Ranges for the Standard Library

NWCPP, May 2015

Eric Niebler

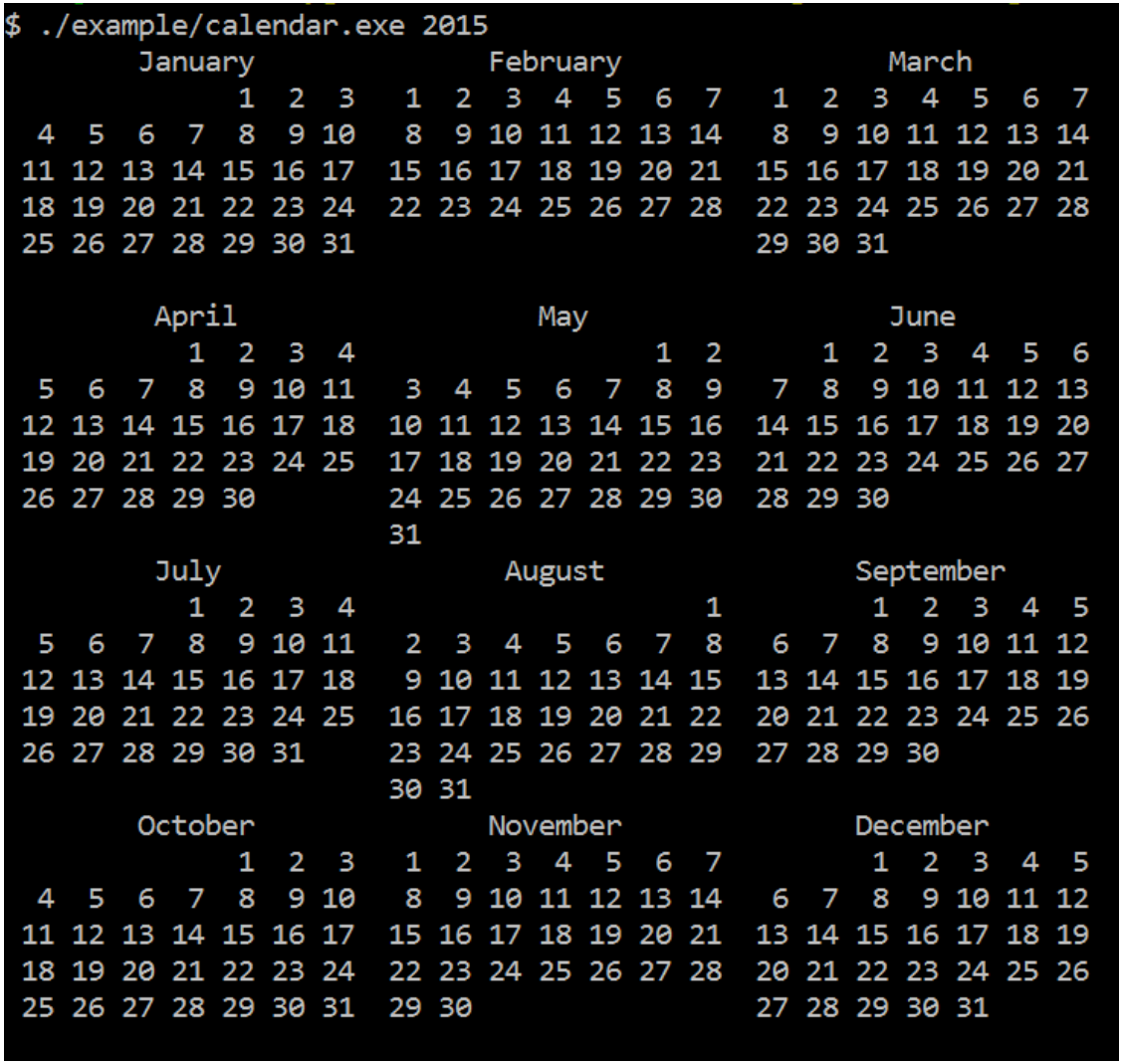
eniebler@boost.org

Welcome to This Talk!

- What are ranges good for?
- What parts make up the whole of ranges?
- How do the parts play together?
- Why should you care?
- What has been proposed for standardization? What *will* be proposed? When will it land?

The idea for this talk was taken from the article "Component programming with ranges" on the D language Wiki.

Goal



```
$ ./example/calendar.exe 2015
       January               February              March
             1  2  3   1  2  3  4  5  6  7   1  2  3  4  5  6  7
 4  5  6  7  8  9 10   8  9 10 11 12 13 14   8  9 10 11 12 13 14
11 12 13 14 15 16 17  15 16 17 18 19 20 21  15 16 17 18 19 20 21
18 19 20 21 22 23 24  22 23 24 25 26 27 28  22 23 24 25 26 27 28
25 26 27 28 29 30 31                       29 30 31

       April                 May                   June
          1  2  3  4                  1  2      1  2  3  4  5  6
 5  6  7  8  9 10 11   3  4  5  6  7  8  9   7  8  9 10 11 12 13
12 13 14 15 16 17 18  10 11 12 13 14 15 16  14 15 16 17 18 19 20
19 20 21 22 23 24 25  17 18 19 20 21 22 23  21 22 23 24 25 26 27
26 27 28 29 30        24 25 26 27 28 29 30  28 29 30
                      31
       July                  August                September
          1  2  3  4                     1         1  2  3  4  5
 5  6  7  8  9 10 11   2  3  4  5  6  7  8   6  7  8  9 10 11 12
12 13 14 15 16 17 18   9 10 11 12 13 14 15  13 14 15 16 17 18 19
19 20 21 22 23 24 25  16 17 18 19 20 21 22  20 21 22 23 24 25 26
26 27 28 29 30 31     23 24 25 26 27 28 29  27 28 29 30
                      30 31
       October               November              December
             1  2  3   1  2  3  4  5  6  7         1  2  3  4  5
 4  5  6  7  8  9 10   8  9 10 11 12 13 14   6  7  8  9 10 11 12
11 12 13 14 15 16 17  15 16 17 18 19 20 21  13 14 15 16 17 18 19
18 19 20 21 22 23 24  22 23 24 25 26 27 28  20 21 22 23 24 25 26
25 26 27 28 29 30 31  29 30                 27 28 29 30 31
```

```cpp
#include <cstddef>
#include <string>
#include <vector>
#include <utility>
#include <iostream>
#include <stdexcept>
#include <functional>
#include <boost/format.hpp>
#include <boost/lexical_cast.hpp>
#include <boost/date_time/gregorian/gregorian.hpp>
#include <range/v3/all.hpp>

namespace greg = boost::gregorian;
using date = greg::date;
using day = greg::date_duration;
using namespace ranges;
using std::cout;

namespace boost { namespace gregorian {
    date &operator++(date &d) { return d = d + day(1); }
    date operator++(date &d, int) { return ++d - day(1); }
}}
namespace ranges {
    template<> struct difference_type<date> {
        using type = date::duration_type::duration_rep::int_type;
    };
}
CONCEPT_ASSERT(Incrementable<date>());

auto dates_in_year(int year) {
    return view::iota(date{year,greg::Jan,1},
                      date{year+1,greg::Jan,1});
}

auto by_month() {
    return view::group_by([](date a, date b) {
        return a.month() == b.month();
    });
}

auto by_week() {
    return view::group_by([](date a, date b) {
        // ++a because week_numer is Mon-Sun and we want Sun-Sat
        return (++a).week_number() == (++b).week_number();
    });
}

std::string format_day(date d) {
    return boost::str(boost::format("%|3|") % d.day());
}

// In:  Range<Range<date>>: month grouped by weeks.
// Out: Range<std::string>: month with formatted weeks.
auto format_weeks() {
    return view::transform([](/*Range<date>*/ auto week) {
        return boost::str(boost::format("%1%%2%%|22t|")
            % std::string((int)front(week).day_of_week() * 3, ' ')
            % (week | view::transform(format_day) | action::join));
    });
}

// Return a formatted string with the title of the month
// corresponding to a date.
std::string month_title(date d) {
    return boost::str(boost::format("%|=22|")
        % d.month().as_long_string());
}

// In:  Range<Range<date>>: year of months of days
// Out: Range<Range<std::string>>: year of months of formatted wks
auto layout_months() {
    return view::transform([](/*Range<date>*/ auto month) {
        int week_count = distance(month | by_week());
        return view::concat(
            view::single(month_title(front(month))),
            month | by_week() | format_weeks(),
            view::repeat_n(std::string(22,' '),6-week_count));
    });
}
```

```cpp
// In:  Range<T>
// Out: Range<Range<T>>, where each inner range has $n$ elements.
//                       The last range may have fewer.
template<class Rng>
class chunk_view : public range_adaptor<chunk_view<Rng>, Rng> {
    CONCEPT_ASSERT(ForwardIterable<Rng>());
    std::size_t n_;
    friend range_access;
    class adaptor;
    adaptor begin_adaptor() {
        return adaptor{n_, ranges::end(this->base())};
    }
public:
    chunk_view() = default;
    chunk_view(Rng rng, std::size_t n)
      : range_adaptor_t<chunk_view>(std::move(rng)), n_(n)
    {}
};

template<class Rng>
class chunk_view<Rng>::adaptor : public adaptor_base {
    std::size_t n_;
    range_sentinel_t<Rng> end_;
    using adaptor_base::prev;
public:
    adaptor() = default;
    adaptor(std::size_t n, range_sentinel_t<Rng> end)
      : n_(n), end_(end)
    {}
    auto current(range_iterator_t<Rng> it) const {
        return view::take(make_range(std::move(it), end_), n_);
    }
    void next(range_iterator_t<Rng> &it) {
        ranges::advance(it, n_, end_);
    }
};

// In:  Range<T>
// Out: Range<Range<T>>, where each inner range has $n$ elements.
//                       The last range may have fewer.
auto chunk(std::size_t n) {
    return make_pipeable([=](auto&& rng) {
        using Rng = decltype(rng);
        return chunk_view<view::all_t<Rng>>{
            view::all(std::forward<Rng>(rng)), n};
    });
}

// Flattens a range of ranges by iterating the inner
// ranges in round-robin fashion.
template<class Rngs>
class interleave_view : public range_facade<interleave_view<Rngs>> {
    friend range_access;
    std::vector<range_value_t<Rngs>> rngs_;
    struct cursor;
    cursor begin_cursor() {
        return {0, &rngs_, view::transform(rngs_, ranges::begin)};
    }
public:
    interleave_view() = default;
    explicit interleave_view(Rngs rngs)
      : rngs_(std::move(rngs))
    {}
};

template<class Rngs>
struct interleave_view<Rngs>::cursor {
    std::size_t n_;
    std::vector<range_value_t<Rngs>> *rngs_;
    std::vector<range_iterator_t<range_value_t<Rngs>>> its_;
    decltype(auto) current() const {
        return *its_[n_];
    }
    void next() {
        if(0 == ((++n_) %= its_.size()))
            for_each(its_, [](auto& it){ ++it; });
    }
    bool done() const {
        return n_ == 0 && its_.end() != mismatch(its_,
            view::transform(*rngs_, ranges::end),
std::not_equal_to<>()).first;
    }
```

```cpp
CONCEPT_REQUIRES(ForwardIterable<range_value_t<Rngs>>())
    bool equal(cursor const& that) const {
        return n_ == that.n_ && its_ == that.its_;
    }
};

// In:  Range<Range<T>>
// Out: Range<T>, flattened by walking the ranges
//                round-robin fashion.
auto interleave() {
    return make_pipeable([](auto&& rngs) {
        using Rngs = decltype(rngs);
        return interleave_view<view::all_t<Rngs>>(
            view::all(std::forward<Rngs>(rngs)));
    });
}

// In:  Range<Range<T>>
// Out: Range<Range<T>>, transposing the rows and columns.
auto transpose() {
    return make_pipeable([](auto&& rngs) {
        using Rngs = decltype(rngs);
        CONCEPT_ASSERT(ForwardIterable<Rngs>());
        return std::forward<Rngs>(rngs)
            | interleave()
            | chunk(distance(rngs));
    });
}

// In:  Range<Range<Range<string>>>
// Out: Range<Range<Range<string>>>, transposing months.
auto transpose_months() {
    return view::transform([](/*Range<Range<string>>*/ auto rng) {
        return rng | transpose();
    });
}

// In:  Range<Range<string>>
// Out: Range<string>, joining the strings of the inner ranges
auto join_months() {
    return view::transform([](/*Range<string>*/ auto rng) {
        return action::join(rng);
    });
}

int main(int argc, char *argv[]) try {
    if(argc < 2) {
        std::cerr << "Please enter the year to format.\n";
        std::cerr << boost::format("  Usage: %1% <year>\n") % argv[0];
        return 1;
    }

    int year = boost::lexical_cast<int>(argv[1]);
    int months_per_line = 3;

    auto calendar =
        // Make a range of all the dates in a year:
        dates_in_year(year)
            // Group the dates by month:
          | by_month()
            // Format the month into a range of strings:
          | layout_months()
            // Group the months that belong side-by-side:
          | chunk(months_per_line)
            // Transpose the rows and columns of the size-by-side months:
          | transpose_months()
            // Ungroup the side-by-side months:
          | view::join
            // Join the strings of the transposed months:
          | join_months();

    // Write the result to stdout:
    copy(calendar, ostream_iterator<>(std::cout, "\n"));
}
catch(std::exception &e) {
    std::cerr << "ERROR: Unhandled exception\n";
    std::cerr << "  what(): " << e.what();
    return 1;
}
```

Step 1

Create a range of dates.

Hello, Date_time!

```cpp
#include <iostream>
#include <boost/date_time/gregorian/gregorian.hpp>

namespace greg = boost::gregorian;
using date = greg::date;
using day = greg::date_duration;

int main()
{
    date fluxx (1955, greg::Nov, 5);
    std::cout << "Great Scott! " << fluxx << "\n";
}
```

```
eric@ERIC-THINK /cygdrive/c/Users/eric/Code/range-build-clang
$ ./example/calendar.exe
Great Scott! 1955-Nov-05
```

Hello, Range!

```cpp
#include <iostream>
#include <range/v3/all.hpp>

using namespace ranges;

int main()
{
    std::cout << view::iota(1,11) << "\n";
}
```

```
eric@ERIC-THINK /cygdrive/c/Users/eric/Code/range-build-clang
$ ./example/calendar.exe
[1,2,3,4,5,6,7,8,9,10]
```

Range-v3: https://github.com/ericniebler/range-v3

Range Views

- Begin/end members return iterator/sentinel
- Lazy sequence algorithms
- Lightweight, non-owning
- Composable
- Non-mutating

Range of dates = ☹

```
int main()
{
    date from(2015,greg::Jan,1);
    date to(2016,greg::Jan,1);

    view::iota(from,to);
}
```

Range of dates = ☹

```
int main()
{
```

```
eric@ERIC-THINK /cygdrive/c/Users/eric/Code/range-build-clang
$ make calendar 2>&1 | fold -w 90 -s
Scanning dependencies of target calendar
[100%] Building CXX object example/CMakeFiles/calendar.dir/calendar.cpp.o
In file included from /cygdrive/c/Users/eric/Code/range-v3/example/calendar.cpp:49:
/cygdrive/c/Users/eric/Code/range-v3/include/range/v3/view/iota.hpp:303:21: error:
static_assert failed "The object passed to view::iota must model the WeaklyIncrementable
concept; that is, it must have pre- and post-increment operators and it must have a
difference_type"
                    CONCEPT_ASSERT_MSG(WeaklyIncrementable<Val>(),
                    ^                  ~~~~~~~~~~~~~~~~~~~~~~~~~~
/cygdrive/c/Users/eric/Code/range-v3/include/range/v3/utility/concepts.hpp:744:28: note:
expanded from macro 'CONCEPT_ASSERT_MSG'
#define CONCEPT_ASSERT_MSG static_assert
                           ^
/cygdrive/c/Users/eric/Code/range-v3/example/calendar.cpp:61:15: note: in instantiation
of function template specialization
'ranges::v3::view::iota_fn::operator()<boost::gregorian::date, boost::gregorian::date,
42, 0>' requested here
    view::iota(from,to);
              ^
1 error generated.
```

Range of dates = HACKHACK

```
namespace boost { namespace gregorian {
    date &operator++(date &d) { return d = d + day(1); }
    date operator++(date &d, int) { return ++d - day(1); }
}}
namespace ranges {
    template<> struct difference_type<date> {
        using type = date::duration_type::duration_rep::int_type;
    };
}
CONCEPT_ASSERT(Incrementable<date>());

int main() {
    date from(2015,greg::Jan,1);
    date to(2016,greg::Jan,1);

    RANGES_FOR(auto d, view::iota(from,to) | view::take(10))
        std::cout << d << '\n';
}
```

Range of dates = HACKHACK

```
namespace boost { namespace gregorian {
    date {          ++(date &d) { return d = d + day(1); }
                    .r(date &d, int) { return ++d - day(1); }

namespace ranges {
    template<> struct difference_type<date> {
        using type = date::duration_type::duration_rep::int_type;
    };
```

Don't do this.

```
eric@ERIC-THINK /cygdrive/c/Users/eric/Code/range-build-clang
$ ./example/calendar.exe
2015-Jan-01
2015-Jan-02
2015-Jan-03
2015-Jan-04
2015-Jan-05
2015-Jan-06
2015-Jan-07
2015-Jan-08
2015-Jan-09
2015-Jan-10
```

Step 2

Group the range of dates into months.

Group Dates into Months

```
auto dates_in_year(int year) {
    return view::iota(date{year,greg::Jan,1},
                      date{year+1,greg::Jan,1});
}

int main() {
    auto year = dates_in_year(2015);
    // Group into months:
    auto months = year | view::group_by([](date a, date b) {
        return a.month() == b.month();
    });

    // Print the first day of each month:
    RANGES_FOR(auto month, months)
        cout << front(month) << '\n';
}
```

Group Dates into Months

```
auto dates_in_year(int year) {
    return view::iota(date{year,greg::Jan,1},
                      date{year+1,greg::Jan,1});
}

int main() {
    auto year = dates_in_year(2015);
    // Group into months:
    auto months = year | view::group_by([](d
        return a.month() == b.month();
    });

    // Print the first day of each month:
    RANGES_FOR(auto month, months)
        cout << front(month) << '\n';
}
```

```
$ ./example/calendar.exe
2015-Jan-01
2015-Feb-01
2015-Mar-01
2015-Apr-01
2015-May-01
2015-Jun-01
2015-Jul-01
2015-Aug-01
2015-Sep-01
2015-Oct-01
2015-Nov-01
2015-Dec-01
```

Refactor for Readability

```
auto by_month() {
    return view::group_by([](date a, date b) {
        return a.month() == b.month();
    });
}

int main() {
    auto year = dates_in_year(2015);

    // Print the first day of each month:
    RANGES_FOR(auto month, year | by_month())
        std::cout << front(month) << '\n';
}
```

Move the group_by expression into its own named adaptor.

Built-in Range Views

adjacent_remove_if	drop_while	map	split
all	empty	move	stride
any_range	filter	partial_sum	tail
bounded	for_each	remove_if	take
c_str	generate	repeat	take_exactly
chunk	generate_n	repeat_n	take_while
concat	group_by	replace	tokenize
const_	indirect	replace_if	transform
counted	intersperse	reverse	unbounded
delimit	iota	single	unique
drop	join	slice	zip[_with]

Step 3

Group months into weeks.

Group Months into Weeks

```
auto by_week() {
    return view::group_by([](date a, date b) {
        // ++a because week_numer is Mon-Sun and we want Sun-Sat
        return (++a).week_number() == (++b).week_number();
    });
}

auto month_by_week() {
    return view::transform([](auto month) {
        return month | by_week();
    });
}

int main() {
    RANGES_FOR(auto month, dates_in_year(2015) | by_month() | month_by_week()) {
        RANGES_FOR(auto week, month)
            cout << view::transform(week, &date::day) << '\n';
        cout << "----\n";
    }
}
```

Group Months into Weeks

```
auto by_week() {
    return view::group_by([](date a, date b) {
        // ++a because week_numer is Mon-Sun and we want Sun-Sat
        return (++a).week_number() == (++b).week_number();
    });
}

auto month_by_week() {
    return view::transform([](auto month) {
        return month | by_week();
    });
}

int main() {
    RANGES_FOR(auto month, dates_in_year(2015) | by                    ) {
        RANGES_FOR(auto week, month)
            cout << view::transform(week, &date::day) << '\n';
        cout << "----\n";
    }
}
```

```
$ ./example/calendar.exe
[1,2,3]
[4,5,6,7,8,9,10]
[11,12,13,14,15,16,17]
[18,19,20,21,22,23,24]
[25,26,27,28,29,30,31]
----
[1,2,3,4,5,6,7]
[8,9,10,11,12,13,14]
[15,16,17,18,19,20,21]
[22,23,24,25,26,27,28]
----
[1,2,3,4,5,6,7]
```

Step 4

Format the weeks

Format the Weeks

```
std::string format_day(date d) {
    return boost::str(boost::format("%|3|") % d.day());
}
```

```
// In:  Range<Range<date>>: month grouped by weeks.
// Out: Range<std::string>: month with formatted weeks.
auto format_weeks() {
    return view::transform([](/*Range<date>*/ auto week) {
        return boost::str(boost::format("%1%%2%%|22t|")
            % std::string((int)front(week).day_of_week() * 3, ' ')
            % (week | view::transform(format_day) | action::join));
    });
}
```

```
// In:  Range<Range<date>>: year of months of days
// Out: Range<Range<std::string>>: year of months of formatted wks
auto layout_months() {
    return view::transform([](/*Range<date>*/ auto month) {
        return month | by_week() | format_weeks();
    });
}
```

Range Actions

- Eager sequence algorithms
- Can operate on and return containers
- Composable
- Potentially mutating

Views vs. Actions

Range Views	Range Actions
Lazy sequence algorithms	Eager sequence algorithms
Lightweight, non-owning	Can operate on and return containers
Composable	Composable
Non-mutating	Potentially mutating

Built-in Range Action

drop	push_front	stride
drop_while	remove_if	take
erase	slice	take_while
insert	sort	transform
join	split	unique
push_back	stable_sort	

So Far, So Good

```
int main() {
    auto year =
        dates_in_year(2015)
          | by_month()
          | layout_months();

    RANGES_FOR(auto month, year)
    {
        RANGES_FOR(std::string week, month)
            cout << week << '\n';
        cout << "----\n";
    }
}
```

```
$ ./example/calendar.exe
                1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
----
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
----
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31
----
           1  2  3  4
```

Step 5

Add month title and padded weeks.

Month Title

```
// Return a formatted string with the title of the month
// corresponding to a date.
std::string month_title(date d) {
    return boost::str(boost::format("%|=22|")
        % d.month().as_long_string());
}

// In:  Range<Range<date>>: year of months of days
// Out: Range<Range<std::string>>: year of months of formatted wks
auto layout_months() {
    return view::transform([](/*Range<date>*/ auto month) {
        return view::concat(
            view::single(month_title(front(month))),
            month | by_week() | format_weeks());
    });
}
```

view::concat lazily concatenates ranges.

view::single creates a 1-element range.

Month Title

```
// Return a formatted string with the title of the month
// corresponding to a date.
std::string month_title(date d) {
    return boost::str(boost::format("%|=22|")
        % d.month().as_long_string());
}

// In:  Range<Range<date>>: year of months of days
// Out: Range<Range<std::string>>: year of months
auto layout_months() {
    return view::transform([](/*Range<date>*/ auto
        return view::concat(
            view::single(month_title(front(month))
            month | by_week() | format_weeks());
    });
}
```

```
$ ./example/calendar.exe
        January
                1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
----
       February
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
----
         March
 1  2  3  4  5  6  7
```

Padding Short Months

A formatted month takes as few as four and as many as six lines.

For side-by-side display of months, they must all occupy the same vertical space.

Pad the short months with empty lines.

Padding Short Months

```
// In:  Range<Range<date>>: year of months of days
// Out: Range<Range<std::string>>: year of months of formatted wks
auto layout_months() {
    return view::transform([](/*Range<date>*/ auto month) {
        int week_count = distance(month | by_week());
        return view::concat(
            view::single(month_title(front(month))),
            month | by_week() | format_weeks(),
            view::repeat_n(std::string(22,' '),6-week_count));
    });
}
```

`view::repeat_n` creates an *N*-element range.

Padding Short Months

```
// In:  Range<Range<date>>: year of months of days
// Out: Range<Range<std::string>>: year of months of
auto layout_months() {
    return view::transform([](/*Range<date>*/ auto mo
        int week_count = distance(month | by_week());
        return view::concat(
            view::single(month_title(front(month))),
            month | by_week() | format_weeks(),
            view::repeat_n(std::string(22,' '),6-week
    });
}
```

```
$ ./example/calendar.exe
       January
                1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31

----
      February
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28


----
       March
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31

----
```

So Far, So Good

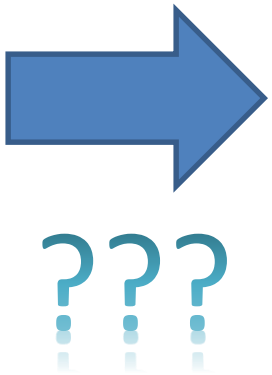
```
int main() {
    auto year =
        dates_in_year(2015)
          | by_month()
          | layout_months();

    RANGES_FOR(auto month, year)
    {
        RANGES_FOR(std::string week, month)
            cout << week << '\n';
        cout << "----\n";
    }
}
```

A “year” is a range of “months”.
A “month” is a range of strings.
Each “month” has exactly 7 lines.

`by_month()` and `layout_months()` are reusable, and work even if the input range of dates is infinite!

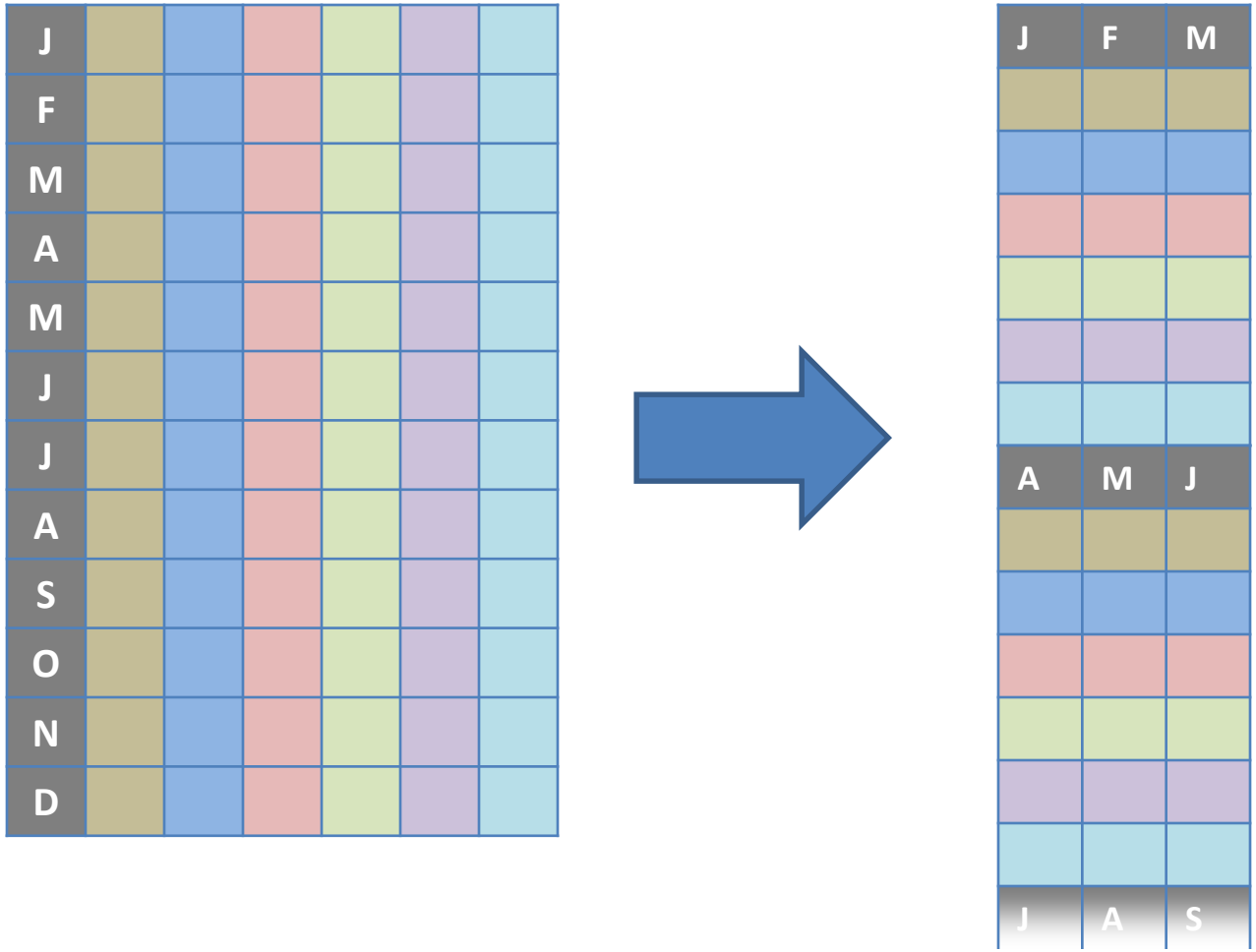
Side-by-Side Month Layout

```
$ ./example/calendar.exe
       January
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31

----
      February
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28


----
       March
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31

----
```




```
$ ./example/calendar.exe 2015
       January               February              March
             1  2  3   1  2  3  4  5  6  7   1  2  3  4  5  6  7
 4  5  6  7  8  9 10   8  9 10 11 12 13 14   8  9 10 11 12 13 14
11 12 13 14 15 16 17  15 16 17 18 19 20 21  15 16 17 18 19 20 21
18 19 20 21 22 23 24  22 23 24 25 26 27 28  22 23 24 25 26 27 28
25 26 27 28 29 30 31                       29 30 31

       April                 May                   June
          1  2  3  4                  1  2      1  2  3  4  5  6
 5  6  7  8  9 10 11   3  4  5  6  7  8  9   7  8  9 10 11 12 13
12 13 14 15 16 17 18  10 11 12 13 14 15 16  14 15 16 17 18 19 20
19 20 21 22 23 24 25  17 18 19 20 21 22 23  21 22 23 24 25 26 27
26 27 28 29 30        24 25 26 27 28 29 30  28 29 30
                      31
       July                  August                September
          1  2  3  4                     1         1  2  3  4  5
 5  6  7  8  9 10 11   2  3  4  5  6  7  8   6  7  8  9 10 11 12
12 13 14 15 16 17 18   9 10 11 12 13 14 15  13 14 15 16 17 18 19
19 20 21 22 23 24 25  16 17 18 19 20 21 22  20 21 22 23 24 25 26
26 27 28 29 30 31     23 24 25 26 27 28 29  27 28 29 30
                      30 31
       October               November              December
             1  2  3   1  2  3  4  5  6  7      1  2  3  4  5
 4  5  6  7  8  9 10   8  9 10 11 12 13 14   6  7  8  9 10 11 12
11 12 13 14 15 16 17  15 16 17 18 19 20 21  13 14 15 16 17 18 19
18 19 20 21 22 23 24  22 23 24 25 26 27 28  20 21 22 23 24 25 26
25 26 27 28 29 30 31  29 30                 27 28 29 30 31
```

Side-by-Side Month Layout



Side-by-Side Month Layout

1. Chunk months into groups of 3's.
2. For each group of 3 months, *transpose* the "rows" and "columns".
3. Join the chunks created in step 1.
4. Join the strings of the inner ranges.
5. Print!
6. Take the rest of the day off.

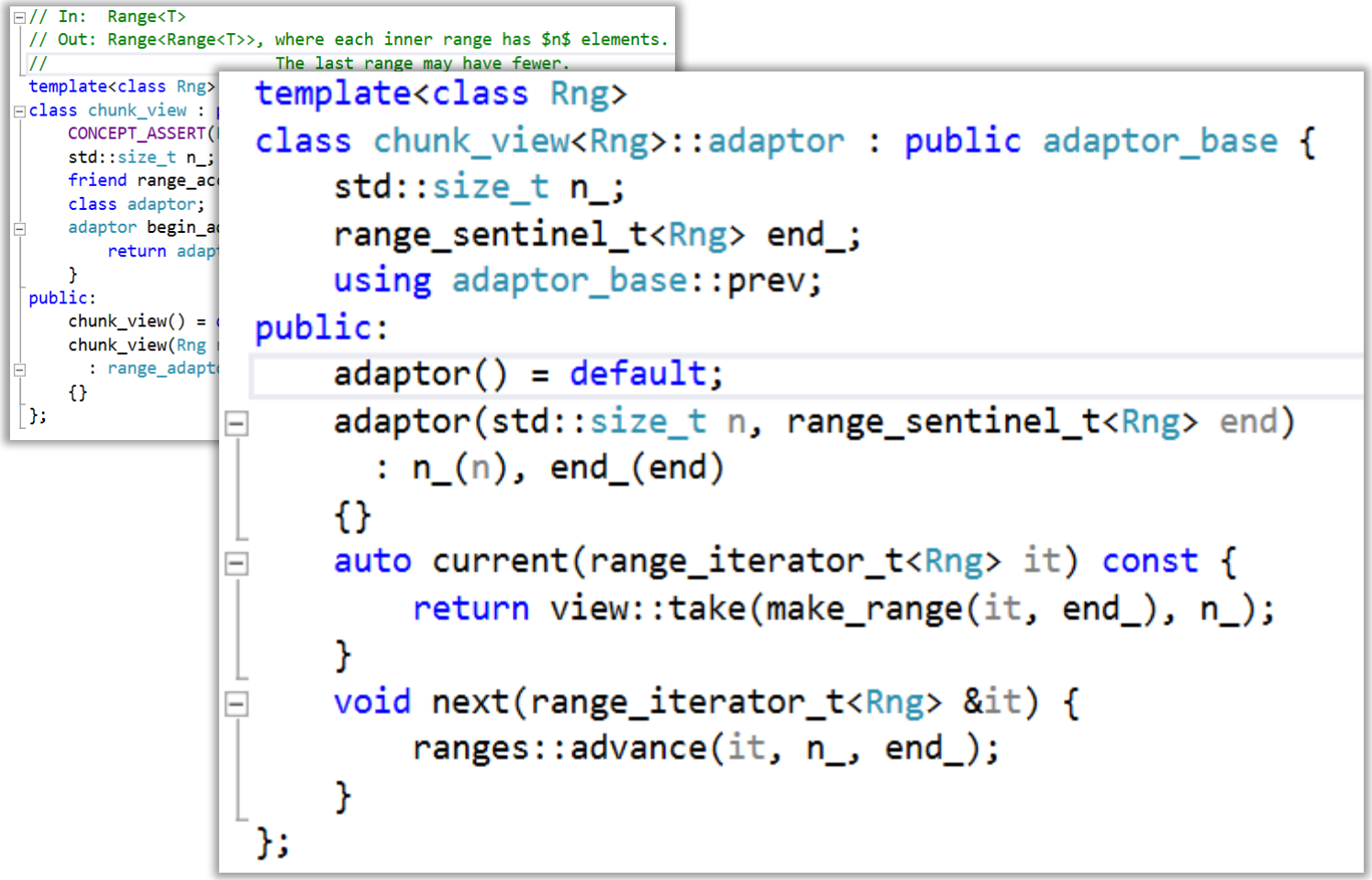
Chunking: Custom Range Adaptor

```
// In:  Range<T>
// Out: Range<Range<T>>, where each inner range has $n$ elements.
//                       The last range may have fewer.
template<class Rng>
class chunk_view : public range_adaptor<chunk_view<Rng>, Rng> {
    CONCEPT_ASSERT(ForwardIterable<Rng>());
    std::size_t n_;
    friend range_access;
    class adaptor;
    adaptor begin_adaptor() {
        return adaptor{n_, ranges::end(this->base())};
    }
public:
    chunk_view() = default;
    chunk_view(Rng rng, std::size_t n)
      : range_adaptor_t<chunk_view>(std::move(rng)), n_(n)
    {}
};
```

Chunking: Custom Range Adaptor

```
// In:  Range<T>
//
//
template<class Rng>
class chunk_view<Rng>::adaptor : public adaptor_base {
    std::size_t n_;
    range_sentinel_t<Rng> end_;
    using adaptor_base::prev;
public:
    adaptor() = default;
    adaptor(std::size_t n, range_sentinel_t<Rng> end)
      : n_(n), end_(end)
    {}
    auto current(range_iterator_t<Rng> it) const {
        return view::take(make_range(it, end_), n_);
    }
    void next(range_iterator_t<Rng> &it) {
        ranges::advance(it, n_, end_);
    }
};
```

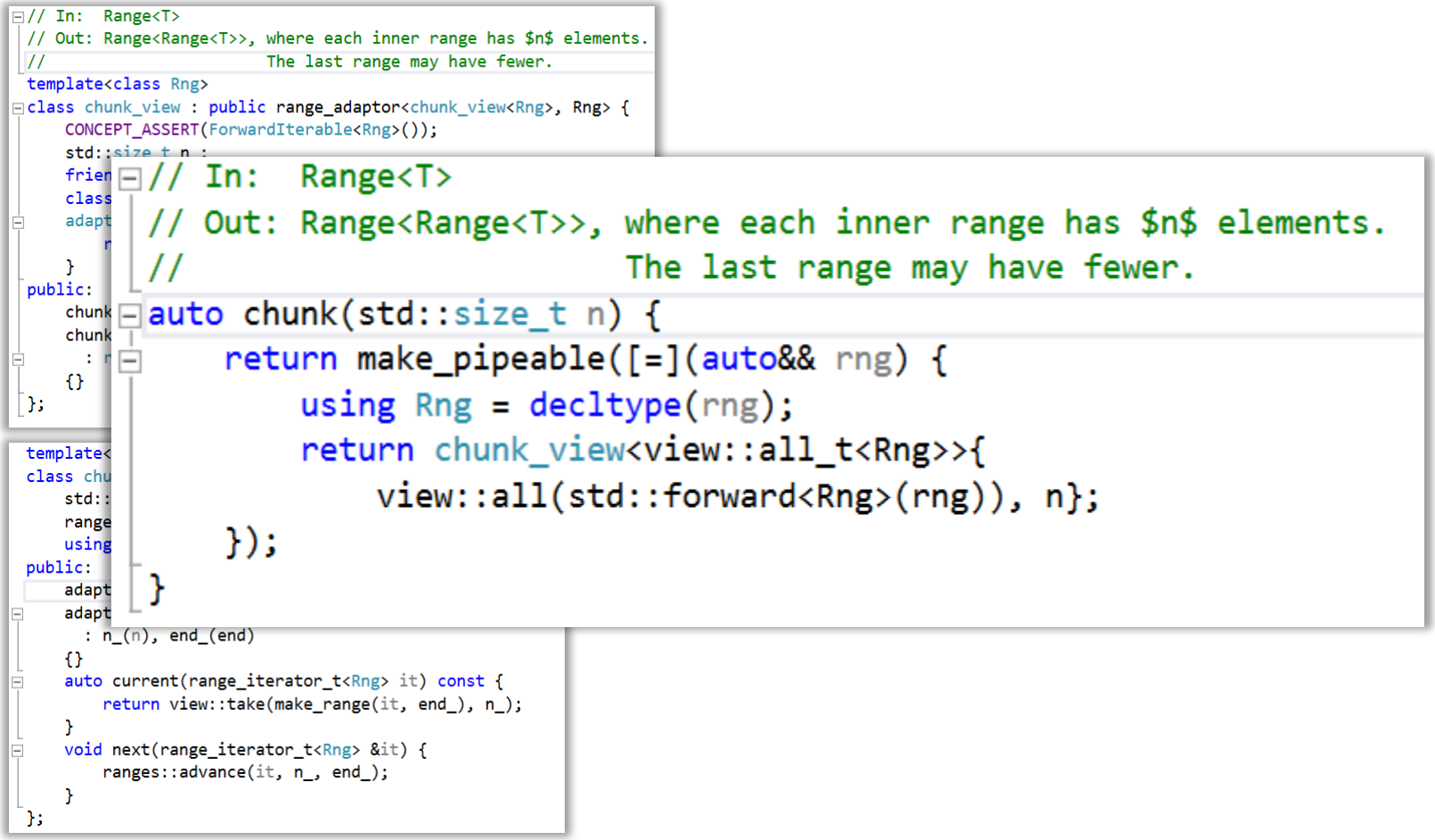
Chunking: Custom Range Adaptor



```
// In:  Range<T>
// Out: Range<Range<T>>, where each inner range has $n$ elements.
//                       The last range may have fewer.
template<class Rng>
class chunk_view : p
    CONCEPT_ASSERT(
    std::size_t n_;
    friend range_ac
    class adaptor;
    adaptor begin_a
        return adapt
    }
public:
    chunk_view() = d
    chunk_view(Rng
      : range_adapt
    {}
};
```

```
template<class Rng>
class chunk_view<Rng>::adaptor : public adaptor_base {
    std::size_t n_;
    range_sentinel_t<Rng> end_;
    using adaptor_base::prev;
public:
    adaptor() = default;
    adaptor(std::size_t n, range_sentinel_t<Rng> end)
      : n_(n), end_(end)
    {}
    auto current(range_iterator_t<Rng> it) const {
        return view::take(make_range(it, end_), n_);
    }
    void next(range_iterator_t<Rng> &it) {
        ranges::advance(it, n_, end_);
    }
};
```

Chunking: Custom Range Adaptor



```
// In:  Range<T>
// Out: Range<Range<T>>, where each inner range has $n$ elements.
//                       The last range may have fewer.
auto chunk(std::size_t n) {
    return make_pipeable([=](auto&& rng) {
        using Rng = decltype(rng);
        return chunk_view<view::all_t<Rng>>{
            view::all(std::forward<Rng>(rng)), n};
    });
}
```

Chunking: Custom Range Adaptor

```
// In:  Range<T>
// Out: Range<Range<T>>, where each inner range has $n$ elements.
//                       The last range may have fewer.
template<class Rng>
class chunk_view : public range_adaptor<chunk_view<Rng>, Rng> {
    CONCEPT_ASSERT(ForwardIterable<Rng>());
    std::size_t n_;
    friend range_access;
    class adaptor;
    adaptor begin_adaptor() {
        return adaptor{n_, ranges::end(this->base())};
    }
public:
    chunk_view() = default;
    chunk_view(Rng rng, std::size_t n)
      : range_adaptor_t<chunk_view>(std::move(rng)),
    {}
};
```

```
// In:  Range<T>
// Out: Range<Range<T>>, where each inner range has $n$ elements.
//                       The last range may have fewer.
auto chunk(std::size_t n) {
    return make_pipeable([=](auto&& rng) {
        using Rng = decltype(rng);
        return chunk_view<view::all_t<Rng>>{
            view::all(std::forward<Rng>(rng)), n};
    });
}
```

```
int main() {
    std::vector<int> v{0,1,2,3,4,5,6,7,8,9};
    RANGES_FOR(auto chunk, v | chunk(3))
        cout << chunk << '\n';
}
```

```
template<class Rng>
class chunk_view<Rng>::adaptor : public adaptor_base
    std::size_t n_;
    range_sentinel_t<Rng> end_;
    using adaptor_base::prev;
public:
    adaptor() = default;
    adaptor(std::size_t n, range_sentinel_t<Rng> end)
      : n_(n), end_(end)
    {}
    auto current(range_iterator_t<Rng> it) const {
        return view::take(make_range(it, end_), n_);
    }
    void next(range_iterator_t<Rng> &it) {
        ranges::advance(it, n_, end_);
    }
};
```

```
$ ./example/calendar.exe
[0,1,2]
[3,4,5]
[6,7,8]
[9]
```

Transpose Range of Ranges

Transpose Range of Ranges

January	Jan Wk 1	Jan Wk 2	Jan Wk 3	Jan Wk 4	Jan Wk 5	Jan Wk 6
February	Feb Wk 1	Feb Wk 2	Feb Wk 3	Feb Wk 4	Feb Wk 5	Feb Wk 6
March	Mar Wk 1	Mar Wk 2	Mar Wk 3	Mar Wk 4	Mar Wk 5	Mar Wk 6

1. Interleave

January
February
March
Jan Wk 1
Feb Wk 1
Mar Wk 1
Jan Wk 2
…

2. Chunk

January	**February**	**March**
Jan Wk 1	Feb Wk 1	Mar Wk 1
Jan Wk 2	Feb Wk 2	Mar Wk 2
Jan Wk 3	Feb Wk 3	Mar Wk 3
Jan Wk 4	Feb Wk 4	Mar Wk 4
Jan Wk 5	Feb Wk 5	Mar Wk 5
Jan Wk 6	Feb Wk 6	Mar Wk 6

Interleave: Custom Range Facade

```
// Flattens a range of ranges by iterating the inner
// ranges in round-robin fashion.
template<class Rngs>
class interleave_view : public range_facade<interleave_view<Rngs>> {
    friend range_access;
    std::vector<range_value_t<Rngs>> rngs_;
    struct cursor;
    cursor begin_cursor() {
        return {0, &rngs_, view::transform(rngs_, ranges::begin)};
    }
public:
    interleave_view() = default;
    explicit interleave_view(Rngs rngs)
      : rngs_(std::move(rngs))
    {}
};
```

Interleave: Custom Range Facade

```
template<class Rngs>
struct interleave_view<Rngs>::cursor  {
    std::size_t n_;
    std::vector<range_value_t<Rngs>> *rngs_;
    std::vector<range_iterator_t<range_value_t<Rngs>>> its_;
    decltype(auto) current() const {
        return *its_[n_];
    }
    void next() {
        if(0 == ((++n_) %= its_.size()))
            for_each(its_, [](auto& it){ ++it; });
    }
    bool done() const {
        return n_ == 0 && its_.end() != mismatch(its_,
            view::transform(*rngs_, ranges::end), std::not_equal_to<>()).first;
    }
    CONCEPT_REQUIRES(ForwardIterable<range_value_t<Rngs>>())
    bool equal(cursor const& that) const {
        return n_ == that.n_ && its_ == that.its_;
    }
};
```

Interleave: Custom Range Facade

```
template<class Rngs>
struct interleave_view<Rngs>::cursor  {
    std::size_t n_;
    std::vector<range_value_t<Rngs>> *rngs_;
    std::vec
    decltype(
        retu
    }
    void nex
        if(0
            +
    }
    bool don
        retu
            view::transform(*rngs_, ranges::end), std::not_equal_to<>()).first;
    }
    CONCEPT_REQUIRES(ForwardIterable<range_value_t<Rngs>>())
    bool equal(cursor const& that) const {
        return n_ == that.n_ && its_ == that.its_;
    }
};
```

```
// In:  Range<Range<T>>
// Out: Range<T>, flattened by walking the ranges
//                round-robin fashion.
auto interleave() {
    return make_pipeable([](auto&& rngs) {
        using Rngs = decltype(rngs);
        return interleave_view<view::all_t<Rngs>>(
            view::all(std::forward<Rngs>(rngs)));
    });
}
```

Interleave: Custom Range Facade

```
// Flattens a range of ranges by iterating the inner
// ranges in round-robin fashion.
template<class Rngs>
class interleave_view : public range_facade<interleave_view<Rngs>> {
    friend range_access;
    std::vector<range
    struct cursor;
    cursor begin_curs
        return {0, &r
    }
public:
    interleave_view()
    explicit interlea
      : rngs_(std::mo
    {}
};
```

```
// In:  Range<Range<T>>
// Out: Range<T>, flattened by walking the ranges
//                round-robin fashion.
auto interleave() {
    return make_pipeable([](auto&& rngs) {
        using Rngs = decltype(rngs);
        return interleave_view<view::all_t<Rngs>>(
            view::all(std::forward<Rngs>(rngs)));
    });
}
```

```
template<class Rngs>
struct interleave_view<Rng
    std::size_t n_;
    std::vector<range_valu
    std::vector<range_iter
    decltype(auto) current
        return *its_[n_];
    }
    void next() {
        if(0 == ((++n_) %= its_.size()))
            for_each(its_, [](auto& it){ ++it; });
    }
    bool done() const {
        return n_ == 0 && its_.end() != mismatch(its_,
            view::transform(*rngs_, ranges::end), std::not_equal_to<>()).first;
    }
    CONCEPT_REQUIRES(ForwardIterable<range_value_t<Rngs>>())
    bool equal(cursor const& that) const {
        return n_ == that.n_ && its_ == that.its_;
    }
};
```

Interleave: Custom Range Facade

```
// Flattens a range of ranges by iterating the inner
// ranges in round-robin fashion.
template<class Rngs>
class interleave_view : public range_facade<interleave_view<Rngs>> {
    friend range_access;
    std::vector<range_value_t<Rngs>> rngs_;
    struct cursor;
    cursor begin_cursor() {
        return {0, &rngs_, view::transform(rngs_, ranges::begin)};
    }
public:
    interleave_view() = default;
    explicit interleave_view(
      : rngs_(std::move(rngs
    {}
};
```

```
// In:  Range<Range<T>>
// Out: Range<T>, flattened by walking the ranges
//                round-robin fashion.
auto interleave() {
    return make_pipeable([](auto&& rngs) {
        using Rngs = decltype(rngs);
        return interleave_view<view::all_t<Rngs>>(
            view::all(std::forward<Rngs>(rngs)));
    });
}
```

```
int main() {
    auto rng = view::repeat_n(view::iota(0,3), 3);

    cout << rng << '\n';
    cout << (rng | interleave()) << '\n';
}
```

```
template<class Rngs>
struct interleave_view<Rngs>::curs
    std::size_t n_;
    std::vector<range_value_t<Rngs
    std::vector<range_iterator_t<r
    decltype(auto) current() const
        return *its_[n_];
    }
    void next() {
        if(0 == ((++n_) %= its_.size()))
            for_each(its_, [](auto& it){ ++it; });
    }
    bool done() const {
        return n_ == 0 && its_.end() != mismatch(its_,
            view::transform(*rngs_, ranges::end), std::not_equal_to<>()).first;
    }
    CONCEPT_REQUIRES(ForwardIterable<range_value_t<Rngs>>())
    bool equal(cursor const& that) const {
        return n_ == that.n_ && its_ == that.its_;
    }
};
```

```
$ ./example/calendar.exe
[[0,1,2],[0,1,2],[0,1,2]]
[0,0,0,1,1,1,2,2,2]
```

Transpose Range of Ranges

```
// In:  Range<Range<T>>
// Out: Range<Range<T>>, transposing the rows and columns.
auto transpose() {
    return make_pipeable([](auto&& rngs) {
        using Rngs = decltype(rngs);
        CONCEPT_ASSERT(ForwardIterable<Rngs>());
        return std::forward<Rngs>(rngs)
            | interleave()
            | chunk(distance(rngs));
    });
}
```

```
int main() {
    auto rng = view::repeat_n(view::iota(0,3), 3);

    cout << rng << '\n';
    cout << (rng | transpose()) << '\n';
}
```

```
$ ./example/calendar.exe
[[0,1,2],[0,1,2],[0,1,2]]
[[0,0,0],[1,1,1],[2,2,2]]
```

Side-by-Side Month Layout

1. Chunk months into groups of 3's.
2. For each group of 3 months, *transpose* the "rows" and "columns".
3. Join the chunks created in step 1
4. Join the strings of the inner ranges.
5. Print!
6. Take the rest of the day off.

Solution

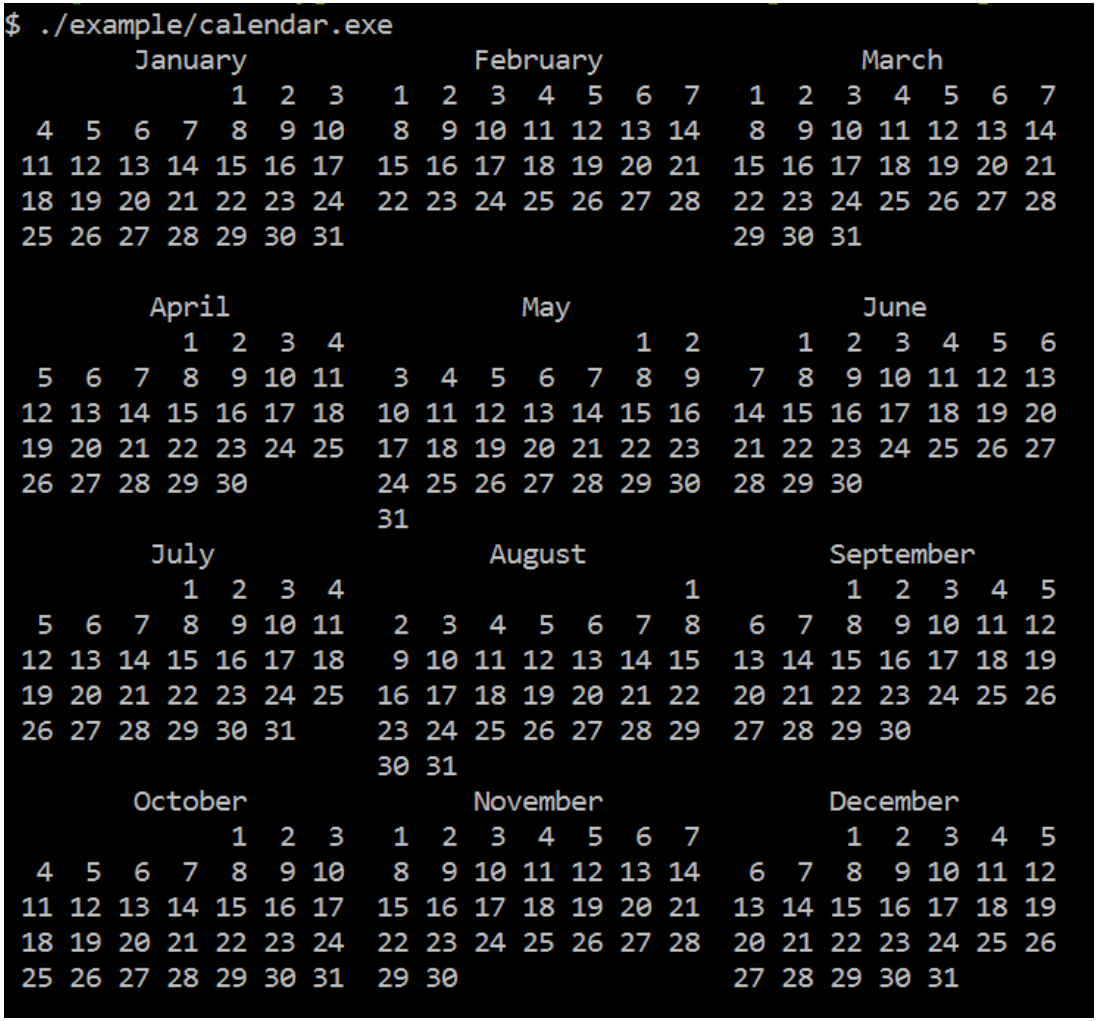
```
int main() {
    copy(
        dates_in_year(2015)     // 0. Make a range of dates.
          | by_month()          // 1. Group the dates by month.
          | layout_months()     // 2. Format the month into a range of
                                //    strings.
          | chunk(3)            // 3. Group the months that belong
                                //    side-by-side.
          | transpose_months()  // 4. Transpose the rows and columns
                                //    of the size-by-side months.
          | view::join          // 6. Ungroup the side-by-side months.
          | join_months(),      // 7. Join the strings of the transposed
                                //    months.
        ostream_iterator<>(std::cout, "\n")
    );
}
```

Solution

```
int main() {
    copy(
        dates_in_year(2015)     // 0. Make a range of dates.
          | by_month()          // 1. Group the dates by month.
          | layout_months()     // 2. Format the month into a range of
                                //    strings.
          | chunk(3)            // 3. Group the months that belong
                                //    side-by-side.
```

```
auto transpose_months() {
    return view::transform([](/*Range<Range<string>>*/ auto rng) {
        return rng | transpose();
    });
}

auto join_months() {
    return view::transform([](/*Range<string>*/ auto rng) {
        return action::join(rng);
    });
}
```

Ta-da!



Calendar Solution

```
int main() {
    copy(
        dates_in_year(2015)      // 0. Make a range
          | by_month()           // 1. Group the dates by month.
          | layout_months()      // 2. Format the mo
                                 //    strings.
          | chunk(3)             // 3. Group the mon
                                 //    side-by-side.
          | transpose_months()   // 4. Transpose the
                                 //    of the size-b
          | view::join           // 6. Ungroup the s
          | join_months(),       // 7. Join the strings of the transposed
                                 //    months.
        ostream_iterator<>(std::cout, "\n")
    );
}
```

Composable

Reusable

Works with infinite ranges

Can show *N* months side-by-side

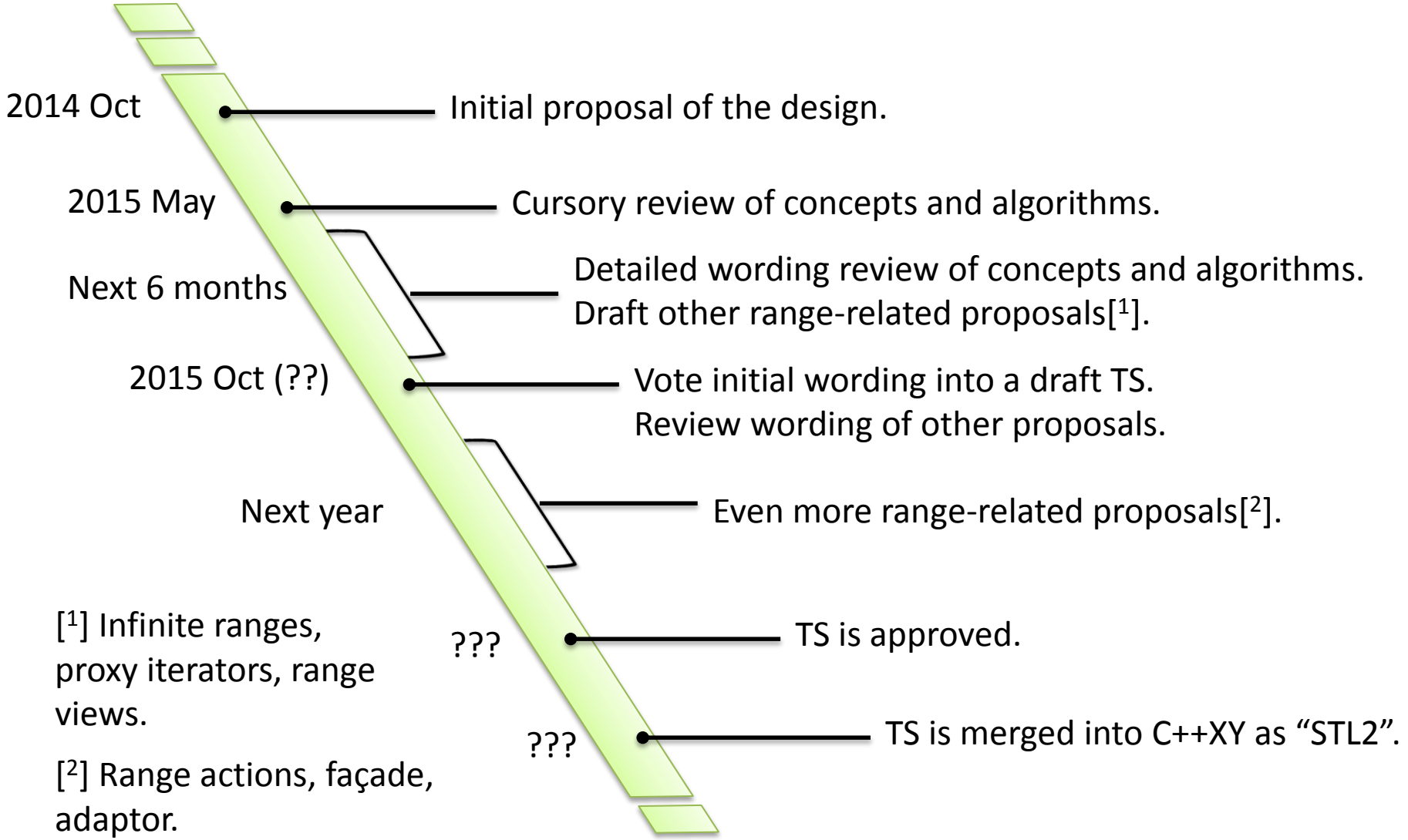
No loops!!!

Correct *by construction*.

Ranges and Standardization

Feature	Already Proposed?	Will be Proposed?
Range concepts		
Range algorithms		
View adaptors		
Range actions		
Façade/Adaptor helpers		

Standardization Timeline



[1] Infinite ranges, proxy iterators, range views.

[2] Range actions, façade, adaptor.

Find Out More

- N4128
  - High-level design, rationale, comparative analysis
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4128.html
- N4382
  - Standard wording for concepts, iterators, algorithms
  - http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2015/n4382.pdf
- Range v3 library
  - C++11 implementation
  - http://www.github.com/ericniebler/range-v3

Acknowledgements

- Andrew Sutton
- Sean Parent
- Herb Sutter and The Standard C++ Foundation

Questions?