
5.7. Clocks and Timers

One of the most obvious libraries a programming language should have is one to deal with date and time. However, experience shows that such a library is harder to design than it sounds. The problem is the amount of flexibility and precision the library should provide. In fact, in the past, the interfaces to system time provided by C and POSIX switched from seconds to milliseconds, then to microseconds, and finally to nanoseconds. The problem was that for each switch, a new interface was provided. For this reason, a precision-neutral library was proposed for C++11. This library is usually called the *chrono library* because its features are defined in `<chrono>`.

In addition, the C++ standard library provides the basic C and POSIX interfaces to deal with calendar time. Finally, you can use the thread library, provided since C++11, to wait for a thread or the program (the main thread) for a period of time.

5.7.1. Overview of the Chrono Library

The chrono library was designed to be able to deal with the fact that timers and clocks might be different on different systems and improve over time in precision. To avoid having to introduce a new time type every 10 years or so — as happened with the POSIX time libraries, for example — the goal was to provide a precision-neutral concept by separating duration and point of time ("timepoint") from specific clocks. As a result, the core of the chrono library consists of the following types or concepts, which serve as abstract mechanisms to specify and deal with points in and durations of time:

- A **duration** of time is defined as a specific number of ticks over a time unit. One example is a duration such as "3 minutes" (3 ticks of a "minute"). Other examples are "42 milliseconds" or "86,400 seconds," which represents the duration of 1 day. This concept also allows specifying something like "1.5 times a third of a second," where 1.5 is the number of ticks and "a third of a second" the time unit used.
- A **timepoint** is defined as combination of a duration and a beginning of time (the so-called **epoch**). A typical example is a timepoint that represents "New Year's Midnight 2000," which is described as "1,262,300,400 seconds since January 1, 1970" (this day is the epoch of the system clock of UNIX and POSIX systems).
- The concept of a timepoint, however, is parametrized by a **clock**, which is the object that defines the epoch of a timepoint. Thus, different clocks have different epochs. In general, operations dealing with multiple timepoints, such as processing the duration/difference between two time-points, require using the same epoch/clock. A clock also provides a convenience function to yield the timepoint of *now*.

In other words, timepoint is defined as a duration before or after an epoch, which is defined by a clock (see Figure 5.4).

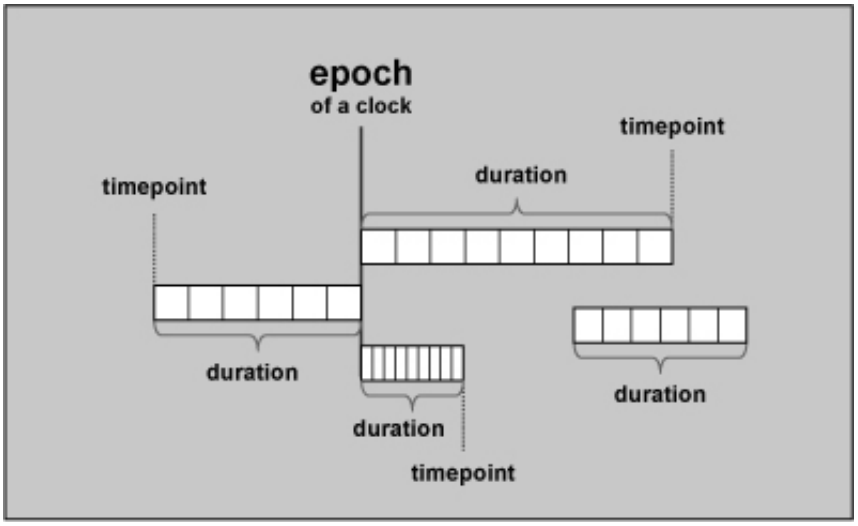

Figure 5.4. Epoch, Durations, and Timepoints

For more details about the motivation and design of these classes, see [*N2661:Chrono*].[27] Let's look into these types and concepts in detail.

[27] I use some quotes of [*N2661:Chrono*] in this book with friendly permission by the authors.

Note that all identifiers of the chrono library are defined in namespace `std::chrono`.

5.7.2. Durations

A duration is a combination of a **value** representing the number of ticks and a **fraction** representing the unit in seconds. Class `ratio` is used to specify the fraction (see Section 5.6, page 140). For example:

Click here to view code image

```
std::chrono::duration<int>                    twentySeconds(20);
std::chrono::duration<double,std::ratio<60>>  halfAMinute(0.5);
```

```
std::chrono::duration<long,std::ratio<1,1000>> oneMillisecond(1);
```

Here, the first template argument defines the type of the ticks, and the optional second template argument defines the unit type in seconds.

Thus, the first line uses seconds as unit type, the second line uses minutes ("$\frac{60}{1}$ seconds"), and the third line uses milliseconds ("$\frac{1}{1000}$ of a second").

For more convenience, the C++ standard library provides the following type definitions:

Click here to view code image

```
namespace std {
  namespace chrono {
    typedef duration<signed int-type >= 64 bits,nano>         nanoseconds;
    typedef duration<signed int-type >= 55 bits,micro>        microseconds;
    typedef duration<signed int-type >= 45 bits,milli>        milliseconds;
    typedef duration<signed int-type >= 35 bits>              seconds;
    typedef duration<signed int-type >= 29 bits,ratio<60>>    minutes;
    typedef duration<signed int-type >= 23 bits,ratio<3600>> hours;
  }
}
```

With them, you can easily specify typical time periods:

```
std::chrono::seconds      twentySeconds(20);
std::chrono::hours        aDay(24);
std::chrono::milliseconds oneMillisecond(1);
```

Arithmetic Duration Operations

You can compute with durations in the expected way (see Table 5.21):

- You can process the sum, difference, product, or quotient of two durations.
- You can add or subtract ticks or other durations.
- You can compare two durations.

Table 5.21. Arithmetic Operations of *duration*s

Operation	Effect
d1 + *d2*	Process sum of durations *d1* and *d2*
d1 - *d2*	Process difference of durations *d1* and *d2*
d * *val*	Return result of *val* times duration *d*
val * *d*	Return result of *val* times duration *d*
d / *val*	Return of the duration *d* divided by value *val*
d1 / *d2*	Compute factor between durations *d1* and *d2*
d % *val*	Result of duration *d* modulo value *val*
d % *d2*	Result of duration *d* modulo the value of *d2*
d1 == *d2*	Return whether duration *d1* is equal to duration *d2*
d1 != *d2*	Return whether duration *d1* differs from duration *d2*
d1 < *d2*	Return whether duration *d1* is shorter than duration *d2*
d1 <= *d2*	Return whether duration *d1* is not longer than duration *d2*
d1 > *d2*	Return whether duration *d1* is longer than duration *d2*
d1 <= *d2*	Return whether duration *d1* is not shorter than duration *d2*
++*d*	Increment duration *d* by 1 tick
d++	Increment duration *d* by 1 tick
--*d*	Decrement duration *d* by 1 tick
d--	Decrement duration *d* by 1 tick
d += *d1*	Extend the duration *d* by the duration *d1*
d -= *d1*	Shorten the duration *d* by the duration *d1*
d *= *val*	Multiply the duration *d* by *val*
d /= *val*	Divide the duration *d* by *val*
d %= *val*	Process duration *d* modulo *val*
d %= *d2*	Process duration *d* modulo the value of *d2*

The important point here is that the unit type of two durations involved in such an operation might be different. Due to a provided overloading of `common_type<>` (see Section 5.4.1, page 124) for `duration` s, the resulting duration will have a unit that is the greatest common divisor of the units of both operands.

For example, after

```
chrono::seconds      d1(42);      // 42 seconds
chrono::milliseconds d2(10);      // 10 milliseconds
```

the expression

```
d1 - d2
```

yields a duration of 41,990 ticks of unit type milliseconds ($\frac{1}{1000}$ seconds).

Or, more generally, after

Click here to view code image

```
chrono::duration<int,ratio<1,3>> d1(1);      // 1 tick of 1/3 second
chrono::duration<int,ratio<1,5>> d2(1);      // 1 tick of 1/5 second
```

the expression

```
d1 + d2
```

yields 8 ticks of $\frac{1}{15}$ second and

```
d1 < d2
```

yields `false` . In both cases, `d1` gets expanded to 5 ticks of $\frac{1}{15}$ second, and `d2` gets expanded to 3 ticks of $\frac{1}{15}$ second. So the sum of 3 and 5 is 8, and 5 is not less than 3.

You can also convert durations into durations of different units, as long as there is an implicit type conversion. Thus, you can convert hours into seconds but not the other way around. For example:

```
std::chrono::seconds twentySeconds(20);   // 20 seconds
std::chrono::hours   aDay(24);            // 24 hours

std::chrono::milliseconds ms(0);          // 0 milliseconds (undefined value
without (0) !)
ms += twentySeconds + aDay;               // 86,420,000 milliseconds
--ms;                                     // 86,419,999 milliseconds
ms *= 2;                                  // 172,839,998 milliseconds
std::cout << ms.count() << " ms" << std::endl;
std::cout << std::chrono::nanoseconds(ms).count() << " ns" << std::endl;
```

These conversions result in the following output:

```
172839998 ms
172839998000000 ns
```

Other Duration Operations

In the preceding example, we use the member `count()` to yield the current number of ticks, which is one of the other operations provided for durations. Table 5.22 lists all operations, members, and types available for durations besides the arithmetic operations of Table 5.21. Note that the default constructor default-initializes (see Section 3.2.1, page 37) its value, which means that for fundamental representation types, the initial value is undefined.

Table 5.22. Other Operations and Types of *duration*s

Operation	Effect
duration d	Default constructor; creates duration (default-initialized)
duration d(d2)	Copy constructor; copies duration (*d2* might have a different unit type)
duration d(val)	Creates duration of *val* ticks of *d*s unit type
d = d2	Assigns duration *d2* to *d* (implicit conversion possible)
d.count()	Returns ticks of the duration *d*
duration_cast<*D*>(*d*)	Returns duration *d* explicitly converted into type *D*
duration::zero()	Yields duration of zero length
duration::max()	Yields maximum possible duration of this type
duration::min()	Yields minimum possible duration of this type
duration::rep	Yields the type of the ticks
duration::period	Yields the type of the unit type

You can use these members to define a convenience function for the output operator `<<` for durations:[28]

[28] Note that this output operator does not work where *ADL* (*argument-dependent lookup*) does not work (see Section 15.11.1, page 812, for details).

```
template <typename V, typename R>
ostream& operator << (ostream& s, const chrono::duration<V,R>& d)
{
    s << "[" << d.count() << " of " << R::num << "/"
                                    << R::den << "]";
    return s;
}
```

Here, after printing the number of ticks with `count()`, we print the numerator and denominator of the unit type used, which is a `ratio` processed at compile time (see Section 5.6, page 140). For example,

```
std::chrono::milliseconds d(42);
std::cout << d << std::endl;
```

will then print:

```
[42 of 1/1000]
```

As we have seen, implicit conversions to a more precise unit type are always possible. However, conversions to a coarser unit type are not, because you might lose information. For example, when converting an integral value of 42,010 milliseconds into seconds, the resulting integral value, 42, means that the precision of having a duration of 10 milliseconds over 42 seconds gets lost. But you can still explicitly force such a conversion with a `duration_cast`. For example:

```
std::chrono::seconds sec(55);
std::chrono::minutes m1 = sec;        // ERROR
std::chrono::minutes m2 =
  std::chrono::duration_cast<std::chrono::minutes>(sec);  // OK
```

As another example, converting a duration with a floating-point tick type also requires an explicit cast to convert it into an integral duration type:

```
std::chrono::duration<double,std::ratio<60>> halfMin(0.5);
std::chrono::seconds s1 = halfMin;    // ERROR
std::chrono::seconds s2 =
  std::chrono::duration_cast<std::chrono::seconds>(halfMin);  // OK
```

A typical example is code that segments a duration into different units. For example, the following code segments a duration of milliseconds into the corresponding hours, minutes, seconds, and milliseconds (to output the first line starting with `raw:` we use the output operator just defined):

```
using namespace std;
using namespace std::chrono;
milliseconds ms(7255042);

// split into hours, minutes, seconds, and milliseconds
hours   hh = duration_cast<hours>(ms);
minutes mm = duration_cast<minutes>(ms % chrono::hours(1));
seconds ss = duration_cast<seconds>(ms % chrono::minutes(1));
milliseconds msec
           = duration_cast<milliseconds>(ms % chrono::seconds(1));

// and print durations and values:
cout << "raw: " << hh << "::" << mm << "::"
                << ss << "::" << msec << endl;
cout << "     " << setfill('0') << setw(2) << hh.count() << "::"
                               << setw(2) << mm.count() << "::"
                               << setw(2) << ss.count() << "::"
                               << setw(3) << msec.count() << endl;
```

Here, the cast

```
std::chrono::duration_cast<std::chrono::hours>(ms)
```

converts the milliseconds into hours, where the values are truncated, not rounded. Thanks to the modulo operator `%`, for which you can even pass a duration as second argument, you can easily process the remaining milliseconds with `ms % std::chrono::hours(1)`, which is then converted into minutes. Thus, the output of this code will be as follows:

```
raw: [2 of 3600/1]::[0 of 60/1]::[55 of 1/1]::[42 of 1/1000]
     02::00::55::042
```

Finally, class `duration` provides three static functions: `zero()`, which yields a duration of 0 seconds, as well as `min()` and `max()`, which yield the minimum and maximum value a duration can have.

5.7.3. Clocks and Timepoints

The relationships between timepoints and clocks are a bit tricky:

- A **clock** defines an epoch and a tick period. For example, a clock might tick in milliseconds since the UNIX epoch (January 1, 1970) or tick in nanoseconds since the start of the program. In addition, a clock provides a type for any timepoint specified according to this clock.

  The interface of a clock provides a function `now()` to yield an object for the current point in time.

- A **timepoint** represents a specific point in time by associating a positive or negative duration to a given clock. Thus, if the duration is "10 days" and the associated clock has the epoch of January 1, 1970, the timepoint represents January 11, 1970.

  The interface of a timepoint provides the ability to yield the epoch, minimum and maximum timepoints according to the clock, and timepoint arithmetic.

Clocks

Table 5.23 lists the type definitions and static members required for each clock.

Table 5.23. Operations and Types of Clocks

Operation	Effect
clock::duration	Yields the duration type of the clock
clock::rep	Yields the type of the ticks (equivalent to *clock*::duration::rep)
clock::period	Yields the type of the unit type (equivalent to *clock*::duration::period)
clock::time_point	Yields the timepoint type of the clock
clock::is_steady	Yields true if the clock is steady
clock::now()	Yields a time_point for the current point in time

The C++ standard library provides three clocks, which provide this interface:

1. The **system_clock** represents timepoints associated with the usual real-time clock of the current system. This clock also provides convenience functions to_time_t() and from_time_t() to convert between any timepoint and the C system time type time_t, which means that you can convert into and from calendar times (see Section 5.7.4, page 158).
2. The **steady_clock** gives the guarantee that it never gets adjusted.[29] Thus, timepoint values never decrease as the physical time advances, and they advance at a steady rate relative to real time.

[29] The steady_clock was initially proposed as monotonic_clock.

3. The **high_resolution_clock** represents a clock with the shortest tick period possible on the current system.

Note that the standard does not provide requirements for the precision, the epoch, and the range (minimum and maximum timepoints) of these clocks. For example, your system clock might have the UNIX epoch (January 1, 1970) as epoch, but this is not guaranteed. If you require a specific epoch or care for timepoints that might not be covered by the clock, you have to use convenience functions to find it out.

For example, the following function prints the properties of a clock:

Click here to view code image

```
// util/clock.hpp

#include <chrono>
#include <iostream>
#include <iomanip>

template <typename C>
void printClockData ()
{
    using namespace std;

    cout << "- precision: ";
    // if time unit is less than or equal to one millisecond
    typedef typename C::period P;     // type of time unit
    if (ratio_less_equal<P,milli>::value) {
        // convert to and print as milliseconds
        typedef typename ratio_multiply<P,kilo>::type TT;
        cout << fixed << double(TT::num)/TT::den
             << " milliseconds" << endl;
    }
    else {
        // print as seconds
        cout << fixed << double(P::num)/P::den << " seconds" << endl;
    }
    cout << "- is_steady: " << boolalpha << C::is_steady << endl;
}
```

We can call this function for the various clocks provided by the C++ standard library:

```
// util/clock1.cpp

#include <chrono>
#include "clock.hpp"

int main()
{
    std::cout << "system_clock: " << std::endl;
    printClockData<std::chrono::system_clock>();
    std::cout << "\nhigh_resolution_clock: " << std::endl;
    printClockData<std::chrono::high_resolution_clock>();
    std::cout << "\nsteady_clock: " << std::endl;
    printClockData<std::chrono::steady_clock>();
}
```

The program might, for example, have the following output:

```
system_clock:
- precision: 0.000100 milliseconds
- is_steady: false

high_resolution_clock:
- precision: 0.000100 milliseconds
- is_steady: true

steady_clock:
- precision: 1.000000 milliseconds
- is_steady: true
```

Here, for example, the system and the high-resolution clock have the same precision of 100 nanoseconds, whereas the steady clock uses milliseconds. You can also see that both the steady clock and high-resolution clock can't be adjusted. Note, however, that this might be very different on other systems. For example, the high-resolution clock might be the same as the system clock.

The `steady_clock` is important to compare or compute the difference of two times in your program, where you processed the current point in time. For example, after

```
auto system_start = chrono::system_clock::now();
```

a condition to check whether the program runs more than one minute:

```
if (chrono::system_clock::now() > system_start + minutes(1))
```

might not work, because if the clock was adjusted in the meantime, the comparison might yield `false`, although the program did run more than a minute. Similarly, processing the elapsed time of a program:

```
auto diff = chrono::system_clock::now() - system_start;
auto sec = chrono::duration_cast<chrono::seconds>(diff);
cout << "this program runs:   " << s.count() << " seconds" << endl;
```

might print a negative duration if the clock was adjusted in the meantime. For the same reason, using timers with other than the `steady_clock` might change their duration when the system clock gets adjusted (see Section 5.7.5, page 160, for details).

Timepoints

With any of these clocks — or even with user-defined clocks — you can deal with timepoints. Class `time_point` provides the corresponding interface, parametrized by a clock:

```
namespace std {
  namespace chrono {
    template <typename Clock,
              typename Duration = typename Clock::duration>
        class time_point;
  }
}
```

Four specific timepoints play a special role:

1. The **epoch**, which the default constructor of class `time_point` yields for each clock.
2. The **current time**, which the static member function `now()` of each clock yields (see Section 5.7.3, page 149).
3. The **minimum timepoint**, which the static member function `min()` of class `time_point` yields for each clock.
4. The **maximum timepoint**, which the static member function `max()` of class `time_point` yields for each clock.

For example, the following program assigns these timepoints to `tp` and prints them converted into a calendar notation:

```
// util/chrono1.cpp

#include <chrono>
#include <ctime>
#include <string>
#include <iostream>

std::string asString (const std::chrono::system_clock::time_point& tp)
{
    // convert to system time:
    std::time_t t = std::chrono::system_clock::to_time_t(tp);
    std::string ts = std::ctime(&t);    // convert to calendar time
    ts.resize(ts.size()-1);             // skip trailing newline
    return ts;
}
```

```
int main()
{
    // print the epoch of this system clock:
    std::chrono::system_clock::time_point tp;
    std::cout << "epoch: " << asString(tp) << std::endl;

    // print current time:
    tp = std::chrono::system_clock::now();
    std::cout << "now:   " << asString(tp) << std::endl;

    // print minimum time of this system clock:
    tp = std::chrono::system_clock::time_point::min();
    std::cout << "min:   " << asString(tp) << std::endl;

    // print maximum time of this system clock:
    tp = std::chrono::system_clock::time_point::max();
    std::cout << "max:   " << asString(tp) << std::endl;
}
```

After including `<chrono>`, we first declare a convenience function `asString()`, which converts a timepoint of the system clock into the corresponding calendar time. With

```
std::time_t t = std::chrono::system_clock::to_time_t(tp);
```

we use the static convenience function `to_time_t()`, which converts a timepoint into an object of the traditional time type of C and POSIX, type `time_t`, which usually represents the number of seconds since the UNIX epoch, January 1, 1970 (see Section 5.7.4, page 157). Then,

```
std::string ts = std::ctime(&t);
```

uses `ctime()` to convert this into a calendar notation, for which

```
ts.resize(ts.size()-1);
```

removes the trailing newline character.

Note that `ctime()` takes the local time zone into account, which has consequences we will discuss shortly. Note also that this convenience function probably will work only for `system_clock`s, the only clocks that provide an interface for conversions to and from `time_t`. For other clocks, such an interface might also work but is not portable, because the other clocks are not required to have epoch of the system time as their internal epoch.

Note also that the output format for timepoints might better get localized by using the `time_put` facet. See Section 16.4.3, page 884, for details, and page 886 for an example.

Inside `main()`, the type of `tp`, declared as

```
std::chrono::system_clock::time_point
```

is equivalent to:[30]

[30] According to the standard, a `system_clock::time_point` could also be identical to `time_point<C2,system_clock::duration>`, where `C2` is a different clock but has the same epoch as `system_clock`.

```
std::chrono::time_point<std::chrono::system_clock>
```

Thus, `tp` is declared as the timepoint of the `system_clock`. Having the clock as template argument ensures that only timepoint arithmetic with the same clock (epoch) is possible.

The program might have the following output:

```
epoch: Thu Jan  1 01:00:00 1970
now:   Sun Jul 24 19:40:46 2011
min:   Sat Mar  5 18:27:38 1904
max:   Mon Oct 29 07:32:22 2035
```

Thus, the default constructor, which yields the epoch, creates a timepoint, which `asString()` converts into

```
Thu Jan 1 01:00:00 1970
```

Note that it's 1 o'clock rather than midnight. This may look a bit surprising, but remember that the conversion to the calendar time with `ctime()` inside `asString()` takes the time zone into account. Thus, the UNIX epoch used here — which, again, is not always

guaranteed to be the epoch of the system time — started at 00:00 in Greenwich, UK. In my time zone, Germany, it was 1 a.m. at that moment, so in my time zone the epoch started at 1 a.m. on January 1, 1970. Accordingly, if you start this program, your output is probably different, according to your time zone, even if your system uses the same epoch in its system clock.

To have the universal time (UTC) instead, you should use the following conversion rather than calling `ctime()`, which is a shortcut for `asctime(localtime(...))` (see Section 5.7.4, page 157):

```
std::string ts = std::asctime(gmtime(&t));
```

In that case, the output of the program would be:

```
epoch: Thu Jan  1 00:00:00 1970
now:   Sun Jul 24 17:40:46 2011
min:   Sat Mar  5 17:27:38 1904
max:   Mon Oct 29 06:32:22 2035
```

Yes, here, the difference is 2 hours for `now()`, because this timepoint is when summer time is used, which leads to a 2-hour difference to UTC in Germany.

In general, `time_point` objects have only one member, the duration, which is relative to the epoch of the associated clock. The timepoint value can be requested by `time_since_epoch()`. For timepoint arithmetic, any useful combination of a timepoint and another timepoint or duration is provided (see Table 5.24).

Table 5.24. Operations of *time_point*s

Operation	Yields	Effect
timepoint t	*timepoint*	Default constructor; creates a timepoint representing the epoch
timepoint t(tp2)	*timepoint*	Creates a timepoint equivalent to *tp2* (the duration unit might be finer grained)
timepoint t(d)	*timepoint*	Creates a timepoint having duration *d* after the epoch
time_point_cast<*C*,*D*>(*tp*)	*timepoint*	Converts *tp* into a timepoint with clock *C* and duration *D* (which might be more coarse grained)
tp += *d*	*timepoint*	Adds duration *d* to the current timepoint *tp*
tp -= *d*	*timepoint*	Subtracts duration *d* from the current timepoint *tp*
tp + *d*	*timepoint*	Returns a new timepoint of *tp* with duration *d* added
d + *tp*	*timepoint*	Returns a new timepoint of *tp* with duration *d* added
tp - *d*	*timepoint*	Returns a new timepoint of *tp* with duration *d* subtracted
tp1 - *tp2*	*duration*	Returns the duration between timepoints *tp1* and *tp2*
tp1 == *tp2*	bool	Returns whether timepoint *tp1* is equal to timepoint *tp2*
tp1 != *tp2*	bool	Returns whether timepoint *tp1* differs from timepoint *tp2*
tp1 < *tp2*	bool	Returns whether timepoint *tp1* is before timepoint *tp2*
tp1 <= *tp2*	bool	Returns whether timepoint *tp1* is not after timepoint *tp2*
tp1 > *tp2*	bool	Returns whether timepoint *tp1* is after timepoint *tp2*
tp1 >= *tp2*	bool	Returns whether timepoint *tp1* is not before timepoint *tp2*
tp.time_since_epoch()	*duration*	Returns the duration between the epoch and timepoint *tp*
timepoint::min()	*timepoint*	Returns the first possible timepoint of type *timepoint*
timepoint::max()	*timepoint*	Returns the last possible timepoint of type *timepoint*

Although the interface uses class `ratio` (see Section 5.6, page 140), which ensures that over-flows by the duration units yield a compile-time error, overflows on the duration values are possible. Consider the following example:

```
// util/chrono2.cpp

#include <chrono>
#include <ctime>
#include <iostream>
#include <string>
using namespace std;

string asString (const chrono::system_clock::time_point& tp)
{
    time_t t = chrono::system_clock::to_time_t(tp); // convert to system time
    string ts = ctime(&t);                          // convert to calendar time
    ts.resize(ts.size()-1);                         // skip trailing newline
    return ts;
}
```

```
int main()
{
    // define type for durations that represent day(s):
    typedef chrono::duration<int,ratio<3600*24>> Days;

    // process the epoch of this system clock
    chrono::time_point<chrono::system_clock> tp;
    cout << "epoch:     " << asString(tp) << endl;

    // add one day, 23 hours, and 55 minutes
    tp += Days(1) + chrono::hours(23) + chrono::minutes(55);
    cout << "later:     " << asString(tp) << endl;

    // process difference from epoch in minutes and days:
    auto diff = tp - chrono::system_clock::time_point();
    cout << "diff:    "
         << chrono::duration_cast<chrono::minutes>(diff).count()
         << " minute(s)" << endl;
    Days days = chrono::duration_cast<Days>(diff);
    cout << "diff:      " << days.count() << " day(s)" << endl;

    // subtract one year (hoping it is valid and not a leap year)
    tp -= chrono::hours(24*365);
    cout << "-1 year:   " << asString(tp) << endl;

    // subtract 50 years (hoping it is valid and ignoring leap years)
    tp -= chrono::duration<int,ratio<3600*24*365>>(50);
    cout << "-50 years: " << asString(tp) << endl;

    // subtract 50 years (hoping it is valid and ignoring leap years)
    tp -= chrono::duration<int,ratio<3600*24*365>>(50);
    cout << "-50 years: " << asString(tp) << endl;
}
```

First, expressions, such as

```
tp = tp + Days(1) + chrono::hours(23) + chrono::minutes(55);
```

or

```
tp -= chrono::hours(24*365);
```

allow adjusting timepoints by using timepoint arithmetic.

Because the precision of the system clock usually is better than minutes and days, you have to explicitly cast the difference between two timepoints to become days:

```
auto diff = tp - chrono::system_clock::time_point();
Days days = chrono::duration_cast<Days>(diff);
```

Note, however, that these operations do not check whether a computation performs an overflow. On my system, the output of the program is as follows:

```
epoch:     Thu Jan  1 01:00:00 1970
later:     Sat Jan  3 00:55:00 1970
diff:      2875 minute(s)
diff:      1 day(s)
-1 year:   Fri Jan  3 00:55:00 1969
-50 years: Thu Jan 16 00:55:00 1919
-50 years: Sat Mar  5 07:23:16 2005
```

You can see the following:

- The cast uses `static_cast<>` for the destination unit, which for ordinary integral unit types means that values are truncated instead of rounded. For this reason, a duration of 47 hours and 55 minutes converts into 1 day.
- Subtracting 50 years of 365 days does not take leap years into account, so the resulting day is January 16 instead of January 3.
- When deducting another 50 years the timepoint goes below the minimum timepoint, which is March 5, 1904 on my system (see Section 5.7.3, page 152), so the result is the year 2005. No error processing is required by the C++ standard library in this case.

This demonstrates that chrono is a duration and a timepoint but not a date/time library. You can compute with durations and timepoints but still have to take epoch, minimum and maximum timepoints, leap years, and leap seconds into account.

5.7.4. Date and Time Functions by C and POSIX

The C++ standard library also provides the standard C and POSIX interfaces to deal with date and time. In `<ctime>`, the macros, types, and functions of `<time.h>` are available in namespace `std`. The types and functions are listed in Table 5.25. In addition,

the macro `CLOCKS_PER_SEC` defines the unit type of `clock()` (which returns the elapsed CPU time in $\frac{1}{CLOCKS\_PER\_SEC}$ seconds). See Section 16.4.3, page 884, for some more details and examples using these time functions and types.

Table 5.25. Definitions in `<ctime>`

Identifier	Meaning
`clock_t`	Type of numeric values of elapsed CPU time returned by `clock()`
`time_t`	Type of numeric values representing timepoints
`struct tm`	Type of "broken down" calendar time
`clock()`	Yields the elapsed CPU time in $\frac{1}{CLOCKS\_PER\_SEC}$ seconds
`time()`	Yields the current time as numeric value
`difftime()`	Yields the difference of two `time_t` in seconds as `double`
`localtime()`	Converts a `time_t` into a `struct tm` taking time zone into account
`gmtime()`	Converts a `time_t` into a `struct tm` not taking time zone into account
`asctime()`	Converts a `struct tm` into a standard calendar time string
`strftime()`	Converts a `struct tm` into a user-defined calendar time string
`ctime()`	Converts a `time_t` into a standard calendar time string taking time zone into account (shortcut for `asctime(localtime(t))`)
`mktime()`	Converts a `struct tm` into a `time_t` and queries weekday and day of the year

Note that `time_t` usually is just the number of seconds since the UNIX epoch, which is January 1, 1970. However, according to the C and C++ standard, this is not guaranteed.

Conversions between Timepoints and Calendar Time

The convenience function to transfer a timepoint to a calendar time string was already discussed in Section 5.7.3, page 153. Here is a header file that also allows converting calendar times into timepoints:

```
// util/timepoint.hpp

#include <chrono>
#include <ctime>
#include <string>

// convert timepoint of system clock to calendar time string
inline
std::string asString (const std::chrono::system_clock::time_point& tp)
{
    // convert to system time:
    std::time_t t = std::chrono::system_clock::to_time_t(tp);
    std::string ts = ctime(&t);     // convert to calendar time
    ts.resize(ts.size()-1);         // skip trailing newline
    return ts;
}

// convert calendar time to timepoint of system clock
inline
std::chrono::system_clock::time_point
makeTimePoint (int year, int mon, int day,
               int hour, int min, int sec=0)
{
    struct std::tm t;
    t.tm_sec = sec;         // second of minute (0 .. 59 and 60 for leap seconds)
    t.tm_min = min;         // minute of hour (0 .. 59)
    t.tm_hour = hour;       // hour of day (0 .. 23)
    t.tm_mday = day;        // day of month (1 .. 31)
    t.tm_mon = mon-1;       // month of year (0 .. 11)
    t.tm_year = year-1900; // year since 1900
    t.tm_isdst = -1;        // determine whether daylight saving time
    std::time_t tt = std::mktime(&t);
    if (tt == -1) {
        throw "no valid system time";
    }
    return std::chrono::system_clock::from_time_t(tt);
}
```

The following program demonstrates these convenience functions:

```
// util/timepoint1.cpp
```

```
#include <chrono>
#include <iostream>
#include "timepoint.hpp"

int main()
{
    auto tp1 = makeTimePoint(2010,01,01,00,00);
    std::cout << asString(tp1) << std::endl;

    auto tp2 = makeTimePoint(2011,05,23,13,44);
    std::cout << asString(tp2) << std::endl;
}
```

The program has the following output:

```
Fri Jan  1 00:00:00 2010
Mon May 23 13:44:00 2011
```

Note again that both `makeTimePoint()` and `asString()` take the local time zone into account. For this reason, the date passed to `makeTimePoint()` matches the output with `asString()`. Also, it doesn't matter whether daylight saving time is used (passing a negative value to `t.tm_isdst` in `makeTimePoint()` causes `mktime()` to attempt to determine whether daylight saving time is in effect for the specified time).

Again, to let `asString()` use the universal time UTC instead, use `asctime(gmtime(...))` rather than `ctime(...)`. For `mktime()`, there is no specified way to use UTC, so `makeTimePoint()` always takes the current time zone into account.

Section 16.4.3, page 884, demonstrates how to use locales to internationalize the reading and writing of time data.

5.7.5. Blocking with Timers

Durations and timepoints can be used to block threads or programs (i.e., the main thread). These blocks can be conditionless or can be used to specify a maximum duration when waiting for a lock, a condition variable, or another thread to end (see Chapter 18):

- `sleep_for()` and `sleep_until()` are provided by `this_thread` to block threads (see Section 18.3.7, page 981).
- `try_lock_for()` and `try_lock_until()` are provided to specify a maximum interval when waiting for a mutex (see Section 18.5.1, page 994).
- `wait_for()` and `wait_until()` are provided to specify a maximum interval when waiting for a condition variable or a future (see Section 18.1.1, page 953 or Section 18.6.4, page 1010).

All the blocking functions that end with ... `_for()` use a duration, whereas all functions that end with ... `_until()` use a timepoint as argument. For example,

```
this_thread::sleep_for(chrono::seconds(10));
```

blocks the current thread, which might be the main thread, for 10 seconds, whereas

```
this_thread::sleep_until(chrono::system_clock::now()
                         + chrono::seconds(10));
```

blocks the current thread until the system clock has reached a timepoint 10 seconds later than now.

Although these calls look the same, they are not! For all ... `_until()` functions, where you pass a timepoint, time adjustments might have an effect. If, during the 10 seconds after calling `sleep_until()`, the system clock gets adjusted, the timeout will be adjusted accordingly. If, for example, we wind the system clock back 1 hour, the program will block for 60 minutes and 10 seconds. If, for example, we adjust the clock forward for more than 10 seconds, the timer will end immediately.

If you use a ... `_for()` function, such as `sleep_for()`, where you pass a duration, or if you use the `steady_clock`, adjustments of the system clock *usually* will have no effect on the duration of timers. However, on hardware where a steady clock is not available, and thus the platform gives no chance to count seconds independently of a possibly adjusted system time, time adjustments can also impact the ... `_for()` functions.

All these timers do not guarantee to be exact. For any timer, there will be a delay because the system only periodically checks for expired timers, and the handling of timers and interrupts takes some time. Thus, durations of timers will take their specified time plus a period that depends on the quality of implementation and the current situation.