Range and Elevation

C++ in a modern world

Steve Love

@IAmSteveLove #accu2014

ACCU April 2014

In the beginning...

The problem with pointers

Example

```
int fn( item_type * invoice )
```

`invoice` might be...

1. a collection
2. an out parameter
3. an optimisation
4. a polymorphic object
5. a function!

Pointer to iterator

Certainty	an iterator is ***always*** a pointer to an element in a collection
Safety	you cannot do arithmetic operations with iterators

Iterators raise the abstraction of pointers (a bit)

C++ iterator

Iterator *traverse the items of a container without exposing any details of how elements are stored.*

Usually travel in pairs and must be kept in sync

Are a leaky abstraction know the iterator concept or rely on the lowest common denominator

The slightly contrived example

```
auto p = partition(begin(s), end(s),
  [](double v) {return v > 0.0;});

transform(begin(s), p, back_inserter(o),
  [](double v) {return v * 0.175; });
```

Or for the 1 liner junkies

```
transform(begin(s), partition(begin(s), end(s),
    [](double v) {return v > 0.0;}),
  back_inserter(o),
  [](double v) {return v * 0.175;});
```

- `partition` modifies its input
- `transform` requires an output range to store results

In range

Not much has changed in more than 15 years:

- C++98 and the algebra of iterators
- Containers are iterator factories
- STL is about algorithms

High Church C++

Allan Kelly from 2000
www.allankelly.net/writing/webonly/highchurch.html

It’s all rather manual

"It is unworthy of excellent men to lose hours like slaves in the labour of calculation which could safely be relegated to anyone else if machines were used."

Gottfried Wilhelm von Leibniz, 1685

We’re still noodling around with begin and end!

The Range advantage

Using Andrei Alexandrescu’s range concept, it’s - in theory - simple(r):

```
auto r = map(filter(s, [](double v) {return v > 0.0;}),
  [](double v) {return v * 0.175;});
```

http://www.slideshare.net/rawwell/iteratorsmustgo, 2009

What was missing

auto	range operations return different range types
lambda	function objects work but lose the locality of reference

Compose thyself

But....functional composition has its limits

```
auto r = map(filter(filter(s,
  [](double v) {return v < 3.0;}),
  [](double v) {return v > 0.0;}),
  [](double v) {return v * 0.175;});
```

Which lambda is which here?

Is the order of operations correct?

Are you *absolutely* sure?

Programming style

Iterator style

Some abstraction improvements over pointers
It’s very verbose

Range style

Definite abstraction improvement on iterators
Composition can lead to an explosion of nesting

Form and function

It’s not that the Range style is *wrong*

It *is* a low-level way of looking at it.

The API *over the top* is what makes a difference.

What else is there?

Python

```
r = (v * 0.175 for v in s if v > 0.0)
```

C#

```
    var r = from v in s
            where v > 0.0
            select v * 0.175;
```

No wonder people think C++ is complicated...

...so can C++ do any better?

Closer to home

C# Fluent-style

```
var r = s.Where(v => v > 0.0)
         .Select(v => v * 0.175);
```

This is enabled by:

- `IEnumerable<T>`
- Extension methods
- Lambda functions

What do we want it to do?

`IEnumerable<T>` represents a range

`IEnumerator<T>` has an Iterator interface

Iterating

1. Get current
2. Move next
3. (is valid)

It’s probably not a bad place to start

The C++ Iterator interface

```
template<typename iterator_type>
class iterable_range
{
  public:
    iterable_range(iterator_type begin, iterator_type end);

    auto operator*() const -> decltype(*this->pos);
    explicit operator bool() const;
    auto operator++() -> iterable_range &;
    auto operator++(int) -> iterable_range;

  private:
    iterator_type pos, end_pos;
};
```

This is pretty close to Andrei’s Range interface - and indeed, the C# `IEnumerator<T>` interface.

So can it be done?

```
auto r = from(s)
         .where([](double v) {return v > 0.0;})
         .select([](double v) {return v * 0.175;});
```

Questions

1. What does `from` return?
    1. `select`? `where`?
2. What is the final return type?
3. Can it use “deferred execution”?
4. Is it even a good idea?

A blind alley

Chaining operations together with '.' requires common base class/interface

```
class range
{
  public:
    template<typename filter>
    range where(filter fn) { }

    template<typename transform>
    range select(transform fn) { }
    // ...
```

A small blink of light

...or extension methods

```
auto r = from(s)<-where( ... )<-select( ... );
```

...tempting!

Yay! Templates!

We can now infer `from`

```
template<typename container_type>
auto from(const container_type & ctr) ->
  iterable_range<typename container_type::const_iterator>
{
  return iterable_range<typename container_type::const_iterator>
    { std::begin(ctr), std::end(ctr) };
}
```

It’ll get worse before it gets better. I promise

Using is everything

- Range operations are backed by an implementing range type
- Each range operation operates on some other range type
- Every range type conforms to the basic Iterator interface
- Each range operation returns a range of its own type

Where’s the simple example?

```
template<typename range_t, typename unary_predicate>
class where_range
{
  private:
    mutable range_t r;
    unary_predicate fn;
    void find_next() const
    {
      while(r && !fn(*r)) ++r;
    }

  public:
    auto operator*() const -> decltype(*r);
    explicit operator bool() const;
    auto operator++() -> where_range &;
    auto operator++(int) -> where_range;
};
```

Not much of a leap

Transformations (or `select`) are even simpler

```
template<typename range_t, typename transformation>
class select_range
{
  private:
    mutable range_t r;
    transformation fn;

  public:
    auto operator*() const -> decltype(fn(*r))
    {
      return fn(*r);
    }
    explicit operator bool() const;
    auto operator++() -> select_range &;
    auto operator++(int) -> select_range;
};
```

What do we want?

```
template<typename range_t, typename filter>
auto where(range_t r, filter fn) -> where_range<range_t, filter>
{
  return where_range<range_t, filter>{r, fn};
}

template<typename range_t, typename xform>
auto select(range_t r, xform fn) -> select_range<range_t, xform>
{
  return select_range<range_t, xform>{r, fn};
}
```

So far, so...what?

That’s fine for a “composing” interface...but we already discounted that

```
auto r = select(
  where(from(s),[](double v) {return v > 0.0;}),
    [](double v) {return v * 0.175;});
```

So far, so...what?

That’s fine for a “composing” interface...but we already discounted that

```
auto r = select(
  where(from(s),[](double v) {return v > 0.0;}),
    [](double v) {return v * 0.175;});
```

Is this better?

```
auto r = from(s)
       | where([](double v) { return v > 0.0; })
       | select([](double v) { return v * 0.175;});
```

But how?

So - `where` is a simple function.

But `where_range` requires a range on which to operate.

```
auto r = from(s)
       | where([](double v) { return v > 0.0; })
```

Only **`operator|`** knows about the range types on both sides of the expression

but not the predicate

Constraints

The `where` function knows what *type* of range, but doesn’t have enough info to make one

Time for...yep: more templates!

Manufacturing magic

```
template<template<class,class> class range_t, typename expr>
class range_factory
{
  public:
    range_factory(expr action) : action{ action } { }

    template<typename range_of>
    auto operator()(range_of r) const -> range_t<range_of, expr>
    {
      return range_t<range_of, expr>{ r, action };
    }

  private:
    expr action;
};
```

New-where

```
template<typename filter>
auto where(filter fn) -> range_factory<where_range, filter>
{
  return range_factory<where_range, filter>{ fn };
}
```

```
template<typename range_t, typename factory_t>
auto operator|(range_t r, factory_t f) -> decltype(f(r))
{
  return f(r);
}
```

Templates are very powerful…and as promised, it’s already got worse!….

Before we go on

It doesn’t take long to get to this:

```
template<template<typename...> class range_type,
  typename next_type,
  typename... others>
class range_N_factory<range_type, next_type, others...>
  : public range_N_factory<range_type, others...>
{
```

Nasty, eh?

Does it matter?

Before we go on

It doesn’t take long to get to this:

```
template<template<typename...> class range_type,
  typename next_type,
  typename... others>
class range_N_factory<range_type, next_type, others...>
  : public range_N_factory<range_type, others...>
{
```

Nasty, eh?

Does it matter?

Only if you ever see it!

The lengths we go to

Fact

You can hide as much template nastiness as you like behind a simple API

Corollary

You should hide as much as you can!

Interoperability

```
auto r = from( ... )
       | where( ... )
       | select( ... );
```

Support for range-based loops:

```
for(auto i : r)
{
  // interesting things with i
}
```

Creating a vector is trivial:

```
auto v = std::vector(begin(r), end(r));
```

But that wasn’t enough

```
auto v = r | to< std::vector >();
```

The range...relegated

The range

It's there, but never actually seen
- a mere implementation detail!

- ~~Iterator style~~
- ~~Range style~~
- Functional style

It’s a start

No `yield` equivalent, no extension methods

would make extending and customising *much* easier

Why do we need another range library?

We don't.

We need a better API for using them.

1. Simpler
2. More expressive
3. Higher abstraction

Which is why **narl** is Not Another Range Library

https://github.com/essennell/narl

A selection of examples

Here are a few parallels between the iterator style and the narl style

...to the tune of The Noveltones - Left Bank Two

all_of

```
TEST_CASE( "all_of", "[narl][stl][all_of]" )
{
  std::vector< int > data { 2, 4, 6 };

  auto stl_result = std::all_of( std::begin( data ), std::end(
      data ),
    []( int i ) { return i % 2 == 0; } );

  auto narl_result = from( data )
                   | all( []( int i ) { return i % 2 == 0; } );

  REQUIRE( narl_result == stl_result );
}
```

any_of

```
TEST_CASE( "any_of", "[narl][stl][any_of]" )
{
  std::vector< int > data { 2, 4, 6 };

  auto stl_result = std::any_of( std::begin( data ), std::end(
      data ),
    []( int i ) { return i == 4; } );

  auto narl_result = from( data )
                   | any( []( int i ) { return i == 4; } );

  REQUIRE( narl_result == stl_result );
}
```

none_of

```
TEST_CASE( "none_of", "[narl][stl][none_of]" )
{
  std::vector< int > data { 2, 4, 6 };

  auto stl_result = std::none_of( std::begin( data ), std::end(
      data ),
    []( int i ) { return i == 3; } );

  auto narl_result = !( from( data )
                      | any( []( int i ) { return i == 3; } ) );

  REQUIRE( narl_result == stl_result );
}
```

count_if

```
TEST_CASE( "count_if", "[narl][stl][count_if]" )
{
  std::vector< int > data { 2, 4, 6 };

  auto stl_result = std::count_if( std::begin( data ), std::end(
       data ),
    []( int i ) { return i > 5; } );

  auto narl_result = from( data )
    | where( []( int i ) { return i > 5; } )
    | count();

  REQUIRE( narl_result == stl_result );
}
```

find_if

```
TEST_CASE( "find_if", "[narl][stl][find_if]" )
{
  std::vector< int > data { 1, 2, 3 };

  auto stl_result = std::find_if( std::begin( data ), std::end(
      data ),
    []( int i ) { return i == 2; } );

  auto narl_result = from( data )
                   | skip_while( []( int i ) { return i != 2; }
                       );

  REQUIRE( *narl_result == *stl_result );
}
```

copy_n

```
TEST_CASE( "copy_n", "[narl][stl][copy_n]" )
{
  std::vector< int > data { 1, 2, 3, 4, 5, 6 };
  std::vector< int > stl_result;

  std::copy_n( std::begin( data ), 3, std::back_inserter(
      stl_result ) );

  auto narl_result = from( data )
    | take( 3 )
    | to< std::vector >();

  REQUIRE( std::equal( std::begin( stl_result ), std::end(
      stl_result ),
    std::begin( narl_result ) ) );
}
```

transform

```
TEST_CASE( "transform", "[narl][stl][transform]" )
{
  std::vector< int > data { 1, 2, 3 };
  std::vector< int > stl_result;

  std::transform( std::begin( data ), std::end( data ),
    std::back_inserter( stl_result ), []( int i ) { return i *
        10; } );

  auto narl_result = from( data )
    | select( []( int i ) { return i * 10; } )
    | to< std::vector >();

  REQUIRE( std::equal( std::begin( stl_result ), std::end(
      stl_result ),
    std::begin( narl_result ) ) );
}
```

generate_n

```
TEST_CASE( "generate_n", "[narl][stl][generate_n]" )
{
  std::vector< int > stl_result;

  std::generate_n( std::back_inserter( stl_result ), 3, []{
      return 5; } );

  auto narl_result = fill_range< int >( 5 )
                   | take( 3 )
                   | to< std::vector >();

  REQUIRE( std::equal( std::begin( stl_result ), std::end(
      stl_result ),
    std::begin( narl_result ) ) );
}
```

remove_copy_if

```
TEST_CASE( "remove_copy_if", "[narl][stl][remove_copy_if]" )
{
  std::vector< int > data { 1, 2, 3, 4 };
  std::vector< int > stl_result;

  std::remove_copy_if( std::begin( data ), std::end( data ),
    std::back_inserter( stl_result ),
    []( int i ) { return i % 2 == 0; } );

  auto narl_result = from( data )
                   | where( []( int i ) { return i % 2 != 0; } )
                   | to< std::vector >();

  REQUIRE( std::equal( std::begin( stl_result ), std::end(
      stl_result ),
    std::begin( narl_result ) ) );
}
```

sort

```
TEST_CASE( "sort comparer", "[narl][stl][sort][comparer]" )
{
  std::vector< int > data { 3, 2, 4, 1, 5 };
  std::vector< int > stl_result { data };

  std::sort( std::begin( stl_result ), std::end( stl_result ),
    std::less< int >() );

  auto narl_result = from( data )
                   | sorted( std::less< int >() )
                   | to< std::vector >();

  REQUIRE( std::equal( std::begin( stl_result ), std::end(
      stl_result ),
    std::begin( narl_result ) ) );
}
```

Is it worth the candle?

It’s quicker for some things, slower for others...

My measurements weren’t very scientific

I’ve not investigated the generated code...

I’m sure it could be improved and optimised

Frankly

Me and narl

"*In symbols one observes an advantage in discovery which is greatest when they express the exact nature of a thing briefly and, as it were, picture it; then indeed the labor of thought is wonderfully diminished.*"
Gottfried Wilhelm von Leibniz 1646-1716

https://github.com/essennell/narl

steve@arventech.com   cvu@accu.org   @IAmSteveLove