hyde::equal(lhs, rhs);
}
```

# The “hidden” Interface

- iterator_core_access is used to grant friendship to a group of classes and free functions
- The iterators public interface is free from implementation artifacts
- Exposing the core interface is safe
  - Class invariants are maintained by the core interface

# Iterator Interoperability

- Iterators derived from iterator_facade specializations are interoperable if there is a (one-way) conversion between their types
- Interoperability using iterator_facade is non-intrusive
- By default interoperability is implemented by converting to the common convertible-to type
- Users can provide overloads for core operations in order to avoid type conversion

# Outline

- Introduction
- Motivation
- New Iterator Concepts
- Iterator Facade
- Iterator Adaptor
- Specialized Adaptors
- When you are a hammer ...

# Iterator Adaptor

- Base class for specialized adaptors
  - Functionality of the base iterator can be largely reused
- Transparent iterator adaptor
  - Operations are forwarded to a base iterator
- Provides extensive defaults

```
template <class Derived,
          class Base,
          class Value          = use_default,
          class Category       = use_default,
          class Reference      = use_default,
          class Difference     = use_default >
class iterator_adaptor;
```

# Iterator Adaptor Example

```
template <class Iterator>
class ReverseIterator
 : public iterator_adaptor< ReverseIterator, Iterator >
{
  typedef iterator_adaptor< ReverseIterator, Iterator > adaptor;
  friend class iterator_core_access;
public:
            ReverseIterator()                  {}
  explicit ReverseIterator(Iterator it)       : adaptor(it) {}
private:
  typename adaptor::reference  dereference() const { return *--base(); }
  void increment()     { --m_i; }
  void decrement()     { ++m_i; }
};
```

# Iterator Adaptor Example

- This is as good as std::reverse_iterator
- There are problems though
  - ReverseIterator<C::iterator> and ReverseIterator<C::const_iterator> are not convertible
  - ReverseIterator<C::iterator> and ReverseIterator<C::const_iterator> are not interoperable
- First step add the conversion

# Iterator Adaptor Example

```
template <class Iterator>
class ReverseIterator
 : public iterator_adaptor< ReverseIterator, Iterator >
{
  ...
public:
            ReverseIterator()                    {}
  explicit ReverseIterator(Iterator it)       : adaptor(it) {}
  template <class OtherIterator>
  ReverseIterator(ReverseIterator<OtherIterator> const& other)
    : adaptor(other) {}
  ...
};
```

# Iterator Adaptor Example

- ReverseIterator<C::iterator> and ReverseIterator<C::const_iterator> are now convertible
- There are problems though
  - std::tr1::is_convertible<
    ReverseIterator<X>,
    ReverseIterator<Y>
    >::value == true; // !!
  - But instantiation fails
- We need constrained genericity

# Iterator Adaptor Example

```
template <class Iterator>
class ReverseIterator
 : public iterator_adaptor< ReverseIterator, Iterator >
{
  ...
  template <class OtherIterator>
  ReverseIterator(ReverseIterator<OtherIterator> const& other,
                  typename enable_if<
                    is_convertible< OtherIterator, Iterator>,
                    void*
                  >::type = 0) : adaptor(other) {}

  ...
};
```

# Iterator Adaptor Example

- This is std::reverse_iterator as it should be
- ReverseIterator<C::iterator> and ReverseIterator<C::const_iterator> are convertible
- ReverseIterator<C::iterator> and ReverseIterator<C::const_iterator> are interoperable
- Core operation overloads can be added to increase efficiency
- The Boost Iterator Library contains a fixed reverse_iterator adaptor

# Outline

- Introduction
- Motivation
- New Iterator Concepts
- Iterator Facade
- Iterator Adaptor
- Specialized Adaptors
- When you are a hammer ...

# Specialized Adaptors

- counting_iterator
  - An iterator over a sequence of consecutive values. Implements a "lazy sequence"
- filter_iterator
  - An iterator over the subset of elements of some sequence which satisfy a given predicate
- Function_output_iterator
  - Output iterator wrapping a unary function object
- indirect_iterator
  - An iterator over the objects *pointed-to* by the elements of some sequence

# Specialized Adaptors

- permutation_iterator
  - An iterator over the elements of some random-access sequence, rearranged according to some sequence of integer indices
- reverse_iterator
  - An iterator which traverses the elements of some bidirectional sequence in reverse. Corrects many of the shortcomings of C++98's std::reverse_iterator
- shared_container_iterator
  - An iterator over elements of a container whose lifetime is maintained by a shared_ptr stored in the iterator

# Specialized Adaptors

- transform_iterator
  - An iterator over elements which are the result of applying some functional transformation to the elements of an underlying sequence
- zip_iterator
  - An iterator over tuples of the elements at corresponding positions of heterogeneous underlying iterators

# When you are a hammer ...

- Sometimes there are better solutions than iterator adaptors

```
struct string_to_int;
vector<string>          strings;
vector<int>             numbers;
...
// Is this really the right way ?
copy(make_transform_iterator(strings.begin(), string_to_int()),
        make_transform_iterator(strings.end(), string_to_int()),
        back_inserter(numbers));
// No!
transform(strings.begin(), strings.end(),
            back_inserter(numbers), string_to_int());
// But
vector<int> more_numbers(make_transform_iterator(strings.begin(),
                                                 string_to_int()),
                         make_transform_iterator(strings.end(),
                                                 string_to_int()));
```

# Summary

- The Boost Iterator Library provides an easy way to create or adapt iterators
- Iterators created using the Boost Iterator Library can be as efficient as handcrafted iterator types
- The interface presented is part of Boost 1.31.0
- A subset of the Boost Iterator Library is proposed for inclusion in the standard library

# References


- “The Boost Iterator Library” www.boost.org

- David Abrahams, Jeremy Siek and Thomas Witt. “Iterator Facade and Adaptor,” ISO/ANSI C++ Standards committee paper (ISO/IEC JTC1/SC22/WG21 paper N1641, ANSI/NCITS J16 paper 04-0081)

- David Abrahams, Jeremy Siek and Thomas Witt. “New Iterator Concepts,” ISO/ANSI C++ Standards committee paper (ISO/IEC JTC1/SC22/WG21 paper N1640, ANSI/NCITS J16 paper 04-0080)