Design Rationale for the `<chrono>` Library

Howard Hinnant
Ripple

Meeting C++ 2019
Structure of `<chrono>`

- Durations:
  - These six durations represent the convenient high-level access.
  - lower-level access is available to clients for creating any duration unit they need.
  - Durations are the heart of the `<chrono>` library.

- Introduced in C++11

- Hours
- Minutes
- Seconds
- Milliseconds
- microseconds
- Nanoseconds
Structure of `<chrono>`

- Introduced in C++11
Structure of `<chrono>`

- Time points:
- Introduced in C++11
Structure of `<chrono>`

- **Clocks:**
  - Introduced in C++11
Evolution of `<chrono>`

Calendrical types:

- duration
- time_point
- clocks
- calendar

- Coming in C++20
Evolution of `<chrono>`

Time zone management:

- Coming in C++20
Evolution of `<chrono>`

And more clocks:

- Coming in C++20
C++20 provides a *complete* time handling library.
chrono in C++20

- **Everything** talked about today, whether it is old types from C++11 (e.g. durations and time_points) or new types in C++20, has a streaming operator in C++20:

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

- C++20 `<chrono>` becomes *much* easier to work with because you can easily print values out, even without knowing their type.

  ```cpp
  auto t0 = steady_clock::now();
  ...
  auto t1 = steady_clock::now();
  cout << "That took " << t1-t0 << '\n'; // That took 657ns
  ```
duration

template<class Rep, class Period = ratio<1>>
class duration;

- duration represents a duration of time, and can come in any unit.
- durations are represented by an arithmetic type, or a class type emulating an arithmetic type.
  - int, long, double, safe<int>, etc.
- duration::period is a compile-time fraction representing the time in seconds between each integral value stored in the duration.
- <chrono> defines several convenience type aliases for common units.
duration

template<class Rep, class Period = ratio<1>>
    class duration;

    • `<chrono>` defines several convenience type aliases for common units.

       nanoseconds       days
       microseconds     weeks
       milliseconds     months
       seconds          years
       minutes
       hours

New in C++20
duration

template<class Rep, class Period = ratio<1>>
class duration;

- Clients can define any custom unit they want.

using dsec = duration<double>;
using frame_rate = duration<int, ratio<1, 60>>;
using safe_ns = duration<safe_int<int64_t>, nano>();
duration

template<class Rep, class Period = ratio<1>>
class duration;

- Durations implicitly convert from coarse to fine:

```cpp
auto limit = 2h;
milliseconds x = limit; // 7'200'000ms
```
duration

template<class Rep, class Period = ratio<1>>
class duration;

- Durations have a named conversion from fine to coarse:

  auto limit = 2h;
milliseconds x = limit;  // 7'200'000ms
auto y = duration_cast<hours>(x);  // 2h
duration

template<class Rep, class Period = ratio<1>>
class duration;

• If the destination is floating-point-based, converts implicitly

    auto limit = 2h;
    milliseconds x = limit;  // 7'200'000ms
    auto y = duration_cast<hours>(x);  // 2h
    duration<double> z = x;  // 7'200.0s

• Implicit truncation error is a compile-time error.
• Round-off error is not a compile-time error.
time_point

template<class Clock, class Duration = typename Clock::duration>
class time_point;

• time_point represents a point in time.
• time_point is a wrapper around a duration.
  • Same value, same representation, just a different meaning.
• time_point offers only a subset of arithmetic algebra so as to catch logic errors at compile-time.
time_point

template<class Clock, class Duration = typename Clock::duration>
class time_point;

• time_point offers only a subset of arithmetic algebra so as to catch logic errors at compile-time.

auto tp1 = system_clock::now();  // tp1 is a time_point
auto tp2 = system_clock::now();  // tp2 is a time_point
auto diff = tp2 - tp1;           // diff is a duration
auto sum = tp2 + tp1;            // compile-time error
time_point

template<class Clock, class Duration = typename Clock::duration>
class time_point;

• time_point is templated on Clock to catch the error of mixing time_points from different clocks.

auto tp1 = system_clock::now();  // tp1 is a time_point
auto tp2 = steady_clock::now();  // tp2 is a time_point
auto diff = tp2 - tp1;           // compile-time error
What is the difference between a time point and a date?

- Example time points:
  - 2019-11-14 10:30:15
  - 2019-11-14 10:30:15.123
  - 2019-11-14 10:30:15.123456
  - 2019-11-14 10:30:15.123456789

Time points can have arbitrarily fine precision.
What is the difference between a time point and a date?

• Example time points:
  • 2019-11-14 10:30:15
  • 2019-11-14 10:30:15.123
  • 2019-11-14 10:30:15.123456
  • 2019-11-14 10:30:15.123456789
  • 2019-11-14 10:30
  • 2019-11-14 10

Time points can have arbitrarily coarse precision.
What is the difference between a time point and a date?

- Example time points:
  - 2019-11-14 10:30:15
  - 2019-11-14 10:30:15.123
  - 2019-11-14 10:30:15.123456
  - 2019-11-14 10:30:15.123456789
  - 2019-11-14 10:30
  - 2019-11-14 10
  - 2019-11-14

When the time point has a precision of a day, we call it a date.
What is the difference between a time point and a date?

- Example time points:
  - 2019-11-14 10:30:15
  - 2019-11-14 10:30:15.123
  - 2019-11-14 10:30:15.123456
  - 2019-11-14 10:30:15.123456789
  - 2019-11-14 10:30
  - 2019-11-14 10
  - 2019-11-14

Each precision has a type in the chrono system:

- `time_point<system_clock, seconds>`
- `time_point<system_clock, milliseconds>`
- `time_point<system_clock, microseconds>`
- `time_point<system_clock, nanoseconds>`
- `time_point<system_clock, minutes>`
- `time_point<system_clock, hours>`
- `time_point<system_clock, days>`
What is the difference between a time point and a date?

- Example time points:
  - 2019-11-14 10:30:15
  - 2019-11-14 10:30:15.123
  - 2019-11-14 10:30:15.123456
  - 2019-11-14 10:30:15.123456789
  - 2019-11-14 10:30
  - 2019-11-14 10
  - 2019-11-14

sys_time<Duration>
is a type alias for
time_point<system_clock, Duration>
What is the difference between a time point and a date?

- Example time points:
  - 2019-11-14 10:30:15
  - 2019-11-14 10:30:15.123
  - 2019-11-14 10:30:15.123456
  - 2019-11-14 10:30:15.123456789
  - 2019-11-14 10
  - 2019-11-14
  
  sys_time<Duration> is a type alias for
time_point<system_clock, Duration>

- Additional convenience type aliases:
  - sys_seconds
  - sys_time<milliseconds>
  - sys_time<microseconds>
  - sys_time<nanoseconds>
  - sys_time<minutes>
  - sys_time<hours>
  - sys_days
What is a calendar?

- A calendar is a collection of dates, where each date has a unique name.

Civil calendar

30.12.1969
31.12.1969
01.01.1970
02.01.1970
03.01.1970
What is a calendar?

- A calendar is a collection of dates, where each date has a unique name.

<table>
<thead>
<tr>
<th>Civil calendar</th>
<th>Julian calendar</th>
</tr>
</thead>
<tbody>
<tr>
<td>01.01.1970</td>
<td>19.12.1969</td>
</tr>
<tr>
<td>02.01.1970</td>
<td>20.12.1969</td>
</tr>
<tr>
<td>03.01.1970</td>
<td>21.12.1969</td>
</tr>
</tbody>
</table>

- Different calendars can refer to the same physical date, but have different names for that date.
What is a calendar?

• A calendar is a collection of dates, where each date has a unique name.

<table>
<thead>
<tr>
<th>Civil calendar</th>
<th>sys_days</th>
</tr>
</thead>
<tbody>
<tr>
<td>31.12.1969</td>
<td>-1</td>
</tr>
<tr>
<td>01.01.1970</td>
<td>0</td>
</tr>
<tr>
<td>02.01.1970</td>
<td>1</td>
</tr>
<tr>
<td>03.01.1970</td>
<td>2</td>
</tr>
</tbody>
</table>

• sys_days is a calendar too!
Calendar Interoperability

- sys_days is the canonical calendar in `<chrono>`. 
• sys_days is the canonical calendar in <chrono>.
• As long as each calendar can convert to and from sys_days, then each calendar can convert to any other calendar.
Calendar Interoperability

- Only these two calendars are in C++20 `<chrono>`.
- Clients can write their own calendars.
  - I've written several of them as proof of concept.
The civil calendar

class year_month_day;
data structure: \{year, month, day\}

- `year_month_day` implicitly converts to and from `sys_days`, with no loss of information (constexpr and noexcept).
- Constructible from a `year`, `month` and `day`.
- Has `year`, `month` and `day` getters.
- Equality and less-than comparable.
- Does `year` and `month`-oriented arithmetic.
- Does not do `day`-oriented arithmetic. `sys_days` does `day`-oriented arithmetic very efficiently.
The civil calendar

class year;

data structure:  {short}

• year represents the "name" of a year in the civil calendar. It does not represent a number of years (a duration).
  • One can subtract two year instances and get a year's duration type.
• year explicitly converts to and from int.
• Equality and less-than comparable.
• Does year-oriented arithmetic.
• Has user-defined literal y, e.g. 2019y.
The civil calendar

class month;

data structure: \{unsigned char\}

- month represents a month of a year. It does not represent a number of months (a duration).
  - One can subtract two month instances and get a months duration type.
- month explicitly converts to and from unsigned.
- Equality and less-than comparable.
- Does month-oriented arithmetic (modulo 12).
- Has inline constexpr constants, e.g. January, February, March, ...
The civil calendar

class day;

data structure: {unsigned char}

- day represents a day of a month. It does not represent a number of days (a duration).
  - One can subtract two day instances and get a days duration type.

- day explicitly converts to and from unsigned.

- Equality and less-than comparable.

- Does day-oriented arithmetic.

- Has user-defined literal d, e.g. 14d.
The civil calendar

class year_month_day;

data structure: \{year, month, day\}

- Typically sizeof is 4 bytes.
class year_month_day;

data structure: \{year, month, day\}

- Typically sizeof is 4 bytes.
- Constructible with conventional syntax operators in 3 different orders:

```cpp
auto ymd = 2019y/November/14d;
auto ymd = 14d/November/2019y;
auto ymd = November/14d/2019y;
```
The civil calendar

class year_month_day;
data structure: \{year, month, day\}

- Typically sizeof is 4 bytes.
- Constructible with conventional syntax operators in 3 different orders:
- Only the first field must be typed, the trailing fields can be integral.

    auto ymd = 2019y/11/14;
    auto ymd = 14d/11/2019;
    auto ymd = November/14/2019;
The civil calendar

class year_month_day;
data structure: \{year, month, day\}

• Or, if you prefer:

    year_month_day ymd\{year\{2019\}, month\{11\}, day\{14\}\};
The civil calendar

class year_month_day;

data structure: \{year, month, day\}

• Construction was designed to be type-safe and readable, but not overly verbose.

• Try to eliminate errors such as
  • \texttt{year\_month\_day\{10, 11, 12\}}.
The civil calendar

class year_month_day;
data structure: {year, month, day}

• Invalid dates are allowed, but are easily detectable.

    auto ymd = November/31/2019;
    assert(ymd.ok() == false);

• Rationale: Invalid dates are not necessarily errors (examples to follow later). And if they are errors, you get to decide if they are fatal, exceptional, or handled with an error code.
The civil calendar

class year_month_day_last;

data structure: \{year, month\}

- Represents the last day of the \{year, month\} pair.
- Constructible from a year and month.
- Implicitly convertible to sys_days (it's a partial calendar).
- Has year and month and day getters.
- Equality and less-than comparable.
- Does year and month-oriented arithmetic.
The civil calendar

class year_month_day_last;
data structure: {year, month}

• Constructible with conventional syntax operators by replacing the day-specifier with last.

    auto ymd = last/November/2019;

• Implicitly convertible to year_month_day.

    year_month_day ymd = November/last/2019;
The civil calendar

More about year and month arithmetic

- Consider:

  auto ymd = 31d/October/2019;
  ymd += months{1};

- ymd has the value 2019y/November/31d
The civil calendar

More about year and month arithmetic

• Consider:

```cpp
auto ymd = 31d/October/2019;
ymd += months{1};
```

• ymd has the value 2019y/November/31d

• To snap to the end of the month:

```cpp
if (!ymd.ok())
    ymd = ymd.year()/ymd.month()/last;
```
The civil calendar

More about year and month arithmetic

• Consider:

```cpp
auto ymd = 31d/October/2019;
ymd += months{1};
```

• `ymd` has the value 2019y/November/31d

• To snap to the end of the month:

```cpp
if (!ymd.ok())
    ymd = ymd.year() / ymd.month() / last;
```

• To overflow into the next month:

```cpp
if (!ymd.ok())
    ymd = sys_days{ymd};
```
The civil calendar

More about year and month arithmetic

• To snap to the end of the month:
  
  ```java
  if (!ymd.ok())
    ymd = ymd.year()/ymd.month()/last;
  ```

• To overflow into the next month:
  
  ```java
  if (!ymd.ok())
    ymd = sys_days{ymd};
  ```

• In either case, the invalid date 2019-11-31 is not a fatal nor exceptional error. It is just an intermediate result.
  
  • You get to decide how to handle it.
The civil calendar

class year_month_weekday;
data structure: \{year, month, weekday_indexed\}

    auto date = Thursday[2]/November/2019;

- Represents dates of the form the 2nd Thursday of November 2019.
- Constructible with conventional syntax
  - Anywhere one can put a day-specifier, one can use a weekday_indexed instead.
- `year_month_weekday` implicitly converts to and from `sys_days`, with no loss of information (constexpr and noexcept).
- This is a second complete civil calendar!
The civil calendar

class year_month_weekday;

data structure: {year, month, weekday_indexed}

    auto date = Thursday[2]/November/2019;

• Has year, month, weekday, and index getters.
• Equality comparable (not less-than).
• Does year and month-oriented arithmetic.
• Will explicitly convert to and from year_month_day by bouncing off of sys_days (just like a user-written calendar).
The civil calendar

class weekday;
data structure: \{unsigned char\}

- \texttt{weekday \textit{explicitly}} converts to and from \texttt{unsigned}.
  - Constructor accepts both C's \texttt{tm} encoding and ISO encoding.
- Explicitly constructible \textit{from} \texttt{sys\_days} (a partial calendar).
- Equality comparable (not less-than).
- Does day-oriented arithmetic (modulo 7).
  - Implies there is no officially supported "first day of the week."
- Has \texttt{inline constexpr} constants, e.g. Monday, Tuesday, Wednesday, ...
The civil calendar

class weekday_indexed;

data structure: \{weekday, integral index\}  // allowed to be 1 byte

• Represents the concept: \(n^{th}\) weekday of an unspecified month.

• `weekday_indexed` constructs from a weekday and an unsigned.

• Constructible with conventional syntax:

  ```cpp
  auto wdi = Thursday[2];
  ```
The civil calendar

class weekday_last;
data structure: {weekday}

- Represents the concept: last weekday of an unspecified month.
- `weekday_last` explicitly constructs from a weekday.
- Constructible with conventional syntax:

  ```
  auto wdi = Thursday[last];
  ```
The civil calendar

More about year and month arithmetic

- Consider:
  
  ```cpp
  auto date = Friday[5]/November/2019;
date += years{1};
  ```

- `date` has the value `Friday[5]/November/2020`. But `November/2020` only has 4 Fridays.
The civil calendar

More about year and month arithmetic

- Consider:
  ```
  auto date = Friday[5]/November/2019;
  date += years{1};
  ```

- `date` has the value `Friday[5]/November/2020`. But `November/2020` only has 4 Fridays.

- To snap to the end of the month (4th Friday of `November/2020`):
  ```
  if (!date.ok())
      date = sys_days{date.year()/date.month()/date.weekday()[last]};
  ```
The civil calendar

More about year and month arithmetic

• Consider:
  ```
  auto date = Friday[5]/November/2019;
  date += years{1};
  ```

• date has the value Friday[5]/November/2020. But November/2020 only has 4 Fridays.

• To snap to the end of the month (4th Friday of November/2020):
  ```
  if (!date.ok())
    date = sys_days{date.year()/date.month()/date.weekday()[last]};
  ```

• To overflow into the next month (1st Friday of December/2020):
  ```
  if (!date.ok())
    date = sys_days{date};
  ```
Time Zones

- `system_clock` (and `sys_time<Duration>`) are Unix Time.
- Unix Time measures time since (and prior) 1970-01-01 00:00:00 UTC excluding leap seconds.
- Yes, C++20 can handle leap seconds but `sys_time` ignores them (we'll get there ...).
Time Zones

• system_clock (and sys_time<Duration>) are Unix Time.
• Unix Time measures time since (and prior) 1970-01-01 00:00:00 UTC excluding leap seconds.
• Yes, C++20 can handle leap seconds but sys_time ignores them (we'll get there ...).
• C++20 adds a time_zone class which is used to transform sys_time<Duration> into "local time".
• C only has the concept of UTC and "local time". C++20 adds to these two concepts the ability to compute with any time zone in the IANA time zone database.
• This means time zone names are portable.
Time Zones

• Examples:

The current UTC time:

```cpp
auto tp = system_clock::now();
2019-11-14 10:13:40.785346
```
Time Zones

• Examples:

The current UTC time:

```cpp
auto tp = system_clock::now();
2019-11-14 10:13:40.785346
```

The current local time:

```cpp
zoned_time tp{current_zone(), system_clock::now()};
2019-11-14 11:13:40.785346 CET
```
Time Zones

• Examples:

The current UTC time:

    auto tp = system_clock::now();
    2019-11-14 10:13:40.785346

The current local time:

    zoned_time tp{current_zone(), system_clock::now()};
    2019-11-14 11:13:40.785346 CET

The current time in Berlin:

    zoned_time tp{"Europe/Berlin", system_clock::now()};
    2019-11-14 11:13:40.785346 CET
Time Zones

template<class Duration, class TimeZonePtr = const time_zone*> class zoned_time;

data structure: {TimeZonePtr, sys_time<Duration>}

- zoned_time is a convenience wrapper of a pointer to a time zone, and a sys_time time_point.
- One can think of it as a triple of {time_zone*, local_time<Duration>, sys_time<Duration}!, but the local time is computed upon demand.
- One can create custom time zones to handle things outside the IANA time zone database (e.g. POSIX time zone strings).

Time Zones

template<class Duration, class TimeZonePtr = const time_zone*> class zoned_time;

data structure: {TimeZonePtr, sys_time<Duration>}

• zoned_time is typically constructed with two arguments.
  • The first argument represents a time_zone.
    • Can be either a time_zone const*, or a string_view.
  • The second argument represents a time_point.
    • Can be a sys_time, local_time, or another zoned_time.

    zoned_time zt{A time zone, A time point};
Time Zones

template<class Duration, class TimeZonePtr = const time_zone*>
class zoned_time;

data structure: {TimeZonePtr, sys_time<Duration>}

zoned_time zt{A time zone, A time point};

• Examples:

    time_zone const*  sys_time

The current local time:

    zoned_time tp{current_zone(), system_clock::now()};
    2019-11-14 11:13:40.785346 CET
Time Zones

template<class Duration, class TimeZonePtr = const time_zone*> 
class zoned_time;

data structure: {TimeZonePtr, sys_time<Duration>}

zoned_time zt{A time zone, A time point};

• Examples:

string_view string_view sys_time

The current Berlin time:

zoned_time tp{"Europe/Berlin", system_clock::now()};

2019-11-14 11:13:40.785346 CET
Time Zones

template<class Duration, class TimeZonePtr = const time_zone*> class zoned_time;

data structure: {TimeZonePtr, sys_time<Duration>}

zoned_time zt{A time zone, A time point};

• Examples:

Midnight Berlin time:

zoned_time tp{"Europe/Berlin", local_days{2019y/11/14}};
2019-11-14 00:00:00 CET
Time Zones

template<class Duration, class TimeZonePtr = const time_zone*>
class zoned_time;

data structure: {TimeZonePtr, sys_time<Duration>}

    zoned_time zt{A time zone, A time point};

• Examples:

    string_view sys_time

1:00 Berlin time:

    zoned_time tp{"Europe/Berlin", sys_days{2019y/11/14}};
    2019-11-14 01:00:00 CET
Time Zones

template<class Duration, class TimeZonePtr = const time_zone*> class zoned_time;

data structure: {TimeZonePtr, sys_time<Duration>}

    zoned_time zt{A time zone, A time point};

• Examples:

1:00 Berlin time:

    zoned_time tp{"Europe/Berlin", local_days{2019y/11/14} + 1h};

    2019-11-14 01:00:00 CET
Time Zones

template<class Duration, class TimeZonePtr = const time_zone*> class zoned_time;

data structure: {TimeZonePtr, sys_time<Duration>}

zoned_time zt{A time zone, A time point};

• tp and tp2 represent the same UTC instant, but in different time zones

1:00 Berlin time:

zoned_time tp{"Europe/Berlin", local_days{2019y/11/14} + 1h};
    2019-11-14 01:00:00 CET

zoned_time tp2{"America/New_York", tp};
    2019-11-13 19:00:00 EST
Time Zones

local_time<Duration>

is a type alias for
time_point<local_t, Duration>

• local_t is "not really a clock."
  • It has no now() function.
• local_time is a time_point with respect to a not-yet-specified time_zone.
  • It can be paired with a time_zone and only then will it refer to an instant in time (e.g. in a zoned_time constructor).
• local_days is just a type alias for local_time<days>.
• Calendars convert back and forth to local_days with the exact same formulas that they use for sys_days.
local_time<Duration>
is a type alias for
time_point<local_t, Duration>

• Calendars convert back and forth to local_days with the exact same formulas that they use for sys_days.

  sys_days{2019y/11/14}  A UTC time_point

  local_days{2019y/11/14}  A somewhat nebulous time_point, until you pair it with a time_zone.

But both contain the value 18214 days.
Time Zones

template<class Duration, class TimeZonePtr = const time_zone*> 
class zoned_time;

data structure: {TimeZonePtr, sys_time<Duration>}

zoned_time tp{"Europe/Berlin", local_days{2019y/11/14} + 1h};

    tp.get_sys_time();  2019-11-14 00:00:00
    tp.get_local_time(); 2019-11-14 01:00:00

• sys_time and local_time are distinct families of time_points
  so that the compiler will catch accidentally mixing them.
• They both have distinct semantics.
• They are both useful.
• They are both available.
for (auto d = January/9/2019; d.year() < 2020y; 
    d = sys_days{d} + weeks{2})
{
    zoned_time london{"Europe/London", local_days{d} + 18h};
    cout << london << '\n';
    cout << zoned_time{"America/New_York", london} << "\n\n";
}
Time Zones

Example: Directions Group meeting times

```cpp
for (auto d = January/9/2019; d.year() < 2020y;
     d = sys_days{d} + weeks{2})
{
    zoned_time london{"Europe/London", local_days{d} + 18h};
    cout << london << 'n';
    cout << zoned_time{"America/New_York", london} << "\n\n";
}
```

2019-01-09 18:00:00 GMT
2019-01-09 13:00:00 EST

2019-01-23 18:00:00 GMT
2019-01-23 13:00:00 EST

...
for (auto d = January/9/2019; d.year() < 2020y; 
   d = sys_days{d} + weeks{2})
{
    zoned_time london{"Europe/London", local_days{d} + 18h};
    cout << london << '
';
    cout << zoned_time{"America/New_York", london} << '

';
}
Time Zones

Example: Directions Group meeting times

```cpp
for (auto d = January/9/2019; d.year() < 2020y;
    d = sys_days{d} + weeks{2})
{
    zoned_time london{"Europe/London", local_days{d} + 18h};
    cout << london << 'n';
    cout << zoned_time{"America/New_York", london} << "n"
}

2019-10-30 18:00:00 GMT
2019-10-30 14:00:00 EDT

...

2019-12-25 18:00:00 GMT
2019-12-25 13:00:00 EST
```
Even though everything has a streaming operator, it may not stream with the format you desire.

C++20 <chrono> fully integrates into C++20 std::format.
  - With all of the flag functionality of std::strftime/std::put_time.
  - And a little more.
Formatting

• Given:

```cpp
auto tp = system_clock::now();
auto tz = locale_zone("Europe/Berlin");
```

• Examples:

```cpp
cout << zoned_time{tz, tp} << '\n';
```

```
2019-11-14 11:13:40.785346 CET
```

The default streaming format
Formatting

• Given:

```cpp
auto tp = system_clock::now();
auto tz = locale_zone("Europe/Berlin");
```

• Examples:

```cpp
cout << format("{:%F %T %Z}\n", zoned_time{tz, tp});
```

2019-11-14 11:13:40.785346 CET

No change.
The default explicitly specified.
Formatting

- Given:
  ```cpp
  auto tp = system_clock::now();
  auto tz = locale_zone("Europe/Berlin");
  ```
- Examples:
  ```cpp
  cout << format("{%d.%m.%Y %T%z}\n", 
                  zoned_time{tz, tp});
  ```

  
  ![14.11.2019 11:13:40.785346+0100](image)

  d.m.y ordering.
  UTC offset instead of time zone abbreviation.
Formatting

• Given:
  
  auto tp = system_clock::now();
  auto tz = locale_zone("Europe/Berlin");

• Examples:
  
  cout << format(locale{"de_DE"}, 
                  "{:d.%m.%Y %T%z}\n",
                  zoned_time{tz, tp});

  14.11.2019 11:13:40,785346+0100

  Decimal point specified by explicit locale. Your OS may not support this locale.
Formatting

- Given:
  ```cpp
cpp
auto tp = system_clock::now();
auto tz = locale_zone("Europe/Berlin");
```
- Examples:
  ```cpp
cpp
cout << format("{:d.%m.%Y %T}\n",
    zoned_time{tz, floor<milliseconds>(tp)});
```
  ```cpp
cpp
14.11.2019 11:13:40.785
```

Precision governed by input time point precision. Dropped UTC offset.
Formatting

- Given:
  
  ```cpp
  auto tp = system_clock::now();
  auto tz = locale_zone("Europe/Berlin");
  ```

- Examples:

  ```cpp
  cout << format("{%d.%m.%Y %T}\n",
                 zoned_time{tz, floor<seconds>(tp)});
  ```

  14.11.2019 11:13:40

  Seconds-precision eliminates decimal point.
Formatting

• All of these types can be formatted:

  zoned_time  local_time  sys_time
  duration    year_month_day  year_month_day_last
  month_day   year        month      day       weekday
  file_time   hh_mm_ss    weekday_indexed
  utc_time    year_month  weekday_last
  gps_time    year_month_weekday
  tai_time    year_month_weekday_last
  month_day_last  month_weekday_last  month_weekday
 Parsing

• In general, if you can `std::format` it, you can `std::chrono::parse` it back in, usually with the same formatting string.

```cpp
system_clock::time_point tp;
cin >> parse("%d.%m.%Y %T%z", tp);
cout << tp << '\n';
```

Clocks

- C++11 introduced system_clock, steady_clock and high_resolution_clock.
- Each clock has its own family of time_points.
- A family of time_points allows different precisions, but not different clocks.
- Arithmetic within a family of time_points results in a time_point or duration with a precision computed by the common_type of the precision of the arguments.
- Arithmetic among different families of time_points is a compile-time error.
Clocks

- `system_clock` measures the time of day and the date.
- `steady_clock` is a stop watch no relationship to a calendar.
- `high_resolution_clock` is typically a type alias of `steady_clock` or `system_clock`. 
Clocks

• C++20 adds:
  • file_clock
  • utc_clock
  • gps_clock
  • tai_clock
Clocks

file_clock
template<class Duration>
  using file_time = time_point<file_clock, Duration>;

  • file_clock is the same type as
    std::file_system::file_time_type::clock.
  • file_clock's epoch is unspecified.
  • file_time_type is returned from functions such as
    file_system::last_write_time(const path& p).
  • file_time can be cast to sys_time (and vice-versa)
    via clock_cast:

    auto tp = clock_cast<system_clock>(last_write_time("/path"));
    last_write_time("/path", clock_cast<file_clock>(tp));
Clocks

\texttt{utc\_clock}
\begin{verbatim}
template<class Duration>
    using utc\_time = time\_point<utc\_clock, Duration>;
\end{verbatim}

- \texttt{utc\_time} is just like \texttt{sys\_time} except that it counts leap seconds.
- Useful when subtracting \texttt{time\_points} across a leap second insertion point and 1s accuracy is required.
- \texttt{clock\_cast} can be used to convert among \texttt{utc\_time}, \texttt{file\_time} and \texttt{sys\_time}.
- \texttt{utc\_clock::now()} is allowed but not required to be accurate during a leap second insertion.
- Formatting and parsing \texttt{utc\_time} allows for 61s in a minute, but only for a \texttt{utc\_time} that is actually referencing a leap second insertion.
Clocks

gps_clock
template<class Duration>
  using gps_time = time_point<gps_clock, Duration>;

• gps_time measures time since Sunday[1]/January/1980 00:00:00 UTC.
• Useful for dealing with time points in the "GPS-shifted" civil calendar.
• clock_cast can be used to convert among gps_time, utc_time, file_time and sys_time.
• gps_clock::now() is allowed but not required to be fed from a GPS receiver.
• formatting and parsing gps_time maps to a civil time that is currently 18s ahead of sys_time and utc_time, and gets another second ahead with each added leap second.
Clocks

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

- tai_time measures time since 1958y/1/1 00:00:00 and is offset 10s ahead of UTC at this date.
- Useful for dealing with time points in the "TAI-shifted" civil calendar.
- clock_cast can be used to convert among tai_time, gps_time, utc_time, file_time and sys_time.
- tai_clock::now() is allowed but not required to be accurate during a leap second insertion.
- formatting and parsing tai_time maps to a civil time that is always 19s ahead of gps_time.
Clocks

time_point\langle A\_clock, \text{Duration}\rangle
clock\_cast\langle A\_clock\rangle\langle \text{time\_point}\langle B\_clock, \text{Duration}\rangle \text{ tp}\rangle;

- User-written clocks can add support to participate in the clock\_cast system with $O(1)$ amount of code (independent of the number of clocks supporting clock\_cast).
- Once clock\_cast is supported by a user-written clock, that clock can clock\_cast bidirectionally to every clock that supports clock\_cast.
Library Design

• Library Design is an engineering process.
  • Both an art and a science.
• There are always tradeoffs to be made among conflicting goals.
• It is an iterative process, as is all engineering.
Library Design

• It is an iterative process, as is all engineering.
  • The first car wasn't Ferrari Enzo.
  • It was a tricycle with a motor attached.
  • It took many years and iterations for engineering technology to evolve from one to another.
• So it goes with software.
Library Design

- And we're still early in the maturing of this industry.
- Study other's code.
- Learn from past successes.
- Learn even more from failures.
Library Design

• Detect as many errors as you can at compile-time.
• Make client code as readable as possible.
• Eliminate ambiguities in client code.
• Encourage your client to write efficient code.
• Offer both low-level and high-level access.
  • Low-level access emphasizes uncompromising performance and flexibility.
  • High-level access emphasizes convenience for common cases.
Library Design

• If you only take away one thing from this talk...

• The readability of the code your clients write is far more important than the readability of your library's synopsis or header.
Q & A

Thank you for your time.