An Introduction to Concepts in C++0x

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

What Are Concepts?

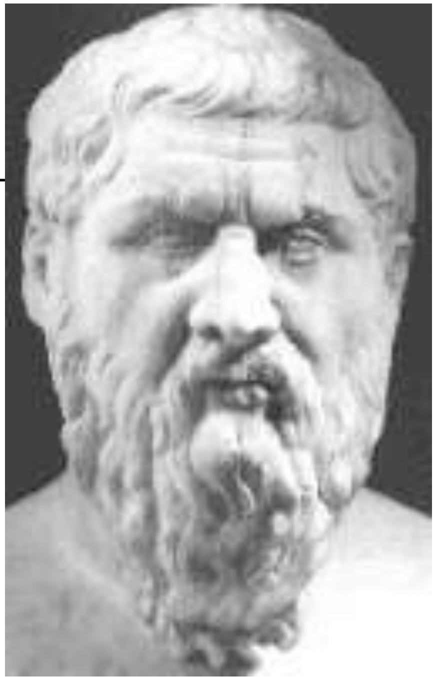
- Concepts are a new language feature for C++0x
- Concepts make templates easier to use
  - Express template requirements directly in code
  - Provide complete type-checking of templates
- Concepts support the Generic Programming paradigm

Concepts Tutorial: Outline

- First half:
  - Generic Programming, Concepts and C++0x
  - Core concepts features
  - Hands-on: Building a mini-STL with Concepts
- Second half:
  - Advanced concepts features
  - Where do we go from here?

Generic Programming, Concepts and C++0x

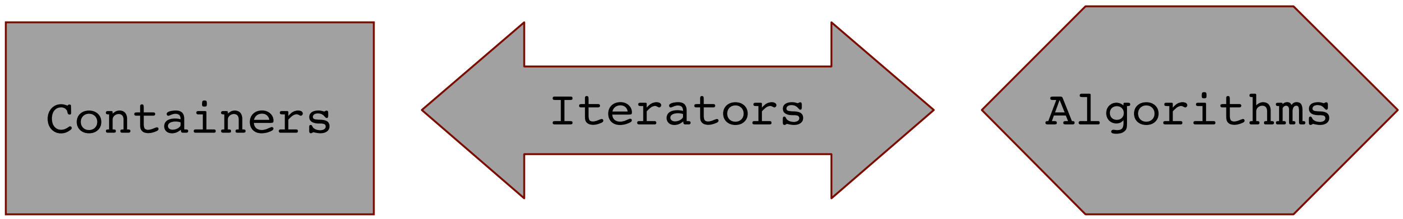
Generic Programming



- A methodology for the development of reusable software libraries
- Three primary tasks:
  - Categorize the abstractions in a domain into **concepts**
  - Implement **generic algorithms** based on the concepts
  - Build concrete **models** of the concepts

The Standard Template Library

- The C++ Standard Template Library embodies Generic Programming
  - First widespread application of Generic Programming
  - Three major kinds of components:



  - Performance: abstraction penalty benchmarks
- Many C++ libraries have followed STL

Generic Programming in C++

- C++ templates enable the application of GP
  - Overloading permits natural abstractions
  - Instantiation eliminates cost of abstractions
  - Many successful, generic libraries in C++

- Significant problems remain:
  - Inability to directly express ideas of GP
  - Generic libraries in C++ are fragile

Fragile C++ Templates

- A generic find() algorithm, from the STL:

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

- Using find():
  - `std::vector<int> v;`
    `find(v.begin(), v.end(), 17); // okay`
  - `std::list<int> l;`
    `find(l.begin(), l.end(), 42); //` **`error!`**

Fragile C++ Templates

- A generic find() algorithm, from the STL:

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

- Error was in the definition of the template:
  - But it was found by an unlucky user!
  - < is not part of the `Input Iterator` concept

Wouldn't It Be Nice...

```
template<typename Iter, typename T>
// Iter is an input iterator, its value_type is
// equality comparable with T
Iter find(Iter first, Iter last, const T& value) {
  while (first < last && !(*first == value))
    ++first;
  return first;
}
```

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

Fragile C++ Templates

```
void f()
{
  list<int> l;
  sort(l.begin(), l.end());
}
```

```
…/c++/4.0.1/bits/stl_algo.h: In function 'void std::sort(_Iter, _Iter) [with _Iter = std::_List_iterator<int>]':
sort.cpp:8:   instantiated from here
.../c++/4.0.1/bits/stl_algo.h:2852: error: no match for 'operator-' in '__last - __first'
.../c++/4.0.1/bits/stl_algo.h: In function 'void std::__final_insertion_sort(_Iter, _Iter) [with _Iter =
std::_List_iterator<int>]':
.../c++/4.0.1/bits/stl_algo.h:2853:   instantiated from 'void std::sort(_Iter, _Iter) [with _Iter =
std::_List_iterator<int>]'
sort.cpp:8:   instantiated from here
.../c++/4.0.1/bits/stl_algo.h:2465: error: no match for 'operator-' in '__last - __first'
.../c++/4.0.1/bits/stl_algo.h:2467: error: no match for 'operator+' in '__first + 16'
.../c++/4.0.1/bits/stl_algo.h:2468: error: no match for 'operator+' in '__first + 16'
.../c++/4.0.1/bits/stl_algo.h: In function 'void std::__insertion_sort(_Iter, _Iter) [with _Iter =
std::_List_iterator<int>]':
.../c++/4.0.1/bits/stl_algo.h:2471:   instantiated from 'void std::__final_insertion_sort(_Iter, _Iter) [with
_Iter = std::_List_iterator<int>]'
.../c++/4.0.1/bits/stl_algo.h:2853:   instantiated from 'void std::sort(_Iter, _Iter) [with _Iter =
std::_List_iterator<int>]'
sort.cpp:8:   instantiated from here
.../c++/4.0.1/bits/stl_algo.h:2377: error: no match for 'operator+' in '__first + 1'
.../c++/4.0.1/bits/stl_algo.h:2471:   instantiated from 'void std::__final_insertion_sort(_Iter, _Iter) [with
_Iter = std::_List_iterator<int>]'
.../c++/4.0.1/bits/stl_algo.h:2853:   instantiated from 'void std::sort(_Iter, _Iter) [with _Iter =
std::_List_iterator<int>]'
sort.cpp:8:   instantiated from here
.../c++/4.0.1/bits/stl_algo.h:2383: error: no match for 'operator+' in '__i + 1'
```

Wouldn't It Be Nice...

```
void f()
{
  list<int> l;
  sort(l.begin(), l.end());
}
```

```
sort.cpp: In function 'void f()':
sort.cpp:8: error: no matching function for call to 'sort(std::_List_iterator<int>,
std::_List_iterator<int>)'
…/c++/4.0.1/bits/stl_algo.h:2839: note: candidates are: void std::sort(_Iter, _Iter)
[with _Iter = std::_List_iterator<int>] <requires clause>
sort.cpp:8: note:   no concept map for requirement
'std::MutableRandomAccessIterator<std::_List_iterator<int> >'
```

Concepts for C++: Goals

- Support for the core ideas of Generic Programming in C++
- Modular type checking for C++ templates
- Performance equivalent to C++ templates
- Complete backward compatibility
- Simplicity
- C++0x

The Concepts Crew

Douglas Gregor

Bjarne Stroustrup

Gabriel Dos Reis

Jaakko Järvi

Jeremy Siek

Andrew Lumsdaine

Jeremiah Willcock

Ronald Garcia

Concepts: Core Features

What is a Concept?

- A concept is a way of describing the requirements on abstractions
- A concept captures commonly occurring abstractions
- A concept forces many similar data structures into a general form
  - Enabling generic algorithms

Concepts Overview

- Three major parts:
  - **Concept definitions**: Specify the behavior of types via requirements.
  - **Requirements clauses**: Specify constraints on template parameters in terms of concepts.
  - **Concept maps**: Specify how types meet the requirements of a concept.

Constrained Templates

- Place constraints on template parameters via a *requirements clause*
  - Uses of the template must satisfy these constraints
  - Definition of the template can assume only what the constraints imply

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

Concept Definitions

- Concept definitions have:
  - A name (`LessThanComparable`)
  - A list of concept arguments (`T`)
  - A list of concept requirements (`operator<`)

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

Putting Things Together

- Constrained `min()` is type-checked against `LessThanComparable`:

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

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

finds `LessThanComparable<T>::operator<`

What's in a Concept Definition?

- Concepts have four kinds of requirements
  - Function requirements
  - Axioms
  - Associated type/template requirements
  - Associated requirements
- Much of the behavior of types can be described with these features

Function Requirements

- Express the requirement for a particular function to exist
  - May be a free function, operator, member function, constructor, etc.
- Examples:

```
concept Regular<typename T> {
  T::T();
  T::T(const T&);
  T::~T();
  void swap(T&, T&);
  bool operator==(const T&, const T&);
}
```

Concept Parameterization

- Concepts can have any number of parameters:

```
concept EqualityComparable<typename T,
                           typename U>
{
  bool operator==(T, U);
  bool operator!=(T, U);
}
```

Axioms

- `LessThanComparable` has a less-than operator, but what does it ***mean***?

```
concept LessThanComparable<typename T> {
  bool operator<(const T&, const T&);

  axiom Irreflexivity(T x) { !(x < x); }

  axiom Asymmetry(T x, T y) {
    if (x < y) !(y < x);
  }

  axiom Transitivity(T x, T y, T z) {
    if (x < y && y < z) x < z;
  }
}
```

Associated Types

- Associated types are types used to describe concepts
  - They vary like type parameters,
  - But live *in the concept body*.
- Example:

```
concept BinaryFunction<typename F,
                       typename T1, typename T2> {

  ???        operator()(F&, const T1&, const T2&);
}
```

Associated Types

- Associated types are types used to describe concepts
  - They vary like type parameters,
  - But live *in the concept body*.
- Example:

```
concept BinaryFunction<typename F,
                       typename T1, typename T2> {
  typename result_type;
  result_type operator()(F&, const T1&, const T2&);
}
```

Using Associated Types

- Write a simple algorithm that calls a `BinaryFunction` with its arguments reversed:

```
template<typename F, typename T1, typename T2>
requires BinaryFunction<F, T2, T1>
???
apply2r(F& f, const T1& t1, const T2& t2)
{
  return f(t2, t1);
}
```

Using Associated Types

- Write a simple algorithm that calls a `BinaryFunction` with its arguments reversed:

```
template<typename F, typename T1, typename T2>
requires BinaryFunction<F, T2, T1>
BinaryFunction<F, T2, T1>::result_type
apply2r(F& f, const T1& t1, const T2& t2)
{
  return f(t2, t1);
}
```

Associated Requirements

- Associated types are like type parameters
  - By default, they can be *any* type
- Associated requirements place requirements on associated types and type parameters:

```
concept BinaryFunction<typename F,
                       typename T1, typename T2> {
  typename result_type;
  requires CopyConstructible<result_type>;
  result_type operator()(F&, const T1&, const T2&);
}
```

Concept Maps

```
template<typename T>
  requires LessThanComparable<T>
  const T& min(const T& x, const T& y);

int x, y; cin >> x >> y;
int smaller = min(x, y); // okay?
```

- To call `min<int>`, we need to satisfy its requirements:
  - `LessThanComparable<int>`

- Use a ***concept map***:

```
concept_map LessThanComparable<int> {}
```

Concept Map Adaptation

```
struct dcomplex {
  double real, imag;
};

template<typename T>
  requires LessThanComparable<T> class set { ... };

concept_map LessThanComparable<dcomplex> {
  bool operator<(dcomplex x, dcomplex y) {
    return (x.real < y.real)
           || (x.real == y.real && x.imag < y.imag);
  }
}

set<dcomplex> cnums;
```

Concept Map Templates

- Is a `vector<T>` `LessThanComparable`?
  - Yes...
  - But only when `T` is `LessThanComparable`
- Express this as a concept map:

```
template<typename T>
requires LessThanComparable<T>
concept_map LessThanComparable<vector<T>> {}
```

Mapping Associated Types

- Can define associated types in concept maps
- Example: turn a function pointer into a `BinaryFunction`

```
concept_map BinaryFunction<int (*)(int, int),
                           int, int>
{
  typedef int result_type;
};
```

Mapping Associated Types

- Can define associated types in concept maps
- Example: turn a function pointer into a `BinaryFunction`

```
template<typename R, typename T1, typename T2>
concept_map BinaryFunction<R (*)(T1, T2),
                           T1, T2>
{
  typedef R result_type;
};
```

Implicit (`auto`) Concepts

- Some concepts have “shallow” semantics.
  - The syntax is enough to imply the semantics.
  - Writing concept maps for them is tedious.
- The compiler will automatically generate (empty) concept maps for `auto` concepts.
- Example:

```
auto
concept Predicate<typename F, typename T> {
  bool operator()(F&, T);
}
```

Implicit (`auto`) Concepts

```
template<typename Pred, typename T1, typename T2>
requires Predicate<Pred, T1> && Predicate<Pred, T2>
bool pred_and(Pred pred, const T1& t1, const T2& t2)
{
  return pred(t1) && pred(t2);
}

struct is_prime {
  bool operator()(int x);
};

if (pred_and(is_prime(), 17, 31))
  cout << "Both numbers are prime." << endl;
```

Concepts Recap

- We have seen the core Concepts features:
  - **Concept definitions**: express requirements on (sets of) types.
  - **Requirements clauses**: state the constraints templates place on their template parameters.
  - **Concept maps**: show *how* a set of types meets the requirements of a concept.

Hands-On:
Building a mini-STL

ConceptGCC

- ConceptGCC is a **prototype** implementation of concepts in C++
- Also includes:
  - Rvalue references
  - Variadic templates
  - decltype
  - Delegating constructors
- Freely available online:
  - http://www.generic-programming.org/software/ConceptGCC

ConceptGCC Basics

- Same command-line parameters as GCC
- Setup: put ConceptGCC in your path

```
export PATH=/opt/conceptgcc-boostcon/bin:$PATH
```

- To compile:

```
conceptg++ source.cpp
```

- To run a program:

`./a.out` (non-Windows) or `./a.exe` (Windows)

Our Task: Build a mini-STL

1. Start with concrete algorithms on integer pointers
   - `find(), count()`
2. "Lift" algorithms to pointers to 'T'
   - Write concepts and constrained templates
3. "Lift" algorithms to iterators
   - More involved concepts, requires clauses
4. Extend to other algorithms:
   - `accumulate(), copy()`

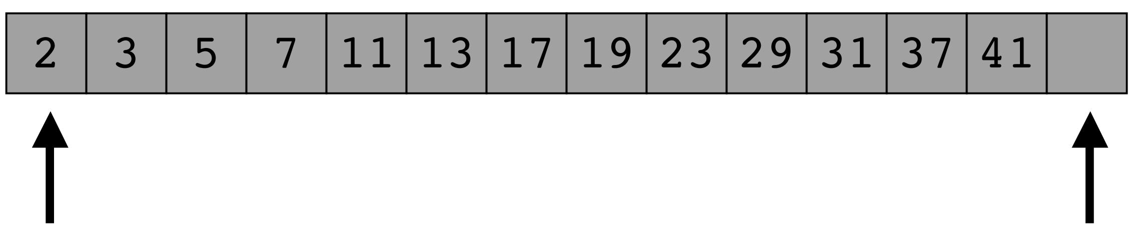
Building a mini-STL

How did it go?

Iterator Concepts

- Iterators abstract the notion of a sequence of values.

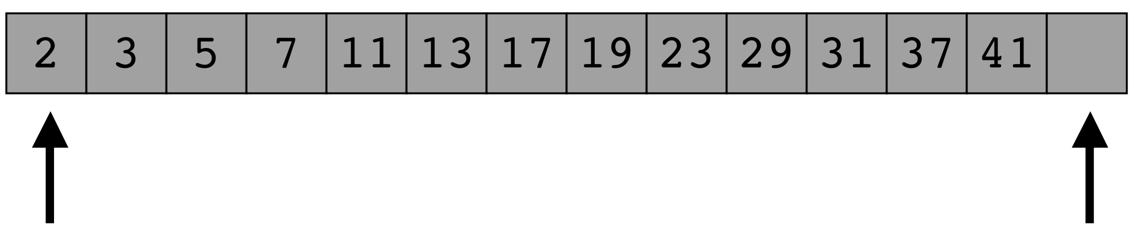
```
concept InputIterator<typename Iter> {



  Iter& operator++(Iter&);      // pre-increment
  Iter operator++(Iter&, int); // post-increment
  bool operator==(Iter, Iter); // equality comparison
  bool operator!=(Iter, Iter); // inequality comparison
     ???     operator*(Iter);  // dereference
};
```



Iterators & Associated Types

- `value_type` is the type that the iterator points to

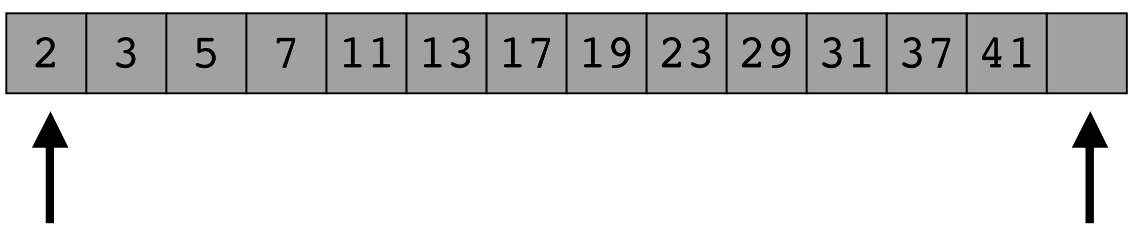
```
concept InputIterator<typename Iter> {
  typename value_type;


  Iter& operator++(Iter&);        // pre-increment
  Iter operator++(Iter&, int); // post-increment
  bool operator==(Iter, Iter); // equality comparison
  bool operator!=(Iter, Iter); // inequality comparison
  value_type operator*(Iter);  // dereference
};
```



Associated Requirements

- `difference_type` measures sequence length

```
concept InputIterator<typename Iter> {
  typename value_type;
  typename difference_type;
  requires SignedIntegral<difference_type>;
  Iter& operator++(Iter&);       // pre-increment
  Iter operator++(Iter&, int); // post-increment
  bool operator==(Iter, Iter); // equality comparison
  bool operator!=(Iter, Iter); // inequality comparison
  value_type operator*(Iter);  // dereference
};
```



Using Associated Types

- Implementing the STL `find` with concepts:

```
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;
}
```

Concept Maps

- We want to call `find` with an array of integers:

```
bool contains(int* array, int n, int value) {
  return find(array, array + n, value) != array + n;
}
```

- *Concept maps* satisfy concept constraints:

```
concept_map InputIterator<int*> {
  typedef int       value_type;
  typedef ptrdiff_t difference_type;
}
```

Concept Maps

- We want to call `find` with an array of integers:

```
bool contains(int* array, int n, int value) {
  return find(array, array + n, value) != array + n;
}
```

- *Concept maps* satisfy concept constraints:

```
template<typename T>
concept_map InputIterator<T*> {
  typedef T         value_type;
  typedef ptrdiff_t difference_type;
}
```

Advanced Concepts Features

Some Syntactic Sugar

```
template<InputIterator Iter, typename T>
requires EqualityComparable<Iter::value_type, T>
Iter find(Iter first, Iter last, const T& value);
```

- is equivalent to:

```
template<typename Iter, typename T>
requires
  InputIterator<Iter> &&
  EqualityComparable<InputIterator<Iter>::value_type,
                     T>
Iter find(Iter first, Iter last, const T& value);
```

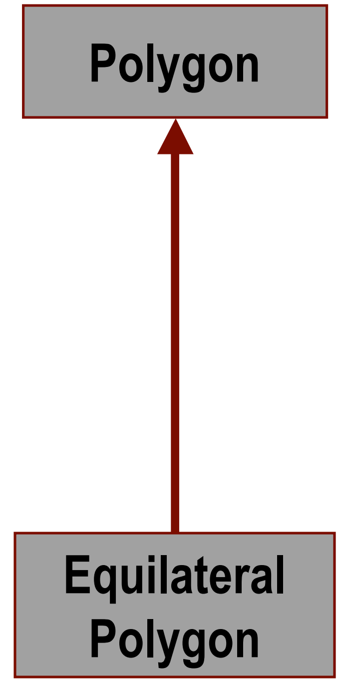
Concept Refinement

- Associated requirements let you aggregate concepts
- Concept *refinement* lets you express a more direct, hierarchical relationship
  - e.g., every `RandomAccessIterator` is a `BidirectionalIterator`
  - can think of it as "concept inheritance"

Concept Refinement

- Let's model a `Polygon` with concepts:

```
concept Polygon<typename Poly> {
  Numeric length_type;
  unsigned num_sides(Poly);
  length_type side_length(Poly, unsigned);
}
```



- And build `EquilateralPolygon`:

```
concept EquilateralPolygon<typename Poly>
          : Polygon<Poly> {
  axiom EqualSides(Poly p, unsigned i, unsigned j) {
    if (i < num_sides(p) && j < num_sides(p))
      side_length(p, i) == side_length(p, j);
  }
}
```

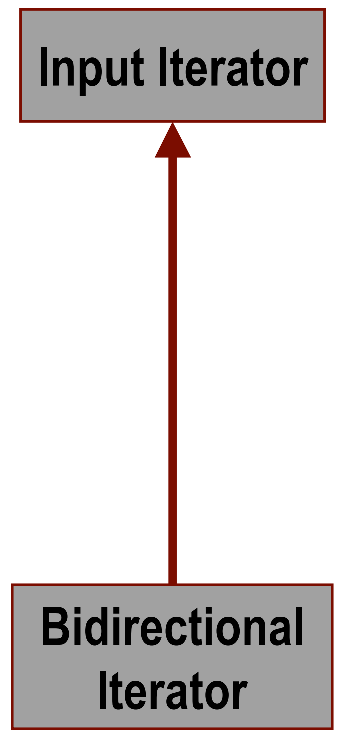
More Concept Refinement

- Remember `BinaryFunction`, with its associated type `result_type`?
- Refine `BinaryFunction` to create a `BinaryPredicate` concept:

```
concept BinaryPredicate<typename F,
                        typename T1, typename T2>
  : BinaryFunction<F, T1, T2>
{
  requires Convertible<result_type, bool>;
}
```

Yet More Concept Refinement

- A bidirectional iterator can move backward:




```
concept BidirectionalIterator<typename Iter>
  : InputIterator<Iter>
{
  Iter& operator--(Iter&);
  Iter operator--(Iter&, int);
}
```

When Should We Refine?

- Two ways of composing concepts:
  - Refinement: Used to express hierarchical, “is-a” relationships.
  - Associated requirements: Used to express “has-a” relationships, requirements on associated types.
- Some technical differences:
  - Refinements are more restricted
  - Refinements “inherit” names
  - Refinements impact concept maps

Concept-Based Overloading

- We express generic algorithms via constrained templates:

```
template<Polygon Poly>
Poly::length_type circumference(Poly const& p) {
  Poly::length_type sum(0);
  for (unsigned i = 0; i < num_sides(p); ++i)
    sum += side_length(p, i);
  return sum;
}
```

- With concepts, we can overload constrained templates:

```
template<EquilateralPolygon Poly>
Poly::length_type circumference(Poly const& p) {
  return num_sides(p) * side_length(p, 0);
}
```

Concept-Based Overloading

```
template<Polygon Poly>
Poly::length_type circumference(Poly const& p); // O(n)

template<EquilateralPolygon Poly>
Poly::length_type circumference(Poly const& p); // O(1)
```

- Which circumference will this call?

```
struct Triangle {... }triangle;
concept_map Polygon<Triangle> { }
circumference(triangle); // O(n)
```

- Which circumference will this call?

```
struct Square {... }square;
concept_map EquilateralPolygon<Square> { }
circumference(square); // O(1)
```

More Than Overloading

- Concept-based overloading isn't overloading
  - It's based on partial ordering of templates
  - Also applicable to class templates

```
template<typename Key, CopyConstructible Value>
   requires EqualityComparable<Key> &&
            CopyConstructible<Key>
   class dictionary {
     list<pair<Key, Value>> storage;
   };

template<LessThanComparable Key, typename Value>
   class dictionary<Key, Value> {
     map<Key, Value> storage;
   };
```

Directing the Partial Ordering

- A template is more specialized than another if
  - It uses refinements of the requirements in the other, or
  - It contains requirements not in the other

- Sometimes, a set of overloads doesn't order very nicely
  - e.g., algorithms of the same name but disjoint requirements
  - Partial ordering results in an ambiguity

! Constraints

- ! constraints state that a specific concept map must *not* exist.

```
template<typename T>
  requires Small<T> void f(const T&); // put on stack
template<typename T>
  requires HeapAllocatable<T> && !Small<T>
  void f(const T&); // put on heap
```

- What if a type is both `Small` and `HeapAllocatable`?
  - Ambiguity!
  - Resolve with a ! constraint.

! Constraints in the Real World

```
template<InputIterator InIter,
         OutputIterator<InIter::value_type> OutIter>
  requires EqualityComparable<InIter::value_type> &&
           Assignable<InIter::value_type> &&
           CopyConstructible<InIter::value_type> &&
           !ForwardIterator<InIter> &&
           !MutableForwardIterator<OutIter>
  OutIter unique_copy(InIter first, InIter last, OutIter result);

template<ForwardIterator InIter,
         OutputIterator<InIter::value_type> OutIter>
  requires EqualityComparable<InIter::reference>
  OutIter unique_copy(InIter first, InIter last, OutIter result);

template<InputIterator InIter, MutableForwardIterator OutIter>
  requires EqualityComparable<OutIter::reference,
                              InIter::value_type> &&
           Assignable<OutIter::reference, InIter::reference> &&
           !ForwardIterator<InIter>
  OutIter unique_copy(InIter first, InIter last, OutIter result );
```

The `SameType` Concept

- Q: How can we state the requirement that two types be equivalent?
- A: We can't. So, it's built-in functionality

```
namespace std {
  concept SameType<typename T1, typename T2>
  { /* unspecified compiler magic */ }
}
```

SameType Example

```
template<InputIterator InIter1,
         InputIterator InIter2,
         OutputIterator<InIter1::value_type> OutIter>
  requires SameType<InIter1::value_type,
                    InIter2::value_type> &&
           LessThanComparable<InIter1::value_type>
  OutIter merge(InIter1 first1, InIter1 last1,
                InIter2 first2, InIter2 last2,
                OutIter result);
```

The `DerivedFrom` Concept

- `DerivedFrom` expresses the notion that a type is publicly derived from another type
  - Again, can't express this directly

```
namespace std {
  concept DerivedFrom<typename Derived,
                      typename Base>
  {
    /* unspecified compiler magic */
  }
}
```

DerivedFrom Example

```
template<typename T>
class shared_ptr {
public:
  template<typename U>
    requires DerivedFrom<U, T>
    explicit shared_ptr(U*);

  template<typename U>
    requires DerivedFrom<T, U>
    operator shared_ptr<U>() const;
};
```

Associated Function Templates

- Use associated function templates to express requirements for a function template:

```
concept Container<typename C> {
  typename value_type;

  template<InputIterator Iter>
    requires Convertible<Iter::value_type,
                         value_type>
    C::C(Iter first, Iter last);
}
```

Default Implementations

- Why does `LessThanComparable` only have <?
  - Sometimes, it is easier to express algorithms using >=, <=, or >
  - <=, >=, > could be defined in terms of <

```
auto concept LessThanComparable<typename T> {
 T operator<(T x, T y);
 T operator>(T x, T y) { return y < x; }
 T operator<=(T x, T y) { return !(y < x); }
 T operator>=(T x, T y) { return !(x < y); }
}
```

Hands-On: Extending and Optimizing STL

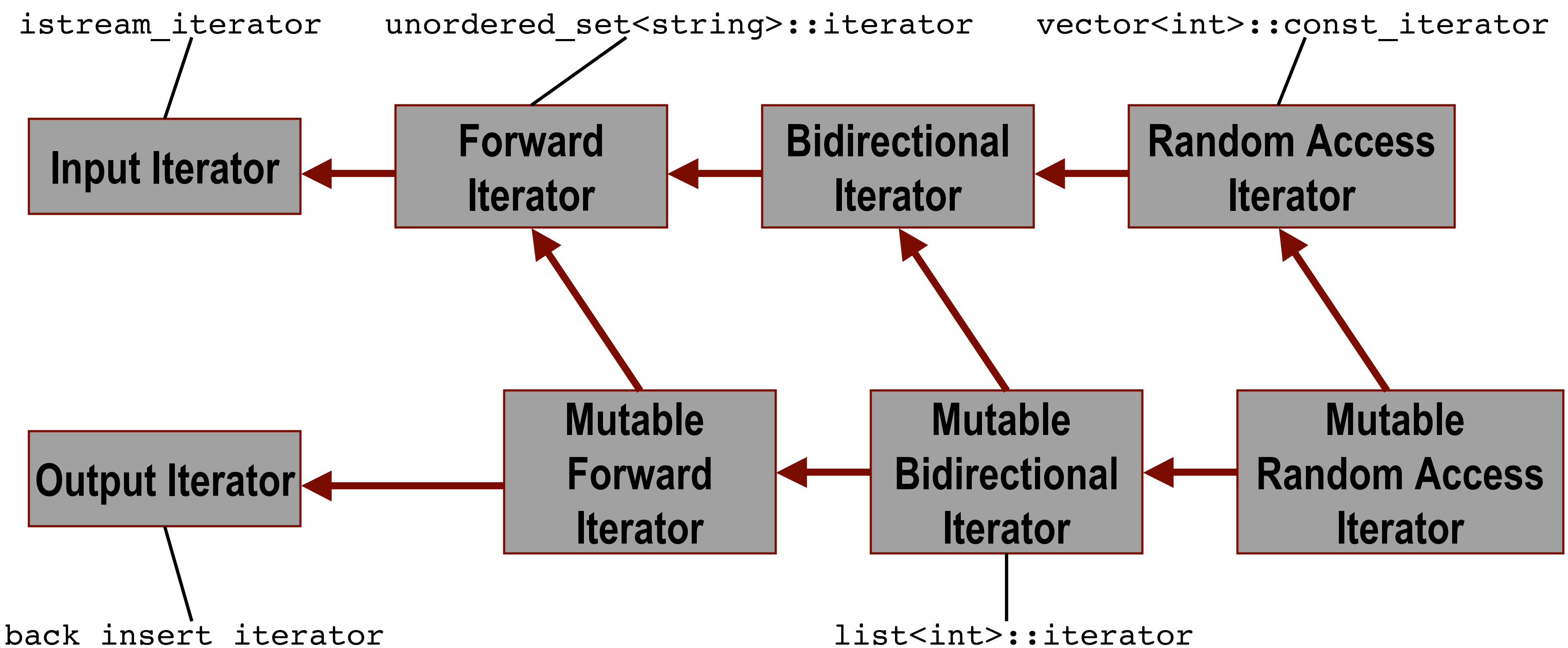
Your Mission...

- Build STL's `advance()` and `distance()`
  - You'll need to build an iterator hierarchy to capture the different variations...
- Build STL's `binary_search()`
  - First with `RandomAccessIterator`s
  - Abstract it to work with `ForwardIterator`s
- When `copy()`ing PODs in contiguous memory, one can use `memmove`()
  - Can you overload `copy()` to do this?
  - Are there other optimizations `copy()` could do?

Extending and Optimizing STL

How did it go?

Iterator Refinements in C++0x



Concept-Based Overloading

- Advance an iterator **x** by **n** steps:

```
template<InputIterator Iter>
void advance(Iter& x, Iter::difference_type n)
{ while (n > 0) { ++x; --n; } } // O(n)

template<BidirectionalIterator Iter>
void advance(Iter& x, Iter::difference_type n) // O(n)
{
  while (n > 0) { ++x; --n; }
  while (n < 0) { --x; ++n; }
}

template<RandomAccessIterator Iter>
void advance(Iter& x, Iter::difference_type n)
{ x = x + n; } // O(1)
```

Concept-Based Overloading

- Just call `advance()` like we always have:

```
list<int>::iterator li = some_list.begin();
advance(li, 17); // calls BidirectionalIterator version

vector<int>::iterator vi = some_vector.begin();
advance(vi, 17); // calls RandomAccessIterator version
```

- Also available inside templates:

```
template<ForwardIterator Iter>
requires LessThanComparable<Iter::value_type>
bool binary_search(Iter f, Iter l, Iter::value_type x)
{
  Iter mid = f;
  ...
  advance(f, distance(f, l) / 2); // O(n) or O(1)
  ...
}
```

Where Do We Go From Here?

You've Seen It All (Almost)

- This tutorial has covered every major feature of concepts
  - Including many minor features
- How will concepts impact C++?
  - Effect on other features
  - Effect on the Standard Library

For-Each Loop in C++0x

- Another likely C++0x feature: for-each loop

```
std::vector<int> values; // fill values...
for (int v : values)
  std::cout << v << ‘ ‘;
```

- We want the for-each loop to work over a variety of containers:
  - Built-in arrays
  - Standard Library containers
  - User-defined containers
  - “Container-like” types

The `Range` Concept

- The C++0x for-each can iterate over any type that meets the requirements of concept `Range`:

```
concept Range<typename X> {
  typename iterator;
  requires InputIterator<iterator>;

  iterator begin(X&);
  iterator end(X&);
}
```


pervasivetechnologylabs
AT INDIANA UNIVERSITY

`Range` Translation Semantics

- A for-each loop like this:

```
vector<int> v;
for (int i : v)
  cout << i << ' ';
```

- Is translated into, e.g.,

```
for(Range<vector<int>>::iterator
      first = Range<vector<int>>::begin(v),
      last = Range<vector<int>>::end(v);
    first != last;
    ++first) {
  int i(*first);
  cout << i << ' ';
}
```

Iteration over Built-In Arrays

- We want to write:

```
int some_primes[6] = { 2, 3, 5, 7, 11, 13 };
for (int prime : some_primes)
  std::cout << prime << ' ';
```

- One (library-defined) concept map suffices:

```
template<typename T, size_t N>
concept_map Range<T[N]> {
  typedef T* iterator;
  T* begin(T array[N]) { return array; }
  T* end(T array[N]) { return array + N; }
}
```

Iteration over Vectors

- One concept map introduces for-each support for vectors:

```
template<typename T>
concept_map Range<vector<T>> {
  typedef vector<T>::iterator iterator;

  iterator begin(vector<T>& vec)
    { return vec.begin(); }

  iterator end(vector<T>& vec)
    { return vec.end(); }
}
```

Iteration over Iterator Sequences

- Example: print all elements equivalent to a given element:

```
std::multiset<std::string, case_insensitive_less>
  elements;
for (std::string s : elements.equal_range("c++0x"))
  std::cout << s << ' ';
```

- Yet another concept map:

```
template<InputIterator Iter>
concept_map Range<pair<Iter, Iter>> {
  typedef Iter iterator;
  Iter begin(pair<Iter, Iter>& p) {return p.first;}
  Iter end(pair<Iter, Iter>& p) {return p.second;}
}
```

Iteration over Containers

- It becomes tedious to write `Range` concept maps for vectors, lists, deques, sets, maps
  - They are already `Containers`
  - One can iterate over any `Container`

```
template<Container C>
concept_map Range<C> {
  typedef Container<C>::iterator iterator;
  iterator begin(C& c) { return c.begin(); }
  iterator end(C& c) { return c.end(); }
}
```

Challenge: Iterate over a Stream

- In Perl, one can iterate over streams:

```
foreach ($line = <STDIN>) {
  // ...
}
```

- Quick exercise:
  - Can you do this with the new for-loop?
  - Can you parse values (e.g., ints) from the stream?

```
concept Range<typename X> {
    typename iterator;
    requires InputIterator<iterator>;

    iterator begin(X&);
    iterator end(X&);
  }
```

My Partial Solution

- `istream_iterator<T>` parses values from an input stream...
- So turn it into a `Range`!

```
template<typename T>
  concept_map Range<istream_iterator<T>> {
    typedef std::istream_iterator<T> iterator;

    iterator begin(istream_iterator<T>& iter) {
      return iter;
    }
    iterator end(istream_iterator<T>&) {
      return std::istream_iterator<T>();
    }

}
```

My Partial Solution - Usage

- Just use the `istream_iterator` as the `Range` argument:

```
int sum = 0;
for (int i : istream_iterator<int>(cin))
  sum += i;
cout << "Sum = " << sum << endl;
```

Concepts in the Standard Library

- The C++ Standard (Template) Library was designed with concepts in mind
  - ... but we didn't have a way to express them
- Evolving the Standard Library with concepts
  - Add concepts instead of "requirements tables"
  - Constrain standard library templates
  - Add concept maps for library types
- Same library, more functionality, easier to use
- Backward compatibility is *very* important

Summary: Concepts for C++0x

- Concepts are a major new feature for C++0x
  - Template type-checking makes templates easier to write and use
  - Concept maps, concept-based overloading make templates more powerful
- Generic Programming for the Masses
  - Easier to build and compose generic libraries
  - Arcane “template tricks” become unnecessary
- Status: Very, very, very likely for C++0x

Questions?

http://www.generic-programming.org/languages/conceptcpp

http://www.generic-programming.org/software/ConceptGCC

Doug Gregor <dgregor@osl.iu.edu>