Evolving a C++ Library to C++0x Concepts

Douglas Gregor <dgregor@osl.iu.edu>
Open Systems Laboratory
Indiana University

Concepts Recap

```
auto concept LessThanComparable<typename T> {
  bool operator<(T, T);
}

template<typename T>
  requires LessThanComparable<T>
  const T& min(const T& x, const T& y) {
    return x < y? x : y;
  }

concept_map LessThanComparable<int> { }

int result = min(17, 42);
```

The Grand Claim(s)

- Concepts will make it easier to write better generic C++ libraries
- Why?
  - Concepts let us say what we mean, in code
  - Type checking of templates, concept maps
  - Many complicated template tricks go away
- The resulting generic libraries will be far more powerful and easier to use

How Will a C++0x Library Look?

- Concepts:
  - The library will contain concepts that describe its domain of applicability
  - Graphs, Matrices, Databases, Functions, etc.
- Generic algorithms:
  - Constrained templates based on those concepts
- Core data structures
  - Concept maps establish relationships between data structures and concepts

How Does it Differ From C++03?

- Short answer: everything with “template” will look a bit different
- Your documentation becomes code:
  - Concept documentation turns into concepts
  - Algorithm constraint documentation turns into requirements clauses
  - Data structure/concept relationships turn into concept maps
- Many of the changes make for a cleaner, tighter specification.

How Do We Get There?

- Two major components to this tutorial
  - How do we get from a C++03 library to a C++0x library using concepts?
  - How do we make that C++0x library backward-compatible with C++03?
- Bonus material: generic library composition

Library Evolution
C++03 ⇒ C++0x

Evolving a Library

1. Get your library building under C++0x
2. Convert concept documentation to concepts (first pass; very minimal)
3. Constrain algorithms
   - Start with the simplest algorithms!
   - You will have to tweak your concepts
4. Add concept maps for your data structures
5. Build in backward compatibility with C++03

Table 28—EqualityComparable requirements

expression	return type	requirement
`a == b`	convertible to `bool`	`==` is an equivalence relation, that is, it satisfies the following properties: — For all `a`, `a == a`. — If `a == b`, then `b == a`. — If `a == b` and `b == c`, then `a == c`.

```
auto concept EqualityComparable<typename T>
{
  bool operator==(T, T);

  axiom Reflexivity(T x) { x == x; }
  axiom Symmetry(T x, T y) {if (x == y) y == x; }
  axiom Transitivity(T x, T y, T z) {
    if (x == y && y == z) x == z;
  }
}
```

Table 72—Input iterator requirements

operation	type	semantics, pre/post-conditions
`X u(a);`	`X`	post: `u` is a copy of `a` A destructor is assumed to be present and accessible.
`u = a;`	`X&`	result: `u` post: `u` is a copy of `a`
`a == b`	convertible to `bool`	`==` is an equivalence relation over its domain.
`a != b`	convertible to `bool`	`bool(a==b) != bool(a!=b)` over the domain of `==`
`*a`	convertible to `T`	pre: `a` is dereferenceable. If `a==b` and `(a,b)` is in the domain of `==` then `*a` is equivalent to `*b`.
`a->m`		pre: `(*a).m` is well-defined Equivalent to `(*a).m`
`++r`	`X&`	pre: `r` is dereferenceable. post: `r` is dereferenceable or `r` is past-the-end. post: any copies of the previous value of `r` are no longer required either to be dereferenceable or to be in the domain of `==`.
`(void)r++`		equivalent to `(void)++r`
`*r++`	`T`	`{ T tmp = *r; ++r; return tmp; }`

InputIterator with Concepts

```
concept InputIterator<typename X>
  : Assignable<X>, EqualityComparable<X>
{
  typename value_type = X::value_type;
  typename difference_type = X::difference_type;
  typename reference = X::reference;
  typename pointer = X::pointer;

  requires SignedIntegral<difference_type> &&
           Convertible<reference, value_type> &&
           Arrowable<pointer, value_type>;
  typename postincrement_result;
  requires Dereferenceable<postincrement_result,
                           value_type>;
  pointer operator->(X);
  X& operator++(X&);
  postincrement_result operator++(X&, int);
  reference operator*(X);
  bool operator!=(X x, X y);
}
```

Table 74—Forward iterator requirements

expression	return type	operational semantics	assertion/note pre/post-condition
`X u;`			note: u might have a singular value. note: a destructor is assumed.
`X()`			note: `X()` might be singular.
`X(a)`			`a == X(a)`.
`X u(a);` `X u = a;`		`X u; u = a;`	post: `u == a`.
`a == b`	convertible to `bool`		== is an equivalence relation.
`a != b`	convertible to `bool`	`!(a == b)`	
`r = a`	`X&`		post: r == a.
`*a`	`T&`		pre: `a` is dereferenceable. `a == b` implies `*a == *b`. If `X` is mutable, `*a = t` is valid.
`a->m`	`U&`	`(*a).m`	pre: `(*a).m` is well-defined.
`++r`	`X&`		pre: `r` is dereferenceable. post: `r` is dereferenceable or `r` is past-the-end. `r == s` and `r` is dereference-able implies `++r == ++s`. `&r == &++r`.

`r++`	convertible to `const X&`	`{ X tmp = r;`
		`++r;`
		`return tmp; }`	

`*r++`	`T&`		

ForwardIterator with Concepts

```
concept ForwardIterator<typename X>
  : InputIterator<X>, DefaultConstructible<X>
{
  requires Convertible<reference, const value_type&> &&
           Arrowable<pointer, const value_type&> &&
           Convertible<postincrement_result, const X&>;
};

concept MutableForwardIterator<typename X>
  : ForwardIterator<X>, BasicOutputIterator<X>
{
  requires SameType<reference, value_type&> &&
           Arrowable<pointer, value_type&>;
};
```

Specifying `for_each`

- Without concepts:

```
template<class InputIterator, class Function>
  Function
  for_each(InputIterator first, InputIterator last,
           Function f);
```

- With concepts:

```
template<InputIterator Iter,
         CopyConstructible Function>
  requires Callable1<Function,
                     InputIterator<Iter>::reference>
  Function
  for_each(Iter first, Iter last, Function f);
```

Let The Compiler Help

```
template<InputIterator Iter, typename T>
requires EqualityComparable<Iter::reference, T>
Iter find(Iter first, Iter last, const T& value) {
  while (first < last && !(*first == value))
    ++first;
  return first;
}
```

ConceptGCC

find.cpp: In function 'Iter find(Iter, Iter, const T&)':
find.cpp:7: error: no match for 'operator<' in 'first < last'

Specifying `copy`

- Without concepts:

```
template<class InputIterator, class OutputIterator>
  OutputIterator
  copy(InputIterator first, InputIterator last,
       OutputIterator result);
```

- Thinking about copy semantics…
  - The current specification allows conversions while we copy.
  - Did we mean for those conversions to happen?
    - If we didn't, are we stuck with them?
    - Many such choices when evolving a library

`copy` with Concepts

- We could tighten the semantics of copy…

```
template<InputIterator InIter,
         BasicOutputIterator OutIter>
  requires SameType<InIter::value_type,
                    OutIter::value_type>
  OutIter
  copy(InIter first, InIter last, OutIter result);
```

- Or model copy as it is today…

```
template<InputIterator InIter, typename OutIter>
  requires OutputIterator<OutIter,
                          InIter::value_type>
  OutIter
  copy(InIter first, InIter last, OutIter result);
```

Today's `advance`

- Without concepts:

```
template <class InputIterator, class Distance>
  void advance(InputIterator& i, Distance n);
```

"Since only random access iterators provide `+` and `–` operators, the library provides two template functions `advance` and `distance`. These functions use `+` and `–` for random access iterators (and are, therefore, constant time for them); for input, forward and bidirectional iterators they use `++` to provide linear time implementations."

Today's `advance`, in practice

```
template<class InputIterator, class Distance>
  void advance_impl(InputIterator& i, Distance,
                    input_iterator_tag);
template<class InputIterator, class Distance>
  void advance_impl(InputIterator& i, Distance,
                    bidirectional_iterator_tag);
template<class InputIterator, class Distance>
  void advance_impl(InputIterator& i, Distance,
                    random_access_iterator_tag);

template <class InputIterator, class Distance>
  void advance(InputIterator& i, Distance n)
  {
    typedef iterator_traits<InputIterator>::iterator_category
      Cat;
    advance_impl(i, n, Cat());
  }
```

advance with enable_if

```
template <class InIter, class Distance>
  typename enable_if<is_input_iterator<InIter>::value
                  && !is_bidir_iterator<InIter>::value
                     >::type
  void advance(InputIterator& i, Distance n);

template <class InIter, class Distance>
  typename enable_if<is_bidir_iterator<InIter>::value
                     && !is_ra_iterator<InIter>::value
                     >::type
  void advance(InputIterator& i, Distance n);

template <class InIter, class Distance>
  typename enable_if<is_ra_iterator<InIter>::value
                     >::type
  void advance(InputIterator& i, Distance n);
```

`advance` with Concepts

```
template <InputIterator Iter>
  void advance(Iter& i, Iter::difference_type n);

template <BidirectionalIterator Iter>
  void advance(Iter& i, Iter::difference_type n);

template <RandomAccessIterator Iter>
  void advance(Iter& i, Iter::difference_type n);
```

Same overloading behavior as before, without the hidden function, traits, or tag dispatching.

Constraining Class Templates

```
template<typename T1, typename T2>
struct pair {
  pair();
  pair(const pair&);
  pair(const T1&, const T2&);
  template<typename U1, typename U2>
    pair(const pair<U1, U2>&);
  pair& operator=(const pair&);

  T1 first;
  T2 second;
};
```

```
template<typename T1, typename T2>
struct pair {
  requires DefaultConstructible<T1> &&
           DefaultConstructible<T2>
    pair();
  requires CopyConstructible<T1> &&
           CopyConstructible<T2>
    pair(const pair&);
  requires CopyConstructible<T1> &&
           CopyConstructible<T2>
    pair(const T1&, const T2&);
  template<typename U1, typename U2>
    requires Convertible<U1, T1> && Convertible<U2, T2>
    pair(const pair<U1, U2>&);
  requires Assignable<T1> && Assignable<T2>
    pair& operator=(const pair&);
};
```

Concept Maps

- Quoth the C++03 Standard:
  "A `vector` satisfies all of the requirements of a container and of a reversible container (given in two tables in 23.1) and of a sequence,"
- Express this as code:

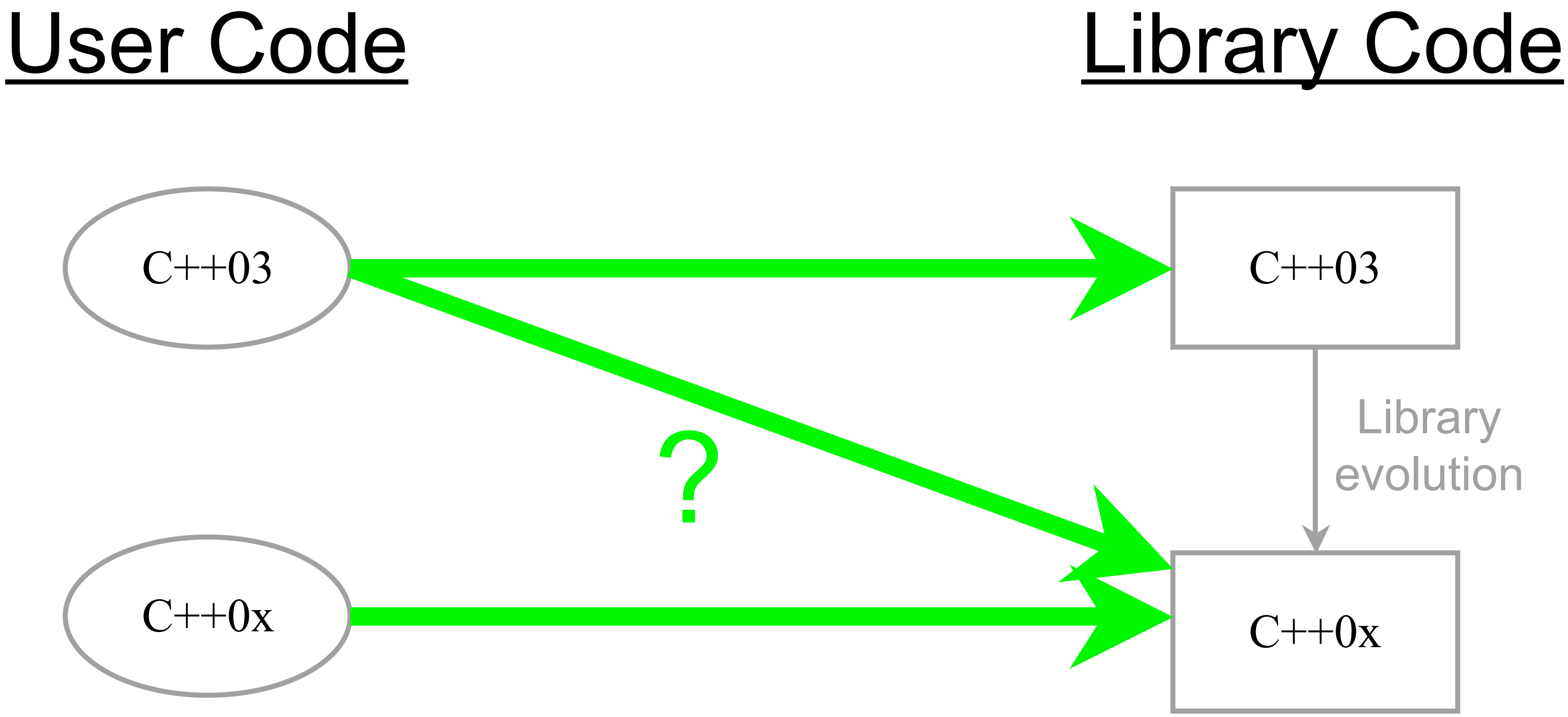
```
template<CopyConstructible T>
concept_map Sequence<vector<T>> { }

template<CopyConstructible T>
concept_map
    ReversibleContainer<vector<T>> { }
```

Checking Concept Maps

raiter.cpp:12: error: 'concept_map MutableRandomAccessIterator<std::_Bit_iterator>' does not meet the nested requirements of its concept
raiter.cpp:12: note: same-type constraint 'std::SameType<MutableRandomAccessIterator<Iter>::reference, MutableRandomAccessIterator<Iter>::value_type&>' is not satisfied ('std::_Bit_reference' is not 'bool&')

```
concept_map
   MutableRandomAccessIterator<_Bit_iterator>
   {}
```

What Backward Compatibility?



User-Defined Iterators

- What if I defined my own iterator type to use with Standard Library algorithms?
  - Without concepts: you need to specialize `iterator_traits` or provide typedefs.
  - With concepts: you need to provide a concept map.
- Can we keep backward compatibility with `iterator_traits`-style iterators?
  - Yes, with a clever concept map template.

Mapping User-Defined Iterators

- Existing iterators will somehow have iterator traits defined

```
struct my_iterator {
  typedef std::input_iterator_tag iterator_category;
  typedef char value_type;
  typedef const char& reference;
  // ...
};
```

- We need our concept map to:
  - Query those iterator traits
  - Match those traits to the right concept
  - Write this automatically:

```
concept_map InputIterator<my_iterator> {} // automatic?
```

Extracting Iterator Traits

```
auto concept _Iter_traits<typename Iter> {
  typename iterator_category =
    iterator_traits<Iter>::iterator_category;
  typename value_type =
    iterator_traits<Iter>::value_type;
  typename reference =
    iterator_traits<Iter>::reference;
  typename pointer =
    iterator_traits<Iter>::pointer;
  typename difference_type =
    iterator_traits<Iter>::difference_type;
}
```

The Escape Hatch

- The `late_check` keyword introduces a late-checked template:
  - Late-checked templates have requires clauses
  - Users must meet the template requirements
  - *Template body is not type-checked until instantiation time.*

Mapping User-Defined Iterators

```
late_check template<typename Iter>
  requires _Iter_traits<Iter> &&
    Convertible<_Iter_traits<Iter>::iterator_category,
                input_iterator_tag>
  concept_map InputIterator<Iter> {
    typedef _Iter_traits<Iter>::value_type
      value_type;
    typedef _Iter_traits<Iter>::difference_type
      difference_type;
    typedef _Iter_traits<Iter>::pointer pointer;
    typedef _Iter_traits<Iter>::reference reference;
    typedef _Iter_traits<Iter>::difference_type
      difference_type;
  };
```

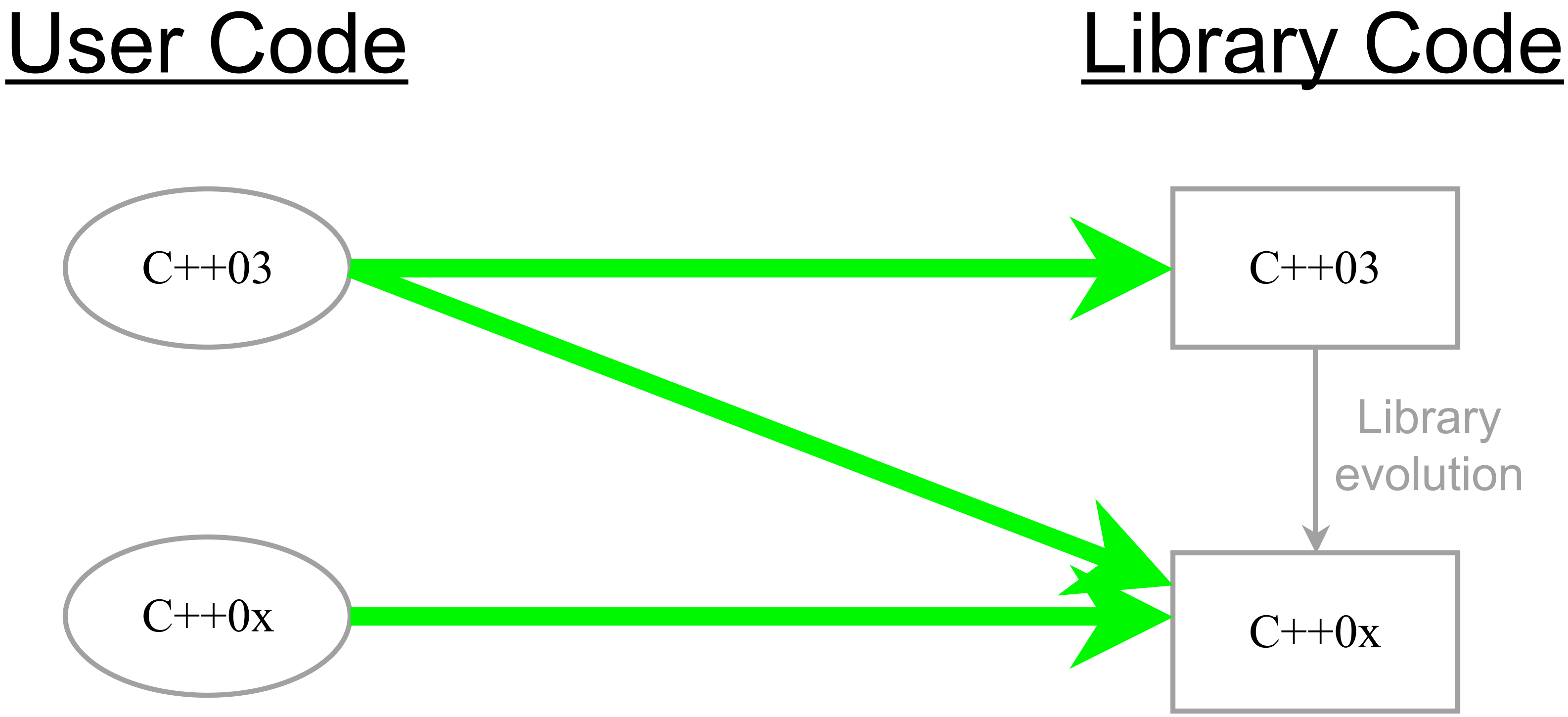
From the User's Perspective

- C++03 uses of the C++0x library "just work":

```
void f(my_iterator first, my_iterator last) {
  std::find(first, last, 'a');
}
```

- Under the hood:
  - Compiler looks for a concept map `InputIterator<my_iterator>`
  - Concept map template queries `iterator_traits`, produces concept map

What Backward Compatibility?

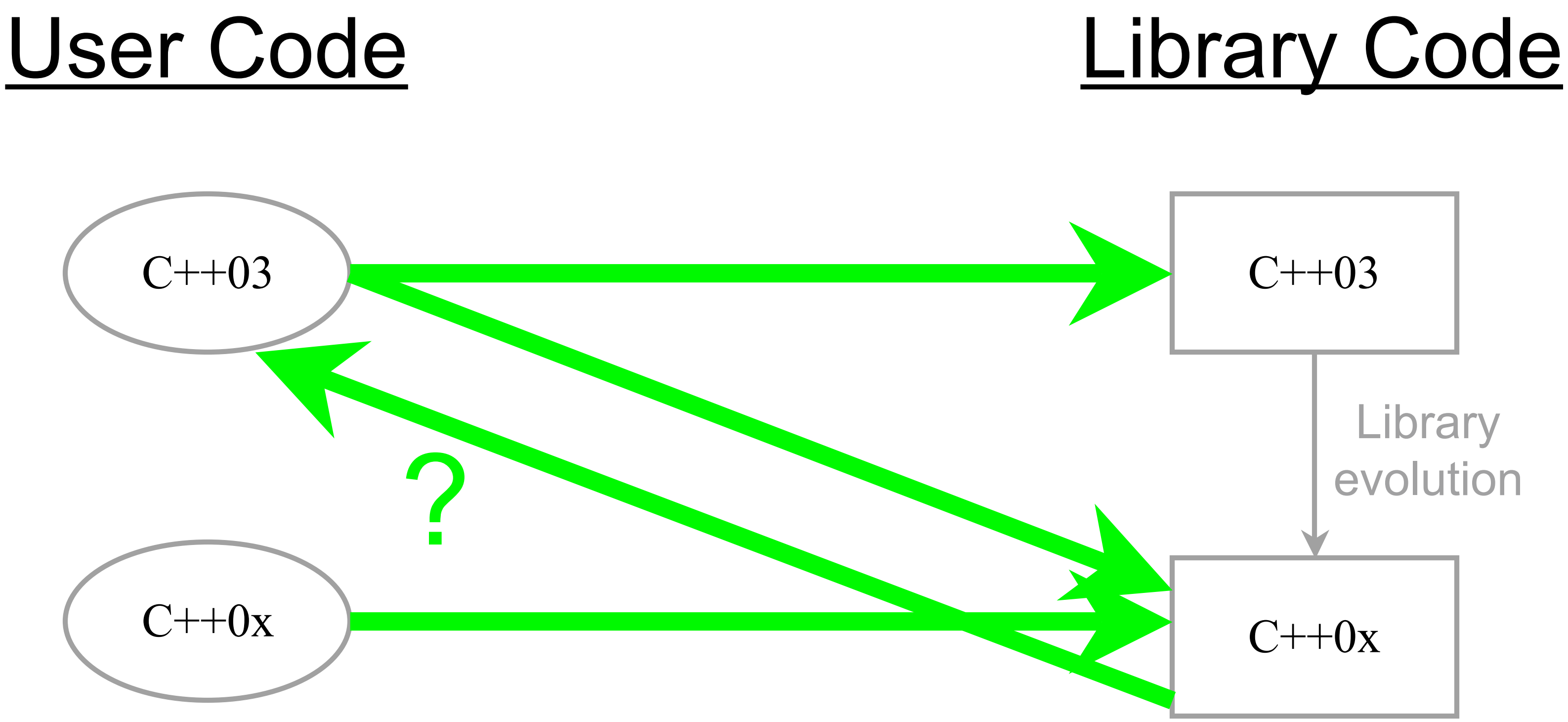


The Death of `iterator_traits`?

- Traits don't work with constrained templates
  - Traits rely entirely on specialization...
  - Specialization does not play well with modular type checking
- Can we kill `iterator_traits`?
  - No: existing code relies on it

```
template<typename Iter>
  typename iterator_traits<Iter>::difference_type
  distance(Iter first, Iter last);
```

What Backward Compatibility?

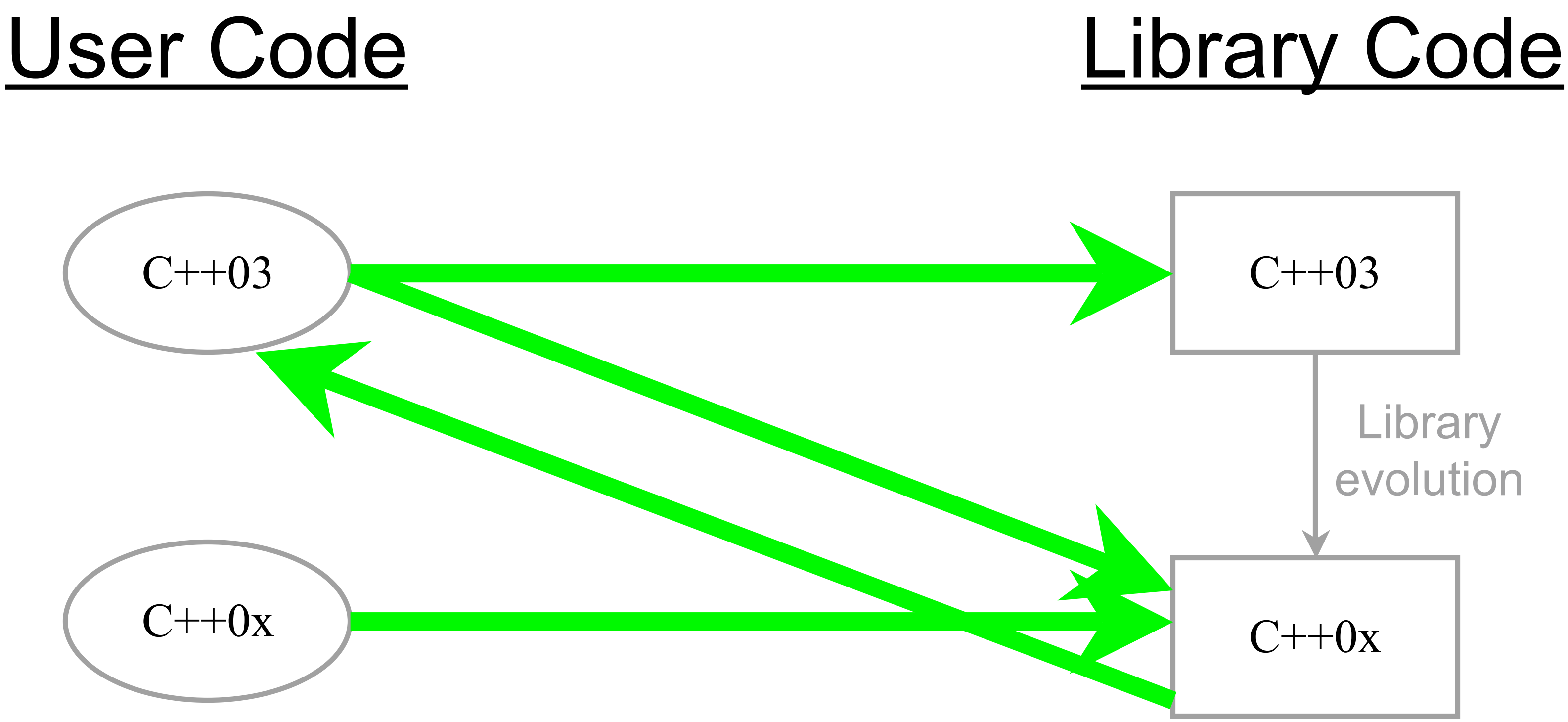


A New Life for `iterator_traits`

- Provide partial specializations of `iterator_traits` based on concepts

```
template<typename Iter>
  requires InputIterator<Iter>
struct iterator_traits<Iter> {
  typedef input_iterator_tag iterator_category;
  typedef InputIterator<Iter>::value_type value_type;
  typedef InputIterator<Iter>::difference_type
    difference_type;
  typedef InputIterator<Iter>::pointer pointer;
  typedef InputIterator<Iter>::reference reference;
};
```

Backward Compatibility



A Better `is_convertible`

```
template<typename T, typename U>
struct is_convertible {
  static const bool value = false;
};

auto concept Convertible<typename T, typename U> {
  operator U(T);
}

template<typename T, typename U>
requires Convertible<T, U>
struct is_convertible<T, U> {
  static const bool value = true;
};
```

“Trait-Informed” Concepts

- We can write a concept to detect true/false:

```
concept True<bool Condition> { }
concept_map True<true> { }
```

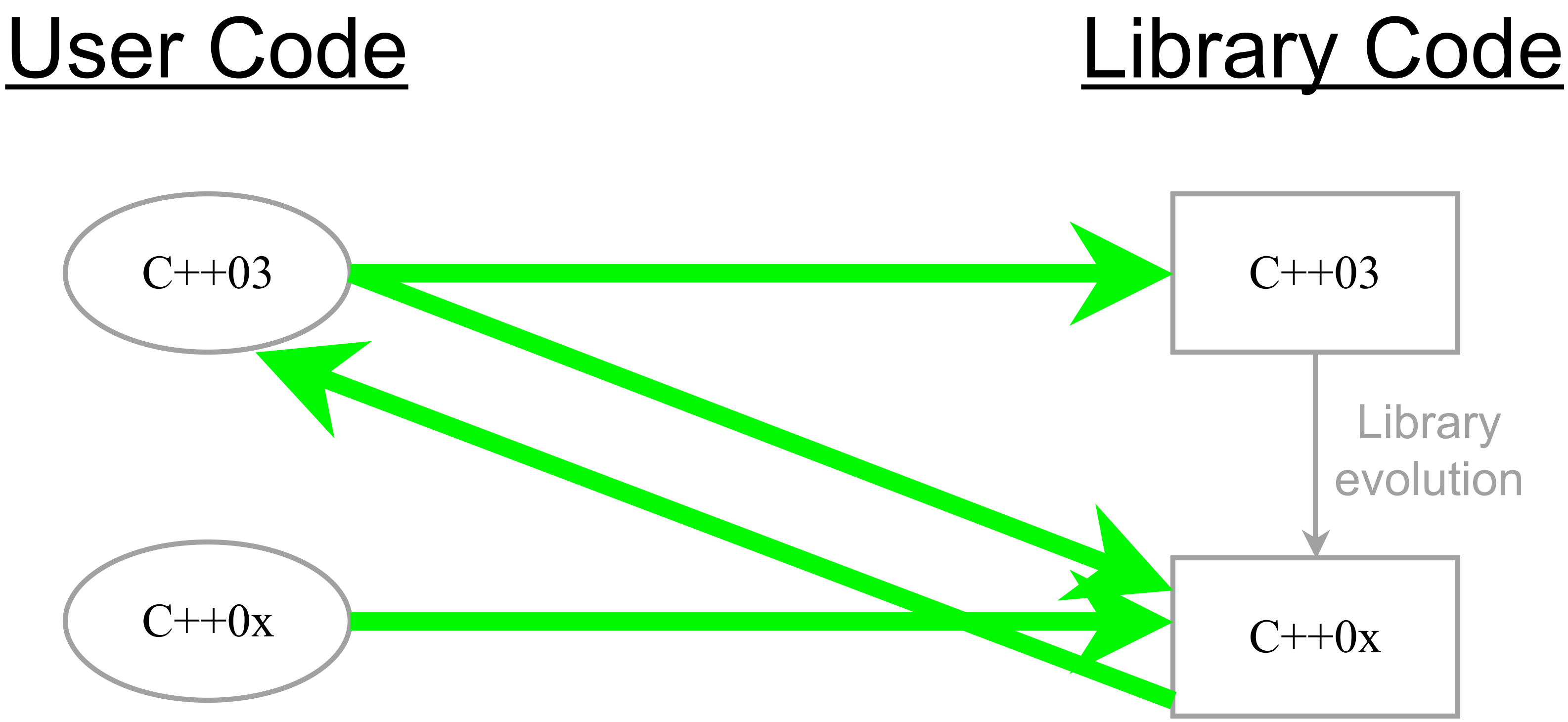
- And use that concept to inform concepts via traits:

```
concept TriviallyDefaultConstructible<typename T>
  : DefaultConstructible<T> { }

template<typename T>
  requires DefaultConstructible<T> &&
        True<has_trivial_default_constructor<T>::value>
concept_map TriviallyDefaultConstructible<T> { }
```

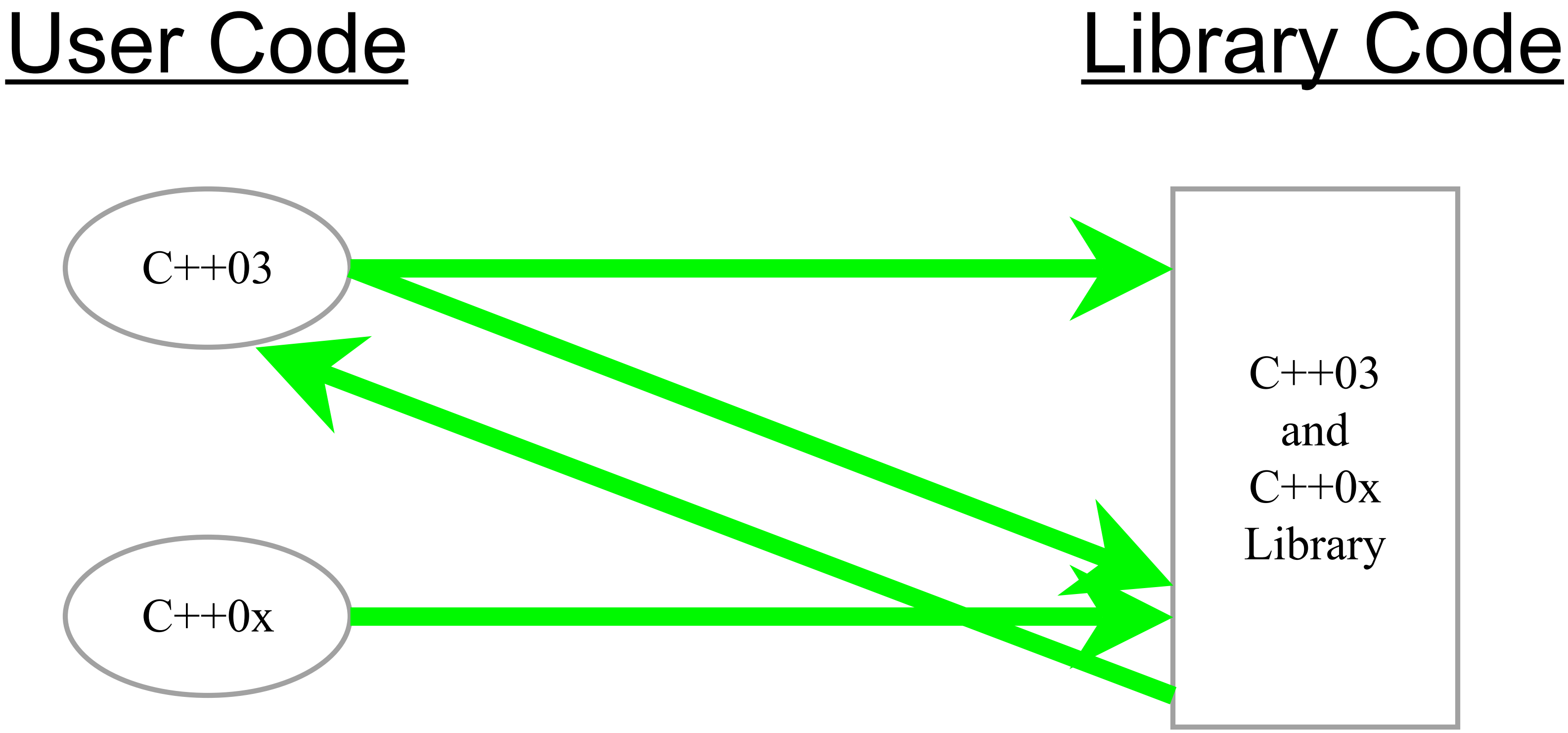
What about C++03?

- Ubiquitous C++0x support is still far off on the horizon
  - We need our libraries to compile as C++03 and C++0x
- Compatibility with C++03 is going to require some effort
  - Macros (as few as we can)
  - “Fake” concepts

C++03/C++0x Libraries Distinct



A More Practical Approach



C++03/C++0x

```
template<typename Iter, typename T>
requires
  InputIterator<Iter> &&
  EqualityComparable<InputIterator<Iter>::value_type, T>
Iter find(Iter first, Iter last, const T& value) {
  while (first != last && !(*first == value))
    ++first;
  return first;
}
```

Macro BOOST_REQUIRES

```
#ifdef BOOST_HAS_CONCEPTS
#  define BOOST_REQUIRES(...) requires __VA_ARGS__
#else
#  define BOOST_REQUIRES(...)
#endif

template<typename Iter, typename T>
BOOST_REQUIRES(
  InputIterator<Iter> &&
  EqualityComparable<InputIterator<Iter>::value_type, T>)
Iter find(Iter first, Iter last, const T& value) {
  while (first != last && !(*first == value))
    ++first;
  return first;
}
```

Dealing with Associated Types

- We handle associated types differently in C++03 vs. C++0x
  - C++03: Traits

```
template<typename Iter>
  typename iterator_traits<Iter>::difference_type
  distance(Iter first, Iter last);
```

  - C++0x: Associated types

```
template<typename Iter>
  BOOST_REQUIRES(InputIterator<Iter>)
  InputIterator<Iter>::difference_type
  distance(Iter first, Iter last);
```

Associated Types, Approach #1

- Remember our concept-based specializations of `iterator_traits`?

```
template<typename Iter>
  BOOST_REQUIRES(InputIterator<Iter>)
  typename iterator_traits<Iter>::difference_type
  distance(Iter first, Iter last);
```

- Unfortunately, this solution is neither here nor there
  - Mixes C++03/C++0x paradigms

Associated Types, Approach #2

- Assumption: C++0x view is cleaner
- Solution: build concept look-alikes in C++03

```
#ifndef BOOST_HAS_CONCEPTS
template<typename Iter>
struct InputIterator : std::iterator_traits<Iter> { };

template<typename Iter>
struct ForwardIterator : InputIterator<Iter> { };
#endif

template<typename Iter>
  BOOST_REQUIRES(InputIterator<Iter>)
  typename InputIterator<Iter>::difference_type
  distance(Iter first, Iter last);
```

Necessary in C++03, harmless in C++0x

Concept-Based Overloading

- Concept-based overloading in C++03 uses two tricks:
  - Tag dispatching
  - `enable_if`/SFINAE
- Concept-based overloading in C++0x is expressed directly via overloads
- Macro solution is possible
  - ... but it is ugly
  - ***really ugly***

An Open Problem

- Good C++03/C++0x implementations of concept-based are unknown
  - The C++03 hacks are egregious
- Known approaches and ideas:
  1. Use three (!) macros and a C++03 dispatching function
  2. Macroize `enable_if` checks for C++03
  3. Completely separate C++03/C++0x overloading/dispatching logic

“Obsoleted” Template Tricks

Trick	Concept Equivalent
Type traits	Concepts
Tag dispatching	Concept-based overloading
Concept checking, Concept archetypes	Constrained templates
`sizeof`, SFINAE tricks	Concepts
`enable_if`	Requirements clause
Adaptors/façades	Concept maps

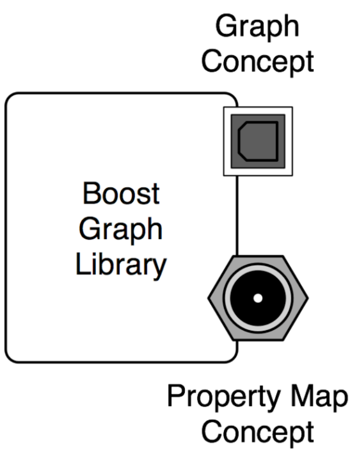
Concepts, Generic Libraries, and Composition

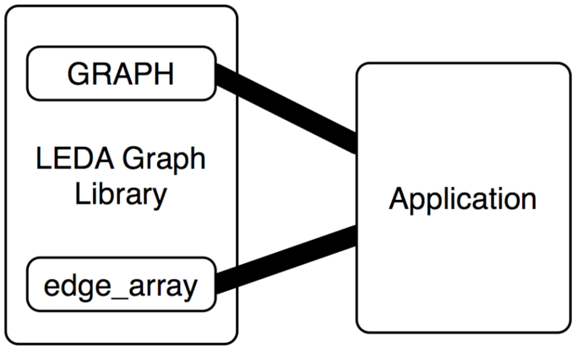
The Boost Graph Library

- We want to “conceptualize” the Boost Graph Library
  - Easier to use, easier to implement
  - More composable with other libraries

```
concept Graph<typename G> {
  typename vertex_type;
  typename edge_type;
  ForwardIterator OutEdgeIterator;

  int num_vertices(G);
  int out_degree(vertex_type, G);
  pair<OutEdgeIterator, OutEdgeIterator>
    out_edges(vertex_type, G);
}
```

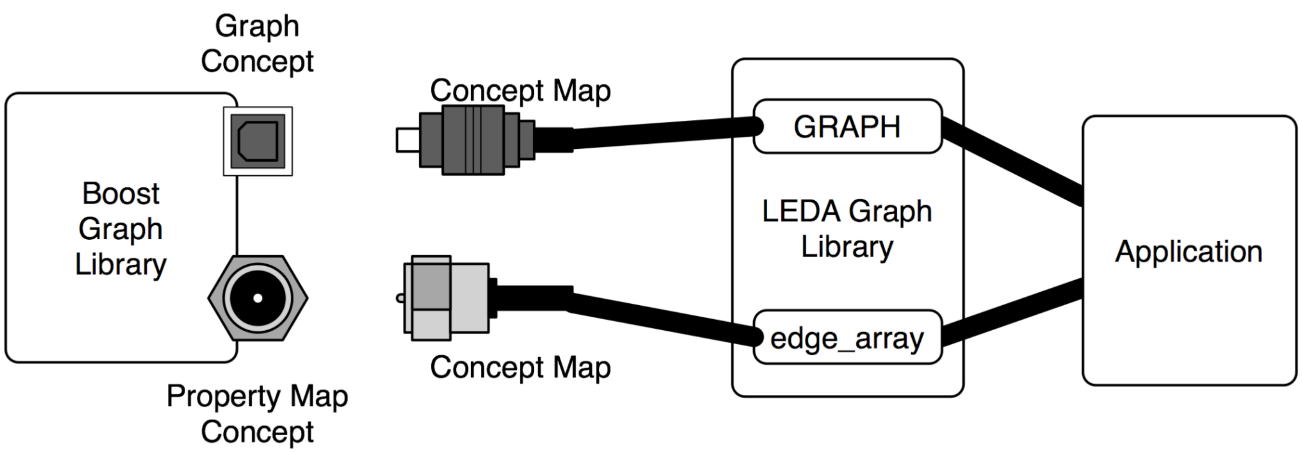
Concept Maps for Composition





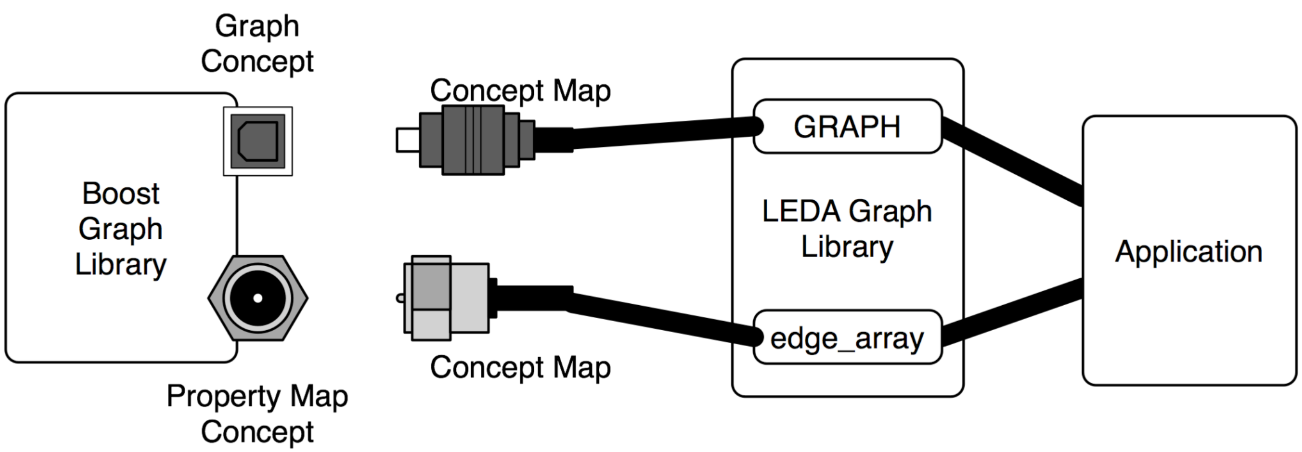

```
leda::GRAPH<Server, Link> internet_graph;
leda::edge_array<double> total_latency;
boost::shortest_paths(internet_graph, start, total_latency);
```

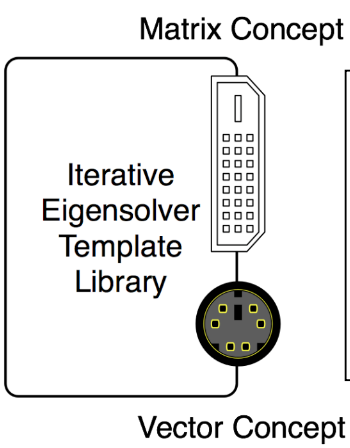
Concept Maps for Composition




```
template<typename V, typename E>
concept_map Graph<leda::GRAPH<V, E> > {
  typedef leda::leda_node vertex_type;
  int num_vertices(const leda::GRAPH<V, E>& g) {
    return g.number_of_nodes();
  }
  int out_degree(vertex_type v, const leda::GRAPH<V, E>&) {
    return outdeg(v);
  }
};
```

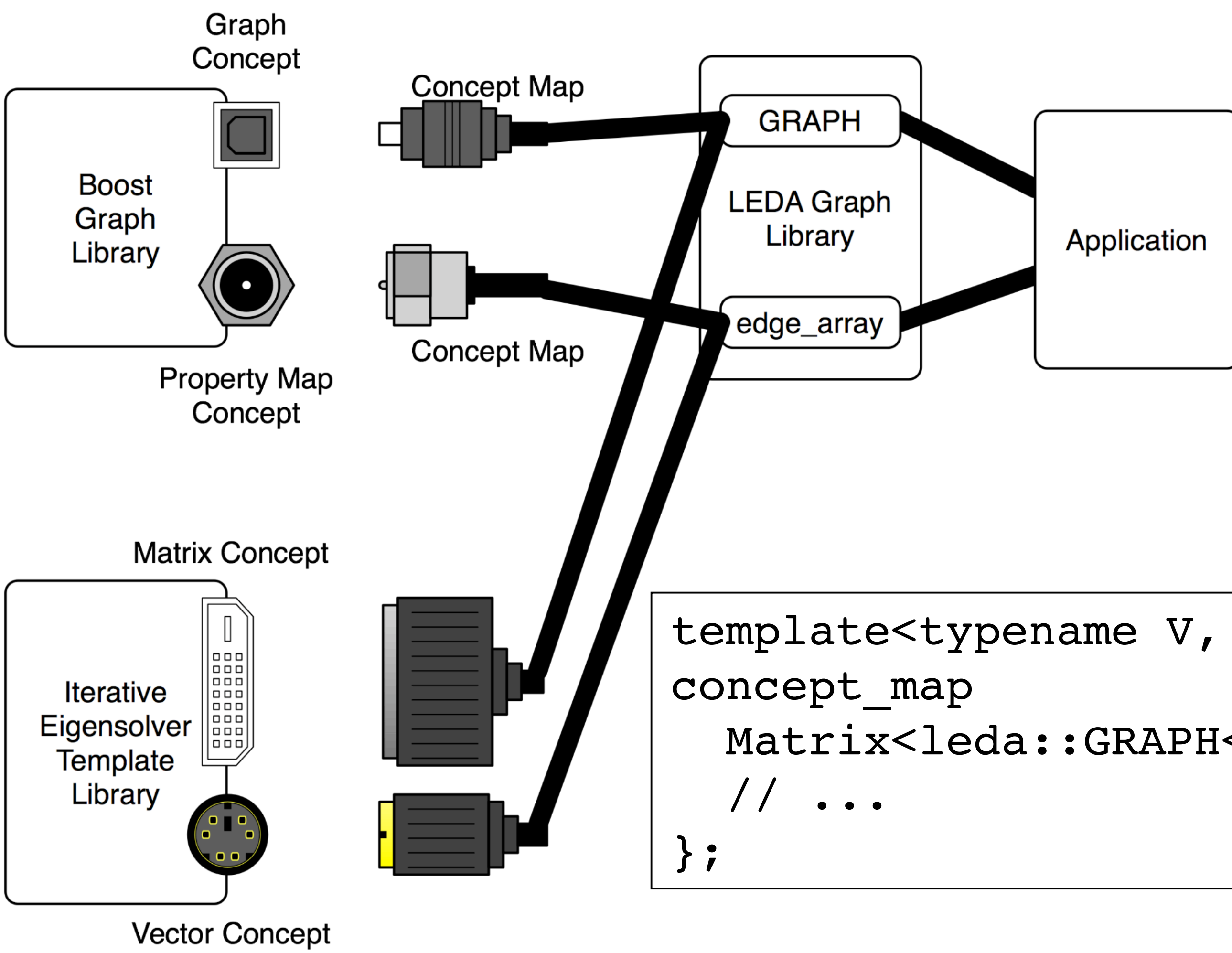
Concept Maps for Composition





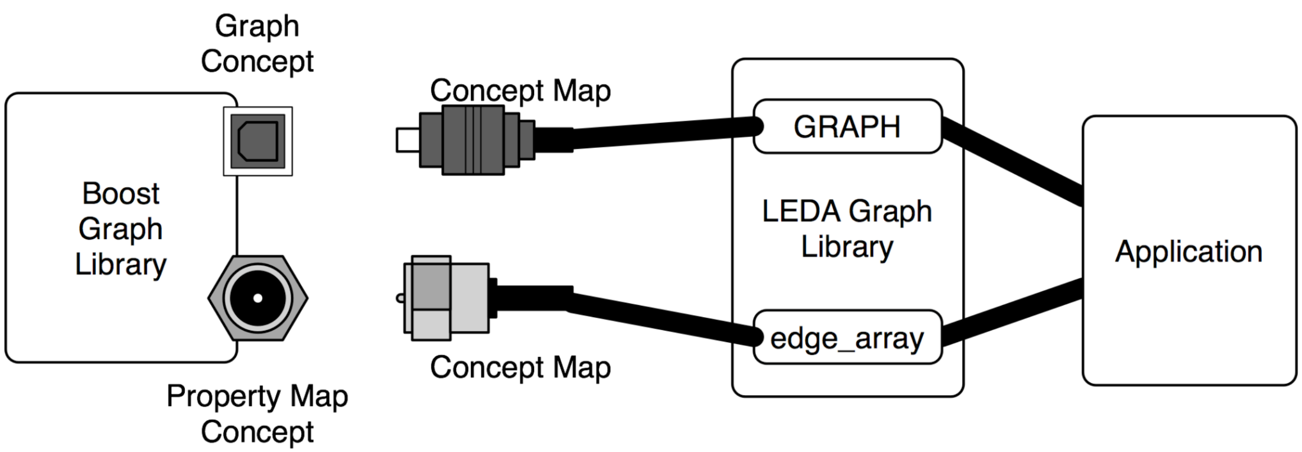

```
leda::GRAPH<Server> internet_graph;
leda::edge_array<double> eigenvector;
double eigenvalue =
  ietl::compute_eigenvector(internet_graph,
                            eigenvector, 0);
```

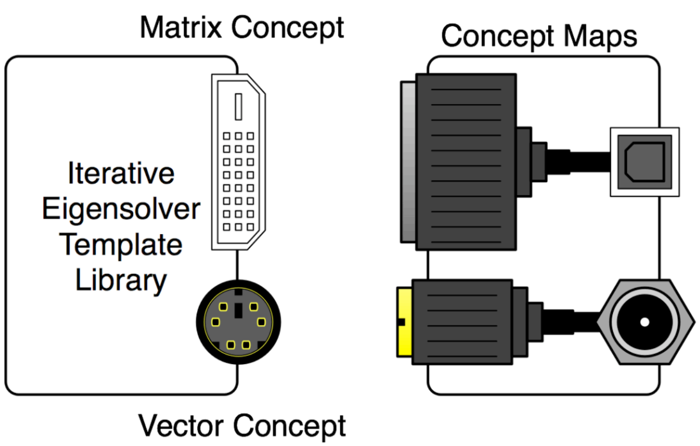
Concept Maps for Composition




```
template<typename V, typename E>
concept_map
  Matrix<leda::GRAPH<V, E> > {
  // ...
};
```

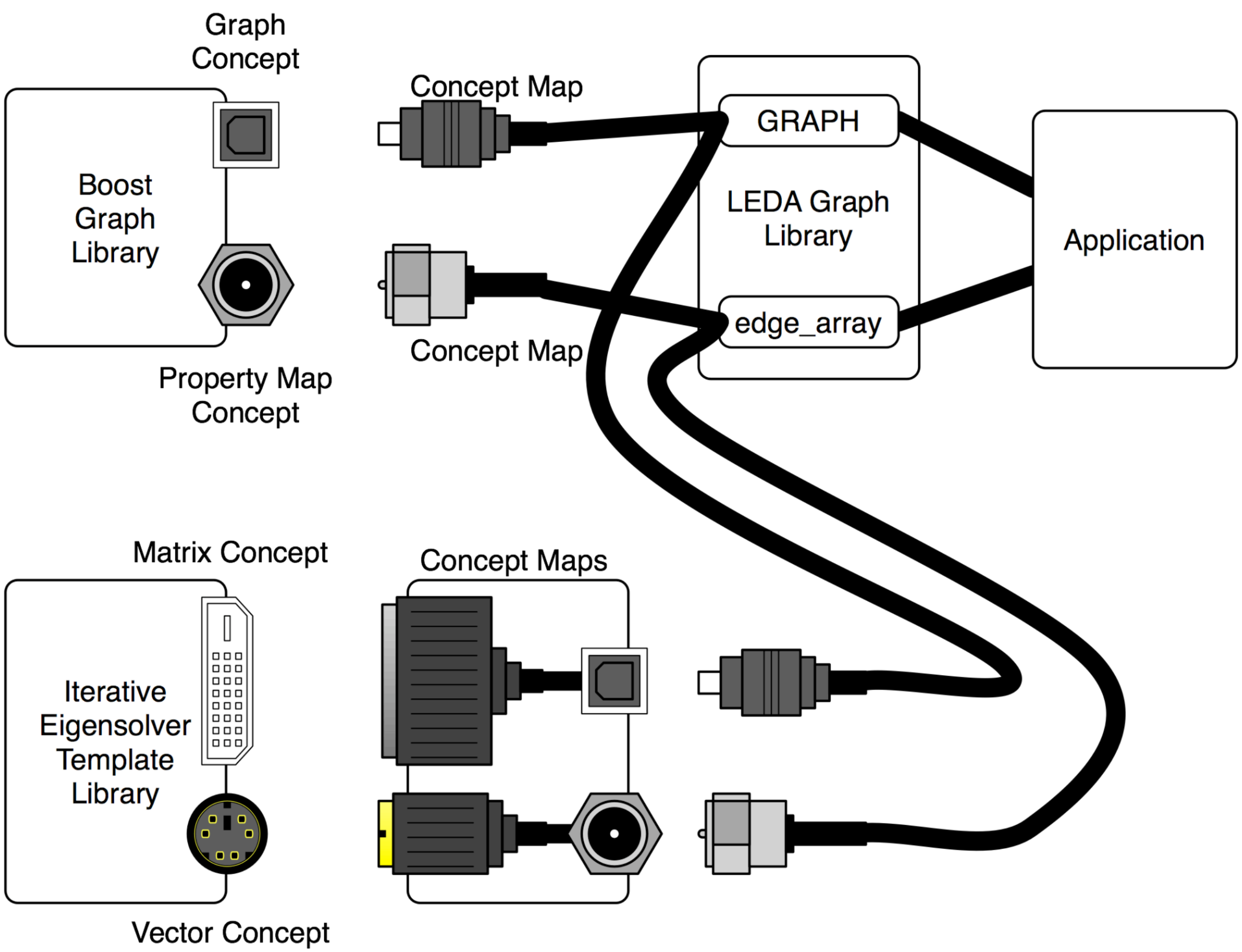
Concept Maps for Composition






```
template<typename G>
requires Graph<G>
concept_map Matrix<G> {
  // ...
};
```

Concept Maps for Composition



Summary: C++0x Evolution

- Evolving a library takes time and patience
  - Start small, with toy versions of your library
  - Evolve from the bottom up
- C++03/C++0x libraries are possible
  - Concept maps, trait specializations for backward compatibility
  - Some macros are necessary
  - Documentation effort is simplified (use C++0x)
  - Better testing coverage; more to test

A Call To Arms

- Boost is a vehicle for change in the C++ community

- Try new C++0x features

- C++0x-specific implementations of Boost libraries

TO HELP MAKE C++0x GREAT