Howard E. Hinnant
2021-03-22

Time Zone Database Parser

Contents

Introduction

I had just completed writing `date`, which is a library for extending `<chrono>` into the realm of calendars, and I was looking around for the most challenging date time problem I could find with which I could demonstrate the power of this new library. "I know," I said to myself, "I'll handle all of the world's time zones, and maybe even leap seconds!" Thus began my journey into a rabbit hole which I knew existed, but had never truly appreciated the intricacies of.

This library adds timezone and leap second support to this date library. This is a separate library from `date` because many clients of `date` do not need timezone nor leap second support, and this support does not come for free (though the cost is quite reasonable).

This library is a **complete** parser of the IANA Time Zone Database. This database contains timezone information that represents the history of local time for many representative locations around the globe. It is updated every few months to reflect changes made by political bodies to time zone boundaries, UTC offsets, and daylight-saving rules. The database also maintains a list of leap seconds from 1972 through the present.

The IANA Time Zone Database contains four specific types of data:

1. Zone: A geographic location with a human-readable name (e.g. "America/New_York") which specifies the offset from UTC and an abbreviation for the zone. This data includes daylight saving rules, if applicable, for the zone. This data is not only the rules currently in effect for the region, but also includes specifications dating back to at least 1970, and in most cases dating back to the mid 1800's (when uniform time was first introduced across regions larger than individual towns and cities).

2. Rule: A specification for a single daylight-saving rule. This helps implement and consolidate the specifications of Zones.

3. link: This is an alternative name for a Zone.

4. leap: The date of the insertion of a leap second.

The library documented herein provides access to *all* of this data, and offers efficient and convenient ways to compute with it. And this is all done based on the `date` library, which in turn is based on the C++11/14 `<chrono>` library. So once you've learned those fundamental libraries, the learning curve for this library is greatly eased.

Description

Here is an overview of all the types we are going to talk about at some point. They are all fully covered in the reference section. This link is just there to give you a view of everything on one quick page so that you don't get lost or overwhelmed. Many of these types will never need to be explicitly named in typical use cases.

> tz_types.jpeg

Everything documented below is in `namespace date`. Explicit references to this namespace in example code below is intentionally omitted in the hopes of reducing verbosity.

What is the current local time?

One of the first things people want to do is find out what the current local time it is. Here is a complete program to print out the local time in human readable format:

```
#include "date/tz.h"
#include <iostream>

int
main()
{
    using namespace date;
    using namespace std::chrono;
    auto t = make_zoned(current_zone(), system_clock::now());
    std::cout << t << '\n';
}
```

This just output for me:

```
2016-05-14 18:33:24.205124 EDT
```

There are some noteworthy points about this program:

- This is a `<chrono>`-based system. The current time is found with `std::chrono::system_clock::now()`.
- The computer's current local time zone is not assumed. If anything is assumed that would be UTC, since this is the time zone that `system_clock` tracks (unspecified but de facto standard).
- Specifying you want to convert `system_clock::time_points` to the current local time zone is as easy as calling `date::current_zone()` and *pairing* that with a `system_clock::time_point` using `date::make_zoned`. This creates a `zoned_time`.
- This `zoned_time` maintains whatever precision it was given. On my platform `system_clock::now()` has microseconds precision, so in this example, `t` has microseconds precision as well.
- Then `t` is simply streamed out. By default the output represents all of the precision it is given.

Everything about the above program can be customized: the precision, the formatting, and the time zone. But by default, things just work, and don't throw away information.

For example let's say we wanted to limit the precision to milliseconds. This can be done by inserting `floor<milliseconds>` in one place. This makes `t` have just a precision of `milliseconds` and that is reflected in the streaming operator with no further effort:

```
auto t = make_zoned(current_zone(), floor<milliseconds>(system_clock::now()));
std::cout << t << '\n';  // 2016-05-14 18:33:24.205 EDT
```

Seconds precision is just as easy:

```
auto t = make_zoned(current_zone(), floor<seconds>(system_clock::now()));
std::cout << t << '\n';  // 2016-05-14 18:33:24 EDT
```

The entire `time_get` / `time_put` formatting capability is also at your fingertips (and at any precision):

```
auto t = make_zoned(current_zone(), system_clock::now());
std::cout << format("%a, %b %d, %Y at %I:%M %p %Z", t) << '\n';
// Sat, May 14, 2016 at 06:33 PM EDT
```

Using any `std::locale` your OS supports:

```
auto t = make_zoned(current_zone(), floor<seconds>(system_clock::now()));
std::cout << format(locale("de_DE"), "%a, %b %d, %Y at %T %Z", t) << '\n';
// Sa, Mai 14, 2016 at 18:33:24 EDT
```

What time is it somewhere *else* in the world?

From the previous section:

Hmm... German locale in an American time zone.

We can fix that easily too:

```
auto zone = locate_zone("Europe/Berlin");
auto t = make_zoned(zone, floor<seconds>(system_clock::now()));
std::cout << format(locale("de_DE"), "%a, %b %d, %Y at %T %Z", t) << '\n';
// So, Mai 15, 2016 at 00:33:24 CEST
```

The `date::locate_zone()` function looks up the IANA time zone with the name "Europe/Berlin" and returns a `const time_zone*` which has no ownership issues and can be freely and cheaply copied around. It is not possible for `locate_zone()` to return `nullptr`, though it might throw an exception if pushed far enough (e.g. `locate_zone("Disney/Mickey_Mouse")`).

You can also call `make_zoned` with the time zone name right in the call:

```
auto t = make_zoned("Europe/Berlin", floor<seconds>(system_clock::now()));
```

The first way is very slightly more efficient if you plan on using `zone` multiple times since it then only has to be looked up once.

How do I convert a `time_zone` from one time zone to another?

So far we've only looked at converting from `system_clock::now()` to a local, or specific time zone. We've used `make_zoned` with the first argument being either `current_zone()` or a specification for some other time zone, and the second argument being a `system_clock::time_point`. So far so good.

But now I have a video-conference meeting on the first Monday of May, 2016 at 9am New York time. I need to communicate that meeting with partners in London and Sydney. And the computation is taking place on a computer in New Zealand (or some other unrelated time zone). What does that look like?

```
#include "date/tz.h"
#include <iostream>

int
main()
{
    using namespace date::literals;
    using namespace std::chrono_literals;
    auto meet_nyc = make_zoned("America/New_York", date::local_days{Monday[1]/May/2016} + 9h);
    auto meet_lon = make_zoned("Europe/London",    meet_nyc);
    auto meet_syd = make_zoned("Australia/Sydney", meet_nyc);
    std::cout << "The New York meeting is " << meet_nyc << '\n';
    std::cout << "The London   meeting is " << meet_lon << '\n';
    std::cout << "The Sydney   meeting is " << meet_syd << '\n';
}
```

The output is the following. But before you forward it, send a generous bonus to the guys in Australia.

```
The New York meeting is 2016-05-02 09:00:00 EDT
The London   meeting is 2016-05-02 14:00:00 BST
The Sydney   meeting is 2016-05-02 23:00:00 AEST
```

The first time, `meet_nyc` is a pairing of a time zone ("America/New_York") with a *local time* (`Monday[1]/May/2016` at 09:00). Note that this input is exactly reflected in the output:

```
The New York meeting is 2016-05-02 09:00:00 EDT
```

The next line creates `meet_lon` with the `zoned_time` `meet_nyc` and a new time zone: "Europe/London". The effect of this pairing is to create a `time_point` with the exact same UTC time point, but associated with a different `time_zone` for localization purposes. That is, after this "converting construction", an invariant is that `meet_lon.get_sys_time() == meet_nyc.get_sys_time()`, even though these two objects refer to different time zones.

The same recipe is followed for creating `meet_syd`. The default formatting for these `zoned_times` is to output the *local* date and time followed by the current time zone abbreviation.

Summary: `zoned_time` is a pairing of local or UTC time with a `time_zone`. The result is a well-specified point in time. And it carries with it the ability to serve as a translator to any other `time_point` which carries time zone information (to any precision).

`local_time` vs `sys_time`

Let's say I want to refer to the New Years Day party at 2017-01-01 00:00:00. I don't want to refer to a specific party at some geographical location. I want to refer to the fact that this moment is celebrated in different parts of the world according to local times. This is called a `local_time`.

```
auto new_years = local_time<days>{2017_y/January/1} + 0h + 0m + 0s;
```

A `local_time<D>` can be created with any duration `D` and is a `std::chrono::time_point` except that `local_time<D>::clock` has no `now()` function. There is no time zone associated with `local_time`.

> `local_time` **is not** the time associated with the current local time the computer is set to.

`local_time` *is* a time associated with an *as yet* unspecified time zone. Only when you pair a `local_time` with a `time_zone` do you get a concrete point in time that can be converted to UTC and other time zones: a `zoned_time`.

There also exist convenience type aliases:

```
using local_seconds = local_time<std::chrono::seconds>;
using local_days    = local_time<days>;
```

In summary: When is 1min after New Years 2017?

```
auto t = local_days{January/1/2017} + 1min;
cout << t << '\n';  // 2017-01-01 00:01
```

When is 1min after New Years 2017 UTC?

```
auto t = sys_days{January/1/2017} + 1min;
cout << t << '\n';  // 2017-01-01 00:01
```

This effectively means that `year_month_day` is also ambiguous as to whether it refers to a local (timezone-less) time or to UTC. You have to specify which when you use it. But that is the nature of how people use dates (points in time with days precision). "There will be a celebration on New Years." In many contexts the time zone is intentionally left unspecified.

When is 1min after New Years 2017 in New York?

```
zoned_seconds t{"America/New_York", local_days{January/1/2017} + 1min};
cout << t << '\n';  // 2017-01-01 00:01:00 EST
```

What time will it be in New York when it is 1min after New Years 2017 UTC?

```
zoned_seconds t{"America/New_York", sys_days{January/1/2017} + 1min};
cout << t << '\n';  // 2016-12-31 19:01:00 EST
```

Summary

We now have 5 concepts and their associated types:

1. Calendars: These are day-precision time points that are typically field structures (multiple fields that create a unique "name" for a day).

   Example calendars include `year_month_day` and `year_month_weekday`. Other examples could include the ISO week-based calendar, the Julian calendar, the Islamic calendar, the Hebrew calendar, the Chinese calendar, the Mayan calendar, etc.

   Calendars can convert to and from both `sys_days` and `local_days`. These two conversions involve identical arithmetic, but have semantic differences.

   Once these conversions are implemented, the calendars are not only interoperable with `zoned_time`, but are also interoperable with each other. That is dates in the Chinese calendar can easily be converted to or from dates in the Mayan calendar even though these two calendars have no knowledge of the other.

   Disclaimer: "date.h" provides only the `year_month_day` and `year_month_weekday` calendars.

2. `sys_time`: This is a serial time point and a `std::chrono::time_point` of arbitrary precision. It has `sys_seconds` and `sys_days` convenience precisions.

   `sys_time` is a `time_point` associated with the return of `system_clock::now()` and represents [Unix Time] which very closely approximates UTC.

3. `local_time`: This is a serial time point and a `std::chrono::time_point` of arbitrary precision. It has `local_seconds` and `local_days` convenience precisions.

   `local_time` is a `time_point` associated with no time zone, and no `clock::now()`. It is the `void*` of `time_points`.

4. `time_zone`: This represents a specific geographical area, and all time zone related information for this area over all time. This includes a name for the area, and for any specific point in time, the UTC offset, the abbreviation, and additional information.

5. `zoned_time`: This is a pairing of a `time_zone` and a `sys_time` (of precision seconds or finer). It can also be equivalently viewed as a pairing of a `time_zone` and a `local_time`. Once constructed it represents a valid point in time, and the `time_zone`, `sys_time` and `local_time` can all be extracted. There exists a `zoned_seconds` convenience precision.

`time_zones` are retrieved from a time zone database. The database also holds information about leap seconds. To make computing with leap seconds easier, there is a clock that takes leap seconds into account: `utc_clock`. This clock has an associated family of time points called `utc_time`.

Full formatting and parsing facilities are available with `time_put`-like formatting strings.

Examples

Flight time

Interesting things can happen to the apparent time when you travel across the globe at high speeds. So departure and arrival times of airplane flights make for good examples involving time zone arithmetic.

```
#include "date/tz.h"
#include <iostream>

int
main()
{
    using namespace std::chrono_literals;
    using namespace date;

    auto departure = make_zoned("America/New_York", local_days{December/30/1978} + 12h + 1min);
    auto flight_length = 14h + 44min;
    auto arrival = make_zoned("Asia/Tehran", departure.get_sys_time() + flight_length);

    std::cout << "departure NYC time:  " << departure << '\n';
    std::cout << "flight time is       " << make_time(flight_length) << '\n';
    std::cout << "arrival Tehran time: " << arrival << '\n';
}
```

The output of the above program is:

```
departure NYC time:  1978-12-30 12:01:00 EST
flight time is       14:44
arrival Tehran time: 1978-12-31 11:45:00 IRST
```

The departure time is formed by transforming the local calendar date time into a `local_time` and pairing that with the "America/New_York" `time_zone` to form a `zoned_time`. The flight time is just an ordinary `chrono::duration`.

The arrival time is formed by retrieving the departure time in terms of `sys_time`, adding the length of the flight, and pairing that `sys_time` with the "Asia/Tehran" `time_zone` to form a `zoned_time`.

By doing the arithmetic (addition of the flight time) in the UTC (well system) time zone, we do not have to worry about things like daylight savings time, or other political changes to the either UTC offset. For example if we change one line to look at the same flight 24 hours later:

```
auto departure = make_zoned("America/New_York", local_days{December/31/1978} + 12h + 1min);
```

Then the output changes to:

```
departure NYC time:  1978-12-31 12:01:00 EST
flight time is       14:44
arrival Tehran time: 1979-01-01 11:15:00 IRST
```

Now we have the flight arriving 30min earlier. This is because the time zone "Asia/Tehran" undergoes an offset change while the plane is in the air, shifting its UTC offset to 30min earlier. Is this the final word on this example? Almost. If accuracy down to the second is required (it is not for a flight arrival), then additional effort needs to be expended. Because there was also a leap second insertion while the plane was in the air. This can be taken into account with the following code:

```
#include "date/tz.h"
#include <iostream>

int
main()
{
    using namespace std::chrono;
    using namespace date;

    auto departure = make_zoned("America/New_York", local_days{December/31/1978} + 12h + 1min);
    auto departure_utc = clock_cast<utc_clock>(departure.get_sys_time());
    auto flight_length = 14h + 44min;
    auto arrival = make_zoned("Asia/Tehran", clock_cast<system_clock>(departure_utc + flight_length));

    std::cout << "departure NYC time:  " << departure << '\n';
    std::cout << "flight time is       " << make_time(flight_length) << '\n';
    std::cout << "arrival Tehran time: " << arrival << '\n';
}
```

This is just like the previous example except that the arithmetic (departure time + flight length) is done in `utc_time` instead of `sys_time`. To accomplish this, there is a conversion from `sys_time` to `utc_time` before the arithmetic, and another conversion from `utc_time` to `sys_time` after the arithmetic. And the result changes to:

```
departure NYC time:  1978-12-31 12:01:00 EST
flight time is       14:44
arrival Tehran time: 1979-01-01 11:14:59 IRST
```

Format conversion

A common task in dealing with dates and times is converting from one string format to another. This library is extremely flexible in handling this task. As an example, let's say that you need to convert strings that look like this:

```
Sun Sep 16 01:03:52 -0500 1973
```

Into strings that look like this:

```
1973-09-16T06:03:52.000Z
```

That is, given a local time with UTC offset, you need to not only update the format to something more modern, but it also has to be converted to the UTC timezone and to a precision of milliseconds. The code to do this is quite straight forward:

```
std::string
convert(const std::string& input)
{
    using namespace std;
    using namespace std::chrono;
    using namespace date;
    istringstream stream{input};
    sys_time<milliseconds> t;
    stream >> parse("%a %b %d %T %z %Y", t);
    if (stream.fail())
        throw runtime_error("failed to parse " + input);
    return format("%FT%TZ", t);
}
```

Let's walk through this:

First, `date::parse` works with `istreams` so you can parse from files, from strings, or anything else that is an `istream`.

Second, while we don't need to parse to a precision of milliseconds, we need to format to that precision. It is easy just to parse into a milliseconds-precision `sys_time` so that we can then just format it back out with no change. If we needed to parse at finer precision than formatting, then we would need to parse at the higher precision, truncate it (by some rounding mode — `truncate`, `floor`, `ceil` or `round`), and then format the truncated value.

To have the `parse` interpret the string as a local time offset by the UTC offset, we need to ask for a `sys_time` to be parsed, and use the `%z` in the proper location. The `parse` function will then subtract the UTC offset to give us the proper `sys_time` value.

If `parse` fails to find *everything* in the parse/format string, exactly as specified, it will set `failbit` in the `istream`.

Finally, once we know we have a successfully parsed `sys_time<milliseconds>` it is a very simple matter to format it back out in whatever format is desired. As confirmed in the [Reference](), `%S` and `%T` are sensitive to the precision of the time point argument, and so there is no need for extension formatting flags to indicate fractional seconds. `%S` and `%T` just work.

Custom time zone

Occasionally the IANA time zone database doesn't quite do *everything* you want. This library allows you to use `zoned_time` with a time zone and/or pointer to time zone of your own making. One common example is the need to have a time zone that has a fixed offset from UTC, but for which that offset isn't known until run time. Below is an example which supplies a custom time zone called `OffsetZone` which can hold a UTC offset with minutes precision.

```
#include "date/tz.h"
#include <iostream>
#include <type_traits>

class OffsetZone
{
    std::chrono::minutes offset_;

public:
    explicit OffsetZone(std::chrono::minutes offset)
        : offset_{offset}
        {}

    template <class Duration>
        auto
        to_local(date::sys_time<Duration> tp) const
        {
            using namespace date;
            using namespace std;
            using namespace std::chrono;
            using LT = local_time<common_type_t<Duration, minutes>>;
            return LT{(tp + offset_).time_since_epoch()};
        }

    template <class Duration>
        auto
        to_sys(date::local_time<Duration> tp) const
        {
            using namespace date;
            using namespace std;
            using namespace std::chrono;
            using ST = sys_time<common_type_t<Duration, minutes>>;
            return ST{(tp - offset_).time_since_epoch()};
        }
```

```
};

int
main()
{
    using namespace date;
    using namespace std::chrono;
    OffsetZone p3_45{3h + 45min};
    zoned_time<milliseconds, OffsetZone*> zt{&p3_45, floor<milliseconds>(system_clock::now())};
    std::cout << zt.get_sys_time() << '\n';
    std::cout << zt.get_local_time() << '\n';
}
```

This just output for me:

```
2017-09-16 17:34:47.560
2017-09-16 21:19:47.560
```

The second template parameter to `zoned_time` is a pointer to a time zone. This example simply creates a `OffsetZone` with a UTC offset of 3:45, and constructs a `OffsetZone` which points to that custom time zone and supplies the current time to the desired precision (whatever that may be).

You don't have to use a built-in pointer to your time zone. You could just as easily use `unique_ptr`, `shared_ptr`, or whatever smart pointer is right for your application. And in C++17, you won't need to supply the template parameters for `zoned_time` (though you still can if you want to). That is, the construction of `zt` above could be simplified down to just this:

```
zoned_time zt{&p3_45, floor<milliseconds>(system_clock::now())};
```

One can *even* have `OffsetZone` serve as its *own* smart pointer by giving it a member `operator->()` that returns itself:

```
const OffsetZone* operator->() const {return this;}
```

This allows you to embed the `OffsetZone` *directly* into the `zoned_time` instead of pointing to an externally held `OffsetZone`:

```
zoned_time<milliseconds, OffsetZone> zt{OffsetZone{3h + 45min}, floor<milliseconds>(system_clock::now())};
```

As it stands, `zoned<Duration, OffsetZone>` can't be streamed with `operator<<` or formatted with `format`. But that can be fixed too: Just give `OffsetZone` a member `get_info` which takes a `sys_time` and returns a [`sys_info`](#sys_info):

```
template <class Duration>
    date::sys_info
    get_info(date::sys_time<Duration>) const
    {
        using namespace date;
        using namespace std::chrono;
        return {sys_seconds::min(), sys_seconds::max(), offset_,
                minutes{0}, offset_ >= minutes{0}
                                ? "+" + date::format("%H%M", offset_)
                                : "-" + date::format("%H%M", -offset_)};
    }
```

Above I've chosen to make the abbreviation for `OffsetZone` equivalent to `%z`, but I could have installed any `std::string` I wanted to. This allows me to say:

```
std::cout << zt << '\n';
```

which just output for me:

```
2017-09-16 21:36:10.913 +0345
```

If I want to make `zoned_time<Duration, OffsetZone>` default constructible, then I need to specialize `zoned_traits<OffsetZone>` with `default_zone()`:

```
namespace date
{

template <>
struct zoned_traits<OffsetZone>
{
    static
    OffsetZone
    default_zone()
    {
        using namespace std::chrono;
        return OffsetZone{minutes{0}};
    }
};

}  // namespace date
```

Now this:

```
zoned_time<milliseconds, OffsetZone> zt;
std::cout << zt << '\n';
```

outputs:

```
1970-01-01 00:00:00.000 +0000
```

And if I wanted to construct a `zoned_time<Duration, OffsetZone>` from a `string`, I need to add `static OffsetZone locate_zone(string name)` to my `zoned_traits` specialization.

```
namespace date
{

template <>
struct zoned_traits<OffsetZone>
{
    static
    OffsetZone
    default_zone()
    {
        using namespace std::chrono;
        return OffsetZone{minutes{0}};
    }

    static
    OffsetZone
    locate_zone(const std::string& name)
    {
        using namespace std::chrono;
        if (name == "UTC")
            return OffsetZone{minutes{0}};
        throw std::runtime_error{"OffsetZone can't handle anything but " + name};
    }
};

}  // namespace date
```

Now this:

```
zoned_time<seconds, OffsetZone> zt{"UTC", floor<seconds>(system_clock::now())};
std::cout << zt << '\n';
```

outputs:

```
2017-09-16 18:09:22 +0000
```

Reference

Everything specified below is in `namespace date`, and accessed via the header `"tz.h"`.

The database

The following data structure is the time zone database, and the following functions access it.

```
struct tzdb
{
    string            version;
    vector<time_zone> zones;
    vector<link>      links;
    vector<leap>      leaps;

    const time_zone* locate_zone(string_view tz_name) const;
    const time_zone* current_zone() const;
};

class tzdb_list
{
    std::atomic<tzdb*> head_{nullptr};  // exposition only

public:
    class const_iterator;

    const tzdb& front() const noexcept;

    const_iterator erase_after(const_iterator p) noexcept;

    const_iterator begin() const noexcept;
    const_iterator end()   const noexcept;

    const_iterator cbegin() const noexcept;
    const_iterator cend()   const noexcept;
};
```

The `tzdb_list` database is a singleton. Access is granted to it via the `get_tzdb_list()` function which returns a reference to it. However this access is only needed for those applications which need to have long uptimes and have a need to update the time zone database while running. Other applications can implicitly access the `front()` of this list via the *read-only* namespace scope functions `get_tzdb()`, `locate_zone()` and `current_zone()`. Each `vector` in `tzdb` is sorted to enable fast lookup. One can iterate over and inspect this database. And multiple versions of the database can be used at once, via the `tzdb_list`.

All information in the IANA time zone database is represented in the above `tzdb` data structure, except for the comments in the database. Thus it is up to you, the client of this library, to decide what to do with this data. This library makes it especially easy and convenient to extract the data in the way that is most commonly used (e.g. time conversions among time zones). But it represents *all* of the data, and hides *none* of it.

If compiled with `-DUSE_OS_TZDB` some of the information above will be missing:

- `version` will have the value `"unknown"` on Linux.
- `links` is missing as this version of the database makes no distinction between links and time zones.
- `leaps` will be missing on those platforms that do not ship with this information, which includes macOS and iOS.

```
const time_zone* tzdb::locate_zone(string_view tz_name) const;
```

Returns: If a `time_zone` is found for which `name() == tz_name`, returns a pointer to that `time_zone`. Otherwise if a `link` is found where `tz_name == link.name()`, then a pointer is returned to the `time_zone` for which `zone.name() == link.target()` [*Note:* A `link` is an alternative name for a `time_zone`. — *end note*]

Throws: If a `const time_zone*` can not be found as described in the *Returns* clause, throws a `runtime_error`. [*Note:* On non-exceptional return, the return value is *always* a pointer to a valid `time_zone`. — *end note*]

```
const time_zone* tzdb::current_zone() const;
```

Returns: A `const time_zone*` referring to the time zone which your computer has set as its local time zone.

```
list<tzdb>& get_tzdb_list();
```

Effects: If this is the first access to the database, will initialize the database. If this call initializes the database, the resulting database will be a `tzdb_list` which holds a single initialized `tzdb`.

If `tz.cpp` was compiled with the configuration macro `AUTO_DOWNLOAD == 1`, initialization will include checking the IANA website for the latest version, and downloading the latest version if your local version is out of date, or doesn't exist at the location referred to by the `install` configuration variable in `tz.cpp`. If `tz.cpp` was compiled with `AUTO_DOWNLOAD == 0`, you will have to download and decompress the IANA database from the IANA website and place it at the location referred to by the `install` configuration variable.

`AUTO_DOWNLOAD == 1` requires linking `tz.cpp` to `libcurl`.

Returns: A reference to the database.

Thread Safety: It is safe to call this function from multiple threads at one time.

Throws: `runtime_error` if for any reason a reference can not be returned to a valid `list<tzdb>&` containing one or more valid `tzdb`.

```
const tzdb& get_tzdb();
```

Returns: `get_tzdb_list().front()`.

```
const time_zone* locate_zone(string_view tz_name);
```

Returns: `get_tzdb().locate_zone(tz_name)` which will initialize the timezone database if this is the first reference to the database.

```
const time_zone* current_zone();
```

Returns: `get_tzdb().current_zone()`.

`tzdb_list::const_iterator` is a constant iterator which meets the forward iterator requirements and has a value type of `tzdb`.

```
const tzdb& tzdb_list::front() const noexcept;
```

Returns: `*head_`.

Remarks: this operation is thread safe with respect to `reload_tzdb()`. [*Note:* `reload_tzdb()` pushes a new `tzdb` onto the front of this container. — *end note*]

```
tzdb::const_iterator tzdb::erase_after(const_iterator p) noexcept;
```

Requires: The iterator following `p` is dereferenceable.

Effects: Erases the `tzdb` referred to by the iterator following `p`.

Returns: An iterator pointing to the element following the one that was erased, or `end()` if no such element exists.

Remarks: No pointers, references or iterators are invalidated except those referring to the erased `tzdb`.

Note: It is not possible to erase the `tzdb` referred to by `begin()`.

```
tzdb::const_iterator tzdb::begin() const noexcept;
```

Returns: An iterator referring to the first `tzdb` in the container.

```
tzdb::const_iterator tzdb::end() const noexcept;
```

Returns: An iterator referring to the position one past the last `tzdb` in the container.

```
tzdb::const_iterator tzdb::cbegin() const noexcept;
```

Returns: `begin()`.

```
tzdb::const_iterator tzdb::cend() const noexcept;
```

Returns: `end()`.

```
const tzdb& reload_tzdb();
```

Effects:

If If `tz.cpp` was compiled with the configuration macro `AUTO_DOWNLOAD == 1`, this function first checks the latest version at the IANA website. If the IANA website is unavailable, or if the latest version is already installed, there are no effects. Otherwise, a new version is available. It is downloaded and installed, and then the program initializes a new `tzdb` from the new disk files and pushes it to the front of the `tzdb_list` accessed by `get_tzdb_list()`.

If `tz.cpp` was compiled with the configuration macro `AUTO_DOWNLOAD == 0`, this function initializes a new `tzdb` from the disk files and pushes it to the front of the `tzdb_list` accessed by `get_tzdb_list()`. You can manually replace the database without ill-effects after your program has called `get_tzdb()` and before it calls `reload_tzdb()`, as there is no access to the files on disk between the first call to `get_tzdb()` and subsequent calls to `reload_tzdb()`.

Returns: `get_tzdb_list().front()`.

Remarks: No pointers, references or iterators are invalidated.

Thread Safety: This function is thread safe with respect to `front()` and `erase_after()`.

Throws: `runtime_error` if for any reason a reference can not be returned to a valid `tzdb`.

Remarks: Not available with `USE_OS_TZDB == 1`.

```
void set_install(const std::string& s);
```

Effects: Sets the location which the database is, or will be, installed at.

Default: If this function is never called, and the macro `INSTALL` is defined, the location of the database is `INSTALL/tzdata` (`INSTALL\tzdata` on Windows). If the macro `INSTALL` is not defined, the default location of the database is `~/Downloads/tzdata` (`%homedrive%\%homepath%\downloads\tzdata` on Windows).

Thread Safety: This function is *not* thread safe. You must provide your own synchronization among threads calling this function. It should be called prior to any other access to the database.

Throws: Anything assignment to `std::string` might throw.

Note: No expansion is attempted on the argument `s` (e.g. expanding `~/`). This is a low-level function meant to enable other expansion techniques such as xdg-user-dirs whose result can be passed directly to `set_install`.

Remarks: Not available with `USE_OS_TZDB == 1`.

The following functions are available only if you compile with the configuration macro `HAS_REMOTE_API == 1`. Use of this API requires linking to `libcurl`. `AUTO_DOWNLOAD == 1` requires `HAS_REMOTE_API == 1`. You will be notified at compile time if `AUTO_DOWNLOAD == 1` and `HAS_REMOTE_API == 0`. If `HAS_REMOTE_API == 1`, then `AUTO_DOWNLOAD` defaults to `1`, otherwise `AUTO_DOWNLOAD` defaults to `0`. On Windows, `HAS_REMOTE_API` defaults to `0`. Everywhere else it defaults to `1`. This is because `libcurl` comes preinstalled everywhere but Windows, but it is available for Windows.

None of these are available with `USE_OS_TZDB == 1`.

[*Note:* Even with `AUTO_DOWNLOAD == 1`, there are no thread-safety issues with this library unless one of the following functions are *explicitly* called by your code:

```
const tzdb& reload_tzdb();
bool remote_download(const std::string& version);
bool remote_install(const std::string& version);
```

Once your program has initialized the `tzdb` singleton, that singleton can *never* be changed without *explicit* use of `reload_tzdb()`. — *end note*]

```
std::string remote_version();
```

Returns: The latest database version number from the IANA website. If the IANA website can not be reached, or if it can be reached but the latest version number is unexpectedly not available, the empty string is returned.

Note: If non-empty, this can be compared with `get_tzdb().version` to discover if you have the latest database installed.

```
bool remote_download(const std::string& version);
```

Effects: If `version == remote_version()` this function will download the compressed tar file holding the latest time zone database from the IANA website. The tar file will be placed at the location indicated by the `install` configuration variable in `tz.cpp`.

Returns: `true` if the database was successfully downloaded, else `false`.

Thread safety: If called by multiple threads, there will be a race on the creation of the tar file at `install`.

```
bool remote_install(const std::string& version);
```

Effects: If `version` refers to the file successfully downloaded by `remote_download()` this function will remove the existing time zone database at `install`, then extract a new database from the tar file and place it at `install`, and finally will delete the tar file.

This function *does not* cause your program to re-initialize itself from this new database. In order to do that, you must call `reload_tzdb()` (or `get_tzdb()` if the database has yet to be initialized). If `tz.cpp` was compiled with `AUTO_DOWNLOAD == 1`, then `reload_tzdb()` uses this API to check if the database is out of date, and reinitializes it with a freshly downloaded database only if it needs to. Indeed, if `AUTO_DOWNLOAD == 1` there is never any need to call `remote_download()` or `remote_install()` explicitly. You can just call `reload_tzdb()` instead. This API is only exposed so that you can take care of this manually if desired (`HAS_REMOTE_API == 1 && AUTO_DOWNLOAD == 0`).

Returns: `true` if the database was successfully replaced by the tar file , else `false`.

Thread safety: If called by multiple threads, there will be a race on the creation of the new database at `install`.

Everything else in this library concerns *read-only* access to this database, and intuitive ways to compute with that information, even while being oblivious to the fact that you *are* accessing a database.

The entire database on disk occupies less than half of the disk space consumed by an average Beatles song. Don't sweat multiple copies of it. It will easily fit in your smart toaster.

`choose`

For some conversions from `local_time` to a `sys_time`, `choose::earliest` or `choose::latest` can be used to convert a non-existent or ambiguous `local_time` into a `sys_time`, instead of throwing an exception.

```
enum class choose {earliest, latest};
```

`nonexistent_local_time`

`nonexistent_local_time` is thrown when one attempts to convert a non-existent `local_time` to a `sys_time` without specifying `choose::earliest` or `choose::latest`.

```
class nonexistent_local_time
    : public std::runtime_error
{
public:
    template <class Duration>
        nonexistent_local_time(local_time<Duration> tp, const local_info& i);
};
```

```
template <class Duration>
nonexistent_local_time::nonexistent_local_time(local_time<Duration> tp,
                                               const local_info& i);
```

Requires: `i.result == local_info::nonexistent`.

Effects: Constructs a `nonexistent_local_time` by initializing the base class with a sequence of `char` equivalent to that produced by `os.str()` initialized as shown below:

```
std::ostringstream os;
os << tp << " is in a gap between\n"
   << local_seconds{i.first.end.time_since_epoch()} + i.first.offset << ' '
   << i.first.abbrev << " and\n"
   << local_seconds{i.second.begin.time_since_epoch()} + i.second.offset << ' '
   << i.second.abbrev
   << " which are both equivalent to\n"
   << i.first.end << " UTC";
```

[*Example:*

```
#include "date/tz.h"
#include <iostream>

int
main()
{
    using namespace date;
    using namespace std::chrono_literals;
    try
    {
        auto zt = make_zoned("America/New_York", local_days{Sunday[2]/March/2016} + 2h + 30min);
    }
    catch (const nonexistent_local_time& e)
    {
        std::cout << e.what() << '\n';
    }
}
```

Which outputs:

```
2016-03-13 02:30:00 is in a gap between
2016-03-13 02:00:00 EST and
2016-03-13 03:00:00 EDT which are both equivalent to
2016-03-13 07:00:00 UTC
```

— end example:]

`ambiguous_local_time`

`ambiguous_local_time` is thrown when one attempts to convert an ambiguous `local_time` to a `sys_time` without specifying `choose::earliest` or `choose::latest`.

```
class ambiguous_local_time
    : public std::runtime_error
{
public:
    template <class Duration>
        ambiguous_local_time(local_time<Duration> tp, const local_info& i);
};
```

```
template <class Duration>
ambiguous_local_time::ambiguous_local_time(local_time<Duration> tp,
                                           const local_info& i);
```

Requires: `i.result == local_info::ambiguous`.

Effects: Constructs an `ambiguous_local_time` by initializing the base class with a sequence of `char` equivalent to that produced by `os.str()` initialized as shown below:

```
std::ostringstream os;
os << tp << " is ambiguous.  It could be\n"
   << tp << ' ' << i.first.abbrev << " == "
   << tp - i.first.offset << " UTC or\n"
   << tp << ' ' << i.second.abbrev  << " == "
   << tp - i.second.offset  << " UTC";
```

[*Example:*

```
#include "date/tz.h"
#include <iostream>

int
main()
{
    using namespace date;
    using namespace std::chrono_literals;
    try
    {
        auto zt = make_zoned("America/New_York", local_days{Sunday[1]/November/2016} + 1h + 30min);
    }
    catch (const ambiguous_local_time& e)
    {
        std::cout << e.what() << '\n';
```

```
    }
}
```

Which outputs:

```
2016-11-06 01:30:00 is ambiguous.  It could be
2016-11-06 01:30:00 EDT == 2016-11-06 05:30:00 UTC or
2016-11-06 01:30:00 EST == 2016-11-06 06:30:00 UTC
```

— *end example:*]

`sys_info`

This structure can be obtained from the combination of a `time_zone` and either a `sys_time`, or `local_time`. It can also be obtained from a `zoned_time` which is effectively a `pair` of a `time_zone` and `sys_time`.

This structure represents a lower-level API. Typical conversions from `sys_time` to `local_time` will use this structure *implicitly*, not *explicitly*.

```
struct sys_info
{
    sys_seconds          begin;
    sys_seconds          end;
    std::chrono::seconds offset;
    std::chrono::minutes save;
    std::string          abbrev;
};
```

The `begin` and `end` fields indicate that for the associated `time_zone` and `time_point`, the `offset` and `abbrev` are in effect in the range `[begin, end)`. This information can be used to efficiently iterate the transitions of a `time_zone`.

The `offset` field indicates the UTC offset in effect for the associated `time_zone` and `time_point`. The relationship between `local_time` and `sys_time` is:

```
offset = local_time - sys_time
```

The `save` field is "extra" information not normally needed for conversion between `local_time` and `sys_time`. If `save != 0min`, this `sys_info` is said to be on "daylight saving" time, and `offset - save` suggests what this `time_zone` *might* use if it were off daylight saving. However this information should not be taken as authoritative. The only sure way to get such information is to query the `time_zone` with a `time_point` that returns an `sys_info` where `save == 0min`. There is no guarantee what `time_point` might return such an `sys_info` except that it is guaranteed *not* to be in the range `[begin, end)` (if `save != 0min` for this `sys_info`).

When compiled with `USE_OS_TZDB == 1`, the underlying database collapses this information down to a `bool` which is fase when daylight saving is not in effect and true when it is. When `USE_OS_TZDB == 1`, the `save` field will be `0min` when daylight saving is not in effect and `1min` when daylight saving is in effect.

The `abbrev` field indicates the current abbreviation used for the associated `time_zone` and `time_point`. Abbreviations are not unique among the `time_zones`, and so one can not reliably map abbreviations back to a `time_zone` and UTC offset.

You can stream out a `sys_info`:

```
std::ostream& operator<<(std::ostream& os, const sys_info& r);
```

`local_info`

This structure represents a lower-level API. Typical conversions from `local_time` to `sys_time` will use this structure *implicitly*, not *explicitly*.

```
struct local_info
{
    enum {unique, nonexistent, ambiguous} result;
    sys_info first;
    sys_info second;
};
```

When a `local_time` to `sys_time` conversion is unique, `result == unique`, `first` will be filled out with the correct `sys_info` and `second` will be zero-initialized. If the conversion stems from a nonexistent `local_time` then `result == nonexistent`, `first` will be filled out with the `sys_info` that ends just prior to the `local_time` and `second` will be filled out with the `sys_info` that begins just after the `local_time`. If the conversion stems from an ambiguous `local_time` then `result == ambiguous`, `first` will be filled out with the `sys_info` that ends just after the `local_time` and `second` will be filled out with the `sys_info` that starts just before the `local_time`.

You can stream out a `local_info`:

```
std::ostream& operator<<(std::ostream& os, const local_info& r);
```

`time_zone`

A `time_zone` represents all time zone transitions for a specific geographic area. `time_zone` construction is undocumented, and done for you during the database initialization. You can gain `const` access to a `time_zone` via functions such as `locate_zone`.

```
class time_zone
{
public:
    time_zone(const time_zone&) = delete;
    time_zone& operator=(const time_zone&) = delete;

    const std::string& name() const;

    template <class Duration> sys_info   get_info(sys_time<Duration> st) const;
    template <class Duration> local_info get_info(local_time<Duration> tp) const;

    template <class Duration>
        sys_time<typename std::common_type<Duration, std::chrono::seconds>::type>
        to_sys(local_time<Duration> tp) const;

    template <class Duration>
        sys_time<typename std::common_type<Duration, std::chrono::seconds>::type>
        to_sys(local_time<Duration> tp, choose z) const;

    template <class Duration>
        local_time<typename std::common_type<Duration, std::chrono::seconds>::type>
        to_local(sys_time<Duration> tp) const;
};

bool operator==(const time_zone& x, const time_zone& y);
bool operator!=(const time_zone& x, const time_zone& y);
bool operator< (const time_zone& x, const time_zone& y);
bool operator> (const time_zone& x, const time_zone& y);
bool operator<=(const time_zone& x, const time_zone& y);
bool operator>=(const time_zone& x, const time_zone& y);

std::ostream& operator<<(std::ostream& os, const time_zone& z)
```

```
const std::string& time_zone::name() const;
```

Returns: The name of the `time_zone`.

Example: "America/New_York".

Note: Here is an unofficial list of `time_zone` names: https://en.wikipedia.org/wiki/List_of_tz_database_time_zones.

```
template <class Duration> sys_info time_zone::get_info(sys_time<Duration> st) const;
```

Returns: A `sys_info` `i` for which `st` is in the range `[i.begin, i.end)`.

```
template <class Duration> local_info time_zone::get_info(local_time<Duration> tp) const;
```

Returns: A `local_info` for `tp`.

```
template <class Duration>
sys_time<typename std::common_type<Duration, std::chrono::seconds>::type>
time_zone::to_sys(local_time<Duration> tp) const;
```

Returns: A `sys_time` that is at least as fine as `seconds`, and will be finer if the argument `tp` has finer precision. This `sys_time` is the UTC equivalent of `tp` according to the rules of this `time_zone`.

Throws: If the conversion from `tp` to a `sys_time` is ambiguous, throws `ambiguous_local_time`. If the conversion from `tp` to a `sys_time` is nonexistent, throws `nonexistent_local_time`.

```
template <class Duration>
sys_time<typename std::common_type<Duration, std::chrono::seconds>::type>
time_zone::to_sys(local_time<Duration> tp, choose z) const;
```

Returns: A `sys_time` that is at least as fine as `seconds`, and will be finer if the argument `tp` has finer precision. This `sys_time` is the UTC equivalent of `tp` according to the rules of this `time_zone`. If the conversion from `tp` to a `sys_time` is ambiguous, returns the earlier `sys_time` if `z == choose::earliest`, and returns the later `sys_time` if `z == choose::latest`. If the `tp` represents a non-existent time between two UTC `time_points`, then the two UTC `time_points` will be the same, and that UTC `time_point` will be returned.

```
template <class Duration>
local_time<typename std::common_type<Duration, std::chrono::seconds>::type>
time_zone::to_local(sys_time<Duration> tp) const;
```

Returns: The `local_time` associated with `tp` and this `time_zone`.

```
bool operator==(const time_zone& x, const time_zone& y);
```

Returns: `x.name() == y.name()`.

```
bool operator!=(const time_zone& x, const time_zone& y);
```

Returns: `!(x == y)`.

```
bool operator<(const time_zone& x, const time_zone& y);
```

Returns: `x.name() < y.name()`.

```
bool operator>(const time_zone& x, const time_zone& y);
```

Returns: `y < x`.

```
bool operator<=(const time_zone& x, const time_zone& y);
```

Returns: `!(y < x)`.

```
bool operator>=(const time_zone& x, const time_zone& y);
```

Returns: `!(x < y)`.

```
std::ostream& operator<<(std::ostream& os, const time_zone& z)
```

Produces an output that is probably more meaningful to me than it is to you. I found it useful for debugging this library.

zoned_traits

`zoned_traits` provides a means for customizing the behavior of `zoned_time<Duration, TimeZonePtr>` for the `zoned_time` default constructor, and constructors taking `string_view`. A specialization for `const time_zone*` is provided by the implementation.

```
template <class T> struct zoned_traits {};

template <>
struct zoned_traits<const time_zone*>
{
    static const time_zone* default_zone();
    static const time_zone* locate_zone(string_view name);
};
```

```
static const time_zone* zoned_traits<const time_zone*>::default_zone();
```

Returns: `date::locate_zone("UTC")`.

```
static const time_zone* zoned_traits<const time_zone*>::locate_zone(string_view name);
```

Returns: `date::locate_zone(name)`.

zoned_time

`zoned_time` represents a logical paring of `time_zone` and a `time_point` with precision `Duration`.

```
template <class Duration, class TimeZonePtr = const time_zone*>
class zoned_time
{
public:
    using duration = common_type_t<Duration, seconds>;

private:
    TimeZonePtr        zone_;  // exposition only
    sys_time<duration> tp_;    // exposition only

public:
    zoned_time();
    zoned_time(const zoned_time&) = default;
    zoned_time& operator=(const zoned_time&) = default;

             zoned_time(const sys_time<Duration>& st);
    explicit zoned_time(TimeZonePtr z);
    explicit zoned_time(string_view name);

    template <class Duration2>
        zoned_time(const zoned_time<Duration2>& zt) noexcept;

    zoned_time(TimeZonePtr z,    const sys_time<Duration>& st);
    zoned_time(string_view name, const sys_time<Duration>& st);

    zoned_time(TimeZonePtr z,    const local_time<Duration>& tp);
    zoned_time(string_view name, const local_time<Duration>& tp);
    zoned_time(TimeZonePtr z,    const local_time<Duration>& tp, choose c);
    zoned_time(string_view name, const local_time<Duration>& tp, choose c);

    template <class Duration2, class TimeZonePtr2>
        zoned_time(TimeZonePtr z, const zoned_time<Duration2, TimeZonePtr2>& zt);
```

```
    template <class Duration2, class TimeZonePtr2>
        zoned_time(TimeZonePtr z, const zoned_time<Duration2, TimeZonePtr2>& zt, choose);

    zoned_time(string_view name, const zoned_time<Duration>& zt);
    zoned_time(string_view name, const zoned_time<Duration>& zt, choose);

    zoned_time& operator=(const sys_time<Duration>& st);
    zoned_time& operator=(const local_time<Duration>& ut);

             operator sys_time<duration>()   const;
    explicit operator local_time<duration>() const;

    TimeZonePtr          get_time_zone()  const;
    local_time<duration> get_local_time() const;
    sys_time<duration>   get_sys_time()   const;
    sys_info             get_info()       const;
};

using zoned_seconds = zoned_time<std::chrono::seconds>;

template <class Duration1, class Duration2, class TimeZonePtr>
bool
operator==(const zoned_time<Duration1, TimeZonePtr>& x,
           const zoned_time<Duration2, TimeZonePtr>& y);

template <class Duration1, class Duration2, class TimeZonePtr>
bool
operator!=(const zoned_time<Duration1, TimeZonePtr>& x,
           const zoned_time<Duration2, TimeZonePtr>& y);

template <class charT, class traits, class Duration, class TimeZonePtr>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os,
               const zoned_time<Duration, TimeZonePtr>& t);

template <class charT, class traits, class Duration, class TimeZonePtr>
    basic_ostream<charT, traits>&
    to_stream(basic_ostream<charT, traits>& os, const charT* fmt,
              const zoned_time<Duration, TimeZonePtr>& tp);

zoned_time()
    -> zoned_time<seconds>;

template <class Duration>
    zoned_time(sys_time<Duration>)
        -> zoned_time<common_type_t<Duration, seconds>>;

template <class TimeZonePtr, class Duration>
    zoned_time(TimeZonePtr, sys_time<Duration>)
        -> zoned_time<common_type_t<Duration, seconds>, TimeZonePtr>;

template <class TimeZonePtr, class Duration>
    zoned_time(TimeZonePtr, local_time<Duration>, choose = choose::earliest)
        -> zoned_time<common_type_t<Duration, seconds>, TimeZonePtr>;

template <class TimeZonePtr, class Duration>
    zoned_time(TimeZonePtr, zoned_time<Duration>, choose = choose::earliest)
        -> zoned_time<common_type_t<Duration, seconds>, TimeZonePtr>;

zoned_time(string_view)
    -> zoned_time<seconds>;

template <class Duration>
zoned_time(string_view, sys_time<Duration>)
    -> zoned_time<common_type_t<Duration, seconds>>;

template <class Duration>
zoned_time(string_view, local_time<Duration>, choose = choose::earliest)
    -> zoned_time<common_type_t<Duration, seconds>>;

template <class Duration, class TimeZonePtr, class TimeZonePtr2>
zoned_time(TimeZonePtr, zoned_time<Duration, TimeZonePtr2>)
    -> zoned_time<Duration, TimeZonePtr>;

template <class Duration, class TimeZonePtr, class TimeZonePtr2>
zoned_time(TimeZonePtr, zoned_time<Duration, TimeZonePtr2>, choose)
    -> zoned_time<Duration, TimeZonePtr>;
```

An invariant of `zoned_time<Duration>` is that it always refers to a valid `time_zone`, and represents a point in time that exists and is not ambiguous.

```
zoned_time<Duration, TimeZonePtr>::zoned_time();
```

> *Remarks:* This constructor does not participate in overload resolution unless the expression `zoned_traits<TimeZonePtr>::default_zone()` is well formed.

Effects: Constructs a `zoned_time` by initializing `zone_` with `zoned_traits<TimeZonePtr>::default_zone()` and default constructing `tp_`.

```
zoned_time<Duration, TimeZonePtr>::zoned_time(const zoned_time&) = default;
zoned_time<Duration, TimeZonePtr>& zoned_time<Duration>::operator=(const zoned_time&) = default;
```

The copy members transfer the associated `time_zone` from the source to the destination. After copying, source and destination compare equal. If `Duration` has `noexcept` copy members, then `zoned_time<Duration>` has `noexcept` copy members.

```
zoned_time<Duration, TimeZonePtr>::zoned_time(const sys_time<Duration>& st);
```

Remarks: This constructor does not participate in overload resolution unless the expression `zoned_traits<TimeZonePtr>::default_zone()` is well formed.

Effects: Constructs a `zoned_time` by initializing `zone_` with `zoned_traits<TimeZonePtr>::default_zone()` and `tp_` with `st`.

```
explicit zoned_time<Duration, TimeZonePtr>::zoned_time(TimeZonePtr z);
```

Requires: `z` refers to a valid `time_zone`.

Effects: Constructs a `zoned_time` initializing `zone_` with `std::move(z)`.

```
explicit zoned_time<Duration, TimeZonePtr>::zoned_time(string_view name);
```

Remarks: This constructor does not participate in overload resolution unless the expression `zoned_traits<TimeZonePtr>::locate_zone(string_view{})` is well formed and `zoned_time` is constructible from the return type of `zoned_traits<TimeZonePtr>::locate_zone(string_view{})`.

Effects: Constructs a `zoned_time` by initializing `zone_` with `zoned_traits<TimeZonePtr>::locate_zone(name)` and default constructing `tp_`.

```
template <class Duration2, TimeZonePtr>
    zoned_time<Duration>::zoned_time(const zoned_time<Duration2, TimeZonePtr>& y) noexcept;
```

Remarks: Does not participate in overload resolution unless `sys_time<Duration2>` is implicitly convertible to `sys_time<Duration>`.

Effects: Constructs a `zoned_time` `x` such that `x == y`.

```
zoned_time<Duration, TimeZonePtr>::zoned_time(TimeZonePtr z, const sys_time<Duration>& st);
```

Requires: `z` refers to a valid `time_zone`.

Effects: Constructs a `zoned_time` by initializing `zone_` with `std::move(z)` and `tp_` with `st`.

```
zoned_time<Duration, TimeZonePtr>::zoned_time(string_view name, const sys_time<Duration>& st);
```

Remarks: This constructor does not participate in overload resolution unless `zoned_time` is constructible from the return type of `zoned_traits<TimeZonePtr>::locate_zone(name)` and `st`.

Effects: Equivalent to construction with `{zoned_traits<TimeZonePtr>::locate_zone(name), st}`.

```
zoned_time<Duration, TimeZonePtr>::zoned_time(TimeZonePtr z, const local_time<Duration>& tp);
```

Requires: `z` refers to a valid `time_zone`.

Remarks: This constructor does not participate in overload resolution unless `declval<TimeZonePtr&>()->to_sys(local_time<Duration>{})` is convertible to `sys_time<duration>`.

Effects: Constructs a `zoned_time` by initializing `zone_` with `std::move(z)` and `tp_` with `zone_->to_sys(t)`.

```
zoned_time<Duration, TimeZonePtr>::zoned_time(string_view name, const local_time<Duration>& tp);
```

Remarks: This constructor does not participate in overload resolution unless `zoned_time` is constructible from the return type of `zoned_traits<TimeZonePtr>::locate_zone(name)` and `tp`.

Effects: Equivalent to construction with `{zoned_traits<TimeZonePtr>::locate_zone(name), tp}`.

```
zoned_time<Duration, TimeZonePtr>::zoned_time(TimeZonePtr z, const local_time<Duration>& tp, choose c);
```

Requires: `z` refers to a valid `time_zone`.

Remarks: This constructor does not participate in overload resolution unless `decltype(declval<TimeZonePtr&>()->to_sys(local_time<Duration>{}, choose::earliest))` is convertible to `sys_time<duration>`.

Effects: Constructs a `zoned_time` by initializing `zone_` with `std::move(z)` and `tp_` with `zone_->to_sys(t, c)`.

```
zoned_time<Duration, TimeZonePtr>::zoned_time(string_view name, const local_time<Duration>& tp, choose c);
```

Remarks: This constructor does not participate in overload resolution unless `zoned_time` is constructible from the return type of `zoned_traits<TimeZonePtr>::locate_zone(name)`, `local_time<Duration>` and `choose`.

Effects: Equivalent to construction with `{zoned_traits<TimeZonePtr>::locate_zone(name), tp, c}`.

```
template <class Duration2, TimeZonePtr>
    zoned_time<Duration, TimeZonePtr>::zoned_time(TimeZonePtr z, const zoned_time<Duration2, TimeZonePtr2>& y);
```

Remarks: Does not participate in overload resolution unless `sys_time<Duration2>` is implicitly convertible to `sys_time<Duration>`.

Requires: `z` refers to a valid time zone.

Effects: Constructs a `zoned_time` by initializing `zone_` with `std::move(z)` and `tp_` with `z.tp_`.

```
template <class Duration2, TimeZonePtr>
    zoned_time<Duration, TimeZonePtr>::zoned_time(TimeZonePtr z, const zoned_time<Duration2, TimeZonePtr2>& y,
                                                  choose);
```

Remarks: Does not participate in overload resolution unless `sys_time<Duration2>` is implicitly convertible to `sys_time<Duration>`.

Requires: `z` refers to a valid time zone.

Effects: Equivalent to construction with `{z, y}`.

Note: The `choose` parameter is allowed here, but has no impact.

```
zoned_time<Duration, TimeZonePtr>::zoned_time(string_view name, const zoned_time<Duration>& y);
```

Remarks: This constructor does not participate in overload resolution unless `zoned_time` is constructible from the return type of `zoned_traits<TimeZonePtr>::locate_zone(name)` and `zoned_time`.

Effects: Equivalent to construction with `{zoned_traits<TimeZonePtr>::locate_zone(name), y}`.

```
zoned_time<Duration, TimeZonePtr>::zoned_time(string_view name, const zoned_time<Duration>& y, choose c);
```

Remarks: This constructor does not participate in overload resolution unless `zoned_time` is constructible from the return type of `zoned_traits<TimeZonePtr>::locate_zone(name)`, `zoned_time`, and `choose`.

Effects: Equivalent to construction with `{locate_zone(name), y, c}`.

Note: The `choose` parameter is allowed here, but has no impact.

```
zoned_time<Duration, TimeZonePtr>& zoned_time<Duration, TimeZonePtr>::operator=(const sys_time<Duration>& st);
```

Effects: After assignment `get_sys_time() == st`. This assignment has no effect on the return value of `get_time_zone()`.

Returns: `*this`.

```
zoned_time<Duration, TimeZonePtr>& zoned_time<Duration, TimeZonePtr>::operator=(const local_time<Duration>& lt);
```

Effects: After assignment `get_local_time() == lt`. This assignment has no effect on the return value of `get_time_zone()`.

Returns: `*this`.

```
zoned_time<Duration, TimeZonePtr>::operator sys_time<duration>() const;
```

Returns: `get_sys_time()`.

```
explicit zoned_time<Duration, TimeZonePtr>::operator local_time<duration>() const;
```

Returns: `get_local_time()`.

```
TimeZonePtr zoned_time<Duration, TimeZonePtr>::get_time_zone() const;
```

Returns: `zone_`.

```
local_time<typename zoned_time<Duration, TimeZonePtr>::duration> zoned_time<Duration, TimeZonePtr>::get_local_time() const;
```

Returns: `zone_->to_local(tp_)`.

```
sys_time<typename zoned_time<Duration, TimeZonePtr>::duration> zoned_time<Duration, TimeZonePtr>::get_sys_time() const;
```

Returns: `tp_`.

```
sys_info zoned_time<Duration, TimeZonePtr>::get_info() const;
```

Returns: `zone_->get_info(tp_)`.

```
template <class Duration1, class Duration2, class TimeZonePtr>
bool
operator==(const zoned_time<Duration1, TimeZonePtr>& x,
           const zoned_time<Duration2, TimeZonePtr>& y);
```

Returns: `x.zone_ == y.zone_ && x.tp_ == y.tp_`.

```
template <class Duration1, class Duration2, class TimeZonePtr>
bool
operator!=(const zoned_time<Duration1, TimeZonePtr>& x,
           const zoned_time<Duration2, TimeZonePtr>& y);
```

Returns: `!(x == y)`.

```
template <class charT, class traits, class Duration, class TimeZonePtr>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os,
           const zoned_time<Duration, TimeZonePtr>& t)
```

Effects: Streams `t` to `os` using the format "%F %T %Z" and the value returned from `t.get_local_time()`.

Returns: `os`.

```
template <class charT, class traits, class Duration, class TimeZonePtr>
basic_ostream<charT, traits>&
to_stream(basic_ostream<charT, traits>& os, const charT* fmt,
          const zoned_time<Duration, TimeZonePtr>& tp);
```

Effects: First obtains a `sys_info` via `tp.get_info()` which for exposition purposes will be referred to as `info`. Then calls `to_stream(os, fmt, tp.get_local_time(), &info.abbrev, &info.offset)`.

Returns: `os`.

`make_zoned`

There exist several overloaded functions named `make_zoned` which serve as factory functions for `zoned_time<Duration>` and will deduce the correct `Duration` from the argument list. In every case the correct return type is `zoned_time<std::common_type_t<Duration, std::chrono::seconds>>`.

```
template <class Duration>
zoned_time<std::common_type_t<Duration, std::chrono::seconds>>
make_zoned(sys_time<Duration> tp)
```

Returns: `{tp}`.

```
template <class Duration>
zoned_time<std::common_type_t<Duration, std::chrono::seconds>>
make_zoned(const time_zone* zone, local_time<Duration> tp)
```

Returns: `{zone, tp}`.

```
template <class Duration>
zoned_time<std::common_type_t<Duration, std::chrono::seconds>>
make_zoned(const std::string& name, local_time<Duration> tp)
```

Returns: `{name, tp}`.

```
template <class Duration>
zoned_time<std::common_type_t<Duration, std::chrono::seconds>>
make_zoned(const time_zone* zone, local_time<Duration> tp, choose c)
```

Returns: `{zone, tp, c}`.

```
template <class Duration>
zoned_time<std::common_type_t<Duration, std::chrono::seconds>>
make_zoned(const std::string& name, local_time<Duration> tp, choose c)
```

Returns: `{name, tp, c}`.

```
template <class Duration>
zoned_time<std::common_type_t<Duration, std::chrono::seconds>>
make_zoned(const time_zone* zone, const zoned_time<Duration>& zt)
```

Returns: `{zone, zt}`.

```
template <class Duration>
zoned_time<std::common_type_t<Duration, std::chrono::seconds>>
make_zoned(const std::string& name, const zoned_time<Duration>& zt)
```

Returns: `{name, zt}`.

```
template <class Duration>
zoned_time<std::common_type_t<Duration, std::chrono::seconds>>
make_zoned(const time_zone* zone, const zoned_time<Duration>& zt, choose c)
```

Returns: `{zone, zt, c}`.

```
template <class Duration>
zoned_time<std::common_type_t<Duration, std::chrono::seconds>>
make_zoned(const std::string& name, const zoned_time<Duration>& zt, choose c)
```

Returns: `{name, zt, c}`.

```
template <class Duration>
zoned_time<std::common_type_t<Duration, std::chrono::seconds>>
make_zoned(const time_zone* zone, const sys_time<Duration>& st)
```

Returns: `{zone, st}`.

```
template <class Duration>
zoned_time<std::common_type_t<Duration, std::chrono::seconds>>
make_zoned(const std::string& name, const sys_time<Duration>& st)
```

Returns: `{name, st}`.

`utc_clock`

```
class utc_clock
{
public:
    using rep                       = a signed arithmetic type;
    using period                    = ratio<unspecified, unspecified>;
    using duration                  = std::chrono::duration<rep, period>;
    using time_point                = std::chrono::time_point<utc_clock>;
    static constexpr bool is_steady = unspecified;

    static time_point now();

    template <class Duration>
    static
    sys_time<std::common_type_t<Duration, std::chrono::seconds>>
    to_sys(const utc_time<Duration>&);

    template <class Duration>
    static
    utc_time<std::common_type_t<Duration, std::chrono::seconds>>
    from_sys(const sys_time<Duration>&);

    template <class Duration>
    static
    local_time<std::common_type_t<Duration, std::chrono::seconds>>
    to_local(const utc_time<Duration>&);

    template <class Duration>
    static
    utc_time<std::common_type_t<Duration, std::chrono::seconds>>
    from_local(const local_time<Duration>&);
};

template <class Duration>
    using utc_time = std::chrono::time_point<utc_clock, Duration>;

using utc_seconds = utc_time<std::chrono::seconds>;
```

In contrast to `sys_time` which does not take leap seconds into account, `utc_clock` and its associated `time_point`, `utc_time`, counts time, *including* leap seconds, since 1970-01-01 00:00:00 UTC. It also provides functions for converting between `utc_time` and `sys_time`. These functions consult `get_tzdb().leaps` to decide how many seconds to add/subtract in performing those conversions.

When compiled with `USE_OS_TZDB == 1`, some platforms do not support leap second information. When this is the case, `utc_clock` will not exist and `MISSING_LEAP_SECONDS == 1`.

```
static utc_clock::time_point utc_clock::now();
```

Returns: `from_sys(system_clock::now())`, or a more accurate value of `utc_time`.

```
template <typename Duration>
static
sys_time<std::common_type_t<Duration, std::chrono::seconds>>
utc_clock::to_sys(const utc_time<Duration>& u);
```

Returns: A `sys_time` `t`, such that `from_sys(t) == u` if such a mapping exists. Otherwise `u` represents a `time_point` during a leap second insertion and the last representable value of `sys_time` prior to the insertion of the leap second is returned.

```
template <typename Duration>
static
utc_time<std::common_type_t<Duration, std::chrono::seconds>>
utc_clock::from_sys(const sys_time<Duration>& t);
```

Returns: A `utc_time` `u`, such that `u.time_since_epoch() - t.time_since_epoch()` is equal to the number of leap seconds that were inserted between `t` and 1970-01-01. If `t` is ambiguous on this issue (i.e. corresponds to the date of leap second insertion), then the conversion counts that leap second as inserted.

```
template <class Duration>
static
local_time<std::common_type_t<Duration, std::chrono::seconds>>
utc_clock::to_local(const utc_time<Duration>& u);
```

Returns: `local_time<Duration>{to_sys(u).time_since_epoch()}`.

```
template <class Duration>
static
utc_time<std::common_type_t<Duration, std::chrono::seconds>>
utc_clock::from_local(const local_time<Duration>& t);
```

Returns: `from_sys(sys_time<Duration>{t.time_since_epoch()})`.

```
template <class CharT, class Traits, class Duration>
std::basic_ostream<class CharT, class Traits>&
operator<<(std::basic_ostream<class CharT, class Traits>& os, const utc_time<Duration>& t)
```

Effects: Calls `to_stream(os, "%F %T", t)`.

Returns: `os`.

```
template <class CharT, class Traits, class Duration>
std::basic_ostream<CharT, Traits>&
to_stream(std::basic_ostream<CharT, Traits>& os, const CharT* fmt,
          const utc_time<Duration>& tp);
```

Effects: Inserts `tp` into `os` using the format string `fmt` as specified by the [`to_stream` formatting flags]. Time points representing leap second insertions which format seconds will show `60` in the seconds field. If `%Z` is in the formatting string `"UTC"` will be used. If `%z` is in the formatting string `"+0000"` will be used.

Returns: `os`.

```
template <class Duration, class CharT, class Traits>
std::basic_istream<CharT, Traits>&
from_stream(std::basic_istream<CharT, Traits>& is, const CharT* fmt,
            utc_time<Duration>& tp, std::basic_string<CharT, Traits>* abbrev = nullptr,
            std::chrono::minutes* offset = nullptr);
```

Effects: Extracts `tp` from `is` using the format string `fmt` as specified by the [`from_stream` formatting flags]. If `%z` is present, the parsed offset will be subtracted from the parsed time. If `abbrev` is not equal to `nullptr`, the information parsed by `%Z` (if present) will be placed in `*abbrev`. If `offset` is not equal to `nullptr`, the information parsed by `%z` (if present) will be placed in `*offset`.

Returns: `is`.

```
template <class Duration>
std::pair<bool, std::chrono::seconds>
is_leap_second(utc_time<Duration> const& t);
```

Returns: Given a list of leap second insertion dates l_i since 1970-01-01, if `t` is in the range $[l_i, l_i + 1s)$, the first member of the `pair` has a value of `true`, otherwise `false`. The second member of the returned `pair` holds the number of leap seconds that have been inserted between `t` and 1970-01-01. If `t` represents a `time_point` prior to 1970-01-01, the value is `0s`. If `t` is in the range $[l_i, l_i + 1s)$, l_i is included in the count.

[*Example:*

```
cout << boolalpha;

auto t = clock_cast<utc_clock>(sys_days{December/31/2016} + 23h + 59min + 59s + 999ms);
auto p = is_leap_second(t);
cout << t << " : {" << p.first << ", " << p.second << "}\n";
// 2016-12-31 23:59:59.999 : {false, 26s}

t += 1ms;
p = is_leap_second(t);
cout << t << " : {" << p.first << ", " << p.second << "}\n";
// 2016-12-31 23:59:60.000 : {true, 27s}

t += 1ms;
p = is_leap_second(t);
cout << t << " : {" << p.first << ", " << p.second << "}\n";
```

```
// 2016-12-31 23:59:60.001 : {true, 27s}

t += 998ms;
p = is_leap_second(t);
cout << t << " : {" << p.first << ", " << p.second << "}\n";
// 2016-12-31 23:59:60.999 : {true, 27s}

t += 1ms;
p = is_leap_second(t);
cout << t << " : {" << p.first << ", " << p.second << "}\n";
// 2017-01-01 00:00:00.000 : {false, 27s}
```

—*end example*]

`tai_clock`

```
class tai_clock
{
public:
    using rep                       = a signed arithmetic type;
    using period                    = ratio<unspecified, unspecified>;
    using duration                  = std::chrono::duration<rep, period>;
    using time_point                = std::chrono::time_point<tai_clock>;
    static constexpr bool is_steady = unspecified;

    static time_point now();

    template <class Duration>
    static
    utc_time<std::common_type_t<Duration, std::chrono::seconds>>
    to_utc(const tai_time<Duration>&) noexcept;

    template <class Duration>
    static
    tai_time<std::common_type_t<Duration, std::chrono::seconds>>
    from_utc(const utc_time<Duration>&) noexcept;

    template <class Duration>
    static
    local_time<std::common_type_t<Duration, std::chrono::seconds>>
    to_local(const tai_time<Duration>&) noexcept;

    template <class Duration>
    static
    tai_time<std::common_type_t<Duration, std::chrono::seconds>>
    from_local(const local_time<Duration>&) noexcept;
};

template <class Duration>
    using tai_time = std::chrono::time_point<tai_clock, Duration>;

using tai_seconds = tai_time<std::chrono::seconds>;
```

`tai_time` counts physical seconds continuously like `utc_time`, but when printed out, *always* has 60 seconds per minute. It's epoch is 1958-01-01 and is offset ahead of `utc_time` by 10s in 1970-01-01. With each leap second, the offset from `utc_time` grows by another second.

When compiled with `USE_OS_TZDB == 1`, some platforms do not support leap second information. When this is the case, `tai_clock` will not exist and `MISSING_LEAP_SECONDS == 1`.

```
static tai_clock::time_point tai_clock::now();
```

Returns: `from_utc(utc_clock::now())`, or a more accurate value of `tai_time`.

```
template <class Duration>
static
utc_time<std::common_type_t<Duration, std::chrono::seconds>>
to_utc(const std::chrono::time_point<tai_clock, Duration>& t) noexcept;
```

Returns: `utc_time<common_type_t<Duration, seconds>>{t.time_since_epoch()} - 378691210s`

Note: `378691210s == sys_days{1970y/January/1} - sys_days{1958y/January/1} + 10s`

```
template <class Duration>
static
tai_time<std::common_type_t<Duration, std::chrono::seconds>>
tai_clock::from_utc(const utc_time<Duration>& t) noexcept;
```

Returns: `tai_time<common_type_t<Duration, seconds>>{t.time_since_epoch()} + 378691210s`

Note: `378691210s == sys_days{1970y/January/1} - sys_days{1958y/January/1} + 10s`

```
template <class Duration>
static
```

```
local_time<std::common_type_t<Duration, std::chrono::seconds>>
tai_clock::to_local(const tai_time<Duration>& t) noexcept;
```

Returns: `local_time<Duration>{t.time_since_epoch()} - (local_days{1970_y/January/1} - local_days{1958_y/January/1})`.

```
template <class Duration>
static
tai_time<std::common_type_t<Duration, std::chrono::seconds>>
tai_clock::from_local(const local_time<Duration>& t) noexcept;
```

Returns: `tai_time<Duration>{t.time_since_epoch()} + (local_days{1970_y/January/1} - local_days{1958_y/January/1})`.

```
template <class CharT, class Traits, class Duration>
std::basic_ostream<class CharT, class Traits>&
operator<<(std::basic_ostream<class CharT, class Traits>& os, const tai_time<Duration>& t)
```

Effects: Calls `to_stream(os, "%F %T", t)`.

Returns: `os`.

```
template <class CharT, class Traits, class Duration>
std::basic_ostream<CharT, Traits>&
to_stream(std::basic_ostream<CharT, Traits>& os, const CharT* fmt,
          const tai_time<Duration>& tp);
```

Effects: Inserts `tp` into `os` using the format string `fmt` as specified by the [`to_stream` formatting flags]. If `%Z` is in the formatting string `"TAI"` will be used. If `%z` is in the formatting string `"+0000"` will be used.

Returns: `os`.

```
template <class Duration, class CharT, class Traits>
std::basic_istream<CharT, Traits>&
from_stream(std::basic_istream<CharT, Traits>& is, const CharT* fmt,
            tai_time<Duration>& tp, std::basic_string<CharT, Traits>* abbrev = nullptr,
            std::chrono::minutes* offset = nullptr);
```

Effects: Extracts `tp` from `is` using the format string `fmt` as specified by the [`from_stream` formatting flags]. If `%z` is present, the parsed offset will be subtracted from the parsed time. If `abbrev` is not equal to `nullptr`, the information parsed by `%Z` (if present) will be placed in `*abbrev`. If `offset` is not equal to `nullptr`, the information parsed by `%z` (if present) will be placed in `*offset`.

Returns: `is`.

gps_clock

```
class gps_clock
{
public:
    using rep                       = a signed arithmetic type;
    using period                    = ratio<unspecified, unspecified>;
    using duration                  = std::chrono::duration<rep, period>;
    using time_point                = std::chrono::time_point<gps_clock>;
    static constexpr bool is_steady = unspecified;

    static time_point now();

    template <class Duration>
    static
    utc_time<std::common_type_t<Duration, std::chrono::seconds>>
    to_utc(const gps_time<Duration>&) noexcept;

    template <class Duration>
    static
    gps_time<std::common_type_t<Duration, std::chrono::seconds>>
    from_utc(const utc_time<Duration>&) noexcept;

    template <class Duration>
    static
    local_time<std::common_type_t<Duration, std::chrono::seconds>>
    to_local(const gps_time<Duration>&) noexcept;

    template <class Duration>
    static
    gps_time<std::common_type_t<Duration, std::chrono::seconds>>
    from_local(const local_time<Duration>&) noexcept;
};

template <class Duration>
    using gps_time = std::chrono::time_point<gps_clock, Duration>;
```

```
using gps_seconds = gps_time<std::chrono::seconds>;
```

`gps_time` counts physical seconds continuously like `utc_time`, but when printed out, *always* has 60 seconds per minute. It's epoch is 1980-01-06 and was equivalent to UTC at that time. If drifts ahead of UTC with each inserted leap second. It is always exactly 19s behind TAI.

When compiled with `USE_OS_TZDB == 1`, some platforms do not support leap second information. When this is the case, `gps_clock` will not exist and `MISSING_LEAP_SECONDS == 1`.

```
static gps_clock::time_point gps_clock::now() noexcept;
```

Returns: `from_utc(utc_clock::now())`, or a more accurate value of `gps_time`.

```
template <class Duration>
static
utc_time<std::common_type_t<Duration, std::chrono::seconds>>
gps_clock::to_utc(const gps_time<Duration>& t) noexcept;
```

Returns: `gps_time<common_type_t<Duration, seconds>>{t.time_since_epoch()} + 315964809s`

Note: `315964809s == sys_days{1980y/January/Sunday[1]} - sys_days{1970y/January/1} + 9s`

```
template <class Duration>
static
gps_time<std::common_type_t<Duration, std::chrono::seconds>>
gps_clock::from_utc(const utc_time<Duration>& t) noexcept;
```

Returns: `gps_time<common_type_t<Duration, seconds>>{t.time_since_epoch()} - 315964809s`

Note: `315964809s == sys_days{1980y/January/Sunday[1]} - sys_days{1970y/January/1} + 9s`

```
template <class Duration>
static
local_time<std::common_type_t<Duration, std::chrono::seconds>>
gps_clock::to_local(const gps_time<Duration>& t) noexcept;
```

Returns:
```
local_time<Duration>{t.time_since_epoch()} +
     (local_days{1980_y/January/Sunday[1]} - local_days{1970_y/January/1}).
```

```
template <class Duration>
static
gps_time<std::common_type_t<Duration, std::chrono::seconds>>
gps_clock::from_local(const local_time<Duration>& t) noexcept;
```

Returns:
```
local_time<Duration>{t.time_since_epoch()} -
     (local_days{1980_y/January/Sunday[1]} - local_days{1970_y/January/1}).
```

```
template <class CharT, class Traits, class Duration>
std::basic_ostream<class CharT, class Traits>&
operator<<(std::basic_ostream<class CharT, class Traits>& os, const gps_time<Duration>& t)
```

Effects: Calls `to_stream(os, "%F %T", t)`.

Returns: `os`.

```
template <class CharT, class Traits, class Duration>
std::basic_ostream<CharT, Traits>&
to_stream(std::basic_ostream<CharT, Traits>& os, const CharT* fmt,
          const gps_time<Duration>& tp);
```

Effects: Inserts `tp` into `os` using the format string `fmt` as specified by the [`to_stream` formatting flags]. If `%Z` is in the formatting string `"GPS"` will be used. If `%z` is in the formatting string `"+0000"` will be used.

Returns: `os`.

```
template <class Duration, class CharT, class Traits>
std::basic_istream<CharT, Traits>&
from_stream(std::basic_istream<CharT, Traits>& is, const CharT* fmt,
            gps_time<Duration>& tp, std::basic_string<CharT, Traits>* abbrev = nullptr,
            std::chrono::minutes* offset = nullptr);
```

Effects: Extracts `tp` from `is` using the format string `fmt` as specified by the [`from_stream` formatting flags]. If `%z` is present, the parsed offset will be subtracted from the parsed time. If `abbrev` is not equal to `nullptr`, the information parsed by `%Z` (if present) will be placed in `*abbrev`. If `offset` is not equal to `nullptr`, the information parsed by `%z` (if present) will be placed in `*offset`.

Returns: `is`.

[*Example:*

The following code prints out equivalent time stamps to millisecond precision for times near the 2015-06-30 leap second insertion. Note that the mapping to `sys_time` during the leap second collapses down to the last instant prior to the leap second. But the maping between UTC, TAI and GPS is all one-to-one.

```
#include "date/tz.h"
#include <iostream>

int
main()
{
    using namespace date;
    using namespace std::chrono;
    auto start = clock_cast<utc_clock>(sys_days{2015_y/July/1} - 500ms);
    auto end = start + 2s;
    for (auto utc = start; utc < end; utc += 100ms)
    {
        auto sys = clock_cast<system_clock>(utc);
        auto tai = clock_cast<tai_clock>(utc);
        auto gps = clock_cast<gps_clock>(utc);
        std::cout << format("%F %T SYS  ==  ", sys)
                  << format("%F %T %Z  ==  ", utc)
                  << format("%F %T %Z  ==  ", tai)
                  << format("%F %T %Z\n", gps);
    }
}
```

Output:

```
2015-06-30 23:59:59.500 SYS  ==  2015-06-30 23:59:59.500 UTC  ==  2015-07-01 00:00:34.500 TAI  ==  2015-07-01 00:00:15.500 GPS
2015-06-30 23:59:59.600 SYS  ==  2015-06-30 23:59:59.600 UTC  ==  2015-07-01 00:00:34.600 TAI  ==  2015-07-01 00:00:15.600 GPS
2015-06-30 23:59:59.700 SYS  ==  2015-06-30 23:59:59.700 UTC  ==  2015-07-01 00:00:34.700 TAI  ==  2015-07-01 00:00:15.700 GPS
2015-06-30 23:59:59.800 SYS  ==  2015-06-30 23:59:59.800 UTC  ==  2015-07-01 00:00:34.800 TAI  ==  2015-07-01 00:00:15.800 GPS
2015-06-30 23:59:59.900 SYS  ==  2015-06-30 23:59:59.900 UTC  ==  2015-07-01 00:00:34.900 TAI  ==  2015-07-01 00:00:15.900 GPS
2015-06-30 23:59:59.999 SYS  ==  2015-06-30 23:59:60.000 UTC  ==  2015-07-01 00:00:35.000 TAI  ==  2015-07-01 00:00:16.000 GPS
2015-06-30 23:59:59.999 SYS  ==  2015-06-30 23:59:60.100 UTC  ==  2015-07-01 00:00:35.100 TAI  ==  2015-07-01 00:00:16.100 GPS
2015-06-30 23:59:59.999 SYS  ==  2015-06-30 23:59:60.200 UTC  ==  2015-07-01 00:00:35.200 TAI  ==  2015-07-01 00:00:16.200 GPS
2015-06-30 23:59:59.999 SYS  ==  2015-06-30 23:59:60.300 UTC  ==  2015-07-01 00:00:35.300 TAI  ==  2015-07-01 00:00:16.300 GPS
2015-06-30 23:59:59.999 SYS  ==  2015-06-30 23:59:60.400 UTC  ==  2015-07-01 00:00:35.400 TAI  ==  2015-07-01 00:00:16.400 GPS
2015-06-30 23:59:59.999 SYS  ==  2015-06-30 23:59:60.500 UTC  ==  2015-07-01 00:00:35.500 TAI  ==  2015-07-01 00:00:16.500 GPS
2015-06-30 23:59:59.999 SYS  ==  2015-06-30 23:59:60.600 UTC  ==  2015-07-01 00:00:35.600 TAI  ==  2015-07-01 00:00:16.600 GPS
2015-06-30 23:59:59.999 SYS  ==  2015-06-30 23:59:60.700 UTC  ==  2015-07-01 00:00:35.700 TAI  ==  2015-07-01 00:00:16.700 GPS
2015-06-30 23:59:59.999 SYS  ==  2015-06-30 23:59:60.800 UTC  ==  2015-07-01 00:00:35.800 TAI  ==  2015-07-01 00:00:16.800 GPS
2015-06-30 23:59:59.999 SYS  ==  2015-06-30 23:59:60.900 UTC  ==  2015-07-01 00:00:35.900 TAI  ==  2015-07-01 00:00:16.900 GPS
2015-07-01 00:00:00.000 SYS  ==  2015-07-01 00:00:00.000 UTC  ==  2015-07-01 00:00:36.000 TAI  ==  2015-07-01 00:00:17.000 GPS
2015-07-01 00:00:00.100 SYS  ==  2015-07-01 00:00:00.100 UTC  ==  2015-07-01 00:00:36.100 TAI  ==  2015-07-01 00:00:17.100 GPS
2015-07-01 00:00:00.200 SYS  ==  2015-07-01 00:00:00.200 UTC  ==  2015-07-01 00:00:36.200 TAI  ==  2015-07-01 00:00:17.200 GPS
2015-07-01 00:00:00.300 SYS  ==  2015-07-01 00:00:00.300 UTC  ==  2015-07-01 00:00:36.300 TAI  ==  2015-07-01 00:00:17.300 GPS
2015-07-01 00:00:00.400 SYS  ==  2015-07-01 00:00:00.400 UTC  ==  2015-07-01 00:00:36.400 TAI  ==  2015-07-01 00:00:17.400 GPS
```

— *end example*]

`clock_cast`

```
template <class DestClock, class SourceClock>
struct clock_time_conversion
{};
```

`clock_time_conversion` serves as trait which can be used to specify how to convert `time_point<SourceClock, Duration>` to `time_point<DestClock, Duration>` via a specialization: `clock_time_conversion<DestClock, SourceClock>`. A specialization of `clock_time_conversion<DestClock, SourceClock>` shall provide a `const`-qualified `operator()` that takes a parameter of type `time_point<SourceClock, Duration>` and returns a `time_point<DestClock, some duration>` representing an equivalent point in time. A program may specialize `clock_time_conversion` if at least one of the template parameters is user-defined clock type.

Several specializations are provided by the implementation:

```
// Identity

template <typename Clock>
struct clock_time_conversion<Clock, Clock>
{
    template <class Duration>
    std::chrono::time_point<Clock, Duration>
    operator()(const std::chrono::time_point<Clock, Duration>& t) const;
};

template <class Duration>
std::chrono::time_point<Clock, Duration>
operator()(const std::chrono::time_point<Clock, Duration>& t) const;
```

Returns: `t`.

```
template <>
struct clock_time_conversion<std::chrono::system_clock, std::chrono::system_clock>
```

```
{
    template <class Duration>
    sys_time<Duration>
    operator()(const sys_time<Duration>& t) const;
};

template <class Duration>
sys_time<Duration>
operator()(const sys_time<Duration>& t) const;
```

Returns: `t`.

```
template <>
struct clock_time_conversion<utc_clock, utc_clock>
{
    template <class Duration>
    utc_time<Duration>
    operator()(const utc_time<Duration>& t) const;
};

template <class Duration>
utc_time<Duration>
operator()(const utc_time<Duration>& t) const;
```

Returns: `t`.

```
template <>
struct clock_time_conversion<local_t, local_t>
{
    template <class Duration>
    local_time<Duration>
    operator()(const local_time<Duration>& t) const;
};

template <class Duration>
local_time<Duration>
operator()(const local_time<Duration>& t) const;
```

Returns: `t`.

```
// system_clock <-> utc_clock

template <>
struct clock_time_conversion<utc_clock, std::chrono::system_clock>
{
    template <class Duration>
    utc_time<std::common_type_t<Duration, std::chrono::seconds>>
    operator()(const sys_time<Duration>& t) const;
};

template <class Duration>
utc_time<std::common_type_t<Duration, std::chrono::seconds>>
operator()(const sys_time<Duration>& t) const;
```

Returns: `utc_clock::from_sys(t)`.

```
template <>
struct clock_time_conversion<std::chrono::system_clock, utc_clock>
{
    template <class Duration>
    sys_time<std::common_type_t<Duration, std::chrono::seconds>>
    operator()(const utc_time<Duration>& t) const;
};

template <class Duration>
sys_time<std::common_type_t<Duration, std::chrono::seconds>>
operator()(const utc_time<Duration>& t) const;
```

Returns: `utc_clock::to_sys(t)`.

```
// system_clock <-> local_t

template <>
struct clock_time_conversion<local_t, std::chrono::system_clock>
{
    template <class Duration>
    local_time<std::common_type_t<Duration, std::chrono::seconds>>
    operator()(const sys_time<Duration>& t) const;
};

template <class Duration>
local_time<std::common_type_t<Duration, std::chrono::seconds>>
operator()(const sys_time<Duration>& t) const;
```

Returns: `local_time<Duration>{t.time_since_epoch()}`.

```
template <>
struct clock_time_conversion<std::chrono::system_clock, local_t>
{
    template <class Duration>
    sys_time<std::common_type_t<Duration, std::chrono::seconds>>
    operator()(const local_t<Duration>& t) const;
};

template <class Duration>
sys_time<std::common_type_t<Duration, std::chrono::seconds>>
operator()(const local_t<Duration>& t) const;
```

Returns: `sys_time<Duration>{t.time_since_epoch()}`.

```
// utc_clock <-> local_t

template <>
struct clock_time_conversion<local_t, utc_clock>
{
    template <class Duration>
    local_time<std::common_type_t<Duration, std::chrono::seconds>>
    operator()(const utc_time<Duration>& t) const;
};

template <class Duration>
local_time<std::common_type_t<Duration, std::chrono::seconds>>
operator()(const utc_time<Duration>& t) const;
```

Returns: `utc_clock::to_local(t)`.

```
template <>
struct clock_time_conversion<utc_clock, local_t>
{
    template <class Duration>
    utc_time<std::common_type_t<Duration, std::chrono::seconds>>
    operator()(const local_t<Duration>& t) const;
};

template <class Duration>
utc_time<std::common_type_t<Duration, std::chrono::seconds>>
operator()(const local_t<Duration>& t) const;
```

Returns: `utc_clock::from_local(t)`.

```
// Clock <-> system_clock

template <class SourceClock>
struct clock_time_conversion<std::chrono::system_clock, SourceClock>
{
    template <class Duration>
    auto
    operator()(const std::chrono::time_point<SourceClock, Duration>& t) const
        -> decltype(SourceClock::to_sys(t));
};

template <class Duration>
auto
operator()(const std::chrono::time_point<SourceClock, Duration>& t) const
    -> decltype(SourceClock::to_sys(t));
```

Remarks: This function does not participate in overload resolution unless `SourceClock::to_sys(t)` is well formed. If `SourceClock::to_sys(t)` does not return `sys_time<`*`some duration`*`>` the program is ill-formed.

Returns: `SourceClock::to_sys(t)`.

```
template <class DestClock>
struct clock_time_conversion<DestClock, std::chrono::system_clock>
{
    template <class Duration>
    auto
    operator()(const sys_time<Duration>& t) const
        -> decltype(DestClock::from_sys(t));
};

template <class Duration>
auto
operator()(const sys_time<Duration>& t) const
    -> decltype(DestClock::from_sys(t));
```

Remarks: This function does not participate in overload resolution unless `DestClock::from_sys(t)` is well formed. If `DestClock::from_sys(t)` does not return `time_point<DestClock, `*`some duration`*`>` the program is ill-formed.

Returns: `DestClock::from_sys(t)`.

```
// Clock <-> utc_clock

template <class SourceClock>
struct clock_time_conversion<utc_clock, SourceClock>
{
    template <class Duration>
    auto
    operator()(const std::chrono::time_point<SourceClock, Duration>& t) const
        -> decltype(SourceClock::to_utc(t));
};
```

```
template <class Duration>
auto
operator()(const std::chrono::time_point<SourceClock, Duration>& t) const
    -> decltype(SourceClock::to_utc(t));
```

Remarks: This function does not participate in overload resolution unless `SourceClock::to_utc(t)` is well formed. If `SourceClock::to_utc(t)` does not return `utc_time<some duration>` the program is ill-formed.

Returns: `SourceClock::to_utc(t)`.

```
template <class DestClock>
struct clock_time_conversion<DestClock, utc_clock>
{
    template <class Duration>
    auto
    operator()(const utc_time<Duration>& t) const
        -> decltype(DestClock::from_utc(t));
};
```

```
template <class Duration>
auto
operator()(const utc_time<Duration>& t) const
    -> decltype(DestClock::from_utc(t));
```

Remarks: This function does not participate in overload resolution unless `DestClock::from_utc(t)` is well formed. If `DestClock::from_utc(t)` does not return `time_point<DestClock, some duration>` the program is ill-formed. .

Returns: `DestClock::from_utc(t)`.

```
// Clock <-> local_t

template <class SourceClock>
struct clock_time_conversion<local_t, SourceClock>
{
    template <class Duration>
    auto
    operator()(const std::chrono::time_point<SourceClock, Duration>& t) const
        -> decltype(SourceClock::to_local(t));
};
```

```
template <class Duration>
auto
operator()(const std::chrono::time_point<SourceClock, Duration>& t) const
    -> decltype(SourceClock::to_local(t));
```

Remarks: This function does not participate in overload resolution unless `SourceClock::to_local(t)` is well formed. If `SourceClock::to_local(t)` does not return `local_time<some duration>` the program is ill-formed.

Returns: `SourceClock::to_local(t)`.

```
template <class Duration>
auto
operator()(const local_time<Duration>& t) const
    -> decltype(DestClock::from_local(t));
```

Remarks: This function does not participate in overload resolution unless `DestClock::from_local(t)` is well formed. If `DestClock::from_local(t)` does not return `time_point<DestClock, some duration>` the program is ill-formed. .

Returns: `DestClock::from_local(t)`.

```
// clock_cast

template <class DestClock, class SourceClock, class Duration>
std::chrono::time_point<DestClock, some duration>
clock_cast(const std::chrono::time_point<SourceClock, Duration>& t);
```

Remarks: This function does not participate in overload resolution unless at least one of the following expressions are well formed:

1. `clock_time_conversion<DestClock, SourceClock>{}(t)`
2. Exactly one of:

- `clock_time_conversion<DestClock, system_clock>{}(`
  `clock_time_conversion<system_clock, SourceClock>{}(t))`
- `clock_time_conversion<DestClock, utc_clock>{}(`
  `clock_time_conversion<utc_clock, SourceClock>{}(t))`

3. Exactly one of:
   - `clock_time_conversion<DestClock, utc_clock>{}(`
     `clock_time_conversion<utc_clock, system_clock>{}(`
     `clock_time_conversion<system_clock, SourceClock>{}(t)))`
   - `clock_time_conversion<DestClock, system_clock>{}(`
     `clock_time_conversion<system_clock, utc_clock>{}(`
     `clock_time_conversion<utc_clock, SourceClock>{}(t)))`

Returns: The first expression in the above list that is well-formed. If item 1 is not well-formed and both expressions in item 2 are well-formed, the `clock_cast` is ambiguous (ill-formed). If items 1 and 2 are not well-formed and both expressions in item 3 are well-formed, the `clock_cast` is ambiguous (ill-formed).

leap

```
class leap
{
public:
    leap(const leap&)            = default;
    leap& operator=(const leap&) = default;

    // Undocumented constructors

    sys_seconds date() const;
};

bool operator==(const leap& x, const leap& y);
bool operator!=(const leap& x, const leap& y);
bool operator< (const leap& x, const leap& y);
bool operator> (const leap& x, const leap& y);
bool operator<=(const leap& x, const leap& y);
bool operator>=(const leap& x, const leap& y);

template <class Duration> bool operator==(const const leap&         x, const sys_time<Duration>& y);
template <class Duration> bool operator==(const sys_time<Duration>& x, const leap&               y);
template <class Duration> bool operator!=(const leap&               x, const sys_time<Duration>& y);
template <class Duration> bool operator!=(const sys_time<Duration>& x, const leap&               y);
template <class Duration> bool operator< (const leap&               x, const sys_time<Duration>& y);
template <class Duration> bool operator< (const sys_time<Duration>& x, const leap&               y);
template <class Duration> bool operator> (const leap&               x, const sys_time<Duration>& y);
template <class Duration> bool operator> (const sys_time<Duration>& x, const leap&               y);
template <class Duration> bool operator<=(const leap&               x, const sys_time<Duration>& y);
template <class Duration> bool operator<=(const sys_time<Duration>& x, const leap&               y);
template <class Duration> bool operator>=(const leap&               x, const sys_time<Duration>& y);
template <class Duration> bool operator>=(const sys_time<Duration>& x, const leap&               y);
```

`leap` is a copyable class that is constructed and stored in the time zone database when initialized. You can explicitly convert it to a `sys_seconds` with the member function `date()` and that will be the date of the leap second insertion. `leap` is equality and less-than comparable, both with itself, and with `sys_time<Duration>`.

When compiled with `USE_OS_TZDB == 1`, some platforms do not support leap second information. When this is the case, `leap` will not exist and `MISSING_LEAP_SECONDS == 1`.

link

```
class link
{
public:
    link(const link&)            = default;
    link& operator=(const link&) = default;

    // Undocumented constructors

    const std::string& name()   const;
    const std::string& target() const;
};

bool operator==(const link& x, const link& y);
bool operator!=(const link& x, const link& y);
bool operator< (const link& x, const link& y);
bool operator> (const link& x, const link& y);
bool operator<=(const link& x, const link& y);
bool operator>=(const link& x, const link& y);
```

A `link` is an alternative name for a `time_zone`. The alternative name is `name()`. The name of the `time_zone` for which this is an alternative name is `target()`. `link`s will be constructed for you when the time zone database is initialized.

When compiled with `USE_OS_TZDB == 1`, `link` will not exist.

Installation

You will need the following four source files: `date.h`, `tz.h`, `tz_private.h` and `tz.cpp`. These sources are located at the github repository https://github.com/HowardHinnant/date.

Compile `tz.cpp` along with your other sources while providing pointers to your compiler for the location of the header files (i.e. `tz.h`).

You can also customize the build by defining macros (e.g. on the command line) as follows:

`INSTALL`

This is the location of your uncompressed IANA Time Zone Database -- tzdataYYYYv.tar.gz (or where you want the software to install it for you if you compile with `AUTO_DOWNLOAD == 1`).

If specified, `"/tzdata"` will be appended to whatever you supply (`"\tzdata"` on Windows).

Default: `"~/Downloads/tzdata"` (`"%homedrive%\%homepath%\downloads\tzdata"` on Windows).

Example: Put the database in the current directory:

```
-DINSTALL=.
```

Warning: When coupled with `AUTO_DOWNLOAD=1`, this *will* overwrite everthing at `INSTALL/tzdata` if it already exists. Set with care.

When compiling with `USE_OS_TZDB == 1` `INSTALL` can not be used. The zic-compiled time zone database will be wherever your OS installed it.

`HAS_REMOTE_API`

If `HAS_REMOTE_API` is 1 then the remote API exists, else it doesn't:

```
std::string remote_version();
bool        remote_download(const std::string& version);
bool        remote_install(const std::string& version);
```

The remote API requires linking against `libcurl` (https://curl.haxx.se/libcurl). On macOS and Linux this is done with `-lcurl`. `libcurl` comes pre-installed on macOS and Linux, but not on Windows. However one can download it for Windows.

Default: `1` on Linux and macOS, `0` on Windows.

Example: Disable the remote API:

```
-DHAS_REMOTE_API=0
```

When compiling with `USE_OS_TZDB == 1` `HAS_REMOTE_API` can not be enabled. You will be using whatever zic-compiled database your OS supplies.

`AUTO_DOWNLOAD`

If `AUTO_DOWNLOAD` is `1` then first access to the timezone database will install it if it hasn't been installed, and if it has, will use the remote API to install the latest version if not already installed.

If `AUTO_DOWNLOAD` is not enabled, you are responsible for keeping your IANA Time Zone Database up to date. New versions of it are released several times a year. This library is not bundled with a specific version of the database already installed, nor is any specific version of the database blessed.

If `AUTO_DOWNLOAD` is `1` then `HAS_REMOTE_API` must be `1`, else a compile-time error will be emitted.

Default: Equal to `HAS_REMOTE_API`.

Example: Disable automatic downloading of the timezone database:

```
-DAUTO_DOWNLOAD=0
```

Warning: This *will* overwrite everthing at `INSTALL/tzdata` if it already exists.

When compiling with `USE_OS_TZDB == 1` `AUTO_DOWNLOAD` can not be enabled. You will be using whatever zic-compiled database your OS supplies.

`USE_OS_TZDB`

If `USE_OS_TZDB` is `1` then this library will use the zic-compiled time zone database provided by your OS. This option relieves you of having to install the IANA time zone database, either manually, or automatically with `AUTO_DOWNLOAD`. This option is not available on Windows.

The OS-supplied database may contain a subset of the information available when using the IANA text-file database. For example on Apple platforms there is no leap second data available, and the time zone transition points are limited to the range of 1901-12-13

20:45:52 to 2038-01-19 03:14:07. The library continues to give results outside this range, but the offsets may not be the same as when using the text-file IANA database. In extreme cases, the reported local time can be off by nearly a day (e.g. America/Adak, prior to 1867-10-17). This is caused (for example) by some regions "jumping" the international date line in the 1800's. Additionally the IANA time zone database version may not be available. If unavailable, the version will be "unknown".

Default: `0`.

Example: Enable the use of the OS-supplied time zone database:

```
-DUSE_OS_TZDB=1
```

`USE_SHELL_API`

If `USE_SHELL_API` is `1` then `std::system` is used to execute commands for downloading the timezone database. This may be useful (for example) if your `tar` utility is installed in some place other than `/usr/bin/tar`.

If `USE_SHELL_API` is `0` then `fork` is used to execute commands for downloading the timezone database (`CreateProcess` on Windows).

Default: `1`.

Example: Enable the use of the shell API:

```
-DUSE_SHELL_API=1
```

Example compile command I commonly use on macOS:

```
clang++ -std=c++14 test.cpp -I../date/include ../date/src/tz.cpp -O3 -lcurl
```

Windows specific:

If you want to enable `HAS_REMOTE_API` and/or `AUTO_DOWNLOAD` on Windows you will have to manually install [curl](https://curl.haxx.se) and [7-zip](http://www.7-zip.org) into their default locations.

If you do not enable `HAS_REMOTE_API`, you will need to also install https://raw.githubusercontent.com/unicode-org/cldr/master/common/supplemental/windowsZones.xml into your `install` location. This will be done for you if you have enabled `HAS_REMOTE_API` and let `AUTO_DOWNLOAD` default to 1.

When `_WIN32` is defined the library assumes that you are consuming the library from another DLL and defines `#define DATE_API __declspec(dllimport)`. If you are including the cpp file directly into your project you can just define `DATE_BUILD_LIB` for not using any of the dllimport/dllexport definitions.

Define `NOMINMAX` to disable the Windows `min` and `max` macros.

mingw users: `-lpthread` and `-lole32` are required.

Linux specific:

You may have to use `-lpthread`. If you're getting a mysterious crash on first access to the database, it is likely that you aren't linking to the `pthread` library. The `pthread` library is used to assure that that library initialization is thread safe in a multithreaded environment.

iOS specific:

In addition to four aforementioned source files you will need following files: `ios.h`, `ios.cpp`.

In Xcode in *[Your Target]->Build Settings->Other C Flags* set following flags: `-DHAS_REMOTE_API=0`, `-DUSE_SHELL_API`, `-x objective-c++`.

Also you have to add IANA database archive (`*.tar.gz`) manually to your project, automatic download for iOS is not supported, this archive will be unpacked automatically into subdirectory *Library/tzdata* of installed application.

Acknowledgements

A database parser is nothing without its database. I would like to thank the founding contributor of the IANA Time Zone Database Arthur David Olson. I would also like to thank the entire group of people who continually maintain it, and especially the IESG-designated TZ Coordinator, Paul Eggert. Without the work of these people, this software would have no data to parse.

I would also like to thank Jiangang Zhuang and Bjarne Stroustrup for invaluable feedback for the timezone portion of this library, which ended up also influencing the date.h library.

And I would also especially like to thank contributors to this library: gmcode, Ivan Pizhenko, Tomasz Kamiński, tomy2105 and Ville Voutilainen.