Chapter 37. Boost.Chrono

The library Boost.Chrono provides a variety of clocks. For example, you can get the current time or you can measure the time passed in a process.

Parts of Boost.Chrono were added to C++11. If your development environment supports C++11, you have access to several clocks defined in the header file `chrono`. However, C++11 doesn't support some features, for example clocks to measure CPU time. Furthermore, only Boost.Chrono supports user-defined output formats for time.

You have access to all Boost.Chrono clocks through the header file `boost/chrono.hpp`. The only extension is user-defined formatting, which requires the header file `boost/chrono_io.hpp`.

Example 37.1. All clocks from Boost.Chrono

```
#include <boost/chrono.hpp>
#include <iostream>

using namespace boost::chrono;

int main()
{
  std::cout << system_clock::now() << '\n';
#ifdef BOOST_CHRONO_HAS_CLOCK_STEADY
  std::cout << steady_clock::now() << '\n';
#endif
  std::cout << high_resolution_clock::now() << '\n';

#ifdef BOOST_CHRONO_HAS_PROCESS_CLOCKS
  std::cout << process_real_cpu_clock::now() << '\n';
  std::cout << process_user_cpu_clock::now() << '\n';
  std::cout << process_system_cpu_clock::now() << '\n';
  std::cout << process_cpu_clock::now() << '\n';
#endif

#ifdef BOOST_CHRONO_HAS_THREAD_CLOCK
  std::cout << thread_clock::now() << '\n';
#endif
}
```

Example 37.1 introduces all of the clocks provided by Boost.Chrono. All clocks have in common the member function `now()`, which returns a timepoint. All timepoints are relative to a universally valid timepoint. This reference timepoint is called *epoch*. An often used epoch is 1 January 1970. Example 37.1 writes the epoch for every timepoint displayed.

Boost.Chrono includes the following clocks:

- `boost::chrono::system_clock` returns the system time. This is the time usually displayed on the desktop of your computer. If you change the time on your computer, `boost::chrono::system_clock` returns the new time. Example 37.1 writes a string to standard output that looks like the following: `13919594042183544 [1/10000000]seconds since Jan 1, 1970`.

  The epoch isn't standardized for `boost::chrono::system_clock`. The epoch 1 January 1970, which is used in these examples, is implementation dependent. However, if you

specifically want to get the time since 1 January 1970, call `to_time_t()`. `to_time_t()` is a static member function that returns the current system time as the number of seconds since 1 January 1970 as a `std::time_t`.

- `boost::chrono::steady_clock` is a clock that will always return a later time when it is accessed later. Even if the time is set back on a computer, `boost::chrono::steady_clock` will return a later time. This time is known as *monotonic time*.

  Example 37.1 displays the number of nanoseconds since the system was booted. The message looks like the following: `10594369282958 nanoseconds since boot`. `boost::chrono::steady_clock` measures the time elapsed since the last boot. However, starting the measurement since the last boot is an implementation detail. The reference point could change with a different implementation.

  `boost::chrono::steady_clock` isn't supported on all platforms. The clock is only available if the macro `BOOST_CHRONO_HAS_CLOCK_STEADY` is defined.

- `boost::chrono::high_resolution_clock` is a type definition for `boost::chrono::system_clock` or `boost::chrono::steady_clock`, depending on which clock measures time more precisely. Thus, the output is identical to the output of the clock `boost::chrono::high_resolution_clock` is based on.

- `boost::chrono::process_real_cpu_clock` returns the CPU time a process has been running. The clock measures the time since program start. Example 37.1 writes a string to standard output that looks like the following: `1000000 nanoseconds since process start-up`.

  You could also get this time using `std::clock()` from `ctime`. In fact, the current implementation of `boost::chrono::process_real_cpu_clock` is based on `std::clock()`.

  The `boost::chrono::process_real_cpu_clock` clock and other clocks measuring CPU time can only be used if the macro `BOOST_CHRONO_HAS_PROCESS_CLOCKS` is defined.

- `boost::chrono::process_user_cpu_clock` returns the CPU time a process spent in *user space*. User space refers to code that runs separately from operating system functions. The time it takes to execute code in operating system functions called by a program is not counted as user space time.

  `boost::chrono::process_user_cpu_clock` returns only the time spent running in user space. If a program is halted for a while, for example through the Windows `Sleep()` function, the time spent in `Sleep()` isn't measured by `boost::chrono::process_user_cpu_clock`.

  Example 37.1 writes a string to standard output that looks like the following: `15600100 nanoseconds since process start-up`.

- `boost::chrono::process_system_cpu_clock` is similar to `boost::chrono::process_user_cpu_clock`. However, this clock measures the time spent in *kernel space*. `boost::chrono::process_system_cpu_clock` returns the CPU time a process spends executing operating system functions.

  Example 37.1 writes a string to the standard output that looks like the following: `0 nanoseconds since process start-up`. Because this example doesn't call operating system functions directly and because Boost.Chrono uses only a few operating system functions, `boost::chrono::process_system_cpu_clock` may return 0.

- `boost::chrono::process_cpu_clock` returns a tuple with the CPU times which are returned by `boost::chrono::process_real_cpu_clock`, `boost::chrono::process_user_cpu_clock` and `boost::chrono::process_system_cpu_clock`. Example 37.1 writes a string to standard output that looks like the following: `{1000000;15600100;0} nanoseconds since process start-up`.

- `boost::chrono::thread_clock` returns the time used by a thread. The time measured by `boost::chrono::thread_clock` is comparable to CPU time, except it is per thread, rather than per process. `boost::chrono::thread_clock` returns the CPU time the thread has been running. It does not distinguish between time spent in user and kernel space.

  `boost::chrono::thread_clock` isn't supported on all platforms. You can only use `boost::chrono::thread_clock` if the macro `BOOST_CHRONO_HAS_THREAD_CLOCK` is defined.

  Boost.Chrono provides the macro, `BOOST_CHRONO_THREAD_CLOCK_IS_STEADY`, to detect whether `boost::chrono::thread_clock` measures monotonic time like `boost::chrono::steady_clock`.

  Example 37.1 writes a string to standard output that looks like the following: `15600100 nanoseconds since thread start-up`.

All of the clocks in Boost.Chrono depend on operating system functions; thus, the operating system determines how precise and reliable the returned times are.

Example 37.2. Adding and subtracting durations using Boost.Chrono

```
#include <boost/chrono.hpp>
#include <iostream>

using namespace boost::chrono;

int main()
{
  process_real_cpu_clock::time_point p = process_real_cpu_clock::now();
  std::cout << p << '\n';
  std::cout << p - nanoseconds{1} << '\n';
  std::cout << p + milliseconds{1} << '\n';
  std::cout << p + seconds{1} << '\n';
  std::cout << p + minutes{1} << '\n';
  std::cout << p + hours{1} << '\n';
}
```

`now()` returns an object of type `boost::chrono::time_point` for all clocks. This type is tightly coupled with a clock because the timepoint is measured relative to a reference timepoint that is defined by a clock. `boost::chrono::time_point` is a template that expects the type of a clock as a parameter. Each clock type provides a type definition for its specialized `boost::chrono::time_point`. For example, the type definition for `process_real_cpu_clock` is `process_real_cpu_clock::time_point`.

Boost.Chrono also provides the class `boost::chrono::duration`, which describes durations. Because `boost::chrono::duration` is also a template, Boost.Chrono provides the six classes `boost::chrono::nanoseconds`, `boost::chrono::milliseconds`, `boost::chrono::microseconds`, `boost::chrono::seconds`, `boost::chrono::minutes`, and `boost::chrono::hours`, which are easier to use.

Boost.Chrono overloads several operators to process timepoints and durations. Example 37.2 subtracts durations from or adds durations to **p** to get new timepoints, which are written to standard output.

Example 37.2 displays all timepoints in nanoseconds. Boost.Chrono automatically uses the smallest unit when timepoints and durations are processed to make sure that results are as precise as possible. If you want to use a timepoint with another unit, you have to cast it.

Example 37.3. Casting timepoints with `boost::chrono::time_point_cast()`

```
#include <boost/chrono.hpp>
#include <iostream>

using namespace boost::chrono;

int main()
{
  process_real_cpu_clock::time_point p = process_real_cpu_clock::now();
  std::cout << p << '\n';
  std::cout << time_point_cast<minutes>(p) << '\n';
}
```

The `boost::chrono::time_point_cast()` function is used like a cast operator. Example 37.3 uses `boost::chrono::time_point_cast()` to convert a timepoint based on nanoseconds to a timepoint in minutes. You must use `boost::chrono::time_point_cast()` in this case because the timepoint cannot be expressed in a less precise unit (minutes) without potentially losing precision. You don't require `boost::chrono::time_point_cast()` to convert from less precise to more precise units.

Boost.Chrono also provides cast operators for durations.

Example 37.4. Casting durations with `boost::chrono::duration_cast()`

```
#include <boost/chrono.hpp>
#include <iostream>

using namespace boost::chrono;

int main()
{
```

```
  minutes m{1};
  seconds s{35};

  std::cout << m + s << '\n';
  std::cout << duration_cast<minutes>(m + s) << '\n';
}
```

Example 37.4 uses the function `boost::chrono::duration_cast()` to cast a duration from seconds to minutes. This example writes `1 minute` to standard output.

Example 37.5. Rounding durations

```
#include <boost/chrono.hpp>
#include <iostream>

using namespace boost::chrono;

int main()
{
  std::cout << floor<minutes>(minutes{1} + seconds{45}) << '\n';
  std::cout << round<minutes>(minutes{1} + seconds{15}) << '\n';
  std::cout << ceil<minutes>(minutes{1} + seconds{15}) << '\n';
}
```

Boost.Chrono also provides functions to round durations when casting. `boost::chrono::round()` rounds up or down, `boost::chrono::floor()` rounds down, and `boost::chrono::ceil()` rounds up. `boost::chrono::floor()` uses `boost::chrono::duration_cast()` – there is no difference between these two functions.

Example 37.5 writes `1 minute`, `1 minute`, and `2 minutes` to standard output.

Example 37.6. Stream manipulators for user-defined output

```
#define BOOST_CHRONO_VERSION 2
#include <boost/chrono.hpp>
#include <boost/chrono/chrono_io.hpp>
#include <iostream>

using namespace boost::chrono;

int main()
{
  std::cout << symbol_format << minutes{10} << '\n';

  std::cout << time_fmt(boost::chrono::timezone::local, "%H:%M:%S") <<
    system_clock::now() << '\n';
}
```

Boost.Chrono provides various stream manipulators to format the output of timepoints and durations. For example, with the manipulator `boost::chrono::symbol_format()`, the time unit is written as a symbol instead of a name. Thus, Example 37.6 displays `10 min`.

The manipulator `boost::chrono::time_fmt()` can be used to set a timezone and a format string. The timezone must be set to `boost::chrono::timezone::local` or `boost::chrono::timezone::utc`. The format string can use flags to refer to various components of a timepoint. For example, Example 37.6 writes a string to the standard output that looks like the following: `15:46:44`.

Beside stream manipulators, Boost.Chrono provides facets for many different customizations. For example, there is a facet that makes it possible to output timepoints in another language.

Note

There are two versions of the input/output functions since Boost 1.52.0. Since Boost 1.55.0, the newer version is used by default. If you use a version older than 1.55.0, you must define the macro `BOOST_CHRONO_VERSION` and set it to 2 for Example 37.6 to work.