A `<chrono>` Tutorial

It’s About Time

Howard Hinnant

sep/19/2016
Where You Can Find This Library

- Everything discussed in this presentation is found in the header `<chrono>.
- Everything is in namespace `std::chrono`. 
What We Will Be Talking About

• Motivation. Why `<chrono>`?

• Time durations

• Points in time

• Clocks

• Examples
Why Bother? (with <chrono>)

• Isn’t an integral count (of seconds or whatever) sufficient?

 sleep(10);

• Sleep for 10 seconds?
• 10 milliseconds?
• 10 nanoseconds?
Why Bother?
(with `<chrono>`)  

- Isn’t an integral count (of seconds or whatever) sufficient?

    `sleep(10ms);`

- Ah: 10 milliseconds.
Why Bother? (with <chrono>)

- In general using an arithmetic type to represent a duration or time point is *intrinsically ambiguous*.

- Help the compiler *help you* to find logic errors at **compile time** by making distinct concepts, distinct types.
What We Will Be Talking About

• Motivation. Why <chrono>?
  • Time durations
  • Points in time
  • Clocks
  • Examples
Time Duration

- A time duration is just a period of time.
- 3 seconds.
- 3 minutes.
- 3 hours.
seconds

- Lets start with `std::chrono::seconds`.
  - `seconds` is an arithmetic-like type.
  - `sizeof(seconds) == 8`.
  - It is trivially destructible.
  - It is trivially default constructible.
  - It is trivially copy constructible.
  - It is trivially copy assignable.
  - It is trivially move constructible.
  - It is trivially move assignable.
- This is all just like `long long` and `int64_t`.
seconds

Very simple, very fast:

class seconds
{
    int64_t sec_; 
public:
    seconds() = default;
    // etc.
    // ...
};
seconds

Scalar-like construction behavior:

seconds s;       // no initialization
seconds s{};    // zero initialization
seconds

Construction:

seconds s = 3;
// error: Not implicitly constructible from int
seconds

Construction:

seconds s = 3;
// error: Not implicitly constructible from int

seconds s{3}; // Ok: 3 seconds
seconds

Construction:

seconds s = 3;
// error: Not implicitly constructible from int

seconds s{3};    // Ok: 3 seconds
cout << s << '\n';  // unfortunately, not ok

But the library I present tomorrow fixes this.
seconds

Construction:

seconds s = 3;
// error: Not implicitly constructible from int

seconds s{3};      // Ok: 3 seconds
cout << s << '\n';  // unfortunately, not ok
cout << s.count() << "s\n";  // 3s
seconds

No implicit path from int to seconds!

```cpp
void f(seconds d)
{
    cout << d.count() << "s\n";
}
```
No implicit path from int to seconds!

```cpp
void f(seconds d)
{
    cout << d.count() << "s\n";
}

f(3);
// error: Not implicitly constructible from int
```

It is just as important what `seconds` won't do as what it `does` do!
No implicit path from int to seconds!

```cpp
void f(seconds d)
{
    cout << d.count() << "s\n";
}

f(3);
```

```
// error: Not implicitly constructible from int
```
seconds

No implicit path from int to seconds!

```cpp
void f(seconds d)
{
    cout << d.count() << "s\n";
}

f(3);
// error: Not implicitly constructible from int
f(seconds{3});  // ok, 3s
f(3s);          // ok, 3s  Requires C++14
seconds x{3};
f(x);           // ok, 3s
```
Addition and subtraction just like int:

```cpp
void f(seconds d)
{
    cout << d.count() << "s\n";
}
```
seconds

Addition and subtraction just like int:

```cpp
void f(seconds d)
{
    cout << d.count() << "s\n";
}

auto x = 3s;
x += 2s;
f(x);       // ok, 5s
x = x - 1s;
f(x);       // ok, 4s
f(x + 1);   // error: seconds + int not allowed
```
seconds

Comparison, all 6 operators, just like int:

```cpp
constexpr auto time_limit = 2s;

void f(seconds d)
{
    if (d <= time_limit)
    {
        cout << "in time: ";
    }
    else
    {
        cout << "out of time: ";
        cout << d.count() << "s\n";
    }
}
```
seconds

Comparison, all 6 operators, just like int:

define constexpr auto time_limit = 2;
define void f(seconds s)
{
    if (d <= time_limit)
        cout << "in time: ";
    else
        cout << "out of time: ";
    cout << d.count() << "s
";
}

error: seconds <= int not allowed
seconds

How much does this cost?!

seconds
f(seconds x, seconds y)
{
    return x + y;
}

int64_t
f(int64_t x, int64_t y)
{
    return x + y;
}

Compile both functions with optimizations on and inspect the assembly.
How much does this cost?!

Exactly the same object code generation (release configuration, except for name mangling).
What is the range?

seconds m = seconds::min();
seconds M = seconds::max();

You can query seconds for its range.

On every platform implemented this is +/- 292 billion years.

If you overflow, you've got issues.
seconds

So seconds is just a wrapper around an integral type, and acts just like an integral type (sans conversions to other integral types).

Is this such a big deal?!

Yes
What if suddenly you needed to transform your *million-line seconds-based* code to deal with *milliseconds*?
milliseconds

<chrono> also has a type called milliseconds...
millisseconds

And millisseconds works just like seconds:

class millisseconds
{
    int64_t ms_;  
public:
    millisseconds() = default;
    // etc.
    // ...
};

Except its range is only +/-292 million years.
milliseconds

So you just search and replace seconds for milliseconds?
So you just search and replace seconds for milliseconds?

No

It is much safer than that!
You can modify a small piece of code at time:

```cpp
void f(seconds d)
{
    cout << d.count() << "s\n";
}
```
milliseconds

You can modify a small piece of code at time:

```cpp
void f(milliseconds d)
{
    cout << d.count() << "ms\n";
}
```
millisecons

Clients either continue to work, or fail at compile time:

```cpp
t
void f(milliseconds d)
{
    cout << d.count() << "ms\n";
}
```
 Clients either continue to work, or fail **at compile time**: 

```cpp
void f(milliseconds d)
{
    cout << d.count() << "ms\n";
}

f(3);
// error: Not implicitly constructible from int
```
Clients either continue to work, or fail at compile time:

```cpp
void f(milliseconds d) {
    cout << d.count() << "ms\n";
}

f(3);
// error: Not implicitly constructible from int
f(seconds{3});  // ok, no change needed! 3000ms
f(3s);          // ok, no change needed! 3000ms
seconds x{3};
f(x);           // ok, no change needed! 3000ms
```
void f(seconds d)
{
    if (d <= time_limit)
        cout << "in time: ";
    else
        cout << "out of time: ";
    cout << d.count() << "s\n";
}
void f(milliseconds d)
{
    if (d <= time_limit)
        cout << "in time: ";
    else
        cout << "out of time: ";
    cout << d.count() << "ms\n";
}
Clients either continue to work, or fail at compile time:

```cpp
contexpr auto time_limit = 2s;

void f(milliseconds d)
{
    if (d <= time_limit)
        cout << "in time: ";
    else
        cout << "out of time: ";
    cout << d.count() << "ms\n";
}

f(3s);  // ok, no change needed! out of time: 3000ms
```
milliseconds

<chrono> knows about the relationship between milliseconds and seconds. It knows it has to multiply by 1000 to convert seconds to milliseconds.
milliseconds

<chrono> knows about the relationship between milliseconds and seconds. It knows it has to multiply by 1000 to convert seconds to milliseconds.

You should not manually code conversions from seconds to milliseconds. It is a simple computation. But it is easy to get wrong in one obscure place out of many in a million-line program.

Let <chrono> do this conversion for you. It does it in only one place, and is tested over many applications. And it is no less efficient than you could have coded manually.
milliseconds

How much does this cost?!

milliseconds
f(seconds x)
{
    return x;
}

int64_t
f(int64_t x)
{
    return x*1000;
}
milliseconds

How much does this cost?!

```
__Z1fNSt3__16chrono8durationIxNS...: ##
@_Z1fNSt3__16chrono8durationIxNS...
  .cfi_startproc
## BB#0:
  pushq %rbp
Ltmp0:
  .cfi_def_cfa_offset 16
Ltmp1:
  .cfi_offset %rbp, -16
  movq %rsp, %rbp
Ltmp2:
  .cfi_def_cfa_register %rbp
  imulq $1000, %rdi, %rax  ## imm = 0x3E8
  popq %rbp
  retq
  .cfi_endproc

__Z1fx:                                 ##
@_Z1fx
  .cfi_startproc
## BB#0:
  pushq %rbp
Ltmp0:
  .cfi_def_cfa_offset 16
Ltmp1:
  .cfi_offset %rbp, -16
  movq %rsp, %rbp
Ltmp2:
  .cfi_def_cfa_register %rbp
  imulq $1000, %rdi, %rax  ## imm = 0x3E8
  popq %rbp
  retq
  .cfi_endproc
```

Exactly the same object code generation (release configuration, except for name mangling).
milliseconds

<chrono> allows you to migrate from seconds to milliseconds a piece at a time. Code across such a transition will either be correct, or will not compile.
milliseconds

Even "mixed mode" arithmetic works just fine:

```cpp
auto x = 2s;
auto y = 3ms;
f(x + y); // 2003ms
f(y - x); // -1997ms
```
In General

If it compiles, it is working.

If it doesn't compile, don't escape the type system (using `count()` to fix it, unless you understand why it didn't work.

If you escape the type system and it compiles, all subsequent run time errors are on you.

I/O or interfacing with legacy code is the typical reason for needing to use `count()`.
What about converting milliseconds to seconds?

In general: If a `<chrono>` conversion is loss-less, then it is implicit.

If a conversion is not loss-less, it does not compile without special syntax.

Example:

```cpp
seconds x = 3400ms; // error: no conversion
```
What about converting milliseconds to seconds?

In general: If a `<chrono>` conversion is *loss-less*, then it is implicit.

If a conversion is not loss-less, it does not compile without special syntax.

Example:

```cpp
seconds x = 3400ms; // error: no conversion
seconds x = duration_cast<seconds>(3400ms); // 3s
```

duration_cast means: convert with truncation towards zero.
What about converting milliseconds to seconds?

duration_cast<duration> truncates towards zero.

In C++1z (hopefully C++17)

floor<duration> truncates towards negative infinity.
round<duration> truncates towards nearest and towards even on a tie.
ceil<duration> truncates towards positive infinity.
What about converting milliseconds to seconds?

Only use an explicit cast when an implicit conversion won't work.

If the implicit conversion compiles, it will be exact.

Otherwise it won't compile and you can make the decision of which rounding mode you need (towards zero, towards infinity, towards negative infinity, towards nearest).
Wait, there's more...

hours
minutes
seconds
milliseconds
microseconds
nanoseconds

Everything I've said about seconds and milliseconds is also true for all of these other units.
All of these units work together seamlessly

hours
minutes
seconds
milliseconds
microseconds
nanoseconds

```cpp
void f(nanoseconds d)
{
    cout << d.count() << "ns\n";
}

auto x = 2h;
auto y = 3us;
f(x + y);  // 7200000003000ns
```
This is overkill for my application

I'm building a TRS-80 emulator and all I need is a 32-bit second.

<std::chrono> still has you covered.

using seconds32 = std::chrono::duration<int32_t>;

seconds32 will interoperate with the entire <chrono> library as described for std::chrono::seconds, but use int32_t as the "representation".
This is overkill for my application

I'm building a TRS-80 emulator and all I need is a 32-bit second.

I meant unsigned 32 bits.

whatever...

using seconds32 = std::chrono::duration<uint32_t>;
This is overkill for my application

I'm building a TRS-80 emulator and all I need is a 32-bit second.

I meant unsigned 32 bits. And I need overflow protection.

Find (or build) a "safeint" library that does what you want, and then:

```cpp
using seconds32 = duration<safe<uint32_t>>;
```
Generalized Representation

In general, you can plug any arithmetic type, or emulation thereof, into `duration<Rep>` and you will get a type that means seconds, using that representation.

Yes, even floating point types.
Generalized Representation

For floating-point representations, you can implicitly convert from any precision without using `duration_cast`. The rationale is that there is no truncation error (only rounding error). And so implicit conversion is safe.

```cpp
using fseconds = duration<float>;

void f(fseconds d)
{
    cout << d.count() << "s\n";
}

f(45ms + 63us);  // 0.045063s
```
Generalized Representation

Can I do generalized representation with milliseconds?

template <class T>
  using my_ms = std::chrono::duration<T, std::milli>;

void f(my_ms<float> d)
{
  cout << d.count() << "ms\n";
}

f(45ms + 63us);  // 45.063ms
Generalized Representation

The standard specifies:

```cpp
using nanoseconds = duration<int_least64_t, nano>;
using microseconds = duration<int_least55_t, micro>;
using milliseconds = duration<int_least45_t, milli>;
using seconds = duration<int_least35_t>;
```
Generalized Representation

The standard specifies:

```cpp
using nano = ratio<1, 1'000'000'000>;
using micro = ratio<1, 1'000'000>;
using milli = ratio<1, 1'000>;
```

Where `ratio<N, D>` is:

```cpp
template <intmax_t N, intmax_t D = 1>
class ratio
{
    static constexpr intmax_t num; // N/D reduced to
    static constexpr intmax_t den; // lowest terms
    using type = ratio<num, den>;
};
```
Generalized Representation

The standard specifies:

```cpp
using nanoseconds = duration<int_least64_t, nano>;
using microseconds = duration<int_least55_t, micro>;
using milliseconds = duration<int_least45_t, milli>;
using seconds = duration<int_least35_t, ratio<1>>;
using minutes = duration<int_least29_t, ratio<60>>;
using hours = duration<int_least23_t, ratio<3600>>;
```

```cpp
template <class Rep, class Period = ratio<1>>
class duration {
  Rep rep_;
public:
};
```
Durations

template <class Rep, class Period = ratio<1>>
class duration {
public:
    using rep = Rep;
    using period = Period;
    // ...
};

Every duration has a nested type rep, which is its representation, and a nested type period which is a fraction representing the period of the duration's "tick" in units of seconds.

milliseconds::rep is int64_t
milliseconds::period::num is 1
milliseconds::period::den is 1000
Generalized Duration Unit

You can build any duration that meets your needs:

```cpp
using frames = duration<int32_t, ratio<1, 60>>;

void f(duration<float, milli> d);
```

And things will just work, following all of the previously outlined rules:

```cpp
f(frames{1});       // 16.6667ms
f(45ms + frames{5}); // 128.333ms
```
Deep Dive

45ms + frames{5}

pseudo syntax:

45 [int64_t, 1/1000] + 5 [int32_t, 1/60]

Everything inside [] is always computed at compile time.
Deep Dive

45ms + frames{5}

pseudo syntax:

45 \texttt{[int64\_t, 1/1000]} + 5 \texttt{[int32\_t, 1/60]}

Find common\_type and convert to it:

45*3 \texttt{[int64\_t, 1/3000]} + 5*50 \texttt{[int64\_t, 1/3000]}

Computed least common multiple \textit{at compile time}!
Deep Dive

45ms + frames{5}

pseudo syntax:

\[ 45 \text{ [int64_t, 1/1000]} + 5 \text{ [int32_t, 1/60]} \]

Find common_type and convert to it:

\[ 45 \times 3 \text{ [int64_t, 1/3000]} + 5 \times 50 \text{ [int64_t, 1/3000]} \]

Do arithmetic in common_type:

\[ 385 \text{ [int64_t, 1/3000]} \]
Deep Dive

45ms + frames{5}

pseudo syntax:

45 [int64_t, 1/1000] + 5 [int32_t, 1/60]

Find common_type and convert to it:

45*3 [int64_t, 1/3000] + 5*50 [int64_t, 1/3000]

Do arithmetic in common_type:

385 [int64_t, 1/3000]

Convert to duration<float, milli>:

128.333 [float, 1/1000]
All of the complicated work is done at compile time.
I know it is a lot

Feel like you've been drinking from the fire hose?

Wait, there's more...
But before we go on

Recall back in the beginning when there were just seconds, and then maybe milliseconds are introduced?

All of this fancy stuff about frames, and nanoseconds, and floating point milliseconds, and 32 bit representations...

It is all there only in case you need it. You don't pay for it if you don't use it. This whole shebang is still just as simple as a wrapper around a int64_t which means seconds.

Simple. Only as complicated as you need it to be.
But before we go on

Simple. Only as complicated as you need it to be.

`seconds` is to `duration<int64_t, ratio<1, 1>>`

as

`string` is to `basic_string<char, char_traits<char>, allocator<char>>`

You can just use it without worrying about the fact that it is a specialization of a template.
But before we go on

Simple. Only as complicated as you need it to be.
And type-safe.

This library lives and dies by converting one type to another.

If the conversion is loss-less (seconds to milliseconds), it can be made implicitly.

If the conversion is lossy (milliseconds to seconds) it can be made with duration_cast.

If the conversion is dangerous, it must be made with explicit conversion syntax (int to seconds or seconds to int).
But before we go on

Simple. Only as complicated as you need it to be. And type-safe.

If you make a reasonable change that doesn't involve explicit type conversion syntax (and it compiles), you can have confidence that you have not introduced a bug.

Use the weakest type conversion possible:

• Implicit if at all possible.
• `duration_cast` if you need to specify truncation.
• `.count()` in a Kobayashi Maru.
What We Will Be Talking About

• Motivation. Why `<chrono>`?

• Time durations

  • Points in time

• Clocks

• Examples
So far we've only talked about time *durations*.

Relax.

Your knowledge of durations will carry over to time *points*.

There is not that much more to learn.
A duration such as 10'000s means *any* 10,000s. Or if you prefer 2h + 46min + 40s.

But:

time_point<system_clock, seconds> tp{10'000s};

Means:

1970-01-01 02:46:40 UTC

(Not specified, but de facto standard)
A `time_point` refers to a specific point in time, with respect to some clock, and has a precision of some duration:

```cpp
template <class Clock,
    class Duration = typename Clock::duration>
class time_point {
    Duration d_;  
public:
    using clock    = Clock;  
    using duration = Duration;  
    // ...
};
```
time_point

template <class Clock,
   class Duration = typename Clock::duration>
class time_point {
   Duration d_;  
public:
   using clock    = Clock;
   using duration = Duration;
   // ...
};

time_points and durations can have the exact same representation, but they mean different things.
**time_point**

When it comes to arithmetic, `time_point` s are similar to pointers: `time_point` s can be subtracted, but not added. Their difference is not another `time_point` but rather a `duration`.

```cpp
auto d = tp1 - tp2;
```

You can add/subtract a `duration` to/from a `time_point`, resulting in another `time_point`.

```cpp
auto tp2 = tp1 + d;
```

It is a 100% self-consistent algebra, type-checked **at compile-time**.
time_point

time_points convert much like the durations do:

Implicitly when the conversion does not involve truncation error.

using namespace std::chrono;
template <class D>
    using sys_time = time_point<system_clock, D>;
sys_time<seconds> tp{5s};    // 5s
sys_time<milliseconds> tp2 = tp; // 5000ms
time_point

time_points convert much like the durations do:

Implicitly when the conversion does not involve truncation error.

With time_point_cast when you want to force a truncation error.

using namespace std::chrono;
template <class D>
    using sys_time = time_point<system_clock, D>;
sys_time<seconds> tp{5s};        // 5s
sys_time<milliseconds> tp2 = tp;   // 5000ms
tp = time_point_cast<seconds>(tp2);  // 5s
time_point

time_points convert much like the durations do:

Implicitly when the conversion does not involve truncation error.

With `time_point_cast` when you want to force a truncation error.

Explicitly when you want to force a `duration` to `time_point` conversion.

With `.time_since_epoch()` when you want to force a `time_point` to `duration` conversion.
What We Will Be Talking About

• Motivation. Why `<chrono>`?
• Time durations
• Points in time
  • Clocks
• Examples
A clock is a bundle of a duration, a time_point and a static function to get the current time.

```cpp
struct some_clock
{
  using duration   = chrono::duration<int64_t, microseconds>;
  using rep        = duration::rep;
  using period     = duration::period;
  using time_point = chrono::time_point<some_clock>;
  static constexpr bool is_steady = false;

  static time_point now() noexcept;
};
```
Clocks

Every `time_point` is associated with a clock.

`time_points` associated with different clocks do not convert to one another.

```cpp
system_clock::time_point tp = system_clock::now();
steady_clock::time_point tp2 = tp;
```

Different clocks

`error: no viable conversion`
Every time_point is associated with a clock.

time_points associated with different clocks do not convert to one another.

Applications can have as many different clocks as they want to.

There are two useful std-supplied clocks:

```cpp
std::chrono::system_clock
std::chrono::steady_clock
```

Ignore `std::chrono::high_resolution_clock` as it is a type alias for one of the above clocks.
clocks

std::chrono::system_clock

Use system_clock when you need time_points that must relate to some calendar.

system_clock can tell you what time of day it is, and what the date is.
Use `std::chrono::steady_clock` when just need a stopwatch.

It is good for timing, but can not give you the time of day.
Whatever clock you use, you can get its time_point like this:

```cpp
clock::time_point tp
```
clocks

And you can get the current time like this:

clock::time_point tp = clock::now();
clocks

Or like this:

    auto tp = clock::now();
What We Will Be Talking About

• Motivation. Why <chrono>?
• Time durations
• Points in time
• Clocks

• Examples
Examples:

Timing:

```cpp
auto t0 = steady_clock::now();
f();
auto t1 = steady_clock::now();
cout << nanoseconds{t1-t0}.count() << "ns\n";
```

Output:

135169457ns
Examples:

Timing:

```cpp
auto t0 = steady_clock::now();
f();
auto t1 = steady_clock::now();
cout << duration<double>{t1-t0}.count() << "s\n";
```

Output:

0.135169s
Examples:

Timing:

```cpp
auto t0 = steady_clock::now();
f();
auto t1 = steady_clock::now();
cout << duration_cast<milliseconds>(t1-t0).count() << "ms\n";
```

Output:

135ms
Examples:

mutex timed try lock:

```cpp
std::timed_mutex mut;
if (mut.try_lock_for(500ms))
    // got the lock
if (mut.try_lock_until(steady_clock::now() + 500ms))
    // got the lock
```
Examples:

Custom duration:

```cpp
using days = duration<int, ratio_multiply<ratio<24>, hours::period>>;

using days = duration<int, ratio<86400>>;  // same thing
```
Examples:

Sleep with custom duration:

```cpp
std::this_thread::sleep_for(days{1});
```
Examples:

Sleep with custom duration:

```cpp
std::this_thread::sleep_for(days{1});
```

```cpp
using weeks = duration<int, ratio_multiply<ratio<7>,
                        days::period>>;
```

```cpp
std::this_thread::sleep_for(weeks{2});
```
Examples:

Time since epoch
(de facto standard 1970-01-01 00:00:00 UTC).

```cpp
auto tp = time_point_cast<seconds>(system_clock::now());
cout << tp.time_since_epoch().count() << "s\n";
1469456123s
```
Examples:

Time since epoch
(de facto standard 1970-01-01 00:00:00 UTC).

```cpp
auto tp = time_point_cast<seconds>(system_clock::now());
cout << tp.time_since_epoch().count() << " s
";
  1469456123s
auto td = time_point_cast<days>(tp);
cout << td.time_since_epoch().count() << " days
";
  17007 days
```
Summary

<chrono>

• duration
• time_point
• clock
Summary

<chrono>

• duration
• time_point
• clock

• Compile-time errors are favored over run-time errors.
• As efficient as hand-written code (or better).
• Feature rich, but you don't pay for features you don't use.
Summary

<chrono>

• duration
• time_point
• clock

• Designed a decade ago.
• Voted into C++11 in 2008.
  • Standard C++ for half a decade now.
• This is not bleeding edge, it is best practice.
Summary

<chrono>

- duration
- time_point
- clock

- Designed a decade ago.
- Voted into C++11 in 2008.
  - Standard C++ for half a decade now.
- This is not bleeding edge, it is best practice.
- Bleeding edge in time computations is my talk tomorrow at 4:45pm which builds upon (not obsoletes) <chrono>...