Building High-Performance Generic Libraries

Truth and Beauty
for
Fun and Profit!

Purpose of Course

- Understand, appreciate, and apply *the generic design process*.
- Gain insight into
  - correctness
  - efficiency
  - reusability
  - your own use of STL
  - abstraction
  - truth and beauty in programming
- Think algorithmically

Generic Library Design

- Developed using a particular process
  - Starts with efficient non-generic code
  - Ends with efficient, flexible, reusable template library
  - Well-defined set of steps
  - But not mechanical!
- Why study this?
  - You may want to write one
  - or use one effectively
  - C++0x language support
  - It’s good for you. Eat your vegetables, too.

Getting There

- Use *the process* on small examples
- Relate these examples to the STL
- Apply *the process* in a dimension not covered by STL

Understanding “Algorithm”

- What is a computer program?

> “… a collection of instructions that describes a task, or set of tasks, to be carried out by a computer.”
>
> -- Wikipedia

- What is an algorithm?

> “… a finite list of well-defined instructions for accomplishing some task that, given an initial state, will terminate in a defined end-state.”
>
> -- Wikipedia

Evolution of Containers

- Containers: pre-1994

```
class Container : public Object {
    virtual Object Get(int) = 0;
    virtual void Sort() = 0;
};
class Vector : public Container { … };
```

“Object Oriented”

- Containers: post-1994

```
std::vector<int> v;
std::sort(v.begin(), v.end());
```

“Algorithm Oriented”

Selective Ignorance

Abstraction is the act of taking several ideas that resemble each other—in other words, several ideas that are similar but not identical—and removing everything but what the ideas have in common.

…Selective ignorance—abstraction—lets us deal with problems that we would otherwise be unable to manage."

-- Andrew Koenig

Why "Algorithm-Centric?"

- We have many abstractions for program structure: functions, classes, modules, etc…
- Algorithm: abstraction of *what the program does*
  - Make the steps of your program explicit.
  - Make it easier to reason about the *performance* of your code.
  - Are easier to verify for *correctness*

Generic Design Step 1

Begin Concretely

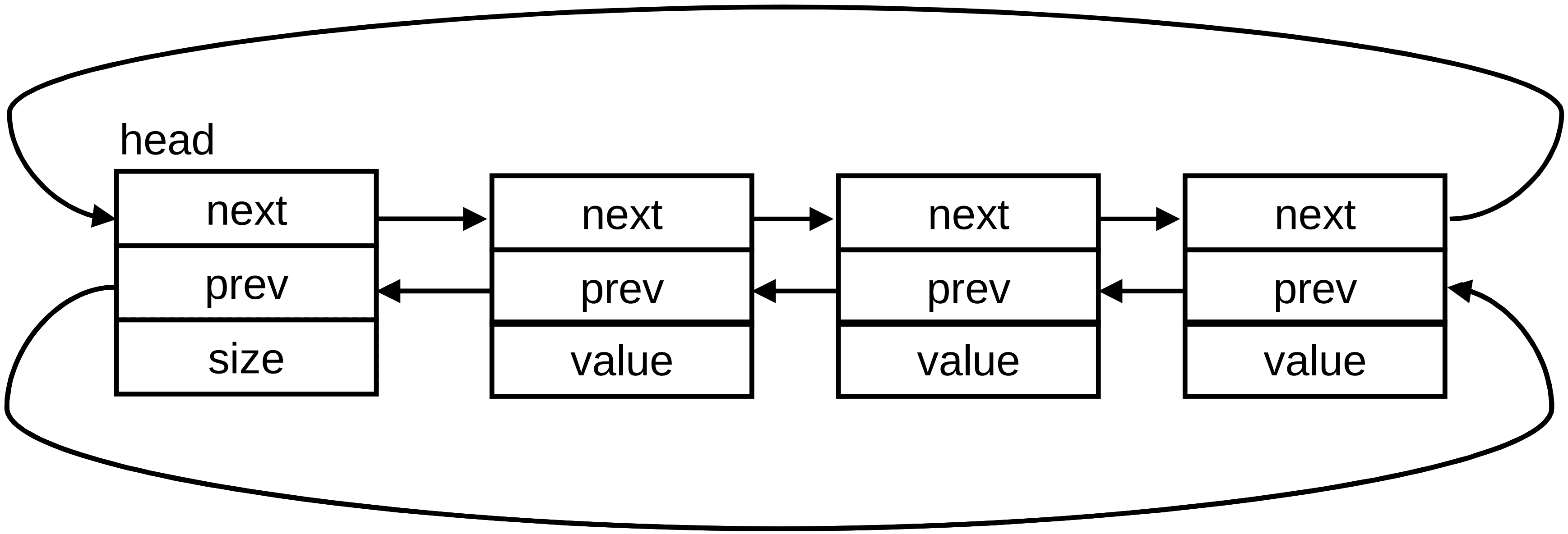
Survey Concrete Implementations

- Principle: flexibility and generality in software should solve real problems
- Generic libraries come from non-generic implementations
- Begin by collecting all the concrete implementations you can find in the domain
- Our scope is *totally unrealistic!*
- In reality, strive for completeness

Programming Challenge #1

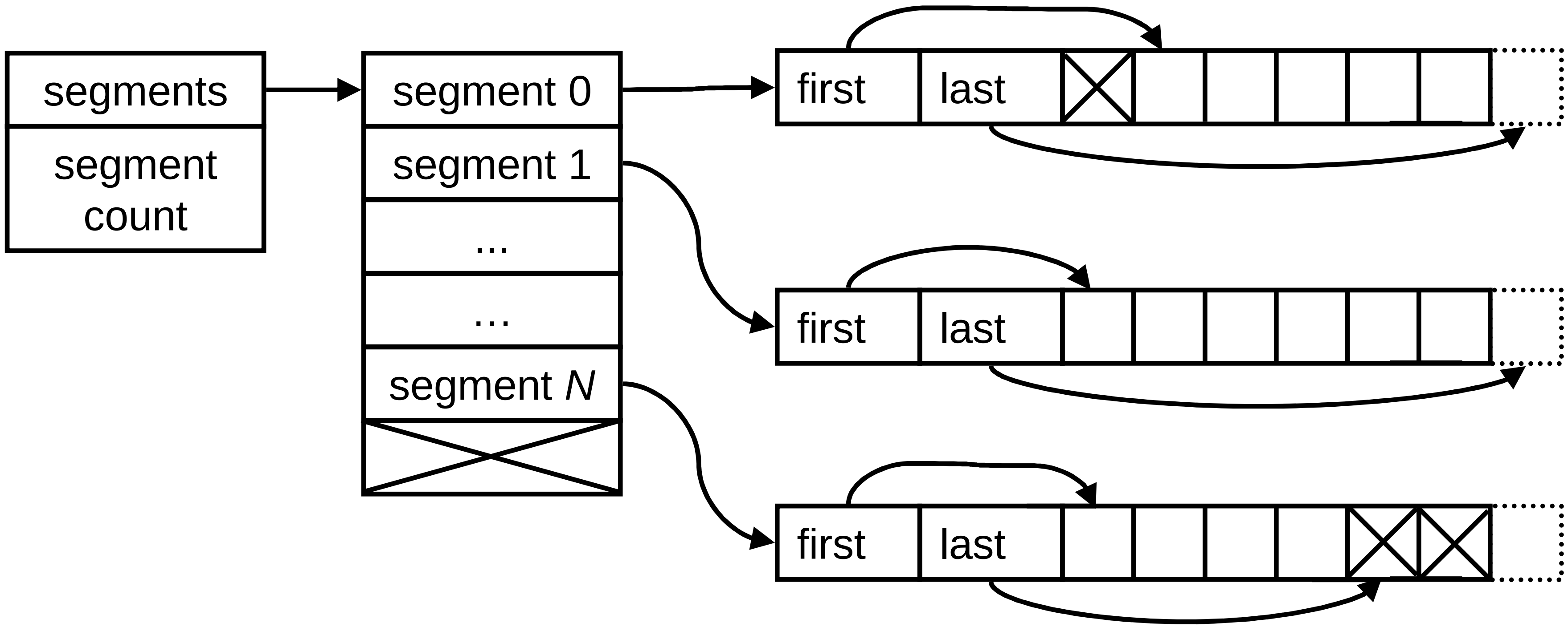
- Implement the algorithm for the data structure you've been given.
- No templates allowed.
- Make it fast!
- It's not a trick question. These are easy.
- https://boost-consulting.com/trac/projects/boostcon

Doubly-Linked List



- Efficiently growable at front and back.
- Linear access to individual elements.

Deque



•Fast random access, like array.
•Efficiently growable at front and back, like list.

Exercise 1

(in progress)

https://boost-consulting.com/trac/projects/boostcon

obj.Transform(fun, out)

```
int* List
  ::Transform(F f, int* o)
{
    typedef ListNode N;
    N* n = head.next;
    while(n != &head) {
        *o++ = f(n->value);
        n = n->next;
    }
    return o;
}
```

```
int* Deque
  ::Transform(F f, int* o)
{
    typedef DequeSegment S;
    S** s = segments;
    S** s2 = s + count;
    for(; s != s2; ++s)
    {
        int* last = (*s)->last;
        int* j = (*s)->first;
        for(; j != last; ++j)
            *o++ = f(*j);
    }
    return o;
}
```

```
// Array
int* Transform(int* in, int n,
   F f, int* o)
{
    for(int i=0; i < n; ++i)
    {
        *o++ = fun(in[i]);
    }
    return o;
}
```

Each algorithm has its own implementation for each data structure

The "NxM Problem"

N Sequence Types

M Algorithms

	`List`	`Deque`	`Array`
`GetAt`			
`Find`			
`Transform`			

NxM Implementations

Riddle me this, Batman…

Why doesn't the STL have this problem?

obj.Transform(fun, out)

```
int* List
  ::Transform(F f, int* o)
{
    typedef ListNode N;
    N* n = head.next;
    while(n != &head) {
        *o++ = f(n->value);
        n = n->next;
    }
    return o;
}
```

```
int* Deque
  ::Transform(F f, int* o)
{
    typedef DequeSegment S;
    S** s = segments;
    S** s2 = s + count;
    for(; s != s2; ++s)
    {
        int* last = (*s)->last;
        int* j = (*s)->first;
        for(; j != last; ++j)
            *o++ = f(*j);
    }
    return o;
}
```

```
// Array
int* Transform(int* in, int n,
   F f, int* o)
{
    for(int i=0; i < n; ++i)
    {
        *o++ = fun(in[i]);
    }
    return o;
}
```

Transform(begin, end, f, o)

```
template< class Iter, class Fun, class Out >
Out transform( Iter begin, Iter end, Fun f, Out o )
{
    while( begin != end )
    {
        *o = f( *begin );
        ++o;
        ++begin;
    }
    return o;
}
```

Why Iterators?

N Sequence Types

M Algorithms

	List	Deque	Array
GetAt			
Find			
Transform			

N Iterator Types

\+

M Algorithms

Generic Design Step 2

"Lifting" to a higher level of abstraction

Lifting: From Specific to General

- Begin with concrete algorithms

```
void Fill( int* array, int size, int value )
{
    for( int i = 0; i < size; ++i )
    {
        array[i] = value;
    }
}
```

```
void List::Fill( int value )
{
    ListNode* begin = head.next;
    ListNode* end = &head;
    while( begin != end )
    {
        begin->value = value;
        begin = begin->next;
    }
}
```

Lifting: From Specific to General

- Eliminate superficial differences

```
void Fill( int* array, int size, int value )
{
    for( int i = 0; i < size; ++i )
    {
        array[i] = value;
    }
}
```

```
void List::Fill( int value )
{
    ListNode* begin = head.next;
    ListNode* end = &head;
    while( begin != end )
    {
        begin->value = value;
        begin = begin->next;
    }
}
```

Lifting: From Specific to General

- Eliminate superficial differences

```
void Fill( int* array, int size, int value )
{
    int* begin = array;
    int* end = array + size;
    while( begin != end )
    {
        *begin = value;
        ++begin;
    }
}
```

```
void List::Fill( int value )
{
    ListNode* begin = head.next;
    ListNode* end = &head;
    while( begin != end )
    {
        begin->value = value;
        begin = begin->next;
    }
}
```

Lifting: From Specific to General

- Eliminate syntactic differences, too

```
int& Get( int* p ) { return *p; }
void Next( int*& p ) { ++p; }

void Fill( int* begin, int* end, int value )
{
    while( begin != end )
    {
        Get(begin) = value;
        Next(begin);
    }
}
```

```
int& Get( ListNode* p ) { return p->value; }
void Next( ListNode*& p ) { p = p->next; }

void Fill( ListNode* begin, ListNode* end,
            int value )
{
    while( begin != end )
    {
        Get(begin) = value;
        Next(begin);
    }
}
```

> *"All problems in computer science can be solved by another level of indirection."* -- Butler Lampson

Lifting: From Specific to General

- Write a template

```
int& Get( int* p ) { return *p; }
void Next( int*& p ) { ++p; }
```

```
int& Get( ListNode* p ) { return p->value; }
void Next( ListNode*& p ) { p = p->next; }
```

```
template< class Iterator, class T >
void Fill( Iterator begin, Iterator end, T value )
{
    while( begin != end )
    {
        Get(begin) = value;
        Next(begin);
    }
}
```

Programming Challenge #2

- Pair up!
- Go to wiki and download code
  - https://boost-consulting.com/trac/projects/boostcon
- Implement same algo using Iterators
- Things to keep in mind:
  - Is the generic algorithm fundamentally the same as the non-generic (aside from syntax)?
  - Is the generic algorithm as fast as the non-generic?
  - What unstated assumptions do the generic algorithms make about their arguments?

Exercise 2

(in progress)

https://boost-consulting.com/trac/projects/boostcon

Don't Forget ...

- Is the generic algorithm fundamentally the same as the non-generic (aside from syntax)?
- Is the generic algorithm as fast as the non-generic?
- What unstated assumptions do the generic algorithms make about their arguments?

Generic Design Step 3

Find Minimal Requirements

Why look at requirements?

- Goal
  - Discover core abstractions in domain
- How?
  - Analyze each algorithm implementation in turn
  - List the assumptions each implementation <u>must</u> make in order to “work”
  - Compare lists and factor out deeper abstractions
- Minimal Assumptions → Abstractions

Find Minimal Requirements

```
template<typename Iterator, typename Value>
Iterator Find(Iterator begin, Iterator end, Value value)
{
    while(begin != end)  {
        if(Get(begin) == value)
            return begin;
        Next(begin);
    }
    return end;
}
```

valid expression	type	complexity	preconditions	semantics/postcondition

Find Minimal Requirements

```
template<typename U, typename V>
U Find(U begin, U end, V value)
{
    while(begin != end)  {
        if(Get(begin) == value)
            return begin;
        Next(begin);
    }
    return end;
}
```

a: object of type **U**

b: object of type **V**

valid expression	type	complexity	preconditions	semantics/postcondition
U y(a) U y = a	U	O(1)	a nonsingular	y is equivalent to a
V z(b) V z = b				

Find Minimal Requirements

```
template<typename U, typename V>
U Find(U begin, U end, V const& value)
{
    while(begin != end)  {
        if(Get(begin) == value)
            return begin;
        Next(begin);
    }
    return end;
}
```

a: object of type **U**

valid expression	type	complexity	preconditions	semantics/postcondition
U y(a) U y = a	U	O(1)	a nonsingular	y is equivalent to a

Find Minimal Requirements

```
template<typename U, typename V>
U Find(U begin, U end, V const& value)
{
    while(begin != end)  {
        if(Get(begin) == value)
            return begin;
        Next(begin);
    }
    return end;
}
```

a, b: object of type **U**

valid expression	type	complexity	preconditions	semantics/postcondition
U y(a) U y = a	U	O(1)	a nonsingular	y is equivalent to a
a != b	convertible to bool	O(1)	a,b in same sequence	!= is an inverse equivalence relation

Find Minimal Requirements

```
template<typename U, typename V>
U Find(U begin, U end, V const& value)
{
    while(begin != end) {
        if(Get(begin) == value)
            return begin;
        Next(begin);
    }
    return end;
}
```

a, b: object of type **U**

X: **U**'s *value type*

valid expression	type	complexity	preconditions	semantics/postcondition
a != b	convertible to bool	O(1)	a,b in same sequence	!= is an inverse equivalence relation
Get(a)	X, X&, or X const&	O(1)	a not past-the-end	Get() is a *regular function*

Find Minimal Requirements

```
template<typename U, typename V>
U Find(U begin, U end, V const& value)
{
    while(begin != end)  {
        if(Get(begin) == value)
            return begin;
        Next(begin);
    }
    return end;
}
```

a, b: object of type **U**

X: **U**'s *value type*

r: object of type **X**

s: object of type **V const**

valid expression	type	complexity	preconditions	semantics/postcondition
Get(a)	X, X&, or X const&	O(1)	a not past-the-end	Get() is a *regular function*
r == s	convertible to bool	O(1)	(r,s) in the domain of ==	??

Find Minimal Requirements

```
template<typename U, typename V>
U Find(U begin, U end, V const& value)
{
    while(begin != end)  {
        if(Get(begin) == value)
            return begin;
        Next(begin);
    }
    return end;
}
```

a, b: object of type **U**

X: **U**'s *value type*

r: object of type **X**

s: object of type **V const**

valid expression	type	complexity	preconditions	semantics/postcondition
r == s	convertible to bool	O(1)	(r,s) in the domain of ==	??
Next(a)		O(1)	a not past-the-end	U z(a); Next(a); assert(a!=z); Next(z); assert(a==z)

Find Minimal Requirements

a, b: object of type **U**

X: **U**'s *value type*

s: object of type **V const**

r: object of type **X**

valid expression	type	complexity	preconditions	semantics/postcondition
U y(a) U y = a	U	O(1)	a nonsingular	y is equivalent to a
a != b	convertible to bool	O(1)	a,b in same sequence	!= is an inverse equivalence relation
Get(a)	X, X&, or X const&	O(1)	a not past-the-end	Get() is a *regular function*
r == s	convertible to bool	O(1)	(r,s) in the domain of ==	??
Next(a)		O(1)	a not past-the-end	U z(a); Next(a); assert(a!=z); Next(z); assert(!(a!=z))

A Semantic Problem

- When x and y have the same type, == is an equivalence relation
  - reflexivity: x == x
  - commutativity: x == y ⇔ y == x
  - transitivity: x == y && y == z → x == z
- These properties give meaning to the idea of "finding a value in a sequence."
- In English: we don't know what it means to find a value of one type in a sequence of another type.

Making Find Meaningful

- Make it possible for == to be an equivalence relation

```
template<typename U, typename V>
U Find(U begin, U end, V const& value)
{
    while(begin != end)  {
        V const& x = Get(begin); // convert to V
        if(x == value)             // compare Vs
            return begin;
        Next(begin);
    }
    return end;
}
```

- Now find 1 in the sequence 0.8, 1.2, 2.3, 1.0
- No, really, we don’t know what this means!

Making Find Meaningful

- Make it possible for == to be an equivalence relation

```
template<typename U>
U Find(
    U begin, U end,
    typename ValueType<U>::type const& value)
{
    while(begin != end)  {
        if(Get(begin) == value)
            return begin;
        Next(begin);
    }
    return end;
}
```

Find Minimal Requirements

a, b: object of type **U** **r**, **s:** objects of type **X**

X: **U**'s *value type*

valid expression	type	complexity	preconditions	semantics/postcondition
U y(a) U y = a	U	O(1)	a nonsingular	y is equivalent to a
a != b	convertible to bool	O(1)	a,b in same sequence	!= is an inverse equivalence relation
Get(a)	X, X&, or X const&	O(1)	a not past-the-end	Get() is a *regular function*
r == s	convertible to bool	O(1)	(r,s) in the domain of ==	== is an equivalence relation
Next(a)		O(1)	a not past-the-end	U z(a); Next(a); assert(a!=z); Next(z); assert(a==z)

Take-Aways

- The process of lifting requirements will reveal unintended constraints
- It will also reveal unintended generality
- Semantics count
- Generalize as far as possible, but not so far that you can't say what the algorithm does anymore.

Are *Minimal* Requirements Sensible?

a, b: object of type **U**

r, **s:** objects of type **X**

X: **U**'s *value type*

valid expression	type	complexity	preconditions	semantics/postcondition
U y(a) U y = a	U	O(1)	a nonsingular	y is equivalent to a
a != b	convertible to bool	O(1)	a,b in same sequence	!= is an inverse equivalence relation
r == s	convertible to bool	O(1)	(r,s) in the domain of ==	== is an equivalence relation
Get(a)	X, X&, or X const&	O(1)	a not past-the-end	Get() is a *regular function*
Next(a)		O(1)	a not past-the-end	U z(a); Next(a); assert(a!=z); Next(z); assert(a==z)

Are *Minimal* Requirements Sensible?

a, **b**: object of type **U**

r, **s:** objects of type **X**

X: **U**'s *value type*

valid expression	type	complexity	preconditions	semantics/postcondition
U y(a) U y = a	U	O(1)	a nonsingular	y == a
a != b a == b	convertible to bool	O(1)	a,b in same sequence	== is an equivalence relation; a==b⇔!(a!=b)
r == s r != s	convertible to bool	O(1)	(r,s) in the domain of ==	== is an equivalence relation; r==s⇔!(r!=s)
Get(a)	X, X&, or X const&	O(1)	a not past-the-end	Get() is a *regular function*
Next(a)		O(1)	a not past-the-end	U z(a); Next(a); assert(a!=z); Next(z); assert(a==z)

Riddle me this, Batman…

What makes it OK to "cluster" equality and inequality?

Generic Design Step 4

Lifting and Clustering Concepts

Why look at requirements?

- Goal
  - Discover core abstractions in domain
- How?
  - Analyze each algorithm implementation in turn
  - List the assumptions each implementation <u>must</u> make in order to "work"
  - Compare lists and factor out deeper abstractions
- Minimal Assumptions → Abstractions

Requirements for GetAt(p, n)

a, b: object of type **U**

r: object of type **X**

X: U’s *reference type*

valid expression	type	complexity	preconditions	semantics/postcondition
U y(a) U y = a	U	O(1)	a nonsingular	y equivalent to a
X s(r) X s = r	X	O(1)	r nonsingular	r equivalent to s
Get(a)	X	O(1)	a not past-the-end	Get() is a *regular function*
Next(a)		O(1)	a not past-the-end	U z(a); Next(a); assert(a!=z); Next(z); assert(a==z)

[p, p^{n+1}) is a valid range, where p^i is the value of p after i applications of Next

Requirements for Find(p,q,v)

a, b: object of type **U** **r**, **s:** objects of type **X**

X: U's *value type*

valid expression	type	complexity	preconditions	semantics/postcondition
U y(a) U y = a	U	O(1)	a nonsingular	y == a
a != b a == b	convertible to bool	O(1)	a,b in same sequence	== is an equivalence relation; a==b⇔!(a!=b)
r == s r != s	convertible to bool	O(1)	(r,s) in the domain of ==	== is an equivalence relation; r==s⇔!(r!=s)
Get(a)	X, X&, or X const&	O(1)	a not past-the-end	Get() is a *regular function*
Next(a)		O(1)	a not past-the-end	U z(a); Next(a); assert(a!=z); Next(z); assert(a==z)

[p, q) is a valid range

Riddle me this, Batman…

Do we have one iterator concept here, or two?

Iterator Requirements

a, b: object of type **U**   **r:** object of type **X**

X: U's *reference type*

valid expression	type	complexity	preconditions	semantics/postcondition
U y(a) U y = a	U	O(1)	a nonsingular	y == a
a != b a == b	convertible to bool	O(1)	a,b in same sequence	== is an equivalence relation; a==b⇔!(a!=b)
X s(r) X s = r	X	O(1)	r nonsingular	r equivalent to s
Get(a)	X	O(1)	a not past-the-end	Get() is a *regular function*
Next(a)		O(1)	a not past-the-end	U z(a); Next(a); assert(a!=z); Next(z); assert(a==z)

Iterator Requirements

a, b: object of type **U**   **r:** object of type **X**

X: U's *reference type*

valid expression	type	complexity	preconditions	semantics/postcondition
U y(a) U y = a	U	O(1)	a nonsingular	y == a
a != b a == b	convertible to bool	O(1)	a,b in same sequence	== is an equivalence relation; a==b⇔!(a!=b)
X s(r) X s = r	X	O(1)	r nonsingular	r equivalent to s
Get(a)	X	O(1)	a not past-the-end	Get() is a *regular function*
Next(a)		O(1)	a not past-the-end	U z(a); Next(a); assert(a!=z); Next(z); assert(a==z)

Lifting EqualityComparable

a, b, c: object of type **U**

valid expression	type	complexity	preconditions	semantics/postcondition
a == b	convertible to bool	O(1)	a,b in the domain of ==	== is an equivalence relation: a==a a==b⇔b==a, a==b&&b==c ⇨ a==c
a != b	convertible to bool	O(1)	a,b in the domain of ==	a==b⇔!(a!=b)

Iterator Requirements

(in addition to EqualityComparable)

a: object of type **U**

r: object of type **X**

X: U's *reference type*

valid expression	type	complexity	preconditions	semantics/postcondition
U y(a) U y = a	U	O(1)	a nonsingular	y == a
Get(a)	X	O(1)	a not past-the-end	Get() is a *regular function*
Next(a)		O(1)	a not past-the-end	U z(a); Next(a); assert(a!=z); Next(z); assert(a==z)

Algorithm Concept Requirements

- GetAt(Iterator start, int n)
  - Iterator is an Iterator
  - [start, start^{n+1}) is a valid range

- Find(Iterator start, Iterator finish, ValueType<Iterator>::type const & val)
  - [start, finish) is a valid range
  - Iterator's *value type* is EqualityComparable

The GetAt() Conundrum

- How do you lift the GetAt() algorithm?

List:

```
int& List::GetAt(int i)
{
    Node* n = head.next;
    for(; ; n = n->next, --i)
    {
        if(0 == i)
            return n->value;
    }
}
```

Template:

```
template< class Iterator >
??? GetAt( Iterator iter, int i )
{
    for(; ; Next(iter), --i)
    {
        if(0 == i)
            return Get(iter);
    }
}
```

What is the return type?

Associated Types

- A type used to describe the requirements of a concept.
- E.g., the Iterator concept has the Get() requirement:

Iterator Concept

Valid Expression	Type
Get(iter)	*Reference type*, accessible via Reference< Iterator >::type

Associated Types and Traits

- Why not this?

```
typename Iterator::Reference
```

```
typename int*::Reference
```

- Use a traits class instead:

```
typename Reference< Iterator >::type
```

The GetAt() Conundrum

- Use a Trait to get to the associated type:

```
template< class Iterator >
typename Reference< Iterator >::type
GetAt( Iterator iter, int i )
{
    for(; ; Next(iter), --i)
    {
        if(0 == i)
            return Get(iter);
    }
}
```

The GetAt() Conundrum, Part Deux

- How do you lift the GetAt() algorithm?

Array:

```
int& GetAt(int* in, int size, int i)
{
   assert(i < size);
   return in[i];
}
```

Fast, O(1)

Template:

```
template< class Iterator >
typename Reference< Iterator >::type
GetAt( Iterator iter, int i )
{
   for(; ; Next(iter), --i)
   {
      if(0 == i)
         return Get(iter);
   }
}
```

Slow, O(N)

The GetAt() Conundrum, Possible Solutions

- Overload GetAt() for pointers?

```
template< class Value >
Value& GetAt(Value* p, int i)
{
    return p[i];
}
```

- Works, but what about Deque::GetAt()?

```
int& Deque::GetAt(int i)
{
    i -= segments[0]->last - segments[0]->first;
    return (i < 0) ?
        *(segments[0]->last + i) :
        *(segments[1+i/Size]->first + i%Size);
}
```

O(1) GetAt Implementations

Array:

```
int& GetAt(int* in, int size, int i)
{ … }
```

Deque:

```
int& Deque::GetAt(int i)
{ … }
```

Vector:

```
int& Vector::GetAt(int i)
{ … }
```

Others ….

Lift!

Template:

```
template< class O1Iter >
typename Reference< O1Iter >::type
O1GetAt( O1Iter iter, int i )
{
    Skip( iter, i );
    return Get( iter );
}
```

Skip() is a new requirement.

Concept Refinement

- Adding requirements to a concept results in a more powerful concept: a *refinement*.
  - E.g., Random Access Iterator
- Not all models can or should meet the new requirements.
  - E.g., Forward Iterator

Concept Refinement

- RandomIterator
  - satisfies the requirements for ForwardIterator …
  - … and the following expressions are valid:

Valid Expression	Type	Complexity	Preconditions	Semantics/Postcondition

Skip(a, n)		O(1)	a not past-the-end; at least n-1 successors to a in sequence	U y(a);
				Skip(a, n);
				assert(a != y || n == 0);
				for(; n--; Next(y));
				assert(a == y);

Generic Design Step 5

Algorithm Specialization

Algorithm Specialization

```
template< class InIter >
typename Reference< InIter >::type
GetAt( InIter iter, int i )
{
    for( ; 0 != i; Next(iter), --i );
    return Get( iter );
}
```

```
template< class RAIter >
typename Reference< RAIter >::type
GetAt( RAIter iter, int i )
{
    Skip( iter, i );
    return Get( iter );
}
```

Error! Overload resolution is ambiguous.

Algorithm Specialization

```
template< class InIter >
typename Reference< InIter >::type
GetAt( InIter iter, int I, Input )
{
    for( ; 0 != i; Next(iter), --i );
    return Get( iter );
}
```

```
template< class RAIter >
typename Reference< RAIter >::type
GetAt( RAIter iter, int I, Random )
{
    Skip( iter, i );
    return Get( iter );
}
```

```
template< class InIter >
typename Reference< InIter >::type
GetAt( InIter iter, int i )
{
    return GetAt( iter, i, Category( iter ) );
}
```

Tag Dispatching

- Query a type for the concept it models
- Manually dispatch to appropriate algorithm

```
// Empty types that represent the concepts
// Inheritance reflects refinement hierarchy; a trick to simplify dispatch
struct Input {};
struct Forward : Input {};
struct Random : Forward {};
```

Valid Expression	Type	Complexity	Pre-conditions	Semantics Post-conditions
Category(a)	One of: - Input - Forward - Random	O(1)		

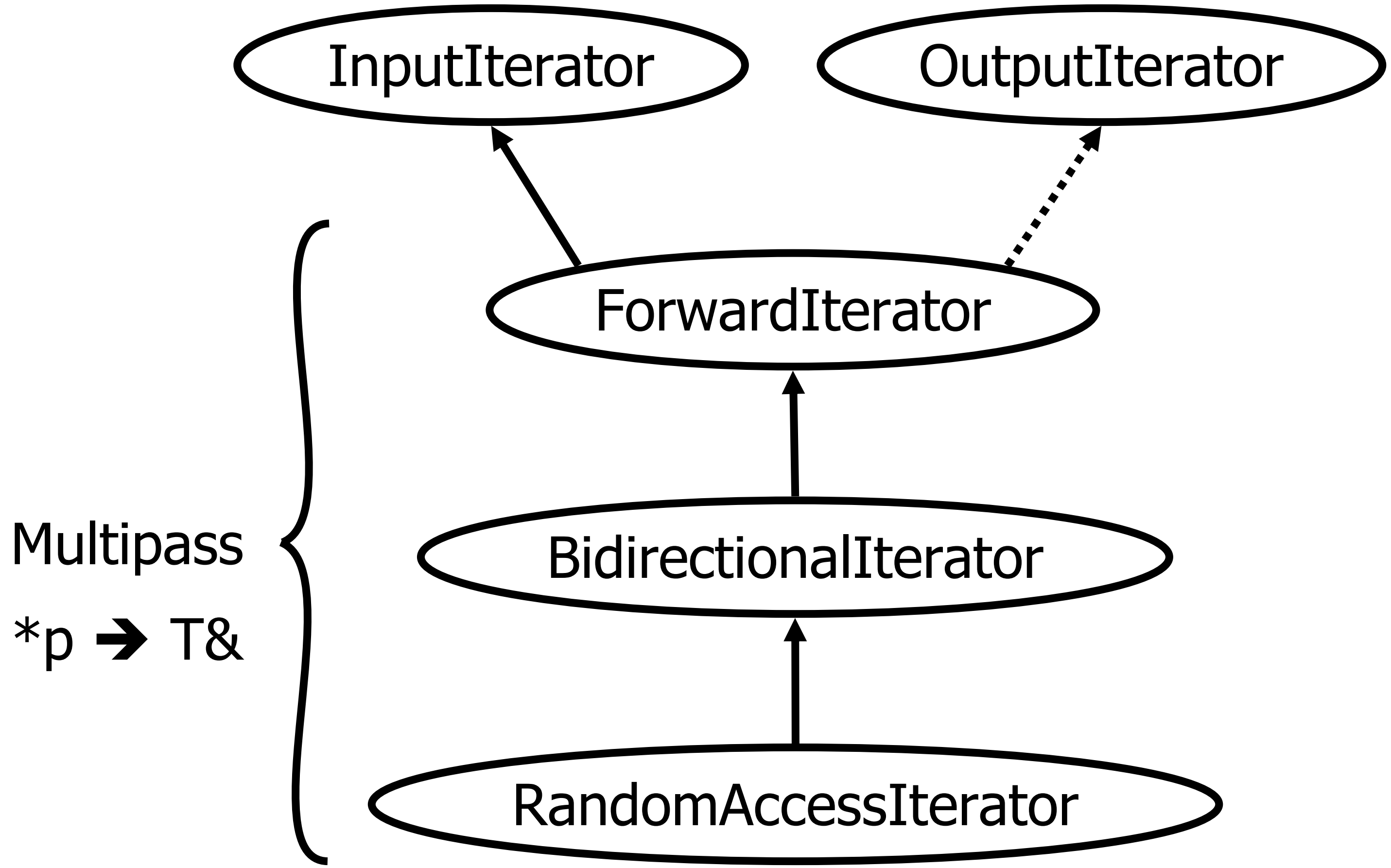
Refinement Summary

- Some algorithms have multiple implementations; e.g., a fast and a slow
- Express the faster algorithms in terms of a more powerful concept refinement
- Concept refinements form a hierarchy
- Use traits and tag dispatching to select the optimal algorithm
  - or wait for C++0x

C++ Iterators: the Pointer Model

- writability:
  - `*p = x`
- copying/comparison:
  - `q = p;        assert( p == q)`
- multipass:
  - `q = p;        x = *p++;     assert( x == *q )`
- reverse traversal:
  - `--p`
- random access:
  - `p += n;    n = p - q;      p < q`

Iterator Concept Hierarchy



Wouldn't a Simpler Iterator Do?

- Java:

```
Iterator iter = c.iterator();
while (iter.hasNext())
{
    Object obj = iter.next();
    // do something with obj
}
```

- This is essentially an istream
- Most other languages do something similar

C++ Iterators: the Pointer Model

- writability:
  - *p = x
- copying/comparison:
  - q = p;   assert(p
- multipass:
  - q = p;   x = *p++ … q)
- reverse traversal:
  - --p
- random access:
  - p += n;   n = p - q;   p < q

Writability

- Syntax:
  - *p = x
- Needed For: any mutating algorithm
  - quicksort
  - transform
  - rotate
  - …

Copying and Comparison

- Syntax/semantics:
  - **q = p**; assert(**p == q**)
- a non-destructive find must modify and return a <u>copy</u>
- comparison is needed to make copy meaningful

Consider Quicksort

4	5	1	7	3	6	0	8	2

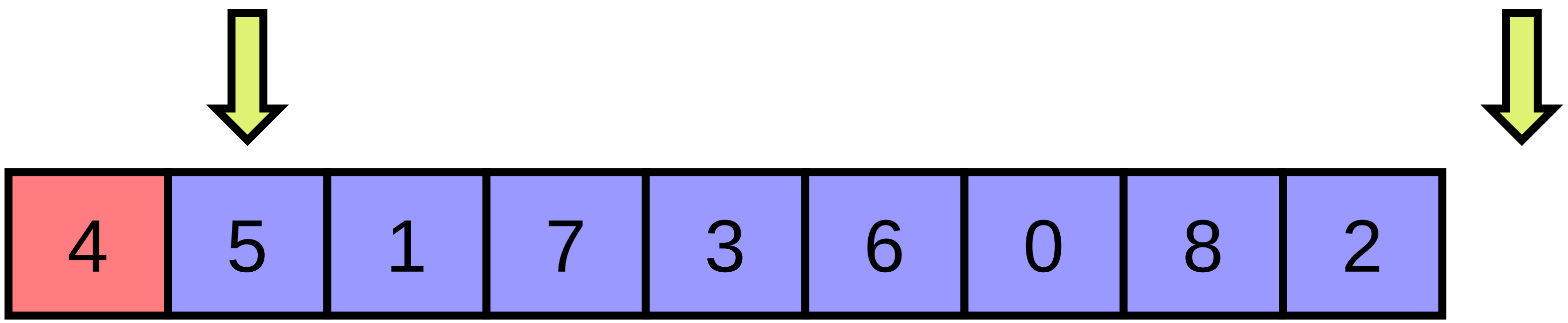
Consider Quicksort

1. Choose pivot element

4	5	1	7	3	6	0	8	2

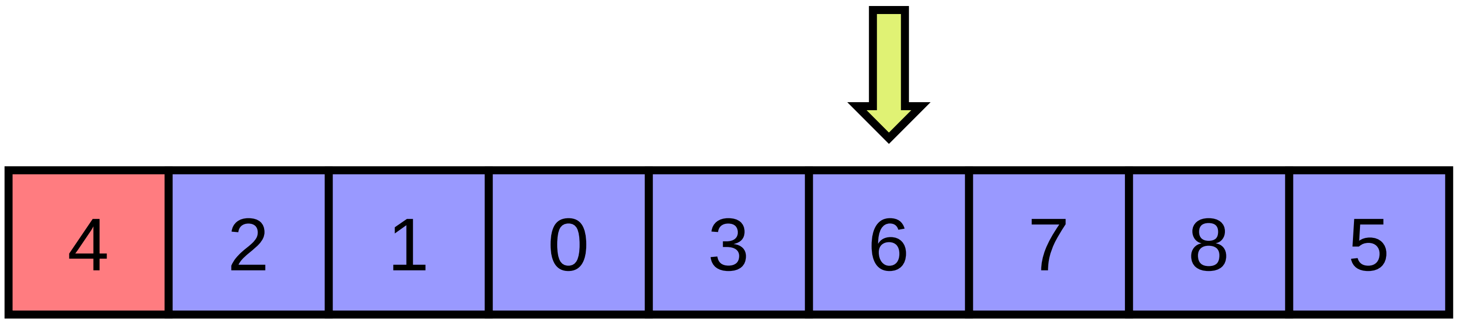
Consider Quicksort

1. Choose pivot element
2. Partition rest of sequence on that value



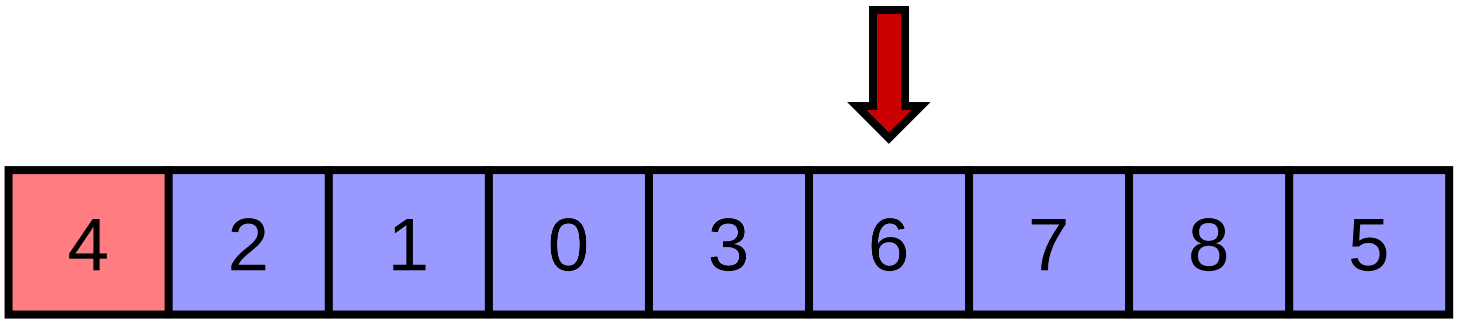
Consider Quicksort

1. Choose pivot element
2. Partition rest of sequence on that value



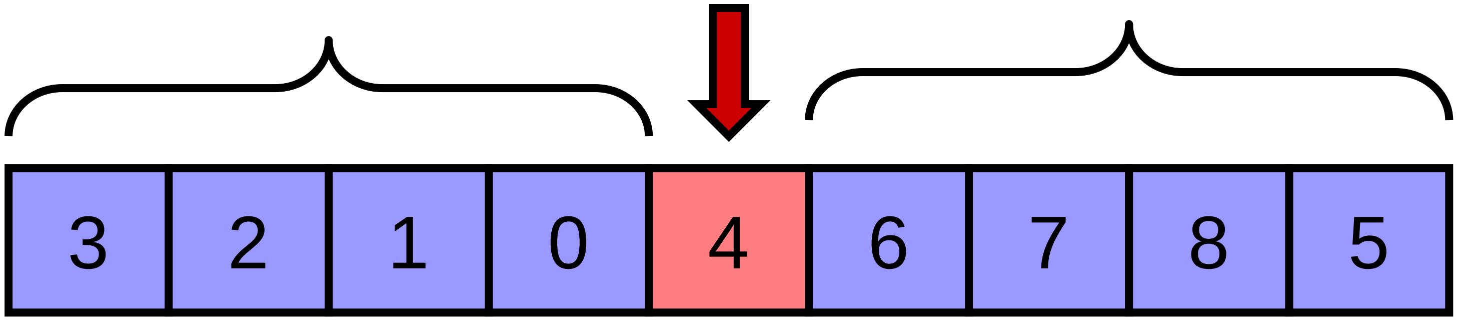
Consider Quicksort

1. Choose pivot element
2. Partition rest of sequence on that value
3. Back up and swap pivot into position



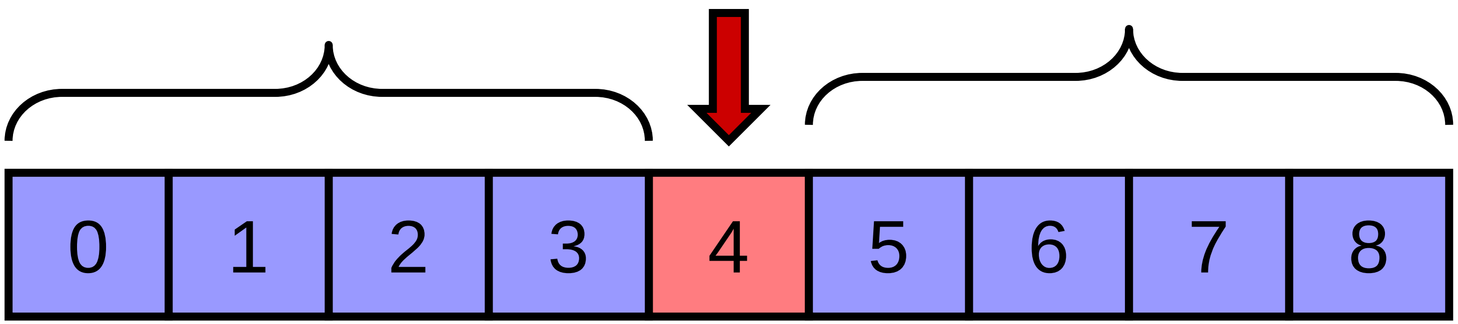
Consider Quicksort

1. Choose pivot element
2. Partition rest of sequence on that value
3. Back up and swap pivot into position
4. Quicksort both sides



Consider Quicksort

1. Choose pivot element
2. Partition rest of sequence on that value
3. Back up and swap pivot into position
4. Quicksort both sides



Copying and Comparison

- Syntax/semantics:
  - **q = p**; assert(**p == q**)
- a non-destructive find must modify and return a copy
- comparison is needed to make copy meaningful
- Allows representation of subranges (needed for divide-and-conquer)

Single-Pass / Multipass Distinction

- Single-pass (input and output iterator)
  - unidirectional tape algorithms (n-way merge)
  - Avoids needless intermediate copies/storage

- Multipass (forward iterator and above)
  - Makes in-place mutation meaningful
  - Allows divide-and-conquer

Reverse (Bidirectional) Traversal

- Syntax:
  - --p, p--
- Needed by:
  - reverse
- Used in specializations of:
  - partition
  - find_end
  - search_n
  - rotate?

Random Access

- Syntax/semantics:
  - q = **p+n**; assert(n == **p - q**); **p < q**
- Needed by:
  - random_shuffle
  - make_heap, sort_heap, etc…
- Used in specializations of:
  - binary_search et. al. ($\log_2$ N steps)
  - find, for_each, transform, et. al (loop unrolling)
  - reverse
  - find_n
  - rotate?

Pointer Model and Syntax

- Close to the machine model:
  - pointers *are* super-efficient iterators
  - iterators often fit in a register

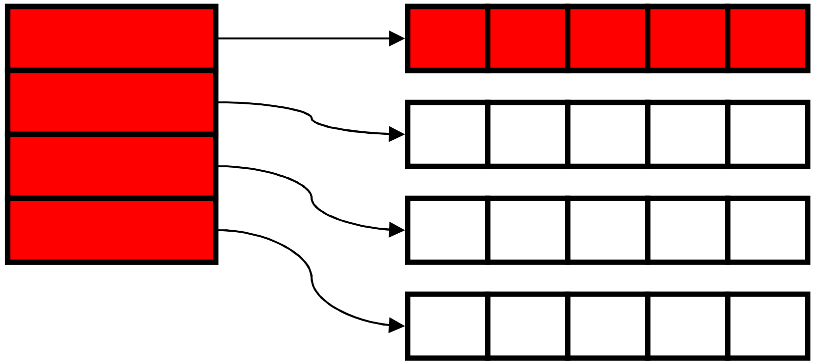
Take-Aways

- C++ iterators are not “just one way to do it.”
- Syntax helps… but it’s not about syntax!
- If we didn’t have iterators, we’d have to invent them
- Abstractions are not invented, but *discovered* through domain analysis

What about efficiency?

- The old, container-based algorithm:

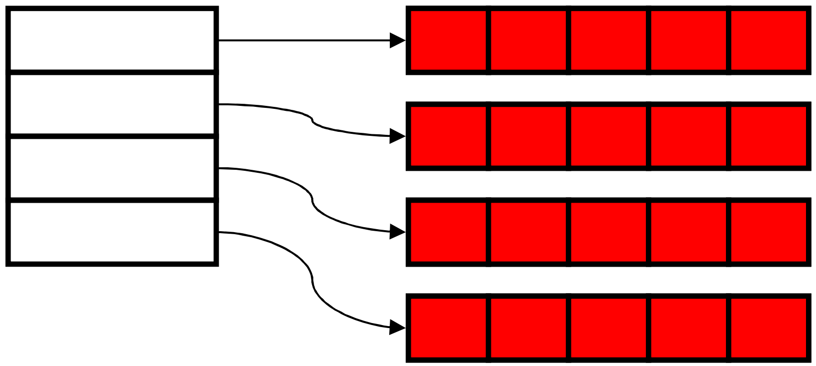
```
int* Deque
  ::Transform(F f, int* o)
{
    DequeSegment** s = segments;
    DequeSegment** s2 = s + count;
    for(; s != s2; ++s)
    {
        int* last = (*s)->last;
        int* j = (*s)->first;
        for(; j != last; ++j)
            *o++ = f(*j);
    }
    return o;
}
```



What about efficiency?

- The new, generic algorithm:

```
template< class Iter, class Fun,
                class Out >
Out transform( Iter begin,
    Iter end, Fun f, Out o )
{
    while( begin != end )
    {
        *o = f( *begin );
        ++o;
        ++begin;
    }
    return o;
}
```

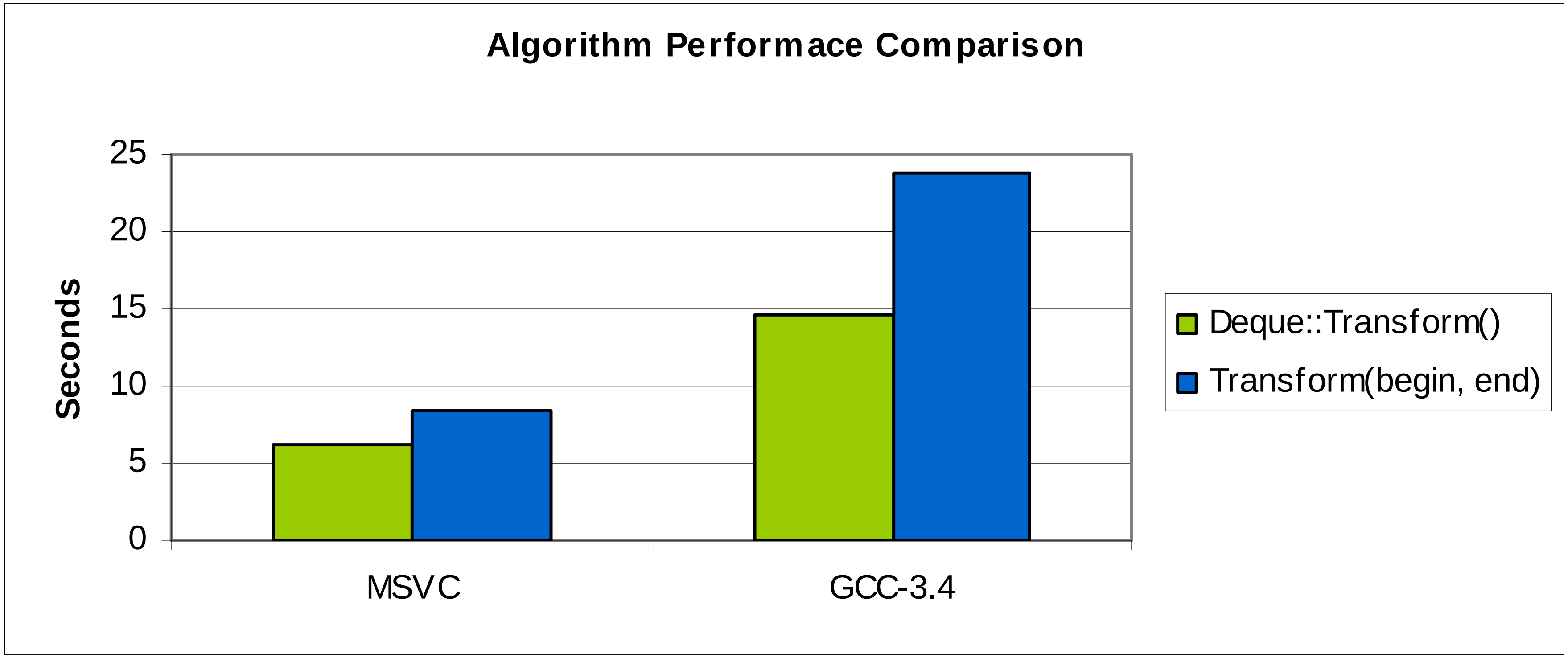


What about efficiency?

```
template< class Iter, class Fun,
              class Out >
Out transform( Iter begin,
   Iter end, Fun f, Out o )
{
   while( begin != end )
   {
      *o = f( *begin );
      ++o;
      ++begin;
   }
   return o;
}
```

```
template< class Iter, class Fun,
              class Out >
Out transform( Iter begin,
   Iter end, Fun f, Out o )
{
   while( begin != end )
   {
      *o = f( *begin );
      ++o;
      ++begin;
      if( begin == end-of-segment &&
         ! at-last-segment ) {
            move-to-next-segment
            begin = begin-of-segment
      }
   }
   return o;
}
```

The Cost(?) of Genericity



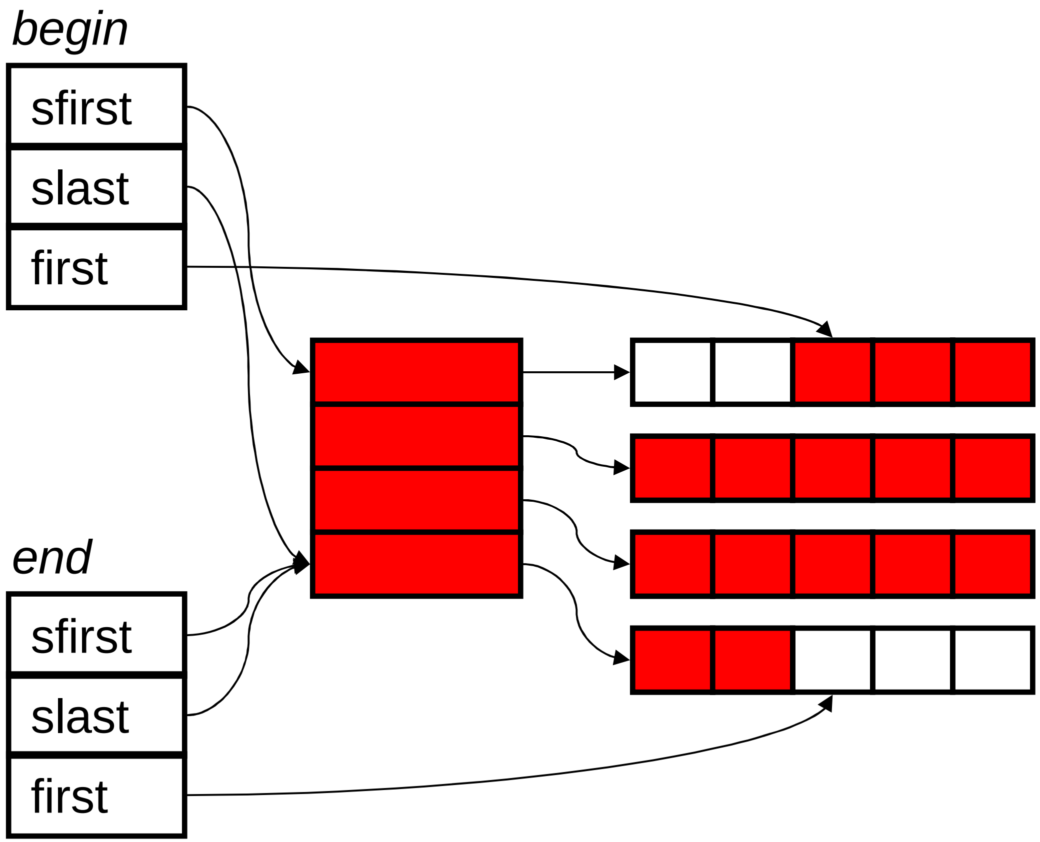
Discussion Points

- Looks like the generic algorithm is not ideal for a class of data structures.
- Where have we seen this problem before?
  - GetAt()
- What was the solution then?
  - Specialization!
- So, let's create a specialization for working with deque iterators

Deque Iterators

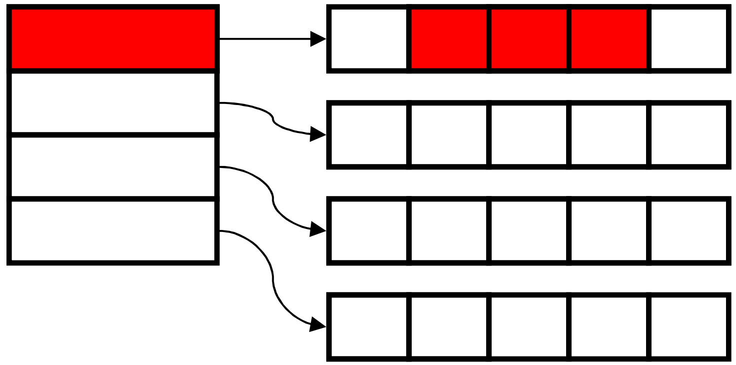
```
struct DequeIter
{
    DequeSegment** sfirst;
    DequeSegment** slast;
    int* first;
    //…
};

struct DequeSegment
{
    static const int Size;
    int* first;
    int* last;
    int data[ Size ];
};
```

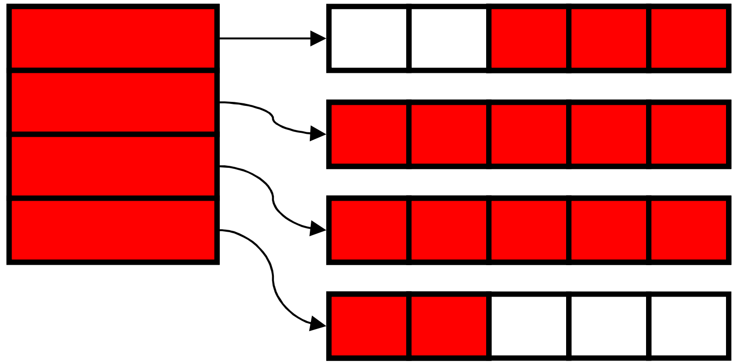
```
void Fill( DequeIter begin, DequeIter end, int value )
```



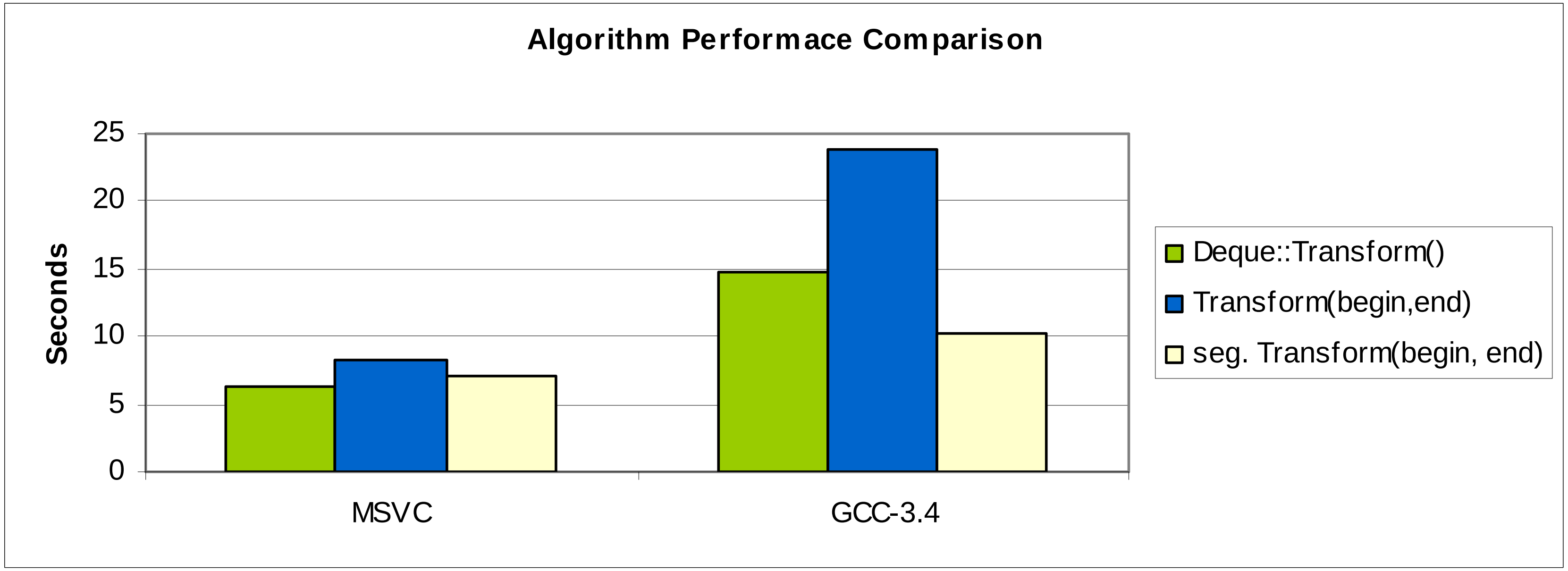
A Segmented Algorithm

```
void
Fill(DequeIter begin, DequeIter end, int value)
{
    if(begin.sfirst == end.sfirst)
        Fill(begin.first, end.first, value);
    else {
        Fill(begin.first, (*begin.sfirst)->last, value);

        // Loop over the intermediate segments
        for( ++begin.sfirst;
               begin.sfirst != end.sfirst;
               ++begin.sfirst)
            Fill((*begin.sfirst)->first,
                 (*begin.sfirst)->last,
                 value);

        // Fill the last segment.
        Fill((*begin.sfirst)->first, end.first, value);
    }
}
```



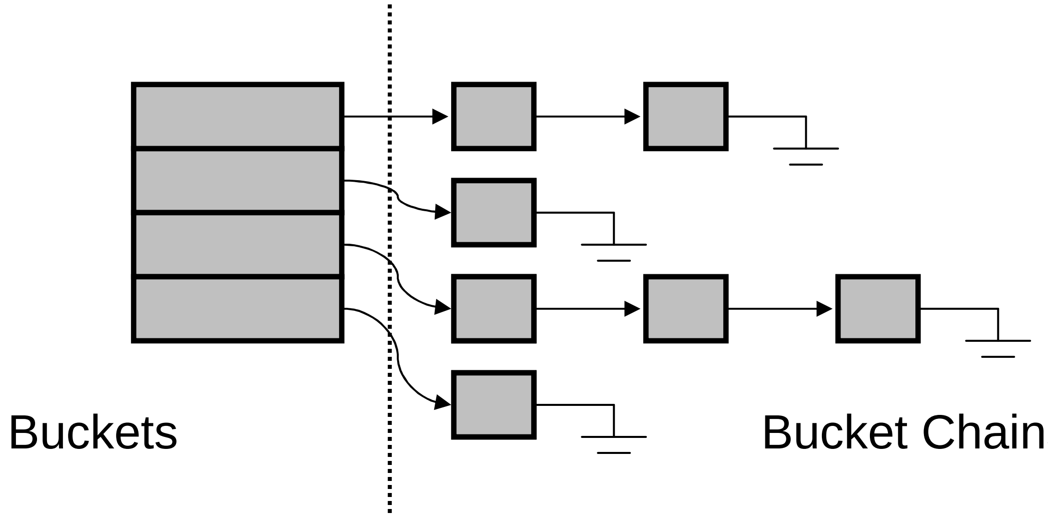


Segmented Algorithm Performance



Other Segmented Data Structures

- Hash: an array of Lists



- Both Hash and Deque would benefit from segmented algorithms

Programming Challenge #3

- Look at the segmented Transform() implementations for Hash and Deque.
- Try to lift a generalized segmented Transform() algorithm.
- Consider the requirements of your algorithm.
- Try to formulate a concept from the requirements.
- See if your concept fits in with the other concepts we've seen so far.

Exercise 3

(in progress)

https://boost-consulting.com/trac/projects/boostcon

Segmented Transform

```
// Segmented transform (fragment)
template<class Iter, class Fun, class Output>
Output TransformImpl(Iter begin, Iter end, Fun fun, Output out, True)
{
    return TransformImpl2(begin, end, fun, out, Segment(begin));
}

// Non-Segmented transform
template<class Iter, class Fun, class Output>
Output TransformImpl(Iter begin, Iter end, Fun fun, Output out, False)
{
    for(; begin != end; Next(begin))
    {
        *out = fun(Get(begin));
        ++out;
    }
    return out;
}

template<class Iter, class Fun, class Output>
Output Transform(Iter begin, Iter end, Fun fun, Output out)
{
    return TransformImpl(begin, end, fun, out, IsSegmented(begin));
}
```

Segmented Transform

```
// Segmented transform implementation
template<class Iter, class Fun, class Output, class Seg>
Output TransformImpl2(Iter begin, Iter end, Fun fun, Output out, Seg* sbegin)
{
    Seg* send = Segment(end);

    if(sbegin == send)
    {
        out = Transform(Local(begin), Local(end), fun, out);
    }
    else
    {
        out = Transform(Local(begin), End(*sbegin), fun, out);

        // Loop over all the intermediate segments
        for(++sbegin; sbegin != send; ++sbegin)
        {
            out = Transform(Begin(*sbegin), End(*sbegin), fun, out);
        }

        // Transform the last sbegin.
        out = Transform(Begin(*sbegin), Local(end), fun, out);
    }

    return out;
}
```

Iterator Requirements

(in addition to EqualityComparable)

a: object of type **U** **X: U**'s *reference type* **r:** object of type **X**

valid expression	type	complexity	preconditions	semantics/postcondition
U y(a) U y = a	U	O(1)	a nonsingular	y == a
Get(a)	X	O(1)	a not past-the-end	Get() is a *regular function*
Next(a)		O(1)	a not past-the-end	U z(a); Next(a); assert(a!=z); Next(z); assert(a==z)
Category(a)	Input Forward Random	O(1)		Tag representing which category a models
IsSegmented(a)	True *or* False	O(1)		True IFF a models *SegmentedIterator*; False otherwise

SegmentedIterator Requirements

(in addition to Iterator)

a: object of type **U**   **S: U**'s *segment type*   **L: S**'s *iterator type*

valid expression	type	complexity	preconditions	semantics
Segment(a)	S*	O(1)	a nonsingular	Pointer to the segment into which **a** points.
Local(a)	L	O(1)	a nonsingular	*Local* iterator to the element to which **a** points, or, if **a** is past-the-end, returns the past-the-end iterator for **a**'s segment.

Segment Requirements

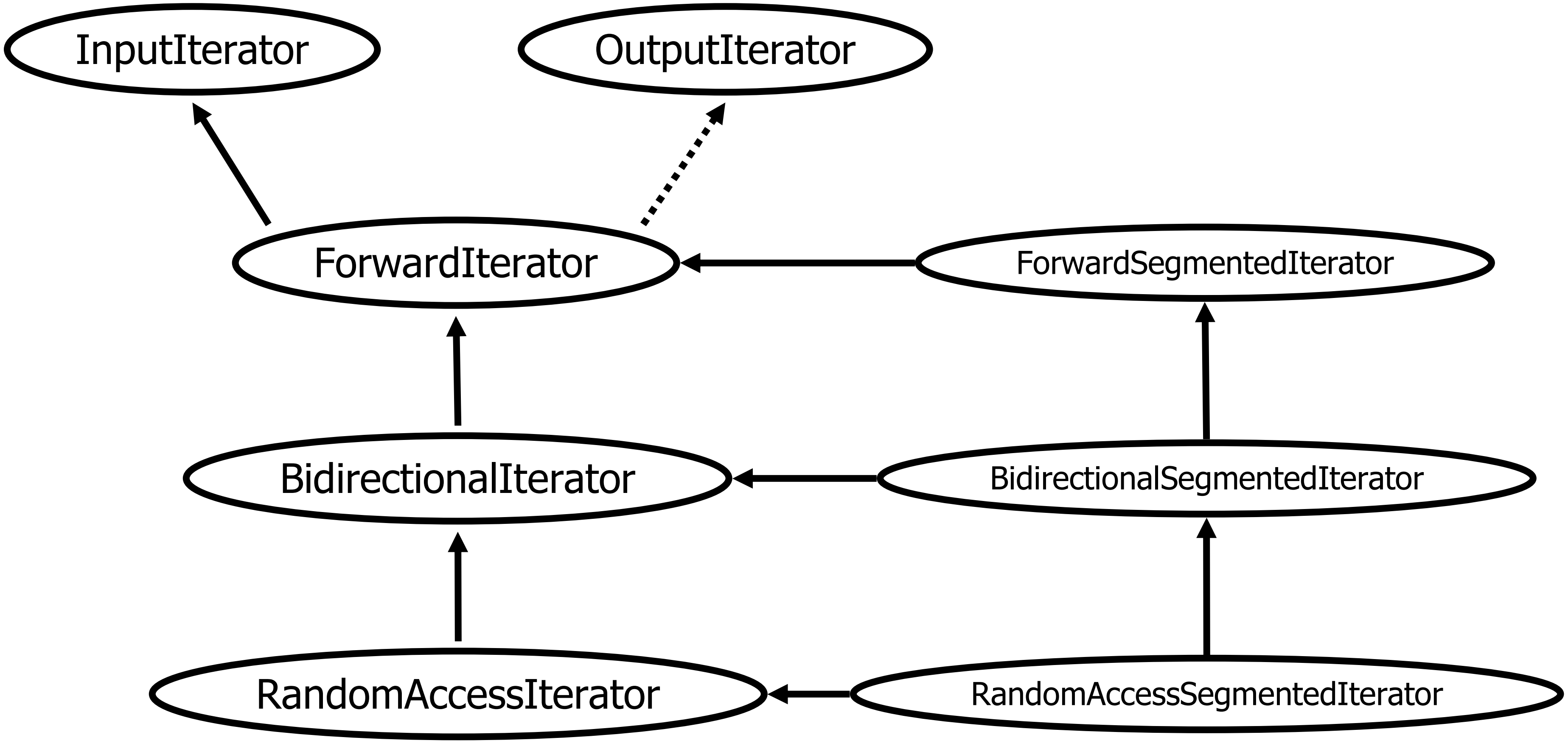
s: object of type **S** **L:** **S**'s *iterator type*

valid expression	type	complexity	preconditions	semantics/postcondition
Begin(s)	L	O(1)		*Local* iterator to the first element in s.
End(s)	L	O(1)		*Local* iterator to the last+1 element in s.

Riddle me this, Batman…

Does SegmentedIterator need to refine Iterator?

Iterator Concept Hierarchy



Concept Checking

Verifying your templates against your concepts

Spot the Bug

```
/// Requires: Type ForwardIterator is a model of
///           the ForwardIterator concept.
///
template<class ForwardIterator, class Value>
void Fill(ForwardIterator begin, ForwardIterator end, Value const &value)
{
    for(; begin < end; Next(begin))
    {
        Get(begin) = value;
    }
}

// Unit test
int main()
{
    int rg[] = {1,2,3,4};
    Fill(rg, rg + 4, 0); // Works!
    assert(!rg[0] && !rg[1] && !rg[2] && !rg[3]);
}
```

Whoops! ForwardIterators don't support less-than comparison.

Testability of Generic Code

- Concepts are part of the *documented* interface (in C++03)
  - I.e., they're not code
- Set of models of a concept is open.
  - Can't test with all of them!
- As a result, unstated requirements can (and do) slip through! ☹

Introducing Boost.ConceptCheck

- Check What?
  - All requirements of an algorithm are documented
  - Arguments meet all documented requirements
- But Why?
  - Implementation’s real requirements easily overlooked
  - Ordinary template error messages are nasty
- And How?
  - Define minimally-conforming *archetypes* of documented concepts
  - Instantiate algorithm implementations on these archetypes
  - Define concept
  - Assert that users’ types meet the same concept requirements

Concept Checking: Step 1 Define a concept

- E.g., LessThanComparable

```
// File: LessThanComparable.h
#include <boost/concept/usage.hpp>

template< class T >
struct LessThanComparable
{
    BOOST_CONCEPT_USAGE(LessThanComparable)
    {
        bool result = (x < x);
    }
 private:
    T x;
};
```

For checking that a type `T` models the `LessThanComparable` concept.

Valid expressions for models of the concept go here.

Concept Checking: Step 2
Assert a type models a concept

- Given `LessThanComparable` ...

```
#include <boost/concept/assert.hpp>
#include "./LessThanComparable.h"

// Assert at compile-time that int models LessThanComparable
BOOST_CONCEPT_ASSERT(( LessThanComparable<int> ));
```

- Use `BOOST_CONCEPT_ASSERT()`
  - ... at namespace, class or function scope
  - ... when you want to issue a compile-time diagnostic if a type fails to model a concept

Concept Checking: Step 2 cont. *or*, Add a *requires* clause

- E.g., a constrained `Min()`

```
#include <boost/concept/requires.hpp>
#include "./LessThanComparable.h"

template< class T >
BOOST_CONCEPT_REQUIRES(
    ((LessThanComparable<T>)),
(const T &)) Min(const T & t, const T & u)
{
    return (t < u) ? t : u;
}
```

- Use `BOOST_CONCEPT_REQUIRES()`
  - ... to declare the return type of constrained function templates
  - ... to state requirements on template params

BOOST_CONCEPT_REQUIRES()

```
BOOST_CONCEPT_REQUIRES( models, result )
```

A preprocessor sequence of models clauses

The function template's return type

```
/// @brief Sass (tr. v.): know, be aware of, meet, have sex with.
template< class Person >
BOOST_CONCEPT_REQUIRES(
    (( Hoopy< Person > ))
    (( Frood< Person > )),

(char const *)) Sass(const Person & person)
{
    return "One hoopy frood who really knows where his towel is";
}
```

Improved error messages

```
#include <boost/concept/requires.hpp>
#include "./LessThanComparable.h" // Definition of LessThanComparable

template< class T >
BOOST_CONCEPT_REQUIRES(
    ((LessThanComparable<T>)),
(const T &)) Min(const T & t, const T & u)
{
    return (t < u) ? t : u;
}

struct S {};

int main()
{
    S a, b;
    S c = Min(a, b);
}
```

```
LessThanComparable.h(9) : error C2676:
binary '<' : 'S' does not define this
operator or a conversion to a type
acceptable to the predefined operator

LessThanComparable.h(8) : while compiling
class template member function
'LessThanComparable<T>::
~LessThanComparable(void)'
   with
   [
       T=S
   ]
```

A more advanced concept

```
template <class X>
struct InputIterator
  : Assignable<X>, EqualityComparable<X>
{
 private:
    typedef std::iterator_traits<X> t;
 public:
    typedef typename t::value_type value_type;
    typedef typename t::difference_type difference_type;
    typedef typename t::reference reference;
    typedef typename t::pointer pointer;
    typedef typename t::iterator_category iterator_category;

    BOOST_CONCEPT_ASSERT((SignedInteger<difference_type>));
    BOOST_CONCEPT_ASSERT((Convertible<iterator_category, std::input_iterator_tag>));
    BOOST_CONCEPT_ASSERT((Convertible<reference, value_type>));

    BOOST_CONCEPT_USAGE(InputIterator)
    {
        X j(i);          // require copy construction
        is_value(*i++);   // require postincrement-dereference returning value_type
        X& x = ++j;      // require preincrement returning X&
    }
 private:
    X i;
             void is_value(const value_type&);
};
```

Use inheritance for concept refinement

Associated types are nested typedefs.

Assert properties of associated types.

Valid expressions

Concept Checking: Step 3
Define an *archetype*

- Archetype:
  - A *minimally* conforming model of a concept

```
// Is this minimal?
struct LessThanComparableArchetype
{
    bool operator< (const LessThanComparableArchetype &) const
    { return false; }
};
```

- ☒ It's copy-constructible
- ☒ It's assignable
- ☒ The return type is bool, instead of just convertible to bool

dwa1

Concept Checking: Step 3 cont. Define an *archetype*

```
#include <boost/concept_archetype.hpp>

template<
  class Base = boost::null_archetype<>
>
struct LessThanComparableArchetype
  : Base
{
    boost::boolean_archetype
    operator< (const LessThanComparableArchetype &) const;
};

BOOST_CONCEPT_ASSERT((
    LessThanComparable< LessThanComparableArchetype<> >
));
```

Predefined archetypes

Type that models no concepts

Parameterization and inheritance allows chaining

boolean_archetype models convertible-to-bool

- Or, just use `boost::less_than_comparable_archetype<>`

Slide 120

dwa1 This one still isn’t truly minimal, because you can pass anything convertible to LessThanComparableArchetype on the RHS.

I think minimal requires something like a non-member, templated operator<(T const&,T const&) with an enable_if that asserts that T is derived from LessThanComparable<U> for some U. It's nasty.
David Abrahams, 5/1/2008

Concept Checking: Step 4
Validate algorithms with archetypes

```
/// Requires: T models LessThanComparable
template< class T >
BOOST_CONCEPT_REQUIRES(
  (( LessThanComparable<T> )),

(T)) Min( T t, T u )
{
    return (t < u) ? t : u;
}



inline void check_Min()
{
    typedef LessThanComparableArchetype<> T;
    const T & x = *boost::optional<T>();
    const T & y = Min(x, x);
}
```

LessThanComparable.h(26) : error C2248: 'boost::null_archetype<>::null_archetype' : cannot access private member declared in class 'boost::null_archetype<>'

boost\concept_archetype.hpp(38) : see declaration of 'boost::null_archetype<>::null_archetype'

This diagnostic occurred in the compiler generated function 'LessThanComparableArchetype<>:: LessThanComparableArchetype(const LessThanComparableArchetype<> &)'

Concept Checking: Step 4
Validate algorithms with archetypes

```
/// Requires: T models LessThanComparable
template< class T >
BOOST_CONCEPT_REQUIRES(
  (( LessThanComparable<T> ))

(const T &)) Min( const T & t, const T & u )
{
    return (t < u) ? t : u;
}



inline void check_Min()
{
    typedef LessThanComparableArchetype<> T;
    const T & x = *boost::optional<T>();
    const T & y = Min(x, x);
}
```

OK!

A more advanced archetype

```
template <class T, class Base = boost::null_archetype<> >
class input_iterator_archetype
  : boost::copy_constructible_archetype< Base >
{
private:
    typedef input_iterator_archetype self;
public:
    typedef std::input_iterator_tag iterator_category;
    typedef T value_type;
    struct reference
      : boost::copy_constructable_archetype<
            boost::convertible_to_archetype<value_type> >
    {};
    typedef const T* pointer;
    typedef std::ptrdiff_t difference_type;
    self & operator=(const self &);
    boost::boolean_archetype operator==(const self &) const;
    boost::boolean_archetype operator!=(const self &) const;
    reference operator*() const;
    self & operator++();
    self operator++(int);
};
```

Note archetype chaining

Final Exercise!

- Use the Boost Concept Check Library to check the coverage of your algorithm from Exercise 2
- Go to http://TODO and download the code for your algorithm.
- Use the concepts we've defined to add requires clauses to your algorithm
- Use the archetypes we've defined to check the coverage of your concept checks
- Hint: you may need to change your algorithm!

For More Information …

- See Matt Austern's paper about segmented algorithms:
  - http://lafstern.org/matt/segmented.pdf
- See Alex Stepanov's papers on generic programming:
  - http://www.stepanovpapers.com/eop/lecture_all.pdf