Boost.Chrono 1.0.0

Howard Hinnant

Beman Dawes

Vicente J. Botet Escriba

Table of Contents

Overview

> “What is time, then? If nobody asks me, I know; if I have to explain it to someone who has asked me, I do not know.""
>
> **-- Augustine**

How to Use This Documentation

This documentation makes use of the following naming and formatting conventions.

- Code is in `fixed width font` and is syntax-highlighted.
- Replaceable text that you will need to supply is in *`italics`*.
- Free functions are rendered in the code font followed by `()`, as in `free_function()`.

- If a name refers to a class template, it is specified like this: `class_template<>`; that is, it is in code font and its name is followed by <> to indicate that it is a class template.
- If a name refers to a function-like macro, it is specified like this: `MACRO()`; that is, it is uppercase in code font and its name is followed by `()` to indicate that it is a function-like macro. Object-like macros appear without the trailing `()`.
- Names that refer to *concepts* in the generic programming sense are specified in CamelCase.

Note

In addition, notes such as this one specify non-essential information that provides additional background or rationale.

Finally, you can mentally add the following to any code fragments in this document:

```
// Include all of Chrono files
#include <boost/chrono.hpp>
```

Motivation

Time

We all deal with time every day of our lives. We've intuitively known it since birth. Thus we are all very familiar with it and believe it to be a simple matter. The modeling of time in computer programs should be similarly simple. The unfortunate truth is that this perceived simplicity is only skin deep. Fortunately, we do not need a terribly complicated solution to meet the bulk of our needs. However, overly simplistic solutions can be dangerous and inefficient, and won't adapt as the computer industry evolves.

Boost.Chrono aims to implement the new time facilities in C++0x, as proposed in **N2661 - A Foundation to Sleep On**. That document provides background and motivation for key design decisions and is the source of a good deal of information in this documentation.

Wall clock versus system and user time

To make the timing facilities of Boost.Chrono more generally useful, the library provides a number of clocks that are thin wrappers around the operating system's process time API, thereby allowing the extraction of wall clock time, user CPU time, and system CPU time of the process. Wall clock time is the sum of CPU time and system CPU time. (On POSIX-like systems, this relies on `times()`. On Windows, it relies on `GetProcessTimes()`.)

Description

The **Boost.Chrono** library provides:

Standard

- A means to represent time durations: managed by the generic `duration` class . Examples of time durations include days, `minutes`, `seconds` and `nanoseconds`, which can be represented with a fixed number of clock ticks per unit. All of these units of time duration are united with a generic interface by the `duration` facility.
- A type for representing points in time: `time_point`. A `time_point` represents an epoch plus or minus a `duration`. The library leaves epochs unspecified. A `time_point` is associated with a *clock*.
- Several clocks, some of which may not be available on a particular platform: `system_clock`, `steady_clock` and `high_resolution_clock`. A clock is a pairing of a `time_point` and `duration`, and a function which returns a `time_point` representing *now*.

Other clocks

To make the timing facilities more generally useful, **Boost.Chrono** provides a number of clocks that are thin wrappers around the operating system's time APIs, thereby allowing the extraction of wall clock time, user CPU time, system CPU time spent by the process,

- `process_real_cpu_clock`, captures wall clock CPU time spent by the current process.
- `process_user_cpu_clock`, captures user-CPU time spent by the current process.
- `process_system_cpu_clock`, captures system-CPU time spent by the current process.
- A tuple-like class `process_cpu_clock`, that captures real, user-CPU, and system-CPU process times together.
- A `thread_clock` thread steady clock giving the time spent by the current thread (when supported by a platform).

Lastly, **Boost.Chrono** includes typeof registration for `duration` and `time_point` to permit using emulated auto with C++03 compilers.

I/O

It provides I/O for `duration` and `time_point`. It builds on `<boost/ratio/ratio_io.hpp>` to provide readable and flexible formatting and parsing for types in `<boost/chrono.hpp>`. The `duration` unit names can be customized through a new facet: `duration_punct`.

Caveat Emptor

The underlying clocks provided by operating systems are subject to many seemingly arbitrary policies and implementation irregularities. That's a polite way of saying they tend to be flakey, and each operating system or even each clock has its own cruel and unusual forms of flakiness. Don't bet the farm on their accuracy, unless you have become deeply familiar with exactly what the specific operating system is guaranteeing, which is often very little.

User's Guide

Getting Started

Installing Chrono

Getting Boost.Chrono

Boost.Chrono is in the latest Boost release in the folder `/boost/chrono`. Documentation, tests and examples folder are at `boost/libs/chrono/`.

You can also access the latest (unstable?) state from the Boost trunk directories boost/chrono and libs/chrono. Just go to here and follow the instructions there for anonymous SVN access.

Where to install Boost.Chrono?

The simple way is to decompress (or checkout from SVN) the files in your BOOST_ROOT directory.

Building Boost.Chrono

Boost.Chrono can be configured as a header-only library. When `BOOST_CHRONO_HEADER_ONLY` is defined the Boost.Chrono is a header-only library. Otherwise is not a header only library and you need to compile it and build the library before use, for example using:

```
bjam libs/chrono/build
```

Requirements

Caution

Boost.Chrono depends on some new traits in **Boost.TypeTraits** which have been added on Boost 1.45.

In particular, **Boost.Chrono** depends on:

Boost.Config	for configuration purposes, ...
Boost.Exception	for throw_exception, ...
Boost.Integer	for cstdint conformance, ...
Boost.MPL	for MPL Assert and bool, logical ...
Boost.Operators	for operators, ...
Boost.Ratio	for ratio, milli, micro, ...
Boost.System	for error_code, ...
Boost.TypeTraits	for is_base, is_convertible, common_type, ...
Boost.Utility/EnableIf	for enable_if, ...

Building an Executable that Uses Boost.Chrono

In addition to link with the **Boost.Chrono** library you need also to link with the **Boost.System** library. Once **Boost.System** will be configurable to be a header only using `BOOST_SYSTEM_INLINED` you will no need to link with it.

Exception safety

All functions in the library are exception-neutral and provide strong guarantee of exception safety as long as the underlying parameters provide it.

Thread safety

All functions in the library are thread-unsafe except when noted explicitly.

As Boost.Chrono doesn't use mutable global variables the thread-safety analysis is limited to the access to each instance variable. It is not thread safe to use a function that modifies the access to a user variable if another can be reading or writing it.

Tested compilers

The implementation will eventually work with most C++03 conforming compilers. Current version has been tested on:

Windows with

- MSVC 10.0

Cygwin 1.5 with

- GCC 3.4.4

Cygwin 1.7 with

- GCC 4.3.4

MinGW with

- GCC 4.4.0
- GCC 4.5.0
- GCC 4.5.0 -std=c++0x
- GCC 4.6.0
- GCC 4.6.0 -std=c++0x

Ubuntu 10.10

- GCC 4.4.5
- GCC 4.4.5 -std=c++0x
- GCC 4.5.1
- GCC 4.5.1 -std=c++0x
- clang 2.8

Initial versions were tested on:

MacOS with GCC 4.2.4 (Some test are needed for the specific Mac files).

Ubuntu Linux with GCC 4.2.4

Note

Please let us know how this works on other platforms/compilers.

Note

Please send any questions, comments and bug reports to boost <at> lists <dot> boost <dot> org.

Hello World!

If all you want to do is to time a program's execution, here is a complete program (stopclock_example.cpp):

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

int main()
{
    boost::chrono::system_clock::time_point start = boost::chrono::system_clock::now();

    for ( long i = 0; i < 10000000; ++i )
    std::sqrt( 123.456L ); // burn some time

    boost::chrono::duration<double> sec = boost::chrono::system_clock::now() - start;
    std::cout << "took " << sec.count() << " seconds\n";
    return 0;
}
```

Output was:

```
took 0.832 seconds
```

Tutorial

Duration

The `duration` is the heart of this library. The interface that the user will see in everyday use is nearly identical to that of **Boost.DateTime** time `duration`s authored by Jeff Garland, both in syntax and in behavior. This has been a very popular boost library for 7 years. There is an enormous positive history with this interface.

The library consists of six units of time `duration`:

- `hours`
- `minutes`
- `seconds`
- `milliseconds`
- `microseconds`
- `nanoseconds`

These units were chosen as a subset of the boost library because they are the most common units used when sleeping, waiting on a condition variable, or waiting to obtain the lock on a mutex. Each of these units is nothing but a thin wrapper around a signed integral count. That is, when you construct `minutes(3)`, all that happens is a `3` is stored inside `minutes`. When you construct `microseconds(3)`, all that happens is a `3` is stored inside `microseconds`.

The only context in which these different types differ is when being converted to one another. At this time, unit-specific compile-time conversion constants are used to convert the source unit to the target unit. Only conversions from coarser units to finer units are allowed (in Boost). This restriction ensures that all conversions are always exact. That is, `microseconds` can always represent any value `minutes` has.

In **Boost.DateTime**, these units are united via inheritance. **Boost.Chrono** instead unites these units through the class template `duration`. That is, in **Boost.Chrono** all six of the above units are nothing but typedefs to different instantiations of `duration`. This change from Boost.DateTime has a far reaching positive impact, while not changing the syntax of the everyday use at all.

The most immediate positive impact is that the library can immediately generate any unit, with any precision it needs. This is sometimes necessary when doing comparisons or arithmetic between `duration`s of differing precision, assuming one wants the comparison and arithmetic to be exact.

A secondary benefit is that by publishing the class template `duration` interface, user code can very easily create `duration`s with any precision they desire. The `ratio` utility is used to specify the precision, so as long as the precision can be expressed by a rational constant with respect to seconds, this framework can exactly represent it (one third of a second is no problem, and neither is one third of a `femto` second). All of this utility and flexibility comes at no cost just by making use of the no-run-time-overhead `ratio` facility.

In Boost.DateTime, `hours` does not have the same representation as `nanoseconds`. The former is usually represented with a `long` whereas a `long long` is required for the latter. The reason for this is simply range. You don't need many hours to cover an extremely large range of time. But this isn't true of nanoseconds. Being able to reduce the sizeof overhead for some units when possible, can be a significant performance advantage.

Boost.Chrono continues, and generalizes that philosophy. Not only can one specify the precision of a `duration`, one can also specify its representation. This can be any integral type, or even a floating-point type. Or it can be a user-defined type which emulates an arithmetic type. The six predefined units all use signed integral types as their representation. And they all have a minimum range of ± 292 years. `nanoseconds` needs 64 bits to cover that range. `hours` needs only 23 bits to cover that range.

So What Exactly is a `duration` and How Do I Use One?

A `duration` has a representation and a tick period (precision).

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

The representation is simply any arithmetic type, or an emulation of such a type. The representation stores a count of ticks. This count is the only data member stored in a `duration`. If the representation is floating-point, it can store fractions of a tick to the precision of the representation. The tick period is represented by a `ratio` and is encoded into the `duration`'s type, instead of stored. The tick period only has an impact on the behavior of the `duration` when a conversion between different `duration`s is attempted. The tick period is completely ignored when simply doing arithmetic among like `duration`s.

Example:

```
typedef boost::chrono::duration<long, boost::ratio<60> > minutes;
minutes m1(3);                  // m1 stores 3
minutes m2(2);                  // m2 stores 2
minutes m3 = m1 + m2;           // m3 stores 5

typedef boost::chrono::duration<long long, boost::micro> microseconds;
microseconds us1(3);            // us1 stores 3
microseconds us2(2);            // us2 stores 2
microseconds us3 = us1 + us2;   // us3 stores 5

microseconds us4 = m3 + us3;    // us4 stores 300000005
```

In the final line of code above, there is an implicit conversion from minutes to microseconds, resulting in a relatively large number of microseconds.

If you need to access the tick count within a `duration`, there is a member `count()` which simply returns the stored tick count.

```
long long tc = us4.count();     // tc is 300000005
```

These `duration`'s have very simple, very predictable, and very observable behavior. After all, this is really nothing but the time-tested interface of Jeff's boost time `duration` library (unified with templates instead of inheritance).

What Happens if I Assign `m3 + us3` to `minutes` Instead of `microseconds`?

```
minutes m4 = m3 + us3;
```

It won't compile! The rationale is that implicit truncation error should not be allowed to happen. If this were to compile, then `m4` would hold `5`, the same value as `m3`. The value associated with `us3` has been effectively ignored. This is similar to the problem of assigning a double to an `int`: the fractional part gets silently discarded.

But What if the Truncation Behavior is What I Want to Do?

There is a `duration_cast` facility to explicitly ask for this behavior:

```
minutes m4 = boost::chrono::duration_cast<minutes>(m3 + us3);  // m4.count() == 5
```

In general, one can perform `duration` arithmetic at will. If `duration_cast` isn't used, and it compiles, the arithmetic is exact. If one wants to override this exact arithmetic behavior, `duration_cast` can be used to explicitly specify that desire. The `duration_cast` has the same efficiency as the implicit conversion, and will even be exact as often as it can.

You can use `duration_cast<>` to convert the `duration` into whatever units you desire. This facility will round down (truncate) if an exact conversion is not possible. For example:

```
boost::chrono::nanoseconds start;
boost::chrono::nanoseconds end;
typedef boost::chrono::milliseconds ms;
ms d = boost::chrono::duration_cast<ms>(end - start);

// d now holds the number of milliseconds from start to end.

std::cout << ms.count() << "ms\n";
```

We can convert to `nanoseconds`, or some integral-based duration which `nanoseconds` will always exactly convert to, then `duration_cast<>` is unnecessary:

```
typedef boost::chrono::nanoseconds ns;
ns d = end - start;
std::cout << ns.count() << "ns\n";
```

If you need seconds with a floating-point representation you can also eliminate the `duration_cast<>`:

```
typedef boost::chrono::duration<double> sec;  // seconds, stored with a double
sec d = end - start;
std::cout << sec.count() << "s\n";
```

If you're not sure if you need `duration_cast<>` or not, feel free to try it without. If the conversion is exact, or if the destination has a floating-point representation, it will compile: else it will not compile.

If you need to use `duration_cast<>`, but want to round up, instead of down when the conversion is inexact, here is a handy little helper function to do so. Writing it is actually a good starter project for understanding **Boost.Chrono**:

```
template <class ToDuration, class Rep, class Period>
ToDuration
round_up(boost::chrono::duration<Rep, Period> d)
{
    // first round down
    ToDuration result = boost::chrono::duration_cast<ToDuration>(d);
    if (result < d)  // comparisons are *always* exact
       ++result;     // increment by one tick period
    return result;
}

typedef boost::chrono::milliseconds ms;
ms d = round_up<ms>(end - start);
// d now holds the number of milliseconds from start to end, rounded up.
std::cout << ms.count() << "ms\n";
```

Trafficking in floating-point Durations

I don't want to deal with writing `duration_cast` all over the place. I'm content with the precision of my floating-point representation.

Not a problem. When the destination of a conversion has floating-point representation, all conversions are allowed to happen implicitly.

```
typedef boost::chrono::duration<double, ratio<60> > dminutes;
dminutes dm4 = m3 + us3;  // dm4.count() == 5.000000083333333
```

How Expensive is All of this?

If you were writing these conversions by hand, you could not make it more efficient. The use of `ratio` ensures that all conversion constants are simplified as much as possible at compile-time. This usually results in the numerator or denominator of the conversion factor simplifying to `1`, and being subsequently ignored in converting the run-time values of the tick counts.

How Complicated is it to Build a Function Taking a `duration` Parameter?

There are several options open to the user:

- If the author of the function wants to accept any `duration`, and is willing to work in floating-point `duration`s, he can simply use any floating-point `duration` as the parameter:

```
void f(boost::chrono::duration<double> d)  // accept floating-point seconds
{
    // d.count() == 3.e-6 when passed boost::chrono::microseconds(3)
}

f(boost::chrono::microseconds(3));
```

- If the author of the function wants to traffic only in integral `duration`s, and is content with handling nothing finer than say nanoseconds (just as an example), he can simply specify nanoseconds as the parameter:

```
void f(boost::chrono::nanoseconds d)
{
    // d.count() == 3000 when passed boost::chrono::microseconds(3)
}

f(boost::chrono::microseconds(3));
```

In this design, if the client wants to pass in a floating-point `duration`, or a `duration` of finer precision than nanoseconds, then the client is responsible for choosing his own rounding mode in the conversion to nanoseconds.

```
boost::chrono::duration<double> s(1./3);  // 1/3 of a second
f(boost::chrono::duration_cast<boost::chrono::nanoseconds>(s));  // round towards zero in conver↵
sion to nanoseconds
```

In the example above, the client of f has chosen "round towards zero" as the desired rounding mode to nanoseconds. If the client has a `duration` that won't exactly convert to nanoseconds, and fails to choose how the conversion will take place, the compiler will refuse the call:

```
f(s);  // does not compile
```

- If the author of the function wants to accept any `duration`, but wants to work with integral representations and wants to control the rounding mode internally, then he can template the function:

```
template <class Rep, class Period>
void f(boost::chrono::duration<Rep, Period> d)
{
    // convert d to nanoseconds, rounding up if it is not an exact conversion
    boost::chrono::nanoseconds ns = boost::chrono::duration_cast<boost::chrono::nanoseconds>(d);
    if (ns < d)
        ++ns;
    // ns.count() == 333333334 when passed 1/3 of a floating-point second
}

  f(boost::chrono::duration<double>(1./3));
```

- If the author in the example does not want to accept floating-point based `duration`s, he can enforce that behavior like so:

```
template <class Period>
void f(boost::chrono::duration<long long, Period> d)
{
    // convert d to nanoseconds, rounding up if it is not an exact conversion
    boost::chrono::nanoseconds ns = boost::chrono::duration_cast<nanoseconds>(d);
    if (ns < d)
        ++ns;
    // ns.count() == 333333334 when passed 333333333333 picoseconds
}
// About 1/3 of a second worth of picoseconds
f(boost::chrono::duration<long long, boost::pico>(333333333333));
```

Clients with floating-point `duration`s who want to use f will now have to convert to an integral `duration` themselves before passing the result to f.

In summary, the author of f has quite a bit of flexibility and control in the interface he wants to provide his clients with, and easy options for manipulating that `duration` internal to his function.

Is it possible for the user to pass a `duration` to a function with the units being ambiguous?

No. No matter which option the author of `f` chooses above, the following client code will not compile:

```
f(3);  // Will not compile, 3 is not implicitly convertible to any __duration
```

Can Durations Overflow?

This depend on the representation. The default typedefs uses a representation that don't handle overflows. The user can define his own representation that manage overflow as required by its application.

Clocks

While `duration`s only have precision and representation to concern themselves, clocks and `time_point`s are intimately related and refer to one another. Because clocks are simpler to explain, we will do so first without fully explaining `time_point`s. Once clocks are introduced, it will be easier to then fill in what a `time_point` is.

A clock is a concept which bundles 3 things:

1. A concrete `duration` type.
2. A concrete `time_point` type.
3. A function called now() which returns the concrete `time_point`.

The standard defines tree system-wide clocks that are associated to the computer time.

- `system_clock` represents system-wide realtime clock that can be synchronized with an external clock.
- `steady_clock` can not be changed explicitly and the time since the initial epoch increase in a steady way.
- `high_resolution_clock` intend to use the system-wide clock provided by the platform with the higest resolution.

Boost.Chrono provides them when supported by the underlying platform. A given platform may not be able to supply all three of these clocks.

The library adds some clocks that are specific to a process or a thread, that is there is a clock per process or per thread.

The user is also able to easily create more clocks.

Given a clock named Clock, it will have:

```
class Clock {
public:
    typedef an arithmetic-like type        rep;
    typedef an instantiation of ratio      period;
    typedef boost::chrono::duration<rep, period> duration;
    typedef boost::chrono::time_point<Clock>     time_point;
    static constexpr bool is_steady =      true or false;

    static time_point now();
};
```

One can get the current time from Clock with:

```
Clock::time_point t1 = Clock::now();
```

And one can get the time `duration` between two `time_point`s associated with Clock with:

```
Clock::duration d = Clock::now() - t1;
```

And one can specify a past or future `time_point` with:

```
Clock::time_point t2 = Clock::now() + d;
```

Note how even if a particular clock becomes obsolete, the next clock in line will have the same API. There is no new learning curve to come up. The only source code changes will be simply changing the type of the clock. The same `duration` and `time_point` framework continues to work as new clocks are introduced. And multiple clocks are safely and easily handled within the same program.

Time Point

A `time_point` represents a point in time, as opposed to a `duration` of time. Another way of saying the same thing, is that a `time_point` represents an epoch plus or minus a `duration`. Examples of `time_point`s include:

- 3 minutes after the computer booted.
- 03:14:07 UTC on Tuesday, January 19, 2038
- 20 milliseconds after I started that timer.

In each of the examples above, a different epoch is implied. Sometimes an epoch has meaning for several millennia. Other times the meaning of an epoch is lost after a while (such as the start of a timer, or when the computer booted). However, if two `time_point`s are known to share the same epoch, they can be subtracted, yielding a valid `duration`, even if the definition of the epoch no longer has meaning.

In **Boost.Chrono**, an epoch is a purely abstract and unspecified concept. There is no type representing an epoch. It is simply an idea that relates (or doesn't) `time_point`s to a clock, and in the case that they share a clock, `time_point`s to one another. `time_point`s associated with different clocks are generally not interoperable unless the relationship between the epochs associated with each clock is known.

So What Exactly is a `time_point` and How Do I Use One?

A `time_point` has a clock and a `duration`.

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

The `time_point`'s clock is not stored. It is simply embedded into the `time_point`'s type and serves two purposes:

1. Because `time_point`s originating from different clocks have different types, the compiler can be instructed to fail if incompatible `time_point`s are used in inappropriate ways.

2. Given a `time_point`, one often needs to compare that `time_point` to "now". This is very simple as long as the `time_point` knows what clock it is defined with respect to.

A `time_point`'s `duration` is stored as the only data member of the `time_point`. Thus `time_point`s and their corresponding `duration` have exactly the same layout. But they have very different meanings. For example, it is one thing to say I want to sleep for 3 minutes. It is a completely different thing to say I want to sleep until 3 minutes past the time I started that timer (unless you just happened to start that timer now). Both meanings (and options for sleeping) have great practical value in common use cases for sleeping, waiting on a condition variable, and waiting for a mutex's lock. These same concepts and tools are found (for example) in Ada.

A timer example:

```
void f()
{
    boost::chrono::steady_clock::time_point start = boost::chrono::steady_clock::now();
    g();
    h();
    duration<double> sec = boost::chrono::steady_clock::now() - start;
    cout << "f() took " << sec.count() << " seconds\n";
}
```

Note that if one is using the `duration` between two clock `time_point`s in a way where the precision of the `duration` matters, it is good practice to convert the clock's `duration` to a known `duration`. This insulates the code from future changes which may be made to the clock's precision in the future. For example `steady_clock` could easily be based on the clock speed of the cpu. When you upgrade to a faster machine, you do not want your code that assumed a certain tick period of this clock to start experiencing run-time failures because your timing code has silently changed meaning.

A delay loop example:

```
// delay for at least 500 nanoseconds:
auto go = boost::chrono::steady_clock::now() + boost::chrono::nanoseconds(500);
while (boost::chrono::steady_clock::now() < go)
    ;
```

The above code will delay as close as possible to half a microsecond, no matter what the precision of `steady_clock` is. The more precise `steady_clock` becomes, the more accurate will be the delay to 500 nanoseconds.

Specific Clocks

system_clock

`system_clock` is useful when you need to correlate the time with a known epoch so you can convert it to a calendar time. Note the specific functions in the `system_clock` class.

steady_clock

`steady_clock` is useful when you need to wait for a specific amount of time. `steady_clock` time can not be reset. As other steady clocks, it is usually based on the processor tick.

Here is a polling solution, but it will probably be too inefficient:

```
boost::chrono::steady_clock::time_point start= chrono::steady_clock::now();
boost::chrono::steady_clock::duration delay= chrono::seconds(5);
while (boost::chrono::steady_clock::now() - start <= delay) {}
```

high_resolution_clock

When available, `high_resolution_clock` is usually more expensive than the other system-wide clocks, so they are used only when the provided resolution is required to the application.

process_cpu_clock

Process and thread clocks are used usually to measure the time spent by code blocks, as a basic time-spent profiling of different blocks of code (Boost.Stopwatch is a clear example of this use).

thread_clock

You can use `thread_clock` whenever you want to measure the time spent by the current thread. For example:

```
boost::chrono::thread_clock::time_point start=boost::chrono::thread_clock::now();
// ... do something ...

typedef boost::chrono::milliseconds ms;
ms d = boost::chrono::thread_clock::now() - start;
// d now holds the number of milliseconds from start to end.
std::cout << ms.count() << "ms\n";
```

If you need seconds with a floating-point representation you can do:

```
typedef boost::chrono::duration<double> sec;  // seconds, stored with a double.
sec d = end - start;
std::cout << sec.count() << "s\n";
```

If you would like to programmatically inspect `thread_clock::duration`, you can get the representation type with `thread_clock::rep`, and the tick period with `thread_clock::period` (which should be a type `ratio` which has nested values `ratio::num` and `ratio::den`). The tick period of `thread_clock` is `thread_clock::period::num / thread_clock::period::den` seconds: `1/1000000000` in this case (`1` billionth of a second), stored in a `long long`.

I/O

Any `duration` can be streamed out to a `basic_ostream`. The run-time value of the `duration` is formatted according to the rules and current format settings for `duration::rep`. This is followed by a single space and then the compile-time unit name of the `duration`. This unit name is built on the string returned from `ratio_string<>` and the data used to construct the `duration_punct` which was inserted into the stream's locale. If a `duration_punct` has not been inserted into the stream's locale, a default constructed `duration_punct` will be added to the stream's locale.

`duration` unit names come in two varieties: long and short. The default constructed `duration_punct` provides names in the long format. These names are English descriptions. Other languages are supported by constructing a `duration_punct` with the proper spellings for "hours", "minutes" and "seconds", and their abbreviations (for the short format). The short or long format can be easily chosen by streaming a `duration_short()` or `duration_long()` manipulator respectively.

A `time_point` is formatted by outputting its internal `duration` followed by a string that describes the `time_point::clock` epoch. This string will vary for each distinct clock, and for each implementation of the supplied clocks.

Example:

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

int main()
{
    using namespace std;
    using namespace boost;

    cout << "milliseconds(3) + microseconds(10) = "
         <<  boost::chrono::milliseconds(3) + boost::chrono::microseconds(10) << '\n';

    cout << "hours(3) + minutes(10) = "
         <<  boost::chrono::hours(3) + boost::chrono::minutes(10) << '\n';

    typedef boost::chrono::duration<long long, boost::ratio<1, 2500000000> > ClockTick;
    cout << "ClockTick(3) + boost::chrono::nanoseconds(10) = "
         <<  ClockTick(3) + boost::chrono::nanoseconds(10) << '\n';

    cout << "\nSet cout to use short names:\n";
    cout << boost::chrono::duration_short;

    cout << "milliseconds(3) + microseconds(10) = "
         <<  boost::chrono::milliseconds(3) + boost::chrono::microseconds(10) << '\n';

    cout << "hours(3) + minutes(10) = "
         <<  boost::chrono::hours(3) + boost::chrono::minutes(10) << '\n';

    cout << "ClockTick(3) + nanoseconds(10) = "
         <<  ClockTick(3) + boost::chrono::nanoseconds(10) << '\n';

    cout << "\nsystem_clock::now() = " << boost::chrono::system_clock::now() << '\n';
#ifdef BOOST_CHRONO_HAS_CLOCK_STEADY
    cout << "steady_clock::now() = " << boost::chrono::steady_clock::now() << '\n';
#endif
    cout << "\nSet cout to use long names:\n"
            << boost::chrono::duration_long
            << "high_resolution_clock::now() = "
            << boost::chrono::high_resolution_clock::now() << '\n';
    return 0;
}
```

The output could be

```
milliseconds(3) + microseconds(10) = 3010 microseconds
hours(3) + minutes(10) = 190 minutes
ClockTick(3) + nanoseconds(10) = 56 [1/5000000000]seconds

Set cout to use short names:
milliseconds(3) + microseconds(10) = 3010 [mu]s
hours(3) + minutes(10) = 190 m
ClockTick(3) + nanoseconds(10) = 56 [1/5000000000]s

system_clock::now() = 129387415616250000 [1/10000000]s since Jan 1, 1970
monotonic_clock::now() = 37297387636417 ns since boot

Set cout to use long names:
high_resolution_clock::now() = 37297387655134 nanoseconds since boot
```

Parsing a `duration` follows rules analogous to the `duration` converting constructor. A value and a unit (short or long) are read from the `basic_istream`. If the `duration` has an integral representation, then the value parsed must be exactly representable in the target `duration` (after conversion to the target `duration` units), else `failbit` is set. `duration`s based on floating-point representations can be parsed using any units that do not cause overflow.

For example a stream containing "5000 milliseconds" can be parsed into seconds, but if the stream contains "5001 milliseconds", parsing into `seconds` will cause `failbit` to be set.

Example:

```
#include <boost/chrono/chrono_io.hpp>
#include <sstream>
#include <cassert>

int main()
{
    using namespace std;

    istringstream in("5000 milliseconds 4000 ms 3001 ms");
    boost::chrono::seconds d(0);
    in >> d;
    assert(in.good());
    assert(d == seconds(5));
    in >> d;
    assert(in.good());
    assert(d == seconds(4));
    in >> d;
    assert(in.fail());
    assert(d == seconds(4));

    return 0;
}
```

Note that a `duration` failure may occur late in the parsing process. This means that the characters making up the failed parse in the stream are usually consumed despite the failure to successfully parse.

Parsing a `time_point` involves first parsing a `duration` and then parsing the epoch string. If the epoch string does not match that associated with `time_point::clock` then failbit will be set.

Example:

```
#include <boost/chrono/chrono_io.hpp>
#include <sstream>
#include <iostream>
#include <cassert>

int main()
{
    using namespace std;

    boost::chrono::high_resolution_clock::time_point t0 = boost::chrono::high_resolu↵
tion_clock::now();
    stringstream io;
    io << t0;
    boost::chrono::high_resolution_clock::time_point t1;
    io >> t1;
    assert(!io.fail());
    cout << io.str() << '\n';
    cout << t0 << '\n';
    cout << t1 << '\n';
    boost::chrono::high_resolution_clock::time_point t = boost::chrono::high_resolu↵
tion_clock::now();
    cout << t << '\n';

    cout << "That took " << t - t0 << '\n';
    cout << "That took " << t - t1 << '\n';

    return 0;
}
```

The output could be:

```
50908679121461 nanoseconds since boot
That took 649630 nanoseconds
```

Here's a simple example to find out how many hours the computer has been up (on this platform):

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

int main()
{
    using namespace std;
    using namespace boost;

    typedef boost::chrono::time_point<boost::chrono::steady_clock, boost::chrono::dura↵
tion<double, boost::ratio<3600> > > T;
    T tp = boost::chrono::steady_clock::now();
    std::cout << tp << '\n';
    return 0;
}
```

The output could be:

```
17.8666 hours since boot
```

Examples

Duration

How you Override the Duration's Default Constructor

Next we show how to override the `duration`'s default constructor to do anything you want (in this case set it to zero). All we need to do is to change the representation

```
namespace I_dont_like_the_default_duration_behavior {

template <class R>
class zero_default
{
public:
    typedef R rep;

private:
    rep rep_;
public:
    zero_default(rep i = 0) : rep_(i) {}
    operator rep() const {return rep_;}

    zero_default& operator+=(zero_default x) {rep_ += x.rep_; return *this;}
    zero_default& operator-=(zero_default x) {rep_ -= x.rep_; return *this;}
    zero_default& operator*=(zero_default x) {rep_ *= x.rep_; return *this;}
    zero_default& operator/=(zero_default x) {rep_ /= x.rep_; return *this;}

    zero_default  operator+ () const {return *this;}
    zero_default  operator- () const {return zero_default(-rep_);}
    zero_default& operator++()       {++rep_; return *this;}
    zero_default  operator++(int)    {return zero_default(rep_++);}
    zero_default& operator--()       {--rep_; return *this;}
    zero_default  operator--(int)    {return zero_default(rep_--);}

    friend zero_default operator+(zero_default x, zero_default y) {return x += y;}
    friend zero_default operator-(zero_default x, zero_default y) {return x -= y;}
    friend zero_default operator*(zero_default x, zero_default y) {return x *= y;}
    friend zero_default operator/(zero_default x, zero_default y) {return x /= y;}

    friend bool operator==(zero_default x, zero_default y) {return x.rep_ == y.rep_;}
    friend bool operator!=(zero_default x, zero_default y) {return !(x == y);}
    friend bool operator< (zero_default x, zero_default y) {return x.rep_ < y.rep_;}
    friend bool operator<=(zero_default x, zero_default y) {return !(y < x);}
    friend bool operator> (zero_default x, zero_default y) {return y < x;}
    friend bool operator>=(zero_default x, zero_default y) {return !(x < y);}
};

typedef boost::chrono::duration<zero_default<long long>, boost::nano        > nanoseconds;
typedef boost::chrono::duration<zero_default<long long>, boost::micro       > microseconds;
typedef boost::chrono::duration<zero_default<long long>, boost::milli       > milliseconds;
typedef boost::chrono::duration<zero_default<long long>                      > seconds;
typedef boost::chrono::duration<zero_default<long long>, boost::ratio<60>   > minutes;
typedef boost::chrono::duration<zero_default<long long>, boost::ratio<3600> > hours;
}
```

Usage

```
using namespace I_dont_like_the_default_duration_behavior;

milliseconds ms;
std::cout << ms.count() << '\n';
```

See the source file example/i_dont_like_the_default_duration_behavior.cpp

Saturating

A "saturating" signed integral type is developed. This type has +/- infinity and a NaN (like IEEE floating-point) but otherwise obeys signed integral arithmetic. This class is subsequently used as the template parameter Rep in boost::chrono::`duration` to demonstrate a duration class that does not silently ignore overflow.

See the source file example/saturating.cpp

xtime Conversions

Example round_up utility: converts d to To, rounding up for inexact conversions Being able to **easily** write this function is a major feature!

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

#include <iostream>

template <class To, class Rep, class Period>
To
round_up(boost::chrono::duration<Rep, Period> d)
{
    To result = boost::chrono::duration_cast<To>(d);
    if (result < d)
        ++result;
    return result;
}
```

To demonstrate interaction with an xtime-like facility:

```
struct xtime
{
    long sec;
    unsigned long usec;
};

template <class Rep, class Period>
xtime
to_xtime_truncate(boost::chrono::duration<Rep, Period> d)
{
    xtime xt;
    xt.sec = static_cast<long>(boost::chrono::duration_cast<seconds>(d).count());
    xt.usec = static_cast<long>(boost::chrono::duration_cast<micro↵
seconds>(d - seconds(xt.sec)).count());
    return xt;
}

template <class Rep, class Period>
xtime
to_xtime_round_up(boost::chrono::duration<Rep, Period> d)
{
    xtime xt;
    xt.sec = static_cast<long>(boost::chrono::duration_cast<seconds>(d).count());
    xt.usec = static_cast<unsigned long>(round_up<boost::chrono::micro↵
seconds>(d - boost::chrono::seconds(xt.sec)).count());
    return xt;
}

microseconds
from_xtime(xtime xt)
{
    return boost::chrono::seconds(xt.sec) + boost::chrono::microseconds(xt.usec);
}

void print(xtime xt)
{
    std::cout << '{' << xt.sec << ',' << xt.usec << "}\n";
}
```

Usage

```
xtime xt = to_xtime_truncate(seconds(3) + boost::chrono::milliseconds(251));
print(xt);
boost::chrono::milliseconds ms = boost::chrono::duration_cast<boost::chrono::milli↵
seconds>(from_xtime(xt));
std::cout << ms.count() << " milliseconds\n";
xt = to_xtime_round_up(ms);
print(xt);
xt = to_xtime_truncate(boost::chrono::seconds(3) + nanoseconds(999));
print(xt);
xt = to_xtime_round_up(boost::chrono::seconds(3) + nanoseconds(999));
print(xt);
```

See the source file xtime.cpp

Clocks

Cycle count

Users can easily create their own clocks, with both points in time and time durations which have a representation and precision of their own choosing. For example if there is a hardware counter which simply increments a count with each cycle of the cpu, one can

very easily build clocks, time points and durations on top of that, using only a few tens of lines of code. Such systems can be used to call the time-sensitive threading API's such as sleep, wait on a condition variable, or wait for a mutex lock. The API proposed herein is not sensitive as to whether this is a 300MHz clock (with a 3 1/3 nanosecond tick period) or a 3GHz clock (with a tick period of 1/3 of a nanosecond). And the resulting code will be just as efficient as if the user wrote a special purpose clock cycle counter.

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

template <long long speed>
struct cycle_count
{
    typedef typename boost::__ratio_multiply__<boost::ratio<speed>, boost::mega>::type
        frequency;  // Mhz
    typedef typename boost::__ratio_divide__<boost::ratio<1>, frequency>::type period;
    typedef long long rep;
    typedef boost::chrono::duration<rep, period> duration;
    typedef boost::chrono::time_point<cycle_count> time_point;

    static time_point now()
    {
        static long long tick = 0;
        // return exact cycle count
        return time_point(duration(++tick));  // fake access to clock cycle count
    }
};

template <long long speed>
struct approx_cycle_count
{
    static const long long frequency = speed * 1000000;  // MHz
    typedef nanoseconds duration;
    typedef duration::rep rep;
    typedef duration::period period;
    static const long long nanosec_per_sec = period::den;
    typedef boost::chrono::time_point<approx_cycle_count> time_point;

    static time_point now()
    {
        static long long tick = 0;
        // return cycle count as an approximate number of nanoseconds
        // compute as if nanoseconds is only duration in the std::lib
        return time_point(duration(++tick * nanosec_per_sec / frequency));
    }
};
```

See the source file cycle_count.cpp

xtime_clock

This example demonstrates the use of a timeval-like struct to be used as the representation type for both `duration` and `time_point`.

```
class xtime {
private:
    long tv_sec;
    long tv_usec;

    void fixup() {
        if (tv_usec < 0) {
            tv_usec += 1000000;
            --tv_sec;
        }
    }

public:
    explicit xtime(long sec, long usec) {
        tv_sec = sec;
        tv_usec = usec;
        if (tv_usec < 0 || tv_usec >= 1000000) {
            tv_sec += tv_usec / 1000000;
            tv_usec %= 1000000;
            fixup();
        }
    }

    explicit xtime(long long usec) {
        tv_usec = static_cast<long>(usec % 1000000);
        tv_sec  = static_cast<long>(usec / 1000000);
        fixup();
    }

    // explicit
    operator long long() const {return static_cast<long long>(tv_sec) * 1000000 + tv_usec;}

    xtime& operator += (xtime rhs) {
        tv_sec += rhs.tv_sec;
        tv_usec += rhs.tv_usec;
        if (tv_usec >= 1000000) {
            tv_usec -= 1000000;
            ++tv_sec;
        }
        return *this;
    }

    xtime& operator -= (xtime rhs) {
        tv_sec -= rhs.tv_sec;
        tv_usec -= rhs.tv_usec;
        fixup();
        return *this;
    }

    xtime& operator %= (xtime rhs) {
        long long t = tv_sec * 1000000 + tv_usec;
        long long r = rhs.tv_sec * 1000000 + rhs.tv_usec;
        t %= r;
        tv_sec = static_cast<long>(t / 1000000);
        tv_usec = static_cast<long>(t % 1000000);
        fixup();
        return *this;
    }

    friend xtime operator+(xtime x, xtime y) {return x += y;}
    friend xtime operator-(xtime x, xtime y) {return x -= y;}
    friend xtime operator%(xtime x, xtime y) {return x %= y;}
```

```
    friend bool operator==(xtime x, xtime y)
        { return (x.tv_sec == y.tv_sec && x.tv_usec == y.tv_usec); }

    friend bool operator<(xtime x, xtime y) {
        if (x.tv_sec == y.tv_sec)
            return (x.tv_usec < y.tv_usec);
        return (x.tv_sec < y.tv_sec);
    }

    friend bool operator!=(xtime x, xtime y) { return !(x == y); }
    friend bool operator> (xtime x, xtime y) { return y < x; }
    friend bool operator<=(xtime x, xtime y) { return !(y < x); }
    friend bool operator>=(xtime x, xtime y) { return !(x < y); }

    friend std::ostream& operator<<(std::ostream& os, xtime x)
        {return os << '{' << x.tv_sec << ',' << x.tv_usec << '}';}
};
```

Clock based on timeval-like struct.

```
class xtime_clock
{
public:
    typedef xtime                                   rep;
    typedef boost::micro                            period;
    typedef boost::chrono::duration<rep, period>    duration;
    typedef boost::chrono::time_point<xtime_clock> time_point;

    static time_point now()
    {
    #if defined(BOOST_CHRONO_WINDOWS_API)
        time_point t(duration(xtime(0)));
        gettimeofday((timeval*)&t, 0);
        return t;

    #elif defined(BOOST_CHRONO_MAC_API)

        time_point t(duration(xtime(0)));
        gettimeofday((timeval*)&t, 0);
        return t;

    #elif defined(BOOST_CHRONO_POSIX_API)
        //time_point t(0,0);

        timespec ts;
        ::clock_gettime( CLOCK_REALTIME, &ts );

        xtime xt( ts.tv_sec, ts.tv_nsec/1000);
        return time_point(duration(xt));

    #endif  // POSIX
    }
};
```

Usage of xtime_clock

```
std::cout << "sizeof xtime_clock::time_point = " << sizeof(xtime_clock::time_point) << '\n';
std::cout << "sizeof xtime_clock::duration = " << sizeof(xtime_clock::duration) << '\n';
std::cout << "sizeof xtime_clock::rep = " << sizeof(xtime_clock::rep) << '\n';
xtime_clock::duration delay(boost::chrono::milliseconds(5));
xtime_clock::time_point start = xtime_clock::now();
while (xtime_clock::now() - start <= delay) {}
xtime_clock::time_point stop = xtime_clock::now();
xtime_clock::duration elapsed = stop - start;
std::cout << "paused " << boost::chrono::::nanoseconds(elapsed).count() << " nanoseconds\n";
```

See the source file example/timeval_demo.cpp

Time Point

min Utility

The user can define a function returning the earliest `time_point` as follows:

```
template <class Clock, class Duration1, class Duration2>
typename boost::common_type<time_point<Clock, Duration1>,
                    time_point<Clock, Duration2> >::type
min(time_point<Clock, Duration1> t1, time_point<Clock, Duration2> t2)
{
    return t2 < t1 ? t2 : t1;
}
```

Being able to **easily** write this function is a major feature!

```
BOOST_AUTO(t1, system_clock::now() + seconds(3));
BOOST_AUTO(t2, system_clock::now() + nanoseconds(3));
BOOST_AUTO(t3, min(t1, t2));
```

See the source file example/min_time_point.cpp

A Tiny Program that Times How Long Until a Key is Struck

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

using namespace boost::chrono;

template< class Clock >
class timer
{
  typename Clock::time_point start;
public:
  timer() : start( Clock::now() ) {}
  typename Clock::duration elapsed() const
  {
    return Clock::now() - start;
  }
  double seconds() const
  {
    return elapsed().count() * ((double)Clock::period::num/Clock::period::den);
  }
};

int main()
{
  timer<system_clock> t1;
  timer<steady_clock> t2;
  timer<high_resolution_clock> t3;

  std::cout << "Type the Enter key: ";
  std::cin.get();

  std::cout << std::fixed << std::setprecision(9);
  std::cout << "system_clock-----------: "
            << t1.seconds() << " seconds\n";
  std::cout << "steady_clock--------: "
            << t2.seconds() << " seconds\n";
  std::cout << "high_resolution_clock--: "
            << t3.seconds() << " seconds\n";

  system_clock::time_point d4 = system_clock::now();
  system_clock::time_point d5 = system_clock::now();

  std::cout << "\nsystem_clock latency-----------: " << (d5 - d4).count() << std::endl;

  steady_clock::time_point d6 = steady_clock::now();
  steady_clock::time_point d7 = steady_clock::now();

  std::cout << "steady_clock latency--------: " << (d7 - d6).count() << std::endl;

  high_resolution_clock::time_point d8 = high_resolution_clock::now();
  high_resolution_clock::time_point d9 = high_resolution_clock::now();

  std::cout << "high_resolution_clock latency--: " << (d9 - d8).count() << std::endl;

  std::time_t now = system_clock::to_time_t(system_clock::now());

  std::cout << "\nsystem_clock::now() reports UTC is "
    << std::asctime(std::gmtime(&now)) << "\n";

  return 0;
}
```

The output of this program run looks like this:

See the source file example/await_keystroke.cpp

24 Hours Display

In the example above we take advantage of the fact that `time_point`s convert as long as they have the same clock, and as long as their internal `duration`s convert. We also take advantage of the fact that a `duration` with a floating-point representation will convert from anything. Finally the I/O system discovers the more readable "hours" unit for our `duration<double, ratio<3600>>`.

There are many other ways to format `duration`s and `time_point`s. For example see ISO 8601. Instead of coding every possibility into `operator<<`, which would lead to significant code bloat for even the most trivial uses, this document seeks to inform the reader how to write custom I/O when desired.

As an example, the function below streams arbitrary `duration`s to arbitrary `basic_ostreams` using the format:

```
[-]d/hh:mm:ss.cc
```

Where:

- `d` is the number of `days`
- `h` is the number of `hours`
- `m` is the number of `minutes`
- `ss.cc` is the number of `seconds` rounded to the nearest hundreth of a second

1. include <boost/chrono/chrono_io.hpp>
2. include <ostream>
3. include <iostream>

```
// format duration as [-]d/hh::mm::ss.cc
template <class CharT, class Traits, class Rep, class Period>
std::basic_ostream<CharT, Traits>&
display(std::basic_ostream<CharT, Traits>& os,
        boost::chrono::duration<Rep, Period> d)
{
    using namespace std;
    using namespace boost;

    typedef boost::chrono::duration<long long, boost::ratio<86400> > days;
    typedef boost::chrono::duration<long long, boost:centi> centiseconds;

    // if negative, print negative sign and negate
    if (d < boost::chrono::duration<Rep, Period>(0))
    {
        d = -d;
        os << '-';
    }
    // round d to nearest centiseconds, to even on tie
    centiseconds cs = boost::chrono::duration_cast<centiseconds>(d);
    if (d - cs > boost::chrono::milliseconds(5)
        || (d - cs == boost::chrono::milliseconds(5) && cs.count() & 1))
        ++cs;
    // separate seconds from centiseconds
    boost::chrono::seconds s = boost::chrono::duration_cast<boost::chrono::seconds>(cs);
    cs -= s;
    // separate minutes from seconds
    boost::chrono::minutes m = boost::chrono::duration_cast<boost::chrono::minutes>(s);
    s -= m;
    // separate hours from minutes
    boost::chrono::hours h = boost::chrono::duration_cast<boost::chrono::hours>(m);
    m -= h;
    // separate days from hours
    days dy = boost::chrono::duration_cast<days>(h);
    h -= dy;
    // print d/hh:mm:ss.cc
    os << dy.count() << '/';
    if (h < boost::chrono::hours(10))
        os << '0';
    os << h.count() << ':';
    if (m < boost::chrono::minutes(10))
        os << '0';
    os << m.count() << ':';
    if (s < boost::chrono::seconds(10))
        os << '0';
    os << s.count() << '.';
    if (cs < boost::chrono::centiseconds(10))
        os << '0';
    os << cs.count();
    return os;
}

int main()
{
    using namespace std;
    using namespace boost;

    display(cout, boost::chrono::steady_clock::now().time_since_epoch()
                  + boost::chrono::duration<long, boost::mega>(1)) << '\n';
    display(cout, -boost::chrono::milliseconds(6)) << '\n';
    display(cout, boost::chrono::duration<long, boost::mega>(1)) << '\n';
    display(cout, -boost::chrono::duration<long, boost::mega>(1)) << '\n';
}
```

The output could be:

```
12/06:03:22.95
-0/00:00:00.01
11/13:46:40.00
-11/13:46:40.00
```

Simulated Thread Interface Demonstration Program

The C++0x standard library's multi-threading library requires the ability to deal with the representation of time in a manner consistent with modern C++ practices. Next is a simulation of this interface.

The non-member sleep functions can be emulated as follows:

```
namespace boost { namespace this_thread {

template <class Rep, class Period>
void sleep_for(const chrono::duration<Rep, Period>& d) {
    chrono::microseconds t = chrono::duration_cast<chrono::microseconds>(d);
    if (t < d)
        ++t;
    if (t > chrono::microseconds(0))
        std::cout << "sleep_for " << t.count() << " microseconds\n";
}

template <class Clock, class Duration>
void sleep_until(const chrono::time_point<Clock, Duration>& t) {
    using namespace chrono;
    typedef time_point<Clock, Duration> Time;
    typedef system_clock::time_point SysTime;
    if (t > Clock::now()) {
        typedef typename common_type<typename Time::duration,
                                     typename SysTime::duration>::type D;
        /* auto */ D d = t - Clock::now();
        microseconds us = duration_cast<microseconds>(d);
        if (us < d)
            ++us;
        SysTime st = system_clock::now() + us;
        std::cout << "sleep_until    ";
        detail::print_time(st);
       std::cout << " which is " << (st - system_clock::now()).count() << " microseconds away\n";
    }
}

}}
```

Next is the `boost::thread::timed_mutex` modified fuctions

```
namespace boost {
struct timed_mutex {
    // ...

    template <class Rep, class Period>
    bool try_lock_for(const chrono::duration<Rep, Period>& d) {
        chrono::microseconds t = chrono::duration_cast<chrono::microseconds>(d);
        if (t <= chrono::microseconds(0))
            return try_lock();
        std::cout << "try_lock_for " << t.count() << " microseconds\n";
        return true;
    }

    template <class Clock, class Duration>
    bool try_lock_until(const chrono::time_point<Clock, Duration>& t)
    {
        using namespace chrono;
        typedef time_point<Clock, Duration> Time;
        typedef system_clock::time_point SysTime;
        if (t <= Clock::now())
            return try_lock();
        typedef typename common_type<typename Time::duration,
          typename Clock::duration>::type D;
        /* auto */ D d = t - Clock::now();
        microseconds us = duration_cast<microseconds>(d);
        SysTime st = system_clock::now() + us;
        std::cout << "try_lock_until ";
        detail::print_time(st);
        std::cout << " which is " << (st - system_clock::now()).count()
          << " microseconds away\n";
        return true;
    }
};
}
```

`boost::thread::condition_variable` time related function are modified as follows:

```
namespace boost {
struct condition_variable
{
    // ...

    template <class Rep, class Period>
    bool wait_for(mutex&, const chrono::duration<Rep, Period>& d) {
        chrono::microseconds t = chrono::duration_cast<chrono::microseconds>(d);
        std::cout << "wait_for " << t.count() << " microseconds\n";
        return true;
    }

    template <class Clock, class Duration>
    bool wait_until(mutex&, const chrono::time_point<Clock, Duration>& t) {
        using namespace boost::chrono;
        typedef time_point<Clock, Duration> Time;
        typedef system_clock::time_point SysTime;
        if (t <= Clock::now())
            return false;
        typedef typename common_type<typename Time::duration,
          typename Clock::duration>::type D;
        /* auto */ D d = t - Clock::now();
        microseconds us = duration_cast<microseconds>(d);
        SysTime st = system_clock::now() + us;
         std::cout << "wait_until     ";
        detail::print_time(st);
        std::cout << " which is " << (st - system_clock::now()).count()
          << " microseconds away\n";
        return true;
    }
};
}
```

Next follows how simple is the usage of this functions:

```
boost::mutex m;
boost::timed_mutex mut;
boost::condition_variable cv;

using namespace boost;

this_thread::sleep_for(chrono::seconds(3));
this_thread::sleep_for(chrono::nanoseconds(300));
chrono::system_clock::time_point time_limit = chrono::sys↵
tem_clock::now() + chrono::__seconds_(4) + chrono::milliseconds(500);
this_thread::sleep_until(time_limit);

mut.try_lock_for(chrono::milliseconds(30));
mut.try_lock_until(time_limit);

cv.wait_for(m, chrono::minutes(1));    // real code would put this in a loop
cv.wait_until(m, time_limit);  // real code would put this in a loop

// For those who prefer floating-point
this_thread::sleep_for(chrono::duration<double>(0.25));
this_thread::sleep_until(chrono::system_clock::now() + chrono::duration<double>(1.5));
```

See the source file example/simulated_thread_interface_demo.cpp

IO

French Output

Example use of output in French

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

int main()
{
    using namespace std;
    using namespace boost;
    using namespace boost::chrono;

    cout.imbue(locale(locale(), new duration_punct<char>
        (
            duration_punct<char>::use_long,
            "secondes", "minutes", "heures",
            "s", "m", "h"
        )));
    hours h(5);
    minutes m(45);
    seconds s(15);
    milliseconds ms(763);
    cout << h << ", " << m << ", " << s << " et " << ms << '\n';
}
```

Output is:

```
5 heures, 45 minutes, 15 secondes et 763 millisecondes
```

See the source file example/french.cpp

External Resources

C++ Standards Committee's current Working Paper	The most authoritative reference material for the library is the C++ Standards Committee's current Working Paper (WP). 20.11 Time utilities "time"
N2661 - A Foundation to Sleep On	From Howard E. Hinnant, Walter E. Brown, Jeff Garland and Marc Paterno. Is very informative and provides motivation for key design decisions
LGW 934. duration is missing operator%	From Terry Golubiewski. Is very informative and provides motivation for key design decisions
LGW 935. clock error handling needs to be specified	From Beman Dawes. This issue has been stated as NAD Future.

Reference

As `constexpr` will not be supported by some compilers, it is replaced in the code by BOOST_CHRONO_CONSTEXPR for constexpr functions and BOOST_CHRONO_STATIC_CONSTEXPR for struct/class static fields. The documentation doesn't use these macros.

Included on the C++0x Recommendation

Header <boost/chrono.hpp>

```
#include <boost/chrono/duration.hpp>
#include <boost/chrono/time_point.hpp>
#include <boost/chrono/system_clocks.hpp>
#include <boost/chrono/typeof/boost/chrono/chrono.hpp>
```

Limitations and Extensions

Next follows limitation respect to the C++0x recomendations:

- constexpr not tested yet.
- The recently steady_clock approved is not yet included.

The current implementation provides in addition:

- clock error handling as specified in clock error handling needs to be specified.
- process and thread clocks.

Configuration Macros

How Assert Behaves?

When `BOOST_NO_STATIC_ASSERT` is defined, the user can select the way static assertions are reported. Define

- `BOOST_CHRONO_USES_STATIC_ASSERT`: define it if you want to use Boost.StaticAssert.
- `BOOST_CHRONO_USES_MPL_ASSERT`: define it if you want to use Boost.MPL static asertions.
- `BOOST_CHRONO_USES_ARRAY_ASSERT`: define it if you want to use internal static asertions.

The default behavior is as `BOOST_CHRONO_USES_ARRAY_ASSERT` was defined.

When `BOOST_CHRONO_USES_MPL_ASSERT` is not defined the following symbols are defined as

```
#define BOOST_CHRONO_A_DURATION_REPRESENTATION_CAN_NOT_BE_A_DURATION \
    "A duration representation can not be a duration"
#define BOOST_CHRONO_SECOND_TEMPLATE_PARAMETER_OF_DURATION_MUST_BE_A_STD_RATIO \
    "Second template parameter of duration must be a boost::ratio"
#define BOOST_CHRONO_DURATION_PERIOD_MUST_BE_POSITIVE \
    "duration period must be positive"
#define BOOST_CHRONO_SECOND_TEMPLATE_PARAMETER_OF_TIME_POINT_MUST_BE_A_BOOST_CHRONO_DURATION \
    "Second template parameter of time_point must be a boost::chrono::duration"
```

Depending on the static assertion used system you will have an hint of the failing assertion either through the symbol or through the text.

How to Build Boost.Chrono as a Header Only Library?

When `BOOST_CHRONO_HEADER_ONLY` is defined the lib is header-only.

If in addition `BOOST_USE_WINDOWS_H` is defined `<windows.h>` is included, otherwise files in `boost/detail/win` are used to reduce the impact of including `<windows.h>`.

Header <boost/chrono/duration.hpp>

This file contains duration specific classes and non-member functions.

```
namespace boost {
  namespace chrono {

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

  }
  template <class Rep1, class Period1, class Rep2, class Period2>
  struct common_type<duration<Rep1, Period1>,
                     duration<Rep2, Period2> >;

  namespace chrono {

    // customization traits
    template <class Rep> struct treat_as_floating_point;
    template <class Rep> struct duration_values;

    // duration arithmetic
    template <class Rep1, class Period1, class Rep2, class Period2>
    typename common_type<duration<Rep1, Period1>, duration<Rep2, Period2> >::type
    operator+(
        const duration<Rep1, Period1>& lhs,
        const duration<Rep2, Period2>& rhs);

    template <class Rep1, class Period1, class Rep2, class Period2>
    typename common_type<duration<Rep1, Period1>, duration<Rep2, Period2> >::type
    operator-(
        const duration<Rep1, Period1>& lhs,
        const duration<Rep2, Period2>& rhs);

    template <class Rep1, class Period, class Rep2>
    duration<typename common_type<Rep1, Rep2>::type, Period>
    operator*(
        const duration<Rep1, Period>& d,
        const Rep2& s);

    template <class Rep1, class Period, class Rep2>
    duration<typename common_type<Rep1, Rep2>::type, Period>
    operator*(
        const Rep1& s,
        const duration<Rep2, Period>& d);

    template <class Rep1, class Period, class Rep2>
    duration<typename common_type<Rep1, Rep2>::type, Period>
    operator/(
        const duration<Rep1, Period>& d,
        const Rep2& s);

    template <class Rep1, class Period1, class Rep2, class Period2>
    typename common_type<Rep1, Rep2>::type
    operator/(
        const duration<Rep1, Period1>& lhs,
        const duration<Rep2, Period2>& rhs);

    #ifdef BOOST_CHRONO_EXTENSIONS
    // Used to get frecuency of events
    template <class Rep1, class Rep2, class Period>
    double operator/(
        const Rep1& s,
```

```
        const duration<Rep2, Period>& d);
    #endif

    // duration comparisons
    template <class Rep1, class Period1, class Rep2, class Period2>
    constexpr bool operator==(
        const duration<Rep1, Period1>& lhs,
        const duration<Rep2, Period2>& rhs);

    template <class Rep1, class Period1, class Rep2, class Period2>
    constexpr bool operator!=(
        const duration<Rep1, Period1>& lhs,
        const duration<Rep2, Period2>& rhs);

    template <class Rep1, class Period1, class Rep2, class Period2>
    constexpr bool operator<(
        const duration<Rep1, Period1>& lhs,
        const duration<Rep2, Period2>& rhs);
    template <class Rep1, class Period1, class Rep2, class Period2>
    constexpr bool operator<=(
        const duration<Rep1, Period1>& lhs,
        const duration<Rep2, Period2>& rhs);

    template <class Rep1, class Period1, class Rep2, class Period2>
    constexpr bool operator> (
        const duration<Rep1, Period1>& lhs,
        const duration<Rep2, Period2>& rhs);
    template <class Rep1, class Period1, class Rep2, class Period2>
    constexpr bool operator>=(
        const duration<Rep1, Period1>& lhs,
        const duration<Rep2, Period2>& rhs);

    // duration_cast

    template <class ToDuration, class Rep, class Period>
    ToDuration duration_cast(const duration<Rep, Period>& d);

    // convenience typedefs
    typedef duration<boost::int_least64_t, nano> nanoseconds;    // at least 64 bits needed
    typedef duration<boost::int_least64_t, micro> microseconds;  // at least 55 bits needed
    typedef duration<boost::int_least64_t, milli> milliseconds;  // at least 45 bits needed
    typedef duration<boost::int_least64_t> seconds;              // at least 35 bits needed
    typedef duration<boost::int_least32_t, ratio< 60> > minutes; // at least 29 bits needed
    typedef duration<boost::int_least32_t, ratio<3600> > hours;  // at least 23 bits needed

  }
}
```

Time-related Traits

Metafunction `treat_as_floating_point<>`

```
template <class Rep> struct treat_as_floating_point
    : boost::is_floating_point<Rep> {};
```

The `duration` template uses the `treat_as_floating_point` trait to help determine if a `duration` with one tick period can be converted to another `duration` with a different tick period. If `treat_as_floating_point<Rep>::value` is `true`, then `Rep` is a floating-point type and implicit conversions are allowed among `duration`s. Otherwise, the implicit convertibility depends on the tick periods of the `duration`s. If `Rep` is a class type which emulates a floating-point type, the author of `Rep` can specialize `treat_as_floating_point` so that `duration` will treat this `Rep` as if it were a floating-point type. Otherwise `Rep` is assumed to be an integral type, or a class emulating an integral type.

Class Template `duration_values`

```
template <class Rep>
struct duration_values
{
public:
    static constexpr Rep zero();
    static constexpr Rep max();
    static constexpr Rep min();
};
```

The `duration` template uses the `duration_values` trait to construct special values of the `duration`'s representation (`Rep`). This is done because the representation might be a class type with behavior which requires some other implementation to return these special values. In that case, the author of that class type should specialize `duration_values` to return the indicated values.

Static Member Function `zero()`

```
static constexpr Rep zero();
```

Returns: `Rep(0)`. **Note:** `Rep(0)` is specified instead of `Rep()` since `Rep()` may have some other meaning, such as an uninitialized value.

Remarks: The value returned corresponds to the additive identity.

Static Member Function `max()`

```
static constexpr Rep max();
```

Returns: `numeric_limits<Rep>::max()`.

Remarks: The value returned compares greater than zero().

Static Member Function `min()`

```
static constexpr Rep min();
```

Returns: `numeric_limits<Rep>::lowest()`.

Remarks: The value returned compares less than or equal to `zero()`.

`common_type` Specialization

```
template <class Rep1, class Period1, class Rep2, class Period2>
struct common_type<chrono::duration<Rep1, Period1>, chrono::duration<Rep2, Period2> >
{
    typedef chrono::duration<typename common_type<Rep1, Rep2>::type, see bellow> type;
};
```

The period of the `duration` indicated by this specialization of `common_type` is the greatest common divisor of `Period1` and `Period2`. This can be computed by forming a `ratio` of the greatest common divisor of `Period1::num` and `Period2::num`, and the least common multiple of `Period1::den` and `Period2::den`.

Note: The typedef type is the `duration` with the largest tick period possible where both `duration` arguments will convert to it without requiring a division operation. The representation of this type is intended to be able to hold any value resulting from this conversion, with the possible exception of round-off error when floating-point `duration`s are involved (but not truncation error).

Class Template `duration<>`

A `duration` measures time between two points in time (`time_point`). A `duration` has a representation which holds a count of ticks, and a tick period. The tick period is the amount of time which occurs from one tick to another in units of a second. It is expressed as a rational constant using `ratio`.

```
namespace boost { namespace chrono {

    template <class Rep, class Period>
    class duration {
    public:
        typedef Rep rep;
        typedef Period period;
    private:
        rep rep_; // exposition only
    public:
        constexpr duration();
        template <class Rep2>
        constexpr explicit duration(const Rep2& r);

        template <class Rep2, class Period2>
        constexpr duration(const duration<Rep2, Period2>& d);

        duration& operator=(const duration&) = default;

        constexpr rep count() const;

        constexpr duration  operator+();
        constexpr duration  operator-();
        duration& operator++();
        duration  operator++(int);
        duration& operator--();
        duration  operator--(int);

        duration& operator+=(const duration& d);
        duration& operator-=(const duration& d);

        duration& operator*=(const rep& rhs);
        duration& operator/=(const rep& rhs);
        duration& operator%=(const rep& rhs);
        duration& operator%=(const duration& rhs);

        static constexpr duration zero();
        static constexpr duration min();
        static constexpr duration max();
    };

}}
```

`Rep` must be an arithmetic type, or a class emulating an arithmetic type, compile diagnostic otherwise. If `duration` is instantiated with the type of `Rep` being a `duration`, compile diagnostic is issued.

`Period` must be an instantiation of `ratio`, compile diagnostic otherwise.

`Period::num` must be positive, compile diagnostic otherwise.

Examples:

- `duration<long, ratio<60> >` holds a count of minutes using a long.
- `duration<long long, milli>` holds a count of milliseconds using a long long.

- `duration<double, ratio<1, 30> >` holds a count using a double with a tick period of 1/30 second (a tick frequency of 30 Hz).

The following members of `duration` do not throw an exception unless the indicated operations on the representations throw an exception.

Constructor `duration(const Rep2&)`

```
template <class Rep2>
constexpr explicit duration(const Rep2& r);
```

Remarks: `Rep2` is implicitly convertible to `rep`, and

- `treat_as_floating_point<rep>::value` is `true`, or
- `!treat_as_floating_point<rep>::value && !treat_as_floating_point<Rep2>::value` is `true`.

If these constraints are not met, this constructor will not participate in overload resolution. **Note:** This requirement prevents construction of an integral-based `duration` with a floating-point representation. Such a construction could easily lead to confusion about the value of the `duration`.

Example:

```
duration<int, milli> d(3.5);  // do not compile
duration<int, milli> d(3);    // ok
```

Effects: Constructs an object of type `duration`.

PostConditions: `count() == static_cast<rep>(r)`.

Constructor `duration(const duration&)`

```
template <class Rep2, class Period2>
constexpr duration(const duration<Rep2, Period2>& d);
```

Remarks: `treat_as_floating_point<rep>::value`, or `ratio_divide<Period2, period>::type::den == 1`, else this constructor will not participate in overload resolution. **note** This requirement prevents implicit truncation error when converting between integral-based `duration`s. Such a construction could easily lead to confusion about the value of the `duration`.

Example:

```
duration<int, milli> ms(3);
duration<int, micro> us = ms;  // ok
duration<int, milli> ms2 = us; // do not compile
```

Effects: Constructs an object of type `duration`, constructing `rep_` from `duration_cast<duration>(d).count()`.

Member Function `count() const`

```
constexpr rep count() const;
```

Returns: `rep_`.

Member Function `operator+() const`

```
constexpr duration operator+() const;
```

Returns: `*this`.

Member Function `operator-() const`

```
constexpr duration operator-() const;
```

Returns: `duration(-rep_)`.

Member Function `operator++()`

```
duration& operator++();
```

Effects: `++rep_`.

Returns: `*this`.

Member Function `operator++(int)`

```
duration operator++(int);
```

Returns: `duration(rep_++)`.

Member Function `operator--()`

```
duration& operator--();
```

Effects: `--rep_`.

Returns: `*this`.

Member Function `operator--(int)`

```
duration operator--(int);
```

Returns: `duration(rep_--)`.

Member Function `operator+=(const duration&)`

```
duration& operator+=(const duration& d);
```

Effects: `rep_ += d.count()`.

Returns: `*this`.

Member Function `operator-=(const duration&)`

```
duration& operator-=(const duration& d);
```

Effects: `rep_ -= d.count()`.

Returns: `*this`.

Member Function `operator%=(const duration&)`

```
duration& operator%=(const duration& d);
```

Effects: `rep_ %= d.count()`.

Returns: `*this`.

Member Function `operator*=(const rep&)`

```
duration& operator*=(const rep& rhs);
```

Effects: `rep_ *= rhs`.

Returns: `*this`.

Member Function `operator/=(const rep&)`

```
duration& operator/=(const rep& rhs);
```

Effects: `rep_ /= rhs`.

Returns: `*this`.

Member Function `operator%=(const rep&)`

```
duration& operator%=(const rep& rhs);
```

Effects: `rep_ %= rhs`.

Returns: `*this`.

Static Member Function `zero()`

```
static constexpr duration zero();
```

Returns: `duration(duration_values<rep>::zero())`.

Static Member Function `min()`

```
static constexpr duration min();
```

Returns: `duration(duration_values<rep>::min())`.

Static Member Function `max()`

```
static constexpr duration max();
```

Returns: `duration(duration_values<rep>::max())`.

`duration` Non-Member Arithmetic

Non-Member Function `operator+(duration,duration)`

```
template <class Rep1, class Period1, class Rep2, class Period2>
typename common_type<duration<Rep1, Period1>, duration<Rep2, Period2> >::type
operator+(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
```

Returns: `CD(lhs) += rhs` where `CD` is the type of the return value.

Non-Member Function `operator-(duration,duration)`

```
template <class Rep1, class Period1, class Rep2, class Period2>
typename common_type<duration<Rep1, Period1>, duration<Rep2, Period2> >::type
operator-(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
```

Returns: `CD(lhs) -= rhs` where `CD` is the type of the return value.

```
template <class Rep1, class Period, class Rep2>
duration<typename common_type<Rep1, Rep2>::type, Period>
operator*(const duration<Rep1, Period>& d, const Rep2& s);
```

Requires: Let `CR` represent the `common_type` of `Rep1` and `Rep2`. This function will not participate in overload resolution unless both `Rep1` and `Rep2` are implicitly convertible to `CR`.

Returns: `duration<CR, Period>(d) *= s`.

Non-Member Function `operator*(Rep1,duration)`

```
template <class Rep1, class Period, class Rep2>
duration<typename common_type<Rep1, Rep2>::type, Period>
operator*(const Rep1& s, const duration<Rep2, Period>& d);
```

Requires: Let `CR` represent the `common_type` of `Rep1` and `Rep2`. This function will not participate in overload resolution unless both `Rep1` and `Rep2` are implicitly convertible to `CR`.

Returns: `d * s`.

Non-Member Function `operator/(duration,Rep2)`

```
template <class Rep1, class Period, class Rep2>
duration<typename common_type<Rep1, Rep2>::type, Period>
operator/(const duration<Rep1, Period>& d, const Rep2& s);
```

Requires: Let `CR` represent the `common_type` of `Rep1` and `Rep2`. This function will not participate in overload resolution unless both `Rep1` and `Rep2` are implicitly convertible to `CR`, and `Rep2` is not an instantiation of `duration`.

Returns: `duration<CR, Period>(d) /= s`.

Non-Member Function `operator/(duration,duration)`

```
template <class Rep1, class Period1, class Rep2, class Period2>
typename common_type<Rep1, Rep2>::type
operator/(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
```

Returns: Let `CD` represent the `common_type` of the two `duration` arguments. Returns `CD(lhs).count() / CD(rhs).count()`.

Non-Member Function `operator/(Rep1,duration)`

Included only if BOOST_CHRONO_EXTENSIONS is defined.

This overloading could be used to get the frequency of an event counted by `Rep1`.

```
template <class Rep1, class Rep2, class Period>
double operator/(const Rep1& s, const duration<Rep2, Period>& d);
```

Remarks: Let `CR` represent the `common_type` of `Rep1` and `Rep2`. This function will not participate in overload resolution unless both `Rep1` and `Rep2` are implicitly convertible to `CR`, and `Rep1` is not an instantiation of `duration`.

Returns: `CR(s)/duration<CR, Period>(d).count()`.

Non-Member Function `operator%(duration,Rep2)`

```
template <class Rep1, class Period, class Rep2>
duration<typename common_type<Rep1, Rep2>::type, Period>
operator%(const duration<Rep1, Period>& d, const Rep2& s);
```

Remarks This function will not participate in overload resolution unless Rep2 must be implicitly convertible to CR(Rep1, Rep2) and Rep2 must not be an instantiation of `duration`.

Returns: `duration`<CR(Rep1,Rep2), Period>(d) %= s.

Non-Member Function `operator%(duration,duration)`

```
template <class Rep1, class Period1, class Rep2, class Period2>
typename common_type<duration<Rep1, Period1>, duration<Rep2, Period2> >::type
operator%(const duration<Rep1, Period1>& lhs,
          const duration<Rep2, Period2>& rhs);
```

Remarks This function will not participate in overload resolution unless

Returns: CD(lhs) %= CD(rhs)

`duration` Non-Member Comparaisons

Non-Member Function `operator==(duration,duration)`

```
template <class Rep1, class Period1, class Rep2, class Period2>
bool operator==(const duration<Rep1, Period1>& lhs,
                const duration<Rep2, Period2>& rhs);
```

Returns: Let `CD` represent the `common_type` of the two `duration` arguments. Returns `CD(lhs).count() == CD(rhs).count()`

Non-Member Function `operator!=(duration,duration)`

```
template <class Rep1, class Period1, class Rep2, class Period2>
bool operator!=(const duration<Rep1, Period1>& lhs,
                const duration<Rep2, Period2>& rhs);
```

Returns: `!(lhs == rhs)`.

Non-Member Function `operator<(duration,duration)`

```
template <class Rep1, class Period1, class Rep2, class Period2>
bool operator< (const duration<Rep1, Period1>& lhs,
                const duration<Rep2, Period2>& rhs);
```

Returns: Let `CD` represent the `common_type` of the two `duration` arguments. Returns `CD(lhs).count() < CD(rhs).count()`

Non-Member Function `operator<=(duration,duration)`

```
template <class Rep1, class Period1, class Rep2, class Period2>
bool operator<=(const duration<Rep1, Period1>& lhs,
                const duration<Rep2, Period2>& rhs);
```

Returns: `!(rhs < lhs)`.

Non-Member Function `operator>(duration,duration)`

```
template <class Rep1, class Period1, class Rep2, class Period2>
bool operator> (const duration<Rep1, Period1>& lhs,
                const duration<Rep2, Period2>& rhs);
```

Returns: `rhs < lhs`.

Non-Member Function `operator>=(duration,duration)`

```
template <class Rep1, class Period1, class Rep2, class Period2>
bool operator>=(const duration<Rep1, Period1>& lhs,
                const duration<Rep2, Period2>& rhs);
```

Returns: `!(lhs < rhs)`.

Non-Member Function `duration_cast(duration)`

```
template <class ToDuration, class Rep, class Period>
ToDuration duration_cast(const duration<Rep, Period>& d);
```

Requires: This function will not participate in overload resolution unless `ToDuration` is an instantiation of `duration`.

Returns: Forms `CF` which is a `ratio` resulting from `ratio_divide<Period, typename ToDuration::period>::type`. Let `CR` be the `common_type` of `ToDuration::rep`, `Rep`, and `intmax_t`.

- If `CF::num == 1` and `CF::den == 1`, then returns `ToDuration(static_cast<typename ToDuration::rep>(d.count()))`
- else if `CF::num != 1` and `CF::den == 1`, then returns `ToDuration(static_cast<typename ToDuration::rep>(static_cast<CR>(d.count()) * static_cast<CR>(CF::num)))`
- else if `CF::num == 1` and `CF::den != 1`, then returns `ToDuration(static_cast<typename ToDuration::rep>(static_cast<CR>(d.count()) / static_cast<CR>(CF::den)))`
- else returns `ToDuration(static_cast<typename ToDuration::rep>(static_cast<CR>(d.count()) * static_cast<CR>(CF::num) / static_cast<CR>(CF::den)))`

Remarks: This function does not rely on any implicit conversions. All conversions must be accomplished through `static_cast`. The implementation avoids all multiplications or divisions when it is known at compile-time that it can be avoided because one or more arguments are `1`. All intermediate computations are carried out in the widest possible representation and only converted to the destination representation at the final step.

`duration` typedefs

```
// convenience typedefs
typedef duration<boost::int_least64_t, nano> nanoseconds;    // at least 64 bits needed
typedef duration<boost::int_least64_t, micro> microseconds;  // at least 55 bits needed
typedef duration<boost::int_least64_t, milli> milliseconds;  // at least 45 bits needed
typedef duration<boost::int_least64_t> seconds;              // at least 35 bits needed
typedef duration<boost::int_least32_t, ratio< 60> > minutes; // at least 29 bits needed
typedef duration<boost::int_least32_t, ratio<3600> > hours;  // at least 23 bits needed
```

`clock` Requirements

A clock represents a bundle consisting of a `duration`, a `time_point`, and a function `now()` to get the current `time_point`. A clock must meet the requirements in the following Table.

In this table `C1` and `C2` denote `clock` types. `t1` and `t2` are values returned from `C1::now()` where the call returning `t1` happens before the call returning `t2` and both of these calls happen before `C1::time_point::max()`.

Table 1. Clock Requirements

expression	return type	operational semantics
`C1::rep`	An arithmetic type or class emulating an arithmetic type.	The representation type of the `duration` and `time_point`.
`C1::period`	`ratio`	The tick period of the clock in seconds.
`C1::duration`	`chrono::duration<C1::rep, C1::period>`	The `duration` type of the `clock`.
`C1::time_point`	`chrono::time_point<C1>` or `chrono::time_point<C2, C1::duration>`	The `time_point` type of the `clock`. Different clocks are permitted to share a `time_point` definition if it is valid to compare their time_points by comparing their respective `duration`s. `C1` and `C2` must refer to the same epoch.
`C1::is_steady`	`constexpr bool`	`true` if `t1 <= t2` is always `true`, else `false`. **Note**: A `clock` that can be adjusted backwards is not steady
`C1::now()`	`C1::time_point`	Returns a `time_point` representing the current point in time.

Models of Clock:

- `system_clock`
- `steady_clock`
- `high_resolution_clock`
- `process_real_cpu_clock`
- `process_user_cpu_clock`
- `process_system_cpu_clock`
- `thread_clock`

Header <boost/chrono/time_point.hpp>

This file contains time_point specific classes and non-member functions.

```
namespace boost {
  namespace chrono {

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

  }
  template <class Clock, class Duration1, class Duration2>
  struct common_type<time_point<Clock, Duration1>,
                     time_point<Clock, Duration2> >;

  namespace chrono {

    // time_point arithmetic
    template <class Clock, class Duration1, class Rep2, class Period2>
    time_point<Clock, typename common_type<Duration1, duration<Rep2, Period2> >::type>
    operator+(const time_point<Clock, Duration1>& lhs, const duration<Rep2, Period2>& rhs);

    template <class Rep1, class Period1, class Clock, class Duration2>
    time_point<Clock, typename common_type<duration<Rep1, Period1>, Duration2>::type>
    operator+(const duration<Rep1, Period1>& lhs, const time_point<Clock, Duration2>& rhs);

    template <class Clock, class Duration1, class Rep2, class Period2>
    time_point<Clock, typename common_type<Duration1, duration<Rep2, Period2> >::type>
    operator-(const time_point<Clock, Duration1>& lhs, const duration<Rep2, Period2>& rhs);

    template <class Clock, class Duration1, class Duration2>
    typename common_type<Duration1, Duration2>::type
    operator-(const time_point<Clock, Duration1>& lhs, const time_point<Clock,
            Duration2>& rhs);

    // time_point comparisons
    template <class Clock, class Duration1, class Duration2>
    constexpr bool operator==(const time_point<Clock, Duration1>& lhs,
                    const time_point<Clock, Duration2>& rhs);
    template <class Clock, class Duration1, class Duration2>
    constexpr bool operator!=(const time_point<Clock, Duration1>& lhs,
                    const time_point<Clock, Duration2>& rhs);
    template <class Clock, class Duration1, class Duration2>
    constexpr bool operator< (const time_point<Clock, Duration1>& lhs,
                    const time_point<Clock, Duration2>& rhs);
    template <class Clock, class Duration1, class Duration2>
    constexpr bool operator<=(const time_point<Clock, Duration1>& lhs,
                    const time_point<Clock, Duration2>& rhs);
    template <class Clock, class Duration1, class Duration2>
    constexpr bool operator> (const time_point<Clock, Duration1>& lhs,
                    const time_point<Clock, Duration2>& rhs);
    template <class Clock, class Duration1, class Duration2>
    constexpr bool operator>=(const time_point<Clock, Duration1>& lhs,
                    const time_point<Clock, Duration2>& rhs);

    // time_point_cast
    template <class ToDuration, class Clock, class Duration>
   constexpr time_point<Clock, ToDuration> time_point_cast(const time_point<Clock, Duration>& t);

  }
}
```

`common_type` specialization

```
template <class Clock, class Duration1, class Duration2>
struct common_type<chrono::time_point<Clock, Duration1>, chrono::time_point<Clock, Duration2> >
{
    typedef chrono::time_point<Clock, typename common_type<Duration1, Duration2>::type> type;
};
```

The `common_type` of two `time_point`s is a `time_point` with the same `clock` (both have the same `clock`), and the `common_type` of the two `duration`s.

Class template `time_point<>`

A `time_point` represents a point in time with respect to a specific clock.

```
template <class Clock, class Duration>
class time_point {
public:
    typedef Clock                     clock;
    typedef Duration                  duration;
    typedef typename duration::rep    rep;
    typedef typename duration::period period;
private:
    duration d_; // exposition only
public:
    constexpr time_point();
    constexpr explicit time_point(const duration& d);

    // conversions
    template <class Duration2>
    time_point(const time_point<clock, Duration2>& t);

    // observer

    constexpr duration time_since_epoch() const;

    // arithmetic

    time_point& operator+=(const duration& d);
    time_point& operator-=(const duration& d);

    // special values

    static constexpr time_point min();
    static constexpr time_point max();
};
```

Clock must meet the `Clock` requirements.

Duration must be an instantiation of `duration`, compile diagnostic otherwise.

Constructor `time_point()`

```
constexpr time_point();
```

Effects: Constructs an object of `time_point`, initializing `d_` with `duration::zero()`. This `time_point` represents the epoch.

Constructor `time_point(const duration&)`

```
constexpr time_point(const duration& d);
```

Effects: Constructs an object of `time_point`, initializing `d_` with `d`. This `time_point` represents the epoch `+ d`.

Copy Constructor `time_point(const time_point&)`

```
template <class Duration2> time_point(const time_point<clock, Duration2>& t);
```

Requires: This function will not participate in overload resolution unless `Duration2` is implicitly convertible to `duration`.

Effects: Constructs an object of `time_point`, initializing `d_` with `t.time_since_epoch()`.

Member Function `time_since_epoch() const`

```
constexpr duration time_since_epoch() const;
```

Returns: `d_`.

Member Function `operator+=`

```
time_point& operator+=(const duration& d);
```

Effects: `d_ += d`.

Returns: `*this`.

Member Function `operator-=`

```
time_point& operator-=(const duration& d);
```

Effects: `d_ -= d`

Returns: `*this`.

Static Member Function `min`

```
static constexpr time_point min();
```

Returns: `time_point(duration::min())`.

Static Member Function `max`

```
static constexpr time_point max();
```

Returns: `time_point(duration::max())`.

`time_point` non-member arithmetic

Non-Member Function `operator+(time_point,duration)`

```
template <class Clock, class Duration1, class Rep2, class Period2>
time_point<Clock, typename common_type<Duration1, duration<Rep2, Period2> >::type>
operator+(const time_point<Clock, Duration1>& lhs,
          const duration<Rep2, Period2>& rhs);
```

Returns: `CT(lhs) += rhs` where `CT` is the type of the return value.

Non-Member Function `operator+(duration,time_point)`

```
template <class Rep1, class Period1, class Clock, class Duration2>
time_point<Clock, typename common_type<duration<Rep1, Period1>, Duration2>::type>
operator+(const duration<Rep1, Period1>& lhs,
          const time_point<Clock, Duration2>& rhs);
```

Returns: `rhs + lhs`.

Non-Member Function `operator-(time_point,duration)`

```
template <class Clock, class Duration1, class Rep2, class Period2>
time_point<Clock, typename common_type<Duration1, duration<Rep2, Period2> >::type>
operator-(const time_point<Clock, Duration1>& lhs,
          const duration<Rep2, Period2>& rhs);
```

Returns: `lhs + (-rhs)`.

Non-Member Function `operator-(time_point,time_point)`

```
template <class Clock, class Duration1, class Duration2>
typename common_type<Duration1, Duration2>::type
operator-(const time_point<Clock, Duration1>& lhs,
          const time_point<Clock, Duration2>& rhs);
```

Returns: `lhs.time_since_epoch() - rhs.time_since_epoch()`.

`time_point` non-member comparisons

Non-Member Function `operator==(time_point,time_point)`

```
template <class Clock, class Duration1, class Duration2>
bool operator==(const time_point<Clock, Duration1>& lhs,
                const time_point<Clock, Duration2>& rhs);
```

Returns: `lhs.time_since_epoch() == rhs.time_since_epoch()`.

Non-Member Function `operator!=(time_point,time_point)`

template <class Clock, class Duration1, class Duration2> bool operator!=(const `time_point`<Clock, Duration1>& lhs, const `time_point`<Clock, Duration2>& rhs);

Returns: `!(lhs == rhs)`.

Non-Member Function `operator<(time_point,time_point)`

```
template <class Clock, class Duration1, class Duration2>
bool operator< (const time_point<Clock, Duration1>& lhs,
                const time_point<Clock, Duration2>& rhs);
```

Returns: lhs.time_since_epoch() < rhs.time_since_epoch().

Non-Member Function `operator<=(time_point,time_point)`

```
template <class Clock, class Duration1, class Duration2>
bool operator<=(const time_point<Clock, Duration1>& lhs,
                const time_point<Clock, Duration2>& rhs);
```

Returns: `!(rhs < lhs)`.

Non-Member Function `operator>(time_point,time_point)`

template <class Clock, class Duration1, class Duration2> bool operator>(const `time_point`<Clock, Duration1>& lhs, const `time_point`<Clock, Duration2>& rhs);

Returns: `rhs < lhs`.

Non-Member Function `operator>=(time_point,time_point)`

```
template <class Clock, class Duration1, class Duration2>
bool operator>=(const time_point<Clock, Duration1>& lhs,
                const time_point<Clock, Duration2>& rhs);
```

Returns: `!(lhs < rhs)`.

Non-Member Function `time_point_cast(time_point)`

```
template <class ToDuration, class Clock, class Duration>
time_point<Clock, ToDuration> time_point_cast(const time_point<Clock, Duration>& t);
```

Requires: This function will not participate in overload resolution unless `ToDuration` is an instantiation of `duration`.

Returns: `time_point<Clock, ToDuration>(duration_cast<ToDuration>(t.time_since_epoch()))`.

Header `<boost/chrono/system_clocks.hpp>`

This file contains the standard clock classes.

```
namespace boost {
  namespace chrono {

    // Clocks
    class system_clock;
    class steady_clock;
    class high_resolution_clock;

  }
}
```

Class `system_clock`

The `system_clock` class provides a means of obtaining the current wall-clock time from the system-wide real-time clock. The current time can be obtained by calling `system_clock::now()`. Instances of `system_clock::time_point` can be converted to and from time_t with the `system_clock::to_time_t()` and `system_clock::to_time_point()` functions. If system clock is not steady, a subsequent call to `system_clock::now()` may return an earlier time than a previous call (e.g. if the operating system clock is manually adjusted, or synchronized with an external clock).

The current implementation of `system_clock` is related an epoch (midnight UTC of January 1, 1970), but this is not in the contract. You need to use the static function static

```
std::time_t to_time_t(const time_point& t);
```

which returns a `time_t` type that is based on midnight UTC of January 1, 1970.

```
class system_clock {
public:
    typedef see bellow          duration;
    typedef duration::rep                       rep;
    typedef duration::period                    period;
    typedef chrono::time_point<system_clock>    time_point;
    static constexpr bool is_steady =           false;

    static time_point  now();                       // throws on error
    static time_point  now(system::error_code & ec);  // never throws

    // Map to C API
    static std::time_t to_time_t(const time_point& t);
    static time_point  from_time_t(std::time_t t);
};
```

`system_clock` satisfy the `Clock` requirements:

- `system_clock::duration::min() < system_clock::duration::zero()` is `true`.
- The nested duration typedef has a resolution that depends on the one provided by the platform.

Static Member Function `to_time_t(time_point)`

time_t to_time_t(const time_point& t);

Returns: A `time_t` such that the `time_t` and `t` represent the same point in time, truncated to the courser of the precisions among `time_t` and `t`.

Static Member Function `from_time_t(time_t)`

```
time_point from_time_t(time_t t);
```

Returns: A `time_point` such that the `time_point` and `t` represent the same point in time, truncated to the coarser of the precisions among `time_point` and `t`.

Macro `BOOST_CHRONO_HAS_CLOCK_STEADY`

Defined if the platform support steady clocks.

Class `steady_clock`

`steady_clock` satisfy the `Clock` requirements.

`steady_clock` class provides access to the system-wide steady clock. The current time can be obtained by calling `steady_clock::now()`. There is no fixed relationship between values returned by `steady_clock::now()` and wall-clock time.

```
#ifdef BOOST_HAS_CLOCK_STEADY
    class steady_clock {
    public:
        typedef nanoseconds                          duration;
        typedef duration::rep                        rep;
        typedef duration::period                     period;
        typedef chrono::time_point<steady_clock>  time_point;
        static constexpr bool is_steady =            true;

        static time_point  now();                         // throws on error
        static time_point  now(system::error_code & ec);  // never throws
    };
#endif
```

Class `high_resolution_clock`

`high_resolution_clock` satisfy the `Clock` requirements.

```
#ifdef BOOST_CHRONO_HAS_CLOCK_STEADY
  typedef steady_clock high_resolution_clock;  // as permitted by [time.clock.hires]
#else
  typedef system_clock high_resolution_clock;  // as permitted by [time.clock.hires]
#endif
```

Header `<boost/chrono/typeof/boost/chrono/chrono.hpp>`

Register `duration`<> and `time_point`<> class templates to **Boost.Typeof**.

Chrono I/O

Header <boost/chrono/chrono_io.hpp>

```
namespace boost {
namespace chrono {

    template <class CharT>
    class duration_punct;

    template <class Clock, class CharT>
    struct clock_string;

    template <class CharT, class Traits>
        std::basic_ostream<CharT, Traits>&
        duration_short(std::basic_ostream<CharT, Traits>& os);

    template <class CharT, class Traits>
        std::basic_ostream<CharT, Traits>&
        duration_long(std::basic_ostream<CharT, Traits>& os);

    template <class CharT, class Traits, class Rep, class Period>
        std::basic_ostream<CharT, Traits>&
        operator<<(std::basic_ostream<CharT, Traits>& os, const duration<Rep, Period>& d);

    template <class CharT, class Traits, class Rep, class Period>
        std::basic_istream<CharT, Traits>&
        operator>>(std::basic_istream<CharT, Traits>& is, duration<Rep, Period>& d)

    template <class CharT, class Traits, class Clock, class Duration>
        std::basic_ostream<CharT, Traits>&
        operator<<(std::basic_ostream<CharT, Traits>& os,
               const time_point<Clock, Duration>& tp);

    template <class CharT, class Traits, class Clock, class Duration>
        std::basic_istream<CharT, Traits>&
        operator>>(std::basic_istream<CharT, Traits>& is,
               time_point<Clock, Duration>& tp);

}
}
```

Template Class duration_punct<>

The `duration` unit names can be customized through the facet: `duration_punct`. `duration` unit names come in two varieties: long and short. The default constructed `duration_punct` provides names in the long format. These names are English descriptions. Other languages are supported by constructing a `duration_punct` with the proper spellings for "hours", "minutes" and "seconds", and their abbreviations (for the short format).

```
template <class CharT>
class duration_punct
    : public std::locale::facet
{
public:
    typedef std::basic_string<CharT> string_type;
    enum {use_long, use_short};

    static std::locale::id id;

    explicit duration_punct(int use = use_long);

    duration_punct(int use,
        const string_type& long_seconds, const string_type& long_minutes,
        const string_type& long_hours, const string_type& short_seconds,
        const string_type& short_minutes, const string_type& short_hours);

    duration_punct(int use, const duration_punct& d);

    template <class Period> string_type short_name() const;
    template <class Period> string_type long_name() const;
    template <class Period> string_type name() const;

    bool is_short_name() const;
    bool is_long_name() const;
};
```

Template Class `clock_string<>`

```
template <class Clock, class CharT>
struct clock_string;
```

This template must be specialized for specific clocks. The specialization must define the following functions

```
static std::basic_string<CharT> name();
static std::basic_string<CharT> since();
```

`clock_string<>::name()` return the clock name, which usually corresponds to the class name. `clock_string<>::since()` return the textual format of the clock epoch.

`clock_string<system_clock>` Specialization

```
template <class CharT>
struct clock_string<system_clock, CharT>
{
    static std::basic_string<CharT> name();
    static std::basic_string<CharT> since();
};
```

`clock_string<>::name()` returns "system_clock". `clock_string<>::since()` returns " since Jan 1, 1970"

`clock_string<steady_clock>` Specialization

```
#ifdef BOOST_CHRONO_HAS_CLOCK_STEADY

template <class CharT>
struct clock_string<steady_clock, CharT>
{
    static std::basic_string<CharT> name();
    static std::basic_string<CharT> since();
};
#endif
```

`clock_string<>::name()` returns "steady_clock". `clock_string<>::since()` returns " since boot"

`clock_string<thread_clock>` Specialization

```
#if defined(BOOST_CHRONO_HAS_THREAD_CLOCK)
template <class CharT>
struct clock_string<thread_clock, CharT>
{
    static std::basic_string<CharT> name();
    static std::basic_string<CharT> since();
};
#endif
```

`clock_string<>::name()` returns "thread_clock". `clock_string<>::since()` returns " since thread start-up"

`clock_string<process_real_cpu_clock>` Specialization

```
template <class CharT>
struct clock_string<process_real_cpu_clock, CharT>
{
    static std::basic_string<CharT> name();
    static std::basic_string<CharT> since();
};
```

`clock_string<>::name()` returns "process_real_cpu_clock". `clock_string<>::since()` returns " since process start-up"

`clock_string<process_user_cpu_clock>` Specialization

```
template <class CharT>
struct clock_string<process_user_cpu_clock, CharT>
{
    static std::basic_string<CharT> name();
    static std::basic_string<CharT> since();
};
```

`clock_string<>::name()` returns "process_user_cpu_clock". `clock_string<>::since()` returns " since process start-up"

`clock_string<process_system_cpu_clock>` Specialization

```
template <class CharT>
struct clock_string<process_system_cpu_clock, CharT>
{
    static std::basic_string<CharT> name();
    static std::basic_string<CharT> since();
};
```

`clock_string<>::name()` returns "process_system_cpu_clock". `clock_string<>::since()` returns " since process start-up"

`clock_string<process_cpu_clock>` Specialization

```
template <class CharT>
struct clock_string<process_cpu_clock, CharT>
{
    static std::basic_string<CharT> name();
    static std::basic_string<CharT> since();
};
```

`clock_string<>::name()` returns "process_cpu_clock". `clock_string<>::since()` returns " since process start-up"

I/O Manipulators

The short or long format can be easily chosen by streaming a `duration_short` or `duration_long` manipulator respectively.

```
template <class CharT, class Traits>
    std::basic_ostream<CharT, Traits>&
    duration_short(std::basic_ostream<CharT, Traits>& os);
```

Effects: Set the `duration_punct` facet to stream `duration`s and `time_point`s as abbreviations.

Returns: the output stream

```
template <class CharT, class Traits>
    std::basic_ostream<CharT, Traits>&
    duration_long(std::basic_ostream<CharT, Traits>& os);
```

Effects: Set the `duration_punct` facet to stream durations and time_points as long text.

Returns: the output stream

I/O Streams Operations

Any `duration` can be streamed out to a `basic_ostream`. The run-time value of the `duration` is formatted according to the rules and current format settings for `duration::rep`. This is followed by a single space and then the compile-time unit name of the `duration`. This unit name is built on the string returned from `ratio_string<>` and the data used to construct the `duration_punct` which was inserted into the stream's locale. If a `duration_punct` has not been inserted into the stream's locale, a default constructed `duration_punct` will be added to the stream's locale.

A `time_point` is formatted by outputting its internal `duration` followed by a string that describes the `time_point::clock` epoch. This string will vary for each distinct clock, and for each implementation of the supplied clocks.

```
template <class CharT, class Traits, class Rep, class Period>
    std::basic_ostream<CharT, Traits>&
    operator<<(std::basic_ostream<CharT, Traits>& os, const duration<Rep, Period>& d);
```

Effects: outputs the `duration` as an abrevieated or long text format depending on the state of the `duration_punct` facet.

Returns: the output stream

```
template <class CharT, class Traits, class Rep, class Period>
    std::basic_istream<CharT, Traits>&
    operator>>(std::basic_istream<CharT, Traits>& is, duration<Rep, Period>& d)
```

Effects: reads a `duration` from the input stream. If a format error is found, the input stream state will be set to `failbit`.

Returns: the input stream

```
template <class CharT, class Traits, class Clock, class Duration>
    std::basic_ostream<CharT, Traits>&
    operator<<(std::basic_ostream<CharT, Traits>& os,
           const time_point<Clock, Duration>& tp);
```

Effects: outputs the `time_point` as an abrevieated or long text format depending on the state of the `duration_punct` facet.

Returns: the output stream

```
template <class CharT, class Traits, class Clock, class Duration>
    std::basic_istream<CharT, Traits>&
    operator>>(std::basic_istream<CharT, Traits>& is,
           time_point<Clock, Duration>& tp);
```

Effects: reads a `time_point` from the input stream. If a format error is found, the input stream state will be set to `failbit`.

Returns: the input stream

Other Clocks

Header `<boost/chrono/process_cpu_clocks.hpp>`

Knowing how long a program takes to execute is useful in both test and production environments. It is also helpful if such timing information is broken down into real (wall clock) time, CPU time spent by the user, and CPU time spent by the operating system servicing user requests.

Process clocks don't include the time spent by the child process.

```
namespace boost { namespace chrono {

    class process_real_cpu_clock;
    class process_user_cpu_clock;
    class process_system_cpu_clock;
    class process_cpu_clock;

    struct process_cpu_clock::times;
    template <class CharT, class Traits>
    std::basic_ostream<CharT, Traits>&
    operator<<(std::basic_ostream<CharT, Traits>& os,
            process_cpu_clock::times const& rhs);

    template <class CharT, class Traits>
    std::basic_istream<CharT, Traits>&
    operator>>(std::basic_istream<CharT, Traits>& is,
            process_cpu_clock::times const& rhs);

    template <>
    struct duration_values<process_cpu_clock::times>;
} }
namespace std {
    template <>
    class numeric_limits<boost::chrono::process_cpu_clock::times>;
}
```

Class `process_real_cpu_clock`

`process_real_cpu_clock` satisfy the `Clock` requirements.

`process_real_cpu_clock` class provides access to the real process wall-clock steady clock, i.e. the real CPU-time clock of the calling process. The process relative current time can be obtained by calling `process_real_cpu_clock::now()`.

```
class process_real_cpu_clock {
public:
    typedef nanoseconds                         duration;
    typedef duration::rep                       rep;
    typedef duration::period                    period;
    typedef chrono::time_point<process_real_cpu_clock>    time_point;
    static constexpr bool is_steady =           true;

    static time_point now( system::error_code & ec = system::throws );
};
```

Class `process_user_cpu_clock`

`process_user_cpu_clock` satisfy the `Clock` requirements.

`process_user_cpu_clock` class provides access to the user CPU-time steady clock of the calling process. The process relative user current time can be obtained by calling `process_user_cpu_clock::now()`.

```
class process_user_cpu_clock {
public:
    typedef nanoseconds                         duration;
    typedef duration::rep                       rep;
    typedef duration::period                    period;
    typedef chrono::time_point<process_user_cpu_clock>    time_point;
    static constexpr bool is_steady =           true;

    static time_point now( system::error_code & ec = system::throws );
};
```

Class `process_system_cpu_clock`

`process_system_cpu_clock` satisfy the `Clock` requirements.

`process_system_cpu_clock` class provides access to the system CPU-time steady clockof the calling process. The process relative system current time can be obtained by calling `process_system_cpu_clock::now()`.

```
class process_system_cpu_clock {
public:
    typedef nanoseconds                         duration;
    typedef duration::rep                       rep;
    typedef duration::period                    period;
    typedef chrono::time_point<process_system_cpu_clock>    time_point;
    static constexpr bool is_steady =           true;

    static time_point now( system::error_code & ec = system::throws );
};
```

Class `process_cpu_clock`

`process_cpu_clock` can be considered as a `tuple<process_real_cpu_clock, process_user_cpu_clock, process_system_cpu_clock>`.

`process_cpu_clock` provides a thin wrapper around the operating system's process time API. For POSIX-like systems, that's the times() function, while for Windows, it's the `GetProcessTimes()` function.

The process relative real, user and system current time can be obtained at once by calling `process_clocks::now()`.

```
class process_cpu_clock
{
public:
    struct times ;

    typedef duration<times,  nano>                  duration;
    typedef duration::rep                           rep;
    typedef duration::period                        period;
    typedef chrono::time_point<process_cpu_clock>   time_point;
    static constexpr bool is_steady =               true;

    static time_point now( system::error_code & ec = system::throws );
};
```

Class `process_cpu_clock::times`

This class is the representation of the `process_cpu_clock::duration` class. As such it needs to implements the arithmetic operators.

```
struct times : arithmetic<times,
    multiplicative<times, process_real_cpu_clock::rep,
    less_than_comparable<times> > >
{
    process_real_cpu_clock::rep   real;    // real (i.e wall clock) time
    process_user_cpu_clock::rep   user;    // user cpu time
    process_system_cpu_clock::rep system;  // system cpu time

    times();
    times(
        process_real_cpu_clock::rep r,
        process_user_cpu_clock::rep u,
        process_system_cpu_clock::rep s);

    bool operator==(times const& rhs);

    times operator+=(times const& rhs);
    times operator-=(times const& rhs);
    times operator*=(times const& rhs);
    times operator/=(times const& rhs);
    bool operator<(times const & rhs) const;
};
```

`process_cpu_clock::times` Input/Output

```
template <class CharT, class Traits>
std::basic_ostream<CharT, Traits>&
operator<<(std::basic_ostream<CharT, Traits>& os,
        process_cpu_clock::times const& rhs);
```

Effect: Output each part separated by ';' and sourrounded by '{', '}'. **Throws**: None.

```
template <class CharT, class Traits>
std::basic_istream<CharT, Traits>&
operator>>(std::basic_istream<CharT, Traits>& is,
        process_cpu_clock::times const& rhs);
```

Effect: overrides the value of rhs if the input stream has the format "{r;u;s}". Otherwise, set the input stream state as failbit | eofbit. **Throws**: None.

`duration_values` Specialization for `times`

```
template <>
struct duration_values<process_cpu_clock::times>
{
    static process_cpu_clock::times zero();
    static process_cpu_clock::times max();
    static process_cpu_clock::times min();
};
```

The `times` specific functions `zero()`, `max()` and `min()` uses the relative functions on the representation of each component.

`numeric_limits` Specialization for `times`

```
namespace std {
    template <>
    class numeric_limits<boost::chrono::process_cpu_clock::times> {
        typedef boost::chrono::process_cpu_clock::times Rep;

    public:
        static const bool is_specialized = true;
        static Rep min();
        static Rep max();
        static Rep lowest();
        static const int digits;
        static const int digits10;
        static const bool is_signed = false;
        static const bool is_integer = true;
        static const bool is_exact = true;
        static const int radix = 0;
    };
}
```

The `times` specialization functions `min()`, `max()` and `lowest()` uses the relative functions on the representation of each component.

Notes

- `min()` returns the tuple of mins.
- `max()` returns the tuple of maxs.
- `lowest()` returns the tuple of lowests.
- `digits` is the sum of (binary) digits.
- `digits10` is the sum of digits10s.

Header `<boost/chrono/thread_clock.hpp>`

Knowing the time a thread takes to execute is useful in both test and production environments.

```
#define BOOST_CHRONO_HAS_THREAD_CLOCK
#define BOOST_CHRONO_THREAD_CLOCK_IS_STEADY
namespace boost { namespace chrono {

    class thread_clock;

} }
```

Macro `BOOST_CHRONO_HAS_THREAD_CLOCK`

This macro is defined if the platform supports thread clocks.

Macro `BOOST_CHRONO_THREAD_CLOCK_IS_STEADY`

This macro is defined if the platform has a thread clock. Its value is true if it is steady and false otherwise.

Class `thread_clock`

`thread_clock` satisfy the `Clock` requirements.

`thread_clock` class provides access to the real thread wall-clock, i.e. the real CPU-time clock of the calling thread. The thread relative current time can be obtained by calling `thread_clock::now()`.

```
class thread_clock {
public:
    typedef nanoseconds                         duration;
    typedef duration::rep                       rep;
    typedef duration::period                    period;
    typedef chrono::time_point<thread_clock>    time_point;
    static constexpr bool is_steady =           BOOST_CHRONO_THREAD_CLOCK_IS_STEADY;

    static time_point now( system::error_code & ec = system::throws );
};
```

Appendices

Appendix A: History

Version 1.0.0, January 6, 2011

- Moved chrono to trunk taking in account the review remarks.
- Documentation revision.

Features:

- Boost_Chrono is now a configurable header-only library version (that also allows the user to choose if the `windows.h` file is included or not).
- Added clock_string<> traits.
- Define chrono-io for all the clocks.
- Add input of process_times representation.

Implementation:

- Use of detail/win files to avoid the use of windows.h file.
- Completed the error_code handling.
- Works now with BOOST_SYSTEM_NO_DEPRECATED.

Fixes:

- Fix some warnings.

- Fix original errors on Mac
- Don't fix the link with boost_system to static.

Test:

- Added test on process and thread clocks.
- Moved to lightweight_test.hpp.
- Able to test multiple configurations.

Doc:

- Removed some not useful parts as the test and the tickets.

Appendix B: Rationale

See **N2661 - A Foundation to Sleep On** which is very informative and provides motivation for key design decisions. This section contains some extracts from this document.

Why duration needs operator%

This operator is convenient for computing where in a time frame a given duration lies. A motivating example is converting a duration into a "broken-down" time duration such as hours::minutes::seconds:

```
class ClockTime
{
    typedef boost::chrono::hours hours;
    typedef boost::chrono::minutes minutes;
    typedef boost::chrono::seconds seconds;
public:
    hours hours_;
    minutes minutes_;
    seconds seconds_;

    template <class Rep, class Period>
      explicit ClockTime(const boost::chrono::duration<Rep, Period>& d)
        : hours_  (boost::chrono::duration_cast<hours>  (d)),
          minutes_(boost::chrono::duration_cast<minutes>(d % hours(1))),
          seconds_(boost::chrono::duration_cast<seconds>(d % minutes(1)))
          {}
};
```

Appendix C: Implementation Notes

Which APIs have been chosen to implement each clock on each platform?

The following table presents a resume of which API is uused for each clock on each platform

Table 2. Clock API correspondence

Clock	Windows Platform	Posix Platform	Mac Platform
`system_clock`	GetSystemTimeAsFileTime	clock_gettime(CLOCK_REALTIME)	gettimeofday
`steady_clock`	QueryPerformanceCounter and QueryPerformanceFrequency	clock_gettime(CLOCK_STEADY)	mach_timebase_info,mach_absolute_time
`process_real_cpu_clock`	GetProcessTimes	times	times
`process_system_cpu_clock`	GetProcessTimes	times	times
`process_user_cpu_clock`	GetProcessTimes	times	times
`process_cpu_clock`	GetProcessTimes	times	times
`thread_clock`	GetThreadTimes	clock_gettime(pthread_getcpuclockid)	clock_gettime(pthread_getcpuclockid)

Appendix D: FAQ

Why does process_cpu_clock sometimes give more cpu seconds than real seconds?

Ask your operating system supplier. The results have been inspected with a debugger, and both for Windows and Linux, that's what the OS appears to be reporting at times.

Are integer overflows in the duration arithmetic detected and reported?

Boost.Ratio avoids all kind of overflow that could result of arithmetic operation and that can be simplified. The typedefs durations don't detect overflow. You will need a duration representation that handles overflow.

Which clocks should be used to benchmarking?

Each clock has his own features. It depends on what do you need to benchmark. Most of the time, you could be interested in using a thread clock, but if you need to measure code subject to synchronization a process clock would be better. If you have a multi-process application, a system-wide clock could be needed.

Which clocks should be used for watching?

For trace purposes, it is probably best to use a system-wide clock.

Appendix E: Acknowledgements

The library's code was derived from Howard Hinnant's time2_demo prototype. Many thanks to Howard for making his code available under the Boost license. The original code was modified by Beman Dawes to conform to Boost conventions.

time2_demo contained this comment:

Much thanks to Andrei Alexandrescu, Walter Brown, Peter Dimov, Jeff Garland, Terry Golubiewski, Daniel Krugler, Anthony Williams.

The file <boost/chrono_io.hpp> has been adapted from the experimental header `<chrono_io>` from Howard Hinnant. Thanks for all Howard.

Howard Hinnant, who is the real author of the library, has provided valuable feedback and suggestions during the development of the library. In particular, The chrono_io_io.hpp source has been adapted from the experimental header `<chrono_io>` from Howard Hinnant.

The acceptance review of Boost.Ratio took place between November 5th and 15th 2010. Many thanks to Anthony Williams, the review manager, and to all the reviewers: David Deakins, John Bytheway, Roland Bock and Paul A. Bristow.

Thanks to Ronald Bock, Andrew Chinoff, Paul A. Bristow and John Bytheway for his help polishing the documentation.

Thanks to Tom Tan for reporting some compiler issues with MSVC V10 beta and MinGW-gcc-4.4.0 and for the many pushing for an homogeneous `process_cpu_clock` clock.

Thanks to Ronald Bock for reporting Valgind issues and for the many suggestions he made concerning the documentation.

Appendix F: Future plans

For later releases

- Add chrono utilities as defined By Howard Hinnant here.