Howard E. Hinnant
2019-12-31

`date`

Contents

Introduction

This paper fully documents a date and time library for use with C++11 and C++14.

Implementation

This entire library is implemented in a single header: date.h and is open source.

It uses the algorithms from `chrono`-Compatible Low-Level Date Algorithms. If you want detailed explanations of the algorithms, go there.

It performs best with C++14, which has vastly improved `constexpr` rules. However, the library will auto-adopt to C++11, sacrificing several `constexpr` declarations. In C++11, this will effectively transfer some computations that should be done at compile-time to run-time. Porting to C++98/03 has not been attempted.

Overview

Here is an overview of all the types we are going to talk about at some point. They are all fully covered in the reference section. This link is just there to give you a view of everything on one quick page so that you don't get lost or overwhelmed. Many of these types will never need to be explicitly named in typical use cases.

date_types.jpeg

This library builds date and date/time support on top of the `<chrono>` library. However it does not support timezones nor leap seconds. A separate library is provided, built on top of this one, for timezone and leap second support. Thus you only pay for such support if you need it.

Major Concepts

A date on a calendar is a time point with a resolution of one day. This is not conceptually different from `std::chrono::system_clock::time_point` which is a time point with a resolution of `std::chrono::system_clock::period` (typically on the order of a microsecond or nanosecond). Time points are divided into two basic types:

1. serial-based
2. field-based

This terminology is gratefully borrowed from N3344. In short, a *serial-based* time point stores a single count of time durations from some epoch. `std::chrono::system_clock::time_point` is an example of a *serial-based* time point. A count of days from some epoch is also a *serial-based* time point. A *field-based* time point represents the same concept, but with several fields that are typically used for human consumption. For example one might represent the time of day with the number of seconds since midnight (*serial-based*), or with a structure such as `{hours, minutes, seconds}` since midnight (*field-based*). Similarly one can represent a date with a count of days, or with a `{year, month, day}` structure. Both representations are useful, and each has its strengths and weaknesses.

Major Types

This library is composed of many types. But here are the most important ones:

1. `sys_days`: A count of days since `system_clock`'s epoch. This is a serial-based time point with a resolution of one day.
2. `year_month_day`: A type that holds a year (e.g. 2015), a month (encoded as 1 thru 12), and a day (encoded as 1 thru 31). This is a field-based time point with a resolution of one day.
3. `year_month_weekday`: A type that holds a year (e.g. 2015), a month (encoded as 1 thru 12), a day of the week (encoded as 0 thru 6), and an index in the range [1, 5] indicating if this is the first, second, etc. weekday of the indicated month. This is a field-based time point with a resolution of one day.

Examples

The entire library is in namespace `date`. The examples in this overview assume:

```
using namespace date;
using namespace std::chrono;
```

in order to cut down on the verbosity.

Example: Constructing dates

`year_month_day`

`constexpr` dates can be constructed from literals in any one of 3 orders:

```
constexpr auto x1 = 2015_y/March/22;
constexpr auto x2 = March/22/2015;
constexpr auto x3 = 22_d/March/2015;
```

`x1`, `x2` and `x3` all have the type: `year_month_day`, and are all equal to one another. Any other order, or any ambiguity is caught at compile time. These three orders were chosen because these are the three orders that people actually use.

Integral types can be converted to `year`, `month`, or `day` to create run time values:

```
int y = 2015;
int m = 3;
int d = 22;
auto x1 = year{y}/m/d;
auto x2 = month{m}/d/y;
auto x3 = day{d}/m/y;
```

As long as the type of the first date component is known, the following components become unambiguous and can be formed either with the proper type, or with an `int` expression. And since there is only one order with `day` in the middle, that order need only properly type the `day` to remain unambiguous:

```
auto x = m/day{d}/y;
```

No matter the order of the `year`, `month` and `day` specifiers, and no matter whether some of the fields implicitly convert from `int`, the type of these expressions is `year_month_day`. However because of `auto`, it is very often unimportant how to spell the somewhat verbose type name `year_month_day`. The compiler knows how to spell the type, and the meaning is clear to the human reader without an explicit type.

If there is *any* ambiguity, the code **will not compile**. If the code does compile, it will be unambiguous and quite readable.

`year_month_day_last`

Anywhere you can specify a `day` you can instead specify `last` to indicate the last day of the month for that year:

```
constexpr auto x1 = 2015_y/February/last;
constexpr auto x2 = February/last/2015;
constexpr auto x3 = last/February/2015;
```

Instead of constructing a `year_month_day`, these expressions construct a `year_month_day_last`. This type always represents the last day of a `year` and `month`. `year_month_day_last` will implicitly convert to `year_month_day`. Note though that because of `auto` both the reader and the writer of the code can very often remain blissfully ignorant of the spelling of `year_month_day_last`. The API of `year_month_day` and `year_month_day_last` are virtually identical so that the learning curve is shallow, and generic code usually does not need to know which type it is operating on.

`year_month_weekday`

Anywhere you can specify a `day` you can instead specify an *indexed* `weekday` to indicate the nth weekday of the month for that year:

```
constexpr auto x1 = 2015_y/March/Sunday[4];
constexpr auto x2 = March/Sunday[4]/2015;
constexpr auto x3 = Sunday[4]/March/2015;
```

The type constructed is `year_month_weekday` which will implicitly convert to and from a `sys_days`. If you want to convert to a `year_month_day` you can do so explicitly, which will implicitly convert to `sys_days` and back:

```
constexpr auto x4 = year_month_day{x3};
```

`year_month_weekday_last`

`weekday`'s can be indexed with `last` and used as above:

```
constexpr auto x1 = 2015_y/March/Sunday[last];
constexpr auto x2 = March/Sunday[last]/2015;
constexpr auto x3 = Sunday[last]/March/2015;
constexpr auto x4 = year_month_day{x3};
cout << x3 << '\n';
cout << x4 << '\n';
```

This creates an object of type `year_month_weekday_last`. The above code outputs:

```
2015/Mar/Sun[last]
2015-03-29
```

Escape hatch from the conventional syntax

If you hate the conventional syntax, don't use it. Every type has a descriptive name, and public type-safe constructors to do the same job. The "conventional syntax" is a non-friended and *zero-abstraction-penalty* layer on top of a complete lower-level API.

```
constexpr auto x1 = 2015_y/March/Sunday[last];
constexpr auto x2 = year_month_weekday_last{year{2015}, month{3u}, weekday_last{weekday{0u}}};
static_assert(x1 == x2,                                   "No matter the API, x1 and x2 have the same value ...");
static_assert(is_same<decltype(x1), decltype(x2)>::value, "... and are the same type");
cout << x1 << '\n';
cout << x2 << '\n';

2015/Mar/Sun[last]
2015/Mar/Sun[last]
```

Example: Today

To get today as a `sys_days`, use `system_clock::now()` and `floor` to convert the `time_point` to a `sys_days`:

```
auto today = floor<days>(system_clock::now());
cout << today << '\n';
```

Currently this outputs for me:

```
2015-03-22
```

To get today as a `year_month_day`, get a `sys_days` as above and convert it to a `year_month_day`:

```
auto today = year_month_day{floor<days>(system_clock::now())};
cout << today << '\n';
```

Which again currently outputs for me:

```
2015-03-22
```

To get today as a `year_month_weekday`, get a `sys_days` as above and convert it to a `year_month_weekday`:

```
auto today = year_month_weekday{floor<days>(system_clock::now())};
cout << today << '\n';
```

Currently this outputs for me:

```
2015/Mar/Sun[4]
```

This indicates that today (2015-03-22) is the fourth Sunday of Mar., 2015.

Example: Contrast with C

Just about everything C++ has for date handling, it inherits from C. I thought it would be fun to contrast an example that comes from the C standard with this library.

From the C standard: 7.27.2.3 The `mktime` function, paragraph 4:

EXAMPLE What day of the week is July 4, 2001?

```
#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]);
}
```

This program outputs:

```
Wednesday
```

Coding the same problem up in this library looks like:

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

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

which outputs:

```
Wed
```

And if we really want to show off, this can all be done at compile time:

```
#include "date.h"

int
main()
{
    using namespace date;
    static_assert(weekday{2001_y/July/4} == Wednesday, "");
}
```

(Date ordering was switched simply to assure the gentle reader that we're not hung up on m/d/y ordering.)

Errors

No exceptions are thrown in the entire library, except those that might be thrown from `std::ostream` when streaming values to a `std::ostream`. Most of the types have a `constexpr const` member function named `ok()` which returns `true` if the object represents a valid date, or date component, and otherwise returns `false`. However it is perfectly fine (and sometimes actually useful) for these objects to contain invalid dates or date components. For example, here is a very simple and remarkably efficient program to output the odd Fridays of every month in 2015:

```
int
main()
{
    using namespace std;
    using namespace date;
    for (auto m = 1; m <= 12; ++m)
    {
        auto meet = year_month_day{m/Friday[1]/2015};
```

```
            cout << meet << '\n';
            meet = meet.year()/meet.month()/(meet.day()+weeks{2});
            cout << meet << '\n';
            meet = meet.year()/meet.month()/(meet.day()+weeks{2});
            if (meet.ok())
                cout << meet << '\n';
        }
    }
```

There is a relatively expensive (expense is relative here — maybe as much as 100ns) conversion from `year_month_weekday` to `year_month_day` at the top of the loop to find the first Friday of the month. This computation can never fail. And after that it is dirt cheap to find the 3rd Friday — you just have to add 2 weeks to the day field. This computation also can never fail. And it is similarly cheap to find the 5th Friday of each month. However, not every month has a fifth Friday. But nevertheless it is ok to form such a `year_month_day`. The computation can never fail, though it may render an invalid date. Afterwards one can simply ask: Did I create a valid date? If so, print it out, otherwise just continue on.

This program is easy to read, easy to write, will fail to compile if any careless errors are made, and extremely high performance. The date computations are dwarfed by the run time costs of the I/O.

Because practically nothing throws exceptions, this entire library is liberally sprinkled with `noexcept`.

A few of the operations have a precondition that `ok() == true`. These are generally conversions to `sys_days`, and month-oriented and weekday-oriented arithmetic. Anywhere there is a precondition, and those places are few, the precondition is checkable with `ok()`.

This library catches many errors (especially ambiguity errors) at compile time. What's left over is left up to `ok()`. Sometimes `!ok()` will represent an error, and other times it will simply represent something innocuous that can be ignored (such as in the example above). If `!ok()` actually represents an error that must be dealt with, the choice of whether to assert or throw is made by the client of this library.

Efficiency

In N3344 Pacifico, Meredith and Lakos present a thorough survey of date types and their performance. This library has been strongly influenced by this excellent paper. `year_month_day` is equivalent to what N3344 terms `YMD_4`. And `sys_days` is equivalent to what N3344 terms `HH_SERIAL_RAW_4`.

N3344 aptly demonstrates that field-based date structures are good at some operations and poor at others. And serial-based implementations are generally good at what the field-based data structures are poor at, and poor at what the field-based data structures are good at. Indeed, this is the genesis of the design of this library: Provide both data structures and let the conversions among the data structures be provided by algorithms as shown in `chrono`-Compatible Low-Level Date Algorithms. And additionally, just provide the API for each data structure that it can do efficiently.

- Field types are good at returning the values of the fields. Serial types aren't (except for weekdays). So `year_month_day` has accessors for `year`, `month` and `day`. And `sys_days` does not.
- Field types are good at month and year-oriented arithmetic. Serial types aren't. So `year_month_day` has month and year-oriented arithmetic. And `sys_days` does not.
- Serial types are good at day-oriented arithmetic. Field types aren't. So `sys_days` has day-oriented arithmetic. And `year_month_day` does not. Though one can perform day-oriented arithmetic on the day field of a `year_month_day`, with no impact on the other fields.
- To efficiently compute a day of the week, one first needs to compute a serial date. So `weekday` is constructible from `sys_days`.

To demonstrate the efficiency of constructing a `year_month_day`, a hypothetical factory function has been created in the table below. And this is compared with a simplistic `struct YMD_4` and an obvious factory function for that. Each is compiled using clang at an optimization of -O2 or higher.

`year_month_day` constructor assembly

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

```
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)};
}
```

```
        .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
```

One can see that the generated assembler is virtually identical for these two factory functions. I.e. the code size and run time overhead of the "cute syntax" for constructing a `year_month_day` is zero, at least in optimized builds.

A similar experiment is made for constructing a `sys_days` from a count of days held in an `int`. To do this one must first create a `days` duration, and then construct the `sys_days` from the `days` duration. This is contrasted with the very simplistic `struct SERIAL_4`.

`sys_days` constructor assembly

```
date::sys_days
make_day_point(int z)
{
    using namespace date;
    return sys_days{days{z}};
}
```

```
struct SERIAL_4
{
    std::int32_t count;
};

SERIAL_4
make_SERIAL_4(int z)
{
    return {z};
}
```

```
        .globl  __Z14make_day_pointi
        .align  4, 0x90
__Z14make_day_pointi:
        .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
        movl    %edi, %eax
        popq    %rbp
        retq
        .cfi_endproc
```

```
        .globl  __Z13make_SERIAL_4i
        .align  4, 0x90
__Z13make_SERIAL_4i:
        .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
        movl    %edi, %eax
        popq    %rbp
        retq
        .cfi_endproc
```

It is easy to see that the generated code is identical, and thus there is no overhead associated with the `sys_days` type. It is also noteworthy that the code for this construction does not actually come from this `date` library, but instead comes from your std::lib header `<chrono>`. `days` is nothing but a *type-alias* for a `std::chrono::duration`, and `sys_days` is nothing but a *type-alias* for a `std::chrono::time_point` (thus the inspiration for the name `sys_days`). So this is also evidence that there is zero overhead for the type-safe system of the `<chrono>` library.

One final example taken from real-world code. Below are two ways of implementing a system which counts seconds since Jan. 1, 2000: One that uses this library and another that simply stores the count of seconds in a `long`. A utility is needed to convert that count to Unix Time, which simply involves shifting the epoch to Jan. 1, 1970.

Shift epoch from `January/1/2000` to `January/1/1970`

```
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 + (sys_days(January/1/2000) - sys_days(January/1/1970));
}
```

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

```
        .globl  __Z11shift_epochNSt3__16chrono10time_pointINS0...
        .align  4, 0x90
__Z11shift_epochNSt3__16chrono10time_pointINS0...
        .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
```

On the left we have a strongly typed solution which makes it explicit that the code is adding a time duration equal to the difference between the two epochs. Though verbose, the code is easy to read. On the right we have the maximally efficient, and yet relatively cryptic implementation with `long`. As it turns out, there are 946,684,800 seconds between these two epochs ... who knew?

clang generates *identical* assembly for these two functions at -O2 and higher. That is, the sub-expression `(sys_days(January/1/2000) - sys_days(January/1/1970))` is reduced down to a number of days (10,957) *at compile time*, and then this sub-expression, which has type `days` is added to a `time_point` with a resolution of `seconds`, which causes the compiler to convert `days{10957}` to `seconds{946684800}`, again *at compile time*.

And in the strongly typed case, if in the future one decides that one wants to count milliseconds instead of seconds, that can be changed in **one** place:

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

Upon recompile the compiler automatically updates the compile-time constant in `shift_epoch` from 946684800 to 946684800000 (and everywhere else such as where you perform the epoch shift in the reverse direction).

What about a date-time type?

You already have a good one. It is called `std::chrono::system_clock::time_point`. And it is completely interoperable with this library. `std::chrono::system_clock::time_point` is a pointer to a particular instant of time, with high precision (that precision is specified differently on each platform). As we saw earlier, one can get a high-precision `time_point` representing *now* with:

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

This library provides a stream insertion operator for `system_clock::time_point` (in `namespace date`).

```
std::cout << tp << '\n';
```

This currently outputs for me:

```
2015-04-19 18:24:42.770911
```

One can turn that into a `time_point` with the precision of a day (which has a type alias called `sys_days`) with:

```
auto dp = floor<days>(tp);
```

And one can convert that day-oriented `time_point` into a `{year, month, day}` field type with:

```
    auto ymd = year_month_day{dp};
```

This section explains how to extract the *time of day* from the above information. `dp` can also be though of as a `time_point` to the beginning of the current day (in the UTC timezone). And so one can get the time `duration` since midnight by subtracting the day-precision `time_point` from the high precision `time_point`: `tp-dp`. This results in a high precision `std::chrono::duration` representing the time since midnight.

This high precision `duration` can be broken down into hours, minutes, seconds, and fractions of a second with a `time_of_day` object, which is most easily created by using the `make_time` factory function:

```
    auto time = make_time(tp-dp);
```

Now both `ymd` and `time` can simply be printed out:

```
    std::cout << ymd << ' ' << time << '\n';
```

Which will result in something like:

```
    2015-04-19 18:24:42.770911
```

Indeed, this is exactly what the stream insertion operator for `system_clock::time_point` does. So if you don't like the default format, this is how you pick apart a `system_clock::time_point` so you can output it in whatever format you prefer.

The above is the current date and time in the UTC timezone with microsecond precision. On the platform I'm writing this, `system_clock::time_point` counts microseconds. When you run this code on your platform, the library will automatically adapt to the precision supported by your platform's implementation of `system_clock::time_point`.

If you prefer output in a local time zone, you will need to discover your current UTC offset and add that to `tp` prior to applying `floor`. Your current UTC offset is a function of both your UTC time point `tp`, and your location. This functionality is not offered by this library, but can be built on top of it. For example here is such a library. But as a simplistic example, here is code that assumes an Eastern US location and daylight savings rules currently in effect:

```
    std::chrono::hours
    utc_offset_Eastern_US(std::chrono::system_clock::time_point tp)
    {
        using namespace date;
        using namespace std::chrono;
        constexpr auto EST = -5h;
        constexpr auto EDT = -4h;
        const auto y = year_month_day{floor<days>(tp)}.year();
        const auto begin = sys_days{Sunday[2]/March/y} + 2h - EST; // EDT begins at this UTC time
        const auto end   = sys_days{Sunday[1]/November/y} + 2h - EDT; // EST begins at this UTC time
        if (tp < begin || end <= tp)
            return EST;
        return EDT;
    }
    ...
    auto tp = system_clock::now();
    tp += utc_offset_Eastern_US(tp);                // tp now in Eastern US timezone
    std::cout << tp << '\n';

    2015-04-19 14:24:42.770911
```

And if you prefer your output in a 12-hour format with a precision of only minutes, that is also easily accomplished:

```
    auto tp = system_clock::now();
    tp += utc_offset_Eastern_US(tp);
    const auto tpm = floor<minutes>(tp);            // truncate to minutes precision
    const auto dp = floor<days>(tpm);
    const auto ymd = year_month_day{dp};
    auto time = make_time(tpm-dp);                  // minutes since midnight
    time.make12();                                  // change to 12-hour format
    std::cout << ymd << ' ' << time << '\n';

    2015-04-19 2:24pm
```

Extensibility

The hub of this library is `sys_days`. This is a *serial-based* time point which simply counts the days since (or before) `January/1/1970`. And ironically this all important hub is nothing but a type alias to a std-defined type. That is, the central theme this library is built around is nothing more than this:

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

Types such as `year_month_day` and `year_month_weekday` provide implicit conversions to and from `sys_days`, and because of this, the C++ language provides explicit conversions between `year_month_day` and `year_month_weekday`.

You can easily build your own types that implicitly convert to and from `sys_days`, and when you do, you automatically gain explicit convertibility to and from every other type which ties into `sys_days`. For example, here is how you could create a custom type that models the ISO week-based calendar:

```
class iso_week
{
    date::year    y_;
    date::weeks   w_;
    date::weekday wd_;

public:
    constexpr iso_week(date::sys_days dp) noexcept
        : iso_week(iso_week_from_day_point(dp))
        {}

    constexpr iso_week(date::year y, date::weeks w, date::weekday wd) noexcept
        : y_(y)
        , w_(w)
        , wd_(wd)
        {}

    constexpr operator date::sys_days() const noexcept
    {
        using namespace date;
        return iso_week_start(y_) + w_ - weeks{1} + (wd_ - Monday);
    }

    friend std::ostream& operator<<(std::ostream& os, const iso_week& x)
    {
        return os << x.y_ << "-W(" << x.w_.count() << ")-" << x.wd_;
    }

private:
    static
    constexpr
    date::sys_days
    iso_week_start(date::year y) noexcept
    {
        using namespace date;
        return sys_days{Thursday[1]/January/y} - (Thursday-Monday);
    }

    static
    constexpr
    iso_week
    iso_week_from_day_point(date::sys_days dp) noexcept
    {
        using namespace date;
        using namespace std::chrono;
        auto y = year_month_day{dp}.year();
        auto start = iso_week_start(y);
        if (dp < start)
        {
            --y;
            start = iso_week_start(y);
        }
        else
        {
            auto const next_start = iso_week_start(y+years{1});
            if (dp >= next_start)
            {
                ++y;
                start = next_start;
            }
        }
        return {y, duration_cast<weeks>(dp - start) + weeks{1}, weekday{dp}};
    }
};
```

The rules for the ISO week-based calendar are fairly simple:

1. Weeks begin with Monday and end on Sunday.
2. The year begins on the Monday prior to the first Thursday in January.
3. A date is specified by the week number [1 - 53], day of the week [Mon - Sun], and year number.
4. The year number is the same as the civil year number for the Thursday that follows the start of the week-based year.

With that in mind, one can easily create a *field-based* data structure that holds a `year`, a week number, and a day of the week, and then provides conversions to and from `sys_days`.

The key points of this class (for interoperability) are the constructor `iso_week(date::sys_days dp)` and the `operator date::sys_days()`.

To aid in these computations a private helper function is created to compute the `sys_days` corresponding to the first day of the week-based year. And according to rule 2, this can be elegantly coded as:

```
return sys_days{Thursday[1]/January/y} - (Thursday-Monday);
```

That is, first find the first Thursday in January for year `y`, and then subtract the number of days required to find the Monday before this day. This could have been done with `days{3}`. But I chose to code this as `(Thursday-Monday)`. Computationally and performance wise, these two choices are identical: they both subtract the literal 3. I chose the latter because I believe it to be more readable. "`3`" is just a magic constant. But "`(Thursday-Monday)`" is the number of days Thursday is past Monday.

The constructor `iso_week(date::sys_days dp)` has to first discover which ISO week-based year the `sys_days dp` falls into. Most often this is the same as the civil (`year_month_day`) year number associated `dp`. But because the week-based year may start a few days earlier or later than `January/1`, the week-based year number may be one less or one greater than the civil year number associated `dp`. Once the proper start of the week-based year is nailed down (in `start`), the translation to the field-based `iso_week` is trivial:

```
return {y, duration_cast<weeks>(dp - start) + weeks{1}, weekday{dp}};
```

The conversion from `iso_week` to `sys_days` is even easier:

```
return iso_week_start(y_) + w_ - weeks{1} + (wd_ - Monday);
```

It is the the start of the week-based year, plus the number of weeks (minus one since this count is 1-based), plus the number of days the `weekday` is past Monday. Note that because `weekday` subtraction is treated as a circular range (always results in a number of days in the range [0, 6]), the logic at this level is simplified. That is, Sunday is 6 days past Monday, not one day before it (and Monday is still one day past Sunday). So the encoding used for `weekday` is irrelevant; safely encapsulated within `weekday`. Said differently, `weekday` arithmetic is unsigned, modulo 7.

With the above code, one can now write programs such as the one below which demonstrates easy convertibility among the `date` *field-types* and your custom *field-type*.

```
int
main()
{
    using namespace date;
    using namespace std;
    using namespace std::chrono;
    auto dp = floor<days>(system_clock::now());
    auto ymd = year_month_day{dp};
    cout << ymd << '\n';
    auto iso = iso_week{ymd};
    cout << iso << '\n';
    auto ymwd = year_month_weekday{iso};
    cout << ymwd << '\n';
    assert(year_month_day{iso} == ymd);
}
```

Which will output something like:

```
2015-05-20
2015-W(21)-Wed
2015/May/Wed[3]
```

As long as you can relate your custom *field-based* structure (be it the Julian calendar, the Hindu calendar, or the Maya calendar) to the number of days before and after civil 1970-01-01, you can achieve interoperability with every other *field-based* structure that does so. And obviously you can then also convert your custom calendar to UTC (`std::chrono::system_clock::time_point`). *This is the Rosetta Stone of time keeping.*

For an example of a fully developed ISO week date calendar which is fully interoperable with this library via the technique described above see `iso_week`.

Range of Validity

As with all numerical representations with a fixed storage size, `durations`, `time_points`, and `year_month_days` have a fixed range, outside of which they overflow. With this library, and with `<chrono>`, the range varies with precision.

On one side `nanoseconds` is represented by a `int64_t` which has a range of about +/- 292 years. And on the other side `year` is represented by a `int16_t` which has a range of about +/- 32 thousand years. It is informative and educational to write software which explores the intersection of these two constraints for various precisions, and outputs the result in terms of a `sys_time`.

Here is a function which will discover the limits for a single durration `D`:

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

template <class D>
void
limit(const std::string& msg)
{
    using namespace date;
    using namespace std;
    using namespace std::chrono;
    using dsecs = sys_time<duration<double>>;
    constexpr auto ymin = sys_days{year::min()/January/1};
    constexpr auto ymax = sys_days{year::max()/12/last};
    constexpr auto dmin = sys_time<D>::min();
    constexpr auto dmax = sys_time<D>::max();
    cout << left << setw(24) << msg << " : [";
    if (ymin > dsecs{dmin})
        cout << ymin;
    else
        cout << dmin;
    cout << ", ";
    if (ymax < dsecs{dmax})
        cout << ymax;
    else
        cout << dmax;
    cout << "]\n";
}
```

The best way to explore limits without risking overflow during the comparison operation itself is to use `double`-based `seconds` for the comparison. By using `seconds` you guarantee that the conversion to the comparison type won't overflow the compile-time machinery of finding the `common_type` of the `durations`, and by using `double` you make overflow or underflow nearly impossible. The use of `double` sacrifices precision, but this is rarely needed for limits comparisons as the two operands of the comparison are typically orders of magnitude apart.

So the code above creates a `sys_time<double>` `time_point` with which to perform the comparisons. Then it finds the min and max of both the `year_month_day` object, and the duration `D`. It then prints out the intersection of these two ranges.

This code can be exercised like so:

```
void
limits()
{
    using namespace std::chrono;
    using namespace std;
    using picoseconds = duration<int64_t, pico>;
    using fs = duration<int64_t, ratio_multiply<ratio<100>, nano>>;
    limit<picoseconds>("picoseconds range is");
    limit<nanoseconds>("nanoseconds range is");
    limit<fs>("VS system_clock range is");
    limit<microseconds>("microseconds range is");
    limit<milliseconds>("milliseconds range is");
    limit<seconds>("seconds range is");
    limit<minutes>("minutes range is");
    limit<hours>("hours range is");
}
```

I've included two extra units: `picoseconds`, and the units used by Visual Studio's `system_clock::time_point`. Units finer than `picoseconds` do not work with this date library because the conversion factors needed to convert to units such as `days` overflow the compile-time machinery. As a practical matter this is not important as the range of a 64 bit femtosecond is only about +/- 2.5 hours. On the other side, units coarser than `hours`, if represented by at least 32 bits, will always have a range far greater than a 16 bit `year`.

The output of this function on Visual Studio, and on clang using libc++ with `-arch i386` is:

```
picoseconds range is     : [1969-09-16 05:57:07.963145224192, 1970-04-17 18:02:52.036854775807]
nanoseconds range is     : [1677-09-21 00:12:43.145224192, 2262-04-11 23:47:16.854775807]
VS system_clock range is : [-27258-04-19 21:11:54.5224192, 31197-09-14 02:48:05.4775807]
microseconds range is    : [-32768-01-01, 32767-12-31]
milliseconds range is    : [-32768-01-01, 32767-12-31]
```

```
seconds range is         : [-32768-01-01, 32767-12-31]
minutes range is         : [-2114-12-08 21:52, 6053-01-23 02:07]
hours range is           : [-32768-01-01, 32767-12-31]
```

Using gcc or clang/libc++ with `-arch x86_64` the output is:

```
picoseconds range is     : [1969-09-16 05:57:07.963145224192, 1970-04-17 18:02:52.036854775807]
nanoseconds range is     : [1677-09-21 00:12:43.145224192, 2262-04-11 23:47:16.854775807]
VS system_clock range is : [-27258-04-19 21:11:54.5224192, 31197-09-14 02:48:05.4775807]
microseconds range is    : [-32768-01-01, 32767-12-31]
milliseconds range is    : [-32768-01-01, 32767-12-31]
seconds range is         : [-32768-01-01, 32767-12-31]
minutes range is         : [-32768-01-01, 32767-12-31]
hours range is           : [-32768-01-01, 32767-12-31]
```

The only difference between these two outputs is that associated with `minutes`. When `minutes` is represented with 32 bits the range is only about +/- 4000 years from 1970. When `minutes` is represented with 64 bits, the limits of the 16 bit year takes precedence.

The take-away point here is two-fold:

1. If you need to check range, do the check using `duration<double>` to ensure your comparison is not itself vulnerable to overflow.
2. If you are dealing units finer than `microseconds`, you may well accidentally experience overflow in surprisingly mundane-looking code. When dealing with dates that may be hundreds of years away from 1970, keep an eye on the precision. And in a surprise move, 32 bit `minutes` can bite if you are several thousand years away from 1970.

Finally note that the civil calendar itself models the rotation and orbit of the earth with an accuracy of only one day in several thousand years. So dates more than several thousand years in the past or future (with a precision of a single day) are of limited practical use with or without numerical overflow. The chief motivation for having large ranges of date computation before overflow happens is to make range checking superflous for most reasonable computations. If you need to handle ranges dealing with geological or astrophysical phenomenon, `<chrono>` can handle it (`attoseconds` to `exaseconds`), but `year_month_day` is the wrong tool for such extremes.

Reference

Here is a detailed specification of the entire library. This specification is detailed enough that you could write your own implementation from it if desired. But feel free to use this one instead. Each type, and each operation is simple and predictable.

```
month_day
month_day_last
month_weekday
month_weekday_last

year_month

year_month_day
year_month_day_last
year_month_weekday
year_month_weekday_last

time_of_day
```

conventional syntax operators

```
constexpr year_month operator/(const year& y, const month& m) noexcept;
constexpr year_month operator/(const year& y, int   m) noexcept;

constexpr month_day operator/(const month& m, const day& d) noexcept;
constexpr month_day operator/(const month& m, int d) noexcept;
constexpr month_day operator/(int   m, const day& d) noexcept;
constexpr month_day operator/(const day& d, const month& m) noexcept;
constexpr month_day operator/(const day& d,   int m) noexcept;

constexpr month_day_last operator/(const month& m, last_spec) noexcept;
constexpr month_day_last operator/(int   m, last_spec) noexcept;
constexpr month_day_last operator/(last_spec, const month& m) noexcept;
constexpr month_day_last operator/(last_spec, int   m) noexcept;

constexpr month_weekday operator/(const month& m, const weekday_indexed& wdi) noexcept;
constexpr month_weekday operator/(int   m, const weekday_indexed& wdi) noexcept;
constexpr month_weekday operator/(const weekday_indexed& wdi, const month& m) noexcept;
constexpr month_weekday operator/(const weekday_indexed& wdi, int   m) noexcept;

constexpr month_weekday_last operator/(const month& m, const weekday_last& wdl) noexcept;
constexpr month_weekday_last operator/(int   m, const weekday_last& wdl) noexcept;
constexpr month_weekday_last operator/(const weekday_last& wdl, const month& m) noexcept;
constexpr month_weekday_last operator/(const weekday_last& wdl, int   m) noexcept;

constexpr year_month_day operator/(const year_month& ym, const day& d) noexcept;
constexpr year_month_day operator/(const year_month& ym, int d) noexcept;
constexpr year_month_day operator/(const year& y, const month_day& md) noexcept;
constexpr year_month_day operator/(int  y, const month_day& md) noexcept;
constexpr year_month_day operator/(const month_day& md, const year& y) noexcept;
constexpr year_month_day operator/(const month_day& md, int  y) noexcept;

constexpr year_month_day_last operator/(const year_month& ym,    last_spec) noexcept;
constexpr year_month_day_last operator/(const year& y, const month_day_last& mdl) noexcept;
constexpr year_month_day_last operator/(int  y, const month_day_last& mdl) noexcept;
constexpr year_month_day_last operator/(const month_day_last& mdl, const year& y) noexcept;
constexpr year_month_day_last operator/(const month_day_last& mdl, int  y) noexcept;

constexpr year_month_weekday operator/(const year_month& ym, const weekday_indexed& wdi) noexcept;
constexpr year_month_weekday operator/(const year&        y, const month_weekday&   mwd) noexcept;
constexpr year_month_weekday operator/(int              y, const month_weekday&   mwd) noexcept;
constexpr year_month_weekday operator/(const month_weekday& mwd, const year&          y) noexcept;
constexpr year_month_weekday operator/(const month_weekday& mwd, int                  y) noexcept;

constexpr year_month_weekday_last operator/(const year_month& ym, const weekday_last& wdl) noexcept;
constexpr year_month_weekday_last operator/(const year& y, const month_weekday_last& mwdl) noexcept;
constexpr year_month_weekday_last operator/(int   y, const month_weekday_last& mwdl) noexcept;
constexpr year_month_weekday_last operator/(const month_weekday_last& mwdl, const year& y) noexcept;
constexpr year_month_weekday_last operator/(const month_weekday_last& mwdl, int   y) noexcept;
```

`time_of_day`
factory function

```
template <class Rep, class Period>
constexpr
time_of_day<std::chrono::duration<Rep, Period>>
make_time(std::chrono::duration<Rep, Period> d) noexcept;
```

convenience
streaming operators

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

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

`to_stream` formatting
`format`
`from_stream` formatting
`parse`

Everything here is contained in the namespace `date`. The literal operators, and the constexpr field literals (e.g. `Sunday`, `January`, etc.) are in namespace `date::literals` and imported into namespace `date`.

days

`days` is a `std::chrono::duration` with a tick period of 24 hours. This definition is not an SI unit but is accepted for use with SI. `days` is the resultant type when subtracting two `sys_days`s.

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

weeks

`weeks` is a `std::chrono::duration` with a tick period of 7 days. This definition is widely recognized and predates the Gregorian calendar. It is consistent with ISO 8601. `weeks` will implicitly convert to `days` but not vice-versa.

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

years

`years` is a `std::chrono::duration` with a tick period of 365.2425 days. This definition accurately describes the length of the average year in the Gregorian calendar. `years` is the resultant type when subtracting two `year` field-based time points. `years` is not implicitly convertible to `days` or `weeks` nor vice-versa. However `years` will implicitly convert to `months`.

```
using years = std::chrono::duration
    <int, std::ratio_multiply<std::ratio<146097, 400>, days::period>>;
```

months

`months` is a `std::chrono::duration` with a tick period of $^1/_{12}$ of a year. This definition accurately describes the length of the average month in the Gregorian calendar. `months` is the resultant type when subtracting two `month` field-based time points. `months` is not implicitly convertible to `days` or `weeks` nor vice-versa. `months` will not implicitly convert to `years`.

```
using months = std::chrono::duration
    <int, std::ratio_divide<years::period, std::ratio<12>>>;
```

Formatting and parsing durations

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

Effects: Inserts `d`, converted to the `common_type<duration, seconds>` into `os` using the format string `fmt` as specified by the `to_stream` formatting flags. The behavior is undefined except for the following flags (or modified versions of these flags): `%H`, `%I`, `%M`, `%p`, `%r`, `%R`, `%S`, `%T`, `%X`, `%n`, `%t` or `%%`.

Returns: `os`.

```
template <class Rep, class Period, class CharT, class Traits, class Alloc = std::allocator<CharT>>
std::basic_istream<CharT, Traits>&
from_stream(std::basic_istream<CharT, Traits>& is, const CharT* fmt,
            std::chrono::duration<Rep, Period>& d,
            std::basic_string<CharT, Traits, Alloc>* abbrev = nullptr,
            std::chrono::minutes* offset = nullptr);
```

Effects: Extracts `d` from `is` using the format string `fmt` as specified by the `from_stream` formatting flags. The behavior is undefined except for the following flags (or modified versions of these flags): `%H`, `%I`, `%M`, `%p`, `%r`, `%R`, `%S`, `%T`, `%X`, `%n`, `%t`, `%z`, `%Z` or `%%`. If `abbrev` is not equal to `nullptr`, the information parsed by `%Z` (if present) will be placed in `*abbrev`. If `offset` is not equal to `nullptr`, the information parsed by `%z` (if present) will be placed in `*offset`. If an invalid date is parsed, or a time of day which is outside of the conventional bounds, `failbit` will be set and `d` will not be altered.

Returns: `is`.

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

Effects: Equivalent to:

```
os << to_string<CharT, Traits>(d.count()) + detail::get_units<CharT>(duration<Rep, typename Period::type>{});
```

Where `to_string` is pseudo code that returns a `std::basic_string<CharT, Traits>` representation of `d.count()`, and `detail::get_units<CharT>()` returns a null-terminated string of `CharT` which depends only on `Period::type` as follows (let `period` be the type `Period::type`):

- If `period` is type `std::atto`, `as`, else
- if `period` is type `std::femto`, `fs`, else
- if `period` is type `std::pico`, `ps`, else
- if `period` is type `std::nano`, `ns`, else
- if `period` is type `std::micro`, `µs` (U+00B5), else
- if `period` is type `std::milli`, `ms`, else
- if `period` is type `std::centi`, `cs`, else
- if `period` is type `std::deci`, `ds`, else
- if `period` is type `std::ratio<1>`, `s`, else
- if `period` is type `std::deca`, `das`, else
- if `period` is type `std::hecto`, `hs`, else
- if `period` is type `std::kilo`, `ks`, else
- if `period` is type `std::mega`, `Ms`, else
- if `period` is type `std::giga`, `Gs`, else
- if `period` is type `std::tera`, `Ts`, else
- if `period` is type `std::peta`, `Ps`, else
- if `period` is type `std::exa`, `Es`, else
- if `period` is type `std::ratio<60>`, `min`, else
- if `period` is type `std::ratio<3600>`, `h`, else
- if `period::den == 1`, `[num]s`, else
- `[num/den]s`.

In the list above the use of num and den refer to the static data members of period which are converted to arrays of CharT using a decimal conversion with no leading zeroes.

Returns: os.

sys_time

sys_time is a convenience template alias for creating a system_clock::time_point but of arbitrary precision. This family of types represents "system time", which closely models UTC. See utc_time in tz.h for a variation of sys_time that accurately takes leap seconds into account when subtracting time_points.

```
template <class Duration>
    using sys_time = std::chrono::time_point<std::chrono::system_clock, Duration>;
```

sys_days

sys_days is a std::chrono::time_point using std::chrono::system_clock and days. This makes sys_days interoperable with std::chrono::system_clock::time_point. It is simply a count of days since the epoch of std::chrono::system_clock which in every implementation is Jan. 1, 1970. sys_days is a serial-based time point with a resolution of days.

```
using sys_days = sys_time<days>;
```

sys_seconds

sys_seconds is a std::chrono::time_point using std::chrono::system_clock and std::chrono::seconds. This makes sys_seconds interoperable with std::chrono::system_clock::time_point. It is simply a count of *non-leap* seconds since the epoch of std::chrono::system_clock which in every implementation is Jan. 1, 1970 00:00:00 UTC. sys_seconds is a serial-based time point with a resolution of seconds. sys_seconds is also widely known as Unix Time.

```
using sys_seconds = sys_time<std::chrono::seconds>;
```

```
template <class CharT, class Traits, class Duration>
std::basic_ostream<CharT, Traits>&
to_stream(std::basic_ostream<CharT, Traits>& os, const CharT* fmt,
          const sys_time<Duration>& tp);
```

Effects: Inserts tp, converted to the common_type<Duration, seconds> into os using the format string fmt as specified by the to_stream formatting flags. If %Z is in the formatting string "UTC" will be used. If %z is in the formatting string "+0000" will be used.

Returns: os.

```
template <class Duration, class CharT, class Traits, class Alloc = std::allocator<CharT>>
std::basic_istream<CharT, Traits>&
from_stream(std::basic_istream<CharT, Traits>& is, const CharT* fmt,
            sys_time<Duration>& tp, std::basic_string<CharT, Traits, Alloc>* abbrev = nullptr,
            std::chrono::minutes* offset = nullptr);
```

Effects: Extracts tp from is using the format string fmt as specified by the from_stream formatting flags. If %z is present, the parsed offset will be subtracted from the parsed time. If abbrev is not equal to nullptr, the information parsed by %Z (if present) will be placed in *abbrev. If offset is not equal to nullptr, the information parsed by %z (if present) will be placed in *offset. If an invalid date is parsed, or a time of day which is outside of the conventional bounds, failbit will be set and tp will not be altered.

Returns: is.

local_time

local_time is a convenience template alias for creating a time_point of arbitrary precision which is not based on any clock at all. This family of types represents a time not associated with any time zone. It is handy in disambiguating calendar timestamps referring to an unspecified timezone, and those referring to UTC.

For example, we can say that the upcoming 2017 New Years will be commonly celebrated at local_time<days>{2017_y/January/1} + 0s. For those in a time zone with a zero offset from UTC, it will be celebrated at the concrete time of sys_days{2017_y/January/1} + 0s. These two timestamps have different types, though both have the exact same representation (a count of seconds), because they mean two *subtly* different things, and are both *quite* useful.

```
struct local_t {};
```

```
template <class Duration>
    using local_time = std::chrono::time_point<local_t, Duration>;
```

`local_days`

`local_days` is a convient way to write `local_time<days>`. The upcoming 2017 New Years will be commonly celebrated at `local_days{2017_y/January/1} + 0s`.

```
using local_days = local_time<days>;
```

`local_seconds`

`local_seconds` is a convient way to write `local_time<seconds>`.

```
using local_seconds = local_time<std::chrono::seconds>;
```

```
template <class CharT, class Traits, class Duration>
std::basic_ostream<CharT, Traits>&
to_stream(std::basic_ostream<CharT, Traits>& os, const CharT* fmt,
          const local_time<Duration>& tp, const std::string* abbrev = nullptr,
          const std::chrono::seconds* offset_sec = nullptr);
```

Effects: Inserts `tp`, converted to the `common_type<Duration, seconds>` into `os` using the format string `fmt` as specified by the [`to_stream` formatting flags](). If `%Z` is in the formatting string `*abbrev` will be used. If `%z` is in the formatting string `*offset_sec` will be used.

If `%Z` is in the formatting string and `abbrev == nullptr`, or if `%z` is in the formatting string and `offset_sec == nullptr`, `failbit` will be set for `os`.

Returns: `os`.

```
template <class Duration, class CharT, class Traits, class Alloc = std::allocator<CharT>>
std::basic_istream<CharT, Traits>&
from_stream(std::basic_istream<CharT, Traits>& is, const CharT* fmt,
            local_time<Duration>& tp, std::basic_string<CharT, Traits, Alloc>* abbrev = nullptr,
            std::chrono::minutes* offset = nullptr);
```

Effects: Extracts `tp` from `is` using the format string `fmt` as specified by the [`from_stream` formatting flags](). If `abbrev` is not equal to `nullptr`, the information parsed by `%Z` (if present) will be placed in `*abbrev`. If `offset` is not equal to `nullptr`, the information parsed by `%z` (if present) will be placed in `*offset`. If an invalid date is parsed, or a time of day which is outside of the conventional bounds, `failbit` will be set and `tp` will not be altered.

Returns: `is`.

`last_spec`

`last_spec` is a `struct` that is `CopyConstructible`. There exists a `constexpr` instance of `last_spec` named `last`. This is simply a tag type. It is used to indicate the last day of a month, or the last weekday of a month.

```
struct last_spec
{
    explicit last_spec() = default;
};
inline namespace literals
{
constexpr last_spec last{};
}
```

`day`

Synopsis

```
class day
{
    unsigned char d_;  // exposition only
public:
    day() = default;
    explicit constexpr day(unsigned d) noexcept;

    constexpr day& operator++() noexcept;
    constexpr day operator++(int) noexcept;
    constexpr day& operator--() noexcept;
    constexpr day operator--(int) noexcept;

    constexpr day& operator+=(const days& d) noexcept;
    constexpr day& operator-=(const days& d) noexcept;
    constexpr explicit operator unsigned() const noexcept;
    constexpr bool ok() const noexcept;
};

constexpr bool operator==(const day& x, const day& y) noexcept;
constexpr bool operator!=(const day& x, const day& y) noexcept;
constexpr bool operator< (const day& x, const day& y) noexcept;
constexpr bool operator> (const day& x, const day& y) noexcept;
constexpr bool operator<=(const day& x, const day& y) noexcept;
constexpr bool operator>=(const day& x, const day& y) noexcept;

constexpr day  operator+(const day&  x, const days& y) noexcept;
constexpr day  operator+(const days& x, const day&  y) noexcept;
constexpr day  operator-(const day&  x, const days& y) noexcept;
constexpr days operator-(const day&  x, const day&  y) noexcept;

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

template<class CharT, class Traits>
std::basic_ostream<CharT, Traits>&
to_stream(std::basic_ostream<class CharT, class Traits>& os, const CharT* fmt,
          const day& d);

template <class CharT, class Traits, class Alloc = std::allocator<CharT>>
std::basic_istream<CharT, Traits>&
from_stream(std::basic_istream<CharT, Traits>& is, const CharT* fmt, day& d,
            std::basic_string<CharT, Traits, Alloc>* abbrev = nullptr,
            std::chrono::minutes* offset = nullptr);

inline namespace literals {
constexpr day operator "" _d(unsigned long long d) noexcept;
}
```

Overview

`day` represents a day of a month. It should only be representing values in the range 1 to 31. However it may hold values outside this range. It can be constructed with any `unsigned` value, which will be subsequently truncated to fit into `day`'s internal storage. `day` is equality and less-than comparable, and participates in basic arithmetic with `days` representing the quantity between any two `day`'s. One can form a `day` literal with `_d`. And one can stream out a `day` for debugging purposes. `day` has explicit conversions to and from `unsigned`.

Specification

`day` is a trivially copyable class type.
`day` is a standard-layout class type.
`day` is a literal class type.

```
explicit constexpr day::day(unsigned d) noexcept;
```

Effects: Constructs an object of type `day` by constructing `d_` with `d`. The value held is unspecified if `d` is not in the range `[0, 255]`.

```
constexpr day& day::operator++() noexcept;
```

Effects: `++d_`.

Returns: `*this`.

```
constexpr day day::operator++(int) noexcept;
```

Effects: `++(*this)`.

Returns: A copy of `*this` as it existed on entry to this member function.

```
constexpr day& day::operator--() noexcept;
```

Effects: `--d_`.

Returns: `*this`.

```
constexpr day day::operator--(int) noexcept;
```

Effects: `--(*this)`.

Returns: A copy of `*this` as it existed on entry to this member function.

```
constexpr day& day::operator+=(const days& d) noexcept;
```

Effects: `*this = *this + d`.

Returns: `*this`.

```
constexpr day& day::operator-=(const days& d) noexcept;
```

Effects: `*this = *this - d`.

Returns: `*this`.

```
constexpr explicit day::operator unsigned() const noexcept;
```

Returns: `d_`.

```
constexpr bool day::ok() const noexcept;
```

Returns: `1 <= d_ && d_ <= 31`.

```
constexpr bool operator==(const day& x, const day& y) noexcept;
```

Returns: `unsigned{x} == unsigned{y}`.

```
constexpr bool operator!=(const day& x, const day& y) noexcept;
```

Returns: `!(x == y)`.

```
constexpr bool operator< (const day& x, const day& y) noexcept;
```

Returns: `unsigned{x} < unsigned{y}`.

```
constexpr bool operator> (const day& x, const day& y) noexcept;
```

Returns: `y < x`.

```
constexpr bool operator<=(const day& x, const day& y) noexcept;
```

Returns: `!(y < x)`.

```
constexpr bool operator>=(const day& x, const day& y) noexcept;
```

Returns: `!(x < y)`.

```
constexpr day  operator+(const day&  x, const days& y) noexcept;
```

Returns: `day{unsigned{x} + y.count()}`.

```
constexpr day  operator+(const days& x, const day&  y) noexcept;
```

Returns: `y + x`.

```
constexpr day  operator-(const day&  x, const days& y) noexcept;
```

Returns: `x + -y`.

```
constexpr days operator-(const day&  x, const day&  y) noexcept;
```

Returns: `days{static_cast<days::rep>(unsigned{x} - unsigned{y})}`.

```
constexpr day operator "" _d(unsigned long long d) noexcept;
```

Returns: `day{static_cast<unsigned>(d)}`.

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

Effects: Inserts a decimal integral text representation of `d` into `os`. Single digit values are prefixed with `'0'`.

Returns: `os`.

```
template<class CharT, class Traits>
std::basic_ostream<CharT, Traits>&
to_stream(std::basic_ostream<class CharT, class Traits>& os, const CharT* fmt,
          const day& d);
```

Effects: Inserts `d` into `os` using the format string `fmt` as specified by the `to_stream` formatting flags. The behavior is undefined except for the following flags (or modified versions of these flags): `%d`, `%e`, `%n`, `%t` or `%%`.

Returns: `os`.

```
template <class CharT, class Traits, class Alloc = std::allocator<CharT>>
std::basic_istream<CharT, Traits>&
from_stream(std::basic_istream<CharT, Traits>& is, const CharT* fmt, day& d,
            std::basic_string<CharT, Traits, Alloc>* abbrev = nullptr,
            std::chrono::minutes* offset = nullptr);
```

Effects: Extracts `d` from `is` using the format string `fmt` as specified by the `from_stream` formatting flags. The behavior is undefined except for the following flags (or modified versions of these flags): `%e`, `%e`, `%n`, `%t`, `%z`, `%Z` or `%%`. If `abbrev` is not equal to `nullptr`, the information parsed by `%Z` (if present) will be placed in `*abbrev`. If `offset` is not equal to `nullptr`, the information parsed by `%z` (if present) will be placed in `*offset`. If an invalid date is parsed, or a time of day which is outside of the conventional bounds, `failbit` will be set and `d` will not be altered. In particular `!d.ok()` is not an indication of error.

Returns: `is`.

`month`

Synopsis

```
class month
{
    unsigned char m_;  // exposition only
public:
    month() = default;
    explicit constexpr month(unsigned m) noexcept;

    constexpr month& operator++() noexcept;
    constexpr month operator++(int) noexcept;
    constexpr month& operator--() noexcept;
    constexpr month operator--(int) noexcept;

    constexpr month& operator+=(const months& m) noexcept;
    constexpr month& operator-=(const months& m) noexcept;

    constexpr explicit operator unsigned() const noexcept;
    constexpr bool ok() const noexcept;
};

constexpr bool operator==(const month& x, const month& y) noexcept;
constexpr bool operator!=(const month& x, const month& y) noexcept;
constexpr bool operator< (const month& x, const month& y) noexcept;
constexpr bool operator> (const month& x, const month& y) noexcept;
constexpr bool operator<=(const month& x, const month& y) noexcept;
constexpr bool operator>=(const month& x, const month& y) noexcept;

constexpr month  operator+(const month&  x, const months& y) noexcept;
constexpr month  operator+(const months& x,  const month& y) noexcept;
constexpr month  operator-(const month&  x, const months& y) noexcept;
constexpr months operator-(const month&  x,  const month& y) noexcept;

template<class CharT, class Traits>
std::basic_ostream<class CharT, class Traits>&
operator<<(std::basic_ostream<class CharT, class Traits>& os, const month& m);

template<class CharT, class Traits>
std::basic_ostream<CharT, Traits>&
to_stream(std::basic_ostream<class CharT, class Traits>& os, const CharT* fmt,
          const month& m);

template <class CharT, class Traits, class Alloc = std::allocator<CharT>>
std::basic_istream<CharT, Traits>&
from_stream(std::basic_istream<CharT, Traits>& is, const CharT* fmt, month& m,
            std::basic_string<CharT, Traits, Alloc>* abbrev = nullptr,
            std::chrono::minutes* offset = nullptr);
```

```
inline namespace literals {
constexpr month January{1};
constexpr month February{2};
constexpr month March{3};
constexpr month April{4};
constexpr month May{5};
constexpr month June{6};
constexpr month July{7};
constexpr month August{8};
constexpr month September{9};
constexpr month October{10};
constexpr month November{11};
constexpr month December{12};
}
```

Overview

`month` represents a month of a year. It should only be representing values in the range 1 to 12. However it may hold values outside this range. It can be constructed with any `unsigned` value, which will be subsequently truncated to fit into `month`'s internal storage. `month` is equality and less-than comparable, and participates in basic arithmetic with `months` representing the quantity between any two `month`'s. One can stream out a `month` for debugging purposes. `month` has explicit conversions to and from `unsigned`. There are 12 `month` constants, one for each month of the year.

Specification

`month` is a trivially copyable class type.
`month` is a standard-layout class type.
`month` is a literal class type.

```
explicit constexpr month::month(unsigned m) noexcept;
```

> *Effects:* Constructs an object of type `month` by constructing `m_` with `m`. The value held is unspecified if `m` is not in the range `[0, 255]`.

```
constexpr month& month::operator++() noexcept;
```

> *Effects:* If `m_ != 12`, `++m_`. Otherwise sets `m_` to 1.
>
> *Returns:* `*this`.

```
constexpr month month::operator++(int) noexcept;
```

> *Effects:* `++(*this)`.
>
> *Returns:* A copy of `*this` as it existed on entry to this member function.

```
constexpr month& month::operator--() noexcept;
```

> *Effects:* If `m_ != 1`, `--m_`. Otherwise sets `m_` to 12.
>
> *Returns:* `*this`.

```
constexpr month month::operator--(int) noexcept;
```

> *Effects:* `--(*this)`.
>
> *Returns:* A copy of `*this` as it existed on entry to this member function.

```
constexpr month& month::operator+=(const months& m) noexcept;
```

> *Effects:* `*this = *this + m`.
>
> *Returns:* `*this`.

```
constexpr month& month::operator-=(const months& m) noexcept;
```

> *Effects:* `*this = *this - m`.
>
> *Returns:* `*this`.

```
constexpr explicit month::operator unsigned() const noexcept;
```

> *Returns:* `m_`.

```
constexpr bool month::ok() const noexcept;
```

Returns: `1 <= m_ && m_ <= 12`.

```
constexpr bool operator==(const month& x, const month& y) noexcept;
```

Returns: `unsigned{x} == unsigned{y}`.

```
constexpr bool operator!=(const month& x, const month& y) noexcept;
```

Returns: `!(x == y)`.

```
constexpr bool operator< (const month& x, const month& y) noexcept;
```

Returns: `unsigned{x} < unsigned{y}`.

```
constexpr bool operator> (const month& x, const month& y) noexcept;
```

Returns: `y < x`.

```
constexpr bool operator<=(const month& x, const month& y) noexcept;
```

Returns: `!(y < x)`.

```
constexpr bool operator>=(const month& x, const month& y) noexcept;
```

Returns: `!(x < y)`.

```
constexpr month  operator+(const month&  x, const months& y) noexcept;
```

Returns: A `month` for which `ok() == true` and is found as if by incrementing (or decrementing if `y < months{0}`) `x`, `y` times. If `month.ok() == false` prior to this operation, behaves as if `*this` is first brought into the range [1, 12] by modular arithmetic. [*Note:* For example `month{0}` becomes `month{12}`, and `month{13}` becomes `month{1}`. — *end note*]

Complexity: O(1) with respect to the value of `y`. That is, repeated increments or decrements is not a valid implementation.

Example: `February + months{11} == January`.

```
constexpr month  operator+(const months& x, const month&  y) noexcept;
```

Returns: `y + x`.

```
constexpr month  operator-(const month&  x, const months& y) noexcept;
```

Returns: `x + -y`.

```
constexpr months operator-(const month&  x, const month&  y) noexcept;
```

Returns: If `x.ok() == true` and `y.ok() == true`, returns a value of `months` in the range of `months{0}` to `months{11}` inclusive. Otherwise the value returned is unspecified.

Remarks: If `x.ok() == true` and `y.ok() == true`, the returned value `m` shall satisfy the equality: `y + m == x`.

Example: `January - February == months{11}` .

```
template<class CharT, class Traits>
std::basic_ostream<class CharT, class Traits>&
operator<<(std::basic_ostream<class CharT, class Traits>& os, const month& m);
```

Effects: If `ok() == true` outputs the same string that would be output for the month field by `asctime`. Otherwise outputs `unsigned{m} << " is not a valid month"`.

Returns: `os`.

```
template<class CharT, class Traits>
std::basic_ostream<CharT, Traits>&
to_stream(std::basic_ostream<class CharT, class Traits>& os, const CharT* fmt,
          const month& m);
```

Effects: Inserts `m` into `os` using the format string `fmt` as specified by the `to_stream` formatting flags. The behavior is undefined except for the following flags (or modified versions of these flags): `%b`, `%B`, `%h`, `%m`, `%n`, `%t` or `%%`.

Returns: `os`.

```
template <class CharT, class Traits, class Alloc = std::allocator<CharT>>
std::basic_istream<CharT, Traits>&
from_stream(std::basic_istream<CharT, Traits>& is, const CharT* fmt, month& m,
            std::basic_string<CharT, Traits, Alloc>* abbrev = nullptr,
            std::chrono::minutes* offset = nullptr);
```

Effects: Extracts `m` from `is` using the format string `fmt` as specified by the [`from_stream` formatting flags]. The behavior is undefined except for the following flags (or modified versions of these flags): `%b`, `%B`, `%h`, `%m`, `%n`, `%t`, `%z`, `%Z` or `%%`. If `abbrev` is not equal to `nullptr`, the information parsed by `%Z` (if present) will be placed in `*abbrev`. If `offset` is not equal to `nullptr`, the information parsed by `%z` (if present) will be placed in `*offset`. If an invalid date is parsed, or a time of day which is outside of the conventional bounds, `failbit` will be set and `y` will not be altered. In particular `!m.ok()` is not an indication of error.

Returns: `is`.

year

Synopsis

```
class year
{
    short y_;  // exposition only
public:
    year() = default;
    explicit constexpr year(int y) noexcept;

    constexpr year& operator++() noexcept;
    constexpr year operator++(int) noexcept;
    constexpr year& operator--() noexcept;
    constexpr year operator--(int) noexcept;

    constexpr year& operator+=(const years& y) noexcept;
    constexpr year& operator-=(const years& y) noexcept;

    constexpr year operator-() const noexcept;
    constexpr year operator+() const noexcept;

    constexpr bool is_leap() const noexcept;

    constexpr explicit operator int() const noexcept;
    constexpr bool ok() const noexcept;

    static constexpr year min() noexcept;
    static constexpr year max() noexcept;
};

constexpr bool operator==(const year& x, const year& y) noexcept;
constexpr bool operator!=(const year& x, const year& y) noexcept;
constexpr bool operator< (const year& x, const year& y) noexcept;
constexpr bool operator> (const year& x, const year& y) noexcept;
constexpr bool operator<=(const year& x, const year& y) noexcept;
constexpr bool operator>=(const year& x, const year& y) noexcept;

constexpr year  operator+(const year&  x, const years& y) noexcept;
constexpr year  operator+(const years& x, const year&  y) noexcept;
constexpr year  operator-(const year&  x, const years& y) noexcept;
constexpr years operator-(const year&  x, const year&  y) noexcept;

template<class CharT, class Traits>
std::basic_ostream<class CharT, class Traits>&
operator<<(std::basic_ostream<class CharT, class Traits>& os, const year& y);

template<class CharT, class Traits>
std::basic_ostream<CharT, Traits>&
to_stream(std::basic_ostream<class CharT, class Traits>& os, const CharT* fmt,
          const year& y);

template <class CharT, class Traits, class Alloc = std::allocator<CharT>>
std::basic_istream<CharT, Traits>&
from_stream(std::basic_istream<CharT, Traits>& is, const CharT* fmt, year& y,
            std::basic_string<CharT, Traits, Alloc>* abbrev = nullptr,
            std::chrono::minutes* offset = nullptr);

inline namespace literals {
```

```
constexpr year operator "" _y(unsigned long long y) noexcept;
}
```

Overview

`year` represents a year in the Gregorian calendar. It shall represent values in the range `[min(), max()]`. It can be constructed with any `int` value, which will be subsequently truncated to fit into `year`'s internal storage. The `year` prior to `year{1}` is `year{0}`, and is a leap year. The `year` prior to `year{0}` is `year{-1}`. This is commonly referred to as the proleptic Gregorian calendar. `year` is equality and less-than comparable, and participates in basic arithmetic with `years` representing the quantity between any two `year`'s. One can form a `year` literal with `_y`. And one can stream out a `year` for debugging purposes. `year` has explicit conversions to and from `int`.

Specification

`year` is a trivially copyable class type.
`year` is a standard-layout class type.
`year` is a literal class type.

`explicit constexpr year::year(int y) noexcept;`

> *Effects:* Constructs an object of type `year` by constructing `y_` with `y`. The value held is unspecified if `y` is not in the range `[-32767, 32767]`.

`constexpr year& year::operator++() noexcept;`

> *Effects:* `++y_`.
>
> *Returns:* `*this`.

`constexpr year year::operator++(int) noexcept;`

> *Effects:* `++(*this)`.
>
> *Returns:* A copy of `*this` as it existed on entry to this member function.

`constexpr year& year::operator--() noexcept;`

> *Effects:* `--y_`.
>
> *Returns:* `*this`.

`constexpr year year::operator--(int) noexcept;`

> *Effects:* `--(*this)`.
>
> *Returns:* A copy of `*this` as it existed on entry to this member function.

`constexpr year& year::operator+=(const years& y) noexcept;`

> *Effects:* `*this = *this + y`.
>
> *Returns:* `*this`.

`constexpr year& year::operator-=(const years& y) noexcept;`

> *Effects:* `*this = *this - y`.
>
> *Returns:* `*this`.

`constepxr year year::operator-() const noexcept;`

> *Returns:* `year{-y_}`.

`constepxr year year::operator+() const noexcept;`

> *Returns:* `*this`.

`constexpr bool year::is_leap() const noexcept;`

> *Returns:* `true` if `*this` represents a leap year, else returns `false`.

`constexpr explicit year::operator int() const noexcept;`

> *Returns:* `y_`.

```
constexpr bool year::ok() const noexcept;
```

> *Returns:* `true`.

```
static constexpr year year::min() noexcept;
```

> *Returns:* `year{-32767}`.

```
static constexpr year year::max() noexcept;
```

> *Returns:* `year{32767}`.

```
constexpr bool operator==(const year& x, const year& y) noexcept;
```

> *Returns:* `int{x} == int{y}`.

```
constexpr bool operator!=(const year& x, const year& y) noexcept;
```

> *Returns:* `!(x == y)`.

```
constexpr bool operator< (const year& x, const year& y) noexcept;
```

> *Returns:* `int{x} < int{y}`.

```
constexpr bool operator> (const year& x, const year& y) noexcept;
```

> *Returns:* `y < x`.

```
constexpr bool operator<=(const year& x, const year& y) noexcept;
```

> *Returns:* `!(y < x)`.

```
constexpr bool operator>=(const year& x, const year& y) noexcept;
```

> *Returns:* `!(x < y)`.

```
constexpr year  operator+(const year&  x, const years& y) noexcept;
```

> *Returns:* `year{int{x} + y.count()}`.

```
constexpr year  operator+(const years& x, const year&  y) noexcept;
```

> *Returns:* `y + x`.

```
constexpr year  operator-(const year&  x, const years& y) noexcept;
```

> *Returns:* `x + -y`.

```
constexpr years operator-(const year&  x, const year&  y) noexcept;
```

> *Returns:* `years{int{x} - int{y}}`.

```
constexpr year operator "" _y(unsigned long long y) noexcept;
```

> *Returns:* `year{static_cast<int>(y)}`.

```
template<class CharT, class Traits>
std::basic_ostream<class CharT, class Traits>&
operator<<(std::basic_ostream<class CharT, class Traits>& os, const year& y);
```

> *Effects:* Inserts a signed decimal integral text representation of `y` into `os`. If the year is less than four decimal digits, pads the year with `'0'` to four digits. If the year is negative, prefixes with `'-'`.
>
> *Returns:* `os`.

```
template<class CharT, class Traits>
std::basic_ostream<CharT, Traits>&
to_stream(std::basic_ostream<class CharT, class Traits>& os, const CharT* fmt,
          const year& y);
```

> *Effects:* Inserts `y` into `os` using the format string `fmt` as specified by the `to_stream` formatting flags. The behavior is undefined except for the following flags (or modified versions of these flags): `%C`, `%y`, `%Y`, `%n`, `%t` or `%%`.

Returns: `os`.

```
template <class CharT, class Traits, class Alloc = std::allocator<CharT>>
std::basic_istream<CharT, Traits>&
from_stream(std::basic_istream<CharT, Traits>& is, const CharT* fmt, year& y,
            std::basic_string<CharT, Traits, Alloc>* abbrev = nullptr,
            std::chrono::minutes* offset = nullptr);
```

Effects: Extracts `y` from `is` using the format string `fmt` as specified by the `from_stream` formatting flags. The behavior is undefined except for the following flags (or modified versions of these flags): `%C`, `%y`, `%Y`, `%n`, `%t`, `%z`, `%Z` or `%%`. If `abbrev` is not equal to `nullptr`, the information parsed by `%Z` (if present) will be placed in `*abbrev`. If `offset` is not equal to `nullptr`, the information parsed by `%z` (if present) will be placed in `*offset`. If an invalid date is parsed, or a time of day which is outside of the conventional bounds, `failbit` will be set and `y` will not be altered. In particular `!y.ok()` is not an indication of error.

Returns: `is`.

weekday

Synopsis

```
class weekday
{
    unsigned char wd_;  // exposition only
public:
    weekday() = default;
    explicit constexpr weekday(unsigned wd) noexcept;
    constexpr weekday(const sys_days& dp) noexcept;
    constexpr explicit weekday(const local_days& dp) noexcept;

    constexpr weekday& operator++() noexcept;
    constexpr weekday operator++(int) noexcept;
    constexpr weekday& operator--() noexcept;
    constexpr weekday operator--(int) noexcept;

    constexpr weekday& operator+=(const days& d) noexcept;
    constexpr weekday& operator-=(const days& d) noexcept;

    constexpr bool ok() const noexcept;
    constexpr weekday_indexed operator[](unsigned index) const noexcept;
    constexpr weekday_last operator[](last_spec) const noexcept;
};

constexpr bool operator==(const weekday& x, const weekday& y) noexcept;
constexpr bool operator!=(const weekday& x, const weekday& y) noexcept;

constexpr weekday operator+(const weekday& x, const days&    y) noexcept;
constexpr weekday operator+(const days&    x, const weekday& y) noexcept;
constexpr weekday operator-(const weekday& x, const days&    y) noexcept;
constexpr days    operator-(const weekday& x, const weekday& y) noexcept;

template<class CharT, class Traits>
std::basic_ostream<class CharT, class Traits>&
operator<<(std::basic_ostream<class CharT, class Traits>& os, const weekday& wd);

template<class CharT, class Traits>
std::basic_ostream<CharT, Traits>&
to_stream(std::basic_ostream<class CharT, class Traits>& os, const CharT* fmt,
          const weekday& wd);

template <class CharT, class Traits, class Alloc = std::allocator<CharT>>
std::basic_istream<CharT, Traits>&
from_stream(std::basic_istream<CharT, Traits>& is, const CharT* fmt, weekday& wd,
            std::basic_string<CharT, Traits, Alloc>* abbrev = nullptr,
            std::chrono::minutes* offset = nullptr);

constexpr weekday Monday{1};
constexpr weekday Tuesday{2};
constexpr weekday Wednesday{3};
constexpr weekday Thursday{4};
constexpr weekday Friday{5};
constexpr weekday Saturday{6};
constexpr weekday Sunday{7};
```

Overview

`weekday` represents a day of the week in the Gregorian calendar. It can be constructed with any `unsigned` value, which will be subsequently truncated to fit into `weekday`'s internal storage. The values [1, 6] map to Monday thru Saturday. Both 0 and 7 map to Sunday. Other values in the range [8, 255] will be stored, and will represent values of `weekday` that are `!ok()`. `weekday` is equality comparable. `weekday` is not less-than comparable because there is no universal consensus on which day is the first day of the week. `weekday`'s comparison and arithmetic operations treat the days of the week as a circular range, with no beginning and no end. One can stream out a `weekday` for debugging purposes. `weekday` has explicit conversions to and from `unsigned`. There are 7 `weekday` constants, one for each day of the week.

A `weekday` can be implicitly constructed from a `sys_days`. This is the computation that discovers the day of the week of an arbitrary date.

A `weekday` can be indexed with either `unsigned` or `last`. This produces new types which represent the first, second, third, fourth, fifth or last weekdays of a month.

Specification

`weekday` is a trivially copyable class type.
`weekday` is a standard-layout class type.
`weekday` is a literal class type.

```
explicit constexpr weekday::weekday(unsigned wd) noexcept;
```

> *Effects:* Constructs an object of type `weekday` which represents a day of the week according to the ISO 8601 mapping of integers to days of the week. Additionally, if `wd == 0`, Sunday is represented. [*Note:* Sunday can be constructed from both 0 and 7 — *end note*]. If `wd` is in the range [8, 255], `wd_` is initialized with wd. If wd is not in the range [0, 255], the value held is unspecified.

```
constexpr weekday(const sys_days& dp) noexcept;
```

> *Effects:* Constructs an object of type `weekday` by computing what day of the week corresponds to the `sys_days dp`, and representing that day of the week in `wd_`.
>
> *Example:* If `dp` represents 1970-01-01, the constructed `weekday` shall represent Thursday by storing 4 in `wd_`.

```
constexpr explicit weekday(const local_days& dp) noexcept;
```

> *Effects:* Constructs an object of type `weekday` by computing what day of the week corresponds to the `local_days dp`, and representing that day of the week in `wd_`.
>
> The value after construction shall be identical to that constructed from `sys_days{dp.time_since_epoch()}`.

```
constexpr weekday& weekday::operator++() noexcept;
```

> *Effects:* If `wd_ != 6`, `++wd_`. Otherwise sets `wd_` to 0.
>
> *Returns:* `*this`.

```
constexpr weekday weekday::operator++(int) noexcept;
```

> *Effects:* `++(*this)`.
>
> *Returns:* A copy of `*this` as it existed on entry to this member function.

```
constexpr weekday& weekday::operator--() noexcept;
```

> *Effects:* If `wd_ != 0`, `--wd_`. Otherwise sets `wd_` to 6.
>
> *Returns:* `*this`.

```
constexpr weekday weekday::operator--(int) noexcept;
```

> *Effects:* `--(*this)`.
>
> *Returns:* A copy of `*this` as it existed on entry to this member function.

```
constexpr weekday& weekday::operator+=(const days& d) noexcept;
```

> *Effects:* `*this = *this + d`.
>
> *Returns:* `*this`.

```
constexpr weekday& weekday::operator-=(const days& d) noexcept;
```

Effects: `*this = *this - d`.

Returns: `*this`.

```
constexpr bool weekday::ok() const noexcept;
```

Returns: `true` if `*this` holds a value in the range Monday thru Sunday, else returns `false`.

```
constexpr weekday_indexed weekday::operator[](unsigned index) const noexcept;
```

Returns: `{*this, index}`.

```
constexpr weekday_last weekday::operator[](last_spec) const noexcept;
```

Returns: `weekday_last{*this}`.

```
constexpr bool operator==(const weekday& x, const weekday& y) noexcept;
```

Returns: `x.wd_ == y.wd_`.

```
constexpr bool operator!=(const weekday& x, const weekday& y) noexcept;
```

Returns: `!(x == y)`.

```
constexpr weekday  operator+(const weekday&  x, const days& y) noexcept;
```

Returns: A `weekday` for which `ok() == true` and is found as if by incrementing (or decrementing if `y < days{0}`) `x`, `y` times. If `weekday.ok() == false` prior to this operation, behaves as if `*this` is first brought into the range [0, 6] by modular arithmetic. [*Note:* For example `weekday{7}` becomes `weekday{0}`. — *end note*]

Complexity: O(1) with respect to the value of `y`. That is, repeated increments or decrements is not a valid implementation.

Example: `Monday + days{6} == Sunday`.

```
constexpr weekday  operator+(const days& x, const weekday&  y) noexcept;
```

Returns: `y + x`.

```
constexpr weekday  operator-(const weekday&  x, const days& y) noexcept;
```

Returns: `x + -y`.

```
constexpr days operator-(const weekday&  x, const weekday&  y) noexcept;
```

Returns: If `x.ok() == true` and `y.ok() == true`, returns a value of `days` in the range of `days{0}` to `days{6}` inclusive. Otherwise the value returned is unspecified.

Remarks: If `x.ok() == true` and `y.ok() == true`, the returned value `d` shall satisfy the equality: `y + d == x`.

Example: `Sunday - Monday == days{6}`.

```
template<class CharT, class Traits>
std::basic_ostream<class CharT, class Traits>&
operator<<(std::basic_ostream<class CharT, class Traits>& os, const weekday& wd);
```

Effects: If `ok() == true` outputs the same string that would be output for the weekday field by `asctime`. Otherwise outputs `unsigned{wd} << " is not a valid weekday"`.

Returns: `os`.

```
template<class CharT, class Traits>
std::basic_ostream<CharT, Traits>&
to_stream(std::basic_ostream<class CharT, class Traits>& os, const CharT* fmt,
          const weekday& wd);
```

Effects: Inserts `wd` into `os` using the format string `fmt` as specified by the `to_stream` formatting flags. The behavior is undefined except for the following flags (or modified versions of these flags): `%a`, `%A`, `%u`, `%w`, `%n`, `%t` or `%%`.

Returns: `os`.

```
template <class CharT, class Traits, class Alloc = std::allocator<CharT>>
std::basic_istream<CharT, Traits>&
from_stream(std::basic_istream<CharT, Traits>& is, const CharT* fmt, weekday& wd,
            std::basic_string<CharT, Traits, Alloc>* abbrev = nullptr,
            std::chrono::minutes* offset = nullptr);
```

Effects: Extracts `wd` from `is` using the format string `fmt` as specified by the `from_stream` formatting flags. The behavior is undefined except for the following flags (or modified versions of these flags): `%a`, `%A`, `%u`, `%w`, `%n`, `%t`, `%z`, `%Z` or `%%`. If `abbrev` is not equal to `nullptr`, the information parsed by `%Z` (if present) will be placed in `*abbrev`. If `offset` is not equal to `nullptr`, the information parsed by `%z` (if present) will be placed in `*offset`. If an invalid date is parsed, or a time of day which is outside of the conventional bounds, `failbit` will be set and `wd` will not be altered. In particular `!wd.ok()` is not an indication of error.

Returns: `is`.

weekday_indexed

Synopsis

```
class weekday_indexed
{
    date::weekday    wd_;     // exposition only
    unsigned char    index_;  // exposition only

public:
    constexpr weekday_indexed(const date::weekday& wd, unsigned index) noexcept;

    constexpr date::weekday weekday() const noexcept;
    constexpr unsigned index() const noexcept;
    constexpr bool ok() const noexcept;
};

constexpr bool operator==(const weekday_indexed& x, const weekday_indexed& y) noexcept;
constexpr bool operator!=(const weekday_indexed& x, const weekday_indexed& y) noexcept;

template<class CharT, class Traits>
std::basic_ostream<class CharT, class Traits>&
operator<<(std::basic_ostream<class CharT, class Traits>& os, const weekday_indexed& wdi);
```

Overview

`weekday_indexed` represents a `weekday` and a small index in the range 1 to 5. This class is used to represent the first, second, third, fourth or fifth weekday of a month. It is most easily constructed by indexing a `weekday`.

[*Example:*

```
constexpr auto wdi = Sunday[2];  // wdi is the second Sunday of an as yet unspecified month
static_assert(wdi.weekday() == Sunday);
static_assert(wdi.index() == 2);
```

— *end example:*]

Specification

`weekday_indexed` is a trivially copyable class type.
`weekday_indexed` is a standard-layout class type.
`weekday_indexed` is a literal class type.

```
constexpr weekday_indexed::weekday_indexed(const date::weekday& wd, unsigned index) noexcept;
```

Effects: Constructs an object of type `weekday_indexed` by constructing `wd_` with `wd` and `index_` with `index`. The values held are unspecified if `!wd.ok()` or `index` is not in the range `[1, 5]`.

```
constexpr weekday weekday_indexed::weekday() const noexcept;
```

Returns: `wd_`.

```
constexpr unsigned weekday_indexed::index() const noexcept;
```

Returns: `index_`.

```
constexpr bool weekday_indexed::ok() const noexcept;
```

Returns: `wd_.ok() && 1 <= index_ && index_ <= 5`.

```
constexpr bool operator==(const weekday_indexed& x, const weekday_indexed& y) noexcept;
```

Returns: `x.weekday() == y.weekday() && x.index() == y.index()`.

```
constexpr bool operator!=(const weekday_indexed& x, const weekday_indexed& y) noexcept;
```

Returns: `!(x == y)`.

```
template<class CharT, class Traits>
std::basic_ostream<class CharT, class Traits>&
operator<<(std::basic_ostream<class CharT, class Traits>& os, const weekday_indexed& wdi);
```

Effects: Inserts `os << wdi.weekday() << '[' << wdi.index() << ']'`.

Returns: `os`.

`weekday_last`

Synopsis

```
class weekday_last
{
    date::weekday wd_;  // exposition only

public:
    explicit constexpr weekday_last(const date::weekday& wd) noexcept;

    constexpr date::weekday weekday() const noexcept;
    constexpr bool ok() const noexcept;
};

constexpr bool operator==(const weekday_last& x, const weekday_last& y) noexcept;
constexpr bool operator!=(const weekday_last& x, const weekday_last& y) noexcept;

template<class CharT, class Traits>
std::basic_ostream<class CharT, class Traits>&
operator<<(std::basic_ostream<class CharT, class Traits>& os, const weekday_last& wdl);
```

Overview

`weekday_last` represents the last `weekday` of a month. It is most easily constructed by indexing a `weekday` with `last`.

[*Example:*

```
constexpr auto wdl = Sunday[last];  // wdl is the last Sunday of an as yet unspecified month
static_assert(wdl.weekday() == Sunday);
```

— *end example:*]

Specification

`weekday_last` is a trivially copyable class type.
`weekday_last` is a standard-layout class type.
`weekday_last` is a literal class type.

```
explicit constexpr weekday_last::weekday_last(const date::weekday& wd) noexcept;
```

Effects: Constructs an object of type `weekday_last` by constructing `wd_` with `wd`.

```
constexpr weekday weekday_last::weekday() const noexcept;
```

Returns: `wd_`.

```
constexpr bool weekday_last::ok() const noexcept;
```

Returns: `wd_.ok()`.

```
constexpr bool operator==(const weekday_last& x, const weekday_last& y) noexcept;
```

Returns: `x.weekday() == y.weekday()`.

```
constexpr bool operator!=(const weekday_last& x, const weekday_last& y) noexcept;
```

Returns: `!(x == y)`.

```
template<class CharT, class Traits>
std::basic_ostream<class CharT, class Traits>&
operator<<(std::basic_ostream<class CharT, class Traits>& os, const weekday_last& wdl);
```

Effects: Inserts `os << wdi.weekday() << "[last]"`.

Returns: `os`.

`month_day`

Synopsis

```
class month_day
{
    date::month m_;  // exposition only
    date::day   d_;  // exposition only

public:
    month_day() = default;
    constexpr month_day(const date::month& m, const date::day& d) noexcept;

    constexpr date::month month() const noexcept;
    constexpr date::day day() const noexcept;
    constexpr bool ok() const noexcept;
};

constexpr bool operator==(const month_day& x, const month_day& y) noexcept;
constexpr bool operator!=(const month_day& x, const month_day& y) noexcept;
constexpr bool operator< (const month_day& x, const month_day& y) noexcept;
constexpr bool operator> (const month_day& x, const month_day& y) noexcept;
constexpr bool operator<=(const month_day& x, const month_day& y) noexcept;
constexpr bool operator>=(const month_day& x, const month_day& y) noexcept;

template<class CharT, class Traits>
std::basic_ostream<class CharT, class Traits>&
operator<<(std::basic_ostream<class CharT, class Traits>& os, const month_day& md);

template<class CharT, class Traits>
std::basic_ostream<CharT, Traits>&
to_stream(std::basic_ostream<class CharT, class Traits>& os, const CharT* fmt,
          const month_day& md);

template <class CharT, class Traits, class Alloc = std::allocator<CharT>>
std::basic_istream<CharT, Traits>&
from_stream(std::basic_istream<CharT, Traits>& is, const CharT* fmt, month_day& md,
            std::basic_string<CharT, Traits, Alloc>* abbrev = nullptr,
            std::chrono::minutes* offset = nullptr);
```

Overview

`month_day` represents a specific `day` of a specific `month`, but with an unspecified `year`. One can observe the different components. One can assign a new value. `month_day` is equality comparable and less-than comparable. One can stream out a `month_day` for debugging purposes.

Specification

`month_day` is a trivially copyable class type.
`month_day` is a standard-layout class type.
`month_day` is a literal class type.

```
constexpr month_day::month_day(const date::month& m, const date::day& d) noexcept;
```

Effects: Constructs an object of type `month_day` by constructing `m_` with `m`, and `d_` with `d`.

```
constexpr month month_day::month() const noexcept;
```

Returns: `m_`.

```
constexpr day month_day::day() const noexcept;
```

Returns: `d_`.

```
constexpr bool month_day::ok() const noexcept;
```

Returns: `true` if `m_.ok()` is true, and if `1_d <= d_`, and if `d_ <=` the number of days in month `m_`. For `m_ == February` the number of days is considered to be 29. Otherwise returns `false`.

```
constexpr bool operator==(const month_day& x, const month_day& y) noexcept;
```

Returns: `x.month() == y.month() && x.day() == y.day()`

```
constexpr bool operator!=(const month_day& x, const month_day& y) noexcept;
```

Returns: `!(x == y)`

```
constexpr bool operator< (const month_day& x, const month_day& y) noexcept;
```

Returns: If `x.month() < y.month()` returns `true`. Else if `x.month() > y.month()` returns `false`. Else returns `x.day() < y.day()`.

```
constexpr bool operator> (const month_day& x, const month_day& y) noexcept;
```

Returns: `y < x`.

```
constexpr bool operator<=(const month_day& x, const month_day& y) noexcept;
```

Returns: `!(y < x)`.

```
constexpr bool operator>=(const month_day& x, const month_day& y) noexcept;
```

Returns: `!(x < y)`.

```
template<class CharT, class Traits>
std::basic_ostream<class CharT, class Traits>&
operator<<(std::basic_ostream<class CharT, class Traits>& os, const month_day& md);
```

Effects: Inserts `os << md.month() << '/' << md.day()`.

Returns: `os`.

```
template<class CharT, class Traits>
std::basic_ostream<CharT, Traits>&
to_stream(std::basic_ostream<class CharT, class Traits>& os, const CharT* fmt,
          const month_day& md);
```

Effects: Inserts `md` into `os` using the format string `fmt` as specified by the `to_stream` formatting flags. The behavior is undefined except for the following flags (or modified versions of these flags): `%b`, `%B`, `%d`, `%e`, `%h`, `%m`, `%n`, `%t` or `%%`.

Returns: `os`.

```
template <class CharT, class Traits, class Alloc = std::allocator<CharT>>
std::basic_istream<CharT, Traits>&
from_stream(std::basic_istream<CharT, Traits>& is, const CharT* fmt, month_day& md,
            std::basic_string<CharT, Traits, Alloc>* abbrev = nullptr,
            std::chrono::minutes* offset = nullptr);
```

Effects: Extracts `md` from `is` using the format string `fmt` as specified by the `from_stream` formatting flags. The behavior is undefined except for the following flags (or modified versions of these flags): `%b`, `%B`, `%d`, `%e`, `%h`, `%m`, `%n`, `%t`, `%z`, `%Z` or `%%`. If `abbrev` is not equal to `nullptr`, the information parsed by `%Z` (if present) will be placed in `*abbrev`. If `offset` is not equal to `nullptr`, the information parsed by `%z` (if present) will be placed in `*offset`. If an invalid date is parsed, or a time of day which is outside of the conventional bounds, `failbit` will be set and `md` will not be altered. In particular `!md.ok()` is not an indication of error.

Returns: `is`.

`month_day_last`

Synopsis

```
class month_day_last
{
    date::month m_;  // exposition only

public:
    constexpr explicit month_day_last(const date::month& m) noexcept;

    constexpr date::month month() const noexcept;
    constexpr bool ok() const noexcept;
};

constexpr bool operator==(const month_day_last& x, const month_day_last& y) noexcept;
constexpr bool operator!=(const month_day_last& x, const month_day_last& y) noexcept;
constexpr bool operator< (const month_day_last& x, const month_day_last& y) noexcept;
constexpr bool operator> (const month_day_last& x, const month_day_last& y) noexcept;
constexpr bool operator<=(const month_day_last& x, const month_day_last& y) noexcept;
constexpr bool operator>=(const month_day_last& x, const month_day_last& y) noexcept;

template<class CharT, class Traits>
std::basic_ostream<class CharT, class Traits>&
operator<<(std::basic_ostream<class CharT, class Traits>& os, const month_day_last& mdl);
```

Overview

`month_day_last` represents the last `day` of a `month`. It is most easily constructed using the expression `m/last` or `last/m`, where `m` is an expression with type `month`.

[*Example:*

```
constexpr auto mdl = February/last;  // mdl is the last day of February of an as yet unspecified year
static_assert(mdl.month() == February);
```

— *end example:*]

Specification

`month_day_last` is a trivially copyable class type.
`month_day_last` is a standard-layout class type.
`month_day_last` is a literal class type.

```
constexpr explicit month_day_last::month_day_last(const date::month& m) noexcept;
```

Effects: Constructs an object of type `month_day_last` by constructing `m_` with `m`.

```
constexpr month month_day_last::month() const noexcept;
```

Returns: `m_`.

```
constexpr bool month_day_last::ok() const noexcept;
```

Returns: `m_.ok()`.

```
constexpr bool operator==(const month_day_last& x, const month_day_last& y) noexcept;
```

Returns: `x.month() == y.month()`.

```
constexpr bool operator!=(const month_day_last& x, const month_day_last& y) noexcept;
```

Returns: `!(x == y)`

```
constexpr bool operator< (const month_day_last& x, const month_day_last& y) noexcept;
```

Returns: `x.month() < y.month()`.

```
constexpr bool operator> (const month_day_last& x, const month_day_last& y) noexcept;
```

Returns: `y < x`.

```
constexpr bool operator<=(const month_day_last& x, const month_day_last& y) noexcept;
```

Returns: `!(y < x)`.

```
constexpr bool operator>=(const month_day_last& x, const month_day_last& y) noexcept;
```

Returns: `!(x < y)`.

```
template<class CharT, class Traits>
std::basic_ostream<class CharT, class Traits>&
operator<<(std::basic_ostream<class CharT, class Traits>& os, const month_day_last& mdl);
```

Effects: Inserts `os << mdl.month() << "/last"`.

Returns: `os`.

`month_weekday`

Synopsis

```
class month_weekday
{
    date::month           m_;    // exposition only
    date::weekday_indexed wdi_;  // exposition only
public:
    constexpr month_weekday(const date::month& m, const date::weekday_indexed& wdi) noexcept;

    constexpr date::month month() const noexcept;
    constexpr date::weekday_indexed weekday_indexed() const noexcept;
    constexpr bool ok() const noexcept;
};

constexpr bool operator==(const month_weekday& x, const month_weekday& y) noexcept;
constexpr bool operator!=(const month_weekday& x, const month_weekday& y) noexcept;

template<class CharT, class Traits>
std::basic_ostream<class CharT, class Traits>&
operator<<(std::basic_ostream<class CharT, class Traits>& os, const month_weekday& mwd);
```

Overview

`month_weekday` represents the nth `weekday` of a `month`, of an as yet unspecified `year`. To do this the `month_weekday` stores a `month` and a `weekday_indexed`.

Specification

`month_weekday` is a trivially copyable class type.
`month_weekday` is a standard-layout class type.
`month_weekday` is a literal class type.

```
constexpr month_weekday::month_weekday(const date::month& m, const date::weekday_indexed& wdi) noexcept;
```

> *Effects:* Constructs an object of type `month_weekday` by constructing `m_` with `m`, and `wdi_` with `wdi`.

```
constexpr month month_weekday::month() const noexcept;
```

> *Returns:* `m_`.

```
constexpr weekday_indexed month_weekday::weekday_indexed() const noexcept;
```

> *Returns:* `wdi_`.

```
constexpr bool month_weekday::ok() const noexcept;
```

> *Returns:* `m_.ok() && wdi_.ok()`.

```
constexpr bool operator==(const month_weekday& x, const month_weekday& y) noexcept;
```

> *Returns:* `x.month() == y.month() && x.weekday_indexed() == y.weekday_indexed()`.

```
constexpr bool operator!=(const month_weekday& x, const month_weekday& y) noexcept;
```

> *Returns:* `!(x == y)`.

```
template<class CharT, class Traits>
std::basic_ostream<class CharT, class Traits>&
operator<<(std::basic_ostream<class CharT, class Traits>& os, const month_weekday& mwd);
```

> *Effects:* Inserts `os << mwd.month() << '/' << mwd.weekday_indexed()`.
>
> *Returns:* `os`.

`month_weekday_last`

Synopsis

```
class month_weekday_last
{
    date::month        m_;     // exposition only
    date::weekday_last wdl_;   // exposition only
public:
    constexpr month_weekday_last(const date::month& m,
                                 const date::weekday_last& wdl) noexcept;

    constexpr date::month        month()        const noexcept;
    constexpr date::weekday_last weekday_last() const noexcept;
    constexpr bool ok() const noexcept;
};

constexpr bool operator==(const month_weekday_last& x, const month_weekday_last& y) noexcept;
constexpr bool operator!=(const month_weekday_last& x, const month_weekday_last& y) noexcept;

template<class CharT, class Traits>
std::basic_ostream<class CharT, class Traits>&
operator<<(std::basic_ostream<class CharT, class Traits>& os, const month_weekday_last& mwdl);
```

Overview

`month_weekday_last` represents the last `weekday` of a `month`, of an as yet unspecified `year`. To do this the `month_weekday_last` stores a `month` and a `weekday_last`.

Specification

`month_weekday_last` is a trivially copyable class type.
`month_weekday_last` is a standard-layout class type.
`month_weekday_last` is a literal class type.

```
constexpr month_weekday_last::month_weekday_last(const date::month& m,
                                                 const date::weekday_last& wdl) noexcept;
```

Effects: Constructs an object of type `month_weekday_last` by constructing `m_` with `m`, and `wdl_` with `wdl`.

```
constexpr month month_weekday_last::month() const noexcept;
```

Returns: `m_`.

```
constexpr weekday_last month_weekday_last::weekday_last() const noexcept;
```

Returns: `wdl_`.

```
constexpr bool month_weekday_last::ok() const noexcept;
```

Returns: `m_.ok() && wdl_.ok()`.

```
constexpr bool operator==(const month_weekday_last& x, const month_weekday_last& y) noexcept;
```

Returns: `x.month() == y.month() && x.weekday_last() == y.weekday_last()`.

```
constexpr bool operator!=(const month_weekday_last& x, const month_weekday_last& y) noexcept;
```

Returns: `!(x == y)`.

```
template<class CharT, class Traits>
std::basic_ostream<class CharT, class Traits>&
operator<<(std::basic_ostream<class CharT, class Traits>& os, const month_weekday_last& mwdl);
```

Effects: Inserts `os << mwdl.month() << '/' << mwdl.weekday_last()`.

Returns: `os`.

year_month

Synopsis

```
class year_month
{
    date::year  y_;  // exposition only
    date::month m_;  // exposition only

public:
    year_month() = default;
    constexpr year_month(const date::year& y, const date::month& m) noexcept;

    constexpr date::year year() const noexcept;
    constexpr date::month month() const noexcept;
    constexpr year_month& operator+=(const months& dm) noexcept;
    constexpr year_month& operator-=(const months& dm) noexcept;
    constexpr year_month& operator+=(const years& dy) noexcept;
    constexpr year_month& operator-=(const years& dy) noexcept;

    constexpr bool ok() const noexcept;
};

constexpr bool operator==(const year_month& x, const year_month& y) noexcept;
constexpr bool operator!=(const year_month& x, const year_month& y) noexcept;
constexpr bool operator< (const year_month& x, const year_month& y) noexcept;
constexpr bool operator> (const year_month& x, const year_month& y) noexcept;
constexpr bool operator<=(const year_month& x, const year_month& y) noexcept;
constexpr bool operator>=(const year_month& x, const year_month& y) noexcept;

constexpr year_month operator+(const year_month& ym, const months& dm) noexcept;
constexpr year_month operator+(const months& dm, const year_month& ym) noexcept;
constexpr year_month operator-(const year_month& ym, const months& dm) noexcept;
constexpr months operator-(const year_month& x, const year_month& y) noexcept;
constexpr year_month operator+(const year_month& ym, const years& dy) noexcept;
constexpr year_month operator+(const years& dy, const year_month& ym) noexcept;
constexpr year_month operator-(const year_month& ym, const years& dy) noexcept;

template<class CharT, class Traits>
std::basic_ostream<class CharT, class Traits>&
operator<<(std::basic_ostream<class CharT, class Traits>& os, const year_month& ym);

template<class CharT, class Traits>
std::basic_ostream<CharT, Traits>&
to_stream(std::basic_ostream<class CharT, class Traits>& os, const CharT* fmt,
          const year_month& ym);

template <class CharT, class Traits, class Alloc = std::allocator<CharT>>
std::basic_istream<CharT, Traits>&
from_stream(std::basic_istream<CharT, Traits>& is, const CharT* fmt, year_month& ym,
            std::basic_string<CharT, Traits, Alloc>* abbrev = nullptr,
            std::chrono::minutes* offset = nullptr);
```

Overview

`year_month` represents a specific `month` of a specific `year`, but with an unspecified `day`. `year_month` is a field-based time point with a resolution of `months`. One can observe the different components. One can assign a new value. `year_month` is equality comparable and less-than comparable. One can stream out a `year_month` for debugging purposes.

Specification

`year_month` is a trivially copyable class type.
`year_month` is a standard-layout class type.
`year_month` is a literal class type.

```
constexpr year_month::year_month(const date::year& y, const date::month& m) noexcept;
```

Effects: Constructs an object of type `year_month` by constructing `y_` with `y`, and `m_` with `m`.

```
constexpr year year_month::year() const noexcept;
```

Returns: `y_`.

```
constexpr month year_month::month() const noexcept;
```

Returns: `m_`.

```
constexpr year_month& operator+=(const months& dm) noexcept;
```

Effects: `*this = *this + dm`.

Returns: `*this`.

```
constexpr year_month& operator-=(const months& dm) noexcept;
```

Effects: `*this = *this - dm`.

Returns: `*this`.

```
constexpr year_month& operator+=(const years& dy) noexcept;
```

Effects: `*this = *this + dy`.

Returns: `*this`.

```
constexpr year_month& operator-=(const years& dy) noexcept;
```

Effects: `*this = *this - dy`.

Returns: `*this`.

```
constexpr bool year_month::ok() const noexcept;
```

Returns: `y_.ok() && m_.ok()`.

```
constexpr bool operator==(const year_month& x, const year_month& y) noexcept;
```

Returns: `x.year() == y.year() && x.month() == y.month()`

```
constexpr bool operator!=(const year_month& x, const year_month& y) noexcept;
```

Returns: `!(x == y)`

```
constexpr bool operator< (const year_month& x, const year_month& y) noexcept;
```

Returns: If `x.year() < y.year()` returns `true`. Else if `x.year() > y.year()` returns `false`. Else returns `x.month() < y.month()`.

```
constexpr bool operator> (const year_month& x, const year_month& y) noexcept;
```

Returns: `y < x`.

```
constexpr bool operator<=(const year_month& x, const year_month& y) noexcept;
```

Returns: `!(y < x)`.

```
constexpr bool operator>=(const year_month& x, const year_month& y) noexcept;
```

Returns: `!(x < y)`.

```
constexpr year_month operator+(const year_month& ym, const months& dm) noexcept;
```

Returns: A `year_month` value `z` such that `z - ym == dm`.

Complexity: O(1) with respect to the value of `dm`.

```
constexpr year_month operator+(const months& dm, const year_month& ym) noexcept;
```

Returns: `ym + dm`.

```
constexpr year_month operator-(const year_month& ym, const months& dm) noexcept;
```

Returns: `ym + -dm`.

```
constexpr months operator-(const year_month& x, const year_month& y) noexcept;
```

Returns: The number of `months` one must add to `y` to get `x`.

```
constexpr year_month operator+(const year_month& ym, const years& dy) noexcept;
```

Returns: `(ym.year() + dy) / ym.month()`.

```
constexpr year_month operator+(const years& dy, const year_month& ym) noexcept;
```

Returns: `ym + dy`.

```
constexpr year_month operator-(const year_month& ym, const years& dy) noexcept;
```

Returns: `ym + -dy`.

```
template<class CharT, class Traits>
std::basic_ostream<class CharT, class Traits>&
operator<<(std::basic_ostream<class CharT, class Traits>& os, const year_month& ym);
```

Effects: Inserts `os << ym.year() << '/' << ym.month()`.

Returns: `os`.

```
template<class CharT, class Traits>
std::basic_ostream<CharT, Traits>&
to_stream(std::basic_ostream<class CharT, class Traits>& os, const CharT* fmt,
          const year_month& ym);
```

Effects: Inserts `ym` into `os` using the format string `fmt` as specified by the `to_stream` formatting flags. The behavior is undefined except for the following flags (or modified versions of these flags): `%b`, `%B`, `%C`, `%h`, `%m`, `%y`, `%Y`, `%n`, `%t` or `%%`.

Returns: `os`.

```
template <class CharT, class Traits, class Alloc = std::allocator<CharT>>
std::basic_istream<CharT, Traits>&
from_stream(std::basic_istream<CharT, Traits>& is, const CharT* fmt, year_month& ym,
            std::basic_string<CharT, Traits, Alloc>* abbrev = nullptr,
            std::chrono::minutes* offset = nullptr);
```

Effects: Extracts `ym` from `is` using the format string `fmt` as specified by the `from_stream` formatting flags. The behavior is undefined except for the following flags (or modified versions of these flags): `%b`, `%B`, `%C`, `%h`, `%m`, `%y`, `%Y`, `%n`, `%t`, `%z`, `%Z` or `%%`. If `abbrev` is not equal to `nullptr`, the information parsed by `%Z` (if present) will be placed in `*abbrev`. If `offset` is not equal to `nullptr`, the information parsed by `%z` (if present) will be placed in `*offset`. If an invalid date is parsed, or a time of day which is outside of the conventional bounds, `failbit` will be set and `ym` will not be altered. In particular `!ym.ok()` is not an indication of error.

Returns: `is`.

year_month_day

Synopsis

```
class year_month_day
{
    date::year  y_;  // exposition only
    date::month m_;  // exposition only
    date::day   d_;  // exposition only

public:
    year_month_day() = default;
    constexpr year_month_day(const date::year& y, const date::month& m, const date::day& d) noexcept;
    constexpr year_month_day(const year_month_day_last& ymdl) noexcept;
    constexpr year_month_day(const sys_days& dp) noexcept;
    constexpr explicit year_month_day(const local_days& dp) noexcept;

    constexpr year_month_day& operator+=(const months& m) noexcept;
    constexpr year_month_day& operator-=(const months& m) noexcept;
    constexpr year_month_day& operator+=(const years& y) noexcept;
    constexpr year_month_day& operator-=(const years& y) noexcept;

    constexpr date::year year()   const noexcept;
    constexpr date::month month() const noexcept;
    constexpr date::day day()     const noexcept;

    constexpr          operator sys_days() const noexcept;
    constexpr explicit operator local_days() const noexcept;
    constexpr bool ok() const noexcept;
};

constexpr bool operator==(const year_month_day& x, const year_month_day& y) noexcept;
constexpr bool operator!=(const year_month_day& x, const year_month_day& y) noexcept;
constexpr bool operator< (const year_month_day& x, const year_month_day& y) noexcept;
constexpr bool operator> (const year_month_day& x, const year_month_day& y) noexcept;
constexpr bool operator<=(const year_month_day& x, const year_month_day& y) noexcept;
constexpr bool operator>=(const year_month_day& x, const year_month_day& y) noexcept;

constexpr year_month_day operator+(const year_month_day& ymd, const months& dm) noexcept;
constexpr year_month_day operator+(const months& dm, const year_month_day& ymd) noexcept;
constexpr year_month_day operator+(const year_month_day& ymd, const years& dy) noexcept;
constexpr year_month_day operator+(const years& dy, const year_month_day& ymd) noexcept;
constexpr year_month_day operator-(const year_month_day& ymd, const months& dm) noexcept;
constexpr year_month_day operator-(const year_month_day& ymd, const years& dy) noexcept;

template<class CharT, class Traits>
std::basic_ostream<class CharT, class Traits>&
operator<<(std::basic_ostream<class CharT, class Traits>& os, const year_month_day& ymd);
```

```
template<class CharT, class Traits>
std::basic_ostream<CharT, Traits>&
to_stream(std::basic_ostream<class CharT, class Traits>& os, const CharT* fmt,
          const year_month_day& ymd);

template <class CharT, class Traits, class Alloc = std::allocator<CharT>>
std::basic_istream<CharT, Traits>&
from_stream(std::basic_istream<CharT, Traits>& is, const CharT* fmt, year_month_day& ymd,
            std::basic_string<CharT, Traits, Alloc>* abbrev = nullptr,
            std::chrono::minutes* offset = nullptr);
```

Overview

`year_month_day` represents a specific `year`, `month`, and `day`. `year_month_day` is a field-based time point with a resolution of `days`. One can observe each field. `year_month_day` supports `years` and `months` oriented arithmetic, but not `days` oriented arithmetic. For the latter, there is a conversion to `sys_days` which efficiently supports `days` oriented arithmetic. There is also a conversion *from* `sys_days`. `year_month_day` is equality and less-than comparable.

Specification

`year_month_day` is a trivially copyable class type.
`year_month_day` is a standard-layout class type.
`year_month_day` is a literal class type.

```
constexpr year_month_day::year_month_day(const date::year& y, const date::month& m, const date::day& d) noexcept;
```

Effects: Constructs an object of type `year_month_day` by constructing `y_` with `y`, `m_` with `m`, and, `d_` with `d`.

```
constexpr year_month_day::year_month_day(const year_month_day_last& ymdl) noexcept;
```

Effects: Constructs an object of type `year_month_day` by constructing `y_` with `ymdl.year()`, `m_` with `ymdl.month()`, and, `d_` with `ymdl.day()`.

Note: This conversion from `year_month_day_last` to `year_month_day` is more efficient than converting a `year_month_day_last` to a `sys_days`, and then converting that `sys_days` to a `year_month_day`.

```
constexpr year_month_day::year_month_day(const sys_days& dp) noexcept;
```

Effects: Constructs an object of type `year_month_day` which corresponds to the date represented by `dp`.

Remarks: For any value of `year_month_day`, `ymd`, for which `ymd.ok()` is `true`, this equality will also be `true`: `ymd == year_month_day{sys_days{ymd}}`.

```
constexpr explicit year_month_day::year_month_day(const local_days& dp) noexcept;
```

Effects: Constructs an object of type `year_month_day` which corresponds to the date represented by `dp`.

Remarks: Equivalent to constructing with `sys_days{dp.time_since_epoch()}`.

```
constexpr year_month_day& year_month_day::operator+=(const months& m) noexcept;
```

Effects: `*this = *this + m;`.

Returns: `*this`.

```
constexpr year_month_day& year_month_day::operator-=(const months& m) noexcept;
```

Effects: `*this = *this - m;`.

Returns: `*this`.

```
constexpr year_month_day& year_month_day::operator+=(const years& y) noexcept;
```

Effects: `*this = *this + y;`.

Returns: `*this`.

```
constexpr year_month_day& year_month_day::operator-=(const years& y) noexcept;
```

Effects: `*this = *this - y;`.

Returns: `*this`.

```
constexpr year year_month_day::year() const noexcept;
```

Returns: `y_`.

```
constexpr month year_month_day::month() const noexcept;
```

Returns: `m_`.

```
constexpr day year_month_day::day() const noexcept;
```

Returns: `d_`.

```
constexpr year_month_day::operator sys_days() const noexcept;
```

Returns: If `ok()`, returns a `sys_days` holding a count of `days` from the `sys_days` epoch to `*this` (a negative value if `*this` represents a date prior to the `sys_days` epoch). Otherwise if `y_.ok() && m_.ok() == true` returns a `sys_days` which is offset from `sys_days{y_/m_/last}` by the number of `days` `d_` is offset from `sys_days{y_/m_/last}.day()`. Otherwise the value returned is unspecified.

Remarks: A `sys_days` in the range `[days{-12687428}, days{11248737}]` which is converted to a `year_month_day`, shall have the same value when converted back to a `sys_days`.

[*Example*:

```
static_assert(year_month_day{sys_days{2017y/January/0}}  == 2016y/December/31);
static_assert(year_month_day{sys_days{2017y/January/31}} == 2017y/January/31);
static_assert(year_month_day{sys_days{2017y/January/32}} == 2017y/February/1);
```

—end example]

```
constexpr explicit year_month_day::operator local_days() const noexcept;
```

Effects: Equivalent to:

```
return local_days{sys_days{*this}.time_since_epoch()};
```

```
constexpr bool year_month_day::ok() const noexcept;
```

Returns: If `y_.ok()` is `true`, and `m_.ok()` is `true`, and `d_` is in the range `[1_d, (y_/m_/last).day()]`, then returns `true`, else returns `false`.

```
constexpr bool operator==(const year_month_day& x, const year_month_day& y) noexcept;
```

Returns: `x.year() == y.year() && x.month() == y.month() && x.day() == y.day()`.

```
constexpr bool operator!=(const year_month_day& x, const year_month_day& y) noexcept;
```

Returns: `!(x == y)`.

```
constexpr bool operator< (const year_month_day& x, const year_month_day& y) noexcept;
```

Returns: If `x.year() < y.year()`, returns `true`. Else if `x.year() > y.year()` returns `false`. Else if `x.month() < y.month()`, returns `true`. Else if `x.month() > y.month()`, returns `false`. Else returns `x.day() < y.day()`.

```
constexpr bool operator> (const year_month_day& x, const year_month_day& y) noexcept;
```

Returns: `y < x`.

```
constexpr bool operator<=(const year_month_day& x, const year_month_day& y) noexcept;
```

Returns: `!(y < x)`.

```
constexpr bool operator>=(const year_month_day& x, const year_month_day& y) noexcept;
```

Returns: `!(x < y)`.

```
constexpr year_month_day operator+(const year_month_day& ymd, const months& dm) noexcept;
```

Returns: `(ymd.year() / ymd.month() + dm) / ymd.day()`.

Remarks: If `ymd.day()` is in the range `[1_d, 28_d]`, the resultant `year_month_day` is guaranteed to return `true` from `ok()`.

```
constexpr year_month_day operator+(const months& dm, const year_month_day& ymd) noexcept;
```

Returns: `ymd + dm`.

```
constexpr year_month_day operator-(const year_month_day& ymd, const months& dm) noexcept;
```

Returns: `ymd + (-dm)`.

```
constexpr year_month_day operator+(const year_month_day& ymd, const years& dy) noexcept;
```

Returns: `(ymd.year() + dy) / ymd.month() / ymd.day()`.

Remarks: If `ymd.month()` is `February` and `ymd.day()` is not in the range `[1_d, 28_d]`, the resultant `year_month_day` is not guaranteed to return `true` from `ok()`.

```
constexpr year_month_day operator+(const years& dy, const year_month_day& ymd) noexcept;
```

Returns: `ymd + dy`.

```
constexpr year_month_day operator-(const year_month_day& ymd, const years& dy) noexcept;
```

Returns: `ymd + (-dy)`.

```
template<class CharT, class Traits>
std::basic_ostream<class CharT, class Traits>&
operator<<(std::basic_ostream<class CharT, class Traits>& os, const year_month_day& ymd);
```

Effects: Inserts `yyyy-mm-dd` where the number of indicated digits are prefixed with `'0'` if necessary..

Returns: `os`.

```
template<class CharT, class Traits>
std::basic_ostream<CharT, Traits>&
to_stream(std::basic_ostream<class CharT, class Traits>& os, const CharT* fmt,
          const year_month_day& ymd);
```

Effects: Inserts `ymd` into `os` using the format string `fmt` as specified by the `to_stream` formatting flags.

If `%z` or `%Z` is used in the `fmt` string `failbit` will be set for `os`.

Returns: `os`.

```
template <class CharT, class Traits, class Alloc = std::allocator<CharT>>
std::basic_istream<CharT, Traits>&
from_stream(std::basic_istream<CharT, Traits>& is, const CharT* fmt, year_month_day& ymd,
            std::basic_string<CharT, Traits, Alloc>* abbrev = nullptr,
            std::chrono::minutes* offset = nullptr);
```

Effects: Extracts `ymd` from `is` using the format string `fmt` as specified by the `from_stream` formatting flags. If `abbrev` is not equal to `nullptr`, the information parsed by `%Z` (if present) will be placed in `*abbrev`. If `offset` is not equal to `nullptr`, the information parsed by `%z` (if present) will be placed in `*offset`. If an invalid date is parsed, or a time of day which is outside of the conventional bounds, `failbit` will be set and `ymd` will not be altered. In particular `!ymd.ok()` is not an indication of error.

Returns: `is`.

`year_month_day_last`

Synopsis

```
class year_month_day_last
{
    date::year           y_;    // exposition only
    date::month_day_last mdl_;  // exposition only

public:
    constexpr year_month_day_last(const date::year& y,
                                  const date::month_day_last& mdl) noexcept;

    constexpr year_month_day_last& operator+=(const months& m) noexcept;
    constexpr year_month_day_last& operator-=(const months& m) noexcept;
    constexpr year_month_day_last& operator+=(const years& y) noexcept;
    constexpr year_month_day_last& operator-=(const years& y) noexcept;
    constexpr date::year           year()           const noexcept;
    constexpr date::month          month()          const noexcept;
    constexpr date::month_day_last month_day_last() const noexcept;
    constexpr date::day            day()            const noexcept;

    constexpr          operator sys_days() const noexcept;
    constexpr explicit operator local_days() const noexcept;
    constexpr bool ok() const noexcept;
};

constexpr bool operator==(const year_month_day_last& x, const year_month_day_last& y) noexcept;
constexpr bool operator!=(const year_month_day_last& x, const year_month_day_last& y) noexcept;
constexpr bool operator< (const year_month_day_last& x, const year_month_day_last& y) noexcept;
constexpr bool operator> (const year_month_day_last& x, const year_month_day_last& y) noexcept;
constexpr bool operator<=(const year_month_day_last& x, const year_month_day_last& y) noexcept;
constexpr bool operator>=(const year_month_day_last& x, const year_month_day_last& y) noexcept;

constexpr year_month_day_last operator+(const year_month_day_last& ymdl, const months& dm) noexcept;
constexpr year_month_day_last operator+(const months& dm, const year_month_day_last& ymdl) noexcept;
constexpr year_month_day_last operator+(const year_month_day_last& ymdl, const years& dy) noexcept;
constexpr year_month_day_last operator+(const years& dy, const year_month_day_last& ymdl) noexcept;
constexpr year_month_day_last operator-(const year_month_day_last& ymdl, const months& dm) noexcept;
constexpr year_month_day_last operator-(const year_month_day_last& ymdl, const years& dy) noexcept;

template<class CharT, class Traits>
std::basic_ostream<class CharT, class Traits>&
operator<<(std::basic_ostream<class CharT, class Traits>& os, const year_month_day_last& ymdl);
```

Overview

`year_month_day_last` represents a specific `year`, `month`, and the last `day` of the `month`. `year_month_day_last` is a field-based time point with a resolution of `days`, except that it is restricted to pointing to the last day of a year and month. One can observe each field. The `day` field is computed on demand. `year_month_day_last` supports `years` and `months` oriented arithmetic, but not `days` oriented arithmetic. For the latter, there is a conversion to `sys_days` which efficiently supports `days` oriented arithmetic. `year_month_day_last` is equality and less-than comparable.

Specification

`year_month_day_last` is a trivially copyable class type.
`year_month_day_last` is a standard-layout class type.
`year_month_day_last` is a literal class type.

```
constexpr year_month_day_last::year_month_day_last(const date::year& y,
                                                   const date::month_day_last& mdl) noexcept;
```

> *Effects:* Constructs an object of type `year_month_day_last` by constructing `y_` with `y` and `mdl_` with `mdl`.

```
constexpr year_month_day_last& year_month_day_last::operator+=(const months& m) noexcept;
```

> *Effects:* `*this = *this + m;`.
>
> *Returns:* `*this`.

```
constexpr year_month_day_last& year_month_day_last::operator-=(const months& m) noexcept;
```

> *Effects:* `*this = *this - m;`.
>
> *Returns:* `*this`.

```
constexpr year_month_day_last& year_month_day_last::operator+=(const years& y) noexcept;
```

> *Effects:* `*this = *this + y;`.
>
> *Returns:* `*this`.

```
constexpr year_month_day_last& year_month_day_last::operator-=(const years& y) noexcept;
```

> *Effects:* `*this = *this - y;`.
>
> *Returns:* `*this`.

```
constexpr year year_month_day_last::year() const noexcept;
```

> *Returns:* `y_`.

```
constexpr month year_month_day_last::month() const noexcept;
```

Returns: `mdl_.month()`.

```
constexpr month_day_last year_month_day_last::month_day_last() const noexcept;
```

Returns: `mdl_`.

```
constexpr day year_month_day_last::day() const noexcept;
```

Returns: A `day` representing the last day of the `year`, `month` pair represented by `*this`.

```
constexpr year_month_day_last::operator sys_days() const noexcept;
```

Effects: Equivalent to: `return sys_days{year()/month()/day()};`

```
constexpr explicit year_month_day_last::operator local_days() const noexcept;
```

Effects: Equivalent to: `return local_days{sys_days{*this}.time_since_epoch()};`

```
constexpr bool year_month_day_last::ok() const noexcept;
```

Returns: `y_.ok() && mdl_.ok()`.

```
constexpr bool operator==(const year_month_day_last& x, const year_month_day_last& y) noexcept;
```

Returns: `x.year() == y.year() && x.month_day_last() == y.month_day_last()`.

```
constexpr bool operator!=(const year_month_day_last& x, const year_month_day_last& y) noexcept;
```

Returns: `!(x == y)`.

```
constexpr bool operator< (const year_month_day_last& x, const year_month_day_last& y) noexcept;
```

Returns: If `x.year() < y.year()`, returns `true`. Else if `x.year() > y.year()` returns `false`. Else returns `x.month_day_last() < y.month_day_last()`.

```
constexpr bool operator> (const year_month_day_last& x, const year_month_day_last& y) noexcept;
```

Returns: `y < x`.

```
constexpr bool operator<=(const year_month_day_last& x, const year_month_day_last& y) noexcept;
```

Returns: `!(y < x)`.

```
constexpr bool operator>=(const year_month_day_last& x, const year_month_day_last& y) noexcept;
```

Returns: `!(x < y)`.

```
constexpr year_month_day_last operator+(const year_month_day_last& ymdl, const months& dm) noexcept;
```

Effects: Equivalent to: `return (ymdl.year() / ymdl.month() + dm) / last`.

```
constexpr year_month_day_last operator+(const months& dm, const year_month_day_last& ymdl) noexcept;
```

Returns: `ymdl + dm`.

```
constexpr year_month_day_last operator-(const year_month_day_last& ymdl, const months& dm) noexcept;
```

Returns: `ymdl + (-dm)`.

```
constexpr year_month_day_last operator+(const year_month_day_last& ymdl, const years& dy) noexcept;
```

Returns: `{ymdl.year()+dy, ymdl.month_day_last()}`.

```
constexpr year_month_day_last operator+(const years& dy, const year_month_day_last& ymdl) noexcept;
```

Returns: `ymdl + dy`.

```
constexpr year_month_day_last operator-(const year_month_day_last& ymdl, const years& dy) noexcept;
```

Returns: `ymdl + (-dy)`.

```
template<class CharT, class Traits>
std::basic_ostream<class CharT, class Traits>&
operator<<(std::basic_ostream<class CharT, class Traits>& os, const year_month_day_last& ymdl);
```

Effects: Inserts `os << ymdl.year() << '/' << ymdl.month_day_last()`.

Returns: `os`.

`year_month_weekday`

Synopsis

```
class year_month_weekday
{
    date::year            y_;    // exposition only
    date::month           m_;    // exposition only
    date::weekday_indexed wdi_;  // exposition only

public:
    year_month_weekday() = default;
    constexpr year_month_weekday(const date::year& y, const date::month& m,
                                 const date::weekday_indexed& wdi) noexcept;
    constexpr year_month_weekday(const sys_days& dp) noexcept;
    constexpr explicit year_month_weekday(const local_days& dp) noexcept;

    constexpr year_month_weekday& operator+=(const months& m) noexcept;
    constexpr year_month_weekday& operator-=(const months& m) noexcept;
    constexpr year_month_weekday& operator+=(const years& y) noexcept;
    constexpr year_month_weekday& operator-=(const years& y) noexcept;

    constexpr date::year year() const noexcept;
    constexpr date::month month() const noexcept;
    constexpr date::weekday weekday() const noexcept;
    constexpr unsigned index() const noexcept;
    constexpr date::weekday_indexed weekday_indexed() const noexcept;

    constexpr          operator sys_days() const noexcept;
    constexpr explicit operator local_days() const noexcept;
    constexpr bool ok() const noexcept;
};

constexpr bool operator==(const year_month_weekday& x, const year_month_weekday& y) noexcept;
constexpr bool operator!=(const year_month_weekday& x, const year_month_weekday& y) noexcept;

constexpr year_month_weekday operator+(const year_month_weekday& ymwd, const months& dm) noexcept;
constexpr year_month_weekday operator+(const months& dm, const year_month_weekday& ymwd) noexcept;
constexpr year_month_weekday operator+(const year_month_weekday& ymwd, const years& dy) noexcept;
constexpr year_month_weekday operator+(const years& dy, const year_month_weekday& ymwd) noexcept;
constexpr year_month_weekday operator-(const year_month_weekday& ymwd, const months& dm) noexcept;
constexpr year_month_weekday operator-(const year_month_weekday& ymwd, const years& dy) noexcept;

template<class CharT, class Traits>
std::basic_ostream<class CharT, class Traits>&
operator<<(std::basic_ostream<class CharT, class Traits>& os, const year_month_weekday& ymwdi);
```

Overview

`year_month_weekday` represents a specific `year`, `month`, and nth `weekday` of the `month`. `year_month_weekday` is a field-based time point with a resolution of `days`. One can observe each field. `year_month_weekday` supports `years` and `months` oriented arithmetic, but not `days` oriented arithmetic. For the latter, there is a conversion to `sys_days` which efficiently supports `days` oriented arithmetic. `year_month_weekday` is equality comparable.

Specification

`year_month_weekday` is a trivially copyable class type.
`year_month_weekday` is a standard-layout class type.
`year_month_weekday` is a literal class type.

```
constexpr year_month_weekday::year_month_weekday(const date::year& y, const date::month& m,
                                                 const date::weekday_indexed& wdi) noexcept;
```

Effects: Constructs an object of type `year_month_weekday` by constructing `y_` with `y`, `m_` with `m`, and `wdi_` with `wdi`.

```
constexpr year_month_weekday(const sys_days& dp) noexcept;
```

Effects: Constructs an object of type `year_month_weekday` which corresponds to the date represented by `dp`.

Remarks: For any value of `year_month_weekday`, `ymdl`, for which `ymdl.ok()` is `true`, this equality will also be `true`: `ymdl == year_month_weekday{sys_days{ymdl}}`.

```
constexpr explicit year_month_weekday(const local_days& dp) noexcept;
```

Effects: Constructs an object of type `year_month_weekday` which corresponds to the date represented by `dp`.

Remarks: Equivalent to constructing with `sys_days{dp.time_since_epoch()}`.

```
constexpr year_month_weekday& year_month_weekday::operator+=(const months& m) noexcept;
```

Effects: `*this = *this + m;`.

Returns: `*this`.

```
constexpr year_month_weekday& year_month_weekday::operator-=(const months& m) noexcept;
```

Effects: `*this = *this - m;`.

Returns: `*this`.

```
constexpr year_month_weekday& year_month_weekday::operator+=(const years& y) noexcept;
```

Effects: `*this = *this + y;`.

Returns: `*this`.

```
constexpr year_month_weekday& year_month_weekday::operator-=(const years& y) noexcept;
```

Effects: `*this = *this - y;`.

Returns: `*this`.

```
constexpr year year_month_weekday::year() const noexcept;
```

Returns: `y_`.

```
constexpr month year_month_weekday::month() const noexcept;
```

Returns: `m_`.

```
constexpr weekday year_month_weekday::weekday() const noexcept;
```

Returns: `wdi_.weekday()`.

```
constexpr unsigned year_month_weekday::index() const noexcept;
```

Returns: `wdi_.index()`.

```
constexpr weekday_indexed year_month_weekday::weekday_indexed() const noexcept;
```

Returns: `wdi_`.

```
constexpr year_month_weekday::operator sys_days() const noexcept;
```

Returns: If `y_.ok() && m_.ok() && wdi_.weekday().ok()`, returns a `sys_days` which represents the date `(index() - 1)*7` days after the first `weekday()` of `year()/month()`. If `index()` is `0` the returned `sys_days` represents the date 7 days prior to the first `weekday()` of `year()/month()`. Otherwise, `!y_.ok() || !m_.ok() || !wdi_.weekday().ok()` and the returned value is unspecified.

```
constexpr explicit year_month_weekday::operator local_days() const noexcept;
```

Effects: Equivalent to: `return local_days{sys_days{*this}.time_since_epoch()};`

```
constexpr bool year_month_weekday::ok() const noexcept;
```

Returns: If `y_.ok()` or `m_.ok()` or `wdi_.ok()` returns `false`, returns `false`. Else if `*this` represents a valid date, returns `true`, else returns `false`.

```
constexpr bool operator==(const year_month_weekday& x, const year_month_weekday& y) noexcept;
```

Returns: `x.year() == y.year() && x.month() == y.month() && x.weekday_indexed() == y.weekday_indexed()`.

```
constexpr bool operator!=(const year_month_weekday& x, const year_month_weekday& y) noexcept;
```

Returns: `!(x == y)`.

```
constexpr year_month_weekday operator+(const year_month_weekday& ymwd, const months& dm) noexcept;
```

Returns: `(ymwd.year() / ymwd.month() + dm) / ymwd.weekday_indexed()`.

```
constexpr year_month_weekday operator+(const months& dm, const year_month_weekday& ymwd) noexcept;
```

Returns: `ymwd + dm`.

```
constexpr year_month_weekday operator-(const year_month_weekday& ymwd, const months& dm) noexcept;
```

Returns: `ymwd + (-dm)`.

```
constexpr year_month_weekday operator+(const year_month_weekday& ymwd, const years& dy) noexcept;
```

Returns: `{ymwd.year()+dy, ymwd.month(), ymwd.weekday_indexed()}`.

```
constexpr year_month_weekday operator+(const years& dy, const year_month_weekday& ymwd) noexcept;
```

Returns: `ymwd + dm`.

```
constexpr year_month_weekday operator-(const year_month_weekday& ymwd, const years& dy) noexcept;
```

Returns: `ymwd + (-dm)`.

```
template<class CharT, class Traits>
std::basic_ostream<class CharT, class Traits>&
operator<<(std::basic_ostream<class CharT, class Traits>& os, const year_month_weekday& ymwd);
```

Effects: Inserts `os << ymwdi.year() << '/' << ymwdi.month() << '/' << ymwdi.weekday_indexed()`.

Returns: `os`.

`year_month_weekday_last`

Synopsis

```
class year_month_weekday_last
{
    date::year         y_;    // exposition only
    date::month        m_;    // exposition only
    date::weekday_last wdl_;  // exposition only

public:
    constexpr year_month_weekday_last(const date::year& y, const date::month& m,
                                      const date::weekday_last& wdl) noexcept;

    constexpr year_month_weekday_last& operator+=(const months& m) noexcept;
    constexpr year_month_weekday_last& operator-=(const months& m) noexcept;
    constexpr year_month_weekday_last& operator+=(const years& y) noexcept;
    constexpr year_month_weekday_last& operator-=(const years& y) noexcept;

    constexpr date::year year() const noexcept;
    constexpr date::month month() const noexcept;
    constexpr date::weekday weekday() const noexcept;
    constexpr date::weekday_last weekday_last() const noexcept;

    constexpr          operator sys_days() const noexcept;
    constexpr explicit operator local_days() const noexcept;
    constexpr bool ok() const noexcept;
};

constexpr
bool
operator==(const year_month_weekday_last& x, const year_month_weekday_last& y) noexcept;

constexpr
bool
operator!=(const year_month_weekday_last& x, const year_month_weekday_last& y) noexcept;

constexpr
year_month_weekday_last
operator+(const year_month_weekday_last& ymwdl, const months& dm) noexcept;
```

```
constexpr
year_month_weekday_last
operator+(const months& dm, const year_month_weekday_last& ymwdl) noexcept;

constexpr
year_month_weekday_last
operator+(const year_month_weekday_last& ymwdl, const years& dy) noexcept;

constexpr
year_month_weekday_last
operator+(const years& dy, const year_month_weekday_last& ymwdl) noexcept;

constexpr
year_month_weekday_last
operator-(const year_month_weekday_last& ymwdl, const months& dm) noexcept;

constexpr
year_month_weekday_last
operator-(const year_month_weekday_last& ymwdl, const years& dy) noexcept;

template<class CharT, class Traits>
std::basic_ostream<class CharT, class Traits>&
operator<<(std::basic_ostream<class CharT, class Traits>& os, const year_month_weekday_last& ymwdl);
```

Overview

`year_month_weekday_last` represents a specific `year`, `month`, and last `weekday` of the `month`. `year_month_weekday_last` is a field-based time point with a resolution of `days`, except that it is restricted to pointing to the last weekday of a year and month. One can observe each field. `year_month_weekday_last` supports `years` and `months` oriented arithmetic, but not `days` oriented arithmetic. For the latter, there is a conversion to `sys_days` which efficiently supports `days` oriented arithmetic. `year_month_weekday_last` is equality comparable.

Specification

`year_month_weekday_last` is a trivially copyable class type.
`year_month_weekday_last` is a standard-layout class type.
`year_month_weekday_last` is a literal class type.

```
constexpr year_month_weekday_last::year_month_weekday_last(const date::year& y, const date::month& m,
                                                           const date::weekday_last& wdl) noexcept;
```

Effects: Constructs an object of type `year_month_weekday_last` by constructing `y_` with `y`, `m_` with `m`, and `wdl_` with `wdl`.

```
constexpr year_month_weekday_last& year_month_weekday_last::operator+=(const months& m) noexcept;
```

Effects: `*this = *this + m;`.

Returns: `*this`.

```
constexpr year_month_weekday_last& year_month_weekday_last::operator-=(const months& m) noexcept;
```

Effects: `*this = *this - m;`.

Returns: `*this`.

```
constexpr year_month_weekday_last& year_month_weekday_last::operator+=(const years& y) noexcept;
```

Effects: `*this = *this + y;`.

Returns: `*this`.

```
constexpr year_month_weekday_last& year_month_weekday_last::operator-=(const years& y) noexcept;
```

Effects: `*this = *this - y;`.

Returns: `*this`.

```
constexpr year year_month_weekday_last::year() const noexcept;
```

Returns: `y_`.

```
constexpr month year_month_weekday_last::month() const noexcept;
```

Returns: `m_`.

```
constexpr weekday year_month_weekday_last::weekday() const noexcept;
```

Returns: `wdl_.weekday()`.

```
constexpr weekday_last year_month_weekday_last::weekday_last() const noexcept;
```

Returns: `wdl_`.

```
constexpr year_month_weekday_last::operator sys_days() const noexcept;
```

Returns: If `ok() == true`, returns a `sys_days` which represents the last `weekday()` of `year()/month()`. Otherwise the returned value is unspecified.

```
constexpr explicit year_month_weekday_last::operator local_days() const noexcept;
```

Effects: Equivalent to: `return local_days{sys_days{*this}.time_since_epoch()};`

```
constexpr bool year_month_weekday_last::ok() const noexcept;
```

Returns: If `y_.ok() && m_.ok() && wdl_.ok()`.

```
constexpr bool operator==(const year_month_weekday_last& x, const year_month_weekday_last& y) noexcept;
```

Returns: `x.year() == y.year() && x.month() == y.month() && x.weekday_last() == y.weekday_last()`.

```
constexpr bool operator!=(const year_month_weekday_last& x, const year_month_weekday_last& y) noexcept;
```

Returns: `!(x == y)`.

```
constexpr year_month_weekday_last operator+(const year_month_weekday_last& ymwdl, const months& dm) noexcept;
```

Effects: Equivalent to: `return (ymwdl.year() / ymwdl.month() + dm) / ymwdl.weekday_last()`.

```
constexpr year_month_weekday_last operator+(const months& dm, const year_month_weekday_last& ymwdl) noexcept;
```

Returns: `ymwdl + dm`.

```
constexpr year_month_weekday_last operator-(const year_month_weekday_last& ymwdl, const months& dm) noexcept;
```

Returns: `ymwdl + (-dm)`.

```
constexpr year_month_weekday_last operator+(const year_month_weekday_last& ymwdl, const years& dy) noexcept;
```

Returns: `{ymwdl.year()+dy, ymwdl.month(), ymwdl.weekday_last()}`.

```
constexpr year_month_weekday_last operator+(const years& dy, const year_month_weekday_last& ymwdl) noexcept;
```

Returns: `ymwdl + dy`.

```
constexpr year_month_weekday_last operator-(const year_month_weekday_last& ymwdl, const years& dy) noexcept;
```

Returns: `ymwdl + (-dy)`.

```
template<class CharT, class Traits>
std::basic_ostream<class CharT, class Traits>&
operator<<(std::basic_ostream<class CharT, class Traits>& os, const year_month_weekday_last& ymwdl);
```

Effects: Inserts `os << ymwdl.year() << '/' << ymwdl.month() << '/' << ymwdl.weekday_last()`.

Returns: `os`.

date composition operators

To understand this API it is not necessary for you to memorize each of these operators. Indeed, that would be detrimental to understanding this API. Instead it is sufficient to known that this collection of operators implement constructions in 3 orders:

1. `y/m/d`
2. `m/d/y`
3. `d/m/y`

The first component in each order must be properly typed, the following components may be specified with the proper type or an `int`.

Anywhere a "day" is required one can also specify one of:

- `last`
- `weekday[i]`
- `weekday[last]`

Sub-field-types such as `year_month` and `month_day` can be created by simply not applying the second division operator for any of the three orders. For example:

```
year_month ym = 2015_y/April;
month_day md1 = April/4;
month_day md2 = 4_d/April;
```

Everything not intended as above is caught as a compile-time error, with the notable exception of an expression that consists of nothing but `int`, which of course has type `int`.

```
auto a = 2015/4/4;       // a == int(125)
auto b = 2015_y/4/4;     // b == year_month_day{year(2015), month(4), day(4)}
auto c = 2015_y/4_d/April; // error: invalid operands to binary expression ('date::year' and 'date::day')
auto d = 2015/April/4;     // error: invalid operands to binary expression ('int' and 'const date::month')
```

The last example may be clear to a human reader. But the compiler doesn't know if `2015` refers to a `year` or a `day`. Instead of guessing, the compiler flags it as an error.

In short, you will either write unambiguous and readable code, or you will get a compile-time error.

`year_month:`

```
constexpr year_month operator/(const year& y, const month& m) noexcept;
```

Returns: `{y, m}`.

```
constexpr year_month operator/(const year& y, int   m) noexcept;
```

Returns: `y / month(m)`.

`month_day:`

```
constexpr month_day operator/(const month& m, const day& d) noexcept;
```

Returns: `{m, d}`.

```
constexpr month_day operator/(const month& m, int d) noexcept;
```

Returns: `m / day(d)`.

```
constexpr month_day operator/(int m, const day& d) noexcept;
```

Returns: `month(m) / d`.

```
constexpr month_day operator/(const day& d, const month& m) noexcept;
```

Returns: `m / d`.

```
constexpr month_day operator/(const day& d, int m) noexcept;
```

Returns: `month(m) / d`.

`month_day_last:`

```
constexpr month_day_last operator/(const month& m, last_spec) noexcept;
```

Returns: `month_day_last{m}`.

```
constexpr month_day_last operator/(int m, last_spec) noexcept;
```

Returns: `month(m) / last`.

`constexpr month_day_last operator/(last_spec, const month& m) noexcept;`

Returns: `m / last`.

`constexpr month_day_last operator/(last_spec, int m) noexcept;`

Returns: `month(m) / last`.

`month_weekday:`

`constexpr month_weekday operator/(const month& m, const weekday_indexed& wdi) noexcept;`

Returns: `{m, wdi}`.

`constexpr month_weekday operator/(int m, const weekday_indexed& wdi) noexcept;`

Returns: `month(m) / wdi`.

`constexpr month_weekday operator/(const weekday_indexed& wdi, const month& m) noexcept;`

Returns: `m / wdi`.

`constexpr month_weekday operator/(const weekday_indexed& wdi, int m) noexcept;`

Returns: `month(m) / wdi`.

`month_weekday_last:`

`constexpr month_weekday_last operator/(const month& m, const weekday_last& wdl) noexcept;`

Returns: `{m, wdl}`.

`constexpr month_weekday_last operator/(int m, const weekday_last& wdl) noexcept;`

Returns: `month(m) / wdl`.

`constexpr month_weekday_last operator/(const weekday_last& wdl, const month& m) noexcept;`

Returns: `m / wdl`.

`constexpr month_weekday_last operator/(const weekday_last& wdl, int m) noexcept;`

Returns: `month(m) / wdl`.

`year_month_day:`

`constexpr year_month_day operator/(const year_month& ym, const day& d) noexcept;`

Returns: `{ym.year(), ym.month(), d}`.

`constexpr year_month_day operator/(const year_month& ym, int d) noexcept;`

Returns: `ym / day(d)`.

`constexpr year_month_day operator/(const year& y, const month_day& md) noexcept;`

Returns: `y / md.month() / md.day()`.

`constexpr year_month_day operator/(int y, const month_day& md) noexcept;`

Returns: `year(y) / md`.

`constexpr year_month_day operator/(const month_day& md, const year& y) noexcept;`

Returns: `y / md`.

```
constexpr year_month_day operator/(const month_day& md, int y) noexcept;
```

Returns: `year(y) / md`.

`year_month_day_last:`

```
constexpr year_month_day_last operator/(const year_month& ym, last_spec) noexcept;
```

Returns: `{ym.year(), month_day_last{ym.month()}}`.

```
constexpr year_month_day_last operator/(const year& y, const month_day_last& mdl) noexcept;
```

Returns: `{y, mdl}`.

```
constexpr year_month_day_last operator/(int y, const month_day_last& mdl) noexcept;
```

Returns: `year(y) / mdl`.

```
constexpr year_month_day_last operator/(const month_day_last& mdl, const year& y) noexcept;
```

Returns: `y / mdl`.

```
constexpr year_month_day_last operator/(const month_day_last& mdl, int y) noexcept;
```

Returns: `year(y) / mdl`.

`year_month_weekday:`

```
constexpr year_month_weekday operator/(const year_month& ym, const weekday_indexed& wdi) noexcept;
```

Returns: `{ym.year(), ym.month(), wdi}`.

```
constexpr year_month_weekday operator/(const year& y, const month_weekday& mwd) noexcept;
```

Returns: `{y, mwd.month(), mwd.weekday_indexed()}`.

```
constexpr year_month_weekday operator/(int y, const month_weekday& mwd) noexcept;
```

Returns: `year(y) / mwd`.

```
constexpr year_month_weekday operator/(const month_weekday& mwd, const year& y) noexcept;
```

Returns: `y / mwd`.

```
constexpr year_month_weekday operator/(const month_weekday& mwd, int y) noexcept;
```

Returns: `year(y) / mwd`.

`year_month_weekday_last:`

```
constexpr year_month_weekday_last operator/(const year_month& ym, const weekday_last& wdl) noexcept;
```

Returns: `{ym.year(), ym.month(), wdl}`.

```
constexpr year_month_weekday_last operator/(const year& y, const month_weekday_last& mwdl) noexcept;
```

Returns: `{y, mwdl.month(), mwdl.weekday_last()}`.

```
constexpr year_month_weekday_last operator/(int y, const month_weekday_last& mwdl) noexcept;
```

Returns: `year(y) / mwdl`.

```
constexpr year_month_weekday_last operator/(const month_weekday_last& mwdl, const year& y) noexcept;
```

Returns: `y / mwdl`.

```
constexpr year_month_weekday_last operator/(const month_weekday_last& mwdl, int y) noexcept;
```

Returns: `year(y) / mwdl`.

`time_of_day`

Overview

```
template <class Duration> class time_of_day;
```

The `time_of_day` class breaks a `std::chrono::duration` which represents the time elapsed since midnight, into a "broken" down time such as hours:minutes:seconds. The `Duration` template parameter dictates the precision to which the time is broken down. This can vary from a course precision of hours to a very fine precision of nanoseconds.

There are 4 specializations of `time_of_day` to handle four precisions:

1. `time_of_day<std::chrono::hours>`

   This specialization handles hours since midnight.

2. `time_of_day<std::chrono::minutes>`

   This specialization handles hours:minutes since midnight.

3. `time_of_day<std::chrono::seconds>`

   This specialization handles hours:minutes:seconds since midnight.

4. `time_of_day<std::chrono::duration<Rep, Period>>`

   This specialization is restricted to `Periods` that fractions of a second to represent. Typical uses are with milliseconds, microseconds and nanoseconds. This specialization handles hours:minute:seconds.fractional_seconds since midnight.

Specification

```
enum {am = 1, pm};
```

Each specialization of `time_of_day` is a trivially copyable class type.
Each specialization of `time_of_day` is a standard-layout class type.
Each specialization of `time_of_day` is a literal class type.

```
time_of_day<std::chrono::hours>
{
public:
    using precision = std::chrono::hours;

    constexpr time_of_day() noexcept;
    constexpr explicit time_of_day(std::chrono::hours since_midnight) noexcept;

    constexpr std::chrono::hours hours() const noexcept;
    constexpr unsigned mode() const noexcept;

    constexpr explicit operator precision() const noexcept;
    constexpr precision to_duration() const noexcept;

    void make24() noexcept;
    void make12() noexcept;
};
```

```
constexpr time_of_day<std::chrono::hours>::time_of_day() noexcept;
```

Effects: Constructs an object of type `time_of_day` in 24-hour format corresponding to 00:00:00 hours after 00:00:00.

Postconditions: `hours()` returns `0h`. `mode()` returns `0`.

```
constexpr explicit time_of_day<std::chrono::hours>::time_of_day(std::chrono::hours since_midnight) noexcept;
```

Effects: Constructs an object of type `time_of_day` in 24-hour format corresponding to `since_midnight` hours after 00:00:00.

Postconditions: `hours()` returns the integral number of hours `since_midnight` is after 00:00:00. `mode()` returns `0`.

```
constexpr std::chrono::hours time_of_day<std::chrono::hours>::hours() const noexcept;
```

Returns: The stored hour of `*this`.

```
constexpr unsigned time_of_day<std::chrono::hours>::mode() const noexcept;
```

Returns: 0 if `*this` is in 24-hour format. Otherwise returns `am` or `pm` corresponding to whether this represents a before-noon time or afternoon time.

```
constexpr explicit time_of_day<std::chrono::hours>::operator precision() const noexcept;
```

Returns: The number of hours since midnight.

```
constexpr precision to_duration() const noexcept;
```

Returns: `precision{*this}`.

```
void time_of_day<std::chrono::hours>::make24() noexcept;
```

Effects: If `*this` is a 12-hour time, converts to a 24-hour time. Otherwise, no effects.

```
void time_of_day<std::chrono::hours>::make12() noexcept;
```

Effects: If `*this` is a 24-hour time, converts to a 12-hour time. Otherwise, no effects.

```
template<class CharT, class Traits>
std::basic_ostream<class CharT, class Traits>&
operator<<(std::basic_ostream<class CharT, class Traits>& os, const time_of_day<std::chrono::hours>& t);
```

Effects: If `t` is a 24-hour time, outputs to `os` according to the `strftime` format: "%H00". "%H" will emit a leading 0 for hours less than 10. Else `t` is a 12-hour time, outputs to `os` according to the `strftime` format: "%I%p" according to the C locale, except that no leading zero is output for hours less than 10.

Returns: `os`.

Example:

```
0100  // 1 in the morning in 24-hour format
1800  // 6 in the evening in 24-hour format
1am   // 1 in the morning in 12-hour format
6pm   // 6 in the evening in 12-hour format
```

```
time_of_day<std::chrono::minutes>
{
public:
    using precision = std::chrono::minutes;

    constexpr time_of_day() noexcept;
    constexpr explicit time_of_day(std::chrono::minutes since_midnight) noexcept;

    constexpr std::chrono::hours   hours() const noexcept;
    constexpr std::chrono::minutes minutes() const noexcept;
    constexpr unsigned mode() const noexcept;

    constexpr explicit operator precision() const noexcept;
    constexpr precision to_duration() const noexcept;

    void make24() noexcept;
    void make12() noexcept;
};
```

```
constexpr time_of_day<std::chrono::minutes>::time_of_day() noexcept;
```

Effects: Constructs an object of type `time_of_day` in 24-hour format corresponding to 0 minutes after 00:00:00.

Postconditions: `hours()` returns `0h`. `minutes()` returns `0min`. `mode()` returns `0`.

```
constexpr explicit time_of_day<std::chrono::minutes>::time_of_day(std::chrono::minutes since_midnight) noexcept;
```

Effects: Constructs an object of type `time_of_day` in 24-hour format corresponding to `since_midnight` minutes after 00:00:00.

Postconditions: `hours()` returns the integral number of hours `since_midnight` is after 00:00:00. `minutes()` returns the integral number of minutes `since_midnight` is after (00:00:00 + `hours()`). `mode()` returns `0`.

```
constexpr std::chrono::hours time_of_day<std::chrono::minutes>::hours() const noexcept;
```

Returns: The stored hour of `*this`.

```
constexpr std::chrono::minutes time_of_day<std::chrono::minutes>::minutes() const noexcept;
```

Returns: The stored minute of `*this`.

```
constexpr unsigned time_of_day<std::chrono::minutes>::mode() const noexcept;
```

Returns: 0 if `*this` is in 24-hour format. Otherwise returns `am` or `pm` corresponding to whether this represents a before-noon time or afternoon time.

```
constexpr explicit time_of_day<std::chrono::minutes>::operator precision() const noexcept;
```

Returns: The number of minutes since midnight.

```
constexpr precision to_duration() const noexcept;
```

Returns: `precision{*this}`.

```
void time_of_day<std::chrono::minutes>::make24() noexcept;
```

Effects: If `*this` is a 12-hour time, converts to a 24-hour time. Otherwise, no effects.

```
void time_of_day<std::chrono::minutes>::make12() noexcept;
```

Effects: If `*this` is a 24-hour time, converts to a 12-hour time. Otherwise, no effects.

```
template<class CharT, class Traits>
std::basic_ostream<class CharT, class Traits>&
operator<<(std::basic_ostream<class CharT, class Traits>& os, const time_of_day<std::chrono::minutes>& t);
```

Effects: If `t` is a 24-hour time, outputs to `os` according to the `strftime` format: "%H:%M". "%H" will emit a leading 0 for hours less than 10. Else `t` is a 12-hour time, outputs to `os` according to the `strftime` format: "%I:%M%p" according to the C locale, except that no leading zero is output for hours less than 10.

Returns: `os`.

Example:

```
01:08   // 1:08 in the morning in 24-hour format
18:15   // 6:15 in the evening in 24-hour format
1:08am  // 1:08 in the morning in 12-hour format
6:15pm  // 6:15 in the evening in 12-hour format
```

```
time_of_day<std::chrono::seconds>
{
public:
    using precision = std::chrono::seconds;

    constexpr time_of_day() noexcept;
    constexpr explicit time_of_day(std::chrono::seconds since_midnight) noexcept;

    constexpr std::chrono::hours   hours() const noexcept;
    constexpr std::chrono::minutes minutes() const noexcept;
    constexpr std::chrono::seconds seconds() const noexcept;
    constexpr unsigned mode() const noexcept;

    constexpr explicit operator precision() const noexcept;
    constexpr precision to_duration() const noexcept;

    void make24() noexcept;
    void make12() noexcept;
};
```

```
constexpr time_of_day<std::chrono::seconds>::time_of_day() noexcept;
```

Effects: Constructs an object of type `time_of_day` in 24-hour format corresponding to 0 seconds after 00:00:00.

Postconditions: `hours()` returns `0h`. `minutes()` returns `0min`. `seconds()` returns `0s`. `mode()` returns `0`.

```
constexpr explicit time_of_day<std::chrono::seconds>::time_of_day(std::chrono::seconds since_midnight) noexcept;
```

Effects: Constructs an object of type `time_of_day` in 24-hour format corresponding to `since_midnight` seconds after 00:00:00.

> *Postconditions:* `hours()` returns the integral number of hours `since_midnight` is after 00:00:00. `minutes()` returns the integral number of minutes `since_midnight` is after (00:00:00 + `hours()`). `seconds()` returns the integral number of seconds `since_midnight` is after (00:00:00 + `hours()` + `minutes()`). `mode()` returns `0`.

```
constexpr std::chrono::hours time_of_day<std::chrono::seconds>::hours() const noexcept;
```

> *Returns:* The stored hour of `*this`.

```
constexpr std::chrono::minutes time_of_day<std::chrono::seconds>::minutes() const noexcept;
```

> *Returns:* The stored minute of `*this`.

```
constexpr std::chrono::seconds time_of_day<std::chrono::seconds>::seconds() const noexcept;
```

> *Returns:* The stored second of `*this`.

```
constexpr unsigned time_of_day<std::chrono::seconds>::mode() const noexcept;
```

> *Returns:* 0 if `*this` is in 24-hour format. Otherwise returns `am` or `pm` corresponding to whether this represents a before-noon time or afternoon time.

```
constexpr explicit time_of_day<std::chrono::seconds>::operator precision() const noexcept;
```

> *Returns:* The number of seconds since midnight.

```
constexpr precision to_duration() const noexcept;
```

> *Returns:* `precision{*this}`.

```
void time_of_day<std::chrono::seconds>::make24() noexcept;
```

> *Effects:* If `*this` is a 12-hour time, converts to a 24-hour time. Otherwise, no effects.

```
void time_of_day<std::chrono::seconds>::make12() noexcept;
```

> *Effects:* If `*this` is a 24-hour time, converts to a 12-hour time. Otherwise, no effects.

```
template<class CharT, class Traits>
std::basic_ostream<class CharT, class Traits>&
operator<<(std::basic_ostream<class CharT, class Traits>& os, const time_of_day<std::chrono::seconds>& t);
```

> *Effects:* If `t` is a 24-hour time, outputs to `os` according to the `strftime` format: "%H:%M%S". "%H" will emit a leading 0 for hours less than 10. Else `t` is a 12-hour time, outputs to `os` according to the `strftime` format: "%I:%M%S%p" according to the C locale, except that no leading zero is output for hours less than 10.
>
> *Returns:* `os`.
>
> *Example:*

```
01:08:03   // 1:08:03 in the morning in 24-hour format
18:15:45   // 6:15:45 in the evening in 24-hour format
1:08:03am  // 1:08:03 in the morning in 12-hour format
6:15:45pm  // 6:15:45 in the evening in 12-hour format
```

```
time_of_day<std::chrono::duration<Rep, Period>>
{
public:
    using precision = The decimal-based duration used to format

    constexpr time_of_day() noexcept;
    constexpr explicit time_of_day(std::chrono::duration<Rep, Period> since_midnight) noexcept;

    constexpr std::chrono::hours   hours() const noexcept;
    constexpr std::chrono::minutes minutes() const noexcept;
    constexpr std::chrono::seconds seconds() const noexcept;
    constexpr precision subseconds() const noexcept;
    constexpr unsigned mode() const noexcept;

    constexpr explicit operator precision() const noexcept;
    constexpr precision to_duration() const noexcept;

    void make24() noexcept;
    void make12() noexcept;
};
```

This specialization shall not exist unless `Period < 1s`.

```
constexpr
time_of_day<std::chrono::duration<Rep, Period>>::time_of_day() noexcept;
```

Effects: Constructs an object of type `time_of_day` in 24-hour format corresponding to 0 fractional seconds after 00:00:00.

Postconditions: `hours()` returns `0h`. `minutes()` returns `0min`. `seconds()` returns `0s`. `subseconds()` returns 0 fractional precision seconds. `mode()` returns `0`.

```
constexpr explicit
time_of_day<std::chrono::duration<Rep, Period>>::time_of_day(std::chrono::duration<Rep, Period> since_midnight) noexcept;
```

Effects: Constructs an object of type `time_of_day` in 24-hour format corresponding to `since_midnight precision` fractional seconds after 00:00:00.

Postconditions: `hours()` returns the integral number of hours `since_midnight` is after 00:00:00. `minutes()` returns the integral number of minutes `since_midnight` is after (00:00:00 + `hours()`). `seconds()` returns the integral number of seconds `since_midnight` is after (00:00:00 + `hours()` + `minutes()`). `subseconds()` returns the integral number of fractional precision seconds `since_midnight` is after (00:00:00 + `hours()` + `minutes()` + `seconds`). `mode()` returns `0`.

```
constexpr std::chrono::hours time_of_day<std::chrono::duration<Rep, Period>>::hours() const noexcept;
```

Returns: The stored hour of `*this`.

```
constexpr std::chrono::minutes time_of_day<std::chrono::duration<Rep, Period>>::minutes() const noexcept;
```

Returns: The stored minute of `*this`.

```
constexpr std::chrono::seconds time_of_day<std::chrono::duration<Rep, Period>>::seconds() const noexcept;
```

Returns: The stored second of `*this`.

```
constexpr
std::chrono::duration<Rep, Period>
time_of_day<std::chrono::duration<Rep, Period>>::subseconds() const noexcept;
```

Returns: The stored subsecond of `*this`.

```
constexpr unsigned time_of_day<std::chrono::duration<Rep, Period>>::mode() const noexcept;
```

Returns: 0 if `*this` is in 24-hour format. Otherwise returns `am` or `pm` corresponding to whether this represents a before-noon time or afternoon time.

```
constexpr explicit time_of_day<std::chrono::duration<Rep, Period>>::operator precision() const noexcept;
```

Returns: The number of subseconds since midnight.

```
constexpr precision to_duration() const noexcept;
```

Returns: `precision{*this}`.

```
void time_of_day<std::chrono::duration<Rep, Period>>::make24() noexcept;
```

Effects: If `*this` is a 12-hour time, converts to a 24-hour time. Otherwise, no effects.

```
void time_of_day<std::chrono::duration<Rep, Period>>::make12() noexcept;
```

Effects: If `*this` is a 24-hour time, converts to a 12-hour time. Otherwise, no effects.

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

Effects: If `t` is a 24-hour time, outputs to `os` according to the `strftime` format: "%H:%M%S.%s". "%H" will emit a leading 0 for hours less than 10. "%s" is not a `strftime` code and represents the fractional seconds. Else `t` is a 12-hour time, outputs to `os` according to the `strftime` format: "%I:%M%S.%s%p" according to the C locale, except that no leading zero is output for hours less than 10.

Returns: `os`.

Example:

```
01:08:03.007   // 1:08:03.007 in the morning in 24-hour format (assuming millisecond precision)
18:15:45.123   // 6:15:45.123 in the evening in 24-hour format (assuming millisecond precision)
1:08:03.007am  // 1:08:03.007 in the morning in 12-hour format (assuming millisecond precision)
6:15:45.123pm  // 6:15:45.123 in the evening in 12-hour format (assuming millisecond precision)
```

`make_time`

```
template <class Rep, class Period>
constexpr
time_of_day<std::chrono::duration<Rep, Period>>
make_time(std::chrono::duration<Rep, Period> d) noexcept;
```

Returns: `time_of_day<std::chrono::duration<Rep, Period>>(d)`.

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

Remarks: This operator shall not participate in overload resolution if `std::chrono::treat_as_floating_point<typename Duration::rep>::value` is true.

Effects: If `Duration{1} < days{1}`:

```
auto const dp = floor<days>(tp);
os << year_month_day(dp) << ' ' << make_time(tp-dp);
```

Otherwise:

```
os << year_month_day(floor<days>(tp));
```

Returns: `os`.

`to_stream` formatting

Each flag begins with a `%`. Some flags can be modified by `E` or `O`. During streaming each flag is replaced according to the table below. All characters in the format string which are not represented in the table below are inserted unchanged into the stream.

`%a`	The locale's abbreviated weekday name. If the value does not contain a valid `weekday`, `std::ios::failbit` is set.
`%A`	The locale's full weekday name. If the value does not contain a valid `weekday`, `std::ios::failbit` is set.
`%b`	The locale's abbreviated month name. If the value does not contain a valid `month`, `std::ios::failbit` is set.
`%B`	The locale's full month name. If the value does not contain a valid `month`, `std::ios::failbit` is set.
`%c`	The locale's date and time representation. The modified command `%Ec` produces the locale's alternate date and time representation.
`%C`	The year divided by 100 using floored division. If the result is a single decimal digit, it is prefixed with `0`. The modified command `%EC` produces the locale's alternative representation of the century.
`%d`	The day of month as a decimal number. If the result is a single decimal digit, it is prefixed with `0`. The modified command `%Od` produces the locale's alternative representation.
`%D`	Equivalent to `%m/%d/%y`.
`%e`	The day of month as a decimal number. If the result is a single decimal digit, it is prefixed with a space. The modified command `%Oe` produces the locale's alternative representation.
`%F`	Equivalent to `%Y-%m-%d`.
`%g`	The last two decimal digits of the ISO week-based year. If the result is a single digit it is prefixed by `0`.
`%G`	The ISO week-based year as a decimal number. If the result is less than four digits it is left-padded with `0` to four digits.
`%h`	Equivalent to `%b`.
`%H`	The hour (24-hour clock) as a decimal number. If the result is a single digit, it is prefixed with `0`. The modified command `%OH` produces the locale's alternative representation.
`%I`	The hour (12-hour clock) as a decimal number. If the result is a single digit, it is prefixed with `0`. The modified

	command `%OI` produces the locale's alternative representation.
`%j`	The day of the year as a decimal number. Jan 1 is `001`. If the result is less than three digits, it is left-padded with `0` to three digits.
`%m`	The month as a decimal number. Jan is `01`. If the result is a single digit, it is prefixed with `0`. The modified command `%Om` produces the locale's alternative representation.
`%M`	The minute as a decimal number. If the result is a single digit, it is prefixed with `0`. The modified command `%OM` produces the locale's alternative representation.
`%n`	A newline character.
`%p`	The locale's equivalent of the AM/PM designations associated with a 12-hour clock.
`%r`	The locale's 12-hour clock time.
`%R`	Equivalent to `%H:%M`.
`%S`	Seconds as a decimal number. If the number of seconds is less than 10, the result is prefixed with `0`. If the precision of the input can not be exactly represented with seconds, then the format is a decimal floating point number with a fixed format and a precision matching that of the precision of the input (or to a microseconds precision if the conversion to floating point decimal seconds can not be made within 18 fractional digits). The character for the decimal point is localized according to the locale. The modified command `%OS` produces the locale's alternative representation.
`%t`	A horizontal-tab character.
`%T`	Equivalent to `%H:%M:%S`.
`%u`	The ISO weekday as a decimal number (1-7), where Monday is 1. The modified command `%Ou` produces the locale's alternative representation.
`%U`	The week number of the year as a decimal number. The first Sunday of the year is the first day of week `01`. Days of the same year prior to that are in week `00`. If the result is a single digit, it is prefixed with `0`. The modified command `%OU` produces the locale's alternative representation.
`%V`	The ISO week-based week number as a decimal number. If the result is a single digit, it is prefixed with `0`. The modified command `%OV` produces the locale's alternative representation.
`%w`	The weekday as a decimal number (0-6), where Sunday is 0. The modified command `%Ow` produces the locale's alternative representation.
`%W`	The week number of the year as a decimal number. The first Monday of the year is the first day of week `01`. Days of the same year prior to that are in week `00`. If the result is a single digit, it is prefixed with `0`. The modified command `%OW` produces the locale's alternative representation.
`%x`	The locale's date representation. The modified command `%Ex` produces the locale's alternate date representation.
`%X`	The locale's time representation. The modified command `%Ex` produces the locale's alternate time representation.
`%y`	The last two decimal digits of the year. If the result is a single digit it is prefixed by `0`.
`%Y`	The year as a decimal number. If the result is less than four digits it is left-padded with `0` to four digits.
`%z`	The offset from UTC in the ISO 8601 format. For example `-0430` refers to 4 hours 30 minutes behind UTC. If the offset is zero, `+0000` is used. The modified commands `%Ez` and `%Oz` insert a `:` between the hours and minutes: `-04:30`. If the offset information is not available, `failbit` will be set.
`%Z`	The time zone abbreviation. If the time zone abbreviation is not available, `failbit` will be set. `UTC` is used for `sys_time`.
`%%`	A `%` character.

format

```
template <class CharT, class Streamable>
std::basic_string<CharT>
format(const std::locale& loc, const CharT* fmt, const Streamable& tp);
```

Remarks: This function does not participate in overload resolution unless `to_stream(std::declval<std::basic_ostream<CharT>&>(), fmt, tp)` is valid.

Effects: Constructs a `std::basic_ostringstream<CharT> os` and imbues it with `loc`. Executes `os.exceptions(std::ios::failbit | std::ios::badbit)`. Calls `to_stream(os, fmt, tp)`.

Returns: `os.str()`.

```
template <class CharT, class Streamable>
std::basic_string<CharT>
format(const CharT* fmt, const Streamable& tp);
```

Remarks: This function does not participate in overload resolution unless `to_stream(std::declval<std::basic_ostream<CharT>&>(), fmt, tp)` is valid.

Effects: Constructs a `std::basic_ostringstream<CharT> os`. Executes `os.exceptions(std::ios::failbit | std::ios::badbit)`. Calls `to_stream(os, fmt, tp)`.

Returns: `os.str()`.

```
template <class CharT, class Traits, class Streamable>
std::basic_string<CharT>
format(const std::locale& loc, const std::basic_string<CharT, Traits>& fmt, const Streamable& tp);
```

Remarks: This function does not participate in overload resolution unless `to_stream(std::declval<std::basic_ostream<CharT>&>(), fmt.c_str(), tp)` is valid.

Effects: Constructs a `std::basic_ostringstream<CharT> os` and imbues it with `loc`. Executes `os.exceptions(std::ios::failbit | std::ios::badbit)`. Calls `to_stream(os, fmt.c_str(), tp)`.

Returns: `os.str()`.

```
template <class CharT, class Traits, class Streamable>
std::basic_string<CharT>
format(const std::basic_string<CharT, Traits>& fmt, const Streamable& tp);
```

Remarks: This function does not participate in overload resolution unless `to_stream(std::declval<std::basic_ostream<CharT>&>(), fmt.c_str(), tp)` is valid.

Effects: Constructs a `std::basic_ostringstream<CharT> os`. Executes `os.exceptions(std::ios::failbit | std::ios::badbit)`. Calls `to_stream(os, fmt.c_str(), tp)`.

Returns: `os.str()`.

`from_stream` formatting

All `from_stream` overloads behave as an unformatted input function. Each overload takes a format string containing ordinary characters and flags which have special meaning. Each flag begins with a `%`. Some flags can be modified by `E` or `O`. During parsing each flag interprets characters as parts of date and time type according to the table below. Some flags can be modified by a width parameter which governs how many characters are parsed from the stream in interpreting the flag. All characters in the format string which are not represented in the table below, except for white space, are parsed unchanged from the stream. A white space character matches zero or more white space characters in the input stream. The C++ specification says that the parsing of month and weekday names is both locale sensitive *and* case insensitive. If you find this not to be the case, file a bug with your std::lib vendor.

`%a`	The locale's full or abbreviated case-insensitive weekday name.
`%A`	Equivalent to `%a`.
`%b`	The locale's full or abbreviated case-insensitive month name.
`%B`	Equivalent to `%b`.
`%c`	The locale's date and time representation. The modified command `%Ec` interprets the locale's alternate date and time representation.
`%C`	The century as a decimal number. The modified command `%NC` where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 2. Leading zeroes are permitted but not required. The modified commands `%EC` and `%OC` interpret the locale's alternative representation of the century.
`%d`	The day of the month as a decimal number. The modified command `%Nd` where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 2. Leading zeroes are permitted but not required. The modified command `%EC` interprets the locale's alternative representation of the day of the month.
`%D`	Equivalent to `%m/%d/%y`.
`%e`	Equivalent to `%d` and can be modified like `%d`.
`%F`	Equivalent to `%Y-%m-%d`. If modified with a width, the width is applied to only `%Y`.

`%g`	The last two decimal digits of the ISO week-based year. The modified command `%Ng` where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 2. Leading zeroes are permitted but not required.
`%G`	The ISO week-based year as a decimal number. The modified command `%NG` where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 4. Leading zeroes are permitted but not required.
`%h`	Equivalent to `%b`.
`%H`	The hour (24-hour clock) as a decimal number. The modified command `%NH` where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 2. Leading zeroes are permitted but not required. The modified command `%OH` interprets the locale's alternative representation.
`%I`	The hour (12-hour clock) as a decimal number. The modified command `%NI` where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 2. Leading zeroes are permitted but not required.
`%j`	The day of the year as a decimal number. Jan 1 is `1`. The modified command `%Nj` where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 3. Leading zeroes are permitted but not required.
`%m`	The month as a decimal number. Jan is `1`. The modified command `%Nm` where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 2. Leading zeroes are permitted but not required. The modified command `%Om` interprets the locale's alternative representation.
`%M`	The minutes as a decimal number. The modified command `%NM` where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 2. Leading zeroes are permitted but not required. The modified command `%OM` interprets the locale's alternative representation.
`%n`	Matches one white space character. [*Note:* `%n`, `%t` and a space, can be combined to match a wide range of white-space patterns. For example `"%n "` matches one or more white space charcters, and `"%n%t%t"` matches one to three white space characters. — *end note*]
`%p`	The locale's equivalent of the AM/PM designations associated with a 12-hour clock. The command `%I` must precede `%p` in the format string.
`%r`	The locale's 12-hour clock time.
`%R`	Equivalent to `%H:%M`.
`%S`	The seconds as a decimal number. The modified command `%NS` where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 2 if the input time has a precision convertible to seconds. Otherwise the default width is determined by the decimal precision of the input and the field is interpreted as a long double in a fixed format. If encountered, the locale determines the decimal point character. Leading zeroes are permitted but not required. The modified command `%OS` interprets the locale's alternative representation.
`%t`	Matches zero or one white space characters.
`%T`	Equivalent to `%H:%M:%S`.
`%u`	The ISO weekday as a decimal number (1-7), where Monday is 1. The modified command `%Nu` where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 1. Leading zeroes are permitted but not required. The modified command `%Ou` interprets the locale's alternative representation.
`%U`	The week number of the year as a decimal number. The first Sunday of the year is the first day of week `01`. Days of the same year prior to that are in week `00`. The modified command `%NU` where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 2. Leading zeroes are permitted but not required.
`%V`	The ISO week-based week number as a decimal number. The modified command `%NV` where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 2. Leading zeroes are permitted but not required.
`%w`	The weekday as a decimal number (0-6), where Sunday is 0. The modified command `%Nw` where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 1. Leading zeroes are permitted but not required. The modified command `%Ou` interprets the locale's alternative representation.
`%W`	The week number of the year as a decimal number. The first Monday of the year is the first day of week `01`. Days of the same year prior to that are in week `00`. The modified command `%NW` where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 2. Leading zeroes are permitted but not required.
`%x`	The locale's date representation. The modified command `%Ex` produces the locale's alternate date representation.

`%X`	The locale's time representation. The modified command `%Ex` produces the locale's alternate time representation.
`%y`	The last two decimal digits of the year. If the century is not otherwise specified (e.g. with `%C`), values in the range [69 - 99] are presumed to refer to the years [1969 - 1999], and values in the range [00 - 68] are presumed to refer to the years [2000 - 2068]. The modified command `%Ny` where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 2. Leading zeroes are permitted but not required. The modified commands `%Ey` and `%Oy` interpret the locale's alternative representation.
`%Y`	The year as a decimal number. The modified command `%NY` where `N` is a positive decimal integer specifies the maximum number of characters to read. If not specified, the default is 4. Leading zeroes are permitted but not required. The modified command `%EY` interprets the locale's alternative representation.
`%z`	The offset from UTC in the format `[+|-]hh[mm]`. For example `-0430` refers to 4 hours 30 minutes behind UTC. And `04` refers to 4 hours ahead of UTC. The modified commands `%Ez` and `%Oz` parse a `:` between the hours and minutes and leading zeroes on the hour field are optional: `[+|-]h[h][:mm]`. For example `-04:30` refers to 4 hours 30 minutes behind UTC. And `4` refers to 4 hours ahead of UTC.
`%Z`	The time zone abbreviation or name. A single word is parsed. This word can only contain characters from the *basic source character set* ([lex.charset] in the C++ standard) that are alphanumeric, or one of `'_'`, `'/'`, `'-'` or `'+'`.
`%%`	A `%` character is extracted.

parse

```
template <class Parsable, class CharT, class Traits, class Alloc>
unspecified
parse(const std::basic_string<CharT, Traits, Alloc>& format, Parsable& tp);
```

Remarks: This function does not participate in overload resolution unless `from_stream(std::declval<std::basic_istream<CharT, Traits>&>(), format.c_str(), tp)` is valid

Returns: A `basic_istream<CharT, Traits>` manipulator that on extraction from a `basic_istream<CharT, Traits> is` calls `from_stream(is, format.c_str(), tp)`.

```
template <class Parsable, class CharT, class Traits, class Alloc>
unspecified
parse(const std::basic_string<CharT, Traits, Alloc>& format, Parsable& tp,
      std::basic_string<CharT, Traits, Alloc>& abbrev);
```

Remarks: This function does not participate in overload resolution unless `from_stream(std::declval<std::basic_istream<CharT, Traits>&>(), format.c_str(), tp, &abbrev)` is valid

Returns: A `basic_istream<CharT, Traits>` manipulator that on extraction from a `basic_istream<CharT, Traits> is` calls `from_stream(is, format.c_str(), tp, &abbrev)`.

```
template <class Parsable, class CharT, class Traits, class Alloc>
unspecified
parse(const std::basic_string<CharT, Traits, Alloc>& format, Parsable& tp,
      std::chrono::minutes& offset);
```

Remarks: This function does not participate in overload resolution unless `from_stream(std::declval<std::basic_istream<CharT, Traits>&>(), format.c_str(), tp, nullptr, &offset)` is valid

Returns: A `basic_istream<CharT, Traits>` manipulator that on extraction from a `basic_istream<CharT, Traits> is` calls `from_stream(is, format.c_str(), tp, nullptr, &offset)`.

```
template <class Parsable, class CharT, class Traits>
unspecified
parse(const std::basic_string<CharT, Traits>& format, Parsable& tp,
      std::basic_string<CharT, Traits>& abbrev, std::chrono::minutes& offset);
```

Remarks: This function does not participate in overload resolution unless `from_stream(std::declval<std::basic_istream<CharT, Traits>&>(), format.c_str(), tp, &abbrev, &offset)` is valid

Returns: A `basic_istream<CharT, Traits>` manipulator that on extraction from a `basic_istream<CharT, Traits> is` calls `from_stream(is, format.c_str(), tp, &abbrev, &offset)`.

```
// const CharT* formats

template <class Parsable, class CharT>
unspecified
parse(const CharT* format, Parsable& tp);
```

Remarks: This function does not participate in overload resolution unless `from_stream(std::declval<std::basic_istream<CharT, Traits>&>(), format, tp)` is valid

Returns: A `basic_istream<CharT, Traits>` manipulator that on extraction from a `basic_istream<CharT, Traits> is` calls `from_stream(is, format, tp)`.

```
template <class Parsable, class CharT, class Traits, class Alloc>
unspecified
parse(const CharT* format, Parsable& tp, std::basic_string<CharT, Traits, Alloc>& abbrev);
```

Remarks: This function does not participate in overload resolution unless `from_stream(std::declval<std::basic_istream<CharT, Traits>&>(), format, tp, &abbrev)` is valid

Returns: A `basic_istream<CharT, Traits>` manipulator that on extraction from a `basic_istream<CharT, Traits> is` calls `from_stream(is, format, tp, &abbrev)`.

```
template <class Parsable, class CharT>
unspecified
parse(const CharT* format, Parsable& tp, std::chrono::minutes& offset);
```

Remarks: This function does not participate in overload resolution unless `from_stream(std::declval<std::basic_istream<CharT, Traits>&>(), format, tp, nullptr, &offset)` is valid

Returns: A `basic_istream<CharT, Traits>` manipulator that on extraction from a `basic_istream<CharT, Traits> is` calls `from_stream(is, format, tp, nullptr, &offset)`.

```
template <class Parsable, class CharT, class Traits, class Alloc>
unspecified
parse(const CharT* format, Parsable& tp,
      std::basic_string<CharT, Traits, Alloc>& abbrev, std::chrono::minutes& offset);
```

Remarks: This function does not participate in overload resolution unless `from_stream(std::declval<std::basic_istream<CharT, Traits>&>(), format, tp, &abbrev, &offset)` is valid

Returns: A `basic_istream<CharT, Traits>` manipulator that on extraction from a `basic_istream<CharT, Traits> is` calls `from_stream(is, format, tp, &abbrev, &offset)`.

Installation

Some of the parsing and formatting flags require help from your std::lib's `time_get` and `time_put` facets for localization help. If you are happy with being restricted to the `"C"` locale, and/or need to port this to an older platform that doesn't support the `time_get` and `time_put` facets, compile with `-DONLY_C_LOCALE=1`. This enables all of the streaming and parsing flags, but makes all of them assume the `"C"` locale, and removes the dependency on the `time_get` and `time_put` facets.