From Iterators to Ranges: The Upcoming Evolution of the Standard Library

Arno Schoedl
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`?
How often do we have to mention `vec`?

Pairs of iterators belong together -> use one object!

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

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

If you have no C++20 compiler: 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);
```

```cpp
tc::sort_unique_inplace(vec, std::less<int>());
```
What are Ranges?

- Containers

- own elements
- deep copying
  - copying copies elements in O(N)

- deep constness
  - `const` objects implies `const` elements
What are Ranges?

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

- Views
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
std::vector<int> v{1,2,4};
auto it=ranges::find(
    v,
    4
); // first element of value 4.

```
struct A {
    int id;
    double data;
};
std::vector<int> v{1,2,4};
auto it=ranges::find_if(
    v,
    [](A const& a){ return a.id==4; } // first element of value 4 in id
);
```

• Similar in semantics
```cpp
std::vector<int> v{1, 2, 4};
auto it = ranges::find(
    v,
    4
); // first element of value 4.
```

vs.

```cpp
struct A {
    int id;
    double data;
};
std::vector<int> v{1, 2, 4};
auto it = ranges::find(
    v | views::transform(std::mem_fn(&A::id)),
    4
); // first element of value 4 in id
```
```cpp
struct A {
    int id;
    double data;
};
std::vector<int> v{1,2,4};
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?
```cpp
struct A {
    int id;
    double data;
};
std::vector<int> v{1, 2, 4};
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)

```cpp
struct A {
    int id;
    double data;
};
std::vector<int> v{1, 2, 4};
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`?

15 / 110
struct A {
    int id;
    double data;
};
std::vector<int> v{1, 2, 4};
auto it = ranges::find(
    v | views::transform(std::mem_fn(&A::id)),
    4
).base(); // 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();
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 \( a.id == 4 \)?

```cpp
auto rng = v | 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

`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

```c
m_func
m_itEnd
```

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

```c
*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
  - unless it is a `std::ranges::borrowed_range`
auto it=ranges::find(
    v | views::transform(std::mem_fn(&A::id)),
    4
).base(); // DOES NOT COMPILE
More Efficient Range Adaptors: Iterator Safety

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

- Iterator from rvalue
- Danger of dangling reference!
More Efficient Range Adaptors: Iterator Safety

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

- No actual dangling reference because of `.base()`
- Silence error
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...
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;
    }

    ...
};
```
Index-based filter_view

```cpp
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);
            m_base.dereference_index(idx);
        } while( idx!=m_base.end_index() && !static_cast<bool>(m_func(m_base.dereference_index(idx))) );
    }
};
```
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
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.

```cpp
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

```c++
auto rng = create_vector() | views::filter(pred1); // DOES NOT COMPILE
```
If adaptor input is rvalue container

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

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

- Return lazily filtered container?

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

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

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

- Return lazily filtered container?

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

ARGH!
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

```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;
}
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)

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 {
    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>(...)


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 Sink>
void traverse_widgets( Sink sink ) {
    if( window1 ) {
        window1->traverse_widgets(std::ref(sink));
    }
    sink(button1);
    sink(listbox1);
    if( window2 ) {
        window2->traverse_widgets(std::ref(sink));
    }
}

- like range of widgets
- but no iterators
template<typename Sink>
void traverse_widgets( Sink sink ) {
    if ( window1 ) {
        window1->traverse_widgets(std::ref(sink));
    }
    sink(button1);
    sink(listbox1);
    if ( window2 ) {
        window2->traverse_widgets(std::ref(sink));
    }
}

mouse_hit_any_widget=tc::any_of(
    [](auto sink){ traverse_widgets(sink); },
    [](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 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: for_each_xxx, "visitor"

```
^       Stack
|       /       
Producer  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 ...
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() {
    ranges::for_each( windows1, []{(auto const& window1) {
        co_yield window1->traverse_widgets(); // DOES NOT COMPILE
    };
    co_yield button1;
    co_yield listbox1;
    ranges::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
<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>Algorithm</td>
<td>Internal Iteration?</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------</td>
</tr>
<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>
</tbody>
</table>
### Algorithm vs. Internal Iteration?

<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>
</tbody>
</table>

### Adaptor vs. Internal Iteration?

<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!
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?
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
std::list<int> lst;
std::vector<int> vec;

tc::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)

```cpp
... void increment_index(index& idx) {
    std::visit(tc::make_overload(
        [&](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);
        }), idx);
...}
```

- Branch for each increment!
concat implementation with indices (3)

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

- Branch for each dereference!
- How avoid all these branches?
concat implementation with indices

... auto dereference_index(index const& idx) const {
    std::visit(tc::make_overload(
        [&](Index1 const& idx1){
            return m_rng1.dereference(idx1);
        },
        [&](Index2 const& idx2){
            return m_rng2.dereference(idx2);
        }
    ), idx);
    ...
}...
**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!
Ranges instead of `std::format`?

- C++20 `std::format` formatters write to output iterators
  - internal iteration!
Ranges instead of `std::format`?

- C++20 `std::format` formatters write to output iterators
  - internal iteration!

- Can rewrite formatters as generator ranges:
  ```cpp
double f = 3.14;
tc::concat("You won ", tc::as_dec(f, 2), " dollars.")
  ```

- single unifying concept instead of separate `std::format`
Ranges instead of `std::format`?

- C++20 `std::format` formatters write to output iterators
  - internal iteration!
- Can rewrite formatters as generator ranges:

```cpp
double f=3.14;
tc::concat("You won ", tc::as_dec(f, 2), " dollars.")
```

- single unifying concept instead of separate `std::format`
- not like `<iostream>`: `double` itself is not a character range:

```cpp
tc::concat("You won ", f, " dollars.")  // DOES NOT COMPILE
```
Ranges instead of std::format (2)

- Extensible by functions returning ranges

```cpp
auto dollars(double f) {
    return tc::concat("$", tc::as_dec(f, 2));
}

double f=3.14;
tc::concat("You won ", dollars(f), ".");
```
Format Strings

tc::concat(
    "<body>", html_escape(
        tc::placeholders( "You won {0} dollars." , tc::as_dec(f,2) )
    ), "</body>"
)
Format Strings

```
tc::concat(
    "<body>", html_escape(
        tc::placeholders( "You won {0} dollars." , tc::as_dec(f, 2) )
    ), "</body>"
)
```

- support for names

```
tc::concat(
    "<body>", html_escape(
        tc::placeholders( "You won {amount} dollars on {date}." ,
            tc::named_arg("amount", tc::as_dec(f, 2)) ,
            tc::named_arg("date", tc::as_ISO8601(
                std::chrono::system_clock::now()
            ))
    ), "</body>"
)
```

- Formatting parameters (#decimal digits etc.) not part of format string
  - Internationalization: translator can rearrange placeholders, but not change parameters
std::string gives us

- Empty Construction

```cpp
std::string s; // compiles
```

- Construction from literal, another string

```cpp
std::string s1("Hello"); // compiles
std::string s2(s1); // compiles
```
std::string gives us

- Empty Construction

```cpp
std::string s; // compiles
```

- Construction from literal, another string

```cpp
std::string s1("Hello"); // compiles
std::string s2(s1); // compiles
```

- Add construction from 1 Range

```cpp
std::string s3(tc::as_dec(3.14,2)); // suggested
std::string s4(tc::concat("You won ", tc::as_dec(3.14,2), " dollars."); // suggested
```
std::string gives us

- Empty Construction

```cpp
std::string s; // compiles
```

- Construction from literal, another string

```cpp
std::string s1("Hello"); // compiles
std::string s2(s1); // compiles
```

- Add construction from 1 Range

```cpp
std::string s3(tc::as_dec(3.14,2)); // suggested
std::string s4(tc::concat("You won ", tc::as_dec(3.14,2), " dollars."))); // suggested
```

- Add construction from N Ranges

```cpp
std::string s5("Hello", " World"); // suggested
std::string s6("You won ", tc::as_dec(3.14,2), " dollars."); // suggested
```
What about existing constructors?

```cpp
std::string s1("A", 3);
std::string s2('A', 3);
std::string s3(3, 'A');
```
What about existing constructors?

```cpp
std::string s1("A", 3); // UB, buffer "A" overrun
std::string s2('A', 3);
std::string s3(3, 'A');
```
What about existing constructors?

```cpp
std::string s1("A", 3); // UB, buffer "A" overrun
std::string s2('A', 3); // Adds 65x Ctrl-C
std::string s3(3, 'A');
```
Formatting Into Containers (2)

- What about existing constructors?

```cpp
std::string s1("A", 3); // UB, buffer "A" overrun
std::string s2('A', 3); // Adds 65x Ctrl-C
std::string s3(3, 'A'); // Adds 3x 'A'
```
What about existing constructors?

```cpp
std::string s1("A", 3); // UB, buffer "A" overrun
std::string s2('A', 3); // Adds 65x Ctrl-C
std::string s3(3, 'A'); // Adds 3x 'A'
```

Deprecate them!

```cpp
std::string s(tc::repeat_n('A', 3)); // suggested, repeat_n as in Range-v3
```
think-cell library uses `tc::explicit_cast` to simulate adding/removing explicit constructors:

```cpp
auto s4=tc::explicit_cast<std::string>("Hello", " World");
auto s5=tc::explicit_cast<std::string>("You won ", tc::as_dec(f,2), " dollar s.");
```
think-cell library uses `tc::explicit_cast` to simulate adding/removing explicit constructors:

```cpp
auto s4=tc::explicit_cast<std::string>("Hello", " World");
auto s5=tc::explicit_cast<std::string>("You won ", tc::as_dec(f, 2), " dollar s."");
```

`tc::cont_emplace_back` wraps `.emplace_back/ .push_back`, uses `tc::explicit_cast` as needed:

```cpp
std::vector<std::string> vec;
tc::cont_emplace_back( vec, tc::as_dec(3.14, 2) );
```
think-cell library uses `tc::explicit_cast` to simulate adding/removing explicit constructors:

```cpp
auto s4=tc::explicit_cast<std::string>("Hello", " World");
auto s5=tc::explicit_cast<std::string>("You won ", tc::as_dec(f, 2), " dollars.");
```

`tc::cont_emplace_back` wraps `.emplace_back`/`.push_back`, uses `tc::explicit_cast` as needed:

```cpp
std::vector<std::string> vec;
tc::cont_emplace_back(vec, tc::as_dec(3.14, 2));
```

Can `tc::append`:

```cpp
std::string s;
tc::append(s, tc::concat("You won ", tc::as_dec(f, 2), " dollars."));
tc::append(s, "You won ", tc::as_dec(f, 2), " dollars.");
```
Fast Formatting Into Containers

- determine string length
- allocate memory for whole string at once
- fill in characters
Fast Formatting Into Containers

- determine string length
- allocate memory for whole string at once
- fill in characters

```cpp
template<
    typename Cont, 
    typename Rng>
auto explicit_cast(Rng const& rng) {
    return Cont(ranges::begin(rng), ranges::end(rng));
}
```

// note: there are more explicit_cast implementations for types other than containers
Fast Formatting Into Containers

- determine string length
- allocate memory for whole string at once
- fill in characters

```cpp
template<
    typename Cont,
    typename Rng>
auto explicit_cast(Rng const& rng) {
    return Cont(ranges::begin(rng), ranges::end(rng));
}
```

// note: there are more explicit_cast implementations for types other than containers

- for non-random-access ranges, `string` ctor runs twice over `rng` :-(
  - first determine size
  - then copy characters
Fast Formatting Into Containers

- avoid traversing `rng` twice
  - `rng` implements `size()` member
  - explicit loop to take advantage of `std::size`

```cpp
template<typename Cont, typename Rng, std::enable_if<
    Rng has size and is not random-access
> >
auto explicit_cast(Rng const& rng) {
    Cont cont;
    cont.reserve( std::size(rng) );
    for(auto it=ranges::begin(rng); it!=ranges::end(rng); ++it) {
        tc::cont_emplace_back(cont, *it);
    }
    return cont;
}
```
Fast Formatting Into Containers

- also have `tc::append`

```cpp
template<typename Cont, typename Rng, std::enable_if<Rng has size and is not random-access>
void append(Cont& cont, Rng const& rng) {
    cont.reserve(cont.size() + std::size(rng));
    for(auto it= ranges::begin(rng); it!=ranges::end(rng); ++it) {
        tc::cont_emplace_back(cont, *it);
    }
}
```
also have `tc::append`

```cpp
template<typename Cont, typename Rng, std::enable_if<Rng has size and is not random-access>
void append(Cont& cont, Rng const& rng) {
    cont.reserve( cont.size() + std::size(rng) );
    for(auto it=ranges::begin(rng); it!=ranges::end(rng); ++it) {
        tc::cont_emplace_back(cont, *it);
    }
}
```

all good?
Fast Formatting Into Containers

- also have `tc::append`

```cpp
template<typename Cont, typename Rng, std::enable_if<
    Rng has size and is not random-access
> >
void append(Cont& cont, Rng const& rng) {
    cont.reserve( cont.size() + std::size(rng) );
    for(auto it=ranges::begin(rng); it!=ranges::end(rng); ++it) {
        tc::cont_emplace_back(cont, *it);
    }
}
```

- `.reserve` is evil!!!
Better reserve

- when adding N elements, guarantee \( O(N) \) moves and \( O(\log(N)) \) memory allocations!

```cpp
template< typename Cont >
void cont_reserve( Cont& cont, typename Cont::size_type n ) {
    if( cont.capacity()<n ) {
        cont.reserve(max(n,cont.capacity()*=8/5));
    }
}
```

```cpp
template< typename Cont, typename Rng, enable_if<
    Rng has size and is not random-access
    > >
void append(Cont& cont, Rng const& rng) {
    tc::cont_reserve( cont.size() + std::size(rng) );
    for(auto it=ranges::begin(rng); it!=ranges::end(rng); ++it) {
        tc::cont_emplace_back(cont, *it);
    }
}
```
template<
typename Cont, typename Rng, enable_if<
    Rng has size and is not random-access
>>
void append(Cont& cont, Rng const& rng) {
    tc::cont_reserve( cont.size() + std::size(rng) );
    for(auto it=ranges::begin(rng); it!=ranges::end(rng); ++it) {
        tc::cont_emplace_back(cont, *it);
    }
}
What about generator ranges?
Appender Customization Point

- introduce `appender` sink for `explicit_cast` and `append` to use

```cpp
template<typename Cont, typename Rng>
void append(Cont& cont, Rng&& rng) {
    tc::for_each(std::forward<Rng>(rng), tc::appender(cont));
}
```
introduce **appender** sink for **explicit_cast** and **append** to use

```cpp
template<typename Cont, typename Rng>
void append(Cont& cont, Rng&& rng) {
    tc::for_each(std::forward<Rng>(rng), tc::appender(cont));
}
```

**appender** customization point

- returned by **container::appender()** member function
- default for **std::** containers

```cpp
template<typename Cont>
struct appender {
    Cont& m_cont;
    template<typename T> void operator()(T&& t) {
        tc::cont_emplace_back(m_cont, std::forward<T>(t));
    }
};
```
Appender Customization Point

- introduce `appender` sink for `explicit_cast` and `append` to use

```cpp
template<typename Cont, typename Rng>
void append(Cont& cont, Rng&& rng) {
    tc::for_each(std::forward<Rng>(rng), tc::appender(cont));
}
```

- `appender` customization point
  - returned by `container::appender()` member function
  - default for `std::` containers

```cpp
template<typename Cont>
struct appender {
    Cont& m_cont;

    template<typename T>
    void operator()(T&& t) {
        tc::cont_emplace_back(m_cont, std::forward<T>(t));
    }
};
```

- Isn't this just `std::back_inserter`?
Chunk Customization Point

- What about `reserve`?
  - Sink needs whole range to call `std::size` before iteration
What about \texttt{reserve}? 
- Sink needs whole range to call \texttt{std::size} before iteration
- new Sink customization point \texttt{chunk}
  - if available, \texttt{tc::for_each} calls it with whole range

```cpp
#include <iostream>

// Customization point for sink

template<typename Cont, enable_if<Cont has reserve()> >
struct reserving_appender : appender<Cont> {

  template<typename Rng, enable_if<Rng has size()> >
  void chunk(Rng&& rng) const {
    auto m_cont = m_cont;
    m_cont.reserve(m_cont.size() + std::size(rng));
    tc::for_each( std::forward<Rng>(rng),
                  static_cast<appender<Cont> const&>(*this) )
  }

};
```
file sink advertises interest in contiguous memory chunks

```cpp
struct file_appender {
    void chunk(std::span<unsigned char const> rng) const {
        std::fwrite(rng.begin(), 1, rng.size(), m_file);
    }
    void operator() (unsigned char ch) const {
        chunk(tc::single(ch));
    }
};
```
Performance: Appender vs Hand-Written

- How much loss compared to hand-written code?
  - trivial formatting task 10x 'A' + 10x 'B' + 10x 'C' best to expose overhead

```c
struct Buffer {
    char achBuffer[1024];
    char* pchEnd=&achBuffer[0];
} buffer;

void repeat_handwritten(char chA, int cchA,
                         char chB, int cchB,
                         char chC, int cchC)
{
    for (auto i = cchA; 0 < i; --i) {
        *buffer.pchEnd=chA;
        ++buffer.pchEnd;
    }
    ... cchB ... chB ...
    ... cchC ... chC ...
}
```
struct Buffer {
  ...
  auto appender() & {
    struct appender_t {
      Buffer* m_buffer;
      void operator()(char ch) noexcept {
        *m_buffer->pchEnd=ch;
        ++m_buffer->pchEnd;
      }
    };
    return appender_t{this};
  }
} buffer;

void repeat_with_ranges(char chA, int cchA, char chB, int cchB, char chC, int cchC) {
  tc::append(buffer, tc::repeat_n(chA,cchA), tc::repeat_n(chB,cchB),
             tc::repeat_n(chC,cchC));
}
Performance: Appender vs Hand-Written

- `repeat_n` iterator-based
  - ~50% more time than hand-written (Visual C++ 15.8)

- `repeat_n` supports internal iteration
  - ~15% more time than hand-written (Visual C++ 15.8)

- Test is worst case: actual work is trivial
  - smaller difference for, e.g., converting numbers to strings
Performance: Custom vs Standard Appender

- toy `basic_string` implementation
  - only heap: pointers `begin`, `end`, `end_of_memory`
- Again trivial formatting task: 10x 'A' + 10x 'B' + 10x 'C'

```cpp
void repeat_with_ranges(
    char chA, int cchA,
    char chB, int cchB,
    char chC, int cchC
) {
    tc::append(mystring,
        tc::repeat_n(chA,cchA), tc::repeat_n(chB,cchB),
        tc::repeat_n(chC,cchC));
}
```
Performance: Custom vs Standard Appender

- Standard Appender

```cpp
template<typename Cont>
struct appender {
    Cont& m_cont;

template<typename T>
void operator()(T&& t) {
    m_cont.emplace_back(std::forward<T>(t));
}
};

template<typename Cont, enable_if<Cont has reserve()> >
struct reserving_appender : appender<Cont> {

template<typename Rng, enable_if<Rng has size()> >
void chunk(Rng&& rng) const {
    tc::cont_reserve(m_cont, m_cont.size()+std::size(rng));
    tc::for_each(std::forward<Rng>(rng),
         static_cast<appender<Cont> const&>(*this)
    );
}
};
```
Performance: Custom vs Standard Appender

- Custom Appender

```cpp
template<
typename Cont>
struct mystring_appender : appender<Cont> {
    Cont& m_cont;

    template<
typename T>
    void operator()(T&& t) {
        m_cont.emplace_back(std::forward<T>(t));
    }

    template<
typename Rng, enable_if<
    Rng has size()> >
    void chunk(Rng&& rng) const {
        tc:: cont_reserve( m_cont, m_cont.size()+std::size(rng) );
        tc:: for_each( std::forward<Rng>(rng),
            [&](auto&& t) {
                *m_cont.m_ptEnd=std::forward<decltype(t)>(t);
                ++m_cont.m_ptEnd;
            }
        );
    }
};
```
Performance: Custom vs. Standard Appender

- String was only 30 characters
- Heap allocation
- Custom Appender ~20% less time (Visual C++ 15.8)
- Requires own `basic_string` implementation
  - uninitialized buffer not exposed by `std::basic_string/std::vector`
Performance: Future Work

- if not all snippets implement `size()`: new customization point `min_size()`?
  - `concat::min_size()` is sum of `min_size()` of components
  - `min_size()` never wrong to return `0`

- custom file appender that fills fixed I/O buffer
  - replace `std::FILE` buffer with own buffer
  - offer unchecked write as long as snippet `size()` still fits
  - new customization point `max_size`?
Conclusion

• Ranges are very useful
• Index-based ranges and generators improve performance over C++20 iterator-based ranges
• Unify ranges with text formatting
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!