C++ GEMS
CHRONO & RATIO

Jeff Garland

C++Now 2012

ALTERNATIVE TITLES

How boost.date_time inspired C++11 to handle time better

Why C++11 is the awesomest language to write timed threading code

PART 1: CHRONO

MOTIVATION – A HORROR STORY

Once upon a time timing was needed for boost thread…and there was xtime

And the sacred docs said:
“An object of type xtime defines a time…”

“This is a temporary solution that will be replaced by a more robust time library once available in Boost.”

Temporary almost became 8

MOTIVATION – A HORROR STORY

What is xtime…

```
struct xtime {
        platform-specific-type sec;
};

int xtime_get(xtime*, int);
```

MOTIVATION – A HORROR STORY

It’s not just xtime – it’s C & Posix

```
struct timespec ts;

/* Delay for a bit */
ts.tv_sec = 2;
ts.tv_nsec = 1030;
nanosleep (&ts, NULL);
```

Isn’t every bit precious?

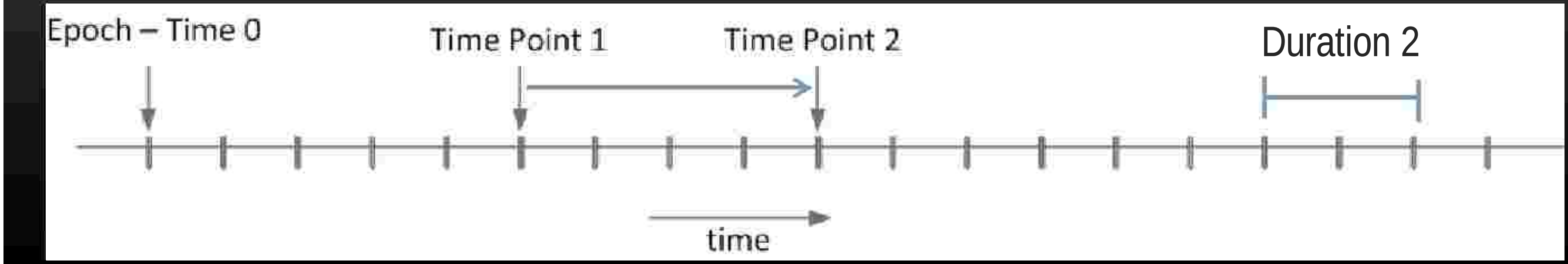
How long is this sleep exactly?

Need to do math on these? Good luck it’s ugly…

Comparison is inefficient

Not awesome – we can do better!

“TIME” FOR A LITTLE THEORY



Time Point – a location in the time continuum

Handy for saying ‘at exactly this time’

Epoch – the anchor point for the counted representation

Durations – a length of time

Handy for saying ‘10 seconds from now’

A count (with a resolution)

10 seconds, 20 milliseconds, etc

Clocks

Tell us the current time point

At a certain resolution…

THREADING INTERFACES WITH TIMES

Timed Locking:

```
bool try_lock_for(   const duration&     relative_time);
bool try_lock_until( const time_point& absolute_time);
```

Condition Variable Methods:

```
bool wait_until(unique_lock<mutex>& lock,
                const time_point& absolute_time,
                Predicate p)

cv_status wait_for(unique_lock<mutex>& lock,
                   const duration& relative_time)
```

MORE THREADING INTERFACES WITH TIMES

Sleeping:

```
void sleep_for (  const  duration&     relative_time);
void sleep_until( const  time_point& absolute_time);
```

std::future, std::shared_future

```
bool wait_for(const duration& relative_time) const
bool wait_until(const time_point& absolute_time) const
```

Last 2 slides – many lies….

In the real world things are a bit more complex

More later…

“TIME” FOR SOME REST – CERTIFIED C++11

Sleeping:

```
using namspace std::chrono;
std::this_thread::sleep_for( milliseconds(100) );
std::this_thread::sleep_for( seconds(2) );


//when c++ crushes java…how long is that exactly?

java.lang.Thread.currentThread().sleep(10000);
```

C++11 is the awesomest

DURATIONS ARE COOL

```
using namespace std::chrono;

microseconds d1(1000);
seconds        d2(1);

d1 += seconds(30);      ← Convert seconds to microseconds
d1 += milliseconds(1) – microseconds(20);
d2++; d1--;
d2*=10;

if (d1 > d2) {…}   //the usual comparisons


std::cout << d1.count() << std::endl;
```

DURATIONS INTERFACE

An arithmetic value type (more later)

Expected Comparison operators

Observers

```
constexpr rep_type count() const
```

Traits

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

DURATIONS – CONSTRUCT/COPY/DESTROY

```
constexpr duration() = default;                    //duration 0
~duration() = default;

duration(const duration&) = default;

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


template <class Rep2, class Period2>

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

DURATIONS INTERFACE - ARITHMETIC

```
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);
```

LET’S SLEEP TILL AN ABSOLUTE TIME

```
system_clock::time_point tp = system_clock::now();

tp += milliseconds(20);

std::this_thread::sleep_until( tp );
```

CLOCK INTERFACE

Clock is a bundle consisting of a duration, a time_point, and a function now() to get the current time

Construction

- None
- static time_point now()

Declared Types

- time_point
- duration

TIME POINTS AND DURATIONS PLAY NICE

```
system_clock::time_point tp;

system_clock::time_point tp2 = tp + seconds(2) + milliseconds(20);
```

duration + time point ? time point duration + duration ? duration

```
nanoseconds ms = tp2 - tp;
```

time point - time point ? duration

```
milliseconds ms = tp2 - tp;
```

← Compiler error – not enough resolution

C++11 is the awesomest

TIME POINT INTERFACE

Construction

```
default     //constructs to clock epoch
call now() on a clock
```

Conversion

```
time_t to_time_t()
duration time_since_epoch()
```

Arithmetic

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

PROBLEMS AND COMPLICATIONS

Clocks are not all created equal…

- Resolution of clock depends on machine
- Machines are changing
- Typically millisecond resolution

How can the C++ standard specify reasonably?

What we want:

- Code that can be portable as possible
- Code that can take full advantage of a platform
- Code that doesn’t have to change as clocks improve
- Code that ‘just works’

Answer - templates of course!

TRY LOCK INTERFACE – THE REAL DEAL

```
template<typename Rep,typename Period>
bool
try_lock_for( std::chrono::duration<Rep,Period> const& relative_time);

template<typename Clock,typename Duration>
bool
try_lock_until( std::chrono::time_point<Clock,Duration> const& absolute_time);
```

The generic interfaces allow for custom clocks to be added – same interface

THREE STANDARD CLOCKS

system_clock

- Represent wall clock time from the system-wide realtime clock
- typically this will be implemented via gettimeofday()
- clock can be adjusted – possibly backward
- user sets time, NTP adjust

steady_clock

- values of time_point never decrease as physical time advances
- values of time_point advance at a steady rate relative to real time
- clock cannot be adjusted

high_resolution_clock

- Clock with the shortest tick period.
- may be a synonym for system_clock or steady_clock.

Beware – platforms will be different – your mileage may vary

PART 2: UNDER THE HOOD -- RATIO

WHERE THE MAGIC HAPPENS

```
//chrono header
    typedef duration<int64_t, nano>          nanoseconds;
    typedef duration<int64_t, micro>         microseconds;
    typedef duration<int64_t, milli>         milliseconds;
    typedef duration<int64_t>                seconds;
    typedef duration<int, ratio< 60>>        minutes;
    typedef duration<int, ratio<3600>>       hours;
```

What is ‘nano’ and what is ratio<3600> doing?

RATIO THE BASICS

Compile time rational numbers

template ratio<N, M>

Math functions that go with

add, subtract, multiply, divide

The magic behind duration to duration conversion

RATIO EXAMPLE

Example – duration unit conversions

1 second is fundamental unit of measure

There are 1000 milliseconds in a second

Milliseconds to seconds -- divide by 1000

```
//abbreviated list from g++ ratio header
 typedef ratio<1,              1000000000> nano;
 typedef ratio<1,                 1000000> micro;
 typedef ratio<1,                    1000> milli;
 typedef ratio<1,                     100> centi;
 typedef ratio<1,                      10> deci;
 typedef ratio<                    10, 1> deca;
 typedef ratio<                   100, 1> hecto;
 typedef ratio<                  1000, 1> kilo;
...
```

CONVERSIONS USING RATIO

```
system_clock::time_point tp;

system_clock::time_point tp2 = tp + seconds(2) + milliseconds(20);
```

Duration added to time point

Duration added to Duration

```
nanoseconds ms = tp2 - tp;
```

Duration subtracted from time point

```
milliseconds ms = tp2 - tp;
```

Compiler error – not enough resolution

CONVERSION FROM SECONDS TO HOURS

1 second is fundamental unit of measure

There are 3600 seconds in a hour

Multiply seconds by 3600

ratio<3600, 1> or shorter version ratio<3600>

CUSTOM DURATIONS – EASY AS PIE

What if I need to deal in other time lengths?

Say 1/2 of a second is important unit

```
typedef std::ratio<1,2> half;
typedef std::chrono::duration<int64_t, half> half_seconds;
```

half_seconds is now useable in all thread/sleep APIS

half_seconds ‘just works’ with all the other durations

CUSTOM DURATIONS IN ACTION

```
half_seconds hs = seconds(10);
std::cout << hs.count() << std::endl; //20
```

```
seconds s = half_seconds(3);
```

error: conversion from ‘half_seconds’ to non-scalar type ‘std::chrono::seconds’ requested

```
seconds s = duration_cast<seconds>(half_seconds(3));
std::cout << s.count() << std::endl; //1
```

PART 3: FINAL THOUGHTS

NOT ALL SWEETNESS AND LIGHT

```
#include <chrono>
#include <iostream>
…
  using namespace std::chrono;

  system_clock::time_point tp = system_clock::now();

  std::cout << tp << std::endl;
```

Compiler error – no operator

BOOST TO THE RESCUE

```
#include <boost/date_time.hpp>

  using namespace std::chrono;
  using namespace boost::posix_time;

  system_clock::time_point tp = system_clock::now();

  ptime tp2(from_time_t(system_clock::to_time_t(tp)));

  std::cout << tp2 << std::endl; // YYYY-MM-DD HH:MM:SS
```

CHRONO VS BOOST DATE.TIME – WHAT NEXT?

Boost date.time needs to be re-written for c++11

Should adopt the duration types from c++11
Should adopt the time_point abstractions (almost)
Should adopt the clocks from chrono

From there – it’s more complicated
ptime stands alone from clocks

Can’t promise when this will happen….

FINAL THOUGHTS

No more excuses – only elegant time code in C++11!

Study the standard library

- powerful tools under the hood

g++4.6 – all examples compiled there – looks good

Thanks to Howard

C++11 is the awesomest!