A Crash Course in Calendars, Dates, Time, and Time Zones

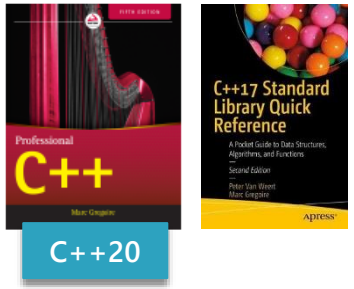
MARC GREGOIRE

Marc Grégoire

- Belgium
- Software architect for Nikon Metrology

- Microsoft VC++ MVP Since 2007

- Author of Professional C++, 2nd, 3rd, 4th, and 5th Edition
- Co-author of C++ Standard Library Quick Reference & C++17 Standard Library Quick Reference



- Founder of the Belgian C++ Users Group (BeCPP)

Agenda

- Compile-Time Rational Numbers } `<ratio>`
- Durations
- Clocks
- Time Points
- Dates
- Time Zones

} `<chrono>`

Agenda

- Compile-Time Rational Numbers
- Durations
- Clocks
- Time Points
- Dates
- Time Zones

Compile-Time Rational Numbers

- Defined in `<ratio>`
- Work with rational numbers at compile time
- Always normalized representation
- Needed for durations in the `<chrono>` library

Compile-Time Rational Numbers

- Define a rational number:

```
using r1 = ratio<1, 60>;  // Represents 1/60
```

- Retrieve numerator and denominator

```
intmax_t num { r1::num };
intmax_t den { r1::den };
cout << format("r1 = {}/{}", num, den); // r1 = 1/60
```

- It's all compile-time constants:

```
intmax_t n { 1 };
intmax_t d { 60 };
using r1 = ratio<n, d>;    // Error
```

Compile-Time Rational Numbers

- Arithmetic with **`ratio_add`**, **`ratio_subtract`**, **`ratio_multiply`**, and **`ratio_divide`**:

```
using r1 = ratio<1, 60>;    // 1/60
using r2 = ratio<1, 30>;    // 1/30
using result = ratio_add<r1, r2>::type;
cout << format("sum = {}/{}", result::num, result::den);// 1/20
```

- Comparisons with **`ratio_equal`**, **`ratio_not_equal`**, **`ratio_less`**, **`ratio_less_equal`**, **`ratio_greater`**, and **`ratio_greater_equal`**:

```
using res = ratio_less<r2, r1>;
cout << format("r2 < r1: {}", res::value); // false
```

Compile-Time Rational Numbers

- Predefined SI type aliases:

```
using atto  = ratio<1, 1'000'000'000'000'000'000>;
using femto = ratio<1, 1'000'000'000'000'000>;
using pico  = rat1, 1'000'000'000'000>;
using nano  = ratioio<<1, 1'000'000'000>;
using micro = ratio<1, 1'000'000>;
using milli = ratio<1, 1'000>;
using centi = ratio<1, 100>;
using deci  = ratio<1, 10>;
using deca  = ratio<10, 1>;
using hecto = ratio<100, 1>;
using kilo  = ratio<1'000, 1>;
using mega  = ratio<1'000'000, 1>;
using giga  = ratio<1'000'000'000, 1>;
using tera  = ratio<1'000'000'000'000, 1>;
using peta  = ratio<1'000'000'000'000'000, 1>;
using exa   = ratio<1'000'000'000'000'000'000, 1>;
```

Agenda

- Compile-Time Rational Numbers
- Durations
- Clocks
- Time Points
- Dates
- Time Zones

Durations

- Interval between two points in time
- Represented by **`std::duration`** from **`<chrono>`**, contains:
  - A tick
  - A tick period = seconds between 2 ticks = rational constant
  - ```
    template <class Rep, class Period = ratio<1>>
    class duration {...}
    ```
    - **`Rep`** = type to represent number of ticks
    ```

Durations – Examples

- Duration with ticks of 1 second:

```
duration<long, ratio<1>> d1;
duration<long> d1;
```

- Duration with ticks of 1 minute:

```
duration<long, ratio<60>> d2;
```

- Duration with ticks of a sixtieth of a second:

```
duration<double, ratio<1, 60>> d3;
```

- Use of predefined SI rational constants:

```
duration<long long, milli> d4;
```

Durations – Operations

- Working with durations

```
// Define 2 durations:
// one expressed as minutes, the other as seconds.
duration<long, ratio<60>> d3 { 10 };  // = 10 minutes
duration<long, ratio<1>> d4 { 14 };   // = 14 seconds

// Compare durations.
if (d3 > d4) { cout << "d3 > d4"; }
else         { cout << "d3 <= d4"; }
```

Durations – Operations

- Working with durations

```
// Increment d4 (= 14sec) with 1.
++d4;     // 15sec
// Multiply d4 by 2.
d4 *= 2;  // 30sec
// Add both durations and store as minutes.
duration<double, ratio<60>> d5 { d3 + d4 };
// Add both durations and store as seconds.
duration<long, ratio<1>> d6 { d3 + d4 };
cout << format("{}min + {}sec = {}min or {}sec",
                d3.count(), d4.count(), d5.count(), d6.count());
    // 10min + 30sec = 10.5min or 630sec
```

Durations – Operations

- Converting durations

```
// Create a duration of 30 seconds.
duration<long> d7 { 30 };

// Convert the seconds of d7 to minutes.
duration<double, ratio<60>> d8 { d7 };
cout << format("{}sec = {}min", d7.count(), d8.count());
                // 30sec = 0.5min

duration<long, ratio<60>> d8 { d7 }; // minutes   // Error!
// Force conversion (0 instead of 0.5)
auto d8 { duration_cast<duration<long, ratio<60>>>(d7) }; // = 0
```

Durations – Predefined & Literals

- Predefined durations in `std::chrono`:

```
using nanoseconds  = duration<X 64 bits, nano>;
using microseconds = duration<X 55 bits, micro>;
using milliseconds = duration<X 45 bits, milli>;
using seconds      = duration<X 35 bits>;
using minutes      = duration<X 29 bits, ratio<60>>;
using hours        = duration<X 23 bits, ratio<3'600>>;
using days   = duration<X 25 bits, ratio_multiply<ratio<24>,
                 hours::period>>;
using weeks  = duration<X 22 bits, ratio_multiply<ratio<7>,
                 days::period>>;
using years  = duration<X 17 bits,
                 ratio_multiply<ratio<146'097, 400>, days::period>>;
using months = duration<X 20 bits, ratio_divide<years::period,
                 ratio<12>>>;
```

Durations – Predefined & Literals

- Standard user-defined duration literals:
  - `h`, `min`, `s`, `ms`, `us`, and `ns`
- Require any of the following using directives:

```
using namespace std;
using namespace std::literals;
using namespace std::chrono_literals;
using namespace std::literals::chrono_literals;
```

Durations – Predefined & Literals

- Predefined durations:

```
duration<long, ratio<60>> d9 { 10 };     // minutes
```

  - Equivalent to:

```
minutes d9 { 10 };
```

  - Or:

```
auto d9 { 10min };
```

- Example:

```
auto t { hours { 1 } + minutes { 23 } + seconds { 45 } };
```

Durations – Predefined & Literals

- Warning: predefined durations use integral types, so:

```
seconds s { 90 };
minutes m { s }; // Error
duration<double, ratio<60>> m { s }; // 1.5
```

Agenda

- Compile-Time Rational Numbers
- Durations
- Clocks
- Time Points
- Dates
- Time Zones

Clocks

- Several clocks available in **<chrono>**:
  - **std::system_clock**: wall clock time from system-wide real-time clock
  - **std::steady_clock**: guarantees it never goes backwards
  - **std::high_resolution_clock**: has shortest possible tick period
  - C++20 adds:
    - **utc_clock**, **tai_clock**, **gps_clock**, and **file_clock**
- Every clock has a **now()** method

Clocks – Examples

- Example:

```
// Get current time as a time_point.
system_clock::time_point tpoint { system_clock::now() };
// Or:
auto tpoint { system_clock::now() };
// Convert to a time_t.
time_t tt { system_clock::to_time_t(tpoint) };
// Convert to local time.
tm* t { localtime(&tt) };
// Write the time to the console.
cout << put_time(t, "%H:%M:%S");
```

Clocks

- Time execution time:

```
// Get start time.
auto start { high_resolution_clock::now() };
// Execute code to benchmark.
double d { 0 };
for (int i { 0 }; i < 1'000'000; ++i) {
    d += sqrt(sin(i) * cos(i));
}
// Get end time and calculate the difference.
auto end { high_resolution_clock::now() };
auto diff { end - start };
// Convert difference into milliseconds.
cout << duration<double, milli> { diff }.count() << "ms";
```

Agenda

- Compile-Time Rational Numbers
- Durations
- Clocks
- Time Points
- Dates
- Time Zones

Time Points

- `std::time_point` in `<chrono>`
- Associated with a **`clock`**
- Represents point in time as **`duration`** relative to epoch (= origin of **`clock`**)
- Support arithmetic (tp = **`time_point`**, d = **`duration`**)

`tp + d = tp`	`tp - d = tp`
`d + tp = tp`	`tp - tp = d`
`tp += d`	`tp -= d`

Time Points

- Example:

```
// Create a time_point representing the epoch of the
// associated steady clock.
time_point<steady_clock> tp1;
// Add 10 minutes to the time_point.
tp1 += minutes { 10 };
// Store the duration between epoch and time_point.
auto d1 { tp1.time_since_epoch() };
// Convert the duration to a duration in seconds.
duration<double> d2 { d1 };
cout << d2.count() << " seconds"; // 600 seconds
```

Time Points

- **Implicit** conversions
- Example: *seconds -> milliseconds*

```
time_point<steady_clock, seconds> tpSeconds { 42s };
// Convert seconds to milliseconds implicitly.
time_point<steady_clock, milliseconds> tpMilliseconds {
    tpSeconds };
cout << tpMilliseconds.time_since_epoch().count() << "ms";
    // 42000ms
```

Time Points

- **Explicit** conversions
- Example: *milliseconds -> seconds*

```
time_point<steady_clock, milliseconds> tpMilliseconds { 42’016ms };
// Convert milliseconds to seconds explicitly.
time_point<steady_clock, seconds> tpSeconds {
    time_point_cast<seconds>(tpMilliseconds) };
// Or:
auto tpSeconds { time_point_cast<seconds>(tpMilliseconds) };

// Convert seconds back to milliseconds.
milliseconds ms { tpSeconds.time_since_epoch() };
cout << ms.count() << "ms"; // 42000ms
```

Agenda

- Compile-Time Rational Numbers
- Durations
- Clocks
- Time Points
- Dates
- Time Zones

Dates

- Since C++20
- Gregorian calendar support
- Several classes to represent dates or part of dates:
  - `year`, `month`, `day`
  - `weekday`, `weekday_indexed`, `weekday_last`
  - `month_day`
  - `year_month`
  - `year_month_day`
  - …

Dates

- Define an `std::year`:

```
year y1 { 2021 };
auto y2 { 2021y };
```

- Define an `std::month`:

```
month m1 { 10 };
auto m2 { October };
```

- Define an `std::day`:

```
day d1 { 27 };
auto d2 { 27d };
```

Dates

- Create a date 2021-10-27:

```
year_month_day fulldate1 { 2021y, October, 27d };
auto fulldate2 { 2021y / October / 27d };
auto fulldate3 { 27d / October / 2021y };
```

- Create a date for the 4th Wednesday of October 2021:

```
year_month_day fulldate4 { Wednesday[4] / October / 2021 };
```

- Create a **`month_day`** for October 27 of an unspecified year:

```
auto oct27 { October / 27d };
```

- Create a **`year_month_day`** for October 27, 2021:

```
auto oct27_2021 { 2021y / oct27 };
```

Dates

- Create a **`month_day_last`** for the last day of an October of an unspecified year:

```
auto lastDayOfAnOctober { October / last };
```

- Create a **`year_month_day_last`** for the last day of October for the year 2021:

```
auto lastDayOfOct2021 { 2021y / lastDayOfAnOctober };
```

- Create a **`year_month_weekday_last`** for the last Monday of October 2021.

```
auto lastMondayOfOct2021 { 2021y / October / Monday[last] };
```

Dates

- New type aliases:

```
template <typename Duration>
using sys_time =
    std::chrono::time_point<std::chrono::system_clock, Duration>;
// Representation of number of seconds since epoch.
using sys_seconds = sys_time<std::chrono::seconds>;
// Representation of number of days since epoch.
using sys_days = sys_time<std::chrono::days>;
```

- *Serial-based* representations (single number) versus *field-based* types like (`year_month_day`)

Dates

- Create a `sys_days` representing today:

```
auto today { floor<days>(system_clock::now()) };
```

- Convert `year_month_day` to `time_point`:

```
system_clock::time_point t1 { sys_days { 2020y / June / 22d } };
```

- Convert `time_point` to `year_month_day`:

```
year_month_day yearmonthday { floor<days>(t1) };
year_month_day today2 { floor<days>(system_clock::now()) };
```

Dates

- Full date with timestamp:

```
auto d2 { sys_days { 2020y / June / 22d } + 9h + 35min + 10s };
```

- Arithmetic

```
auto d3 { d2 + days { 5 } };  // Add 5 days to d2
```

- Streaming dates

```
cout << d2 << '\n' << d3;
```

- Output

```
2020-06-22 09:35:10
2020-06-27 09:35:10
```

Dates

- Careful:

```
auto d2 { sys_days { 2020y / June / 22d } + 9h + 35min + 10s };
auto d3 { d2 + years { 1 } }; // Add 1 year to d2
```

- Result

```
d2 = 2020-06-22 09:35:10
d3 = 2021-06-22 15:24:22
```

⚠ Adding 1 year does not add:
86,400 * 365 = 31,536,000 seconds
but
86,400 * ((365 * 400) + 97) / 400) = 31,556,952 seconds

Dates

- You can use field-based types instead

```
auto d2 { sys_days { 2020y / June / 22d } + 9h + 35min + 10s };

// Split d2 into days and remaining seconds.
sys_days d2_days{ time_point_cast<days>(d2) };
seconds d2_seconds{ d2 - d2_days };

// Convert the d2_days serial type to field-based type.
year_month_day ymd2{ d2_days };

// Add 1 year.
year_month_day ymd3{ ymd2 + years{ 1 } };
auto d3{ sys_days{ ymd3 } + d2_seconds };
```

- Result

```
t2 = 2020-06-22 09:35:10
t3 = 2021-06-22 09:35:10
```

Agenda

- Compile-Time Rational Numbers
- Durations
- Clocks
- Time Points
- Dates
- Time Zones

Time Zones

- Time zone database:
  - = list of time zones, including things like daylight saving time descriptions
- Access time zone database

```
std::chrono::get_tzdb()
```

- List all available time zones:

```
const auto& database { get_tzdb() };
for (const auto& timezone : database.zones) {
    cout << timezone.name() << endl;
}
```

Time Zones

- Get a specific time zone:

```
auto* brussels { locate_zone("Europe/Brussels") };
auto* gmt { locate_zone("GMT") };
auto* current { current_zone() };
```

- Conversion between time zones:

```
// Convert the current system time to GMT.
gmt->to_local(system_clock::now());
// Construct a UTC time. (2020-06-22 09:35:10 UTC)
auto t { sys_days { 2020y / June / 22d } + 9h + 35min + 10s };
// Convert UTC time to Brussels' local time.
brussels->to_local(t);
```

Time Zones

- Construct a specific time in a specific time zone:

```
// Construct a local time in the Brussels' time zone.
auto* brussels { locate_zone("Europe/Brussels") };
zoned_time<hours> brusselsTime {
    brussels, local_days { 2020y / June / 22d } + 9h };

// Convert to New York time.
zoned_time<hours> newYorkTime {
    "America/New_York", brusselsTime };
```

Agenda

- Compile-Time Rational Numbers
- Durations
- Clocks
- Time Points
- Dates
- Time Zones

Questions