Welcome to the Time Zone

Howard Hinnant
cppcon 2016

sep/20/2016
tue[3]/sep/2016
tue/38/2016

ripple
Where this library fits

IANA tz database

NTP Server

"tz.h"  
{This talk concentrates here.}

"date.h"  
{My Cppcon 2015 talk was here.}

<chrono>  
{Yesterday's talk was here.}

OS

hardware
Philosophy

- This library accurately parses all of the information in the IANA time zone database and presents it to the client with no errors.

  http://www.iana.org/time-zones

- No excuses.
Philosophy

Time Zone Database

The Time Zone Database (often called tz or zoneinfo) contains code and data that represent the history of local time for many representative locations around the globe. It is updated periodically to reflect changes made by political bodies to time zone boundaries, UTC offsets, and daylight-saving rules. Its management procedure is documented in BCP 175: Procedures for Maintaining the Time Zone Database.

Latest version

Time Zone Data v. 2016f (Released 2016-07-05)  tzdata2016f.tar.gz (305.9kb)
Time Zone Code v. 2016f (Released 2016-07-05)  tzcode2016f.tar.gz (190.3kb)
Type Safety

• What does "type safety" mean?

```c
civil_time
f(int y, int m, int d, int h, int mn, int s)
{
    // Do some computations ...
    return {y, mn, d, h, m, s};
}

Assuming civil_time has a constructor taking 6 ints, this compiles.
```
Type Safety

- What does "type safety" mean?

```c
typedef struct civil_time {
    int y, mn, d, h, m, s;
} civil_time;

civil_time f(int y, int m, int d, int h, int mn, int s) {
    // Do some computations ...
    return {y, mn, d, h, m, s};
}
```

Oops!
Run-time error!
Type Safety

• What does "type safety" mean?

civil_time
f(year y, month m, day d, hours h, minutes mn, seconds s)
{
    // Do some computations ...
    return {y, mn, d, h, m, s};
}

Assuming civil_time has a constructor taking 6 different corresponding types, this does not compile.
What does "type safety" mean?

civil_time
f(year y, month m, day d, hours h, minutes mn, seconds s)
{
    // Do some computations ...
    return {y, mn, d, h, m, s};
}

 error: no viable conversion from 'minutes' to 'month'
    return {y, mn, d, h, m, s};
          ^~

 error: no viable conversion from 'month' to 'minutes'
    return {y, mn, d, h, m, s};
      ^
Type Safety

- What does "type safety" mean?
- "Type safety" means that if you accidentally mix concepts (units or whatever), the compiler catches the mistake for you, *before* it becomes a *run-time error*.

- Compile-time errors, Good.
- Run-time errors, Bad.

This library stresses type safety.
Extension of `<chrono>`

• This library is a logical extension of `<chrono>.

• It does not replace `<chrono>.

• It builds upon `<chrono>.

• Thus it interoperates with `<chrono>` seamlessly.

• If you have trouble telling where `<chrono>` stops and this library begins, that is by design.
Primary Concepts

1. Calendar: A field-based structure that names a day.
   • E.g.: \{year, month, day\}. No associated time zone.

2. `sys_time<D>`: A `chrono::time_point` based on `system_clock` with precision D.
   • Unix Time, which is a very close approximation to UTC.

3. `local_time<D>`: A `chrono::time_point` based on 1970-01-01, with precision D, but no associated time zone.

4. `time_zone`: A geographical area and its full history of time zone rules.

5. `zoned_time<D>`: A pairing of a `local_time<D>` and a `time_zone`.
Calendars

1. Calendar: A field-based structure that names a day.
   - E.g.: \{year, month, day\}. \textit{No} associated time zone.

\begin{align*}
\text{year\_month\_day} & \quad \{\text{year, month, day}\} \\
\text{year\_month\_weekday} & \quad \{\text{year, month, weekday, index}\} \\
\text{iso\_week::year\_weeknum\_weekday} & \quad \{\text{year, index, weekday}\} \\
\text{julian::year\_month\_day} & \quad \{\text{year, month, day}\} \\
\text{islamic::year\_month\_day} & \quad \{\text{year, month, day}\}
\end{align*}

These are all just different ways of giving human readable names to days.
sys_time\langle D \rangle

2. sys_time\langle D \rangle: A chrono::time_point based on system_clock with precision D.
   • Unix Time, which is a very close approximation to UTC.

   template <class D>
   using sys_time = time_point<system_clock, D>;

   Convenience type aliases:
   using sys_seconds = sys_time<seconds>;
   using sys_days = sys_time<days>;

   Calendar types implicitly convert to and from sys_days.
   sys_days is the canonical name for all days which all calendars must translate to and from.
3. `local_time<D>`: A `chrono::time_point` based on 1970-01-01, with precision D, but *no* associated time zone.

```
template <class D>
  using local_time = time_point<local_t, D>;
```

Convenience type aliases:
```
  using local_seconds = local_time<seconds>;
  using local_days = local_time<days>;
```

Calendar types explicitly convert to and from `local_days`. The exact same math is used for `local_days` conversion as is used for `sys_days` conversion.
time_zone

4. time_zone: A geographical area and its full history of time zone rules.

A time_zone can be located with:

```cpp
auto tz = locate_zone("America/Los_Angeles");
```

The computer's current time zone is:

```cpp
auto tz = current_zone();
```
5. `zoned_time<D>`: A pairing of a `local_time<D>` and a `time_zone`.

A `zoned_time` represents the `local_time` in the `time_zone`.

The duration `D` is always seconds or finer.

```cpp
cout << make_zoned("America/Los_Angeles",
    local_days{2016_y/sep/20} + 16h + 45min) << '\n';

2016-09-20 16:45:00 PDT
```

`make_zoned` is a factory function for `zoned_time` which deduces the required precision.
5. **zoned_time<D>**: A pairing of a `local_time<D>` and a `time_zone`.

   sys_time can also be used to construct a `zoned_time`:

   ```
cout << make_zoned("America/Los_Angeles",
                    sys_days{2016_y/sep/20} + 23h + 45min) << '\n';
```

   ```
cout << make_zoned("America/Los_Angeles",
                    local_days{2016_y/sep/20} + 16h + 45min) << '\n';
```

   2016-09-20 16:45:00 PDT

   Both of these output the same time (in PDT).
5. **zoned_time<D>**: A pairing of a `local_time<D>` and a `time_zone`.

Alternative calendars fold in seamlessly:

```cpp
cout << make_zoned("America/Los_Angeles",
                   sys_days{tue[3]/sep/2016} + 23h + 45min) << '\n';
```

```cpp
cout << make_zoned("America/Los_Angeles",
                   local_days{iso_week::tue/38/2016} + 16h + 45min) << '\n';
```

2016-09-20 16:45:00 PDT

Both of these output the same time (in PDT).
5. zoned_time<D>: A pairing of a local_time<D> and a time_zone.

Handle arbitrary precision seamlessly:

```cpp
cout << make_zoned("America/Los_Angeles",
    sys_days{sep/21/2016} + 23h + 45min + 2ms) << '\n';
cout << make_zoned("America/Los_Angeles",
    local_days{2016_y/9/20} + 16h + 45min + 2ms) << '\n';

2016-09-20 16:45:00.002 PDT
```

Both of these output the same time (in PDT).
5. zoned_time<D>: A pairing of a local_time<D> and a time_zone.

This outputs the current local time in Los Angeles:

```cpp
cout << make_zoned("America/Los_Angeles", system_clock::now()) << '\n';
```

This outputs the current local time in the current local time zone:

```cpp
cout << make_zoned(current_zone(), system_clock::now()) << '\n';
```

2016-09-20 17:23:56.048936 PDT
Convert One Time Zone to Another

```cpp
auto meet_nyc = make_zoned("America/New_York",
   local_days{mon[1]/may/2016} + 9h);

auto meet_lon = make_zoned("Europe/London", meet_nyc);

   2016-05-02 14:00:00 BST

When a zoned_time is constructed from another zoned_time, their UTC equivalents are matched.
Formatting

The function `format` takes strftime-like format flags and a time point, and returns a `std::string`.

```cpp
auto utc_now = system_clock::now();
cout << format("%a, %b %d %Y at %I:%M %p %Z\n", utc_now);
    // Wed, Sep 21 2016 at 12:15 AM UTC

auto pdt_now = make_zoned(current_zone(), utc_now);
cout << format("%a, %b %d %Y at %I:%M %p %Z\n", pdt_now);
    // Tue, Sep 20 2016 at 05:15 PM PDT
```
The function `format` takes strftime-like format flags and a time point, and returns a `std::string`.

```cpp
class std::runtime_error
```

auto pdt_now = make_zoned(current_zone(), utc_now);
cout << format("%a, %b %d %Y at %I:%M %p %Z\n", pdt_now);

Tue, Sep 20 2016 at 05:15 PM PDT
```

auto lt_now = pdt_now.get_local_time();
cout << format("%a, %b %d %Y at %I:%M %p %Z\n", lt_now);

std::runtime_error: Can not format local_time with %Z
The function `format` takes strftime-like format flags and a time point, and returns a `std::string`.

```cpp
class std { 
    public:
        template<typename T>
        static std::string format(const char* fmt, T value) {
            std::stringstream ss;
            ss << fmt; // Convert strftime-like format flags
            return ss.str();
        }
};
```

```cpp
auto pdt_now = make_zoned(current_zone(), utc_now);
auto lt_now = pdt_now.get_local_time();
cout << format("%a, %b %d %Y at %I:%M %p %Z\n", pdt_now);
```

```
Tue, Sep 20 2016 at 05:15 PM PDT
```

```cpp
cout << format("%a, %b %d %Y at %I:%M %p\n", lt_now);
```

```
Tue, Sep 20 2016 at 05:15 PM
```
The function `format` takes strftime-like format flags and a time point, and returns a `std::string`.

- `format` can format:
  - `sys_time<D>`
  - `zoned_time<D>`
  - `local_time<D>`
- Exception thrown if `%Z` or `%z` is used with `local_time`. 
The function `format` takes strftime-like format flags and a time point, and returns a `std::string`.

Prefix your format calls with any `std::locale` your OS supports.

The global locale is the default.

```cpp
cout << format(locale{"fi_FI"},
               "%a, %b %d %Y at %I:%M %p\n", lt_now);

Ti, Syy 20 2016 at 05:15 pm
```
Formatting

The function `format` takes strftime-like format flags and a time point, and returns a `std::string`.

%S and %T output fractional seconds to the precision of the input time point.

```cpp
    cout << format("%F %T\n", utc_now);
    2016-09-21 00:15:37.269308

    cout << format("%F %T\n", floor<seconds>(utc_now));
    2016-09-21 00:15:37
```
The function `format` takes strftime-like format flags and a time point, and returns a `std::string`.

"Wide" format strings return "wide" std strings.

```cpp
wcout << format(L"%F %T\n", utc_now);
2016-09-21 00:15:37.269308
```
Parsing

The function `parse` parses a `basic_istream<CharT, Traits>` according to a `basic_string<CharT, Traits>` format, into a `sys_time<D>` or `local_time<D>`.

```cpp
system_clock::time_point utc_tp;
istringstream in{"Wed, Sep 21 2016 at 12:15 AM UTC"};
in >> parse("%a, %b %d %Y at %I:%M %p %Z", utc_tp);
cout << utc_tp << '\n';

2016-09-21 00:15:00.000000
```

The `iostream` iostate flags will be set accordingly (which may throw `ios_base::failure`).
Parsing

The function `parse` parses a `basic_istream<CharT, Traits>` according to a `basic_string<CharT, Traits>` format, into a `sys_time<D>` or `local_time<D>`.

```
istringstream in{"Wed, Sep 21 2016 at 12:15 AM UTC"};
in >> parse("%a, %b %d %Y at %I:%M %p %Z", utc_tp);

%Z requires a time zone abbreviation but has no impact on the value parsed.
```
Parsing

The function parse parses a basic_istream<CharT, Traits> according to a basic_string<CharT, Traits> format, into a sys_time<D> or local_time<D>.

```cpp
std::string s;
istringstream in{"Wed, Sep 21 2016 at 12:15 AM UTC"};
in >> parse("%a, %b %d %Y at %I:%M %p %Z", utc_tp, s);

s == "UTC"
```

One can optionally recover the abbreviation parsed by %Z into a basic_string<CharT, Traits>.
The function `parse` parses a `basic_istream<CharT, Traits>` according to a `basic_string<CharT, Traits>` format, into a `sys_time<D>` or `local_time<D>`.

```cpp
system_clock::time_point utc_tp;
istringstream in{"Tue, Sep 20 2016 at 5:15 PM -0700"};
in >> parse("%a, %d %b %d %Y at %I:%M %p %z", utc_tp);
cout << utc_tp << '\n';
```

2016-09-21 00:15:00.000000

Use of `%z` combined with a `sys_time<D>` will interpret the input data as a local time and will use the offset to convert to UTC.
Parsing

The function `parse` parses a `basic_istream<CharT, Traits>` according to a `basic_string<CharT, Traits>` format, into a `sys_time<D>` or `local_time<D>`.

```cpp
local_seconds local_tp;
istringstream in{"Tue, Sep 20 2016 at 5:15 PM -0700"};
in >> parse("%a, %b %d %Y at %I:%M %p %z", local_tp);
cout << local_tp << '\n';

2016-09-20 17:15:00
```

Use of `%z` combined with a `local_time<D>` will interpret the input data as a local time, requires the offset to be parsed, but then ignores it in assigning the local time value.
Parsing

The function parse parses a basic_istream<CharT, Traits> according to a basic_string<CharT, Traits> format, into a sys_time<D> or local_time<D>.

```cpp
minutes offset;
local_seconds local_tp;
istringstream in{"Tue, Sep 20 2016 at 5:15 PM -0700"};
in >> parse("%a, %b %d %Y at %I:%M %p %z", local_tp,
cout << local_tp << 'n';

2016-09-20 17:15:00

offset == -420min
```

Optionally one can supply a minutes duration to parse into.
Parsing

The function `parse` parses a `basic_istream<CharT, Traits>` according to a `basic_string<CharT, Traits>` format, into a `sys_time<D>` or `local_time<D>`.

```cpp
local_time<milliseconds> local_tp;
istringstream in{"Tue, Sep 20 2016 at 5:15:37.002 PM"};
in >> parse("%a, %b %d %Y at %I:%M:%S %p", local_tp);
cout << local_tp << '\n';

2016-09-20 17:15:37.002
```

%S and %T will parse sub-second precision if the input `time_point` has sub-second precision.
Parsing

The function `parse` parses a `basic_istream<CharT, Traits>` according to a `basic_string<CharT, Traits>` format, into a `sys_time<D>` or `local_time<D>`.

- parse can parse:
  - `sys_time<D>`
  - `local_time<D>`
local_time Arithmetic

local_time arithmetic across daylight saving boundaries is safe and easy.

```cpp
auto meeting = make_zoned("America/New_York", local_days{mar/11/2016} + 9h);
for (int i = 0; i < 4; ++i)
{
    cout << meeting << " == "
        << format("%F %T %Z\n", meeting.get_sys_time());
    meeting = meeting.get_local_time() + days{1};
}
```

This outputs the time of a 9am meeting for 4 days across a daylight saving boundary.
auto meeting = make_zoned("America/New_York",
    local_days{mar/11/2016} + 9h);
for (int i = 0; i < 4; ++i)
{
    cout << meeting << " == "
        << format("%F %T %Z\n", meeting.get_sys_time());
    meeting = meeting.get_local_time() + days{1};
}

2016-03-11 09:00:00 EST == 2016-03-11 14:00:00 UTC
2016-03-12 09:00:00 EST == 2016-03-12 14:00:00 UTC
2016-03-13 09:00:00 EDT == 2016-03-13 13:00:00 UTC
2016-03-14 09:00:00 EDT == 2016-03-14 13:00:00 UTC
local_time Arithmetic

If your local_time arithmetic results in an ambiguous or non-existent local time, an exception will be thrown.

Otherwise it will do what is intuitively the right thing.

sys_time arithmetic will unapologetically do exactly what it advertises to do, never resulting in an exception.
auto meeting = make_zoned("America/New_York", 
    local_days{mar/11/2016} + 9h);

for (int i = 0; i < 4; ++i)
{
    cout << meeting << " == " 
        << format("%F %T %Z\n", meeting.get_sys_time());
    meeting = meeting.get_sys_time() + days{1};
}

2016-03-11 09:00:00 EST == 2016-03-11 14:00:00 UTC
2016-03-12 09:00:00 EST == 2016-03-12 14:00:00 UTC
2016-03-13 10:00:00 EDT == 2016-03-13 14:00:00 UTC
2016-03-14 10:00:00 EDT == 2016-03-14 14:00:00 UTC
Converting Abbreviation to Time Zone

Given: 2016-09-20 17:15:37 PDT

How do you convert "PDT" into a time zone?

In general, you can't.

But this library gives you enough tools to give you a fighting chance.

```cpp
istringstream in{"2016-09-20 17:15:37 PDT"};
local_seconds tp;
string abbrev;
parse(in, "%F %T %Z", tp, abbrev);
auto v = find_by_abbrev(tp, abbrev);
```
Converting Abbreviation to Time Zone

template <class Duration>
auto
find_by_abbrev(local_time<Duration> tp, const string& abr) {
    vector<zoned_time<common_type_t<Duration, seconds>>> r;
    for (auto const& z : get_tzdb().zones) {
        auto i = z.get_info(tp);
        switch (i.result) {
            case local_info::unique:
            case local_info::ambiguous:
            case local_info::nonexistent:
        }
    }
    return r;
}
Converting Abbreviation to Time Zone

```cpp
auto i = z.get_info(tp);
switch (i.result) {
    case local_info::unique:
        if (i.first.abbrev == abr)
            r.push_back(make_zoned(&z, tp));
        break;
    case local_info::ambiguous:
    case local_info::nonexistent:
```
Converting Abbreviation to Time Zone

```cpp
auto i = z.get_info(tp);
switch (i.result) {
  case local_info::unique:
  case local_info::ambiguous:
    if (i.first.abbrev == abr)
      r.push_back(make_zoned(&z, tp, choose::earliest));
    else if (i.second.abbrev == abr)
      r.push_back(make_zoned(&z, tp, choose::latest));
    break;
  case local_info::nonexistent:
    break;
}
```
Converting Abbreviation to Time Zone

auto i = z.get_info(tp);
switch (i.result) {
    case local_info::unique:
    case local_info::ambiguous:
    case local_info::nonexistent:
        break;
}
Converting Abbreviation to Time Zone

```cpp
istringstream in{"2016-09-20 17:15:37 PDT"};
local_seconds tp;
string abbrev;
parse(in, "%F %T %Z", tp, abbrev);
auto v = find_by_abbrev(tp, abbrev);
for (auto const& zt : v)
    cout << format("%F %T %z ", zt)
    << zt.get_time_zone()->name() << '\n';
```

2016-09-20 17:15:37 -0700 America/Dawson
2016-09-20 17:15:37 -0700 America/Los_Angeles
2016-09-20 17:15:37 -0700 America/Tijuana
2016-09-20 17:15:37 -0700 America/Vancouver
2016-09-20 17:15:37 -0700 America/Whitehorse
2016-09-20 17:15:37 -0700 PST8PDT
Converting Abbreviation to Time Zone

Perhaps you can choose among these time zones with some additional information about your situation.

2016-09-20 17:15:37 -0700 America/Dawson
2016-09-20 17:15:37 -0700 America/Los_Angeles
2016-09-20 17:15:37 -0700 America/Tijuana
2016-09-20 17:15:37 -0700 America/Vancouver
2016-09-20 17:15:37 -0700 America/Whitehorse
2016-09-20 17:15:37 -0700 PST8PDT
Getting The Name of Your Time Zone

Say you get a `time_zone` without knowing its name:

```cpp
auto tz = current_zone();
auto zt = make_zoned(tz, system_clock::now());
```

Is there a way to serialize this time such that its `time_zone` can be recovered 100% of the time?
Getting The Name of Your Time Zone

Say you get a `time_zone` without knowing its name:

```cpp
auto tz = current_zone();
auto zt = make_zoned(tz, system_clock::now());
```

Is there a way to serialize this time such that its `time_zone` can be recovered 100% of the time?

Time zone abbreviations (%Z) can be ambiguous.

Time zone offsets (%z) can give you the correct UTC time but still don't give you the time zone. To relate past and future time stamps known to be in the same time zone, you need to know the time zone rules.
Say you get a `time_zone` without knowing its name:

```cpp
auto tz = current_zone();
auto zt = make_zoned(tz, system_clock::now());
cout << format("%F %T ", zt) << tz->name() << '\n';
```

2016-09-20 17:15:37.151362 America/Los_Angeles

Use the `time_zone` name instead of an abbreviation.
Getting The Name of Your Time Zone

Now you can parse it back in like this:

```cpp
istringstream in{"2016-09-20 17:15:37.151362
   " America/Los_Angeles"};
std::string tz_name;
local_time<microseconds> tp;
parse(in, "%F %T %Z", tp, tz_name);
auto zt = make_zoned(tz_name, tp);
```

By using the name of the `time_zone` in place of the abbreviation you've recovered 100% of the information about this time stamp and how it relates to past and future time stamps in the same `time_zone`. 
Computing With Leap Seconds

- `system_clock(sys_time<D>)` ignores the existence of leap seconds (as if the clock stops ticking during the leap second). This is unspecified, but the de facto standard.

- Thus `sys_time<D>` subtraction across a leap second insertion doesn't count the leap second.

- Leap second insertion dates are part of the IANA timezone database.

- This library presents that data as `utc_clock`, `utc_time<D>`, and conversions between `utc_time<D>` and `sys_time<D>`. 
Computing With Leap Seconds

```cpp
auto start = to_utc_time(sys_days{2015_y/jul/1} - 400ms);
auto end = start + 2s;
for (auto utc = start; utc < end; utc += 200ms) {
    auto sys = to_sys_time(utc);
    std::cout << utc << " UTC == " << sys << " SYS\n";
}
```

2015-07-01 00:00:00.000 UTC == 2015-07-01 00:00:00.000 SYS
2015-07-01 00:00:00.200 UTC == 2015-07-01 00:00:00.200 SYS
Computing With Leap Seconds

```cpp
auto t0 = sys_days{2015_y/jul/1} - 200ms;
auto t1 = sys_days{2015_y/jul/1} + 200ms;
cout << t1 - t0 << '\n';

400ms
```

```cpp
auto t0 = to_utc_time(sys_days{2015_y/jul/1} - 200ms);
auto t1 = to_utc_time(sys_days{2015_y/jul/1} + 200ms);
cout << t1 - t0 << '\n';

1400ms
```
Computing With Leap Seconds

auto t0 = sys_days{2016_y/jul/1} - 200ms;
auto t1 = sys_days{2016_y/jul/1} + 200ms;
cout << t1 - t0 << '\n';

400ms

auto t0 = to_utc_time(sys_days{2016_y/jul/1} - 200ms);
auto t1 = to_utc_time(sys_days{2016_y/jul/1} + 200ms);
cout << t1 - t0 << '\n';

400ms
Computing With Leap Seconds

- There also exists `tai_clock` and `gps_clock`.
- Bidirectional conversions exist between all four `time_points` via `to_XXX_time(t)` free functions.
Computing With Leap Seconds

```cpp
auto start = to_utc_time(sys_days{2015_y/jul/1} - 400ms);
auto end = start + 2s;
for (auto utc = start; utc < end; utc += 200ms) {
    auto tai = to_tai_time(utc);
    std::cout << utc << " UTC == " << tai << " TAI\n";
}
```

<table>
<thead>
<tr>
<th>Date</th>
<th>Time (UTC)</th>
<th>Time (TAI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015-06-30</td>
<td>23:59:59.600</td>
<td>2015-07-01 00:00:34.600</td>
</tr>
<tr>
<td>2015-06-30</td>
<td>23:59:59.800</td>
<td>2015-07-01 00:00:34.800</td>
</tr>
<tr>
<td>2015-06-30</td>
<td>23:59:60.000</td>
<td>2015-07-01 00:00:35.000</td>
</tr>
<tr>
<td>2015-06-30</td>
<td>23:59:60.200</td>
<td>2015-07-01 00:00:35.200</td>
</tr>
<tr>
<td>2015-06-30</td>
<td>23:59:60.400</td>
<td>2015-07-01 00:00:35.400</td>
</tr>
<tr>
<td>2015-06-30</td>
<td>23:59:60.600</td>
<td>2015-07-01 00:00:35.600</td>
</tr>
<tr>
<td>2015-06-30</td>
<td>23:59:60.800</td>
<td>2015-07-01 00:00:35.800</td>
</tr>
<tr>
<td>2015-07-01</td>
<td>00:00:00.000</td>
<td>2015-07-01 00:00:36.000</td>
</tr>
<tr>
<td>2015-07-01</td>
<td>00:00:00.200</td>
<td>2015-07-01 00:00:36.200</td>
</tr>
</tbody>
</table>
A timezone library has been presented which is:

- An extension of `<chrono>`.
- Type safe.
- Full functionality.
- Presents all IANA TimeZone data.
- Leap-second aware.

[https://howardhinnant.github.io/date/tz.html](https://howardhinnant.github.io/date/tz.html)