think-cell

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?

Ranges in C++20

```
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

think-cell

```
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!

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

Ranges in C++20

think-cell

```
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!

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

Can try it now: 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

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

Why do I think I know something about ranges?

think-cell

- 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

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

- Better:

```
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

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

- Better:

```
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

```
            Range
             /\
            /  \
           /    \
   Container   View
```

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

```
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(
    v | views::transform(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(
    v | views::transform(std::mem_fn(&A::id)),
    4
); // first element of value 4 in id
```

What is `it` pointing to?

Transform Adaptor (2)

think-cell

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

What is `it` pointing to?

- `int`!

Transform Adaptor (2)

```
struct A {
    int id;
    double data;
};
std::vector<A> v={...};
auto it=ranges::find(
    v | views::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`?

Transform Adaptor (2)

think-cell

```
struct A {
    int id;
    double data;
};
std::vector<A> v={...};
auto it=ranges::find(
    v | views::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`?

```
auto it=ranges::find(
    v | views::transform(std::mem_fn(&A::id)),
    4
).base();
```

Transform Adaptor (3): C++20 Nannying

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

Transform Adaptor (3): C++20 Nannying

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

- Protects you from dangling iterators!

Transform Adaptor (3): C++20 Nannying

```
auto it=ranges::find(
    v | views::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

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

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

```
auto it=tc::find<tc::return_element>(
    tc::transform(v, std::mem_fn(&A::id)),
    4
).base();
```

Transform Adaptor Implementation

think-cell

```
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.id==4`?

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

- Lazy! Filter executed while iterating

Filter Adaptor Implementation

think-cell

```
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

`views::filter(m_func3)(views::filter(m_func2)(views::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_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

`views::filter(m_func3)(views::filter(m_func2)(views::filter(m_func1, ...)))` ?

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

- Still not insanely great...

think-cell

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 const& 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

think-cell

Index-based filter_view

```
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

think-cell

Index-based filter_view

```
template<typename Base, typename Func>
struct filter_view {
    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

C++20 Ranges and rvalue containers

If adaptor input is lvalue container

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

```
auto v = create_vector();
auto rng = v | views::filter(pred1);
```

C++20 Ranges and rvalue containers

If adaptor input is rvalue container

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

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

C++20 Ranges and rvalue containers

If adaptor input is rvalue container

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

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

- Return lazily filtered container?

```
auto foo() {
    auto vec=create_vector();
    return std::make_tuple(vec, views::filter(pred)(vec));
}
```

C++20 Ranges and rvalue containers

think-cell

If adaptor input is rvalue container

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

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

- Return lazily filtered container?

```
auto foo() {
    auto vec=create_vector();
    return std::make_tuple(vec, views::filter(pred)(vec)); // DANGLING REFER
ENCE!
}
```

ARGH!

think-cell and rvalue containers

think-cell

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

```
auto foo() {
    return tc::filter(creates_vector(),pred1);
}
```

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)

think-cell

```
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_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)

think-cell

```
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)

```
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>(...) ) { ... }
```

Generator Ranges

think-cell

```
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

think-cell

```
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

think-cell

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

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

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 ...

Coroutines

Can both consumer and producer be bottom-of-stack?

- Yes, with coroutines

```
// 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

```
// 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)

think-cell

- 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

```
// 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

Algorithm	Internal Iteration?
find	no (single pass iterators)
binary_search	no (random access iterators)

Internal Iteration often good enough

Algorithm	**Internal Iteration?**
find	no (single pass iterators)
binary_search	no (random access iterators)
for_each	yes
accumulate	yes
all_of	yes
any_of	yes
none_of	yes
...	

Internal Iteration often good enough

Algorithm	**Internal Iteration?**
find	no (single pass iterators)
binary_search	no (random access iterators)
for_each	yes
accumulate	yes
all_of	yes
any_of	yes
none_of	yes
...	

Adaptor	**Internal Iteration?**
tc::filter	yes
tc::transform	yes

So allow ranges that support only internal iteration!

any_of implementation

think-cell

```
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

think-cell

```
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

think-cell

First idea: exception!

- too slow:-(

Second idea:

```
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

concat

```
std::list<int> lst;
std::vector<int> vec;

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

concat implementation with indices

```
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:
...
```

concat implementation with indices (2)

```
...
    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)

```
...
    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

```
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

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

I hate the range-based for loop!

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;
    }
}
```

Now that we have all this range stuff

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

think-cell

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=ranges::any_of( rng, is_prime );
```

THANK YOU!