FROM ITERATORS TO RANGES: THE UPCOMING EVOLUTION OF THE STL

Nikolay Fedotenko, M3233

RANGES

- **Boost.Range**
- **Range V3** (Eric Niebler)
- **think-cell public library** (evolved from Boost.Range)

WHY RANGES?

```
std::vector<T> vec=...;
std::sort( vec.begin(), vec.end() );
vec.erase( std::unique( vec.begin(), vec.end() ), vec.end() );
```

How often do we have to mention **vec**?
Pairs of iterators belong together => put into one object!

```
tc::unique_inplace(tc::sort(vec));
```

Much nicer!

RANGES IN C++11

- Range-based for

```
for ( int& i : <range_expression> ) {
...
}
```

- Universal access to begin/end

```
std::begin/end(<range_expression>)
```

FUTURE OF RANGES

- Eric Niebler's pet project
- Range's Technical Specification (<algorithm> supports ranges)

```
namespace ranges {
    template< typename Rng, typename What >
    decltype(auto) find( Rng && rng, What const& what ) {
        return std::find(
            ranges::begin(rng),
            ranges::end(rng),
            what
        );
    }
}
```

- Range V3 – preview of what Eric wants to standardize

WHAT ARE RANGES?

- Containers (vector, string, list etc.)
  - own elements
  - deep copying
    - copying copies of elements in O(n)
  - deep constness
    - const objects implies const elements
- Views

VIEWS

```
template<typename It>
struct iterator_range {
	It m_itBegin;
	It m_itEnd;
	It begin() const {
		return m_itBegin;
	}
	It end() const {
		return m_itEnd;
	}
};
```

- reference elements
- shallow copying
  - copying copies reference in O(1)
- shallow constness
  - view object const independent of element const

MORE INTERESTING VIEWS: RANGE ADAPTORS

```
std::vector<int> v;
auto it = ranges::find(v, 4);                    // first element of value 4
```

vs.

```
struct A {
    int id;
    double data;
};
std::vector<A> v={…};
auto it = ranges::find_if(
    v,
    [ ](A const& a) { return a.id == 4; } );  // first element of value 4 in id
```

- Similar in semantics
- Not at all similar in syntax

TRANSFORM ADAPTOR

```
std::vector<int> v;
auto it = ranges::find(v, 4);                    // first element of value 4
```

vs.

```
struct A {
	int id;
	double data;
};
std::vector<A> v={…};
auto it = ranges::find_if(
	tc::transform(v, std::mem_fn(&A::id)), 4);   // first element of value 4 in id
```

TRANSFORM ADAPTOR (2)

```
struct A {
	int id;
	double data;
};
std::vector<A> v={…};
auto it = ranges::find_if(
	tc::transform(v, std::mem_fn(&A::id)), 4);		// it points to int
```

```
auto it = ranges::find_if(
	tc::transform(v, std::mem_fn(&A::id)), 4).base();	// it points to A
```

TRANSFORM ADAPTOR IMPLEMENTATION

```
template<typename Base, typename Func>
struct transform_range {
	struct iterator {
	private:
		Func m_func;  // in every iterator
		decltype( tc::begin(std::declval<Base&>()) ) m_it;
	public:
		decltype(auto) operator*() const {
			return m_func(*m_it);
		}
		 decltype(auto) base() const {
			return (m_it);
		}
		…
	};
};
```

FILTER ADAPTOR

Range of all a with a.id == 4 ?

```
tc::filter(
	v,
	[ ](A const& a) { return 4 == a.id; }
);
```

- Lazy! Filter executed while iterating

FILTER ADAPTOR IMPLEMENTATION

```
template<typename Base, typename Func>
struct filter_range {
    struct iterator {
    private:
        Func m_func;    // functor and TWO iterators
        decltype( ranges::begin(std::declval<Base&>()) ) m_it;
        decltype( ranges::begin(std::declval<Base&>()) ) m_itEnd;
    public:
        iterator& operator++() {
            ++m_it;
            while( m_it != m_itEnd && !static_cast<bool>(m_func(*m_it)) )
                ++m_it;                 // why static_cast<bool> ?
            return *this;
        }
        …
    };
};
```

HOW DOES ITERATOR LOOK LIKE OF

tc::filter(tc::filter(tc::filter(…))) ?

```
m_func3
m_it3
    m_func2
    m_it2
        m_func1
        m_it1;
        m_itEnd1;
    m_itEnd2
        m_func1
        m_it1;
        m_itEnd1;
m_itEnd3
    m_func2
    m_it2
        m_func1
        m_it1;
        m_itEnd1;
    m_itEnd2
        m_func1
        m_it1;
        m_itEnd1;
```

Boost.Range does this! Definitely not efficient…

MORE EFFICIENT RANGE ADAPTORS

Must keep iterators small
Idea: adaptor object carries everything that is common for all iterators

```
m_func
m_itEnd
```

Iterators carry reference to adaptor object (for common stuff) and base iterator

```
*m_rng
m_it
```

- Iterator cannot outlive its range

AGAIN: HOW DOES ITERATOR LOOK LIKE OF

tc::filter(tc::filter(tc::filter(…))) ?

```
m_rng3
m_it3
	m_rng2
	m_it2
		m_rng1
		m_it1
```

- Range V3 State of The Art
- Still not insanely great…

INDEX CONCEPT

Index

- Like iterator
- But all operations require its range object

```
template<typename Base, typename Func>
struct index_range {
	…
	using Index = …;
	Index begin_index() const;
	Index end_index() const;
	void increment_index( Index& idx ) const;
	void decrement_index( Index& idx ) const;
	reference dereference( Index& idx ) const;
	…
};
```

INDEX-ITERATOR COMPATIBILITY

- Index from Iterator
  - using Index = Iterator
  - Index operations = Iterator operations
- Iterator from Index

```
template<typename IndexRng>
struct iterator_for_index {
	IndexRng* m_rng;
	typename IndexRng::Index m_idx;

	iterator& operator++() {
		m_rng.increment_index(m_idx);
		return *this;
	}
	…
};
```

SUPER-EFFICIENT RANGE ADAPTORS WITH INDICES

Index-based filter_range

```
template<typename Base, typename Func>
struct filter_range {
    Func m_func;
    Base& m_base;

    using Index = typename Base::Index;
    void increment_index( Index& idx ) const {
        do {
            m_base.increment_index(idx);
        } while ( idx != m_base.end_index()
            && !m_func(m_base.dereference_index(idx)) );
    }
};
```

SUPER-EFFICIENT RANGE ADAPTORS WITH INDICES

Index-based filter_range

```
template<typename Base, typename Func>
struct filter_range {
        Func m_func;
        Base& m_base;

        using Index = typename Base::Index;
        …
```

```
template<typename IndexRng>
struct iterator_for_index {
        IndexRng* m_rng;
        typename IndexRng::Index m_idx;
        …
```

- All iterators are two pointers
  - irrespective of stacking depth

RANGE V3 AND LVALUE CONTAINERS

If adaptor input is lvalue container

- view::filter creates view
- view is reference, O(1) copy, shallow constness etc.

```
auto rng = view::filter(vec, pred1);
bool b = ranges::any_of(rng, pred2);
```

RANGE V3 AND RVALUE CONTAINERS

If adaptor input is rvalue container

- view::filter cannot create view
- view would hold dangling reference to rvalue

```
auto rng = view::filter(create_vector(), pred1);    // does not compile
bool b = ranges::any_of(rng, pred2);
```

RANGE V3 AND RVALUE CONTAINERS

If adaptor input is rvalue container

- view::filter cannot create view
- view would hold dangling reference to rvalue

```
auto rng = action::filter(create_vector(), pred1);        // compiles
bool b = ranges::any_of(rng, pred2);
```

Big Trap

- not lazy anymore!

THINK-CELL AND RVALUE CONTAINERS

If adaptor input is lvalue container

- tc::filter creates view
- view is reference, O(1) copy, shallow constness etc.

If adaptor input is rvalue container

- tc::filter creates container
- aggregates rvalue container, deep copy, deep constness etc.

Always lazy

- Laziness and container-ness are orthogonal concepts

```
auto rng = tc::filter(vec, pred1);
bool b = ranges::any_of(rng, pred2);
```

```
auto rng = tc::filter(creates_vector(), pred1);
bool b = ranges::any_of(rng, pred2);
```

GENERATOR RANGES

```
template<typename Func>
void traverse_widgets(Func func) {
	if (window1) {
		window1->traverse_widgets(std::ref(func));
	}
	func(button1);
	func(listbox1);
	if (window2) {
		window2->traverse_widgets(std::ref(func));
	}
}
```

- like range of widgets
- but no iterators

GENERATOR RANGES

```
template<typename Func>
void traverse_widgets(Func func) {
	if (window1) {
		window1->traverse_widgets(std::ref(func));
	}
	func(button1);
	func(listbox1);
	if (window2) {
		window2->traverse_widgets(std::ref(func));
	}
}
```

```
mouse_hit_any_widget = tc::any_of(
	[ ] (auto func) { traverse_widgets(func); },
	[ ] (auto const& widget) { return widget.mouse_hit(); }
);
```

ANY_OF IMPLEMENTATION

```
namespace tc {
    template<typename Rng>
    bool any_of(Rng const& rng) {
        bool bResult = false;
        tc::enumerate( rng, [&](bool_context b) {
            bResult = bResult || b;
        } );
        return bResult;
    }
}
```

- tc::enumerate is common interface for iterator, index and generator ranges
- Ok?
  - std::any_of stops when true is encountered!

I HATE THE RANGE-BASED FOR LOOP!

because it encourages people to write this:

```
bool b = false;
for (int n : rng) {
	if ( is_prime(n) ) {
		b = true;
		break;
	}
}
```

instead of this:

```
bool b = tc::any_of( rng, is_prime );
```

RANGES

- **Boost.Range**:
  - https://www.boost.org/doc/libs/1_67_0/libs/range/doc/html/index.html
- **Range V3**:
  - https://github.com/ericniebler/range-v3
- **think-cell**:
  - https:://github.com/think-cell/range
- **Lecture (Arno Schödl)**:
  - https://youtu.be/vrCtS6FDay8

THANK YOU