Durations and Clocks

- C++11 Chrono Overview
- Durations and ...
- ... Time Points (Clocks)

C++11: Chrono

With C++11 a library component for managing date and time was introduced (beyond what was available for long because of C compatibility).

- As many similar libraries it makes a clear distinction between
  - durations and
  - time points.

The feature that makes this library shine is its flexibility with respect to the usual trade-off between resolution, range, and space requirements of the underlying type (to store a duration or time point).

The chapter on the C++11 *Chrono Library Part* from Nicolay Josuttis Books, referenced above, has also been made available online here: http://www.informit.com/articles/article.aspx?p=1881386&seqNum=2

`std::chrono` – Durations

Though the duration type is fully configurable through a template*, most programs will probably choose from one of the predefined types that satisfies their needs for resolution:

- `std::chrono::nanoseconds` at least 64 bit signed
- `std::chrono::microseconds` at least 55 bit signed
- `std::chrono::milliseconds` at least 45 bit signed
- `std::chrono::seconds` at least 35 bit signed
- `std::chrono::minutes` at least 29 bit signed
- `std::chrono::hours` at least 23 bit signed

> Duration type conversions to a *finer grained* type will always happen automatically, while conversions to *coarser grained* type require an `std::chrono::duration_cast`.

A 64-bit type the minimum requirement for nanosecond resolution – with the **minimum requirement** for the other types adapted accordingly – the minimum range of a duration covers ±500 years.

*: E.g. a duration type could well count in 5/17 microseconds if that matches the resolution of a hardware timer exactly and allows for precise calculations without any rounding errors or occasional jitter.

Duration Example (1)

For a basic use it needs only to be understood that automatic conversion happen as long as the target duration counts in finer grained units ...

```
#include <chrono>
…
std::chrono::minutes m{22};  // m.count() is 22
std::chrono::seconds s{17};  // s.count() is 17
s += m;                      // s.count() is 1337 (22*60 + 17)
s *= 100;                    // s.count() is 133700
```

... while assignments to coarser grained durations are an error ...

```
m = s;                       // does NOT compile
```

... unless an `std::chrono::duration_cast` is applied:*

```
m = std::chrono::duration_cast<std::chrono::minutes>(s);
                             // m.count() is 2228
```

Duration Example (2)

To avoid long namespace prefixes, namespace aliases are handy:*

```
namespace sc = std::chrono;   // abbreviating std::chrono:: to sc::
…                                    // continuing from previous page
sc::hours h = sc::duration_cast<sc::hours>(m); // m.count() is 2228
//   auto h = sc::duration_cast<sc::hours>(m);
```

Finally a useful helper to turn durations into something readable:

```
std::string to_string(sc::seconds sec) {
    const auto h = sc::duration_cast<sc::hours>(sec);
    const auto m = sc::duration_cast<sc::minutes>(sec-h);
    const auto s = sc::duration_cast<sc::seconds>(sec-h-m);
    return std::to_string(h.count()) + "h"
         + std::to_string(m.count()) + "m"
         + std::to_string(s.count()) + "s";
}
…             // continuing from above
… to_string(s) … // returns "37h8m20s"
```

*: These seems especially useful to abbreviate nested `std::` namespaces – like those from the `<chrono>` – while keeping a gentle reminder the identifier following is from the standard library.

`std::chrono` – Clocks

A duration (type) combined with an *epoch*[*] is a *clock* that represents a time point.

Which kind of clocks are supported is basically implementation defined with the following minimum requirements:

- `std::chrono::system_clock` – represents the usual "wall-clock" or "calendar date & time" of a computer system;
- `std::high_resolution_clock` – the clock with the best resolution available (but with a more or less frequent wrap-around);
- `std::chrono::steady_clock` – probably not tied to a specific calendar date and with the special guarantee that it will only advance.

Only the last allows to reliably determine a real time span as difference of two time points returned from its static member function `now()`.

[*]: Per definition the epoch of a clock is the time point represented by the duration zero. From its epoch a clock will reach into the past and into the future, usually symmetrically if an ordinary signed integral or floating point type is used.

Clock Examples (1)

Following are the attributes of `std::chrono::system_clock` ...*

```
resolution : 1/1000000
value range: -9223372036854775808 .. 9223372036854775807
since epoch: 45+ years
          or: 16581+ days
          or: 397964+ hours
          or: 23877893+ minutes
          or: 1432673591+ seconds
or in ticks: 1432673591377941
```

---

*: ... determined on the author's system with the helper function below:

```
template<typename Clock>
 void show_clock() {
        using period = typename Clock::period;
        using limits = std::numeric_limits<typename Clock::rep>;
        std::cout << "resolution : " << period::num << '/' << period::den << '\n';
        std::cout << "value range: " << limits::min() << " .. " << limits::max() << '\n';
        const auto tse = Clock::now().time_since_epoch();
        const auto sse = sc::duration_cast<sc::seconds>(tse).count();
        std::cout << "since epoch: " << sse /60/60/24/365 << "+ years\n";
        std::cout << "          or: " << sse /60/60/24 << "+ days\n";
        std::cout << "          or: " << sse /60/60 << "+ hours\n";
        std::cout << "          or: " << sse /60 << "+ minutes\n";
        std::cout << "          or: " << sse << "+ seconds\n";
        std::cout << "or in ticks: " << tse.count() << '\n';
 }
```

Clock Examples (2)

Clocks can also be used to determine the time passed between two time points:*

```
template<typename TestCode>
void test_timing(unsigned repeat, TestCode testrun) {
    using sc = std::chrono;
    const auto started = sc::high_resolution_clock::now();
    for (auto i = 0; i < repeat; ++i) testrun();
    const auto ended = sc::high_resolution_clock::now();
    const auto delta = ended - started;
    const auto nanosec = sc:duration_cast<sc::nanoseconds>(delta);
    const auto per_run = nanosec.count() / repeat;
    std::cout << nanosec << " ns total for " << repeat << " runs"
                            " = " << per_run << " ns per run\n";
}
…
test_timing(100*1000, // repeat 100.000 times ...
            [](){
                … // ... this some code fragment
            });
```

*: Note that – as far as shown here – real time is measured, not CPU time, but `boost::chrono` has also clocks for measuring CPU time.

`std::chrono` – Operations

Operators are overloaded to support mixed durations and time points:

Operand Type	Operation	Operand Type	Result Type
duration	plus or minus	duration	duration
time point	plus or minus	duration	time point
time point	minus	time point	duration
duration	multiplied with	plain number	duration
duration	divided by	plain number	duration
duration	modulo	plain number	duration
duration	divided by	duration	plain number
duration	modulo	duration	duration

Combinations not listed in the table above result in compile time errors.

For operands with standard types* of different resolution the result will use the appropriate type with the finer grained resolution.

*: When non-standard types are combined the required result type will be calculated accordingly. E.g. to store the sum of a duration counting in 10/21 seconds and another one counting in 14/15 seconds, a result type counting in 1/105 seconds will be used.

Boost: Chrono

Boost.Chrono implements the C++11 conformant Chrono classes with some additions.

Such target the area of measuring CPU-time, i.e.

- clocks to which ticks only get added when the CPU is active for the current process,
- usually making a difference between *User Time* and *System Time*.

(i) For more information on the option to measure CPU time with the additional clocks provided by Boost.Chrono see section Other Clocks in http://www.boost.org/doc/libs/release/doc/html/chrono/reference.html

Boost: Date & Time

Boost.Date_time has little in common with C++11 Chrono, except for maintaining a similar semantic difference between durations and time points.

> While legacy code using that library will still be around for some years, on the long run it can be expected that the importance and user base of this library may decrease and code be updated to use `std::chrono`.*

*: Note that with [Boost.Chrono] the Chrono Library as standardized with C++11 is now available on the Boost platform too.