Iterators and Ranges for numerical problems

Karsten Ahnert

Ambrosys GmbH, Potsdam

November 19, 2014

Outline

Introduction

Iterators

Unique way to traverse containers
Unique way to apply iterative IO
Unique way of expressing algorithms

Example – basic use

```
for( auto iter = values.begin() ;
     iter != values.end() ;
     ++iter )
{
    cout << *iter << endl;
}
```

Example – basic use

```cpp
for( auto iter = values.begin() ;
     iter != values.end() ;
     ++iter )
{
    cout << *iter << endl;
}
```

C++11 - use range based for

```cpp
for( auto v : values )
{
    cout << v << endl;
}
```

Example – Container traversal

```
list< double > values;
list< double > values2( values.size() );
```

Can be used in

```
transform( values.begin() , values.end() ,
           values2.begin() ,
           []( double x ) {
               return x * 2.0; } );
```

Example – Container traversal

```
vector< double > values;
vector< double > values2( values.size() );
```

Can be used in

```
transform( values.begin() , values.end() ,
           values2.begin() ,
           []( double x ) {
               return x * 2.0; } );
```

Examples – IO

Input

```
vector< double > values;
copy_if( istream_iterator< double >( cout ) ,
         istream_iterator< double >() ,
         back_inserter( values ) ,
         []( double x ) { return x > 0.0; } );
```
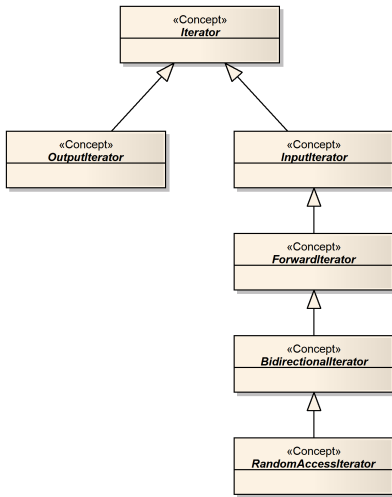
Output

```
vector< double > values;
// fill values
copy_if( values.begin() , values.end() ,
         ostream_iterator< double >( std::cout , "\n" ) ,
         []( double x ) { return x > 0.0; } );
```
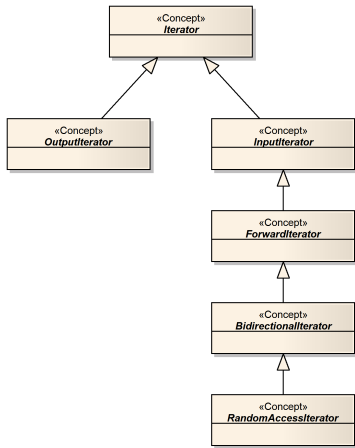
Examples – Combine algorithms

Find a nice real life example.

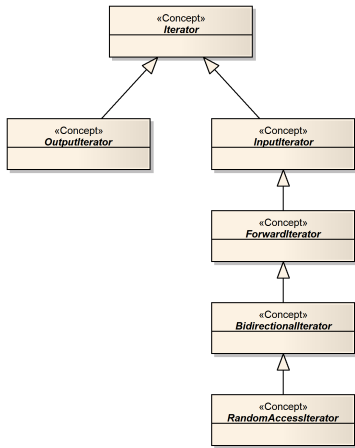# Iterator types – Concepts

Iterator types – Concepts

OutputIterator

```
*i = o;
*i++ = o;
i++;
++i;
```

Are special, `back_inserter`, `ostream_iterator`, ...
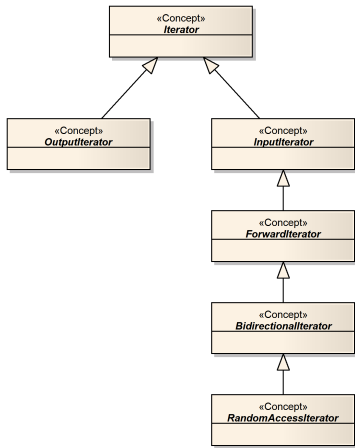
Iterator types – Concepts

InputIterator a.k.a. Single-Pass
Iterator

```
bool r = i != j;
val x = *i;
iterator j = ++i;
i++;
val x = *i++;
```

`istream_iterator`,
`istreambuf_iterator`

But, if `i == j` then `++i != ++j`

Iterator types – Concepts

«Concept»
Iterator

«Concept»
OutputIterator

«Concept»
InputIterator

«Concept»
ForwardIterator

«Concept»
BidirectionalIterator

«Concept»
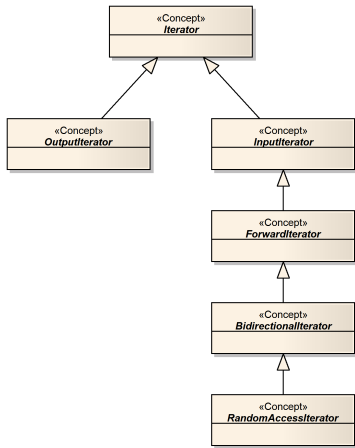***RandomAccessIterator***

ForwardIterator

```
iterator j = i++;
```

But, if `i == j` then `++i == ++j`

Iterator types – Concepts

BidirectionalIterator

```
iterator j = --i;
iterator j = i--;
val x = *i--;
```

```
map< K , V >::iterator,
list< T >::iterator
```

Iterator types – Concepts

«Concept»
Iterator

«Concept»
OutputIterator

«Concept»
InputIterator

«Concept»
ForwardIterator

«Concept»
BidirectionalIterator

«Concept»
RandomAccessIterator
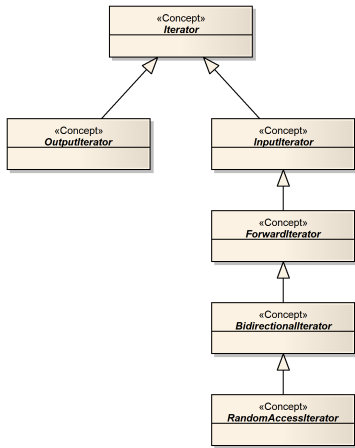
RandomAccessIterator

```
i += n;
i -= n;
val x = i[n];
long dist = i - j;
bool b = i < j;
```

vector< T >::iterator

Algorithms

I all_of	F remove	is_partitioned	is_heap
I any_of	F remove_if	partition	is_heap_until
I none_of	I,O remove_copy	partition_copy	make_heap
I for_each	I,O remove_copy_if	stable_partition	push_heap
I count	F replace	partition_point	pop_heap
I count_if	F replace_if	is_sorted	sort_heap
I mismatch	I,O replace_copy	is_sorted_until	max
I equal	I,O replace_copy_if	sort	max_element
I find	F swap_ranges	partial_sort	min
I find_if	F iter_swap	partial_sort_copy	min_element
I find_if_not	B reverse	stable_sort	minmax
F find_end	B,O reverse_copy	nth_element	minmax_element
I,F find_first_if	F rotate	lower_bound	lexicographical_compare
F adjacent_find	F,O rotate_copy	upper_bound	is_permutation
F search	R random_shuffle	binary_search	next_permutation
F search_n	R shuffle	equal_range	prev_permutation
I,O copy	F unique	merge	iota
I,O copy_if	I,O unique_copy	inplace_merge	accumulate
I,O copy_n		includes	inner_product
B,O copy_backward		set_difference	adjacent_difference
I,O move		set_intersection	partial_sum
B,O move_backward		set_symmetric_difference	
F fill		set_union	
F fill_n			
I,O transform			
F generate			
I generate_n			

Ranges

Simplifying iterators
Generalization of iterators
First defined in Boost
Soon in the standard library?

```
vector< double > values;
boost::for_each( values , []( double x ) { cout
    << x << endl; } );
```

Ranges – more examples from boost

Filters
complicated algorithms

Ranges in Boost

Ranges are pairs of iterators.
Memory overhead
Filters grow exponential in size

Ranges for the native C++

Introduces new concepts:
Iterable, Container, Sentinel
The range is the main abstraction not the iterator
It holds all informations
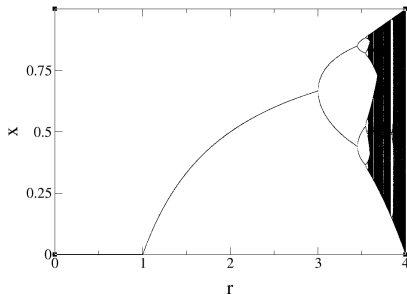Concepts, asymmetric algorithms, sentinels have their own type.

Iterators and ranges for dynamical systems

Dynamical systems – Maps

$$x_{n+1} = f(x_n)$$

Example: Logistic map
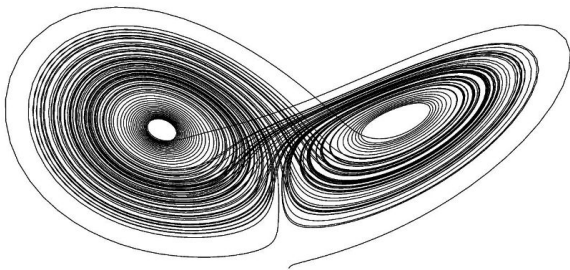
$$x_{n+1} = r\,x_n(1 - x_n)$$

Dynamical systems – ODEs

$$\frac{\mathrm{d}x}{\mathrm{d}t} = f(x, t)$$

Example: Lorenz attractor

$$\dot{x} = \sigma(y - x) \quad , \quad \dot{y} = x(\rho - z) - y \quad , \quad \dot{z} = xy - \beta z$$

Numerical solution:

$$x(t + \Delta t) = F(x(t))$$

Newton method

Find the root

$$0 = f(x)$$

Newtons method

- Choose x_0
- Iterate $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$

Map range

Abstraction for $x_{n+1} = f(x_n)$

Two versions:

1. `map_range` - stop predicate
2. `counted_map_range` - iterates n-times

Models the SinglePassRange concept

Map range - applications

- Generalized iota
- Functional random number generators
- Ordinary differential equations
- Maps (dynamical maps)

Map range – implementation

```cpp
template< typename T1 , ... > class map_range
{
    struct iterator { ... };
 public:
    // ...
    iterator begin() { return iterator( this ); }
    iterator end() { return iterator( nullptr ); }
    // ...
};
```

Range algorithms redirect to iterator algorithms:

```cpp
template< typename R , typename F >
void for_each( R const& r , F f ) {
    std::for_each( r.begin() , e.end() , f );
}
```

Map range

```
template< typename T , typename F , typename C >
class map_range
{
    struct iterator { ... };

  public:
    map_range( T value , F func , C condition )
    : m_value { std::move( value ) }
    , m_func { std::move( func ) }
    , m_condition( condition )
    {}

    iterator begin() const { return iterator( this ); }
    iterator end() const { return iterator( nullptr ); }

private:
    mutable T m_value;
    mutable F m_func;
    C m_condition;
};
```

Map range

```cpp
struct iterator {
    iterator( map_range const* _r ) : r( _r ) {}

    iterator& operator++() {
        r->m_value = r->m_func( r->m_value );
        if( r->m_condition( r->m_value ) ) {
            r = nullptr;
        }
        return *this;
    }

    T& operator*() const {
        return r->m_value; }
    bool operator==( iterator const& o ) const {
        return ( r == o.r ); }
    bool operator!=( iterator const& o ) const {
        return ! ( *this == o );
    }

    map_range const* r;
};
```

Counted map range

```cpp
template< typename T , typename F >
class counted_map_range
{
    struct iterator { ... };

public:
    counted_map_range( T value , F func , size_t
        max_iterations )
    : m_current_iteration { 0 }
    , m_max_iterations { max_iterations }
    , m_value { std::move( value ) }
    , m_func { std::move( func ) }
    {}

    iterator begin() const { return iterator( this ); }
    iterator end( void ) { return iterator( nullptr ); }

private:
    mutable size_t m_current_iteration = 0;
    const size_t m_max_iterations;
    mutable T m_value;
    mutable F m_func;
};
```

Factory functions

```cpp
template< typename T , typename F , typename C >
auto make_map_range( T t , F f , C condition )
{
    return map_range< T , F , C >(
        std::move( t ) ,
        std::move( f ) ,
        std::move( condition ) );
}
```

```cpp
template< typename T , typename F >
auto make_counted_map_range( T t , F f , size_t
    max_iterations )
{
    return counted_map_range< T , F >(
        std::move( t ) ,
        std::move( f ) ,
        max_iterations );
}
```

Examples – generalized iota

Generalized Iota:

```
size_t n = 10;
auto iota = make_counted_map_range( 1 , []( auto x ) {
    return x * 2; } , 10 );

std::vector< int > values;
boost::copy( iota_range , std::back_inserter( values ) );
for( auto i : values ) { cout << i << endl; }
```

Examples – generalized iota

Problem: We can not easily generate a square iota:
$1, 4, 9, 16, 25, 36, ...$

Examples – generalized iota

Problem: We can not easily generate a square iota:
$1, 4, 9, 16, 25, 36, ...$
Introduce a projected map range.

```
auto iota_range = make_projected_counted_map_range(
    1
    , []( auto x ) { return x+1 ; }
    , 11
    , []( auto x ) { return x*x; }
    );
for( auto i : iota_range ) { std::cout << i << std::endl; }
```

Example - logistic map

```
double r = 3.2;
auto l = [r]( auto x ) {return r * x * ( 1.0 - x ); };
auto range = make_counted_map_range( 0.5 , l , 1000 );
for( auto x : range ) { cout << x << endl; }
```

Example – Newton method

```
auto newton_range(
    auto x , auto f , auto df ,
    auto break_condition )
{
    return make_map_range(
        x ,
        [f,df]( auto x ) { return x - f( x ) / df( x ); } ,
        break_condition );
}
```

Example – Newton method

```cpp
auto f = []( auto x ) { return exp(-x*x) - 0.5; };
auto df = []( auto x ) { return -2.0*x * exp(-x*x); };
auto cond  = [f]( auto x ) {
    return std::abs(f(x)) < 1.0e-12; };

auto range = newton_range( 1.0 , f , df , cond );

for( auto r : range )
    std::cout << r << " : " << f( r ) << std::endl;
```
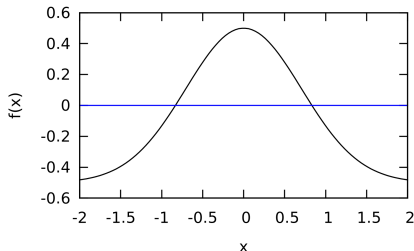
Example – ODE solver

Solve $\dot{x} = f(x, t)$, Solver $F : x(t + \delta t) = F(x(t))$
Example Boost.Odeint:

```cpp
namespace odeint = boost::numeric::odeint;

auto lorenz = []( auto const& x , auto& dxdt , auto t )
{
    dxdt[0] = 10.0 * ( x[1] - x[0] );
    dxdt[1] = 28.0 * x[0] - x[1] - x[0] * x[2];
    dxdt[2] = -8.0 / 3.0 * x[2] + x[0] * x[1];
};

using state_type = std::array< double , 3 >;
odeint::runge_kutta4< state_type > stepper;

state_type x {{ 10.0 , 10.0 , 10.0 }};
double t = 0.0 , dt = 0.01;
stepper.do_step( lorenz , x , t , dt );
t += dt;
```

Integrate functions:

```cpp
auto obs = []( auto x , auto t ) { std::cout << t << " " << x[0] << "\n"; };
odeint::integrate_const( stepper , lorenz , x , 0.0 , 10.0 , dt , obs );
```

Example – ODE Solver

```cpp
auto make_ode_range( auto sys , auto stepper , auto x ,
    auto t0 , auto dt , auto t1 )
{
    auto solve = [sys,stepper,dt]( auto x ) mutable {
        stepper.do_step( sys , x.first , x.second , dt );
        x.second += dt;
        return x; };
    auto cond = [t1]( auto const& x ) { return x.second > t1; };
    auto range = make_map_range( std::make_pair(x,t0) , solver , cond );
    return range;
}
```

Can be used as

```cpp
state_type x {{ 10.0 , 10.0 , 10.0 }};
stepper_type stepper;
auto range = make_ode_range( lorenz , stepper , x , 0.0 , 0.1 , 100.0 );

for( auto r : range )
    std::cout << r.second << " " << r.first[0] << " " << r.first[1] << "\n";
```

ODE ranges

Superior to integrate functions:

- Break conditions are easy
- ODE-Ranges can be used in a natural C++ way, `find`, `transform`, etc.

But, the implementation has some drawbacks -> custom range implementations

- Better performance.
- Step size control – complicated iteration.
- Break condition can use the last two values.

Iterators for GPUs algorithms

High-level libraries for GPUs

1. Thrust
2. VexCL
3. Boost.Compute
4. ViennaCL
5. Cuda-MTL

Thrust

STL-like library for Cuda
Design is based on iterators

Iterators in Thrust

```
device_vector::iterator
host_vector::iterator
```
special iterators
Algorithms

Implementation details of Thrust iterators

Special iterators for Thrust

zip iterator
transform iterator

Special problems - and solutions

Norm

Special problems - and solutions

Bucket sort

Solving an ensemble of low-dimensional ODEs

Lorenz example and ODEs

Conclusion

Outlook

References