From Iterators To Ranges

The Upcoming Evolution Of the Standard Library
Questions

- Who knows C++20 Ranges?
- Who knows any other Range library (Boost.Range, Ranges V3, think-cell)?
- Who uses ranges in everyday programming?
```cpp
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`?
**Ranges in C++20**

```cpp
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 -> use one object!

```cpp
std::sort(vec);
vec.erase(std::unique(vec),vec.end());
```
ranges in C++20

```cpp
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 -> use one object!

```cpp
std::sort(vec);
vec.erase(std::unique(vec),vec.end());
```

Can try it now: [https://github.com/ericniebler/range-v3](https://github.com/ericniebler/range-v3)
Why do I think I know something about ranges?

- think-cell has a range library
  - evolved from Boost.Range
- 1 million lines of production code use it
- Library and production code evolve together
  - ready to change library and production code anytime
  - no obstacle to library design changes
  - large code base to try them out
Why do I think I know something about ranges?

- think-cell has a range library
  - evolved from Boost.Range
- 1 million lines of production code use it
- Library and production code evolve together
  - ready to change library and production code anytime
  - no obstacle to library design changes
  - large code base to try them out

```cpp
std::sort(vec);
vec.erase(std::unique(vec), vec.end());
```
Why do I think I know something about ranges?

- think-cell has a range library
  - evolved from Boost.Range
- 1 million lines of production code use it
- Library and production code evolve together
  - ready to change library and production code anytime
  - no obstacle to library design changes
  - large code base to try them out

```cpp
std::sort(vec);
vec.erase(std::unique(vec),vec.end());
```

- Better:

```cpp
tc::sort_unique_inplace(vec);
```
Why do I think I know something about ranges?

- think-cell has a range library
  - evolved from Boost.Range

- 1 million lines of production code use it

- Library and production code evolve together
  - ready to change library and production code anytime
  - no obstacle to library design changes
  - large code base to try them out

```cpp
std::sort(vec);
vec.erase(std::unique(vec),vec.end());
```

- Better:

```cpp
tc::sort_unique_inplace(vec);
tc::sort_unique_inplace(vec, less);
```
What are Ranges?

- Containers
  - `vector`
  - `string`
  - `list`

  - own elements
  - deep copying
    - copying copies elements in $O(N)$
  - deep constness
    - `const` objects implies `const` elements
What are Ranges?

- Containers
  - vector
  - string
  - list
  - own elements
  - deep copying
    - copying copies elements in O(N)
  - deep constness
    - const objects implies const elements

- Views
template<typename It>
struct subrange {
    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

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

vs.

```cpp
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

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

vs.

```cpp
struct A {
    int id;
    double data;
};
std::vector<A> v={...};
auto it=ranges::find(
    v | ranges::view::transform(std::mem_fn(&A::id)),
    4
); // first element of value 4 in id
```
Transform Adaptor (2)

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

What is \texttt{it} pointing to?
```
Transform Adaptor (2)

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

What is `it` pointing to?

- `int`!
Transform Adaptor (2)

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

What is `it` pointing to?

- `int`!

What if I want `it` to point to `A`?
```cpp
struct A {
    int id;
    double data;
};
std::vector<A> v={...};
auto it=ranges::find(
    v | ranges::view::transform(std::mem_fn(&A::id)),
    4
); // first element of value 4 in id
```

What is `it` pointing to?

- `int`

What if I want `it` to point to `A`?

```cpp
auto it=ranges::find(
    v | ranges::view::transform(std::mem_fn(&A::id)),
    4
).base();
```
auto it=ranges::find(
    v | ranges::view::transform(std::mem_fn(&A::id)),
    4
).base(); // DOES NOT COMPILE
auto it=ranges::find(
    v | ranges::view::transform(std::mem_fn(&A::id)),
    4
).base(); // DOES NOT COMPILE

- Protects you from dangling iterators!
auto it=ranges::find(
    v | ranges::view::transform(std::mem_fn(&A::id)),
    4
).base(); // DOES NOT COMPILE

- Protects you from dangling iterators!
- But there is no dangling iterator!
Transform Adaptor (3): C++20

Nannying

```cpp
auto it=ranges::find(
    v | ranges::view::transform(std::mem_fn(&A::id)),
    4
).base(); // DOES NOT COMPILE
```

- Protects you from dangling iterators!
- But there is no dangling iterator!

```cpp
auto it=tc::find<tc::return_element>(
    tc::transform(v, std::mem_fn(&A::id)),
    4
).base();
```
template<typename Base, typename Func>
struct transform_view {
    struct iterator {
        private:
            Func m_func; // in every iterator, hmmm...
            decltype(ranges::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 \(a\cdot id==4\) ?

```cpp
auto rng = v | ranges::views::filter([](A const& a){ return 4==a.id; } );
```

- Lazy! Filter executed while iterating
template<
typename Base,
typename Func>
struct filter_view {
    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 would iterator look like of

```cpp
ranges::view::filter(m_func3)(ranges::view::filter(m_func2)
(ranges::view::filter(m_func1, ...)))
```
m_func3
m_it3
  m_func2
  m_it2
    m_func1
    m_it1;
    m_itEnd1;
  m_itEnd2
    m_func1
    m_itEnd1;
    m_itEnd1;
m_itEnd3
  m_func2
  m_it2
    m_func1
    m_it1;
    m_itEnd1;
    m_itEnd1;
  m_itEnd2
    m_func1
    m_itEnd1;
    m_itEnd1;

Boost.Range did this! ARGH!
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
```
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
```

- C++20 State of the Art
- C++20 iterators cannot outlive their range
Again: How does iterator look like of

```
ranges::view::filter(m_func3)(ranges::view::filter(m_func2)
(ranges::view::filter(m_func1, ...))) ?
```

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

- Still not insanely great...
Beyond C++20 Ranges:

Index Concept

Index

- Like iterator
- But all operations require its range object

```cpp
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 const& idx ) const;
    ...
};
```
Index-Iterator Compatibility

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

```cpp
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;
    }

    ...
};
```
template<typename Base, typename Func>
struct filter_view {
    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
Super-Efficient Range Adaptors With Indices

Index-based filter_view

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

    using Index= typename Base::Index;
    ...
}
```

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

- All iterators are two pointers
  - irrespective of stacking depth
If adaptor input is lvalue container

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

```cpp
auto v = create_vector();
auto rng = v | ranges::view::filter(pred1);
```
If adaptor input is rvalue container

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

```cpp
auto rng = create_vector() | ranges::view::filter(pred1); // DOES NOT COMPILE
```
C++20 Ranges and rvalue containers

If adaptor input is rvalue container

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

```cpp
auto rng = create_vector() | ranges::view::filter(pred1); // DOES NOT COMPILE
```

- Return lazily filtered container?

```cpp
auto foo() {
    auto vec=create_vector();
    return std::make_tuple(vec, ranges::view::filter(pred)(vec));
}
```
If adaptor input is rvalue container

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

```cpp
auto rng = create_vector() | ranges::view::filter(pred1); // DOES NOT COMPILE
```

- Return lazily filtered container?

```cpp
auto foo() {
    auto vec=create_vector();
    return std::make_tuple(vec, ranges::view::filter(pred)(vec)); // DANGLING REFERENCE!
}
```

ARGH!
think-cell and rvalue containers

If adaptor input is lvalue container

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

If adaptor input is rvalue container

- \texttt{tc::filter} creates container
- aggregates rvalue container, deep copy, deep constness etc.

Always lazy

- Laziness and container-ness are orthogonal concepts

```cpp
auto vec=create_vector();
auto rng=tc::filter(vec,pred1);

auto foo() {
    return tc::filter(creates_vector(),pred1);
}
```
Beyond C++20 Ranges:

More Flexible Algorithm Returns

template< typename Rng, typename What >
decltype(auto) find( Rng && rng, What const& what ) {
    auto const itEnd=ranges::end(rng);
    for( auto it=ranges::begin(rng); it!=itEnd; ++it )
        if( *it==what )
            return it;
    return itEnd;
}
More Flexible Algorithm Returns (2)

```cpp
template< typename Pack, typename Rng, typename What >
dcltype(auto) find( Rng && rng, What const & what ) {
    auto const itEnd=ranges::end(rng);
    for( auto it=ranges::begin(rng); it!=itEnd; ++it )
        if( *it==what )
            return Pack::pack(it,rng);
    return Pack::pack_singleton(rng);
}

struct return_element_or_end {
    static auto pack(auto it, auto&& rng) {
        return it;
    }
    static auto pack_singleton(auto&& rng) {
        return ranges::end(rng);
    }
}

auto it=find<return_element_or_end>(...)
```
More Flexible Algorithm Returns (3)

```cpp
template< typename Pack, typename Rng, typename What >
decltype(auto) find( Rng && rng, What const& what ) {
    auto const itEnd=ranges::end(rng);
    for( auto it=ranges::begin(rng); it!=itEnd; ++it )
        if( *it==what )
            return Pack::pack(it, rng);
    return Pack::pack_singleton(rng);
}

struct return_element {
    static auto pack(auto it, auto&& rng) {
        return it;
    }
    static auto pack_singleton(auto && rng) {
        std::assert(false);
        return ranges::end(rng);
    }
}

auto it=find<return_element>(...)
```
More Flexible Algorithm Returns (3)

```cpp
template< typename Pack, typename Rng, typename What >
decltype(auto) find( Rng && rng, What const& what ) {
    auto const itEnd=ranges::end(rng);
    for( auto it=ranges::begin(rng); it!=itEnd; ++it )
        if( *it==what )
            return Pack::pack(it,rng);
    return Pack::pack_singleton(rng);
}

struct return_element_or_null {
    static auto pack(auto it, auto&& rng) {
        return tc::element_t<decltype(it)>(it);
    }
    static auto pack_singleton(auto&& rng) {
        return tc::element_t<decltype(ranges::end(rng))>();
    }
}

if( auto it=find<return_element_or_null>(...) ) { ... }
```
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
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();
    }
);
External Iteration

- Consumer calls producer to get new element
- example: C++ iterators

```
^ Stack       Producer     Producer
  |            /          /          \
  Consumer   Consumer   Consumer
```

- Consumer is at bottom of stack
- Producer is at top of stack
External iteration (2)

Consumer is at bottom of stack

- contiguous code path for whole range
- easier to write
- better performance
  - state encoded in instruction pointer
  - no limit for stack memory

Producer is at top of stack

- contiguous code path for each item
- harder to write
- worse performance
  - single entry point, must restore state
  - fixed amount of memory or go to heap
Internal Iteration

- Producer calls consumer to offer new element
- example: EnumThreadWindows

Producer is at bottom of stack

- ... all the advantages of being bottom of stack ...

Consumer is at top of stack

- ... all the disadvantages of being top of stack ...
Can both consumer and producer be bottom-of-stack?

- Yes, with coroutines

```cpp
// does not compile, conceptual
generator<widget&> traverse_widgets() {
  if( window1 ) {
    window1->traverse_widgets();
  }
  co_yield button1;
  co_yield listbox1;
  if( window2 ) {
    window2->traverse_widgets();
  }
}
```
Coroutines (2)

- Stackful
  - use two stacks and switch between them
  - very expensive
    - implemented as OS fibers
    - 1 MB of virtual memory per coroutine

- Stackless (C++20)
  - whole callstack must be coroutine-d

```cpp
// does not compile, conceptual
generator<widget&> traverse_widgets() {
    if( window1 ) {
        co_yield window1->traverse_widgets();
    }
    co_yield button1;
    co_yield listbox1;
    if( window2 ) {
        co_yield window2->traverse_widgets();
    }
}
```
Coroutines (2)

- Stackful
  - use two stacks and switch between them
  - very expensive
    - implemented as OS fibers
    - 1 MB of virtual memory per coroutine

- Stackless (C++20)
  - whole callstack must be coroutine-d

```cpp
// does not compile, conceptual
generator<widget&> traverse_widgets() {
  tc::for_each( windows1, (auto const& window2) {
    co_yield window1->traverse_widgets(); // DOES NOT COMPILE
  });
  co_yield button1;
  co_yield listbox1;
  tc::for_each( windows2, (auto const& window2) {
    co_yield window2->traverse_widgets(); // DOES NOT COMPILE
  }
}
```
Coroutines (2)

- Stackful
  - use two stacks and switch between them
  - very expensive
    - implemented as OS fibers
    - 1 MB of virtual memory per coroutine

- Stackless (C++20)
  - can only yield in top-most function
  - still a bit expensive
    - dynamic jump to resume point
    - save/restore some registers
    - no aggressive inlining
Internal Iteration often good enough

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Internal Iteration?</th>
</tr>
</thead>
<tbody>
<tr>
<td>find</td>
<td>no (single pass iterators)</td>
</tr>
<tr>
<td>binary_search</td>
<td>no (random access iterators)</td>
</tr>
</tbody>
</table>

Internal Iteration often good enough

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Internal Iteration?</th>
</tr>
</thead>
<tbody>
<tr>
<td>find</td>
<td>no (single pass iterators)</td>
</tr>
<tr>
<td>binary_search</td>
<td>no (random access iterators)</td>
</tr>
<tr>
<td>for_each</td>
<td>yes</td>
</tr>
<tr>
<td>accumulate</td>
<td>yes</td>
</tr>
<tr>
<td>all_of</td>
<td>yes</td>
</tr>
<tr>
<td>any_of</td>
<td>yes</td>
</tr>
<tr>
<td>none_of</td>
<td>yes</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
### Internal Iteration often good enough

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Internal Iteration?</th>
</tr>
</thead>
<tbody>
<tr>
<td>find</td>
<td>no (single pass iterators)</td>
</tr>
<tr>
<td>binary_search</td>
<td>no (random access iterators)</td>
</tr>
<tr>
<td>for_each</td>
<td>yes</td>
</tr>
<tr>
<td>accumulate</td>
<td>yes</td>
</tr>
<tr>
<td>all_of</td>
<td>yes</td>
</tr>
<tr>
<td>any_of</td>
<td>yes</td>
</tr>
<tr>
<td>none_of</td>
<td>yes</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Adaptor</th>
<th>Internal Iteration?</th>
</tr>
</thead>
<tbody>
<tr>
<td>tc::filter</td>
<td>yes</td>
</tr>
<tr>
<td>tc::transform</td>
<td>yes</td>
</tr>
</tbody>
</table>

So allow ranges that support only internal iteration!
namespace tc {
    template< typename Rng >
    bool any_of( Rng const& rng ) {
        bool bResult=false;
        tc::for_each( rng, [&] (bool_context b) {
            bResult = bResult || b;
        } );
        return bResult;
    }
}

- `tc::for_each` is common interface for iterator, index and generator ranges
- Ok?
any_of implementation

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

- **tc::for_each** is common interface for iterator, index and generator ranges
- Ok?
  - **ranges::any_of** stops when true is encountered!
Interruptable Generator Ranges

First idea: exception!
Interruptable Generator Ranges

First idea: exception!

- too slow:-(
Interruptable Generator Ranges

First idea: exception!

- too slow:-(

Second idea:

```cpp
enum break_or_continue {
    break_,
    continue_
};

template< typename Rng >
bool any_of( Rng const& rng ) {
    bool bResult=false;
    tc::for_each( rng, [&](bool_context b){
        bResult=bResult || b;
        return bResult ? break_ : continue_;
    } );
    return bResult;
}
```
Interruptable Generator Ranges (2)

- Generator Range can elide `break_` check
  - If functor returns `break_or_continue`
    - break if `break_` is returned.
  - If functor returns anything else,
    - nothing to check, always continue
```cpp
std::list<int> lst;
std::vector<int> vec;

std::for_each( tc::concat(lst, vec), [](int i) {
    ...
});
```
concat implementation with indices

```cpp
template<typename Rng1, typename Rng2>
struct concat_range {
    private:
        using Index1 = typename range_index<Rng1>::type;
        using Index2 = typename range_index<Rng2>::type;

        Rng1& m_rng1;
        Rng2& m_rng2;
        using index = std::variant<Index1, Index2>;
    public:
        ...
```
void increment_index(index& idx) {
    idx.switch(
        [&](Index1& idx1) {
            m_rng1.increment_index(idx1);
            if (m_rng1.at_end_index(idx1)) {
                idx = m_rng2.begin_index();
            }
        },
        [&](Index2& idx2) {
            m_rng2.increment_index(idx2);
        });
    }

• Branch for each increment!
concat implementation with indices

(3)

... 

auto dereference_index(index const& idx) const {
    return idx.switch(
        [&](Index1 const& idx1){
            return m_rng1.dereference(idx1);
        },
        [&](Index2 const& idx2){
            return m_rng2.dereference(idx2);
        });
    }
};

• Branch for each dereference!
• How avoid all these branches?
concat implementation with indices (3)

```cpp
... auto dereference_index(index const& idx) const {
    return idx.switch(
        [&](Index1 const& idx1){
            return m_rng1.dereference(idx1);
        },
        [&](Index2 const& idx2){
            return m_rng2.dereference(idx2);
        }
    );
    ...
};
```

- Branch for each dereference!
- How avoid all these branches?
  - With Generator Ranges!
concat implementation as generator range

```cpp
template<typename Rng1, typename Rng2>
struct concat_range {
    private:
        Rng1 m_rng1;
        Rng2 m_rng2;

    public:
        ...

        // version for non-breaking func
        template<typename Func>
        void operator() (Func func) {
            tc::for_each(m_rng1, func);
            tc::for_each(m_rng2, func);
        }
};
```

- Even iterator-based ranges sometimes perform better with generator interface!
Now that we have all this range stuff

- URL of our range library: https://github.com/think-cell/range
Now that we have all this range stuff

- URL of our range library: https://github.com/think-cell/range

I hate the range-based for loop!
Now that we have all this range stuff

- URL of our range library: https://github.com/think-cell/range

I hate the range-based for loop!

because it encourages people to write this

```cpp
bool b=false;
for( int n : rng ) {
    if( is_prime(n) ) {
        b=true;
        break;
    }
}
```
Now that we have all this range stuff

- URL of our range library: https://github.com/think-cell/range

I hate the range-based for loop!

because it encourages people to write this

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

instead of this

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

THANK YOU!