C++20's <chrono> Calendars and Time Zones in MSVC

MIYA NATSUHARA

Cppcon
The C++ Conference
2021
October 24-29

C++20's <chrono> Calendars and Time Zones in MSVC

Miya Natsuhara ("MEE-yuh Not-soo-HAR-uh")
Miya.Natsuhara@microsoft.com
Software Engineer, Visual C++ Libraries

Welcome to CppCon 2021!

Join #visual_studio channel on CppCon Discord
https://aka.ms/cppcon/discord

- Meet the Microsoft C++ team
- Ask any questions
- Discuss the latest announcements

Take our survey
https://aka.ms/cppcon

Before we begin...

- Please use the Q&A tab for questions during the session (typed)
  - If you want to raise your hand to ask a question, please hold them until after the session is over.
- When asking questions,
  please reference slide numbers

About Me:

- Miya Natsuhara
- Software Engineer on the Visual C++ Libraries Team (Microsoft)
  - Started in April 2020
- Lecturer at the University of Washington
- Cute pup parent

Agenda

- Part I: Development Process
  - Implementing in the Open
  - Organization of Work
- Part II: Calendrical Types
- Part III: Clocks
- Part IV: Leap Seconds
  - What are leap seconds?
  - How did we implement them in MSVC?
- Part V: Time Zones
  - Overview of [time.zone]
  - IANA database challenges
- Part VI: Conclusion

Part I: Development Process

Development Process

- Implemented in our microsoft/STL open-source GitHub repo with the help of our amazing contributors!
  - In particular, thanks to statementreply, Matt Stephanson, and Daniel Marshall

- A GitHub project tracking the issues, PRs, discussions, etc. related to C++20 chrono: Extensions to <chrono> (github.com)

- Feature branch (feature/chrono) for rapid development and collaboration

- Code Review Videos!
  - clocks, clock_cast, leap_seconds (https://youtu.be/WX3OmVu4lAs)
  - time_zone and time_zone_link (https://youtu.be/MODhhr7m-5s)
  - system_clock::now(), file_clock, leap second awareness (https://youtu.be/c7DT28TV0AY)

Part II: Calendrical Types

Calendrical Types ([time.cal])

- Lots of new class types
    - `chrono::day`
    - `chrono::month`
    - `chrono::year`
    - `chrono::month_day`
    - `chrono::weekday`
    - `chrono::weekday_indexed`
    - `chrono::year_month_day`
    - `chrono::year_month_day_last`
    - `chrono::year_month_weekday_last`
    - …

Some examples: Simple Calendrical Types

```cpp
#include <chrono>
#include <iostream>

using namespace std::chrono;
int main() {
    year y{2021};
    std::cout << y << "\n";

    month m{October};
    auto result = m + months{3};
    std::cout << result << "\n";

}
```

Output:

2021

Jan

Some examples: Simple Calendrical Types (continued)

```cpp
#include <chrono>
#include <iostream>

using namespace std::chrono;

int main() {
    weekday wd{Thursday};
    auto result = wd + days{4};
    std::cout << result << "\n";

    weekday sun1{0};
    weekday sun2{7};
    std::cout << sun1 << "\t" << sun2 << "\n";

    weekday_indexed wdi{wd, 4}; // fourth Thursday
    std::cout << wdi << "\n";
}
```

Output:

```
Mon
Sun	Sun
Thu[4]
```

Some examples: Compound Calendrical Types

```cpp
#include <chrono>
#include <iostream>

using namespace std::chrono;

int main() {
    year this_year{2021};
    year last_year{2020};

    year_month_day ymd{this_year, October, day{28}};
    std::cout << ymd << "\n";

    month_weekday mwd{November, Thursday[4]};
    std::cout << mwd << "\n";

    month_day_last mdlast{February};

    year_month_day_last ymdlast_leap{last_year, mdlast};
    year_month_day_last ymdlast_noleap{this_year, mdlast};

    std::cout << ymdlast_leap << "\t" << ymdlast_leap.day() << "\n";
    std::cout << ymdlast_noleap << "\t" << ymdlast_noleap.day() << "\n";
}
```

Output:

```
2021-10-28
Nov/Thu[4]
2020/Feb/last	29
2021/Feb/last	28
```

Some examples: **operator/** with new chrono literals

```
#include <chrono>
#include <iostream>

using namespace std::chrono;

int main() {
    year_month_day ymd1{October/28d/2021y};
    year_month_day ymd2{2021y/October/28d};
    std::cout << ymd1 << "\t" << ymd2 << "\n";

    year_month_day_last
        ymdlast{2021y/(October + months{1})/last};
    std::cout << ymdlast << "\n";
    month_day md{October/28};
    std::cout << md << "\n";
    month_weekday mwd{March/Wednesday[2]};
    std::cout << mwd << "\n";
}
```

Output:

```
2021-10-28        2021-10-28
2021/Nov/last
Oct/28
Mar/Wed[2]
```

Full example

```cpp
#include <chrono>
#include <iostream>
using namespace std::chrono;

int main()
{
    std::cout << "Patch Tuesdays in 2021:\n";
    year target_year{2021};
    for (int mo = 1; mo <= 12; ++mo) {
         // the second Tuesday of each month
        year_month_weekday patch_tues{mo/Tuesday[2]/target_year};
        year_month_day ymd{sys_days{patch_tues}};
        std::cout << ymd.month() << " " << ymd.day() << "\n";
    }
}
```

Output:

```
Patch Tuesdays in 2021:
Jan 12
Feb 09
Mar 09
Apr 13
May 11
Jun 08
Jul 13
Aug 10
Sep 14
Oct 12
Nov 09
Dec 14
```

Why calendrical types? Type Safety

The "simple" calendrical types (e.g., `day`, `month`, `year`) are very straightforward – what's the point in creating types for these concepts at all? Couldn't we just use `unsigned int` for each?

Consider our `year_month_day` example using `operator/`:

```
year_month_day ymd1{October/28d/2021y};
year_month_day ymd2{2021y/October/28d};
```

Without the `day` and `year` types, these would look like

```
year_month_day ymd1{October/28/2021};
year_month_day ymd2{2021/October/28};
```

It would be very easy to accidentally interpret 2021 as the day and 28 as the year in the implementation! Have these "simple" types improves type safety as we can make sure we are interpreting these values as the appropriate type.

Why calendrical types? Abstraction

The “simple” calendrical types (e.g., `day`, `month`, `year`) are very straightforward – what’s the point in creating types for these concepts at all? Couldn’t we just use `unsigned int` for each?

While you *could* just represent these values with `unsigned int`s or the underlying data, these new calendrical types also provide valuable *abstraction*.

To use `weekday`s, `month`s, `weekday_indexed`s, etc. effectively, we don’t need to know or understand the details of the wrap-around math, formatting details, etc. We can just use them and be clients of them!

Plus, the types often end up boiling down to constants anyway!

Part III: Clocks

What is a clock?

For a type T to qualify as a Clock, it must satisfy each of the following conditions:

- The following qualified-ids must be valid and denote a type
  - `T::rep`
  - `T::period`
  - `T::duration`
  - `T::time_point`
- The following expressions must be well-formed when treated as an unevaluated operand
  - `T::is_steady`
  - `T::now()`

The newly-added trait `is_clock` detects whether a given type satisfies the Clock requirements.

```cpp
#include <chrono>
#include <ratio>

using namespace std::chrono;

class MyClock {
public:
    using rep = long long;
    using period = std::milli; // millisecond granularity
    using duration = milliseconds;
    using time_point = time_point<MyClock>;

    static constexpr bool is_steady = false;

    static time_point now() {}
};

static_assert(is_clock_v<MyClock>);
```

C++20’s chrono adds several new clocks ([time.clock])

Existing clocks (pre-C++20)

- `system_clock`

- `(file_clock)`

New clocks (in C++20)

- `utc_clock`
- `tai_clock`
- `gps_clock`

- `(file_clock)`

C++20's chrono adds several new clocks ([time.clock])

Existing clocks (pre-C++20)

- `system_clock`

- `(file_clock)`

New clocks (in C++20)

- **`utc_clock`**
- `tai_clock`
- `gps_clock`

- `(file_clock)`

Why have all these different clocks?

Clock	Epoch	General Description	Tracks leap seconds?
`system_clock`	Jan 1, 1970 00:00:00	Tracks UTC or GMT* time.	No
`utc_clock`	Jan 1, 1970 00:00:00	Tracks UTC or GMT* time.	Yes
`tai_clock`	Jan 1, 1958 00:00:00	Tracks International Atomic Time which uses a weighted average of many atomic clocks to track time.	No
`gps_clock`	First Sunday of Jan, 1980 (Jan 6, 1980 00:00:00)	Tracks the time maintained by the GPS satellites' atomic clocks.	No
`file_clock`	*Unspecified* Typically: Jan 1, 1970 on POSIX; Jan 1, 1601 on Windows	Used to create the `time_point` system used for `file_time_type`	*Unspecified*

* Coordinated Universal Time (UTC) or Greenwich Mean Time (GMT)

`local_t`...the pseudo clock

- `local_t` sort of acts like a clock, but isn't one...it's a "pseudo clock"!
  - e.g., `is_clock_v<local_t>` is `false`
  - `local_t` has no member `now()`, so it doesn't meet the clock requirements

- It is meant to represent a local time with respect to a not-yet-specified time zone
  - More to come on this when we get to time zones and related matters soon!

clock_cast

clock_cast allows you to convert a time_point for one clock to an equivalent time_point for another clock.

This conversion takes the epochs of the source and destination clock into account as well as whether each clock keeps track of leap seconds.

Internally, uses clock_time_conversions which each use to_utc, from_utc, to_sys, and from_sys as needed.

Part IV: Leap Seconds

What is a leap second?

<u>Definition</u>: a one-second time adjustment that is occasionally applied to UTC time to accommodate discrepancies that develop between precise time and observed solar time.

Year	June 30	Dec 31	Year	June 30	Dec 31
1972	+1	+1	1989		+1
1973		+1	1990		+1
1974		+1	1992	+1	
1975		+1	1993	+1	
1976		+1	1994	+1	
1977		+1	1995		+1
1978		+1	1997	+1	
1979		+1	1998		+1
1981	+1		2005		+1
1982	+1		2008		+1
1983	+1		2012	+1	
1985	+1		2015	+1	
1987		+1	2016		+1

Types of leap seconds

<u>Positive leap seconds</u>:

An extra second is inserted, so the seconds on a UTC [illegible] **60**, 00.

This is the only typ[illegible] has occurred so fa[illegible] table).

<u>Negative leap seconds:</u>

A second is *removed*, so the seconds on a UTC clock might read 57, 58, 00 (skipping over second 59).

No leap second removals have occurred yet, but they theoretically could.

Year	June 30	Dec 31	Year	June 30	Dec 31
1972	+1	+1	1989		+1
1973		+1	1990		+1
1974		+1	1992	+1	
1975		+1	1993	+1	
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	+1
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	+1
1981	+1		2005		+1
1982	+1		2008		+1
1983	+1		2012	+1	
1985	+1		2015	+1	
1987		+1	2016		+1

Back to Part III: Clocks

(just for a moment)

clock_cast example

```
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    tai_time tai_now = tai_clock::now();
    std::cout << tai_now << "\n";

    utc_time utc_now = clock_cast<utc_clock>(tai_now);
    std::cout << utc_now << "\n";
}
```

Output:

```
2021-10-19 01:13:06.2810034
2021-10-19 01:12:29.2810034
```

Here there is a 37 second discrepancy…
This is because:

- The difference between the `tai_clock` and `utc_clock` epochs means that there is an initial discrepancy of 10s
- `utc_clock` tracks leap seconds while `tai_clock` does not, and 27 leap seconds that have elapsed so far.
- So there is a 37s difference total, accounting for initial discrepancy and leap seconds

Back to Part IV: Leap Seconds

(sorry for the detour)

The leap second story in MSVC…

It’s complicated.

The leap second story in MSVC…

- Previously, the Windows operating system did not keep track of leap seconds.
  - Leap seconds were not tracked individually but instead when the OS synchronized its time at the next interval, it would notice that it was 1 second behind and make the adjustment then.

- BUT as of Windows 10 October 2018 update and Windows Server 2019, Windows OS will now track leap seconds!
  - HOWEVER, they will not track leap seconds prior to 2018.

The leap second story in MSVC…how we implement it

- We read post-2018 leap seconds from a Windows registry
  - `SYSTEM\CurrentControlSet\Control\LeapSecondInformation`
- For pre-2018 leap seconds, we maintain a static constexpr table to pull data from.
- Note that we don't currently have a way to detect upcoming leap seconds on older Windows OSes (before the 2018 October update).
  - However, because leap seconds happen infrequently, we plan to update this static table periodically so older OSes can still detect more recent leap seconds (they will just need to update the libraries).

The leap second story in MSVC…(sources)

- Leap Seconds for the IT Pro: What you need to know - Microsoft Tech Community by Dan Cuomo

- Leap Seconds for the AppDev: What you should know - Microsoft Tech Community by Daniel Havey

Part V: Time Zones

Time Zones ([time.zone])

- chrono now includes an interface for accessing the [IANA time zone database].
- This functionality requires several new types to be added to the library:
  - `tzdb`
  - `tzdb_list`
  - `time_zone`
  - `zoned_time`
  - `time_zone_link`
  - `ambiguous_local_time`
  - `nonexistent_local_time`
  - …

time_zone

- Represents a “time zone” and all time zone transitions for a specific geographic area (e.g., America/New York).

- Stores its name as well as details like the UTC offset for that time zone, whether it is during “daylight saving”, etc.

`zoned_time`

- Represents a pairing of a `time_zone` and a `time_point` (so a specific point in time in the context of a particular time zone).

- Can produce the `local_time` or equivalent `sys_time`

sys_time and local_time conversions

- NOTE:
  - ```
    template<class Duration>
      using sys_time = time_point<system_clock, Duration>;
    ```
  - ```
    template<class Duration>
      using local_time  = time_point<local_t, Duration>;
    ```

Conversions between `sys_time` and `local_time` are usually uncontroversial, but there are a few cases where things get tricky!

(Spoiler alert: it's all Daylight Saving Time's fault)

`sys_time` and `local_time` conversions

It's possible for a conversion from a `local_time` to a `sys_time` to throw two different exceptions:

- `ambiguous_local_time`
  - Consider a `local_time` that occurs during a daylight saving time transition when an extra hour is inserted. When that happens, that one-hour block essentially happens twice.
  - If the `local_time` to be converted occurs in that period of time, there are two potential `sys_time`s that it could be converted to.
  - If the choice of earlier or later time isn't specified (through `choose::earliest` or `choose::latest`), an `ambiguous_local_time` exception is thrown.
- `nonexistent_local_time`
  - Consider a `local_time` that occurs during a daylight saving time transition when an hour is lost.
  - If the `local_time` to be converted occurs in that period of time, there is no `sys_time` that it can correspond to so a `nonexistent_local_time` exception is thrown.

`ambiguous_local_time` example

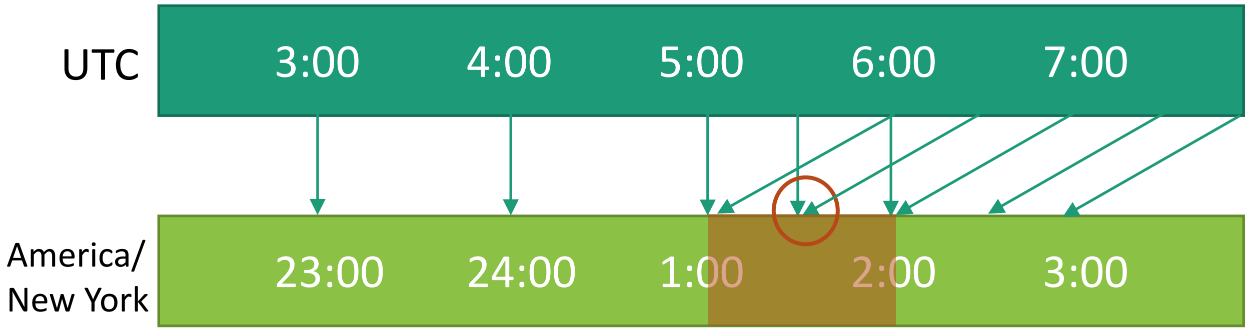
```cpp
#include <chrono>
#include <iostream>

using namespace std::chrono;

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

Output:

```
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
```



Note that the US Fall Daylight Saving Transition in 2016 occurred on Sunday Nov 6, 2016 at 2:00am. At this time, clocks were turned **backward** 1 hour to Sunday November 6, 2016 at 1:00am standard time instead.

So, the `time_point` Nov 6, 2016 1:30am essentially occurred twice during that early morning!

nonexistent_local_time example

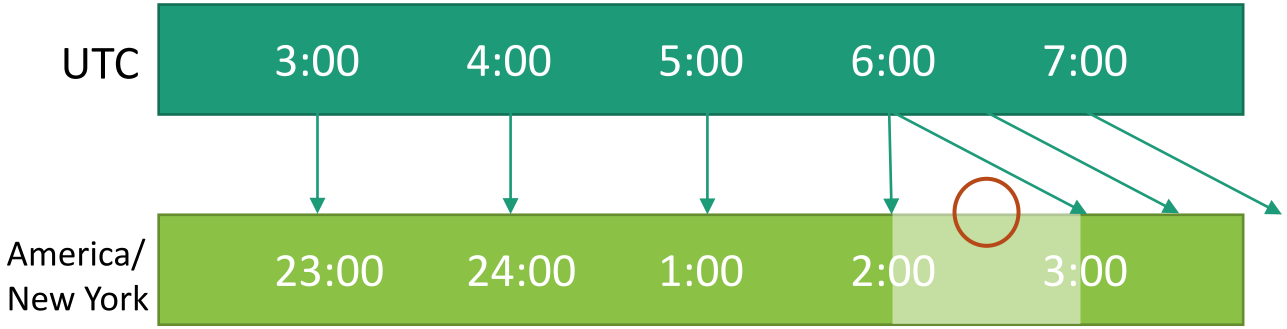
```
#include <chrono>
#include <iostream>

using namespace std::chrono;

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

Output:

```
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
```



Note that the US Spring Daylight Saving Transition in 2016 occurred on Sunday March 13, 2016 at 2:00am. At this time, clocks were turned **forward** 1 hour to Sunday March 13, 2016 at 3:00am standard time instead.

So, the `time_point` March 13, 2016 2:30am did not exist during that early morning!

tzdb

- This is a type that stores data from the time zone database.
- Specifically, it contains data members:
  - `string                    version;`
  - `vector<time_zone>         zones;`
  - `vector<time_zone_link> links;`
  - `vector<leap_second>       leap_seconds;`
- And has member functions:
  - `const time_zone* locate_zone(string_view tz_name) const;`
  - `const time_zone* current_zone() const;`

Getting a `tzdb`

- There also exists a type called `tzdb_list` which contains a list of `tzdb` objects.
  - Since `tzdb` objects can be tied to newer versions of the time zone data, this list can contain various versions of the `tzdb`.

- The `tzdb_list` is a singleton, and should be accessed via the `get_tzdb_list()` function. Then you can access individual `tzdb`s from the list using the `tzdb_list::const_iterator`.
  - Alternatively, you can just use the `get_tzdb()` function which just returns the `front()` of the `tzdb_list`.

tzdb example

```
#include <algorithm>
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    const auto& db = get_tzdb();

    std::cout << "Time Zone descriptions:\n";
    std::for_each(db.zones.begin(), db.zones.end(),
        [](const time_zone& z)
        {
            std::cout << "Zone: " << z.name() << "\n";
        });
}
```

Output

```
Time Zone descriptions:
Zone: Africa/Abidjan
Zone: Africa/Accra
Zone: Africa/Addis_Ababa
Zone: Africa/Algiers
Zone: Africa/Asmera
Zone: Africa/Bamako
Zone: Africa/Bangui
Zone: Africa/Banjul
Zone: Africa/Bissau
Zone: Africa/Blantyre
Zone: Africa/Brazzaville
Zone: Africa/Bujumbura
…
```

Where does this data come from?

- The Standard describes these facilities as interfacing with IANA database.
- The IANA tz database is actually shipped with Unix-like systems – the standard path of the data is /usr/share/zoneinfo
  - This is the case for Linux distributions, macOS, and some other Unix-like systems
- But for Windows? Not so much...

The MSVC story for time zone data

We had a few options...

- Ship the entire IANA time zone database with the library
- Try to figure out some networking scenario to pull in the data when needed
- Find an alternate source for the data

The MSVC story for time zone data

We had a few options...

- Ship the entire IANA time zone database with the library

The IANA time zone database is huge (1.25MB) – we would have some very unhappy customers if their MSVC STL grew by such a large amount. PLUS, how would we handle updates to the data?

- Try to figure out some networking scenario to pull in the data when needed

The STL currently does not have parts requiring networking and we would really like to avoid adding that requirement...

- Find an alternate source for the data

This is where we landed!

The MSVC story for time zone data

After doing some research, we found the ICU library which ships as part of the Windows 10 operating system! *

The ICU data is not *equivalent* to the IANA database, but it is derived from the same IANA data.

We'll be able to receive updates to the time zone data through OS updates through Windows Update!

* Only available in more recent Windows OSes (19H1 and after)

Differences between IANA and ICU time zone data

IANA

- Defines "standard" and "alternate" names for time zones.
- Has a few older, somewhat abnormal time zones such as EWT (Eastern War Time) which existed between 1942 and 1945.
- Contains the time zone America/Nuuk, relatively newly renamed from America/Godthab

ICU

- ICU defines "canonical" and "non-canonical" names for time zones.
- Contains the recently outdated name America/Godthab instead of the new name America/Nuuk

Unfortunately, there is no easy mapping between these two different classification systems.
So, MSVC STL considers all ICU time zones (canonical and non-canonical) to be `time_zones`.

Part VI: Conclusion

Takeaways

- C++20 `<chrono>` is super cool and has a ton of new stuff!
- The design of the library presented some challenges for Windows, but we made it work!
- Time is so much more complicated than you ever could have imagined :)

Questions?

Enjoy the rest of the conference!

Join #visual_studio channel on CppCon Discord
https://aka.ms/cppcon/discord

- Meet the Microsoft C++ team
- Ask any questions
- Discuss the latest announcements

Take our survey
https://aka.ms/cppcon

Our Sessions

Monday 25th

- **Implementing C++ Modules: Lessons Learned, Lessons Abandoned** – Cameron DaCamara & Gabriel Dos Reis

Tuesday 26th

- **Documentation in The Era of Concepts and Ranges** – Sy Brand & Christopher Di Bella (Google)
- **Static Analysis and Program Safety in C++: Making it Real** – Sunny Chatterjee
- **In-memory and Persistent Representations of C++** – Gabriel Dos Reis (online 27th)
- **Extending and Simplifying C++: Thoughts on pattern Matching using \`is\` and \`as** – Herb Sutter

Wednesday 27th

- **What's New in Visual Studio: 64-bit IDE, C++20, WSL 2, and more** – Sy Brand & Marian Luparu

Thursday 28th

- **C++20's <chrono> Calendars and Time Zones in MSVC** – Miya Natsuhara
- **An Editor Can Do That? Debugging Assembly Language and GPU Kernels in Visual Studio Code** – Julia Reid
- **Why does std::format do that?** – Charlie Barto
- **Finding bugs using path-sensitive static analysis** – Gabor Horvath (online 29th)