A C++14 Approach to Dates and Times

Howard Hinnant

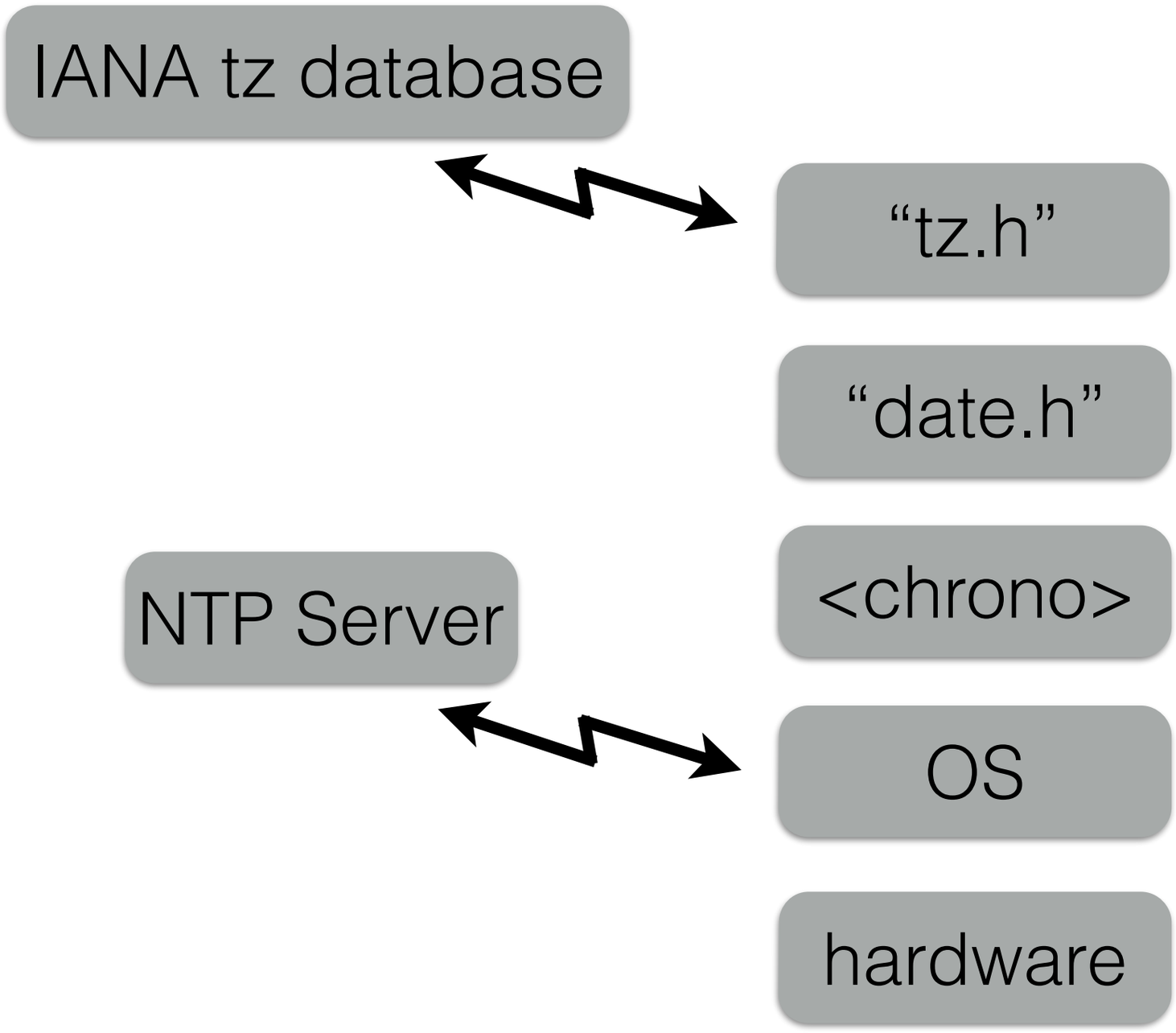
cppcon 2015

sep/25/2015

Big Picture

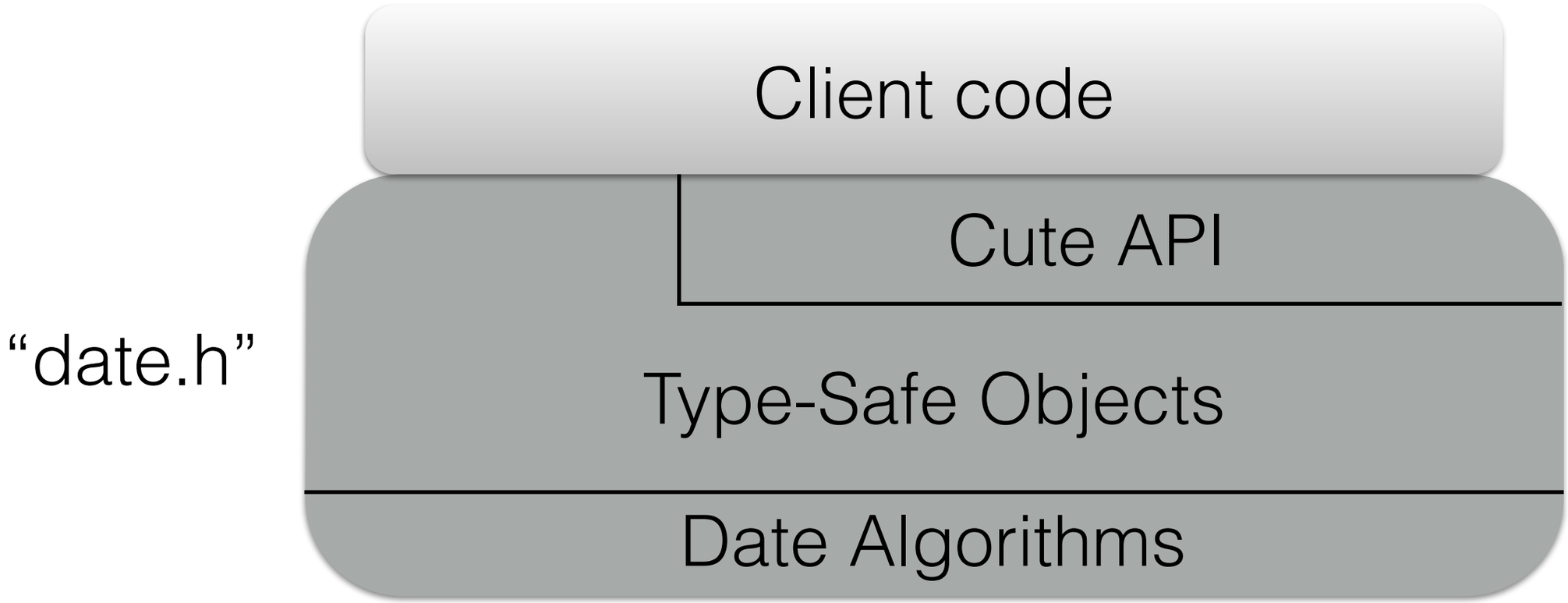
- This date library is a seamless extension of the existing <chrono> library into the realm of calendars.
- It is minimalistic — a single header.
- It won't do everything you want.
- It provides *efficient* building blocks so you can easily do for yourself everything you want.

Where this library fits



This talk concentrates here.

Where this library fits



- Clients can write to either the "cute API" or to the lower level type-safe object API.
- The algorithms are completely encapsulated within the type-safe objects and their conversions to one another.

Date Algorithms

For example:

- Convert {year, month, day} triple into a serial count of days.

```
constexpr
int
days_from_civil(int y, unsigned m, unsigned d) noexcept
{
  y -= m <= 2;
  const Int era = (y >= 0 ? y : y-399) / 400;
  const unsigned yoe = static_cast<unsigned>(y - era * 400);
  const unsigned doy = (153*(m + (m > 2 ? -3 : 9)) + 2)/5 + d-1;
  const unsigned doe = yoe * 365 + yoe/4 - yoe/100 + doy;
  return era * 146097 + static_cast<Int>(doe) - 719468;
}
```

Date Algorithms

For example:

- Convert a serial count of days into a {year, month, day} triple.

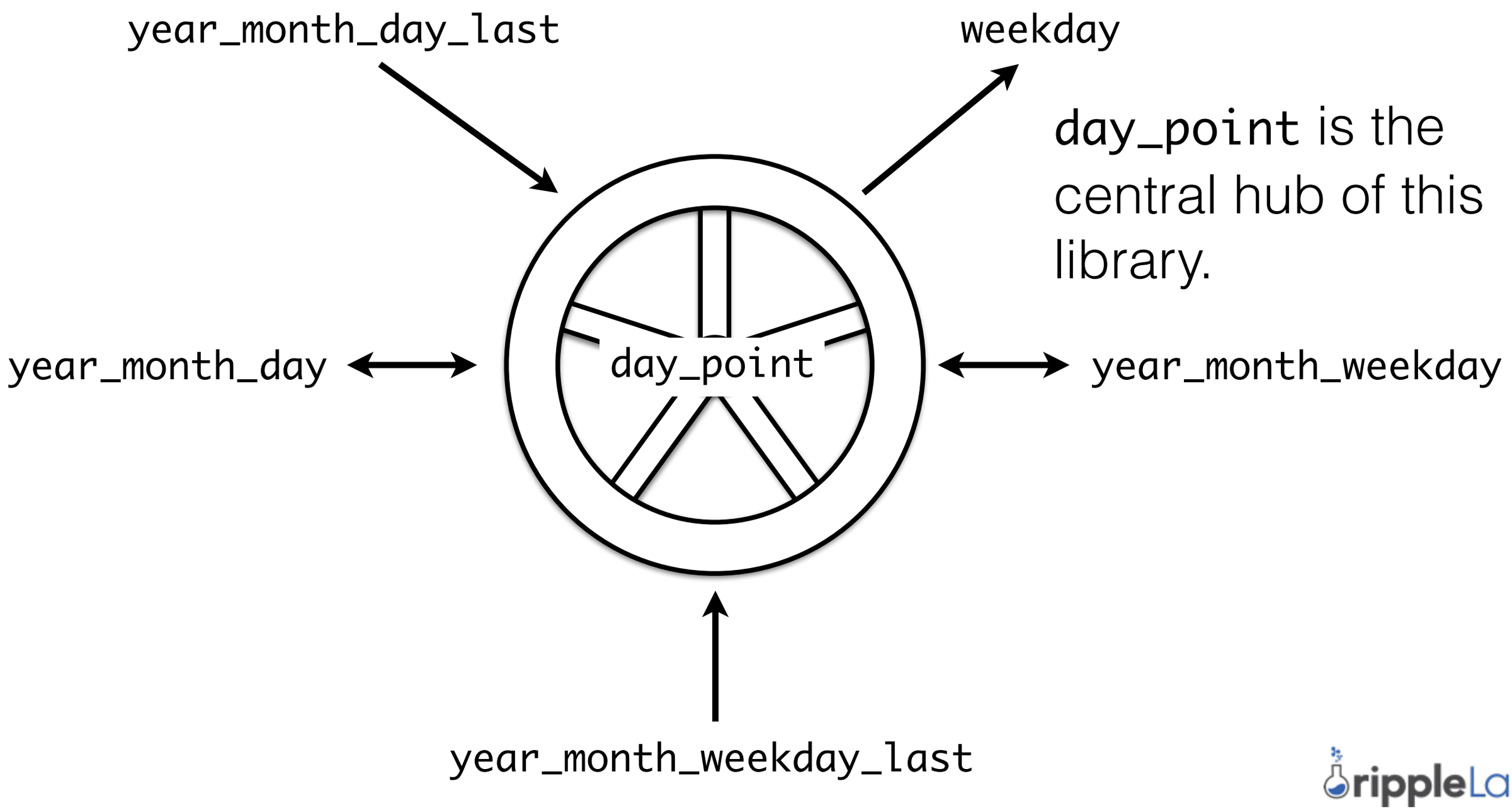
```
constexpr
std::tuple<int, unsigned, unsigned>
civil_from_days(int z) noexcept
{
  z += 719468;
  const Int era = (z >= 0 ? z : z - 146096) / 146097;
  const unsigned doe = static_cast<unsigned>(z - era * 146097);
  const unsigned yoe = (doe - doe/1460 + doe/36524 - doe/146096) / 365;
  const Int y = static_cast<Int>(yoe) + era * 400;
  const unsigned doy = doe - (365*yoe + yoe/4 - yoe/100);
  const unsigned mp = (5*doy + 2)/153;
  const unsigned d = doy - (153*mp+2)/5 + 1;
  const unsigned m = mp + (mp < 10 ? 3 : -9);
  return std::tuple<Int, unsigned, unsigned>(y + (m <= 2), m, d);
}
```

Date Algorithms

- Every date algorithm has been unit tested for every single day over a range of +/- a million years.
- That is, way more than need be.

http://howardhinnant.github.io/date_algorithms.html

Type-Safe Objects



`day_point` is the central hub of this library.

rippleLabs

Type-Safe Objects

Not picking the right data structure is the most common reason for performance problems.

— *Alexander Stepanov*

This library offers you a selection of data structures for dates, and encourages you to build more of your own.

```
year_month_day
{
   year  y;
   month m;
   day   d;
};
```

```
day_point
{
  days count;
};
```

Type-Safe Objects

- Type conversions execute date algorithms.

One Date Algorithm

Another Date Algorithm

Cute API

- A set of overloaded division operators that compose field specifiers into field-based types.

```
year / month / day  ⟶  year_month_day
```

Cute API

- A set of overloaded division operators that compose field specifiers into field-based types.
- Constants and literals aid when field components are known at code-writing time.

```
sun[last]/mar/2015  ⟶  year_month_weekday_last
```

Cute API

- A set of overloaded division operators that compose field specifiers into field-based types.
- Constants and literals aid when field components are known at code-writing time.
- Traditional constructor syntax is always available.

```
sun[last]/mar/2015  ⟶  year_month_weekday_last


year_month_weekday_last(year(2015),
                        month(3),
                        weekday_last(weekday(0)))
```

year_month_day

A field type

- Construction:

```
auto ymd = 2015_y/sep/25;
```

- Observable fields:

```
assert(ymd.year() == 2015_y);
assert(ymd.month() == sep);
assert(ymd.day() == 25_d);
```

year_month_day

A field type

- Construction:

```
constexpr auto ymd = 2015_y/sep/25;
```

- Observable fields:

```
static_assert(ymd.year() == 2015_y, “”);
static_assert(ymd.month() == sep, “”);
static_assert(ymd.day() == 25_d, “”);
```

- Everything is available at compile time (if you like).

year_month_day

A field type

- Construction:

```
constexpr auto ymd = 2015_y/sep/25;
```

- Year and month arithmetic:

```
auto next_month = ymd + months{1};
auto last_year  = ymd - years{1};
```

- Use `day_point` for day-oriented arithmetic.

year_month_day

A field type

- Construction:

```
constexpr auto ymd = 2015_y/sep/25;
```

- Simple, readable streaming:

```
cout << ymd << '\n';  // 2015-09-25
```

year_month_day

A field type

- Each field can be *explicitly* converted to and from integral types to enable custom I/O.

```
int y, m, d;
cin >> y >> m >> d;
auto ymd = year(y)/month(m)/day(d);
```

year_month_day

A field type

- Each field can be explicitly converted to and from integral types to enable custom I/O.

```
int y, m, d;
cin >> y >> m >> d;
auto ymd = year(y)/m/d;
```

- Only the first field needs to be explicit.

year_month_day

A field type

- Each field can be explicitly converted to and from integral types to enable custom I/O.

```
int y, m, d;
cin >> y >> m >> d;
auto ymd = day(d)/m/y;
```

- day/month/year order is also ok.

year_month_day

A field type

- Each field can be explicitly converted to and from integral types to enable custom I/O.

```
int y, m, d;
cin >> y >> m >> d;
auto ymd = month(m)/d/y;
```

- month/day/year order is also ok.
- As long as the first field is specified, the rest is unambiguous.

year_month_day

A field type

- Each field can be explicitly converted to and from integral types to enable custom I/O.

```
int y, m, d;
cin >> y >> m >> d;
auto ymd = y/month(m)/d;
                ~^~~~~~~~~
error: invalid operands to binary expression
('int' and 'date::month')
```

- Anything ambiguous or invalid is caught at compile time.

Handling Invalid Dates

- What is this: `2015_y/jan/31 + months{1}`
- Possibilities:

`assert`

`throw something`

`2015_y/feb/28`

`2015_y/mar/3`

- I've seen all of these solutions and they are all valid. This library chose none of them, forcing you to decide.

Handling Invalid Dates

- What is this: `2015_y/jan/31 + months{1}`

  `2015_y/feb/31`

- I've seen all of these solutions and they are all valid. This library chose none of them, forcing you to decide.
- This library allows you to create invalid dates, and then gives you the ability to check for invalid dates and decide what to do with them.
- This leads to high-performance low-level code upon which you can optimally build in checks exactly where you need them.

rippleLabs

Handling Invalid Dates

assert

```
ymd += m;                // 2015_y/feb/31
assert(ymd.ok());  // Ensure the result is valid!
```

Every field type comes with an `ok()` member function.

Handling Invalid Dates

throw something

```
auto result = ymd + m; // 2015_y/feb/31
if (!result.ok())
{
    std::ostringstream os;
    os << ymd << " + " << m.count()
       << " months results in " << result;
    throw std::domain_error(os.str());
}
```

Handling Invalid Dates

2015_y/feb/28

```
ymd += m;                  // 2015_y/feb/31
if (!ymd.ok())
    ymd = ymd.year()/ymd.month()/last;
// ymd == 2015_y/feb/28
```

Handling Invalid Dates

2015_y/mar/3

```
ymd += m;               // 2015_y/feb/31
if (!ymd.ok())
    ymd = day_point{ymd};
// ymd == 2015_y/mar/3
```

Handling Invalid Dates

- Errors are either caught at compile-time (e.g. invalid/ambiguous ordering), or can be detected at run-time with `ok()`.
- The library does not check `ok()` internally.
  - You decide where and how often to check `ok()`.
- The library does not throw exceptions.
  - This library is exception safe, so you can throw exceptions whenever you want to.

last

- When ever and where ever it is legal to specify a `day`, it is also legal to specify `last`.

```
auto ymd = 2015_y/feb/last;
auto ymd = feb/last/2015;
auto ymd = last/feb/2015;
```

- This represents the last day of the year/month combination.
- The expression has type `year_month_day_last`.
- The API of `year_month_day_last` is virtually identical to that of `year_month_day`.

last

- When ever and where ever it is legal to specify a `day`, it is also legal to specify `last`.

```
ymd += m;                       // 2015_y/feb/31
if (!ymd.ok())
    ymd = ymd.year()/ymd.month()/last;
```

- This represents the last day of the year/month combination.
- The expression has type `year_month_day_last`.
- The API of `year_month_day_last` is virtually identical to that of `year_month_day`.
- `year_month_day_last` is even implicitly convertible to `year_month_day`.

Indexed weekdays

- When ever and where ever it is legal to specify a `day`, it is also legal to specify an *indexed weekday*.

```
auto ymwd = 2015_y/sep/fri[4];
auto ymwd = sep/fri[4]/2015;
auto ymwd = fri[4]/sep/2015;
```

- The fourth Friday of September of 2015.
  - The type is `year_month_weekday`.
- Change field type to `year_month_day`:

```
cout << ymwd << '\n';          2015/Sep/Fri[4]
year_month_day ymd{ymwd};
cout << ymd << '\n';           2015-09-25
```

Indexed weekdays

- You can also index `weekday` with `last`:

```
auto ymwd = 2015_y/sep/fri[last];
auto ymwd = sep/fri[last]/2015;
auto ymwd = fri[last]/sep/2015;
```

- The last Friday of September of 2015.
  - The type is `year_month_weekday_last`.
- Change field type to `year_month_day`:

```
cout << ymwd << '\n';          2015/Sep/Fri[last]
year_month_day ymd{ymwd};
cout << ymd << '\n';           2015-09-25
```

day_point

- `day_point` is a serial-based `time_point` with a resolution of a day.
- `day_point` is a `std::chrono::time_point`.
  - `day_point` is a simple count of days since the `std::chrono::system_clock` epoch.

```
using day_point =
    std::chrono::time_point<std::chrono::system_clock, days>;
```

- `day_point` is the central theme of this library, and is nothing more than type-alias for a type in `<chrono>`.

day_point

```
day_point dp = sep/25/2015;
cout << dp.time_since_epoch().count() << '\n';
```

```
16703
```

- It has been 16,703 days since 1970-01-01.
- day-oriented arithmetic is very efficient on this type!

```
dp += days{2};
cout << dp.time_since_epoch().count() << '\n';
```

```
16705
```

Converting system_clock::time_point to day_point

- `day_point` *is* a `time_point` (with a coarse duration).
- `time_point`s can be cast with `time_point_cast`:

```
day_point dp = time_point_cast<days>(system_clock::now());
```

- `time_point_cast` truncates towards zero.
  - This gives unexpected results for dates prior to 1970 (negative time_points).

Converting system_clock::time_point to day_point

- `day_point` *is* a `time_point` (with a coarse duration).
- `time_point`s can be cast with `time_point_cast`:

```
day_point dp = floor<days>(system_clock::now());
```

- `time_point_cast` truncates towards zero.
  - This gives unexpected results for dates prior to 1970 (negative time_points).
- `floor` is just like `time_point_cast`, except that it rounds towards negative infinity.

Converting system_clock::time_point to day_point

```
day_point dp = floor<days>(system_clock::now());
cout << dp.time_since_epoch().count() << '\n';
```

```
16703
```

- It has been 16,703 days since 1970-01-01.

Date & Time

- day_point is a time_point!

```
auto tp = day_point{jan/3/1970};
assert(tp.time_since_epoch() == days{2});
```

Date & Time

- You can add hours to it:

```
auto tp = day_point{jan/3/1970} + 7h;
assert(tp.time_since_epoch() == 55h);
```

Date & Time

- You can add minutes to it:

```
auto tp = day_point{jan/3/1970} + 7h + 33min;
assert(tp.time_since_epoch() == 3333min);
```

Date & Time

- You can add seconds to it:

```
auto tp = day_point{jan/3/1970} + 7h + 33min + 20s;
assert(tp.time_since_epoch() == 200000s);
```

Date & Time

- You can add seconds to it:

```
auto tp = day_point{jan/3/1970} + 7h + 33min + 20s;
assert(tp.time_since_epoch() == 200000s);
```

- You can recover the day_point from it:

```
auto dp = floor<days>(tp);
assert(dp.time_since_epoch() == days{2});
```

Date & Time

- You can add seconds to it:

```
auto tp = day_point{jan/3/1970} + 7h + 33min + 20s;
assert(tp.time_since_epoch() == 200000s);
```

- You can recover the day_point from it:

```
auto dp = floor<days>(tp);
assert(dp.time_since_epoch() == days{2});
```

- You can get the time duration since midnight:

```
auto s = tp - dp;
assert(s == 27200s);
```

Date & Time

- You can add seconds to it:

```
auto tp = day_point{jan/3/1970} + 7h + 33min + 20s;
assert(tp.time_since_epoch() == 200000s);
```

- You can recover the day_point from it:
- You can get the time duration since midnight:

```
auto s = tp - dp;
assert(s == 27200s);
```

- You can break the duration into a h:m:s field type:

```
auto time = make_time(s);
assert(time.hours() == 7h);
assert(time.minutes() == 33min);
assert(time.seconds() == 20s);
```

Date & Time

- You can add seconds to it:

```
auto tp = day_point{jan/3/1970} + 7h + 33min + 20s;
```

- You can recover the day_point from it:
- You can get the time duration since midnight:
- You can break the duration into a h:m:s field type:

```
auto time = make_time(s);
assert(time.hours() == 7h);
assert(time.minutes() == 33min);
assert(time.seconds() == 20s);
```

- You can break the day_point into a y/m/d field type:

```
auto ymd = year_month_day{dp};
assert(ymd.year() == 1970_y);
assert(ymd.month() == jan);
assert(ymd.day() == 3_d);
```

Date & Time

- Much of the implementation and interface of all of this comes from your existing C++14 `<chrono>` header.
- This is a *seamless* extension of C++14 `<chrono>` `time_point`s and `duration`s into the realm of calendars.

How expensive is all this?!

Instead of running timing tests, I've constructed sample code and compared optimized assembly with simplistic obvious "C-like" solutions.

How expensive is all this?!

Compare factory functions for `year_month_day` with that for a simplistic struct.

"date.h"

```
date::year_month_day
make_year_month_day(int y, int m, int d)
{
    using namespace date;
    return year{y}/m/d;
}
```

C-like

```
struct YMD_4
{
    std::int16_t year;
    std::uint8_t month;
    std::uint8_t day;
};

YMD_4
make_YMD_4(int y, int m, int d)
{
    return {static_cast<std::int16_t>(y),
            static_cast<std::uint8_t>(m),
            static_cast<std::uint8_t>(d)};
}
```

How expensive is all this?!

```
    .globl __Z19make_year_month_dayiii
    .align 4, 0x90
__Z19make_year_month_dayiii:
    .cfi_startproc
## BB#0:
    pushq  %rbp
Ltmp2:
    .cfi_def_cfa_offset 16
Ltmp3:
    .cfi_offset %rbp, -16
    movq   %rsp, %rbp
Ltmp4:
    .cfi_def_cfa_register %rbp
    shll   $24, %edx
    shll   $16, %esi
    andl   $16711680, %esi
    movzwl %di, %eax
    orl %edx, %eax
    orl %esi, %eax
    popq   %rbp
    retq
    .cfi_endproc
```

```
    .globl __Z10make_YMD_4iii
    .align 4, 0x90
__Z10make_YMD_4iii:
    .cfi_startproc
## BB#0:
    pushq  %rbp
Ltmp2:
    .cfi_def_cfa_offset 16
Ltmp3:
    .cfi_offset %rbp, -16
    movq   %rsp, %rbp
Ltmp4:
    .cfi_def_cfa_register %rbp
    shll   $24, %edx
    shll   $16, %esi
    andl   $16711680, %esi
    movzwl %di, %eax
    orl %esi, %eax
    orl %edx, %eax
    popq   %rbp
    retq
    .cfi_endproc
```

rippleLabs

How expensive is all this?!

```
date::year_month_day
make_year_month_day(int y, int m, int d)
{
    using namespace date;
    return year{y}/m/d;
}
```

- The “Cute API” has zero space/time overhead!

How expensive is all this?!

Shift time point (measured in seconds)
epoch from 2000-01-01 to 1970-01-01:

“date.h”

```
using time_point = std::chrono::time_point
  <std::chrono::system_clock, std::chrono::seconds>;

time_point
shift_epoch(time_point t)
{
  using namespace date;
  return t + (day_point{jan/1/2000} - day_point{jan/1/1970});
}
```

C-like

```
long
shift_epoch(long t)
{
    return t + 946684800;
}
```

How expensive is all this?!

```
    .globl __Z11shift_epochNSt3__...
    .align 4, 0x90
__Z11shift_epochNSt3__...
    .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
    leaq   946684800(%rdi), %rax
    popq   %rbp
    retq
    .cfi_endproc
```

```
    .globl __Z11shift_epochl
    .align 4, 0x90
__Z11shift_epochl:
    .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
    leaq   946684800(%rdi), %rax
    popq   %rbp
    retq
    .cfi_endproc
```

How expensive is all this?!

“date.h”

```
using time_point = std::chrono::time_point
  <std::chrono::system_clock, std::chrono::seconds>;

time_point
shift_epoch(time_point t)
{
  using namespace date;
  return t + (day_point{jan/1/2000} - day_point{jan/1/1970});
}
```

convert to serial

convert to serial

Subtract to get 10,957 days

Convert to 946,684,800s

All at compile time!

C-like

```
long
shift_epoch(long t)
{
    return t + 946684800;
}
```

Inter-library comparison

Bloomberg bdlt

boost date_time

whatever we're calling this

- I want to plan an event for the 5^{th} Friday of every month which has one.
    - This happens 4, sometimes 5 times a year.
- How easy is this to code?
- How expensive is it?

Inter-library comparison

- I want to plan an event for the 5th Friday of every month which has one.
    - This happens 4, sometimes 5 times a year.
    - How easy is this to code?
    - How expensive is it?
- Fair API for test:

```
struct ymd
{
    std::int16_t y;
    std::uint8_t m;
    std::uint8_t d;
};

std::pair<std::array<ymd, 5>, std::uint32_t>
fifth_friday(int y);
```

Inter-library comparison

- Full disclosure, full optimizations on (-O3):

```
#include <iostream>

int
main()
{
    using namespace std;
    using namespace std::chrono;
    int y;
    std::cin >> y; // make sure results aren't computed at compile time
    auto t0 = steady_clock::now();
    auto p = fifth_friday(y);
    auto t1 = steady_clock::now();
     for (int i = 0; i < p.second; ++i)
        std::cout << p.first[i].y
                  << '-' << unsigned{p.first[i].m}
                  << '-' << unsigned{p.first[i].d} << '\n';
    cout << (t1-t0).count() << '\n';
}
```

Inter-library comparison

- Which will output something like this:

```
2015-1-30
2015-5-29
2015-7-31
2015-10-30
<time in nanoseconds>
```

- I'm reporting the average of ten runs on an idle 4 core MacBook Pro, using this command line:

```
$ a.out < tempfile
```

Inter-library comparison

This Is Not A Comprehensive Comparative Test!

Inter-library comparison

Bloomberg

```
typedef std::pair<std::array<ymd, 5>, std::uint32_t> fifth_friday_profile;

static const fifth_friday_profile profiles[14] = {
    { {{ {0, 3, 31},  {0, 6, 30},  {0, 9, 29},  {0, 12, 29},  {0,  0,  0} }}, 4 },
    { {{ {0, 3, 30},  {0, 6, 29},  {0, 8, 31},  {0, 11, 30},  {0,  0,  0} }}, 4 },
    { {{ {0, 3, 29},  {0, 5, 31},  {0, 8, 30},  {0, 11, 29},  {0,  0,  0} }}, 4 },
    { {{ {0, 1, 31},  {0, 5, 30},  {0, 8, 29},  {0, 10, 31},  {0,  0,  0} }}, 4 },
    { {{ {0, 1, 30},  {0, 5, 29},  {0, 7, 31},  {0, 10, 30},  {0,  0,  0} }}, 4 },
    { {{ {0, 1, 29},  {0, 4, 30},  {0, 7, 30},  {0, 10, 29},  {0, 12, 31} }}, 5 },
    { {{ {0, 4, 29},  {0, 7, 29},  {0, 9, 30},  {0, 12, 30},  {0,  0,  0} }}, 4 },
    { {{ {0, 3, 30},  {0, 6, 29},  {0, 8, 31},  {0, 11, 30},  {0,  0,  0} }}, 4 },
    { {{ {0, 3, 29},  {0, 5, 31},  {0, 8, 30},  {0, 11, 29},  {0,  0,  0} }}, 4 },
    { {{ {0, 2, 29},  {0, 5, 30},  {0, 8, 29},  {0, 10, 31},  {0,  0,  0} }}, 4 },
    { {{ {0, 1, 31},  {0, 5, 29},  {0, 7, 31},  {0, 10, 30},  {0,  0,  0} }}, 4 },
    { {{ {0, 1, 30},  {0, 4, 30},  {0, 7, 30},  {0, 10, 29},  {0, 12, 31} }}, 5 },
    { {{ {0, 1, 29},  {0, 4, 29},  {0, 7, 29},  {0,  9, 30},  {0, 12, 30} }}, 5 },
    { {{ {0, 3, 31},  {0, 6, 30},  {0, 9, 29},  {0, 12, 29},  {0,  0,  0} }}, 4 }
};
```

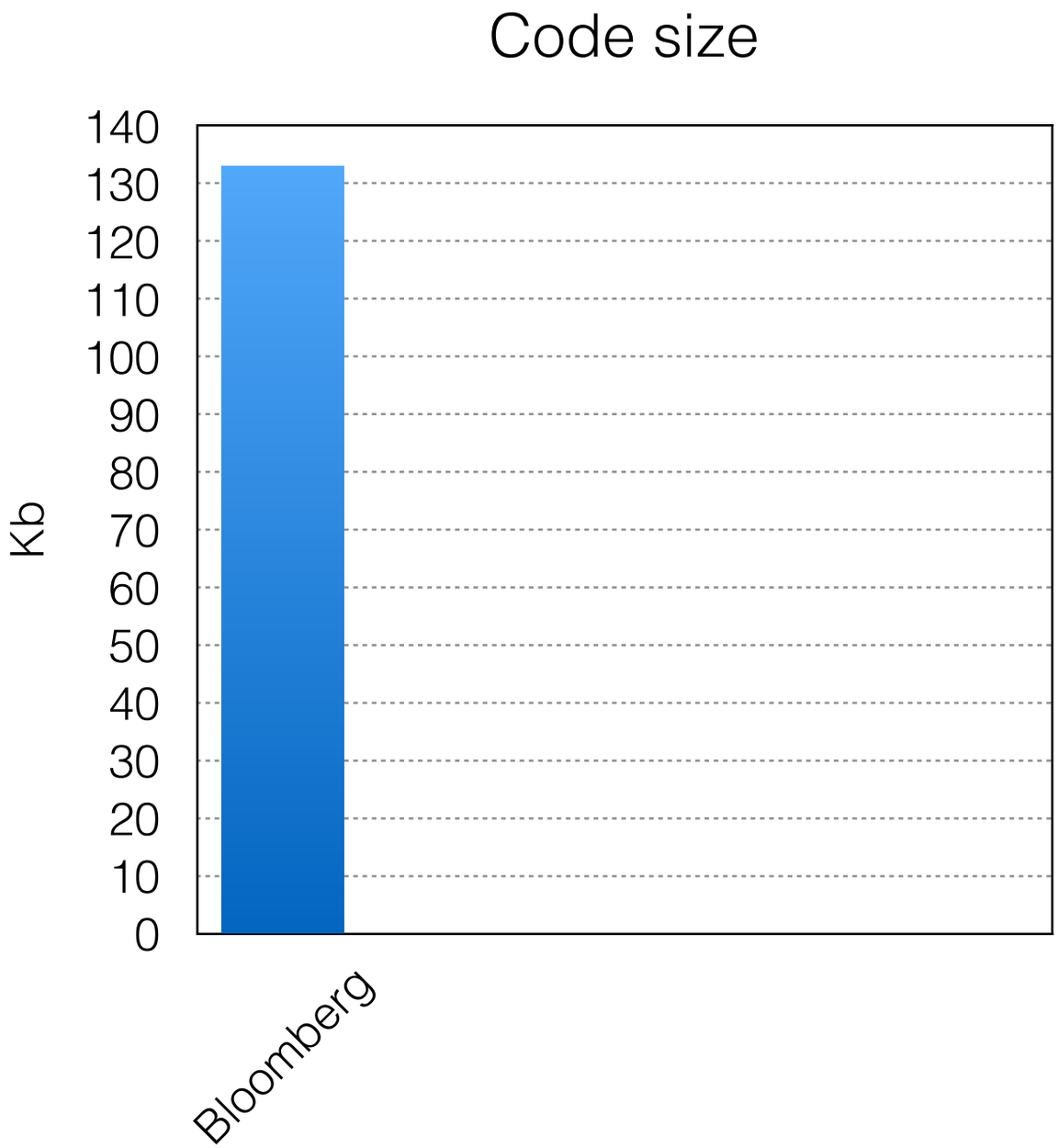
rippleLabs

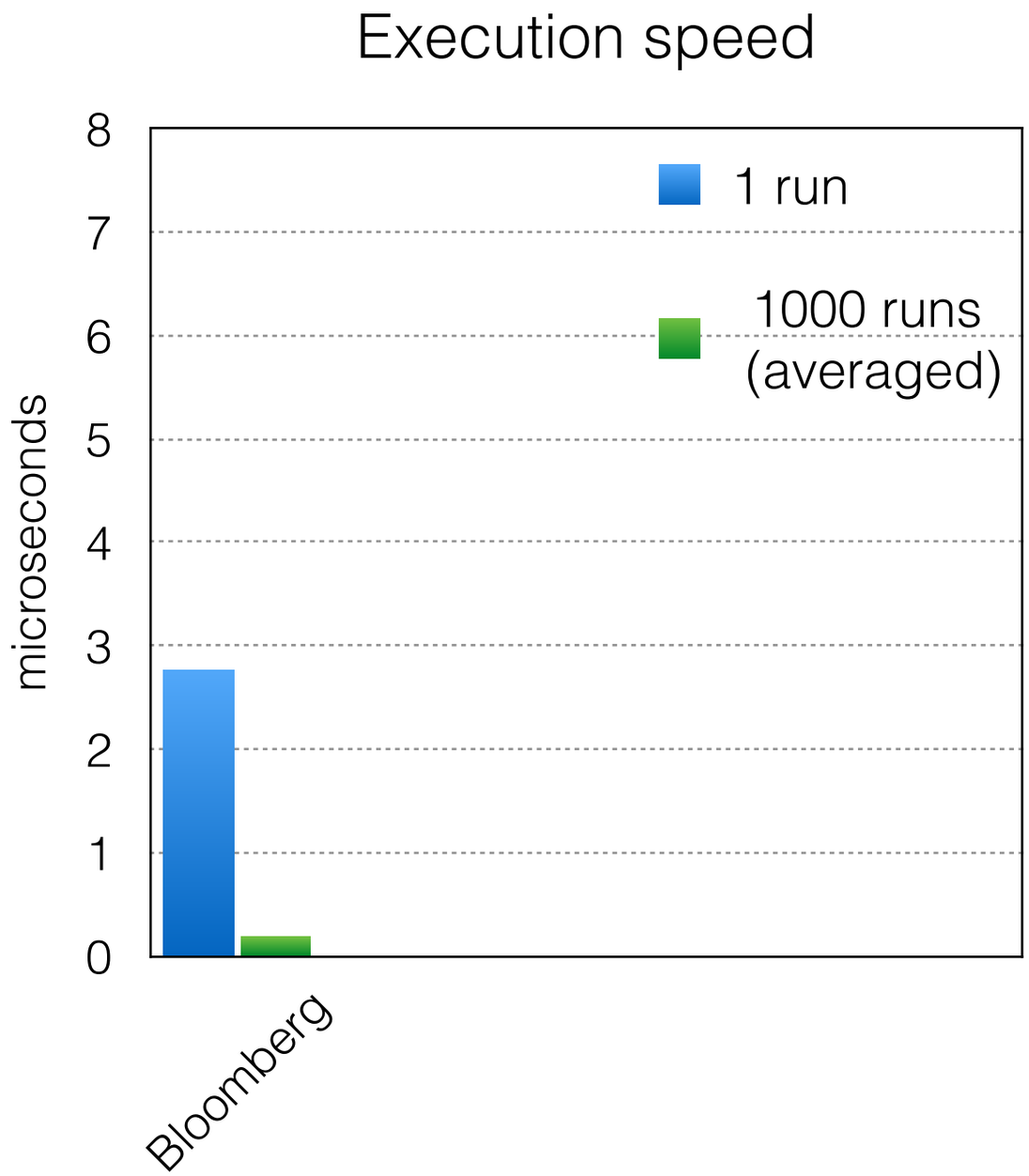
Inter-library comparison

Bloomberg

```
std::pair<std::array<ymd, 5>, std::uint32_t>
fifth_friday(int year)
{
    typedef BloombergLP::bdlt::SerialDateImpUtil SDIU;
    int index = SDIU::ymdToDayOfWeek(year, 1, 1) - 1;
    if (SDIU::isLeapYear(year))
        index += 7;
    fifth_friday_profile profile = profiles[index];
    const std::int16_t y = static_cast<std::int16_t>(year);
    for (ymd& date: profile.first)
        date.y = y;
    return profile;
}
```

Inter-library comparison





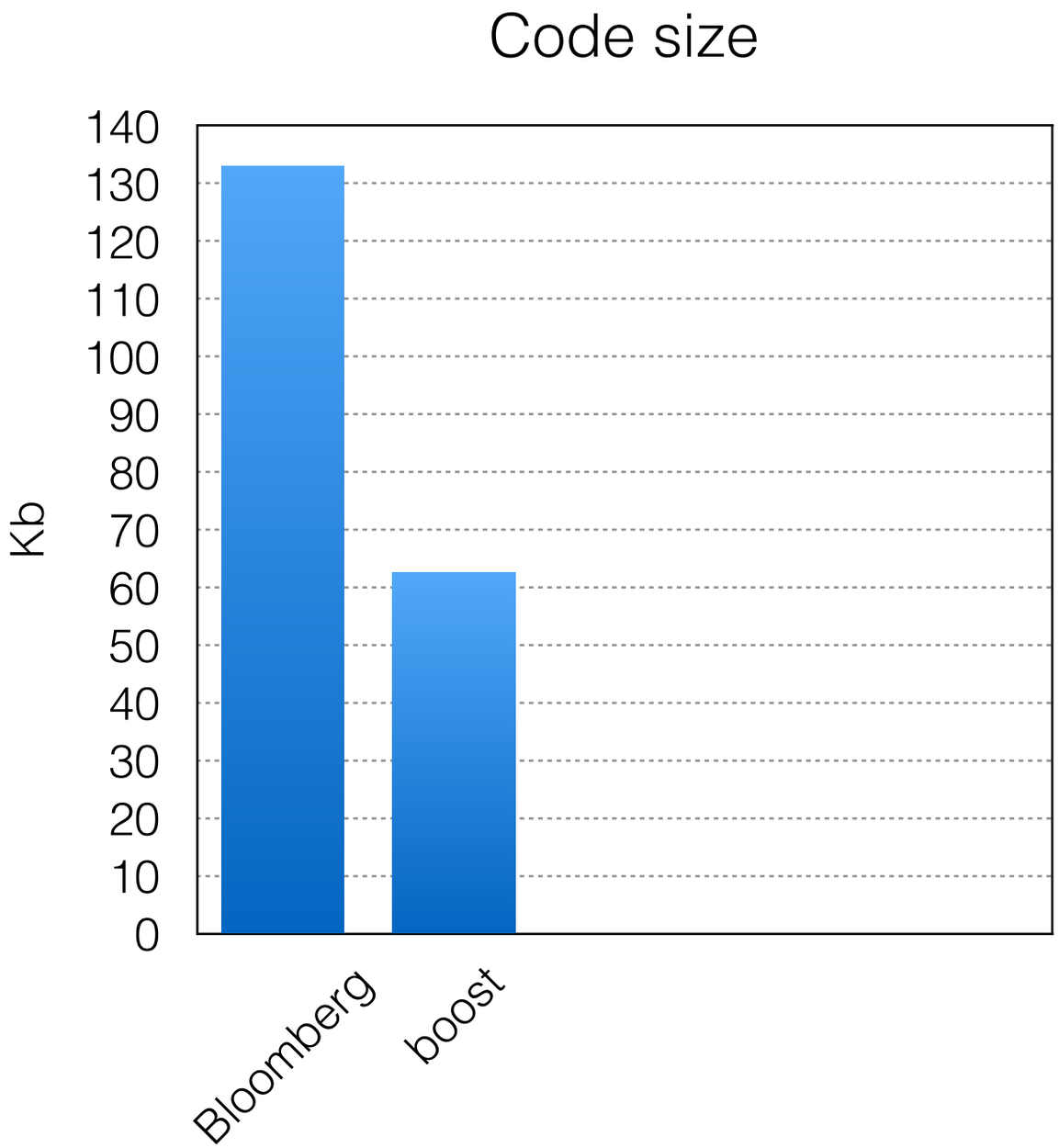
rippleLabs

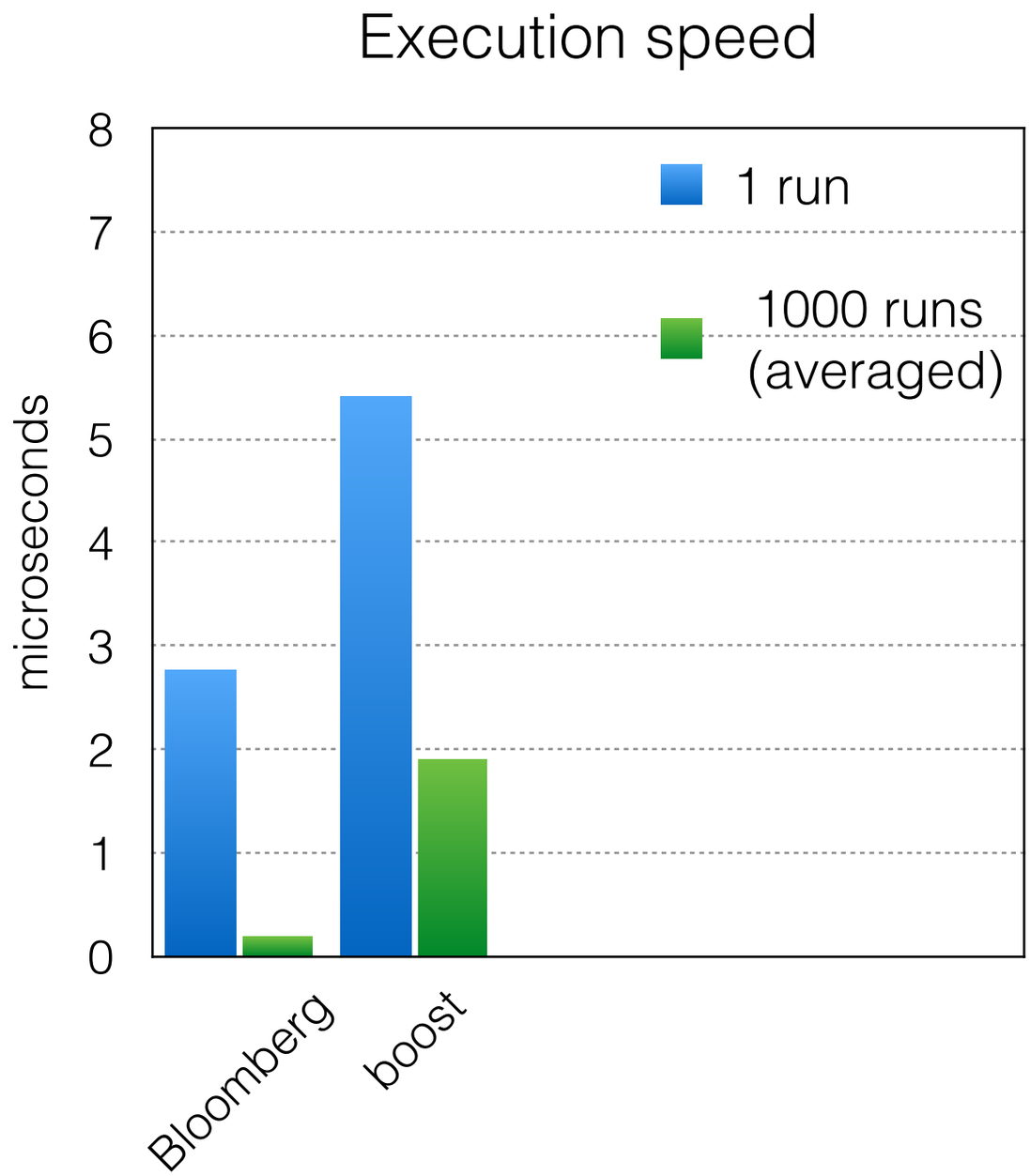
Inter-library comparison

boost

```
std::pair<std::array<ymd, 5>, std::uint32_t>
fifth_friday(int y)
{
    using namespace boost::gregorian;
    std::array<ymd, 5> dates;
    std::uint32_t n = 0;
    for (auto m = 1u; m <= 12; ++m)
    {
        auto d = nth_day_of_the_week_in_month(nth_kday_of_month::fifth,
                                              Friday, m).get_date(y);
        auto day = d.day();
        if (day >= 29)
        {
            dates[n].y = y;
            dates[n].m = m;
            dates[n].d = day;
            ++n;
        }
    }
    return {dates, n};
}
```

Inter-library comparison





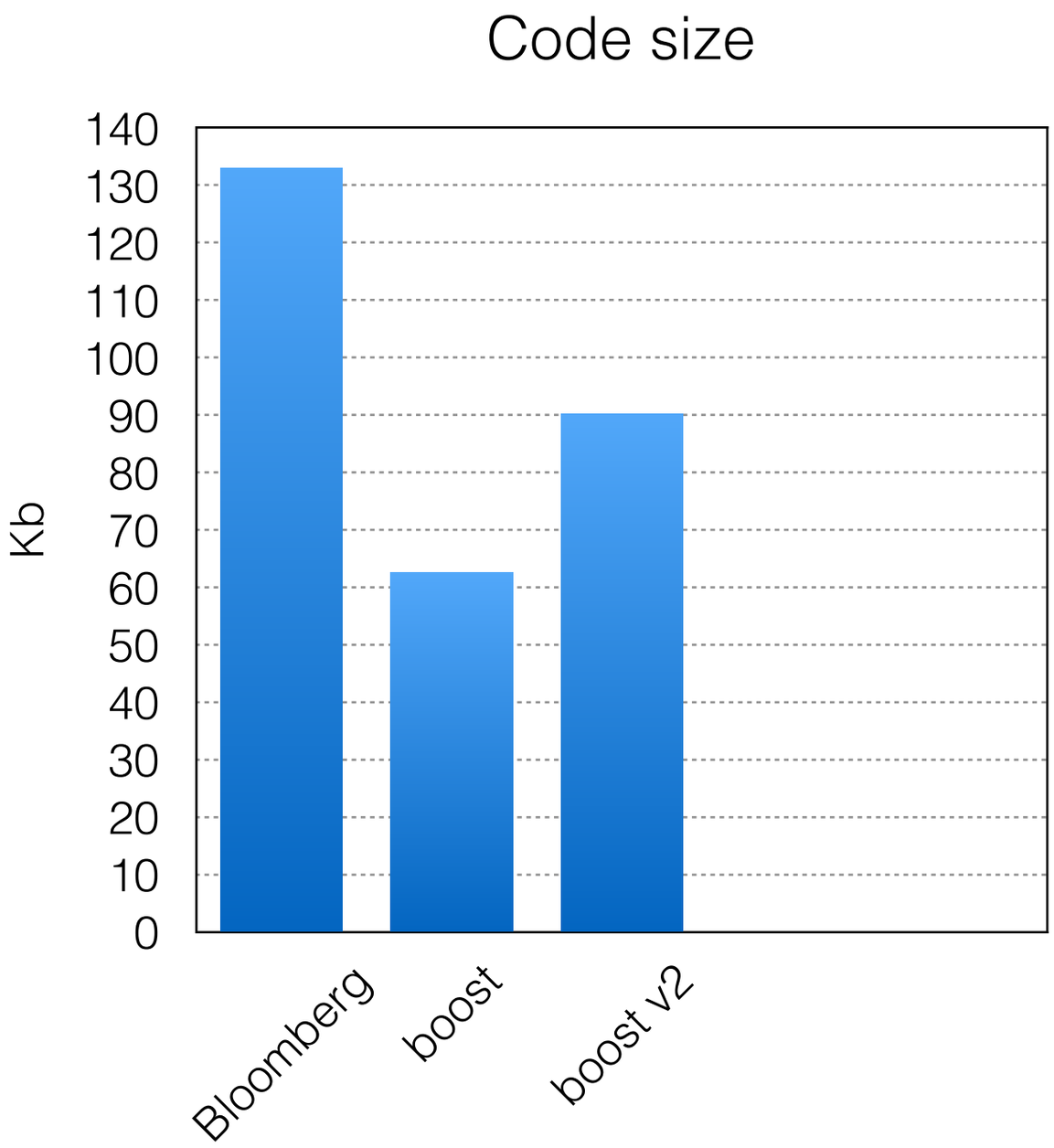
rippleLabs

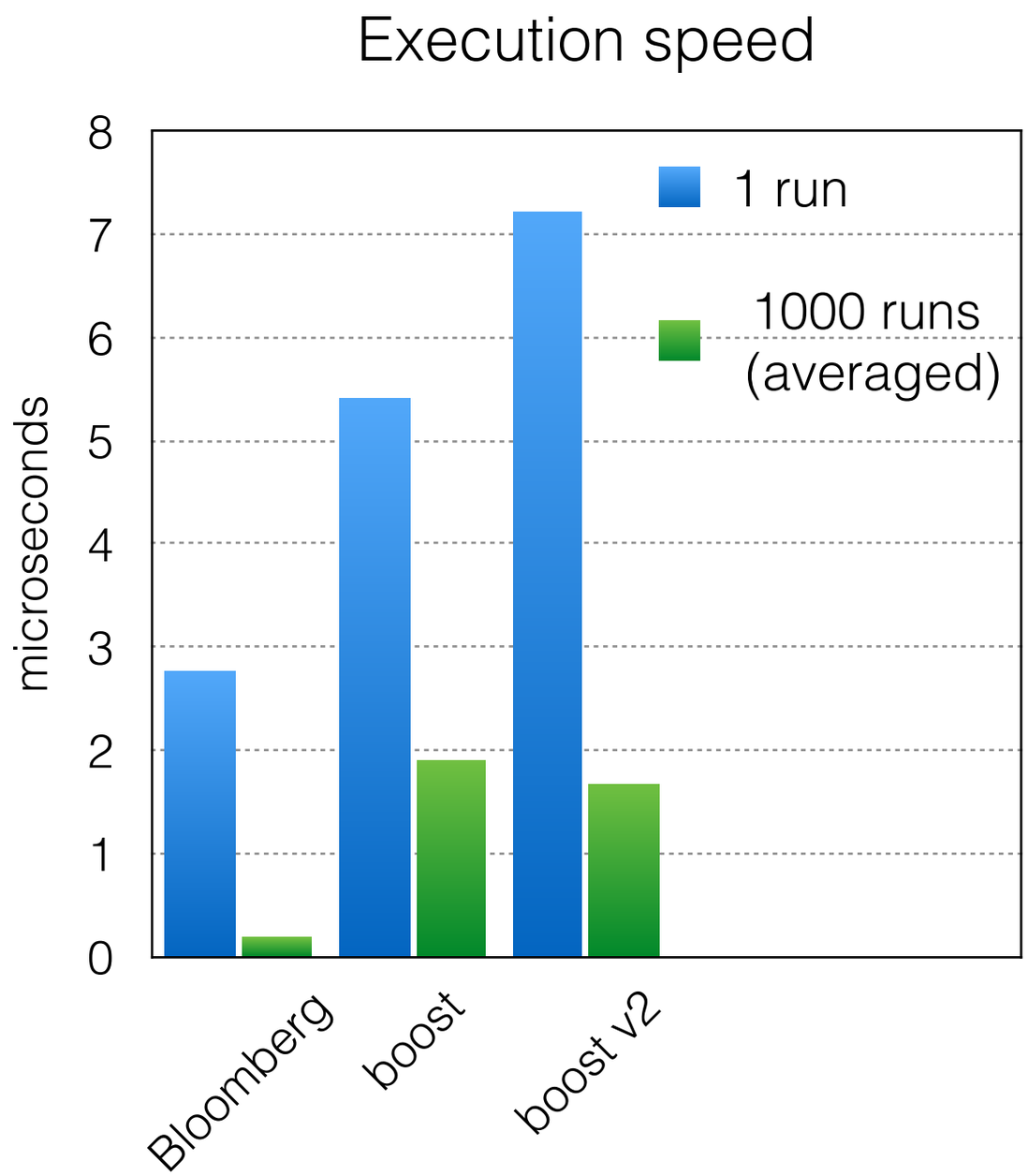
Inter-library comparison

boost v2 — under construction

```
std::pair<std::array<ymd, 5>, std::uint32_t>
fifth_friday(int y)
{
    using namespace boost::date_time2;
    std::array<ymd, 5> dates;
    std::uint32_t n = 0;
    for (auto m = 1u; m <= 12; ++m)
    {
        day_of_week dow(Fifth, Fri, m, y);
        auto day = year_month_day(dow).day_of_month;
        if (day >= 29)
        {
            dates[n].y = y;
            dates[n].m = m;
            dates[n].d = day;
            ++n;
        }
    }
    return {dates, n};
}
```

Inter-library comparison





rippleLabs

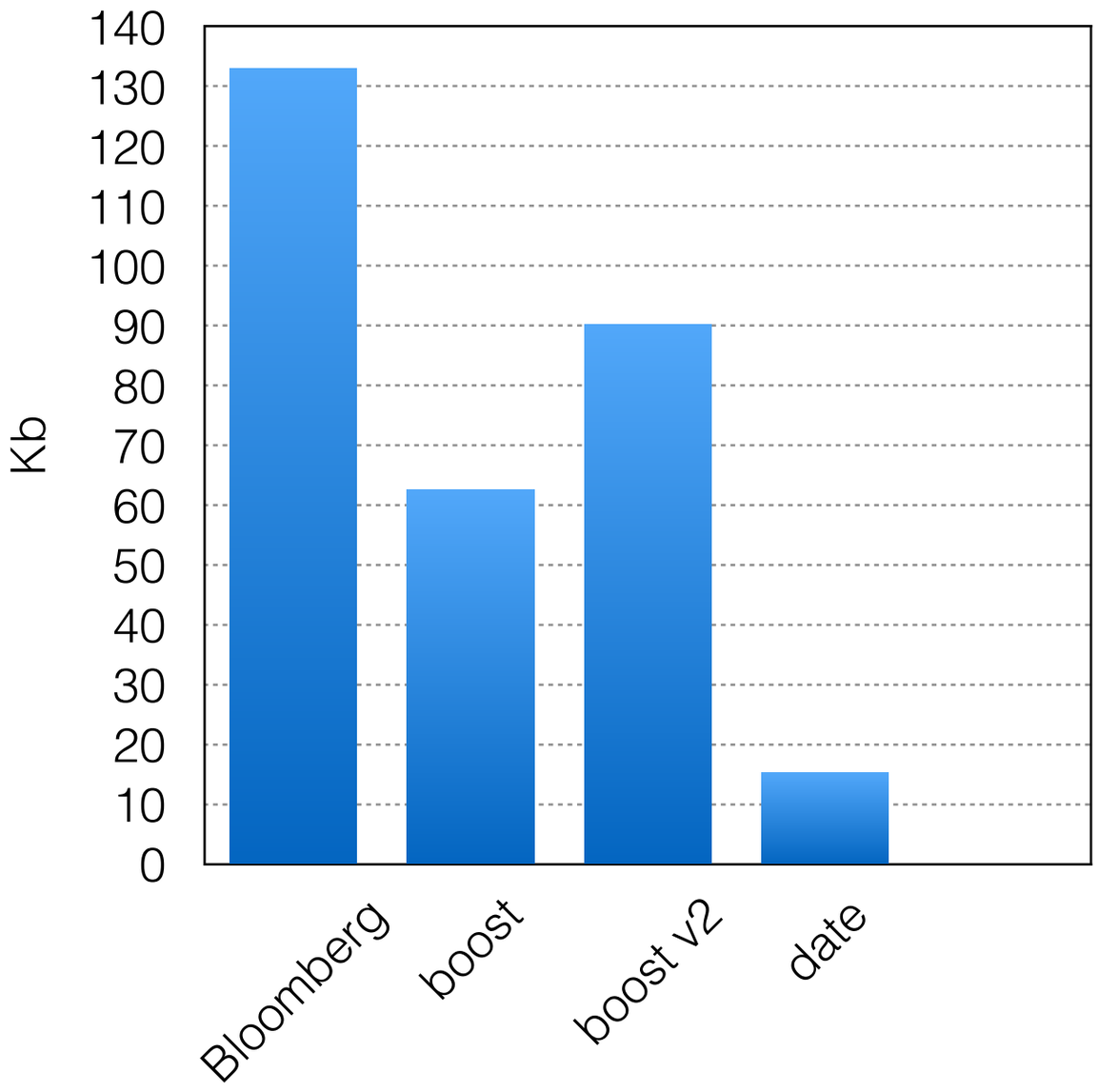
Inter-library comparison

date

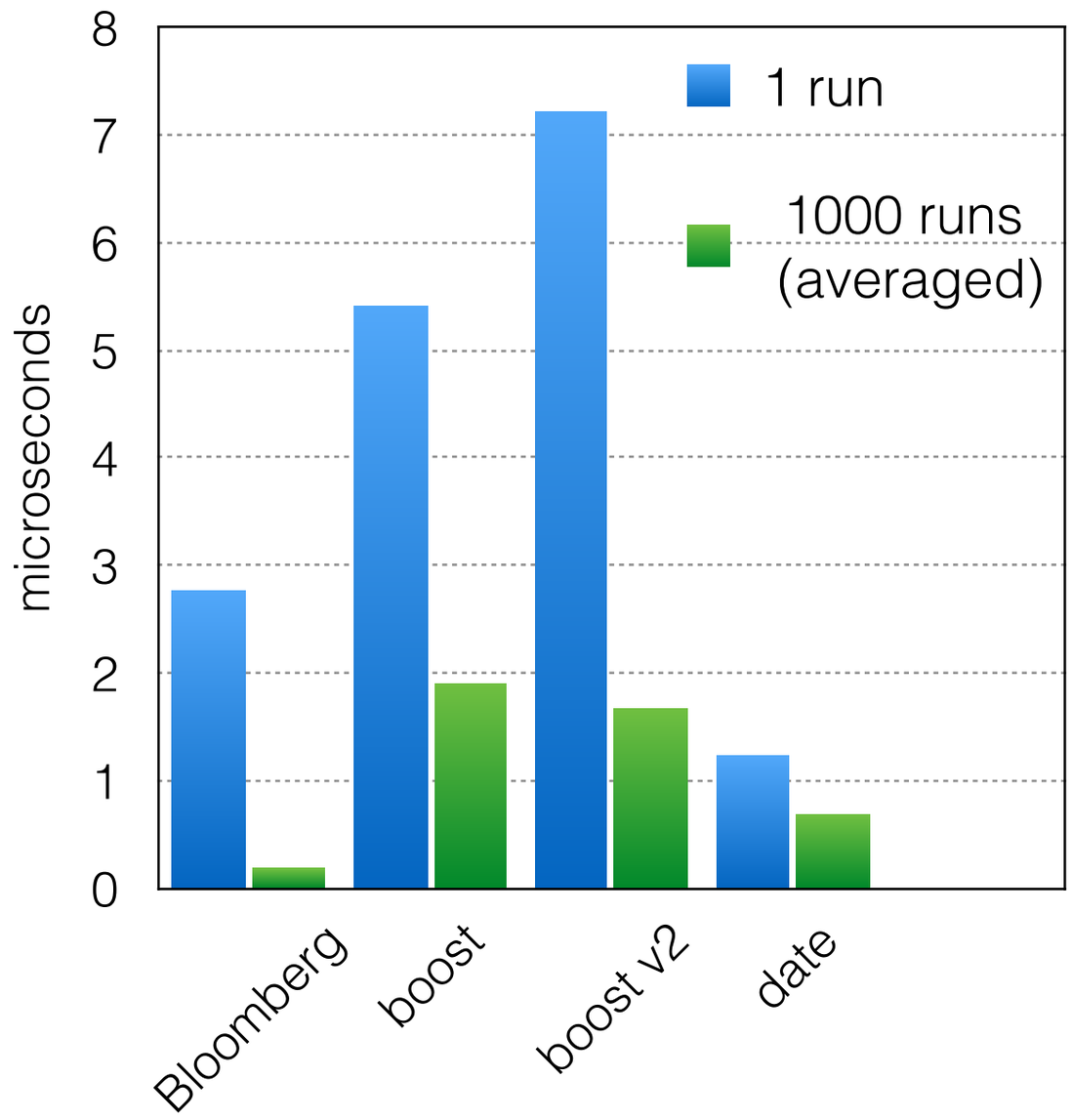
```
std::pair<std::array<ymd, 5>, std::uint32_t>
fifth_friday(int y)
{
    using namespace date;
    std::array<ymd, 5> dates;
    std::uint32_t n = 0;
    for (auto m = 1u; m <= 12; ++m)
    {
        auto d = year_month_weekday{fri[last]/m/y};
        if (d.index() == 5)
        {
            auto x = year_month_day{d};
            dates[n].y = y;
            dates[n].m = m;
            dates[n].d = unsigned{x.day()};
            ++n;
        }
    }
    return {dates, n};
}
```

Inter-library comparison

Code size



Execution speed



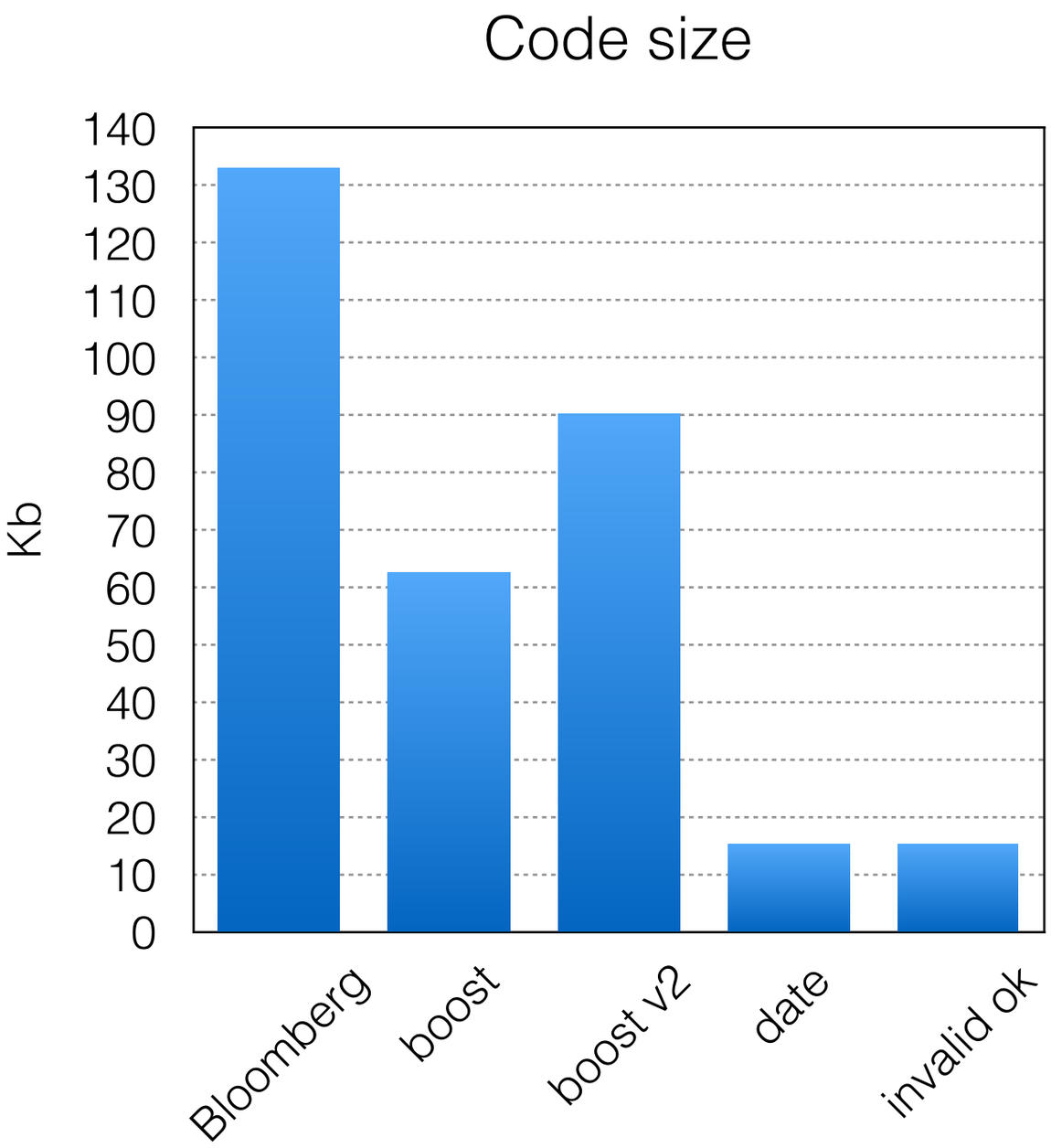
rippleLabs

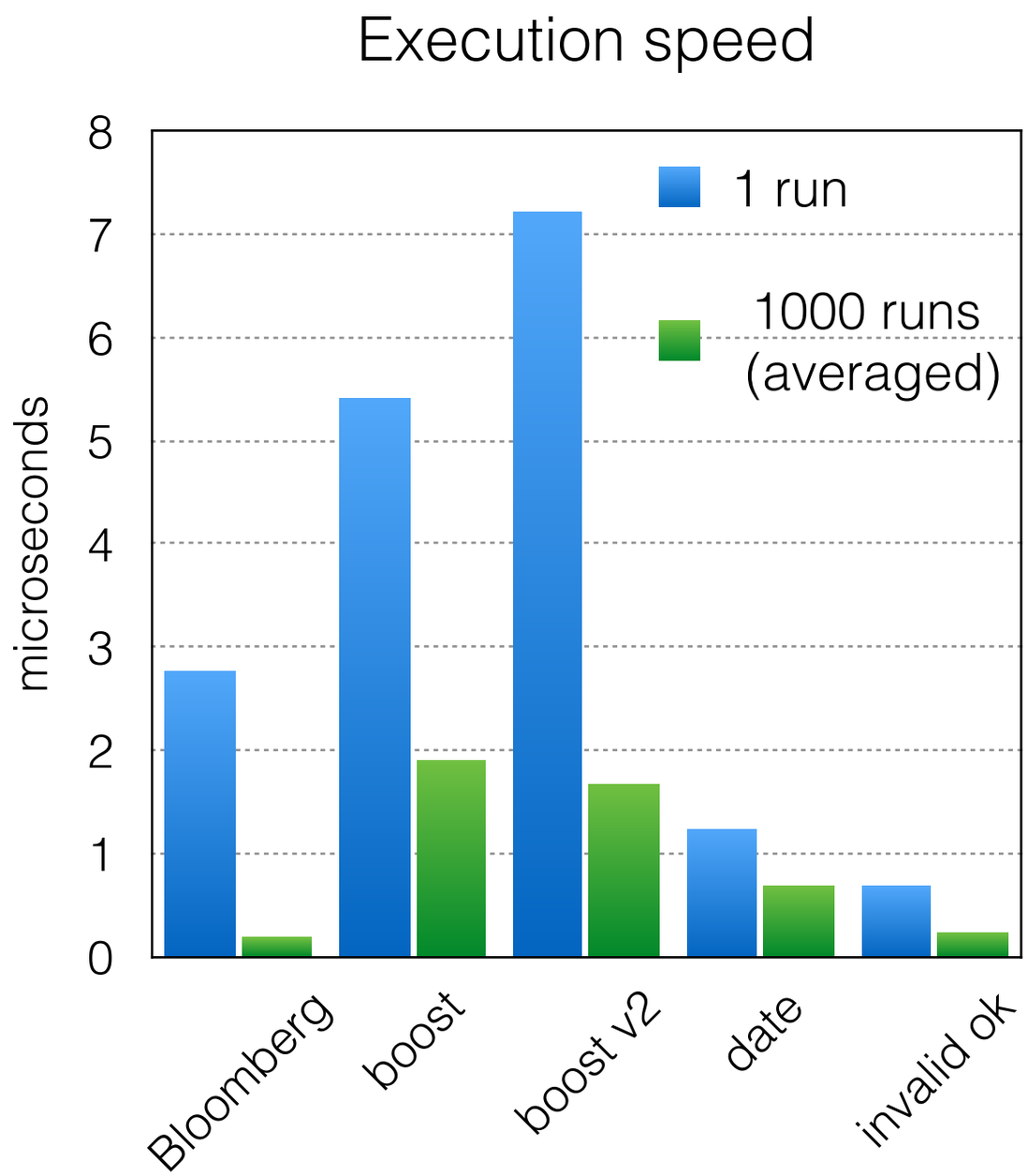
Inter-library comparison

invalid date is ok

```
std::pair<std::array<ymd, 5>, std::uint32_t>
fifth_friday(int y)
{
    using namespace date;
    std::array<ymd, 5> dates;
    std::uint32_t n = 0;
    for (auto m = 1u; m <= 12; ++m)
    {
        auto d = fri[5]/m/y;
        if (d.ok())
        {
            auto x = year_month_day{d};
            dates[n].y = y;
            dates[n].m = m;
            dates[n].d = unsigned{x.day()};
            ++n;
        }
    }
    return {dates, n};
}
```

Inter-library comparison





rippleLabs

Inter-library comparison

- date.h is small.
- date.h is fast.
- The ability to create invalid dates without being scolded can be both a readability and performance advantage.

What day of the week is July 4, 2001?

From the C standard:

```
#include <stdio.h>
#include <time.h>

static const char *const wday[] = {
  "Sunday", "Monday", "Tuesday", "Wednesday",
  "Thursday", "Friday", "Saturday", "-unknown-"};

int main() {
  struct tm time_str;
  time_str.tm_year = 2001 - 1900;
  time_str.tm_mon  = 7 - 1;
  time_str.tm_mday = 4;
  time_str.tm_hour = 0;
  time_str.tm_min  = 0;
  time_str.tm_sec  = 0;
  time_str.tm_isdst  = -1;
  if (mktime(&time_str) == (time_t)(-1))
    time_str.tm_wday = 7;
  printf("%s\n", wday[time_str.tm_wday]);
}
```

What day of the week is July 4, 2001?

```
#include <stdio.h>
#include <time.h>

static const char *const wday[]
  "Sunday", "Monday", "Tuesday"
  "Thursday", "Friday", "Saturd

int main() {
  struct tm time_str;
  time_str.tm_year = 2001 - 190
  time_str.tm_mon  = 7 - 1;
  time_str.tm_mday = 4;
  time_str.tm_hour = 0;
  time_str.tm_min  = 0;
  time_str.tm_sec  = 0;
  time_str.tm_isdst  = -1;
  if (mktime(&time_str) == (tim
    time_str.tm_wday = 7;
  printf("%s\n", wday[time_str.
}
```

- It is a little easier with the date lib.

```
#include "date.h"
#include <iostream>

int
main()
{
  using namespace date;
  std::cout << weekday{2001_y/jul/4} << '\n';
}
```

What day of the week is July 4, 2001?

```
#include <stdio.h>
#include <time.h>

static const char *const wday[]
  "Sunday", "Monday", "Tuesday"
  "Thursday", "Friday", "Saturc

int main() {
  struct tm time_str;
  time_str.tm_year = 2001 - 190
  time_str.tm_mon  = 7 - 1;
  time_str.tm_mday = 4;
  time_str.tm_hour = 0;
  time_str.tm_min  = 0;
  time_str.tm_sec  = 0;
  time_str.tm_isdst  = -1;
  if (mktime(&time_str) == (tim
    time_str.tm_wday = 7;
  printf("%s\n", wday[time_str.
}
```

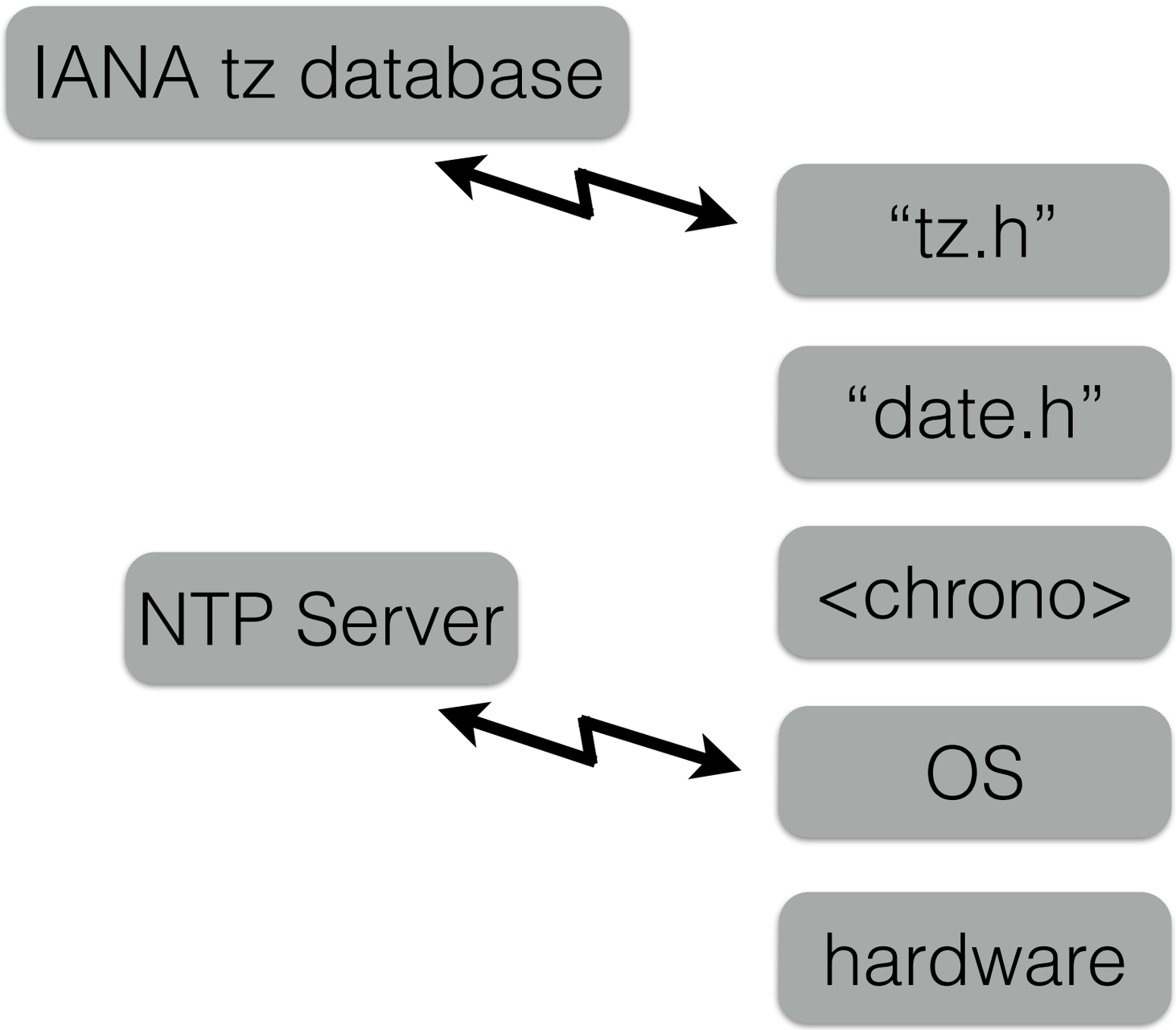
- And it can be done at compile time.

```
#include "date.h"


int
main()
{
  using namespace date;
  static_assert(weekday{2001_y/jul/4} == wed);
}
```

rippleLabs

Just a hint of tz



Just a hint of tz

"tz.h"

- Everything shown so far is implicitly in the UTC timezone.
- To work with other timezones you need this additional, higher-level library.

```
auto zone = locate_zone("America/Los_Angeles");
auto now = floor<milliseconds>(system_clock::now());
auto local = zone->to_local(now);
cout << now          << ' ' << "UTC"          << '\n';
cout << local.first << ' ' << local.second << '\n';
```

```
2015-09-25 16:50:06.123 UTC
2015-09-25 09:50:06.123 PDT
```

Just a hint of tz

"tz.h"

- Everything shown so far is implicitly in the UTC timezone.
- To work with other timezones you need this additional, higher-level library.

```
auto zone = current_zone();
auto now = floor<milliseconds>(system_clock::now());
auto local = zone->to_local(now);
cout << now          << ' ' << "UTC"          << '\n';
cout << local.first << ' ' << local.second << '\n';
```

```
2015-09-25 16:50:06.123 UTC
2015-09-25 09:50:06.123 PDT
```

Just a hint of tz

"tz.h"

- Everything shown so far is implicitly in the UTC timezone.
- To work with other timezones you need this additional, higher-level library.
- Works with the *complete* IANA TZ database history.

```
auto local = zone->to_local(day_point{feb/10/1942});
cout << local.first << ' ' << local.second << '\n';
```

```
1942-02-09 17:00:00 PWT
```

Just a hint of tz

"tz.h"

- Everything shown so far is implicitly in the UTC timezone.
- To work with other timezones you need this additional, higher-level library.
- Works with the *complete* IANA TZ database history.
- Includes facilities for computing with leap-seconds (which are also part of the IANA TZ database).

```
auto local = zone->to_local(day_point{feb/10/1942});
cout << local.first << ' ' << local.second << '\n';
```

```
1942-02-09 17:00:00 PWT
```

Summary

- A high performance, minimal API extension is made to `<chrono>` enabling easy and intuitive calendrical computations.
- This library does not do everything everybody wants it to do.
  - This is not a kitchen sink API.
- Instead it enables everybody to do for themselves what they want to do, safely, easily and efficiently.

http://howardhinnant.github.io/date_v2.html

I’m standing on the shoulders of giants

- Toward a Standard C++ ‘Date’ Class (N3344)
  - Stefano Pacifico, Alisdair Meredith, John Lakos
- Boost date_time
  - Jeff Garland
- Relaxing constraints on constexpr functions (N3652)
  - Richard Smith
- Generalized Constant Expressions (N1521)
  - Gabriel Dos Reis, Bjarne Stroustrup