Mandating the Standard Library:
Clause 27 - Time library

With the adoption of P0788R3, we have a new way of specifying requirements for the library clauses of the standard. This is one of a series of papers reformulating the requirements into the new format. This effort was strongly influenced by the informational paper P1369R0.

The changes in this paper are of the following types:

— Change 'participate in overload resolution' wording into "Constraints" elements.
— Change 'Remarks' about thread safety into 'Synchronization'.
— Change some 'Remarks' into 'Ensures'.
— Change some 'Remarks' into 'Expects'.
— "<classname>isCpp17LessThanComparable" is changed to '"<classname> meets the Cpp17LessThanComparable requirements".
— Removal of non-essential effects wording of the form 'constructs an object of type <classname>.'

This paper covers Clause 27 (Time library). The entire clause is reproduced here, but the changes are in the following sections:

— time.clock.req 27.3
— time.duration.cons 27.5.1
— time.duration.nonmember 27.5.5
— time.duration.cast 27.5.7
— time.duration.alg 27.5.9
— time.point.cons 27.6.1
— time.point.cast 27.6.7
— time.clock.system.members 27.7.1.2
— time.clock.system.nonmembers 27.7.1.3
— time.clock.cast.sys 27.7.9.4
— time.clock.cast.utc 27.7.9.5
— time.clock.cast.fn 27.7.9.6
— time.cal.day.overview 27.8.3.1
— time.cal.month.overview 27.8.4.1
— time.cal.year.overview 27.8.5.1
— time.cal.wd.overview 27.8.6.1
— time.cal.wd.members 27.8.7.2
— time.cal.wdidx.members 27.8.7.2
— time.cal.wdlast.members 27.8.8.2
— time.cal.md.last 27.8.9.1
— time.cal.mdlast 27.8.10
— time.cal.mwd.members 27.8.11.2
— time.cal.mwd.members 27.8.11.2
— time.cal.mwd.last 27.8.12.2
— time.cal.ym.members 27.8.13.2
— time.cal.ym.members 27.8.13.2
— time.cal.ym.members 27.8.13.2
— time.cal.ym.members 27.8.13.2
— time.cal.ym.members 27.8.13.2
— time.cal.ym.members 27.8.13.2
— time.cal.ym.last 27.8.14.1
— time.cal.ym.members 27.8.14.2
— time.cal.ym.members 27.8.14.2
— time.cal.ym.members 27.8.14.2
— time.cal.ym.members 27.8.14.2
— time.cal.ym.members 27.8.14.2
— time.cal.ym.last 27.8.15.1
— time.cal.ym.last 27.8.15.2
— time.zone.db.access 27.11.2.3
— time.zone.db.remote 27.11.2.4
— time.zone.zonedtime.ctor 27.11.7.2
— time.zone.exception.nonexist 27.11.3.1
— time.zone.exception.ambig 27.11.3.2
— time.parse 27.13

Changes from R0:
— Rebase against N4830.
— incorporate June telecom review suggestions, including:
— Reworked some of the "Expects" elements to use "is/was" instead of "shall".
— time.traits.duration.values - reverted changes, left 'remarks' and 'shalls' in place para3, para5, para7
— Correct macro for Editors notes about ordering and other issues so it doesn’t look like an add.
— 27.5.1 p5: markup problem; 'duration_cast...' needs to be black.
— time.duration.nonmember para8 and para12 add 'is true' to is_convertable.
— time.clock.system.nonmembers para 1 - was an 'if' so logic is backwards.
— many Daniel K update suggestions for is_convertible_v<Rep2, common_type_t<Rep1, Rep2>.

Thanks to Dan Sunderland and Marshall Clow for encouragement, guidance, and help with the tools. Also thanks to Daniel Krügler for detailed review and expert wording help.
27 Time library

27.1 General

This Clause describes the chrono library (27.2) and various C functions (27.14) that provide generally useful
time utilities, as summarized in Table 85.

Table 85: Time library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>27.3</td>
<td>Cpp17Clock requirements</td>
</tr>
<tr>
<td>27.4</td>
<td>Time-related traits</td>
</tr>
<tr>
<td>27.5</td>
<td>Class template duration</td>
</tr>
<tr>
<td>27.6</td>
<td>Class template time_point</td>
</tr>
<tr>
<td>27.7</td>
<td>Clocks</td>
</tr>
<tr>
<td>27.8</td>
<td>Civil calendar</td>
</tr>
<tr>
<td>27.9</td>
<td>Class template hh_mm_ss</td>
</tr>
<tr>
<td>27.10</td>
<td>12/24 hour functions</td>
</tr>
<tr>
<td>27.11</td>
<td>Time zones</td>
</tr>
<tr>
<td>27.12</td>
<td>Formatting</td>
</tr>
<tr>
<td>27.13</td>
<td>Parsing</td>
</tr>
<tr>
<td>27.14</td>
<td>C library time utilities</td>
</tr>
</tbody>
</table>

Let STATICALLY-WIDEN<charT>("...") be "...", if charT is char and L"...", if charT is wchar_t.

27.2 Header <chrono> synopsis

namespace std {
    namespace chrono {
        // 27.5, class template duration
        template<class Rep, class Period = ratio<1>> class duration;

        // 27.6, class template time_point
        template<class Clock, class Duration = typename Clock::duration> class time_point;
    }

    // 27.4.3, common_type specializations
    template<class Rep1, class Period1, class Rep2, class Period2>
    struct common_type<chrono::duration<Rep1, Period1>,
                       chrono::duration<Rep2, Period2>>;

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

    namespace chrono {
        // 27.4, customization traits
        template<class Rep> struct treat_as_floating_point;
        template<class Rep> struct duration_values;
        template<class Rep>
            inline constexpr bool treat_as_floating_point_v = treat_as_floating_point<Rep>::value;

        template<class T> struct is_clock;
        template<class T> inline constexpr bool is_clock_v = is_clock<T>::value;

        // 27.5.5, duration arithmetic
        template<class Rep1, class Period1, class Rep2, class Period2>
        constexpr common_type_t<chrono::duration<Rep1, Period1>,
                             chrono::duration<Rep2, Period2>>
            operator+(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

§ 27.2
template<class Rep1, class Period1, class Rep2, class Period2>
constexpr common_type_t<duration<Rep1, Period1>, duration<Rep2, Period2>>
operator-(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

template<class Rep1, class Period, class Rep2>
constexpr duration<common_type_t<Rep1, Rep2>, Period>
operator*(const duration<Rep1, Period>& d, const Rep2& s);

template<class Rep1, class Period, class Rep2>
constexpr duration<common_type_t<Rep1, Rep2>, Period>
operator*(const Rep1& s, const duration<Rep2, Period>& d);

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

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

// 27.5.6, duration comparisons
template<class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator==(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

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

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

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

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

// 27.5.6, duration comparisons
requires see below
constexpr auto operator<=>(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

// 27.5.7, duration_cast
template<class ToDuration, class Rep, class Period>
constexpr ToDuration duration_cast(const duration<Rep, Period>& d);

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

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

template<class ToDuration, class Rep, class Period, class Alloc = allocator<charT>>
constexpr ToDuration round(const duration<Rep, Period>& d);

// 27.5.10, duration I/O
template<class charT, class traits, class Rep, class Period>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const duration<Rep, Period>& d);

template<class charT, class traits, class Rep, class Period, class Alloc = allocator<charT>>
basic_istream<charT, traits>&
from_stream(basic_istream<charT, traits>& is, const charT* fmt, duration<Rep, Period>& d, basic_string<charT, traits, Alloc>* abbrev = nullptr, minutes* offset = nullptr);
// convenience typedefs
using nanoseconds = duration<signed integer type of at least 64 bits, nano>;
using microseconds = duration<signed integer type of at least 55 bits, micro>;
using milliseconds = duration<signed integer type of at least 45 bits, milli>;
using seconds = duration<signed integer type of at least 35 bits>;
using minutes = duration<signed integer type of at least 29 bits, ratio< 60>>;
using hours = duration<signed integer type of at least 23 bits, ratio<3600>>;
using days = duration<signed integer type of at least 25 bits, ratio_multiply<ratio<24>, hours::period>>;
using weeks = duration<signed integer type of at least 22 bits, ratio_multiply<ratio<7>, days::period>>;
using years = duration<signed integer type of at least 17 bits, ratio_multiply<ratio<146097, 400>, days::period>>;
using months = duration<signed integer type of at least 20 bits, ratio_divide<years::period, ratio<12>>>;

// 27.6.5, time_point arithmetic
template<class Clock, class Duration1, class Rep2, class Period2>
constexpr time_point<Clock, common_type_t<Duration1, duration<Rep2, Period2>>>
operator+(const time_point<Clock, Duration1>& lhs, const duration<Rep2, Period2>& rhs);

template<class Rep1, class Period1, class Clock, class Duration2>
constexpr time_point<Clock, common_type_t<duration<Rep1, Period1>, Duration2>>
operator+(const duration<Rep1, Period1>& lhs, const time_point<Clock, Duration2>& rhs);

template<class Clock, class Duration1, class Rep2, class Period2>
constexpr time_point<Clock, common_type_t<Duration1, duration<Rep2, Period2>>>
operator-(const time_point<Clock, Duration1>& lhs, const duration<Rep2, Period2>& rhs);

template<class Clock, class Duration1, class Duration2>
constexpr common_type_t<Duration1, Duration2>
operator-(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);

// 27.6.6, time_point comparisons
template<class Clock, class Duration1, class Duration2>
constexpr bool operator==(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);

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

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

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

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

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

// 27.6.7, time_point_cast
template<class ToDuration, class Clock, class Duration>
constexpr time_point<Clock, ToDuration>
time_point_cast(const time_point<Clock, Duration>& t);

template<class ToDuration, class Clock, class Duration>
constexpr time_point<Clock, ToDuration> floor(const time_point<Clock, Duration>& tp);

template<class ToDuration, class Clock, class Duration>
constexpr time_point<Clock, ToDuration> ceil(const time_point<Clock, Duration>& tp);

template<class ToDuration, class Clock, class Duration>
constexpr time_point<Clock, ToDuration> round(const time_point<Clock, Duration>& tp);
// 27.5.9, specialized algorithms
template<class Rep, class Period>
    constexpr duration<Rep, Period> abs(duration<Rep, Period> d);

// 27.7.1, class system_clock
class system_clock;

    template<class Duration>
        using sys_time = time_point<system_clock, Duration>;
    using sys_seconds = sys_time<seconds>;
    using sys_days = sys_time<days>;

    template<class charT, class traits>
        basic_ostream<charT, traits>&
            operator<<(basic_ostream<charT, traits>& os, const sys_time<Duration>& tp);

// 27.7.2, class utc_clock
class utc_clock;

    using utc_time = time_point<utc_clock, Duration>;
    using utc_seconds = utc_time<seconds>;

    template<class charT, class traits>
        basic_ostream<charT, traits>&
            operator<<(basic_ostream<charT, traits>& os, const utc_time<Duration>& t);

// 27.7.3, class tai_clock
class tai_clock;

    using tai_time = time_point<tai_clock, Duration>;
    using tai_seconds = tai_time<seconds>;

    template<class charT, class traits>
        basic_ostream<charT, traits>&
            operator<<(basic_ostream<charT, traits>& os, const tai_time<Duration>& t);

struct leap_second_info;

    template<class Duration>
        leap_second_info get_leap_second_info(const utc_time<Duration>& ut);

// 27.2
minutes* offset = nullptr);

// 27.7.4, class gps_clock
class gps_clock;

template<class Duration>
    using gps_time = time_point<gps_clock, Duration>;
using gps_seconds = gps_time<seconds>;

template<class charT, class traits, class Duration>
    basic_ostream<charT, traits>&
        operator<<(basic_ostream<charT, traits>& os, const gps_time<Duration>& t);

template<class charT, class traits, class Duration, class Alloc = allocator<charT>>
    basic_istream<charT, traits>&
        from_stream(basic_istream<charT, traits>& is, const charT* fmt,
            gps_time<Duration>& tp, basic_string<charT, traits, Alloc>* abbrev = nullptr,
            minutes* offset = nullptr);

// 27.7.5, type file_clock
using file_clock = see below;

template<class Duration>
    using file_time = time_point<file_clock, Duration>;

template<class charT, class traits, class Duration>
    basic_ostream<charT, traits>&
        operator<<(basic_ostream<charT, traits>& os, const file_time<Duration>& tp);

template<class charT, class traits, class Duration, class Alloc = allocator<charT>>
    basic_istream<charT, traits>&
        from_stream(basic_istream<charT, traits>& is, const charT* fmt,
            file_time<Duration>& tp, basic_string<charT, traits, Alloc>* abbrev = nullptr,
            minutes* offset = nullptr);

// 27.7.6, class steady_clock
class steady_clock;

// 27.7.7, class high_resolution_clock
class high_resolution_clock;

// 27.7.8, local time
struct local_t {};

template<class Duration>
    using local_time = time_point<local_t, Duration>;
using local_seconds = local_time<seconds>;
using local_days = local_time<days>;

template<class charT, class traits, class Duration>
    basic_ostream<charT, traits>&
        operator<<(basic_ostream<charT, traits>& os, const local_time<Duration>& tp);

template<class charT, class traits, class Duration, class Alloc = allocator<charT>>
    basic_istream<charT, traits>&
        from_stream(basic_istream<charT, traits>& is, const charT* fmt,
            local_time<Duration>& tp, basic_string<charT, traits, Alloc>* abbrev = nullptr,
            minutes* offset = nullptr);

// 27.7.9, time_point conversions

template<class DestClock, class SourceClock>
    struct clock_time_conversion;

template<class DestClock, class SourceClock, class Duration>
    auto clock_cast(const time_point<SourceClock, Duration>& t);
// 27.8.2, class last_spec
struct last_spec;

// 27.8.3, class day
class day;

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

cconstexpr 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>
  basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const day& d);

template<class charT, class traits, class Alloc = allocator<charT>>
  basic_istream<charT, traits>&
    from_stream(basic_istream<charT, traits>& is, const charT* fmt,
                day& d, basic_string<charT, traits, Alloc>* abbrev = nullptr,
                minutes* offset = nullptr);

// 27.8.4, class month
class month;

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

cconstexpr 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>
  basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const month& m);

template<class charT, class traits, class Alloc = allocator<charT>>
  basic_istream<charT, traits>&
    from_stream(basic_istream<charT, traits>& is, const charT* fmt,
                month& m, basic_string<charT, traits, Alloc>* abbrev = nullptr,
                minutes* offset = nullptr);

// 27.8.5, class year
class year;

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

cconstexpr 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>
  basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const year& y);

template<class charT, class traits, class Alloc = allocator<charT>>
  basic_istream<charT, traits>&
    from_stream(basic_istream<charT, traits>& is, const charT* fmt,
                year& y, basic_string<charT, traits, Alloc>* abbrev = nullptr,
                minutes* offset = nullptr);
// 27.8.6, class weekday
class weekday;

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

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const weekday& wd);

template<class charT, class traits, class Alloc = allocator<charT>>
basic_istream<charT, traits>&
from_stream(basic_istream<charT, traits>& is, const charT* fmt,
weekday& wd, basic_string<charT, traits, Alloc>* abbrev = nullptr,
minutes* offset = nullptr);

// 27.8.7, class weekday_indexed
class weekday_indexed;

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

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const weekday_indexed& wdi);

// 27.8.8, class weekday_last
class weekday_last;

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


template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const weekday_last& wdl);

// 27.8.9, class month_day
class month_day;

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


template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const month_day& md);


template<class charT, class traits, class Alloc = allocator<charT>>
basic_istream<charT, traits>&
from_stream(basic_istream<charT, traits>& is, const charT* fmt,
month_day& md, basic_string<charT, traits, Alloc>* abbrev = nullptr,
minutes* offset = nullptr);

// 27.8.10, class month_day_last
class month_day_last;

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


template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const month_day_last& mdl);
// 27.8.11, class month_weekday
class month_weekday;

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

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const month_weekday& mwd);

// 27.8.12, class month_weekday_last
class month_weekday_last;

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

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const month_weekday_last& mwdl);

// 27.8.13, class year_month
class year_month;

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

constexpr strong_ordering 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;

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

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const year_month& ym);

template<class charT, class traits, class Alloc = allocator<charT>>
basic_istream<charT, traits>&
from_stream(basic_istream<charT, traits>& is, const charT* fmt,
year_month& ym, basic_string<charT, traits, Alloc>* abbrev = nullptr,
minutes* offset = nullptr);

// 27.8.14, class year_month_day
class year_month_day;

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

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

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const year_month_day& ymd);

template<class charT, class traits, class Alloc = allocator<charT>>
basic_istream<charT, traits>&
from_stream(basic_istream<charT, traits>& is, const charT* fmt,
year_month_day& ymd,
// 27.8.15, class year_month_dayLast
class year_month_day_last;

constexpr bool operator==(const year_month_day_last& x,
const year_month_day_last& y) noexcept;
constexpr strong_ordering 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>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const year_month_day_last& ymdl);

// 27.8.16, class year_month_weekday
class year_month_weekday;

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>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const year_month_weekday& ymwd);

// 27.8.17, class year_month_weekdayLast
class year_month_weekday_last;

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& ymwl, const months& dm) noexcept;
constexpr year_month_weekday_last
operator+(const months& dm, const year_month_weekday_last& ymwl) noexcept;
constexpr year_month_weekday_last
operator+(const year_month_weekday_last& ymwl, const years& dy) noexcept;

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const year_month_weekday_last& ymwl);
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>
  basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const year_month_weekday_last& ymwdl);

// 27.8.18, civil calendar 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 day& d, const year_month& ym) noexcept;
constexpr year_month_day
  operator/(const day& d, int y) 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;

// 27.9, class template hh_mm_ss

template<class Duration> class hh_mm_ss;

template<class charT, class traits, class Duration>
    basic_ostream<charT, traits>&
        operator<<(basic_ostream<charT, traits>& os, const hh_mm_ss<Duration>& hms);

// 27.10, 12/24 hour functions
constexpr bool is_am(const hours& h) noexcept;
constexpr bool is_pm(const hours& h) noexcept;
constexpr hours make12(const hours& h) noexcept;
constexpr hours make24(const hours& h, bool is_pm) noexcept;

// 27.11.2, time zone database

struct tzdb;
class tzdb_list;

// 27.11.2.3, time zone database access
const tzdb& get_tzdb();
tzdb_list& get_tzdb_list();
const time_zone* locate_zone(string_view tz_name);
const time_zone* current_zone();

// 27.11.2.4, remote time zone database support
const tzdb& reload_tzdb();
string remote_version();

// 27.11.3, exception classes
class nonexistent_local_time;
class ambiguous_local_time;

// 27.11.4, information classes
struct sys_info;
template<class charT, class traits>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const sys_info& si);

struct local_info;
template<class charT, class traits>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const local_info& li);

// 27.11.5, class time_zone
enum class choose {earliest, latest};
class time_zone;

bool operator==(const time_zone& x, const time_zone& y) noexcept;
strong_ordering operator<=>(const time_zone& x, const time_zone& y) noexcept;

// 27.11.6, class template zoned_traits
template<class T> struct zoned_traits;

// 27.11.7, class template zoned_time
template<class Duration, class TimeZonePtr = const time_zone*> class zoned_time;
using zoned_seconds = zoned_time<seconds>;

template<class Duration1, class Duration2, class TimeZonePtr>
    bool operator==(const zoned_time<Duration1, TimeZonePtr>& x,
                   const zoned_time<Duration2, TimeZonePtr>& y);

template<class charT, class traits, class Duration, class TimeZonePtr>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os,
               const zoned_time<Duration, TimeZonePtr>& t);

// 27.11.8, leap second support
class leap;

bool operator==(const leap& x, const leap& y);
strong_ordering operator<=>(const leap& x, const leap& y);

template<class Duration>
    bool operator==(const leap& x, const sys_time<Duration>& y);
template<class Duration>
    bool operator<( const leap& x, const sys_time<Duration>& y);
template<class Duration>
    bool operator< ( const sys_time<Duration>& x, const leap& y);
template<class Duration>
    bool operator> ( const leap& x, const sys_time<Duration>& y);
template<class Duration>
    bool operator> ( const sys_time<Duration>& x, const leap& y);
template<class Duration>
    bool operator<= (const leap& x, const sys_time<Duration>& y);
template<class Duration>
    bool operator>= (const sys_time<Duration>& x, const leap& y);
template<three_way_comparable_with<sys_seconds> Duration>
    auto operator<=>(const leap& x, const sys_time<Duration>& y);

// 27.11.9, class link
class link;
bool operator==(const link& x, const link& y);
strong_ordering operator<=>(const link& x, const link& y);

// 27.12. formatting
template<class Duration>
struct local_time_format_{Duration};  // exposition only

// exposition only
template<class Duration>
local_time_format_{Duration}(
    local_time_{Duration} time, const string* abbrev = nullptr,
    const seconds* offset_sec = nullptr);

}  // namespace chrono

}  // namespace chrono

namespace model {

}  // namespace model

§ 27.2
template<
class charT, class traits, class Alloc, class Parsable>
unspecified
parse(const basic_string<charT, traits, Alloc>& format, Parsable& tp,
basic_string<charT, traits, Alloc>& abbrev, minutes& offset);

// calendrical constants
inline constexpr last_spec last{};
inline constexpr weekday Sunday{0};
inline constexpr weekday Monday{1};
inline constexpr weekday Tuesday{2};
inline constexpr weekday Wednesday{3};
inline constexpr weekday Thursday{4};
inline constexpr weekday Friday{5};
inline constexpr weekday Saturday{6};
inline constexpr month January{1};
inline constexpr month February{2};
inline constexpr month March{3};
inline constexpr month April{4};
inline constexpr month May{5};
inline constexpr month June{6};
inline constexpr month July{7};
inline constexpr month August{8};
inline constexpr month September{9};
inline constexpr month October{10};
inline constexpr month November{11};
inline constexpr month December{12};

inline namespace literals {
inline namespace chrono_literals {
// 27.5.8, suffixes for duration literals
constexpr chrono::hours operator"h(unsigned long long);
constexpr chrono::duration<unspecified, ratio<3600, 1>> operator"h(long double);
constexpr chrono::minutes operator"min(unsigned long long);
constexpr chrono::duration<unspecified, ratio<60, 1>> operator"min(long double);
constexpr chrono::seconds operator"s(unsigned long long);
constexpr chrono::duration<unspecified> operator"s(long double);
constexpr chrono::milliseconds operator"ms(unsigned long long);
constexpr chrono::duration<unspecified, milli> operator"ms(long double);
constexpr chrono::microseconds operator"us(unsigned long long);
constexpr chrono::duration<unspecified, micro> operator"us(long double);
constexpr chrono::nanoseconds operator"ns(unsigned long long);
constexpr chrono::duration<unspecified, nano> operator"ns(long double);

// 27.8.3.3, non-member functions
constexpr chrono::day operator"d(unsigned long long d) noexcept;
// 27.8.5.3, non-member functions
constexpr chrono::year operator"y(unsigned long long y) noexcept;
}
}
namespace chrono {
using namespace literals::chrono_literals;
}
}
27.3 Cpp17Clock requirements

A clock is a bundle consisting of a duration, a time_point, and a function now() to get the current time_point. The origin of the clock’s time_point is referred to as the clock’s epoch. A clock shall meet the requirements in Table 86.

In Table 86 C1 and C2 denote clock types. t1 and t2 are values returned by C1::now() where the call returning t1 happens before (?) the call returning t2 and both of these calls occur before C1::time_point::max().

[Note: This means C1 did not wrap around between t1 and t2. — end note]

Table 86: Cpp17Clock requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1::rep</td>
<td>An arithmetic type or a class emulating an arithmetic type</td>
<td>The representation type of C1::duration.</td>
</tr>
<tr>
<td>C1::period</td>
<td>a specialization of ratio</td>
<td>The tick period of the clock in seconds.</td>
</tr>
<tr>
<td>C1::duration</td>
<td>chrono::duration&lt; C1::rep, C1::period &gt;</td>
<td>The duration type of the clock.</td>
</tr>
<tr>
<td>C1::time_point</td>
<td>chrono::time_point&lt; C1 &gt; or chrono::time_point&lt; C2, C1::duration &gt;</td>
<td>The time_point type of the clock. C1 and C2 shall refer to the same epoch.</td>
</tr>
<tr>
<td>C1::is_steady</td>
<td>const bool</td>
<td>true if t1 &lt;= t2 is always true and the time between clock ticks is constant, otherwise false.</td>
</tr>
<tr>
<td>C1::now()</td>
<td>C1::time_point</td>
<td>Returns a time_point object representing the current point in time.</td>
</tr>
</tbody>
</table>

3 [Note: The relative difference in durations between those reported by a given clock and the SI definition is a measure of the quality of implementation. — end note]

A type TC meets the Cpp17TrivialClock requirements if:

(4.1) — TC meets the Cpp17Clock requirements (27.3),

(4.2) — the types TC::rep, TC::duration, and TC::time_point meet the Cpp17EqualityComparable (Table ??) and Cpp17LessThanComparable (Table ??) requirements and the requirements of numeric types (??).

[Note: This means, in particular, that operations on these types will not throw exceptions. — end note]

(4.3) — values of the types TC::rep, TC::duration, and TC::time_point are swappable (??),

(4.4) — the function TC::now() does not throw exceptions, and

(4.5) — the type TC::time_point::clock meets the Cpp17TrivialClock requirements, recursively.

27.4 Time-related traits

27.4.1 treat_as_floating_point

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

The duration template uses the treat_as_floating_point trait to help determine if a duration object can be converted to another duration with a different tick period. If treat_as_floating_point_v<Rep> is true, then implicit conversions are allowed among durations. Otherwise, the implicit convertibility depends on the tick periods of the durations. [Note: The intention of this trait is to indicate whether a given class behaves like a floating-point type, and thus allows division of one value by another with acceptable loss of precision. If treat_as_floating_point_v<Rep> is false, Rep will be treated as if it behaved like an integral type for the purpose of these conversions. — end note]
27.4.2 duration_values

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

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

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

1. **Returns:** `Rep(0)`. [Note: `Rep(0)` is specified instead of `Rep()` because `Rep()` may have some other meaning, such as an uninitialized value. — end note]

2. **Remarks:** The value returned shall be the additive identity.

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

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

4. **Remarks:** The value returned shall compare less than or equal to `zero()`.

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

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

6. **Remarks:** The value returned shall compare greater than `zero()`.

### 27.4.3 Specializations of common_type

```
template<class Rep1, class Period1, class Rep2, class Period2>
struct common_type<chrono::duration<Rep1, Period1>, chrono::duration<Rep2, Period2>> {
  using type = chrono::duration<common_type_t<Rep1, Rep2>, see below>;
};
```

1. **The period of the duration** indicated by this specialization of `common_type` shall be the greatest common divisor of `Period1` and `Period2`. [Note: This can be computed by forming a ratio of the greatest common divisor of `Period1::num` and `Period2::num` and the least common multiple of `Period1::den` and `Period2::den`. — end note]

2. **The typedef name** `type` is a synonym for the `duration` with the largest tick period possible where both `duration` arguments will convert to it without requiring a division operation. The representation of this type is intended to be able to hold any value resulting from this conversion with no truncation error, although floating-point durations may have round-off errors. — end note]

```
template<class Clock, class Duration1, class Duration2>
struct common_type<chrono::time_point<Clock, Duration1>, chrono::time_point<Clock, Duration2>> {
  using type = chrono::time_point<Clock, common_type_t<Duration1, Duration2>>;
};
```

3. **The common type of two `time_point` types is a `time_point` with the same clock as the two types and the common type of their two durations.**

### 27.4.4 Class template is_clock

```
template<class T> struct is_clock;
```

1. **is_clock** is a `Cpp17UnaryTypeTrait` with a base characteristic of `true_type` if `T` meets the `Cpp17Clock` requirements (27.3), otherwise `false_type`. For the purposes of the specification of this trait, the extent to which an implementation determines that a type cannot meet the `Cpp17Clock` requirements is unspecified, except that as a minimum a type `T` shall not qualify as a `Cpp17Clock` unless it meets all of the following conditions:
— the qualified-ids T::rep, T::period, T::duration, and T::time_point are valid and each denotes a type (??),
— the expression T::is_steady is well-formed when treated as an unevaluated operand,
— the expression T::now() is well-formed when treated as an unevaluated operand.

The behavior of a program that adds specializations for is_clock is undefined.

27.5 Class template duration

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

```cpp
namespace std::chrono {
    template<class Rep, class Period = ratio<1>>
    class duration {
        public:
            using rep = Rep;
            using period = typename Period::type;

        private:
            rep rep_;  // exposition only

        public:
            // 27.5.1, construct/copy/destroy
            constexpr duration() = default;
            template<class Rep2>
                constexpr explicit duration(const Rep2& r);
            template<class Rep2, class Period2>
                constexpr duration(const duration<Rep2, Period2>& d);
            ~duration() = default;
            duration& operator=(const duration&);  // ill-formed?
            duration(const duration&) = default;
            duration& operator=(const duration& d);
        }
    }
}
```

Rep shall be an arithmetic type or a class emulating an arithmetic type. If duration is instantiated with a duration type as the argument for the template parameter Rep, the program is ill-formed.

If Period is not a specialization of ratio, the program is ill-formed. If Period::num is not positive, the program is ill-formed.
Members of `duration` shall not throw exceptions other than those thrown by the indicated operations on their representations.

The defaulted copy constructor of `duration` shall be a constexpr function if and only if the required initialization of the member `rep_` for copy and move, respectively, would satisfy the requirements for a constexpr function.

[Example:
```
duration<long, ratio<60>> d0;  // holds a count of minutes using a long
duration<long long, milli> d1;  // holds a count of milliseconds using a long long
duration<double, ratio<1, 30>> d2;  // holds a count with a tick period of \frac{1}{30} of a second  
                                      // (30 Hz) using a double
```
— end example]

### 27.5.1 Constructors

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

Remarks: Constraints: This constructor shall not participate in overload resolution unless `Rep2` is implicitly convertible to `rep` is `convertible_v<Rep2, rep>` is true and

(1.1) — `treat_as_floating_point_v<rep>` is true or

(1.2) — `treat_as_floating_point_v<Rep2>` is false.

[Example:
```
duration<int, milli> d(3);  // OK
duration<int, milli> d(3.5); // error
```
— end example]

Effects: Constructs an object of type `duration`.

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

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

Remarks: Constraints: This constructor shall not participate in overload resolution unless no overflow is induced in the conversion and `treat_as_floating_point_v<rep>` is true or both `ratio_divide<Period2, period>::den` is 1 and `treat_as_floating_point_v<Rep2>` is false. [Note: This requirement prevents implicit truncation error when converting between integral-based `duration` types. Such a construction could easily lead to confusion about the value of the `duration`. — end note]

[Example:
```
duration<int, milli> ms(3);  // OK
duration<int, micro> us = ms; // OK
duration<int, milli> ms2 = us; // error
```
— end example]

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

### 27.5.2 Observer

**Template:**
```
constexpr rep count() const;
```

Returns: `rep_`

### 27.5.3 Arithmetic

**Template:**
```
constexpr common_type_t<duration> operator()(const);
```

Returns: `common_type_t<duration>(*this)`.

```
constexpr common_type_t<duration> operator-() const;
```

Returns: `common_type_t<duration>(-rep_)`.

§ 27.5.3
constexpr duration& operator++();

    Effects: As if by ++rep_.

    Returns: *this.

cconstexpr duration operator++(int);

    Returns: duration(rep_++).

cconstexpr duration& operator--();

    Effects: As if by --rep_.

    Returns: *this.

cconstexpr duration operator--(int);

    Returns: duration(rep_--).

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

    Effects: As if by: rep_ += d.count();

    Returns: *this.

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

    Effects: As if by: rep_ -= d.count();

    Returns: *this.

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

    Effects: As if by: rep_ *= rhs;

    Returns: *this.

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

    Effects: As if by: rep_ /= rhs;

    Returns: *this.

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

    Effects: As if by: rep_ %= rhs;

    Returns: *this.

cconstexpr duration& operator%=(const duration& rhs);

    Effects: As if by: rep_ %= rhs.count();

    Returns: *this.

27.5.4 Special values

static constexpr duration zero() noexcept;

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

static constexpr duration min() noexcept;

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

static constexpr duration max() noexcept;

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

27.5.5 Non-member arithmetic

In the function descriptions that follow, unless stated otherwise, let CD represent the return type of the function.
template<class Rep1, class Period1, class Rep2, class Period2>
constexpr common_type_t<duration<Rep1, Period1>, duration<Rep2, Period2>>
operator+(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

Returns: CD(CD(lhs).count() + CD(rhs).count()).

2.5.6 Comparisons

In the function descriptions that follow, CT represents common_type_t<A, B>, where A and B are the types of the two arguments to the function.
template<class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator==(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
2
Returns: CT(lhs).count() == CT(rhs).count().

template<class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator<(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
3
Returns: CT(lhs).count() < CT(rhs).count().

template<class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator>(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
4
Returns: rhs < lhs.

template<class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator<=(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
5
Returns: !(rhs < lhs).

template<class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator>=(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
6
Returns: !(lhs < rhs).

template<class Rep1, class Period1, class Rep2, class Period2>
requires three_way_comparable<typename CT::rep>
constexpr auto operator<=>(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
7
Returns: CT(lhs).count() <=> CT(rhs).count().

27.5.7 duration_cast [time.duration.cast]

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

Remarks: Constraints: This function shall not participate in overload resolution unless ToDuration is a specialization of duration.

Returns: Let CF be ratio_divide<Period, typename ToDuration::period>, and CR be common_type<typename ToDuration::rep, Rep, intmax_t>::type.

\[(2.1)\] If CF::num == 1 and CF::den == 1, returns
ToDuration(static_cast<typename ToDuration::rep>(d.count()))

\[(2.2)\] Otherwise, if CF::num != 1 and CF::den == 1, returns
ToDuration(static_cast<typename ToDuration::rep>(
  static_cast<CR>(d.count()) * static_cast<CR>(CF::num)))

\[(2.3)\] Otherwise, if CF::num == 1 and CF::den != 1, returns
ToDuration(static_cast<typename ToDuration::rep>(
  static_cast<CR>(d.count()) / static_cast<CR>(CF::den)))

\[(2.4)\] Otherwise, returns
ToDuration(static_cast<typename ToDuration::rep>(
  static_cast<CR>(d.count()) * static_cast<CR>(CF::num) / static_cast<CR>(CF::den)))

Note: This function does not use any implicit conversions; all conversions are done with static_cast. It avoids multiplications and divisions when it is known at compile time that one or more arguments is 1. Intermediate computations are carried out in the widest representation and only converted to the destination representation at the final step. — end note]
template<class ToDuration, class Rep, class Period>
  constexpr ToDuration floor(const duration<Rep, Period>& d);

  Remarks: Constraints: This function shall not participate in overload resolution unless ToDuration is a specialization of duration.

  Returns: The greatest result t representable in ToDuration for which t <= d.

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

  Remarks: Constraints: This function shall not participate in overload resolution unless ToDuration is a specialization of duration.

  Returns: The least result t representable in ToDuration for which t >= d.

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

  Remarks: Constraints: This function shall not participate in overload resolution unless ToDuration is a specialization of duration, and treat_as_floating_point_v<typename ToDuration::rep> is false.

  Returns: The value of ToDuration that is closest to d. If there are two closest values, then return the value t for which t % 2 == 0.

27.5.8 Suffixes for duration literals

This subclause describes literal suffixes for constructing duration literals. The suffixes h, min, s, ms, us, ns denote duration values of the corresponding types hours, minutes, seconds, milliseconds, microseconds, and nanoseconds respectively if they are applied to integral literals.

If any of these suffixes are applied to a floating-point literal the result is a chrono::duration literal with an unspecified floating-point representation.

If any of these suffixes are applied to an integer literal and the resulting chrono::duration value cannot be represented in the result type because of overflow, the program is ill-formed.

[Example: The following code shows some duration literals.

```cpp
using namespace std::chrono_literals;
auto constexpr aday=24h;
auto constexpr lesson=45min;
auto constexpr halfanhour=0.5h;
```

—end example]

constexpr chrono::hours operator"h(unsigned long long hours);
constexpr chrono::duration<unspecified, ratio<3600, 1>> operator"h(long double hours);

Returns: A duration literal representing hours hours.

constexpr chrono::minutes operator"min(unsigned long long minutes);
constexpr chrono::duration<unspecified, ratio<60, 1>> operator"min(long double minutes);

Returns: A duration literal representing minutes minutes.

constexpr chrono::seconds operator"s(unsigned long long sec);
constexpr chrono::duration<unspecified> operator"s(long double sec);

Returns: A duration literal representing sec seconds.

[Note: The same suffix s is used for basic_string but there is no conflict, since duration suffixes apply to numbers and string literal suffixes apply to character array literals. — end note]

constexpr chrono::milliseconds operator"ms(unsigned long long msec);
constexpr chrono::duration<unspecified, milli> operator"ms(long double msec);

Returns: A duration literal representing msec milliseconds.

constexpr chrono::microseconds operator"us(unsigned long long usec);
constexpr chrono::duration<unspecified, micro> operator"us(long double usec);

Returns: A duration literal representing usec microseconds.
constexpr chrono::nanoseconds operator"ns(unsigned long long nsec);
constexpr chrono::duration<unspecified, nano> operator"ns(long double nsec);

11 Returns: A duration literal representing nsec nanoseconds.

27.5.9 Algorithms [time.duration.alg]

template<class Rep, class Period>
constexpr duration<Rep, Period> abs(duration<Rep, Period> d);

1 Remarks: Constraints: This function shall not participate in overload resolution unless numeric_limits<Rep>::is_signed is true.

2 Returns: If d >= d.zero(), return d, otherwise return -d.

27.5.10 I/O [time.duration.io]

template<class charT, class traits, class Rep, class Period>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const duration<Rep, Period>& d);

1 Requires: Mandates: Rep is an integral type whose integer conversion rank (??) is greater than or equal to that of short, or a floating-point type. charT is char or wchar_t.

2 Effects: Forms a basic_string<charT, traits> from d.count() using to_string if charT is char, or to_wstring if charT is wchar_t. Appends the units suffix described below to the basic_string. Inserts the resulting basic_string into os. [Note: This specification ensures that the result of this streaming operation will obey the width and alignment properties of the stream. — end note]

3 The units suffix depends on the type Period::type as follows:

   (3.1) — If Period::type is atto, the suffix is "as".
   (3.2) — Otherwise, if Period::type is femto, the suffix is "fs".
   (3.3) — Otherwise, if Period::type is pico, the suffix is "ps".
   (3.4) — Otherwise, if Period::type is nano, the suffix is "ns".
   (3.5) — Otherwise, if Period::type is micro, the suffix is "µs" ("\u00b5\u0073").
   (3.6) — Otherwise, if Period::type is milli, the suffix is "ms".
   (3.7) — Otherwise, if Period::type is centi, the suffix is "cs".
   (3.8) — Otherwise, if Period::type is deci, the suffix is "ds".
   (3.9) — Otherwise, if Period::type is ratio<1>, the suffix is "s".
   (3.10) — Otherwise, if Period::type is deca, the suffix is "das".
   (3.11) — Otherwise, if Period::type is hecto, the suffix is "hs".
   (3.12) — Otherwise, if Period::type is kilo, the suffix is "ks".
   (3.13) — Otherwise, if Period::type is mega, the suffix is "Ms".
   (3.14) — Otherwise, if Period::type is giga, the suffix is "Gs".
   (3.15) — Otherwise, if Period::type is tera, the suffix is "Ts".
   (3.16) — Otherwise, if Period::type is peta, the suffix is "Ps".
   (3.17) — Otherwise, if Period::type is exa, the suffix is "Es".
   (3.18) — Otherwise, if Period::type is ratio<60>, the suffix is "min".
   (3.19) — Otherwise, if Period::type is ratio<3600>, the suffix is "h".
   (3.20) — Otherwise, if Period::type is ratio<86400>, the suffix is "d".
   (3.21) — Otherwise, if Period::type::den == 1, the suffix is "[num]s".
   (3.22) — Otherwise, the suffix is "[num/den]s".

In the list above the use of num and den refer to the static data members of Period::type, which are converted to arrays of charT using a decimal conversion with no leading zeroes.
4 If `Period::type` is `micro`, but the character U+00B5 cannot be represented in the encoding used for `charT`, the unit suffix "us" is used instead of "µs".

5 Returns: os.

6 template<class charT, class traits, class Rep, class Period, class Alloc = allocator<charT>>
   basic_istream<charT, traits>&
   from_stream(basic_istream<charT, traits>& is, const charT* fmt,
               duration<Rep, Period>& d,
               basic_string<charT, traits, Alloc>* abbrev = nullptr,
               minutes* offset = nullptr);

6 Effects: Attempts to parse the input stream `is` into the duration `d` using the format flags given in the NTCTS `fmt` as specified in 27.13. If the parse parses everything specified by the parsing format flags without error, and yet none of the flags impacts a duration, `d` will be assigned a zero value. If `%Z` is used and successfully parsed, that value will be assigned to `*abbrev` if `abbrev` is non-null. If `%z` (or a modified variant) is used and successfully parsed, that value will be assigned to `*offset` if `offset` is non-null.

7 Returns: `is`.

27.6 Class template `time_point`

```
namespace std::chrono {
    template<class Clock, class Duration = typename Clock::duration>
    class time_point {
        public:
            using clock = Clock;
            using duration = Duration;
            using rep = typename duration::rep;
            using period = typename duration::period;

            private:
                duration d_; // exposition only

            public:

                // 27.6.1, construct
                constexpr time_point(); // has value epoch
                constexpr explicit time_point(const duration& d); // same as time_point() + d
                template<class Duration2>
                constexpr time_point(const time_point<Clock, Duration2>& t);

                // 27.6.2, observer
                constexpr duration time_since_epoch() const;

                // 27.6.3, arithmetic
                constexpr time_point& operator++();
                constexpr time_point operator++(int);
                constexpr time_point& operator--();
                constexpr time_point operator--(int);
                constexpr time_point& operator+=(const duration& d);
                constexpr time_point& operator-=(const duration& d);

                // 27.6.4, special values
                static constexpr time_point min() noexcept;
                static constexpr time_point max() noexcept;
            };
    }
```

1 `Clock` shall either meet the `Cpp17Clock` requirements (27.3) or be the type `local_t`.

2 If `Duration` is not an instance of `duration`, the program is ill-formed.
27.6.1 Constructors

constexpr time_point();

Effects: Constructs an object of type time_point, initializing d_ with duration::zero(). Such a time_point object represents the epoch.

constexpr explicit time_point(const duration& d);

Effects: Constructs an object of type time_point, initializing d_ with d. Such a time_point object represents the epoch + d.

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

Remarks: Constraints: This constructor shall not participate in overload resolution unless Duration2 is implicitly convertible to duration is_convertible_v<Duration2, duration> is true.

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

27.6.2 Observer

constexpr duration time_since_epoch() const;

Returns: d_.

27.6.3 Arithmetic

constexpr time_point operator++();

Effects: ++d_.

Returns: *this.

constexpr time_point operator++(int);

Returns: time_point(d_++).

constexpr time_point& operator--();

Effects: --d_.

Returns: *this.

constexpr time_point& operator--(int);

Returns: time_point(d_--).

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

Effects: As if by: d_ += d;

Returns: *this.

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

Effects: As if by: d_ -= d;

Returns: *this.

27.6.4 Special values

static constexpr time_point min() noexcept;

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

static constexpr time_point max() noexcept;

Returns: time_point(duration::max()).
27.6.5 Non-member arithmetic

\[
\text{template}\langle\text{class Clock, class Duration1, class Rep2, class Period2}\rangle
\text{constexpr time_point}\langle\text{Clock, common_type_t<Duration1, duration<Rep2, Period2>>}\rangle
\text{operator+}(\text{const time_point}\langle\text{Clock, Duration1}\rangle& \text{lhs}, \text{const duration<Rep2, Period2>}& \text{rhs});
\]

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

\[
\text{template}\langle\text{class Rep1, class Period1, class Clock, class Duration2}\rangle
\text{constexpr time_point}\langle\text{Clock, common_type_t<duration<Rep1, Period1>, Duration2>>}\rangle
\text{operator+}(\text{const duration<Rep1, Period1>}& \text{lhs}, \text{const time_point}\langle\text{Clock, Duration2}\rangle& \text{rhs});
\]

Returns: rhs + lhs.

\[
\text{template}\langle\text{class Clock, class Duration1, class Rep2, class Period2}\rangle
\text{constexpr time_point}\langle\text{Clock, common_type_t<duration<Rep2, Period2>, Duration1>>}\rangle
\text{operator-}(\text{const time_point}\langle\text{Clock, Duration1}\rangle& \text{lhs}, \text{const duration<Rep2, Period2>}& \text{rhs});
\]

Returns: CT(lhs.time_since_epoch() - rhs), where CT is the type of the return value.

\[
\text{template}\langle\text{class Clock, class Duration1, class Duration2}\rangle
\text{constexpr time_point}\langle\text{Clock, common_type_t<Duration1, Duration2>>}\rangle
\text{operator-}(\text{const time_point}\langle\text{Clock, Duration1}\rangle& \text{lhs}, \text{const time_point}\langle\text{Clock, Duration2}\rangle& \text{rhs});
\]

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

27.6.6 Comparisons

\[
\text{template}\langle\text{class Clock, class Duration1, class Duration2}\rangle
\text{constexpr bool operator==}(\text{const time_point}\langle\text{Clock, Duration1}\rangle& \text{lhs}, \text{const time_point}\langle\text{Clock, Duration2}\rangle& \text{rhs});
\]

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

\[
\text{template}\langle\text{class Clock, class Duration1, class Duration2}\rangle
\text{constexpr bool operator<(\text{const time_point}\langle\text{Clock, Duration1}\rangle& \text{lhs}, \text{const time_point}\langle\text{Clock, Duration2}\rangle& \text{rhs});
\]

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

\[
\text{template}\langle\text{class Clock, class Duration1, class Duration2}\rangle
\text{constexpr bool operator>(\text{const time_point}\langle\text{Clock, Duration1}\rangle& \text{lhs}, \text{const time_point}\langle\text{Clock, Duration2}\rangle& \text{rhs});
\]

Returns: rhs < lhs.

\[
\text{template}\langle\text{class Clock, class Duration1, class Duration2}\rangle
\text{constexpr bool operator<}(\text{const time_point}\langle\text{Clock, Duration1}\rangle& \text{lhs}, \text{const time_point}\langle\text{Clock, Duration2}\rangle& \text{rhs});
\]

Returns: !(rhs < lhs).

\[
\text{template}\langle\text{class Clock, class Duration1, class Duration2}\rangle
\text{constexpr bool operator>}(\text{const time_point}\langle\text{Clock, Duration1}\rangle& \text{lhs}, \text{const time_point}\langle\text{Clock, Duration2}\rangle& \text{rhs});
\]

Returns: !(lhs < rhs).

\[
\text{template}\langle\text{class Clock, class Duration1, class Duration2}\rangle
\text{three_way_comparable_with<Duration1> Duration2}
\text{constexpr auto operator<=>}(\text{const time_point}\langle\text{Clock, Duration1}\rangle& \text{lhs}, \text{const time_point}\langle\text{Clock, Duration2}\rangle& \text{rhs});
\]

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

27.6.7 time_point_cast

\[
\text{template}\langle\text{class ToDuration, class Clock, class Duration}\rangle
\text{constexpr time_point}\langle\text{Clock, ToDuration}\rangle\text{time_point_cast(\text{const time_point}\langle\text{Clock, Duration}\rangle& \text{t});}
\]

Remarks: Constraints: This function shall not participate in overload resolution unless ToDuration is a specialization of duration.
Returns:
  time_point<Clock, ToDuration>(duration_cast<ToDuration>(t.time_since_epoch()))

template<class ToDuration, class Clock, class Duration>
constexpr time_point<Clock, ToDuration> floor(const time_point<Clock, Duration>& tp);

Remarks: Constraints: This function shall not participate in overload resolution unless ToDuration is a specialization of duration.

Returns: time_point<Clock, ToDuration>(floor<ToDuration>(tp.time_since_epoch()))

template<class ToDuration, class Clock, class Duration>
constexpr time_point<Clock, ToDuration> ceil(const time_point<Clock, Duration>& tp);

Remarks: Constraints: This function shall not participate in overload resolution unless ToDuration is a specialization of duration.

Returns: time_point<Clock, ToDuration>(ceil<ToDuration>(tp.time_since_epoch()))

template<class ToDuration, class Clock, class Duration>
constexpr time_point<Clock, ToDuration> round(const time_point<Clock, Duration>& tp);

Remarks: Constraints: This function shall not participate in overload resolution unless ToDuration is a specialization of duration, and treat_as_floating_point_v<typename ToDuration::rep> is false.

Returns: time_point<Clock, ToDuration>(round<ToDuration>(tp.time_since_epoch()))

27.7 Clocks

The types defined in this subclause shall meet the Cpp17TrivialClock requirements (27.3) unless otherwise specified.

27.7.1 Class system_clock

namespace std::chrono {
  class system_clock {
  public:
    using rep = unspecified;
    using period = ratio<unspecified, unspecified>;
    using duration = chrono::duration<rep, period>;
    using time_point = chrono::time_point<system_clock>;
    static constexpr bool is_steady = unspecified;

    static time_point now() noexcept;
  // mapping to/from C type time_t
    static time_t to_time_t (const time_point& t) noexcept;
    static time_point from_time_t(time_t t) noexcept;
  
  }
}

Objects of type system_clock represent wall clock time from the system-wide realtime clock. Objects of type sys_time<Duration> measure time since (and before) 1970-01-01 00:00:00 UTC excluding leap seconds. This measure is commonly referred to as Unix time. This measure facilitates an efficient mapping between sys_time and calendar types (27.8). [Example:
sys_seconds{sys_days{1970y/January/1}}.time_since_epoch() is 0s.
sys_seconds{sys_days{2000y/January/1}}.time_since_epoch() is 946'684'800s, which is 10'957 * 86'400s.
—end example]

27.7.1.2 Members

using system_clock::rep = unspecified;

Requirements: Constraints: system_clock::duration::min() < system_clock::duration::zero() shall be true.
Note: This implies that \texttt{rep} is a signed type. — end note

static \texttt{time_t} \texttt{to_time_t} (const \texttt{time_point} & \texttt{t}) noexcept;

Returns: A \texttt{time_t} object that represents the same point in time as \texttt{t} when both values are restricted to the coarser of the precisions of \texttt{time_t} and \texttt{time_point}. It is implementation-defined whether values are rounded or truncated to the required precision.

static \texttt{time_point} \texttt{from_time_t} (\texttt{time_t} \texttt{t}) noexcept;

Returns: A \texttt{time_point} object that represents the same point in time as \texttt{t} when both values are restricted to the coarser of the precisions of \texttt{time_t} and \texttt{time_point}. It is implementation-defined whether values are rounded or truncated to the required precision.

27.7.1.3 Non-member functions

\begin{verbatim}
27.7.2 Class \texttt{utc_clock}

27.7.2.1 Overview

namespace std::chrono {
    class \texttt{utc_clock} {
        public:
            using \texttt{rep} = \texttt{a signed arithmetic type};
            using \texttt{period} = \texttt{ratio\langle unspecified, unspecified\rangle};
            using \texttt{duration} = \texttt{chrono\langle duration\langle rep, period\rangle\rangle};
            using \texttt{time_point} = \texttt{chrono\langle time_point\langle utc_clock\rangle\rangle};

\end{verbatim}

In contrast to \texttt{sys\_time}, which does not take leap seconds into account, \texttt{utc\_clock} and its associated \texttt{time\_point}, \texttt{utc\_time}, count time, including leap seconds, since 1970-01-01 00:00:00 UTC. [\textit{Example}: \texttt{clock\_cast<utc\_clock>(sys\_seconds(sys\_days{1970y/January/1})).time\_since\_epoch()} is 0s.\texttt{clock\_cast<utc\_clock>(sys\_seconds(sys\_days{2000y/January/1})).time\_since\_epoch()} is 946'684'822s, which is 10'957 * 86'400s + 22s. —end example]

\texttt{utc\_clock} is not a \textit{Cpp17TrivialClock} unless the implementation can guarantee that \texttt{utc\_clock::now()} does not propagate an exception. [\textit{Note}: noexcept(from\_sys(system\_clock::now())) is false. —end note]

### 27.7.2.2 Member functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{static time_point now();}</td>
<td>Returns: from_sys(system_clock::now()), or a more accurate value of utc_time.</td>
</tr>
</tbody>
</table>

#### 27.7.2.2.1 Time conversion

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{template&lt;class Duration&gt; static sys_time&lt;common_type_t&lt;Duration, seconds&gt;&gt; to_sys(const utc_time&lt;Duration&gt;&amp; t);}</td>
<td>Returns: A sys_time t, such that from_sys(t) == u if such a mapping exists. Otherwise u represents a time_point during a leap second insertion and the last representable value of sys_time prior to the insertion of the leap second is returned.</td>
</tr>
</tbody>
</table>

#### 27.7.2.2.2 Time since epoch

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
</table>
| \texttt{template<class Duration> static utc\_time<common\_type\_t<Duration, seconds>> from\_sys(const sys\_time<Duration>& t);} | Returns: A utc\_time u, such that u.time\_since\_epoch() - t.time\_since\_epoch() is equal to the number of leap seconds that were inserted between t and 1970-01-01. If t is exactly the date of leap second insertion, then the conversion counts that leap second as inserted. [\textit{Example}: \texttt{auto t = sys\_days(July/1/2015) - 2ns; auto u = utc\_clock::from\_sys(t);} \texttt{assert(u.time\_since\_epoch() - t.time\_since\_epoch() == 25s);} \texttt{t += 1ns;} \texttt{u = utc\_clock::from\_sys(t);} \texttt{assert(u.time\_since\_epoch() - t.time\_since\_epoch() == 25s);} \texttt{t += 1ns;} \texttt{u = utc\_clock::from\_sys(t);} \texttt{assert(u.time\_since\_epoch() - t.time\_since\_epoch() == 26s);} \texttt{t += 1ns;} \texttt{u = utc\_clock::from\_sys(t);} \texttt{assert(u.time\_since\_epoch() - t.time\_since\_epoch() == 26s);} —end example]

### 27.7.2.3 Non-member functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{template&lt;class charT, class traits, class Duration&gt; basic_ostream&lt;charT, traits&gt;&amp;}</td>
<td></td>
</tr>
</tbody>
</table>

§ 27.7.2.3
operator<<(basic_ostream<charT, traits>& os, const utc_time<Duration>& t);

Effects: Equivalent to:
return os << format("{:%F %T}", t);

[Example:
auto t = sys_days(July/1/2015) - 500ms;
auto u = clock_cast<utc_clock>(t);
for (auto i = 0; i < 8; ++i, u += 250ms)
cout << u << " UTC"

Produces this output:
2015-06-30 23:59:59.500 UTC
2015-06-30 23:59:59.750 UTC
2015-06-30 23:59:60.000 UTC
2015-06-30 23:59:60.250 UTC
2015-06-30 23:59:60.500 UTC
2015-06-30 23:59:60.750 UTC
2015-07-01 00:00:00.000 UTC
2015-07-01 00:00:00.250 UTC
—end example]

template<class charT, class traits, class Duration, class Alloc = allocator<charT>>
from_stream(basic_istream<charT, traits>& is, const charT* fmt, utc_time<Duration>& tp, basic_string<charT, traits, Alloc>* abbrev = nullptr, minutes* offset = nullptr);

Effects: Attempts to parse the input stream is into the utc_time tp using the format flags given in the NTCTS fmt as specified in 27.13. If the parse fails to decode a valid date, is.setstate(ios_base::failbit) shall be called and tp shall not be modified. If %Z is used and successfully parsed, that value will be assigned to *abbrev if abbrev is non-null. If %z (or a modified variant) is used and successfully parsed, that value will be assigned to *offset if offset is non-null. Additionally, the parsed offset will be subtracted from the successfully parsed timestamp prior to assigning that difference to tp.

Returns: is.

struct leap_second_info {
  bool  is_leap_second;
  seconds elapsed;
};

The type leap_second_info has data members and special members specified above. It has no base classes or members other than those specified.

template<class Duration>
leap_second_info get_leap_second_info(const utc_time<Duration>& ut);

Returns: A leap_second_info where is_leap_second is true if ut is during a leap second insertion, and otherwise false. elapsed is the number of leap seconds between 1970-01-01 and ut. If is_leap_second is true, the leap second referred to by ut is included in the count.

27.7.3 Class tai_clock
27.7.3.1 Overview
static time_point now();

template<class Duration>
static utc_time<common_type_t<Duration, seconds>>
to_utc(const tai_time<Duration> &t) noexcept;

template<class Duration>
static tai_time<common_type_t<Duration, seconds>>
from_utc(const utc_time<Duration> &t) noexcept;
}

1 The clock tai_clock measures seconds since 1958-01-01 00:00:00 and is offset 10s ahead of UTC at this date. That is, 1958-01-01 00:00:00 TAI is equivalent to 1957-12-31 23:59:50 UTC. Leap seconds are not inserted into TAI. Therefore every time a leap second is inserted into UTC, UTC falls another second behind TAI. For example by 2000-01-01 there had been 22 leap seconds inserted so 2000-01-01 00:00:00 UTC is equivalent to 2000-01-01 00:00:32 TAI (22s plus the initial 10s offset).

2 tai_clock is not a Cpp17TrivialClock unless the implementation can guarantee that tai_clock::now() does not propagate an exception. [Note: noexcept(from_utc(tai_clock::now())) is false. — end note]

27.7.3.2 Member functions

static time_point now();

Returns: from_utc(tai_clock::now()), or a more accurate value of tai_time.

template<class Duration>
static utc_time<common_type_t<Duration, seconds>>
to_utc(const tai_time<Duration> & t) noexcept;

Returns:
utc_time<common_type_t<Duration, seconds>>{t.time_since_epoch()} - 378691210s
[Note:
378691210s == sys_days{1970y/January/1} - sys_days{1958y/January/1} + 10s
— end note]

template<class Duration>
static tai_time<common_type_t<Duration, seconds>>
from_utc(const utc_time<Duration> & t) noexcept;

Returns:
tai_time<common_type_t<Duration, seconds>>{t.time_since_epoch()} + 378691210s
[Note:
378691210s == sys_days{1970y/January/1} - sys_days{1958y/January/1} + 10s
— end note]

27.7.3.3 Non-member functions

template<class charT, class traits, class Duration>
basic_ostream<charT, traits>&
operator<<<basic_ostream<charT, traits>& os, const tai_time<Duration>& t);

Effects: Equivalent to:

return os << format(STATICALLY-WIDEN<charT>("{%F %T}", t));

[Example:
auto st = sys_days{2000y/January/1};
auto tt = clock_cast<tai_clock>(st);
cout << format("{0:%F %T %Z} == {1:%F %T %Z}\n", st, tt);
Produces this output:
2000-01-01 00:00:00 UTC == 2000-01-01 00:00:32 TAI
— end example]
template<class charT, class traits, class Duration, class Alloc = allocator<charT>>
basic_istream<charT, traits>&
    from_stream(basic_istream<charT, traits>& is, const charT* fmt,
        tai_time<Duration>& tp, basic_string<charT, traits, Alloc>* abbrev = nullptr,
        minutes* offset = nullptr);

Effects: Attempts to parse the input stream is into the tai_time tp using the format flags given in the
NTCTS fmt as specified in 27.13. If the parse fails to decode a valid date, is.setstate(ios_base::failbit)
shall be called and tp shall not be modified. If %Z is used and successfully parsed, that value will be assigned to *abbrev if abbrev is non-null. If %z (or a modified variant) is used
and successfully parsed, that value will be assigned to *offset if offset is non-null. Additionally,
the parsed offset will be subtracted from the successfully parsed timestamp prior to assigning that
difference to tp.

Returns: is.

27.7.4 Class gps_clock
27.7.4.1 Overview

namespace std::chrono {
    class gps_clock {
        public:
            using rep = a signed arithmetic type;
            using period = ratio<unspecfied, unspecified>;
            using duration = std::chrono::duration<rep, period>;
            using time_point = std::chrono::time_point<gps_clock>;
            static constexpr bool is_steady = unspecified;

            static time_point now();

            template<class Duration>
            static utc_time<common_type_t<Duration, seconds>>
                to_utc(const gps_time<Duration>&) noexcept;

            template<class Duration>
            static gps_time<common_type_t<Duration, seconds>>
                from_utc(const utc_time<Duration>&) noexcept;
    };
}

1 The clock gps_clock measures seconds since the first Sunday of January, 1980 00:00:00 UTC. Leap seconds
are not inserted into GPS. Therefore every time a leap second is inserted into UTC, UTC falls another
second behind GPS. Aside from the offset from 1958y/January/1 to 1980y/January/Sunday[1], GPS is
behind TAI by 19s due to the 10s offset between 1958 and 1970 and the additional 9 leap seconds inserted
2 gps_clock is not a Cpp17TrivialClock unless the implementation can guarantee that gps_clock::now() does not propagate an exception. [Note: noexcept(from_utc(utc_clock::now())) is false. — end note]

27.7.4.2 Member functions

static time_point now();

Returns: from_utc(utc_clock::now()), or a more accurate value of gps_time.

template<class Duration>
static utc_time<common_type_t<Duration, seconds>>
    to_utc(const gps_time<Duration>& t) noexcept;

Returns:
    gps_time<common_type_t<Duration, seconds>>(t.time_since_epoch()) + 315964809s
[Note:
    315964809s == sys_days{1980y/January/Sunday[1]} - sys_days{1970y/January/1} + 9s
    — end note]
template<class Duration>
static gps_time<common_type_t<Duration, seconds>>
    from_utc(const utc_time<Duration>& t) noexcept;

Returns:  
gps_time<common_type_t<Duration, seconds>>\{t.time_since_epoch()\} - 315964809s

[Note:  
315964809s == sys_days\{1980y/January/Sunday\[1\]} - sys_days\{1970y/January/1\} + 9s
—end note]  

27.7.4.3 Non-member functions

template<class charT, class traits, class Duration>
basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const gps_time<Duration>& t);

Effects: Equivalent to:  
return os << format(STATICALLY-WIDEN\langle\text{charT}\rangle{\"{:%F %T}\"}, t);

[Example:  
auto st = sys_days\{2000y/January/1\};  
auto gt = clock_cast\langle\text{gps_clock}\rangle\{st\};  
cout << format\{"\{0:\%F \%T \%Z\} == \{1:\%F \%T \%Z\}\n\", st, gt\};  
Produces this output:  
2000-01-01 00:00:00 UTC == 2000-01-01 00:00:13 GPS
—end example]  

template<class charT, class traits, class Duration, class Alloc = allocator<charT>>
basic_istream<charT, traits>&
    from_stream(basic_istream<charT, traits>& is, const charT* fmt,
        gps_time<Duration>& tp, basic_string<charT, traits, Alloc>* abbrev = nullptr,
        minutes* offset = nullptr);

Effects: Attempts to parse the input stream is into the gps_time tp using the format flags given in
the NTCTS fmt as specified in 27.13. If the parse fails to decode a valid date, is.setstate(ios_base::failbit)
shall be called and tp shall not be modified. If %Z is used and successfully parsed,
that value will be assigned to *abbrev if abbrev is non-null. If %z (or a modified variant) is used
and successfully parsed, that value will be assigned to *offset if offset is non-null. Additionally,
the parsed offset will be subtracted from the successfully parsed timestamp prior to assigning that
difference to tp.

Returns: is.

27.7.5 Type file_clock

27.7.5.1 Overview

namespace std::chrono {
    using file_clock = see below;
}

file_clock is an alias for a type meeting the Cpp17TrivialClock requirements (27.3), and using a signed
arithmetic type for file_clock::rep. file_clock is used to create the time_point system used for file_type (??). Its epoch is unspecified, and noexcept(file_clock::now()) is true. [Note: The type
that file_clock denotes may be in a different namespace than std::chrono, such as std::filesystem.
—end note]

27.7.5.2 Member functions

The type denoted by file_clock provides precisely one of the following two sets of static member functions:

template<class Duration>
static sys_time<see below>
    to_sys(const file_time<Duration>&);
template<class Duration>
static file_time<see below>
  from_sys(const sys_time<Duration>&);

or:

template<class Duration>
static utc_time<see below>
  to_utc(const file_time<Duration>&);
template<class Duration>
static file_time<see below>
  from_utc(const utc_time<Duration>&);

These member functions shall provide time_point conversions consistent with those specified by utc_clock, tai_clock, and gps_clock. The Duration of the resultant time_point is computed from the Duration of the input time_point.

27.7.5.3 Non-member functions

template<class charT, class traits, class Duration>
basic_ostream<charT, traits>&
  operator<<(basic_ostream<charT, traits>& os, const file_time<Duration>& t);

Effects: Equivalent to:

  return os << format(STATICALLY-WIDEN<charT>("{:%F %T}"), t);

2

template<class charT, class traits, class Duration, class Alloc = allocator<charT>>
basic_istream<charT, traits>&
  from_stream(basic_istream<charT, traits>& is, const charT* fmt,
               file_time<Duration>& tp, basic_string<charT, traits, Alloc>* abbrev = nullptr,
               minutes* offset = nullptr);

Effects: Attempts to parse the input stream is into the file_time tp using the format flags given in the NTCTS fmt as specified in 27.13. If the parse fails to decode a valid date, is.setstate(ios_base::failbit) shall be called and tp shall not be modified. If %Z is used and successfully parsed, that value will be assigned to *abbrev if abbrev is non-null. If %z (or a modified variant) is used and successfully parsed, that value will be assigned to *offset if offset is non-null. Additionally, the parsed offset will be subtracted from the successfully parsed timestamp prior to assigning that difference to tp.

Returns: is.

3

27.7.6 Class steady_clock

namespace std::chrono {
  class steady_clock {
  public:
    using rep = unspecified;
    using period = ratio<unspecified, unspecified>;
    using duration = chrono::duration<rep, period>;
    using time_point = chrono::time_point<unspecified, duration>;
    static constexpr bool is_steady = true;
    static time_point now() noexcept;
  };
  }

1 Objects of class steady_clock represent clocks for which values of time_point never decrease as physical time advances and for which values of time_point advance at a steady rate relative to real time. That is, the clock may not be adjusted.

27.7.7 Class high_resolution_clock

namespace std::chrono {
  class high_resolution_clock {
  public:
    using rep = unspecified;
    using period = ratio<unspecified, unspecified>;
  };
  }

§ 27.7.7
using duration = chrono::duration<rep, period>;
using time_point = chrono::time_point<unspecified, duration>;
static constexpr bool is_steady = unspecified;

static time_point now() noexcept;
};

1 Objects of class `high_resolution_clock` represent clocks with the shortest tick period. `high_resolution_clock` may be a synonym for `system_clock` or `steady_clock`.

### 27.7.8 Local time

The family of time points denoted by `local_time<Duration>` are based on the pseudo clock `local_t`. `local_t` has no member `now()` and thus does not meet the clock requirements. Nevertheless `local_time<Duration>` serves the vital role of representing local time with respect to a not-yet-specified time zone. Aside from being able to get the current time, the complete `time_point` algebra is available for `local_time<Duration>` (just as for `sys_time<Duration>`).

```
template<class charT, class traits, class Duration>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const local_time<Duration>& lt);
```

2 Effects:

`os << sys_time<Duration>{lt.time_since_epoch()};`

3 Returns: `os`.

```
template<class charT, class traits, class Duration, class Alloc = allocator<charT>>
basic_istream<charT, traits>&
from_stream(basic_istream<charT, traits>& is, const charT* fmt,
local_time<Duration>& tp, basic_string<charT, traits, Alloc>* abbrev = nullptr,
minutes* offset = nullptr);
```

4 Effects: Attempts to parse the input stream `is` into the `local_time` `tp` using the format flags given in the NTCTS `fmt` as specified in 27.13. If the parse fails to decode a valid date, `is.setstate(ios_base::failbit)` shall be called and `tp` shall not be modified. If `%Z` is used and successfully parsed, that value will be assigned to `*abbrev` if `abbrev` is non-null. If `%z` (or a modified variant) is used and successfully parsed, that value will be assigned to `*offset` if `offset` is non-null.

5 Returns: `is`.

### 27.7.9 time_point conversions

#### 27.7.9.1 Class template `clock_time_conversion`

```
namespace std::chrono {
    template<class DestClock, class SourceClock>
    struct clock_time_conversion {};
}
```

1 `clock_time_conversion` serves as a trait which can be used to specify how to convert a source `time_point` of type `time_point<SourceClock, Duration>` to a destination `time_point` of type `time_point<DestClock, Duration>` via a specialization: `clock_time_conversion<DestClock, SourceClock>`. A specialization of `clock_time_conversion<DestClock, SourceClock>` shall provide a const-qualified `operator()` that takes a parameter of type `time_point<SourceClock, Duration>` and returns a `time_point<DestClock, OtherDuration>` representing an equivalent point in time. `OtherDuration` is a `chrono::duration` whose specialization is computed from the input `Duration` in a manner which can vary for each `clock_time_conversion` specialization. A program may specialize `clock_time_conversion` if at least one of the template parameters is a user-defined clock type.

2 Several specializations are provided by the implementation, as described in 27.7.9.2, 27.7.9.3, 27.7.9.4, and 27.7.9.5.
27.7.9.2 Identity conversions

```cpp
template<class Clock>
struct clock_time_conversion<Clock, Clock> {
    template<class Duration>
    time_point<Clock, Duration>
    operator()(const time_point<Clock, Duration>& t) const;
};

template<class Duration>
time_point<Clock, Duration>
operator()(const time_point<Clock, Duration>& t) const;

Returns: t.
```

```cpp
template<>
struct clock_time_conversion<system_clock, system_clock> {
    template<class Duration>
    sys_time<Duration>
    operator()(const sys_time<Duration>& t) const;
};

template<class Duration>
sys_time<Duration>
operator()(const sys_time<Duration>& t) const;

Returns: t.
```

```cpp
template<>
struct clock_time_conversion<utc_clock, utc_clock> {
    template<class Duration>
    utc_time<Duration>
    operator()(const utc_time<Duration>& t) const;
};

template<class Duration>
utc_time<Duration>
operator()(const utc_time<Duration>& t) const;

Returns: t.
```

27.7.9.3 Conversions between system_clock and utc_clock

```cpp
template<>
struct clock_time_conversion<utc_clock, system_clock> {
    template<class Duration>
    utc_time<common_type_t<Duration, seconds>>
    operator()(const sys_time<Duration>& t) const;
};

template<class Duration>
utc_time<common_type_t<Duration, seconds>>
operator()(const sys_time<Duration>& t) const;

Returns: utc_clock::from_sys(t).
```

```cpp
template<>
struct clock_time_conversion<system_clock, utc_clock> {
    template<class Duration>
    sys_time<common_type_t<Duration, seconds>>
    operator()(const utc_time<Duration>& t) const;
};

template<class Duration>
sys_time<common_type_t<Duration, seconds>>
operator()(const utc_time<Duration>& t) const;

Returns: utc_clock::to_sys(t).
```
27.7.9.4 Conversions between system_clock and other clocks

```cpp
template<class SourceClock>
struct clock_time_conversion<system_clock, SourceClock> {
    template<class Duration>
    auto operator()(const time_point<SourceClock, Duration>& t) const
    -> decltype(SourceClock::to_sys(t));
};

template<class Duration>
auto operator()(const time_point<SourceClock, Duration>& t) const
-> decltype(SourceClock::to_sys(t));

Remarks: Constraints: This function does not participate in overload resolution unless SourceClock::to_sys(t) is well-formed. If Mandates: SourceClock::to_sys(t) does not return sys_time<Duration>, where Duration is a valid chrono::duration specialization, the program is ill-formed.

Returns: SourceClock::to_sys(t).
```

```cpp
template<class DestClock>
struct clock_time_conversion<DestClock, system_clock> {
    template<class Duration>
    auto operator()(const sys_time<Duration>& t) const
    -> decltype(DestClock::from_sys(t));
};

template<class Duration>
auto operator()(const sys_time<Duration>& t) const
-> decltype(DestClock::from_sys(t));

Remarks: Constraints: This function does not participate in overload resolution unless DestClock::from_sys(t) is well-formed. If Mandates: DestClock::from_sys(t) does not return time_point<DestClock, Duration>, where Duration is a valid chrono::duration specialization, the program is ill-formed.

Returns: DestClock::from_sys(t).
```

27.7.9.5 Conversions between utc_clock and other clocks

```cpp
template<class SourceClock>
struct clock_time_conversion<utc_clock, SourceClock> {
    template<class Duration>
    auto operator()(const time_point<SourceClock, Duration>& t) const
    -> decltype(SourceClock::to_utc(t));
};

template<class Duration>
auto operator()(const time_point<SourceClock, Duration>& t) const
-> decltype(SourceClock::to_utc(t));

Remarks: Constraints: This function does not participate in overload resolution unless SourceClock::to_utc(t) is well-formed. If Mandates: SourceClock::to_utc(t) does not return utc_time<Duration>, where Duration is a valid chrono::duration specialization, the program is ill-formed.

Returns: SourceClock::to_utc(t).
```

```cpp
template<class DestClock>
struct clock_time_conversion<DestClock, utc_clock> {
    template<class Duration>
    auto operator()(const utc_time<Duration>& t) const
    -> decltype(DestClock::from_utc(t));
};

template<class Duration>
auto operator()(const utc_time<Duration>& t) const
-> decltype(DestClock::from_utc(t));

Remarks: Constraints: This function does not participate in overload resolution unless DestClock::from_utc(t) is well-formed. If Mandates: DestClock::from_utc(t) does not return time_point<DestClock, Duration>, where Duration is a valid chrono::duration specialization, the program is ill-formed.

Returns: DestClock::from_utc(t).
```
\section*{27.7.9.6 Function template \texttt{clock\_cast}}

\begin{verbatim}
template<class DestClock, class SourceClock, class Duration>
auto clock_cast(const time_point<SourceClock, Duration>& t);
\end{verbatim}

1 \textbf{Remarks - Constraints:} This function does not participate in overload resolution unless at least one of the following clock time conversion expressions is well-formed:

\begin{enumerate}
\item clock_time_conversion<DestClock, SourceClock>{(t)}
\item clock_time_conversion<DestClock, system_clock>{
  clock_time_conversion<system_clock, SourceClock>{(t)}
}\item clock_time_conversion<DestClock, utc_clock>{
  clock_time_conversion<utc_clock, SourceClock>{(t)}
}\item clock_time_conversion<DestClock, utc_clock>{
  clock_time_conversion<utc_clock, system_clock>{
    clock_time_conversion<system_clock, SourceClock>{(t)}}
}\item clock_time_conversion<DestClock, system_clock>{
  clock_time_conversion<system_clock, utc_clock>{
    clock_time_conversion<utc_clock, SourceClock>{(t)}}
\end{enumerate}

A clock time conversion expression is considered better than another clock time conversion expression if it involves fewer \texttt{operator()} calls on \texttt{clock\_time\_conversion} specializations. If, among the well-formed clock time conversion expressions from the above list, there is not a unique best expression, the \texttt{clock\_cast} is ambiguous and the program is ill-formed.

2 \textbf{Returns:} The best well-formed clock time conversion expression in the above list.

\section*{27.8 The civil calendar}

\subsection*{27.8.1 In general}

The types in \section{27.8} describe the civil (Gregorian) calendar and its relationship to \texttt{sys\_days} and \texttt{local\_days}.

\subsection*{27.8.2 Class last\_spec}

\begin{verbatim}
namespace std::chrono {
  struct last_spec {
    explicit last_spec() = default;
  };
}
\end{verbatim}

The type \texttt{last\_spec} is used in conjunction with other calendar types to specify the last in a sequence. For example, depending on context, it can represent the last day of a month, or the last day of the week of a month.

\subsection*{27.8.3 Class day}

\begin{verbatim}
namespace std::chrono {
  class day {
    unsigned char d_;    // exposition only
  public:
    day() = default;
    constexpr explicit day(unsigned d) noexcept;
    constexpr day& operator++() noexcept;
    constexpr day 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;
  }
}
\end{verbatim}

§ 27.8.3.1
constexpr explicit operator unsigned() const noexcept;
constexpr bool ok() const noexcept;
};

1 day represents a day of a month. It normally holds values in the range 1 to 31, but may hold non-negative values outside this range. It can be constructed with any unsigned value, which will be subsequently truncated to fit into day’s unspecified internal storage. day must meet the Cpp17EqualityComparable (Table ??) and Cpp17LessThanComparable (Table ??) requirements, and participates in basic arithmetic with days objects, which represent a difference between two day objects.

day is a trivially copyable and standard-layout class type.

27.8.3.2 Member functions

constexpr explicit day(unsigned d) noexcept;
1 Effects: Constructs an object of type day by initializing init initializes d_ with d. The value held is unspecified if d is not in the range [0, 255].

constexpr day& operator++() noexcept;
2 Effects: ++d_.
3 Returns: *this.

constexpr day operator++(int) noexcept;
4 Effects: ++(*this).
5 Returns: A copy of *this as it existed on entry to this member function.

constexpr day& operator--() noexcept;
6 Effects: --d_.
7 Returns: *this.

constexpr day operator--(int) noexcept;
8 Effects: --(*this).
9 Returns: A copy of *this as it existed on entry to this member function.

constexpr day& operator+=(const days& d) noexcept;
10 Effects: *this = *this + d.
11 Returns: *this.

constexpr day& operator-=(const days& d) noexcept;
12 Effects: *this = *this - d.
13 Returns: *this.

constexpr explicit operator unsigned() const noexcept;
14 Returns: d_.

constexpr bool ok() const noexcept;
15 Returns: 1 <= d_ && d_ <= 31.

27.8.3.3 Non-member functions

constexpr bool operator==(const day& x, const day& y) noexcept;
1 Returns: unsigned{x} == unsigned{y}.

constexpr strong_ordering operator<=>(const day& x, const day& y) noexcept;
2 Returns: unsigned{x} <=> unsigned{y}.

constexpr day operator+(const day& x, const days& y) noexcept;
3 Returns: day(unsigned{x} + y.count()).
constexpr day operator+(const days& x, const day& y) noexcept;

Returns: y + x.

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

Returns: x + -y.

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

Returns: days{int(unsigned{x}) - int(unsigned{y})}.

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const day& d);

Effects: Equivalent to:

return os << (d.ok() ?
    format(STATICALLY-WIDEN<charT>("{:%d}"), d) :
    format(STATICALLY-WIDEN<charT>("{:%d} is not a valid day"), d));

template<class charT, class traits, class Alloc = allocator<charT>>
basic_istream<charT, traits>&
from_stream(basic_istream<charT, traits>& is, const charT* fmt,
    day& d, basic_string<charT, traits, Alloc>* abbrev = nullptr,
    minutes* offset = nullptr);

Effects: Attempts to parse the input stream is into the day d using the format flags given in the NTCTS fmt as specified in 27.13. If the parse fails to decode a valid day, is.setstate(ios_base::failbit) shall be called and d shall not be modified. If %Z is used and successfully parsed, that value will be assigned to *abbrev if abbrev is non-null. If %z (or a modified variant) is used and successfully parsed, that value will be assigned to *offset if offset is non-null.

Returns: is.

cconstexpr day operator"d(unsigned long long d) noexcept;

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

§ 27.8.4 Class month

27.8.4.1 Overview

namespace std::chrono {

class month {

    unsigned char m_; // exposition only

public:

    month() = default;

cconstexpr explicit month(unsigned m) noexcept;

cconstexpr month& operator++() noexcept;

cconstexpr month operator++(int) noexcept;

cconstexpr month& operator--() noexcept;

cconstexpr month operator--(int) noexcept;

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

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

cconstexpr explicit operator unsigned() const noexcept;

cconstexpr bool ok() const noexcept;

};
}

1 month represents a month of a year. It normally holds values in the range 1 to 12, but may hold non-negative values outside this range. It can be constructed with any unsigned value, which will be subsequently truncated to fit into month’s unspecified internal storage. month meets the Cpp17EqualityComparable (Table ??) and Cpp17LessThanComparable (Table ??) requirements, and participates in basic arithmetic with months objects, which represent a difference between two month objects.

2 month is a trivially copyable and standard-layout class type.
27.8.4.2 Member functions

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

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

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

Effects: *this += months{1}.
Returns: *this.
```

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

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

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

Effects: *this -= months{1}.
Returns: *this.
```

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

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

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

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

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

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

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

Returns: m_.
```

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

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

27.8.4.3 Non-member functions

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

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

```cpp
constexpr strong_ordering operator<=>(const month& x, const month& y) noexcept;

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

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

Returns: month{modulo(static_cast<long long>(unsigned{x}) + (y.count() - 1), 12) + 1}
where modulo(n, 12) computes the remainder of n divided by 12 using Euclidean division. [Note: Given a divisor of 12, Euclidean division truncates towards negative infinity and always produces a remainder in the range of [0, 11]. Assuming no overflow in the signed summation, this operation results in a month holding a value in the range [1, 12] even if !x.ok(). —end note] [Example: February + months{11} == January. —end example]
```

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

Returns: y + x.
```
constexpr month operator-(const month& x, const month& y) noexcept;

Returns: \( x + -y \).

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

Returns: If \( x.ok() == \text{true} \) and \( y.ok() == \text{true} \), returns a value \( m \) in the range \([\text{months}\{0\}, \text{months}\{11\}]\) satisfying \( y + m == x \). Otherwise the value returned is unspecified. [Example: January - February == months\{11\}. — end example]

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const month& m);

Effects: Equivalent to:

\[
\begin{align*}
& \text{return os } \ll \text{(m.ok())} \? \\
& \quad \text{format(os.getloc(), STATICALLY-WIDEN<charT>(&{\text{"{:%b}\"}}, m)) :} \\
& \quad \text{format(os.getloc(), STATICALLY-WIDEN<charT>(&{" is not a valid month"}, static_cast<unsigned>(m))));
\end{align*}
\]

template<class charT, class traits, class Alloc = allocator<charT>>
basic_istream<charT, traits>&
from_stream(basic_istream<charT, traits>& is, const charT* fmt, const month& m, basic_string<charT, traits, Alloc>* abbrev = nullptr, minutes* offset = nullptr);

Effects: Attempts to parse the input stream \( \text{is} \) into the month \( m \) using the format flags given in the NTCTS \( \text{fmt} \) as specified in 27.13. If the parse fails to decode a valid month, \( \text{is.setstate(ios_\text{-base::failbit})} \) shall be called and \( m \) shall not be modified. If \%Z is used and successfully parsed, that value will be assigned to \(*\text{abbrev}\) if \text{abbrev}\) is non-null. If \%z (or a modified variant) is used and successfully parsed, that value will be assigned to \(*\text{offset}\) if \text{offset}\) is non-null.

Returns: \( \text{is} \).

27.8.5 Class year

27.8.5.1 Overview

namespace std::chrono {

class year {

  short y_; // exposition only

public:

  year() = default;

  constexpr explicit 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;

};

} // namespace std::chrono

year represents a year in the civil calendar. It can represent values in the range \([\text{min()}, \text{max()}]\). It can be constructed with any \text{int} value, which will be subsequently truncated to fit into year’s unspecified internal
storage. year meets the Cpp17EqualityComparable (Table ??) and Cpp17LessThanComparable (Table ??) requirements, and participates in basic arithmetic with years objects, which represent a difference between two year objects.

2 year is a trivially copyable and standard-layout class type.

### 27.8.5.2 Member functions

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

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

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

2 **Effects:** ++y_.

3 **Returns:** *this.

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

4 **Effects:** ++(*this).

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

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

6 **Effects:** --y_.

7 **Returns:** *this.

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

8 **Effects:** --(*this).

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

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

10 **Effects:** *this = *this + y.

11 **Returns:** *this.

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

12 **Effects:** *this = *this - y.

13 **Returns:** *this.

```cpp
constexpr year operator+() const noexcept;
```

14 **Returns:** *this.

```cpp
constexpr year year::operator-() const noexcept;
```

15 **Returns:** year{-y_}.

```cpp
constexpr bool is_leap() const noexcept;
```

16 **Returns:** y_ % 4 == 0 && (y_ % 100 != 0 || y_ % 400 == 0).

```cpp
constexpr explicit operator int() const noexcept;
```

17 **Returns:** y_.

```cpp
constexpr bool ok() const noexcept;
```

18 **Returns:** min().y_ <= y_ && y_ <= max().y_.

static constexpr year min() noexcept;

19 **Returns:** year{-32767}.

static constexpr year max() noexcept;

20 **Returns:** year{32767}.
27.8.5.3 Non-member functions

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

Returns: int(x) == int(y).

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

Returns: int(x) <=> int(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}}.

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const year& y);

Effects: Equivalent to:
return os << (y.ok() ?
    format(STATICALLY-WIDEN<charT>("{:%Y}"), y) :
    format(STATICALLY-WIDEN<charT>("{:%Y} is not a valid year"), y));

template<class charT, class traits, class Alloc = allocator<charT>>
basic_istream<charT, traits>&
from_stream(basic_istream<charT, traits>& is, const charT* fmt,
year& y, basic_string<charT, traits, Alloc>* abbrev = nullptr,
minutes* offset = nullptr);

Effects: Attempts to parse the input stream is into the year y using the format flags given in the NTCTS fmt as specified in 27.13. If the parse fails to decode a valid year, is.setstate(ios_base::failbit) shall be called and y shall not be modified. If %Z is used and successfully parsed, that value will be assigned to *abbrev if abbrev is non-null. If %z (or a modified variant) is used and successfully parsed, that value will be assigned to *offset if offset is non-null.

Returns: is.

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

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

27.8.6 Class weekday

27.8.6.1 Overview

namespace std::chrono {

    class weekday {
        unsigned char wd_;  // exposition only

        public:
            weekday() = default;
            constexpr explicit 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;

    } // namespace std::chrono

§ 27.8.6.1
weekday represents a day of the week in the civil calendar. It normally holds values in the range 0 to 6, corresponding to Sunday through Saturday, but it may hold non-negative values outside this range. It can be constructed with any unsigned value, which will be subsequently truncated to fit into weekday’s unspecified internal storage. weekday meets the Cpp17EqualityComparable requirements. [Note: weekday is not Cpp17LessThanComparable because there is no universal consensus on which day is the first day of the week. weekday’s arithmetic operations treat the days of the week as a circular range, with no beginning and no end. —end note]

weekday is a trivially copyable and standard-layout class type.

27.8.6.2 Member functions

constexpr explicit weekday(unsigned wd) noexcept;

Effects: Constructs an object of type weekday by initializing wd_ with wd == 7 ? 0 : wd. The value held is unspecified if wd is not in the range [0, 255].

constexpr weekday(const sys_days& dp) noexcept;

Effects: Computes what day of the week corresponds to the sys_days dp, and representing initializes that day of the week in wd_.

[Example: If dp represents 1970-01-01, the constructed weekday represents Thursday by storing 4 in wd_. —end example]

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 initializes that day of the week in wd_.

Remarks: Ensures: The value after construction is identical to that constructed from sys_days{dp.time_-since_epoch()}.
constexpr weekday& operator-=(const days& d) noexcept;
Effects: *this = *this - d.
Returns: *this.

constexpr unsigned c_encoding() const noexcept;
Returns: wd_.

constexpr unsigned iso_encoding() const noexcept;
Returns: wd_ == 0u ? 7u : wd_.

constexpr bool ok() const noexcept;
Returns: wd_ <= 6.

constexpr weekday_indexed operator[](unsigned index) const noexcept;
Returns: {*this, index}.

constexpr weekday_last operator[]() const noexcept;
Returns: weekday_last{*this}.

27.8.6.3 Non-member functions [time.cal.wd.nonmembers]

constexpr bool operator==(const weekday& x, const weekday& y) noexcept;
Returns: x.wd_ == y.wd_.

constexpr weekday operator+(const weekday& x, const days& y) noexcept;
Returns: weekday{modulo(static_cast<long long>(x.wd_) + y.count(), 7)}
where modulo(n, 7) computes the remainder of n divided by 7 using Euclidean division. [Note: Given a
divisor of 7, Euclidean division truncates towards negative infinity and always produces a remainder
in the range of [0, 6]. Assuming no overflow in the signed summation, this operation results in a
weekday holding a value in the range [0, 6] even if !x.ok(). —end note] [Example: Monday +
days{6} == Sunday. —end example]

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.wd_ == y.wd_.

constexpr days operator-(const weekday& x, const weekday& y) noexcept;
Returns: If x.ok() == true and y.ok() == true, returns a value d in the range [days{0}, days{6}]
satisfying y + d == x. Otherwise the value returned is unspecified. [Example: Sunday - Monday ==
days{6}. —end example]

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const weekday& wd);
Effects: Equivalent to:
return os << (wd.ok() ?
   format(os.getloc(), STATICALLY-WIDEN<charT>("{%a}"), wd) :
   format(os.getloc(), STATICALLY-WIDEN<charT>("{} is not a valid weekday"),
   static_cast<unsigned>(wd.wd_));

template<class charT, class traits, class Alloc = allocator<charT>>
basic_istream<charT, traits>&
from_stream(basic_istream<charT, traits>& is, const charT* fmt,
weekday& wd, basic_string<charT, traits, Alloc>* abbrev = nullptr,
minutes* offset = nullptr);

7 Effects: Attempts to parse the input stream is into the weekday wd using the format flags given in the NTCTS fmt as specified in 27.13. If the parse fails to decode a valid weekday, is.setstate(ios_base::failbit) shall be called and wd shall not be modified. If %Z is used and successfully parsed, that value will be assigned to *abbrev if abbrev is non-null. If %z (or a modified variant) is used and successfully parsed, that value will be assigned to *offset if offset is non-null.

8 Returns: is.

27.8.7 Class weekday_indexed

27.8.7.1 Overview

namespace std::chrono {
    class weekday_indexed {
        chrono::weekday wd_;     // exposition only
        unsigned char index_;    // exposition only

        public:
            weekday_indexed() = default;
            constexpr weekday_indexed(const chrono::weekday& wd, unsigned index) noexcept;

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

1 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.

2 [Note: A weekday_indexed object can be constructed by indexing a weekday with an unsigned. —end note]  [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]

3 weekday_indexed is a trivially copyable and standard-layout class type.

27.8.7.2 Member functions

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

1 Effects: Constructs an object of type weekday_indexed by initializing 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 chrono::weekday weekday() const noexcept;

2 Returns: wd_.

constexpr unsigned index() const noexcept;

3 Returns: index_.

constexpr bool ok() const noexcept;

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

27.8.7.3 Non-member functions

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

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

template<class charT, class traits>
    basic_ostream<charT, traits>& operator<<<basic_ostream<charT, traits>& os, const weekday_indexed& wdi);

2Effects: Equivalent to:
auto i = wdi.index();
return os << (i >= 1 && i <= 5 ?
    format(os.getloc(), STATICALLY-WIDEN<charT>("{}[{}]"), wdi.weekday(), i) :
    format(os.getloc(), STATICALLY-WIDEN<charT>("{}[{} is not a valid index"]"),
        wdi.weekday(), i));

27.8.8 Class weekday_last

27.8.8.1 Overview

namespace std::chrono {

    class weekday_last {
    chrono::weekday wd_; // exposition only

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

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

1 weekday_last represents the last weekday of a month.
2 [Note: A weekday_last object can be constructed by indexing a weekday with last. — end note] [Example:

    constexpr auto wdl = Sunday[last]; // wdl is the last Sunday of an as yet unspecified month
    static_assert(wdl.weekday() == Sunday);
—end example]
3 weekday_last is a trivially copyable and standard-layout class type.

27.8.8.2 Member functions

    constexpr explicit weekday_last(const chrono::weekday& wd) noexcept;
1 Effects: Constructs an object of type weekday_last by initializing wd_ with wd.

    constexpr chrono::weekday weekday() const noexcept;
2 Returns: wd_.

    constexpr bool ok() const noexcept;
3 Returns: wd_.ok().

27.8.8.3 Non-member functions

    constexpr bool operator==(const weekday_last& x, const weekday_last& y) noexcept;
1 Returns: x.weekday() == y.weekday().

    template<class charT, class traits>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const weekday_last& wdl);
2 Effects: Equivalent to:

    return os << format(os.getloc(), STATICALLY-WIDEN<charT>("{}[last]"), wdl.weekday());

27.8.9 Class month_day

27.8.9.1 Overview

namespace std::chrono {

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

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

    };
month_day represents a specific day of a specific month, but with an unspecified year. month_day meets the Cpp17EqualityComparable (Table ??) and Cpp17LessThanComparable (Table ??) requirements.

2 month_day is a trivially copyable and standard-layout class type.

### 27.8.9.2 Member functions

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

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

2 **Returns:** `m_`.

```cpp
constexpr chrono::month month() const noexcept;
```

3 **Returns:** `d_`.

```cpp
constexpr chrono::day day() const noexcept;
```

4 **Returns:** `true` if `m_.ok()` is `true`, `1d <= d_`, and `d_` is less than or equal to the number of days in month `m_`; otherwise returns `false`. When `m_ == February`, the number of days is considered to be 29.

### 27.8.9.3 Non-member functions

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

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

```cpp
constexpr strong_ordering operator<=>(const month_day& x, const month_day& y) noexcept;
```

2 **Effects:** Equivalent to:

```cpp
if (auto c = x.month() <=> y.month(); c != 0) return c;
return x.day() <=> y.day();
```

```cpp
template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const month_day& md);
```

3 **Effects:** Equivalent to:

```cpp
return os << format(os.getloc(), STATICALLY-WIDEN<charT>("/{}/"),
    md.month(), md.day());
```

```cpp
template<class charT, class traits, class Alloc = allocator<charT>>
from_stream(basic_istream<charT, traits>&& is, const charT* fmt,
    month_day& md, basic_string<charT, traits, Alloc>** abbrev = nullptr,
    minutes* offset = nullptr);
```

4 **Effects:** Attempts to parse the input stream `is` into the `month_day md` using the format flags given in the NTCTS `fmt` as specified in 27.13. If the parse fails to decode a valid `month_day`, `is.setstate(ios_base::failbit)` shall be called and `md` shall not be modified. If `%Z` is used and successfully parsed, that value will be assigned to `*abbrev` if `abbrev` is non-null. If `%z` (or a modified variant) is used and successfully parsed, that value will be assigned to `*offset` if `offset` is non-null.

5 **Returns:** `is`.

### 27.8.10 Class month_day_last

```cpp
namespace std::chrono {
    class month_day_last {
        chrono::month m_; // exposition only
    }
}
```
month_day_last represents the last day of a month.

[Note: A month_day_last object can be constructed using the expression m/last or last/m, where m is an expression of type month. — end note] [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]

month_day_last is a trivially copyable and standard-layout class type.

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

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

constexpr month month() const noexcept;

Returns: m_.

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

Returns: x.month() <=> y.month().

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const month_day_last& mdl);

Effects: Equivalent to:
return os << format(os.getloc(), STATICALLY-WIDEN<charT>("{}/last"), mdl.month());

7.8.11 Class month_weekday [time.cal.mwd]

7.8.11.1 Overview [time.cal.mwd.overview]

namespace std::chrono {

class month_weekday {

    chrono::month m_;       // exposition only
    chrono::weekday_indexed wdi_;       // exposition only

public:

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

    constexpr month m() const noexcept;

    constexpr weekday_indexed wdi() const noexcept;

    constexpr bool ok() const noexcept;

};

} // namespace std::chrono

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

[Example:
constexpr auto mwd = February/Tuesday[3]; // mwd is the third Tuesday of February of an as yet unspecified year
static_assert(mwd.month() == February);
static_assert(mwd.weekday_indexed() == Tuesday[3]);

§ 27.8.11.1
3 month_weekday is a trivially copyable and standard-layout class type.

27.8.11.2 Member functions

constexpr month_weekday(const chrono::month& m, const chrono::weekday_indexed& wdi) noexcept;
1 Effects: Constructs an object of type month_weekday by initializing m_ with m, and wdi_ with wdi.

constexpr chrono::month month() const noexcept;
2 Effects: Equivalent to:

constexpr chrono::weekday_indexed weekday_indexed() const noexcept;
3 Effects: Equivalent to:

constexpr bool ok() const noexcept;
4 Returns: m_.ok() && wdi_.ok().

27.8.11.3 Non-member functions

constexpr bool operator==(const month_weekday& x, const month_weekday& y) noexcept;
1 Returns: x.month() == y.month() && x.weekday_indexed() == y.weekday_indexed().

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const month_weekday& mwd);
2 Effects: Equivalent to:

27.8.12 Class month_weekday_last

27.8.12.1 Overview

namespace std::chrono {
    class month_weekday_last {
        chrono::month m_; // exposition only
        chrono::weekday_last wdl_; // exposition only

        public:
            constexpr month_weekday_last(const chrono::month& m,
                                          const chrono::weekday_last& wdl) noexcept;

            constexpr chrono::month month() const noexcept;

            constexpr chrono::weekday_last weekday_last() const noexcept;

            constexpr bool ok() const noexcept;
    };
}

1 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.

2 Example:

    constexpr auto mwd = February/Tuesday[last]; // mwd is the last Tuesday of February of an as yet unspecified year
    static_assert(mwd.month() == February);
    static_assert(mwd.weekday_last() == Tuesday[last]);

   — end example

3 month_weekday_last is a trivially copyable and standard-layout class type.
27.8.12.2 Member functions

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

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

```cpp
constexpr chrono::month month() const noexcept;
```

**Returns:** `m_`.

```cpp
constexpr chrono::weekday_last weekday_last() const noexcept;
```

**Returns:** `wdl_`.

```cpp
constexpr bool ok() const noexcept;
```

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

27.8.12.3 Non-member functions

```cpp
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()`.

```cpp
template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const month_weekday_last& mwdl);
```

**Effects:** Equivalent to:
```
return os << format(os.getloc(),
"\{}/{}",
mwdl.month(), mwdl.weekday_last());
```

27.8.13 Class year_month

27.8.13.1 Overview

```cpp
namespace std::chrono {
    class year_month {
    // exposition only
    chrono::year y_;   // exposition only
    chrono::month m_;   // exposition only

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

**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**. **year_month** meets the **Cpp17EqualityComparable** (Table ??) and **Cpp17LessThanComparable** (Table ??) requirements.

**year_month** is a trivially copyable and standard-layout class type.

27.8.13.2 Member functions

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

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

§ 27.8.13.2
constexpr chrono::year year() const noexcept;
Returns: \( y_\). 

constexpr chrono::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 ok() const noexcept;
Returns: \( y_.ok() && m_.ok() \).

27.8.13.3 Non-member functions

constexpr bool operator==(const year_month& x, const year_month& y) noexcept;
Returns: \( x.year() == y.year() && x.month() == y.month() \).

constexpr strong_ordering operator<=>(const year_month& x, const year_month& y) noexcept;
Effects: Equivalent to:
\[
\text{if (auto c = x.year() <=> y.year(); c != 0) return c; return x.month() <=> y.month();}
\]

constexpr year_month operator+(const year_month& ym, const months& dm) noexcept;
Returns: A year_month value \( z \) such that \( z - ym == dm \).
Complexity: \( \Theta(1) \) 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:
\[
x.year() - y.year() + months\{static\_cast\<\text{int}\>\{\text{unsigned}\{x.month()\}\} - \\
\text{static\_cast\<\text{int}\>\{\text{unsigned}\{y.month()\}\}}\}
\]

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>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>&& os, const year_month& ym);

Effects: Equivalent to:
return os << format(os.getloc(), STATICALLY-WIDEN<charT>("{}/{}"),
ym.year(), ym.month());

template<class charT, class traits, class Alloc = allocator<charT>>
basic_istream<charT, traits>&
from_stream(basic_istream<charT, traits>&& is, const charT* fmt,
year_month& ym, basic_string<charT, traits, Alloc>* abbrev = nullptr,
minutes* offset = nullptr);

Effects: Attempts to parse the input stream is into the year_month ym using the format flags given in the
NTCTS fmt as specified in 27.13. If the parse fails to decode a valid year_month, is.setstate(ios_base::failbit)
shall be called and ym shall not be modified. If %Z is used and successfully parsed,
that value will be assigned to *abbrev if abbrev is non-null. If %z (or a modified variant) is used and
successfully parsed, that value will be assigned to *offset if offset is non-null.

Returns: is.

27.8.14 Class year_month_day

namespace std::chrono {

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

public:
    year_month_day() = default;
    constexpr year_month_day(const chrono::year& y, const chrono::month& m,
        const chrono::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 chrono::year year() const noexcept;
    constexpr chrono::month month() const noexcept;
    constexpr chrono::day day() const noexcept;
    constexpr operator sys_days() const noexcept;
    constexpr explicit operator local_days() const noexcept;
    constexpr bool ok() const 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 chrono::year year() const noexcept;
    constexpr chrono::month month() const noexcept;
    constexpr chrono::day day() const noexcept;

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

    template<class charT, class traits, class Alloc = allocator<charT>>
    basic_istream<charT, traits>&
    from_stream(basic_istream<charT, traits>&& is, const charT* fmt,
        year_month& ym, basic_string<charT, traits, Alloc>* abbrev = nullptr,
        minutes* offset = nullptr);
    
    template<class charT, class traits>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>&& os, const year_month& ym);
    
    Effects: Equivalent to:
    return os << format(os.getloc(), STATICALLY-WIDEN<charT>("{}/{}"),
        ym.year(), ym.month());

    template<class charT, class traits, class Alloc = allocator<charT>>
    basic_istream<charT, traits>&
    from_stream(basic_istream<charT, traits>&& is, const charT* fmt,
        year_month& ym, basic_string<charT, traits, Alloc>* abbrev = nullptr,
        minutes* offset = nullptr);

    Effects: Attempts to parse the input stream is into the year_month ym using the format flags given in the
    NTCTS fmt as specified in 27.13. If the parse fails to decode a valid year_month, is.setstate(ios_base::failbit)
    shall be called and ym shall not be modified. If %Z is used and successfully parsed,
    that value will be assigned to *abbrev if abbrev is non-null. If %z (or a modified variant) is used and
    successfully parsed, that value will be assigned to *offset if offset is non-null.

    Returns: is.

1 year_month_day represents a specific year, month, and day. year_month_day is a field-based time point
with a resolution of days. [Note: 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. — end note] year_month_day meets the Cpp17EqualityComparable (Table ??)
and Cpp17LessThanComparable (Table ??) requirements.

2 year_month_day is a trivially copyable and standard-layout class type.
27.8.14.2 Member functions

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

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

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

**Effects:** Constructs an object of type `year_month_day` by initializing `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` may be more efficient than converting a `year_month_day_last` to a `sys_days`, and then converting that `sys_days` to a `year_month_day`. —end note]

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

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

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

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

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

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

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

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

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

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

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

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

```cpp
constexpr chrono::year year() const noexcept;
```

**Returns:** `y_`.

```cpp
constexpr chrono::month month() const noexcept;
```

**Returns:** `m_`.

```cpp
constexpr chrono::day day() const noexcept;
```

**Returns:** `d_`.

```cpp
constexpr 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()` is true, returns `sys_days{y_/m_/1d} + (d_ - 1d)`. 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

cconstexpr explicit operator local_days() const noexcept;

Returns: local_days{sys_days{*this}.time_since_epoch()}.  

cconstexpr bool ok() const noexcept;

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

27.8.14.3 Non-member functions

cconstexpr 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().

ccconstexpr strong_ordering operator<=>(const year_month_day& x, const year_month_day& y) noexcept;

Effects: Equivalent to:

if (auto c = x.year() <=> y.year(); c != 0) return c;
if (auto c = x.month() <=> y.month(); c != 0) return c;
return x.day() <=> y.day();

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

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

[Note: If ymd.day() is in the range [1d, 28d], ok() will return true for the resultant year_month_day.
— end note]

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

Returns: ymd + dm.

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

Returns: ymd + (-dm).

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

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

[Note: If ymd.month() is February and ymd.day() is not in the range [1d, 28d], ok() may return false for the resultant year_month_day. — end note]

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

Returns: ymd + dy.

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

Returns: ymd + (-dy).

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const year_month_day& ymd);

Effects: Equivalent to:

return os << (ymd.ok() ?
format(STATICALLY-WIDEN<charT>("{:%F}", ymd) :
format(STATICALLY-WIDEN<charT>("{:%F} is not a valid date"), ymd));
Effects: Attempts to parse the input stream is into the `year_month_day ymd` using the format flags given in the NTCTS `fmt` as specified in 27.13. If the parse fails to decode a valid `year_month_day`, `is.setstate(ios_base::failbit)` shall be called and `ymd` shall not be modified. If `%Z` is used and successfully parsed, that value will be assigned to `*abbrev` if `abbrev` is non-null. If `%z` (or a modified variant) is used and successfully parsed, that value will be assigned to `*offset` if `offset` is non-null.

Returns: is.

### 27.8.15 Class `year_month_day_last` [time.cal.ymdlast]

#### 27.8.15.1 Overview [time.cal.ymdlast.overview]

```cpp
namespace std::chrono {
    class year_month_day_last {
        chrono::year y_; // exposition only
        chrono::month_day_last mdl_; // exposition only

        public:
            constexpr year_month_day_last(const chrono::year& y, const chrono::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 chrono::year year() const noexcept;
            constexpr chrono::month month() const noexcept;
            constexpr chrono::month_day_last month_day_last() const noexcept;
            constexpr chrono::day day() const noexcept;
            constexpr operator sys_days() const noexcept;
            constexpr explicit operator local_days() const noexcept;
            constexpr bool ok() const noexcept;
    };
}
```

`year_month_day_last` represents the last day of a specific year and 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. [Note: `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. —end note] `year_month_day_last` meets the Cpp17EqualityComparable (Table ??) and Cpp17LessThanComparable (Table ??) requirements.

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

#### 27.8.15.2 Member functions [time.cal.ymdlast.members]

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

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

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

Effects: *this = *this + m.

Returns: *this.

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

Effects: *this = *this - m.

Returns: *this.
constexpr year_month_day_last& operator+=(const years& y) noexcept;

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

castexpr year_month_day_last& operator-=(const years& y) noexcept;

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

castexpr year_month_day_last& operator+(const years& dy) noexcept;

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

§ 27.8.15.3 58
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>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const year_month_day_last& ymdl);

Effects: Equivalent to:
return os << format(os.getloc(), STATICALLY-WIDEN<charT>("{}/{}"),
ymdl.year(), ymdl.month_day_last());

27.8.16 Class year_month_day_last

27.8.16.1 Overview

namespace std::chrono {
    class year_month_day_last {
        // exposition only
        chrono::year y_;    
        chrono::month m_;   
        chrono::weekday_indexed wdi_;  

    public:
        year_month_day_last() = default;
        constexpr year_month_day_last(const chrono::year& y, const chrono::month& m,
                                       const chrono::weekday_indexed& wdi) noexcept;
        constexpr explicit year_month_day_last(const sys_days& dp) noexcept;
        constexpr explicit year_month_day_last(const local_days& dp) 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 chrono::year year() const noexcept;
        constexpr chrono::month month() const noexcept;
        constexpr chrono::weekday weekday() const noexcept;
        constexpr unsigned index() const noexcept;
        constexpr chrono::weekday_indexed weekday_indexed() const noexcept;
        constexpr operator sys_days() const noexcept;
        constexpr explicit operator local_days() const noexcept;
        constexpr bool ok() const noexcept;
    };
}

1 year_month_day_last represents a specific year, month, and \(n\)th weekday of the month. year_month_day_last is a field-based time point with a resolution of days. [Note: 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. — end note] year_month_day_last is Cpp17EqualityComparable (Table ??).

2 year_month_day_last is a trivially copyable and standard-layout class type.

27.8.16.2 Member functions

constexpr year_month_day_last(const chrono::year& y, const chrono::month& m,
                             const chrono::weekday_indexed& wdi) noexcept;

Effects: Constructs an object of type year_month_day_last by initializing 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 ymdl of type year_month_weekday for which ymdl.ok() is true, ymdl == year_month_weekday{sys_days{ymdl}} is true.

constexpr explicit year_month_weekday(const local_days& dp) noexcept;

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

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

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

Effects: *this = *this + m.

Returns: *this.

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

Effects: *this = *this - m.

Returns: *this.

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

Effects: *this = *this + y.

Returns: *this.

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

Effects: *this = *this - y.

Returns: *this.

constexpr chrono::year year() const noexcept;

Returns: y_.

constexpr chrono::month month() const noexcept;

Returns: m_.

constexpr chrono::weekday weekday() const noexcept;

Returns: wdi_.weekday().

constexpr unsigned index() const noexcept;

Returns: wdi_.index().

constexpr chrono::weekday_indexed weekday_indexed() const noexcept;

Returns: wdi_.

constexpr operator sys_days() const noexcept;

Returns: If y_.ok() && m_.ok() && wdi_.weekday().ok(), returns a sys_days that 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 the returned value is unspecified.

constexpr explicit operator local_days() const noexcept;

Returns: local_days{sys_days{*this}.time_since_epoch()}.  

constexpr bool ok() const noexcept;

Returns: If any of y_.ok(), m_.ok(), or wdi_.ok() is false, returns false. Otherwise, if *this represents a valid date, returns true. Otherwise, returns false.
27.8.16.3 Non-member functions

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

Returns:
\( x.\text{year}() == y.\text{year}() \&\& x.\text{month}() == y.\text{month}() \&\& x.\text{weekday\_indexed}() == y.\text{weekday\_indexed}() \)

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

Returns: \( (ymwd.\text{year}() / ymwd.\text{month}() + dm) / ymwd.\text{weekday\_indexed}() \).

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: \( ymwd.\text{year}() + dy, ymwd.\text{month}(), ymwd.\text{weekday\_indexed}() \).

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

Returns: ymwd + dy.

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

Returns: ymwd + (-dy).

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const year_month_weekday& ymwd);

Effects: Equivalent to:
\[
\text{return os} \ll \text{format(os.getloc(), STATICALLY-WIDEN<charT>("{}\slash{}\slash{}"),}
\text{ymwd.\text{year}(), ymwd.\text{month}(), ymwd.\text{weekday\_indexed}());}
\]
```

27.8.17 Class year_month_weekday_last

27.8.17.1 Overview

```
namespace std::chrono {

class year_month_weekday_last {

\text{chrono::year y_;} \quad \text{// exposition only}
\text{chrono::month m_;} \quad \text{// exposition only}
\text{chrono::weekday\_last wdl_;} \quad \text{// exposition only}

\text{public:}
\text{constexpr year_month_weekday_last(const chrono::year& y, const chrono::month& m,}
\text{ const chrono::weekday\_last& wdl) noexcept;}

\text{constexpr year_month_weekday_last\_last\_operator\_}(\text{const months& m) noexcept;}
\text{constexpr year_month_weekday_last\_last\_operator\_}(\text{const months& m) noexcept;}
\text{constexpr year_month_weekday_last\_last\_operator\_}(\text{const years& y) noexcept;}
\text{constexpr year_month_weekday_last\_last\_operator\_}(\text{const years& y) noexcept;}

\text{constexpr chrono::year year(); const noexcept;}
\text{constexpr chrono::month month(); const noexcept;}
\text{constexpr chrono::weekday\_last weekday(); const noexcept;}
\text{constexpr chrono::\_last\_weekday\_last\_last\_last\_last\_last\_last() const noexcept;}
\text{constexpr operator sys\_days() const noexcept;}
\text{constexpr explicit operator local\_days() const noexcept;}
\text{constexpr bool ok() const noexcept;}
}
```

1 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. [Note: \texttt{year\_month\_weekday\_last} supports years- and months-oriented arithmetic, but not days-oriented arithmetic. For the latter, there is a conversion to \texttt{sys\_days}, which efficiently supports days-oriented arithmetic. — end note] \texttt{year\_month\_weekday\_last} is \texttt{C++17EqualityComparable} (Table ??).

\texttt{year\_month\_weekday\_last} is a trivially copyable and standard-layout class type.

### 27.8.17.2 Member functions

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

\textbf{Effects:} Constructs an object of type \texttt{year\_month\_weekday\_last} by initializing \texttt{y\_} with \texttt{y}, \texttt{m\_} with \texttt{m}, and \texttt{wdl\_} with \texttt{wdl}.

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

\textbf{Effects:} \texttt{*this} = \texttt{*this} + \texttt{m}.

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

\textbf{Effects:} \texttt{*this} = \texttt{*this} - \texttt{m}.

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

\textbf{Effects:} \texttt{*this} = \texttt{*this} + \texttt{y}.

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

\textbf{Effects:} \texttt{*this} = \texttt{*this} - \texttt{y}.

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

\textbf{Returns:} \texttt{y\_}.

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

\textbf{Returns:} \texttt{m\_}.

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

\textbf{Returns:} \texttt{wdl\_.weekday()}.

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

\textbf{Returns:} \texttt{wdl\_}.

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

\textbf{Returns:} If \texttt{ok()} == \texttt{true}, returns a \texttt{sys\_days} that represents the last \texttt{weekday()} of \texttt{year()}/\texttt{month()}. Otherwise the returned value is unspecified.

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

\textbf{Returns:} \texttt{local\_days\{sys\_days\{*this\}.time\_since\_epoch\()\}}.

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

\textbf{Returns:} \texttt{y\_.ok()} \texttt{&& m\_.ok()} \texttt{&& wdl\_.ok()}.

### 27.8.17.3 Non-member functions

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

\textbf{Returns:} \texttt{x.year() == y.year()} \texttt{&& x.month() == y.month()} \texttt{&& x.weekday\_last() == y.weekday\_last()}

\§ 27.8.17.3
```cpp
constexpr year_month_weekday_last
operator+(const year_month_weekday_last& ymwdl, const months& dm) noexcept;

2 Returns: (ymwdl.year() / ymwdl.month() + dm) / ymwdl.weekday_last().

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

3 Returns: ymwdl + dm.

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

4 Returns: ymwdl + (-dm).

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

5 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;

6 Returns: ymwdl + dy.

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

7 Returns: ymwdl + (-dy).

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const year_month_weekday_last& ymwdl);

8 Effects: Equivalent to:

return os << format(os.getloc(), STATICALLY-WIDEN<charT>("{}/{}/{}"),
                    ymwdl.year(), ymwdl.month(), ymwdl.weekday_last());
```

### 27.8.18 Conventional syntax operators

A set of overloaded `operator/` functions provides a conventional syntax for the creation of civil calendar dates.

2 [Note: The year, month, and day are accepted in any of the following 3 orders:

- `year/month/day`
- `month/day/year`
- `day/month/year`

Anywhere a `day` is required, any of the following can also be specified:

- `last`
- `weekday[i]`
- `weekday[last]`

—end note]

3 [Note: Partial-date types such as `year_month` and `month_day` can be created by not applying the second division operator for any of the three orders. For example:

- `year_month ym = 2015y/April;`
- `month_day md1 = April/4;`
- `month_day md2 = 4d/April;`

—end note]

4 [Example:

```cpp
auto a = 2015/4/4;  // a == int(125)
auto b = 2015y/4/4;  // b == year_month_day(year(2015), month(4), day(4))
auto c = 2015y/4d/April;  // a == int(125)
auto d = 2015/4d/April;  // error: no viable operator/ for first /
```—end example]
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).

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: month(m) / d.

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

Returns: month(m) / d.

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.

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.
```cpp
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.

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_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.

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_month& ym, 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.
```

§ 27.8.18
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: \{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: y / mwd.

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: \{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: y / mwdl.

27.9 Class template hh_mm_ss

27.9.1 Overview

namespace std::chrono {
    template<class Duration> class hh_mm_ss {
        public:
            static constexpr unsigned fractional_width = see below;
            using precision = see below;

            constexpr hh_mm_ss() noexcept : hh_mm_ss(Duration::zero()) {}  
            constexpr explicit hh_mm_ss(Duration d) ;

            constexpr bool is_negative() const noexcept;
            constexpr chrono::hours hours() const noexcept;
            constexpr chrono::minutes minutes() const noexcept;
            constexpr chrono::seconds seconds() const noexcept;
            constexpr precision subseconds() const noexcept;

            constexpr explicit operator precision() const noexcept;
            constexpr precision to_duration() const noexcept;
        } 
    } 
}
The `hh_mm_ss` class template splits a `duration` into a multi-field time structure `hours:minutes:seconds` and possibly `subseconds`, where `subseconds` will be a duration unit based on a non-positive power of 10. The `Duration` template parameter dictates the precision to which the time is split. A `hh_mm_ss` models negative durations with a distinct `is_negative` getter that returns `true` when the input duration is negative. The individual duration fields always return non-negative durations even when `is_negative()` indicates the structure is representing a negative duration.

2 If `Duration` is not an instance of `duration`, the program is ill-formed.

### 27.9.2 Members

#### static constexpr unsigned fractional_width = see below;

`fractional_width` is the number of fractional decimal digits represented by `precision`. `fractional_width` has the value of the smallest possible integer in the range `[0, 18]` such that `precision` will exactly represent all values of `Duration`. If no such value of `fractional_width` exists, then `fractional_width` is 6. [Example: See Table 87 for some durations, the resulting `fractional_width`, and the formatted fractional second output of `Duration{1}`.

<table>
<thead>
<tr>
<th>Duration</th>
<th>fractional_width</th>
<th>Formatted fractional second output</th>
</tr>
</thead>
<tbody>
<tr>
<td>hours, minutes, and seconds</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>milliseconds</td>
<td>3</td>
<td>0.001</td>
</tr>
<tr>
<td>microseconds</td>
<td>6</td>
<td>0.000001</td>
</tr>
<tr>
<td>nanoseconds</td>
<td>9</td>
<td>0.000000001</td>
</tr>
<tr>
<td>duration&lt;int, ratio&lt;1, 2&gt;&gt;</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>duration&lt;int, ratio&lt;1, 3&gt;&gt;</td>
<td>6</td>
<td>0.333333</td>
</tr>
<tr>
<td>duration&lt;int, ratio&lt;1, 4&gt;&gt;</td>
<td>2</td>
<td>0.25</td>
</tr>
<tr>
<td>duration&lt;int, ratio&lt;1, 5&gt;&gt;</td>
<td>1</td>
<td>0.2</td>
</tr>
<tr>
<td>duration&lt;int, ratio&lt;1, 6&gt;&gt;</td>
<td>6</td>
<td>0.166666</td>
</tr>
<tr>
<td>duration&lt;int, ratio&lt;1, 7&gt;&gt;</td>
<td>6</td>
<td>0.142857</td>
</tr>
<tr>
<td>duration&lt;int, ratio&lt;1, 8&gt;&gt;</td>
<td>3</td>
<td>0.125</td>
</tr>
<tr>
<td>duration&lt;int, ratio&lt;1, 9&gt;&gt;</td>
<td>6</td>
<td>0.111111</td>
</tr>
<tr>
<td>duration&lt;int, ratio&lt;1, 10&gt;&gt;</td>
<td>1</td>
<td>0.1</td>
</tr>
<tr>
<td>duration&lt;int, ratio&lt;756, 625&gt;&gt;</td>
<td>4</td>
<td>0.2096</td>
</tr>
</tbody>
</table>

— end example]

#### using precision = see below;

`precision` is

`duration<common_type_t<Duration::rep, seconds::rep>, ratio<1, 10^fractional_width>>`

`constexpr explicit hh_mm_ss(Duration d);`

3 **Effects:** Constructs an object of type `hh_mm_ss` which represents the `Duration` `d` with precision `precision`.

(3.1) — Initializes `is_neg` with `d < Duration::zero()`.

(3.2) — Initializes `h` with `duration_cast<chrono::hours>(abs(d))`.

(3.3) — Initializes `m` with `duration_cast<chrono::minutes>(abs(d) - hours())`.

(3.4) — Initializes `s` with `duration_cast<chrono::seconds>(abs(d) - hours() - minutes())`. 

§ 27.9.2
If `treat_as_floating_point_v<precision::rep>` is true, initializes `ss` with `abs(d) - hours() - minutes() - seconds()`. Otherwise, initializes `ss` with `duration_cast<precision>(abs(d) - hours() - minutes() - seconds())`.

**Note:** When `precision::rep` is integral and `precision::period` is `ratio<1>`, `subseconds()` always returns a value equal to 0s. —end note

**Ensures:** If `treat_as_floating_point_v<precision::rep>` is true, `to_duration()` returns `d`, otherwise `to_duration()` returns `duration_cast<precision>(d)`.

```cpp
constexpr bool is_negative() const noexcept;
Returns: `is_neg`.
```

```cpp
constexpr chrono::hours hours() const noexcept;
Returns: `h`.
```

```cpp
constexpr chrono::minutes minutes() const noexcept;
Returns: `m`.
```

```cpp
constexpr chrono::seconds seconds() const noexcept;
Returns: `s`.
```

```cpp
constexpr precision subseconds() const noexcept;
Returns: `ss`.
```

```cpp
constexpr precision to_duration() const noexcept;
Returns: If `is_neg`, returns `-(h + m + s + ss)`, otherwise returns `h + m + s + ss`.
```

```cpp
constexpr explicit operator precision() const noexcept;
Returns: `to_duration()`.
```

### 27.9.3 Non-members

```cpp
template<class charT, class traits, class Duration>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const hh_mm_ss<Duration>& hms);
```

**Effects:** Equivalent to:

```cpp
return os << format(os.getloc(),
    hms.is_negative() ? STATICALLY-WIDEN<charT>("-{:%T}"
        : STATICALLY-WIDEN<charT>("{:%T}",
    abs(hms.to_duration())));
```

```cpp
[Example:

```cpp
for (auto ms : {-4083007ms, 4083007ms, 65745123ms}) {
    hh_mm_ss hms{ms};
    cout << hms << "\n";
}  
cout << hh_mm_ss(65745s) << "\n";

Produces the output (assuming the "C" locale):
-01:08:03.007
01:08:03.007
18:15:45.123
18:15:45
—end example]
```

### 27.10 12/24 hours functions

These functions aid in translating between a 12h format time of day and a 24h format time of day.

```cpp
constexpr bool is_am(const hours& h) noexcept;
Returns: `0h <= h && h <= 11h`.
```
constexpr bool is_pm(const hours& h) noexcept;

Returns: 12h <= h && h <= 23h.

constexpr hours make12(const hours& h) noexcept;

Returns: The 12-hour equivalent of h in the range [1h, 12h]. If h is not in the range [0h, 23h], the value returned is unspecified.

constexpr hours make24(const hours& h, bool is_pm) noexcept;

Returns: If is_pm is false, returns the 24-hour equivalent of h in the range [0h, 11h], assuming h represents an ante meridiem hour. Otherwise, returns the 24-hour equivalent of h in the range [12h, 23h], assuming h represents a post meridiem hour. If h is not in the range [1h, 12h], the value returned is unspecified.

### 27.11 Time zones

#### 27.11.1 In general

27.11 describes an interface for accessing the IANA Time Zone database described in RFC 6557, that interoperates with sys_time and local_time. This interface provides time zone support to both the civil calendar types (27.8) and to user-defined calendars.

#### 27.11.2 Time zone database

**Class tzdb**

```cpp
namespace std::chrono {

struct tzdb {
    string version;
    vector<time_zone> zones;
    vector<link> links;
    vector<leap> leaps;

    const time_zone* locate_zone(string_view tz_name) const;
    const time_zone* current_zone() const;
};
}
```

Each vector in a tzdb object is sorted to enable fast lookup.

```cpp
const time_zone* locate_zone(string_view tz_name) const;
```

Returns: If a time_zone is found for which name() == tz_name, returns a pointer to that time_zone. Otherwise if a link is found for which tz_name == link.name(), then a pointer is returned to the time_zone for which zone.name() == link.target(). [Note: A link specifies an alternative name for a time_zone. — end note]

Throws: If a const time_zone* cannot be found as described in the Returns: clause, throws a runtime_error. [Note: On non-exceptional return, the return value is always a pointer to a valid time_zone. — end note]

```cpp
const time_zone* current_zone() const;
```

Returns: A pointer to the time zone which the computer has set as its local time zone.

**Class tzdb_list**

```cpp
namespace std::chrono {

class tzdb_list {
public:
    tzdb_list(const tzdb_list&) = delete;
    tzdb_list& operator=(const tzdb_list&) = delete;

    // unspecified additional constructors

    class const_iterator;

    const tzdb& front() const noexcept;
};
}
```
const_iterator erase_after(const_iterator p);
const_iterator begin() const noexcept;
const_iterator end() const noexcept;
const_iterator cbegin() const noexcept;
const_iterator cend() const noexcept;
};

1 The tzdb_list database is a singleton; the unique object of type tzdb_list can be accessed via the get_tzdb_list() function. [Note: This access is only needed for those applications that need to have long uptimes and have a need to update the time zone database while running. Other applications can implicitly access the front() of this list via the read-only namespace scope functions get_tzdb(), locate_zone(), and current_zone(). — end note] The tzdb_list object contains a list of tzdb objects.

tzdb_list::const_iterator is a constant iterator which meets the Cpp17ForwardIterator requirements and has a value type of tzdb.

const tzdb& front() const noexcept;

[Editor’s note: reordered synchronization before Returns]

Remarks: Synchronization: This operation is thread-safe with respect to reload_tzdb(). [Note: reload_tzdb() pushes a new tzdb onto the front of this container. — end note]

Returns: A reference to the first tzdb in the container.

const_iterator erase_after(const_iterator p);

Requires: The iterator following p is dereferenceable.
Effects: Erases the tzdb referred to by the iterator following p.
Returns: An iterator pointing to the element following the one that was erased, or end() if no such element exists.
Remarks: Ensures: No pointers, references, or iterators are invalidated except those referring to the erased tzdb. [Note: It is not possible to erase the tzdb referred to by begin(). — end note]

Throws: Nothing.

const_iterator begin() const noexcept;
Returns: An iterator referring to the first tzdb in the container.

const_iterator end() const noexcept;
Returns: An iterator referring to the position one past the last tzdb in the container.

const_iterator cbegin() const noexcept;
Returns: begin().

const_iterator cend() const noexcept;
Returns: end().

27.11.2.3 Time zone database access [time.zone.db.access]
tzdb_list& get_tzdb_list();

Effects: If this is the first access to the time zone database, initializes the database. If this call initializes the database, the resulting database will be a tzdb_list holding a single initialized tzdb.

[Editor’s note: reordered synchronization before Returns]

Remarks: Synchronization: It is safe to call this function from multiple threads at one time.
Returns: A reference to the database.

Throws: runtime_error if for any reason a reference cannot be returned to a valid tzdb_list containing one or more valid tzdb's.
const tzdb& get_tzdb();

Returns: get_tzdb_list().front().

const time_zone* locate_zone(string_view tz_name);

Returns: get_tzdb().locate_zone(tz_name).

[Note: The time zone database will be initialized if this is the first reference to the database. — end note]

const time_zone* current_zone();

Returns: get_tzdb().current_zone().

27.11.2.4 Remote time zone database support

The local time zone database is that supplied by the implementation when the program first accesses the
database, for example via current_zone(). While the program is running, the implementation may choose
to update the time zone database. This update shall not impact the program in any way unless the program
calls the functions in this subclause. This potentially updated time zone database is referred to as the remote
time zone database.

const tzdb& reload_tzdb();

Effects: This function first checks the version of the remote time zone database. If the versions of the
local and remote databases are the same, there are no effects. Otherwise the remote database is pushed
to the front of the tzdb_list accessed by get_tzdb_list().

[Editor’s note: reordered synchronization before Returns]

Remarks: Synchronization: This function is thread-safe with respect to get_tzdb_list().front() and
getzdb_list().erase_after().

[Editor’s note: reordered ensures before Returns]

Remarks: Ensures: No pointers, references, or iterators are invalidated.

Returns: get_tzdb_list().front().

Throws: runtime_error if for any reason a reference cannot be returned to a valid tzdb.

string remote_version();

Returns: The latest remote database version.

[Note: This can be compared with get_tzdb().version to discover if the local and remote databases
are equivalent. — end note]

27.11.3 Exception classes

27.11.3.1 Class nonexistent_local_time

namespace std::chrono {

class nonexistent_local_time : public runtime_error {
public:
    template<class Duration>
    nonexistent_local_time(const local_time<Duration>& tp, const local_info& i);
};
}

nonexistent_local_time is thrown when an attempt is made to convert a non-existent local_time to a
sys_time without specifying choose::earliest or choose::latest.

template<class Duration>
nonexistent_local_time(const local_time<Duration>& tp, const local_info& i);

Requires: i.result == local_info::nonexistent.

Effects: Constructs a nonexistent_local_time by initializing Initializes the base class with a sequence
of char equivalent to that produced by os.str() initialized as shown below:

    os << tp << " is in a gap between\n" << local_seconds{i.first.end.time_since_epoch()} + i.first.offset << ', '

§ 27.11.3.1
<< i.first.abbrev << " and
<< local_seconds(i.second.begin.time_since_epoch()) + i.second.offset << 
<< i.second.abbrev
<< " which are both equivalent to
<< i.first.end << " UTC";

4 [Example:
#include <chrono>
#include <iostream>

int main() {
    using namespace std::chrono;
    try {
        auto zt = zoned_time("America/New_York",
                            local_days{Sunday[2]/March/2016} + 2h + 30min);
    } catch (const nonexistent_local_time& e) {
        std::cout << e.what() << '\n';
    }
}

Produces the output:
2016-03-13 02:30:00 is in a gap between
2016-03-13 02:00:00 EST and
2016-03-13 03:00:00 EDT which are both equivalent to
2016-03-13 07:00:00 UTC

— end example]

27.11.3.2 Class ambiguous_local_time

namespace std::chrono {
    class ambiguous_local_time : public runtime_error {
        public:
            template<class Duration>
            ambiguous_local_time(const local_time<Duration>& tp, const local_info& i);
    }
}

1 ambiguous_local_time is thrown when an attempt is made to convert an ambiguous local_time to a
sys_time without specifying choose::earliest or choose::latest.

template<class Duration>
ambiguous_local_time(const local_time<Duration>& tp, const local_info& i);

2 Requires: i.result == local_info::ambiguous.

3 Effects: Constructs an ambiguous_local_time by initializing the base class with a sequence of
char equivalent to that produced by os.str() initialized as shown below:

    ostringstream os;
    os << tp << " is ambiguous. It could be
    " << i.first.abbrev << " == "
    << tp - i.first.offset << " UTC or"
    " << i.second.abbrev << " == "
    << tp - i.second.offset << " UTC";

4 [Example:
#include <chrono>
#include <iostream>

int main() {
    using namespace std::chrono;
    try {
        auto zt = zoned_time("America/New_York",
                            local_days{Sunday[2]/November/2016} + 1h + 30min);
    } catch (const ambiguous_local_time& e) {
        std::cout << e.what() << '\n';
27.11.4 Information classes

27.11.4.1 Class sys_info

```cpp
namespace std::chrono {
    struct sys_info {
        sys_seconds begin;
        sys_seconds end;
        seconds offset;
        minutes save;
        string abbrev;
    };
}
```

A `sys_info` object can be obtained from the combination of a `time_zone` and either a `sys_time` or `local_time`. It can also be obtained from a `zoned_time`, which is effectively a pair of a `time_zone` and `sys_time`.

[Note: This type provides a low-level interface to time zone information. Typical conversions from `sys_time` to `local_time` will use this class implicitly, not explicitly. — end note]

The `begin` and `end` data members indicate that, for the associated `time_zone` and `time_point`, the `offset` and `abbrev` are in effect in the range `[begin, end)`. This information can be used to efficiently iterate the transitions of a `time_zone`.

The `offset` data member indicates the UTC offset in effect for the associated `time_zone` and `time_point`. The relationship between `local_time` and `sys_time` is:

```cpp
offset = local_time - sys_time
```

The `save` data member is extra information not normally needed for conversion between `local_time` and `sys_time`. If `save` != 0min, this `sys_info` is said to be on “daylight saving” time, and `offset - save` suggests what offset this `time_zone` might use if it were off daylight saving time. However, this information should not be taken as authoritative. The only sure way to get such information is to query the `time_zone` with a `time_point` that returns a `sys_info` where `save` == 0min. There is no guarantee what `time_point` might return such a `sys_info` except that it is guaranteed not to be in the range `[begin, end)` (if `save` != 0min for this `sys_info`).

The `abbrev` data member indicates the current abbreviation used for the associated `time_zone` and `time_point`. Abbreviations are not unique among the `time_zones`, and so one cannot reliably map abbreviations back to a `time_zone` and UTC offset.

```cpp
template<class charT, class traits>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const sys_info& r);
```

Effects: Streams out the `sys_info` object `r` in an unspecified format.

Returns: `os`.

27.11.4.2 Class local_info

```cpp
namespace std::chrono {
    struct local_info {
        static constexpr int unique = 0;
        static constexpr int nonexistent = 1;
        static constexpr int ambiguous = 2;
        int result;
        sys_info first;
        sys_info second;
    };
}
```

§ 27.11.4.2
Note: This type provides a low-level interface to time zone information. Typical conversions from local_time to sys_time will use this class implicitly, not explicitly. — end note

Describes the result of converting a local_time to a sys_time as follows:

(2.1) — When a local_time to sys_time conversion is unique, result == unique, first will be filled out with the correct sys_info, and second will be zero-initialized.

(2.2) — If the conversion stems from a nonexistent local_time then result == nonexistent, first will be filled out with the sys_info that ends just prior to the local_time, and second will be filled out with the sys_info that begins just after the local_time.

(2.3) — If the conversion stems from an ambiguous local_time, then result == ambiguous, first will be filled out with the sys_info that ends just after the local_time, and second will be filled out with the sys_info that starts just before the local_time.

template<class charT, class traits>
  basic_ostream<charT, traits>&
  operator<<(basic_ostream<charT, traits>& os, const local_info& r);

Effects: Streams out the local_info object r in an unspecified format.

Returns: os.

27.11.5 Class time_zone

27.11.5.1 Overview

A time_zone represents all time zone transitions for a specific geographic area. time_zone construction is unspecified, and performed as part of database initialization. [Note: const time_zone objects can be accessed via functions such as locate_zone. — end note]

string_view name() const noexcept;

Returns: The name of the time_zone.

[Example: "America/New_York". — end example]
template<class Duration>
    sys_info get_info(const sys_time<Duration>& st) const;
Returns: A sys_info i for which st is in the range [i.begin, i.end).

template<class Duration>
    local_info get_info(const local_time<Duration>& tp) const;
Returns: A local_info for tp.

template<class Duration>
    sys_time<common_type_t<Duration, seconds>>
    to_sys(const local_time<Duration>& tp) const;
Returns: A sys_time that is at least as fine as seconds, and will be finer if the argument tp has finer precision. This sys_time is the UTC equivalent of tp according to the rules of this time_zone.

Throws: If the conversion from tp to a sys_time is ambiguous, throws ambiguous_local_time. If the tp represents a non-existent time between two UTC time_points, throws nonexistent_local_time.

template<class Duration>
    sys_time<common_type_t<Duration, seconds>>
    to_sys(const local_time<Duration>& tp, choose z) const;
Returns: A sys_time that is at least as fine as seconds, and will be finer if the argument tp has finer precision. This sys_time is the UTC equivalent of tp according to the rules of this time_zone. If the conversion from tp to a sys_time is ambiguous, returns the earlier sys_time if z == choose::earliest, and returns the later sys_time if z == choose::latest. If the tp represents a non-existent time between two UTC time_points, then the two UTC time_points will be the same, and that UTC time_point will be returned.

template<class Duration>
    local_time<common_type_t<Duration, seconds>>
    to_local(const sys_time<Duration>& tp) const;
Returns: The local_time associated with tp and this time_zone.

27.11.5.3 Non-member functions

bool operator==(const time_zone& x, const time_zone& y) noexcept;
Returns: x.name() == y.name().

strong_ordering operator<=>(const time_zone& x, const time_zone& y) noexcept;
Returns: x.name() <=> y.name().

27.11.6 Class template zoned_traits

class std::chrono {
    template<class T> struct zoned_traits {}; 
} zoned_traits provides a means for customizing the behavior of zoned_time<Duration, TimeZonePtr> for the zoned_time default constructor, and constructors taking string_view. A specialization for const time_zone* is provided by the implementation:

namespace std::chrono {
    template<> struct zoned_traits<const time_zone*> {
        static const time_zone* default_zone();
        static const time_zone* locate_zone(string_view name);
    };
}
static const time_zone* default_zone();
Returns: std::chrono::locate_zone("UTC").

static const time_zone* locate_zone(string_view name);
Returns: std::chrono::locate_zone(name).
27.11.7 Class template \texttt{zoned\_time}

27.11.7.1 Overview

namespace std::chrono {
    template<class Duration, class TimeZonePtr = const time\_zone*> 
    class zoned\_time {
        public:
            using duration = common\_type\_t<Duration, seconds>;

        private:
            TimeZonePtr zone_;  // exposition only
            sys\_time<duration> tp_;  // exposition only

            using traits = zoned\_traits<TimeZonePtr>;  // exposition only

        public:
            zoned\_time();
            zoned\_time(const zoned\_time&) = default;
            zoned\_time\& operator=(const zoned\_time&) = default;
            zoned\_time(const sys\_time<Duration>& st);
            explicit zoned\_time(TimeZonePtr z);
            explicit zoned\_time(string\_view name);

            template<class Duration2>
                zoned\_time(const zoned\_time<Duration2, TimeZonePtr>& zt) noexcept;
            zoned\_time(TimeZonePtr z, const sys\_time<Duration>& st);
            zoned\_time(string\_view name, const sys\_time<Duration>& st);
            zoned\_time(TimeZonePtr z, const local\_time<Duration>& tp);
            zoned\_time(string\_view name, const local\_time<Duration>& tp);
            zoned\_time(TimeZonePtr z, const local\_time<Duration>& tp, choose c);
            zoned\_time(string\_view name, const local\_time<Duration>& tp, choose c);

            template<class Duration2, class TimeZonePtr2>
                zoned\_time(TimeZonePtr z, const zoned\_time<Duration2, TimeZonePtr2>& zt);
            template<class Duration2, class TimeZonePtr2>
                zoned\_time(TimeZonePtr z, const zoned\_time<Duration2, TimeZonePtr2>& zt, choose);
            zoned\_time(string\_view name, const zoned\_time<Duration>& zt);
            zoned\_time(string\_view name, const zoned\_time<Duration>& zt, choose);

            zoned\_time\& operator=(const sys\_time<Duration>& st);
            zoned\_time\& operator=(const local\_time<Duration>& ut);

            operator sys\_time<duration>\() const;
            explicit operator local\_time<duration>\() const;

            TimeZonePtr get\_time\_zone\() const;
            local\_time<duration> get\_local\_time\() const;
            sys\_time<duration> get\_sys\_time\() const;
            sys\_info get\_info\() const;
        };

        zoned\_time() -> zoned\_time<seconds>;

        template<class Duration>
            zoned\_time(sys\_time<Duration>)
                -> zoned\_time<common\_type\_t<Duration, seconds>>;

        template<class TimeZonePtr, class Duration>
            zoned\_time(TimeZonePtr, sys\_time<Duration>)
                -> zoned\_time<common\_type\_t<Duration, seconds>, TimeZonePtr>;
    };
};
template<class TimeZonePtr, class Duration>
  zoned_time(TimeZonePtr, local_time<Duration>, choose = choose::earliest)
  -> zoned_time<common_type_t<Duration, seconds>, TimeZonePtr>;

template<class TimeZonePtr, class Duration>
  zoned_time(TimeZonePtr, zoned_time<Duration>, choose = choose::earliest)
  -> zoned_time<common_type_t<Duration, seconds>, TimeZonePtr>;

zoned_time(string_view) -> zoned_time<seconds>;

template<class Duration>
  zoned_time(string_view, sys_time<Duration>)
  -> zoned_time<common_type_t<Duration, seconds>>;

template<class Duration, class TimeZonePtr, class TimeZonePtr2>
  zoned_time(TimeZonePtr, zoned_time<Duration, TimeZonePtr2>, choose = choose::earliest)
  -> zoned_time<Duration, TimeZonePtr>;

zoned_time represents a logical pairing of a time_zone and a time_point with precision Duration. zoned_time<Duration> maintains the invariant that it always refers to a valid time zone and represents a point in time that exists and is not ambiguous in that time zone.

If Duration is not a specialization of chrono::duration, the program is ill-formed.

27.11.7.2 Constructors

zoned_time();

Remarks: Constraints: This constructor does not participate in overload resolution unless traits::default_zone() is a well-formed expression.

Effects: Constructs a zoned_time by initializing zone_ with traits::default_zone() and default constructing tp_.

zoned_time(const sys_time<Duration>& st);

Remarks: Constraints: This constructor does not participate in overload resolution unless traits::default_zone() is a well-formed expression.

Effects: Constructs a zoned_time by initializing zone_ with traits::default_zone() and tp_ with st.

explicit zoned_time(TimeZonePtr z);

Requires: Expects: z refers to a time zone.

Effects: Constructs a zoned_time by initializing zone_ with std::move(z).

explicit zoned_time(string_view name);

Remarks: Constraints: This constructor does not participate in overload resolution unless traits::locate_zone(string_view{}) is a well-formed expression and zoned_time is constructible from the return type of traits::locate_zone(string_view{}).

Effects: Constructs a zoned_time by initializing zone_ with traits::locate_zone(name) and default constructing tp_.

template<class Duration2>
  zoned_time(const zoned_time<Duration2, TimeZonePtr>& y) noexcept;

Remarks: Constraints: Does not participate in overload resolution unless sys_time<Duration> is implicitly convertible to sys_time<Duration2> is_convertible_v<sys_time<Duration2>, sys_time<Duration>> is true.

Effects: Constructs a zoned_time by initializing zone_ with y.zone_ and tp_ with y.tp_.

§ 27.11.7.2 77
zoned_time(TimeZonePtr z, const sys_time<Duration>& st);

**Requires:** z refers to a time zone.

**Effects:** Constructs a `zoned_time` by initializing `zone_` with `std::move(z)` and `tp_` with `st`.

zoned_time(string_view name, const sys_time<Duration>& st);

**Remarks:**

**Constraints:** This constructor does not participate in overload resolution unless `zoned_time` is constructible from the return type of `traits::locate_zone(name)` and `st`.

**Effects:** Equivalent to construction with `{traits::locate_zone(name), st}`.

zoned_time(TimeZonePtr z, const local_time<Duration>& tp);

**Requires:** z refers to a time zone.

**Effects:** Constructs a `zoned_time` by initializing `zone_` with `std::move(z)` and `tp_` with `zone_->to_sys(tp)`.

zoned_time(string_view name, const local_time<Duration>& tp);

**Remarks:**

**Constraints:** This constructor does not participate in overload resolution unless `decltype(declval<TimeZonePtr&>()->to_sys(local_time<Duration>())` is convertible to `sys_time<duration>`.

**Effects:** Equivalent to construction with `{traits::locate_zone(name), tp}`.

zoned_time(TimeZonePtr z, const local_time<Duration>& tp, choose c);

**Requires:** z refers to a time zone.

**Remarks:**

**Constraints:** This constructor does not participate in overload resolution unless `decltype(declval<TimeZonePtr&>()->to_sys(local_time<Duration>(), choose::earliest))` is convertible to `sys_time<duration>`.

**Effects:** Constructs a `zoned_time` by initializing `zone_` with `std::move(z)` and `tp_` with `zone_->to_sys(tp, c)`.

zoned_time(string_view name, const local_time<Duration>& tp, choose c);

**Remarks:**

**Constraints:** This constructor does not participate in overload resolution unless `zoned_time` is constructible from the return type of `traits::locate_zone(name)`, `local_time<Duration>`, and `choose`.

**Effects:** Equivalent to construction with `{traits::locate_zone(name), tp, c}`.

template<class Duration2, class TimeZonePtr2>

zoned_time(TimeZonePtr z, const zoned_time<Duration2, TimeZonePtr2>& y);

**Remarks:**

**Constraints:** Does not participate in overload resolution unless `sys_time<Duration2>` is implicitly convertible to `sys_time<Duration>`.

**Effects:** Constructs a `zoned_time` by initializing `zone_` with `std::move(z)` and `tp_` with `y.tp_`.

template<class Duration2, class TimeZonePtr2>

zoned_time(TimeZonePtr z, const zoned_time<Duration2, TimeZonePtr2>& y, choose);

**Remarks:**

**Constraints:** Does not participate in overload resolution unless `sys_time<Duration2>` is implicitly convertible to `sys_time<Duration>`.

**Effects:** Equivalent to construction with `{z, y}`.

[Note: The choose parameter has no effect. — end note]
zoned_time(string_view name, const zoned_time<Duration>& y);

Remarks: Constraints: This constructor does not participate in overload resolution unless zoned_time is constructible from the return type of traits::locate_zone(name) and zoned_time.
Effects: Equivalent to construction with {traits::locate_zone(name), y}.

zoned_time(string_view name, const zoned_time<Duration>& y, choose c);
Remarks: Constraints: This constructor does not participate in overload resolution unless zoned_time is constructible from the return type of traits::locate_zone(name), zoned_time, and choose.
Effects: Equivalent to construction with {traits::locate_zone(name), y, c}.

[Note: The choose parameter has no effect. — end note]

27.11.7.3 Member functions

zoned_time& operator=(const sys_time<Duration>& st);

Effects: After assignment, get_sys_time() == st. This assignment has no effect on the return value of get_time_zone().
Returns: *this.

zoned_time& operator=(const local_time<Duration>& lt);
Effects: After assignment, get_local_time() == lt. This assignment has no effect on the return value of get_time_zone().
Returns: *this.

operator sys_time<duration>() const;
Returns: get_sys_time().

explicit operator local_time<duration>() const;
Returns: get_local_time().

TimeZonePtr get_time_zone() const;
Returns: zone_.

local_time<duration> get_local_time() const;
Returns: zone_->to_local(tp_).

sys_time<duration> get_sys_time() const;
Returns: tp_.

sys_info get_info() const;
Returns: zone_->get_info(tp_).

27.11.7.4 Non-member functions

template<class Duration1, class Duration2, class TimeZonePtr>
bool operator==(const zoned_time<Duration1, TimeZonePtr>& x,
const zoned_time<Duration2, TimeZonePtr>& y);
Returns: x.zone_ == y.zone_ && x.tp_ == y.tp_.

template<class charT, class traits, class Duration, class TimeZonePtr>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os,
const zoned_time<Duration, TimeZonePtr>& t);
Effects: Streams the value returned from t.get_local_time() to os using the format "%F %T %Z".
Returns: os.
27.11.8 Class `leap` [time.zone.leap]

27.11.8.1 Overview [time.zone.leap.overview]

```cpp
namespace std::chrono {
  class leap {
    public:
      leap(const leap&) = default;
      leap& operator=(const leap&) = default;

      // unspecified additional constructors

      constexpr sys_seconds date() const noexcept;
  };
}
```

1. Objects of type `leap` representing the date of the leap second insertions are constructed and stored in the time zone database when initialized.

2. [Example:

```cpp
for (auto& l : get_tzdb().leaps)
  if (l <= 2018y/March/17d)
    cout << l.date() << 'n';
```

Produces the output:

1972-07-01 00:00:00
1973-01-01 00:00:00
1974-01-01 00:00:00
1975-01-01 00:00:00
1976-01-01 00:00:00
1977-01-01 00:00:00
1978-01-01 00:00:00
1979-01-01 00:00:00
1980-01-01 00:00:00
1981-07-01 00:00:00
1982-07-01 00:00:00
1983-07-01 00:00:00
1984-07-01 00:00:00
1985-07-01 00:00:00
1986-01-01 00:00:00
1987-01-01 00:00:00
1988-01-01 00:00:00
1989-01-01 00:00:00
1990-01-01 00:00:00
1991-01-01 00:00:00
1992-07-01 00:00:00
1993-07-01 00:00:00
1994-07-01 00:00:00
1995-01-01 00:00:00
1996-01-01 00:00:00
1997-01-01 00:00:00
1998-01-01 00:00:00
1999-01-01 00:00:00
2000-01-01 00:00:00
2001-01-01 00:00:00
2002-01-01 00:00:00
2003-01-01 00:00:00
2004-01-01 00:00:00
2005-01-01 00:00:00
2006-01-01 00:00:00
2007-01-01 00:00:00
2008-01-01 00:00:00
2009-01-01 00:00:00
2010-01-01 00:00:00
2011-01-01 00:00:00
2012-01-01 00:00:00
2013-01-01 00:00:00
2014-01-01 00:00:00
2015-01-01 00:00:00
2016-01-01 00:00:00
2017-01-01 00:00:00

—end example]

27.11.8.2 Member functions [time.zone.leap.members]

```cpp
constexpr sys_seconds date() const noexcept;
```

1. **Returns**: The date and time at which the leap second was inserted.

27.11.8.3 Non-member functions [time.zone.leap.nonmembers]

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

1. **Returns**: `x.date() == y.date()`. 
constexpr strong_ordering operator<=>(const leap& x, const leap& y) noexcept;

   Returns: x.date() <=> y.date().

template<class Duration>
constexpr bool operator==(const leap& x, const sys_time<Duration>& y) noexcept;

   Returns: x.date() == y.

template<class Duration>
constexpr bool operator<(const leap& x, const sys_time<Duration>& y) noexcept;

   Returns: x < y.date().

template<class Duration>
constexpr bool operator<(const sys_time<Duration>& x, const leap& y) noexcept;

   Returns: x < y.

template<class Duration>
constexpr bool operator<=(const leap& x, const sys_time<Duration>& y) noexcept;

   Returns: !(y < x).

template<class Duration>
constexpr bool operator<=(const sys_time<Duration>& x, const leap& y) noexcept;

   Returns: !(y < x).

template<class Duration>
constexpr bool operator>=(const leap& x, const sys_time<Duration>& y) noexcept;

   Returns: !(x < y).

template<class Duration>
constexpr bool operator>=(const sys_time<Duration>& x, const leap& y) noexcept;

   Returns: !(x < y).

template<three_way_comparable_with<sys_seconds> Duration>
constexpr auto operator<=>(const leap& x, const sys_time<Duration>& y) noexcept;

   Returns: x.date() <=> y.

27.11.9 Class link

27.11.9.1 Overview

   namespace std::chrono {
   class link {
   public:
      link(link&&) = default;
      link& operator=(link&&) = default;

      // unspecified additional constructors

      string_view name() const noexcept;
      string_view target() const noexcept;
   };
   }

   A link specifies an alternative name for a time_zone. links are constructed when the time zone database is initialized.
27.11.9.2 Member functions

\[\text{string\_view name()} \text{ const noexcept;}\]

1

_**Returns:**_ The alternative name for the time zone.

\[\text{string\_view target()} \text{ const noexcept;}\]

2

_**Returns:**_ The name of the `time\_zone` for which this `link` provides an alternative name.

27.11.9.3 Non-member functions

\[\text{bool operator==}(\text{const link & x, const link & y}) \text{ noexcept;}\]

1

_**Returns:**_ `x.name()` == `y.name()`.

\[\text{strong\_ordering operator\<\>}(\text{const link & x, const link & y}) \text{ noexcept;}\]

2

_**Returns:**_ `x.name()` <= `y.name()`.

27.12 Formatting

1

Each `formatter` specialization in the chrono library (27.2) meets the `Formatter` requirements (??). The `parse` member functions of these formatters interpret the format specification as a `chrono\_format\_spec` according to the following syntax:

\[
\text{chrono\_format\_spec}:
\text{fill-and-align}_\text{opt} \text{ width}_\text{opt} \text{ precision}_\text{opt} \text{ chrono\_specs}_\text{opt}
\]

\[
\text{chrono\_specs}:
\text{conversion\_spec} \text{ chrono\_specs} \text{ conversion\_spec} \\
\text{chrono\_specs} \text{ literal\_char}
\]

\[
\text{literal\_char}:
\text{any character other than \{ or \}
\]

\[
\text{conversion\_spec}:
\% \text{ modifier}_\text{opt} \text{ type}
\]

\[
\text{modifier}: \text{one of}
\]

\[
\text{E O}
\]

\[
\text{type}: \text{one of}
\]

\[
\text{a A b B c C d D e E f G g h H i j m M n p r R s T u U v w w x X y Y z Z %}
\]

The productions `fill-and-align`, `width`, and `precision` are described in ??, Giving a `precision` specification in the `chrono\_format\_spec` is valid only for `std::chrono::duration` types where the representation type Rep is a floating-point type. For all other Rep types, an exception of type `format\_error` is thrown if the `chrono\_format\_spec` contains a `precision` specification. All ordinary multibyte characters represented by `literal\_char` are copied unchanged to the output.

2

Each conversion specifier `conversion\_spec` is replaced by appropriate characters as described in Table 88. Some of the conversion specifiers depend only on the locale that is passed to the formatting function if the latter takes one, or the global locale otherwise. If the formatted object does not contain the information the conversion specifier refers to, an exception of type `format\_error` is thrown.

3

Unless explicitly requested, the result of formatting a chrono type does not contain time zone abbreviation and time zone offset information. If the information is available, the conversion specifiers `%Z` and `%z` will format this information (respectively). [Note: If the information is not available and a `%Z` or `%z` conversion specifier appears in the `chrono\_format\_spec`, an exception of type `format\_error` is thrown, as described above. — end note]

Table 88: Meaning of conversion specifiers

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%a</code></td>
<td>The locale’s abbreviated weekday name. If the value does not contain a valid weekday, an exception of type <code>format\_error</code> is thrown.</td>
</tr>
<tr>
<td><code>%A</code></td>
<td>The locale’s full weekday name. If the value does not contain a valid weekday, an exception of type <code>format\_error</code> is thrown.</td>
</tr>
</tbody>
</table>
### Table 88: Meaning of conversion specifiers (continued)

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%b</code></td>
<td>The locale’s abbreviated month name. If the value does not contain a valid month, an exception of type <code>format_error</code> is thrown.</td>
</tr>
<tr>
<td><code>%B</code></td>
<td>The locale’s full month name. If the value does not contain a valid month, an exception of type <code>format_error</code> is thrown.</td>
</tr>
<tr>
<td><code>%c</code></td>
<td>The locale’s date and time representation. The modified command <code>%Ec</code> produces the locale’s alternate date and time representation.</td>
</tr>
<tr>
<td><code>%C</code></td>
<td>The year divided by 100 using floored division. If the result is a single decimal digit, it is prefixed with 0. The modified command <code>%EC</code> produces the locale’s alternative representation of the century.</td>
</tr>
<tr>
<td><code>%d</code></td>
<td>The day of month as a decimal number. If the result is a single decimal digit, it is prefixed with 0. The modified command <code>%Od</code> produces the locale’s alternative representation.</td>
</tr>
<tr>
<td><code>%D</code></td>
<td>Equivalent to <code>%m/%d/%y</code>.</td>
</tr>
<tr>
<td><code>%e</code></td>
<td>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 <code>%Oe</code> produces the locale’s alternative representation.</td>
</tr>
<tr>
<td><code>%F</code></td>
<td>Equivalent to <code>%Y-%m-%d</code>.</td>
</tr>
<tr>
<td><code>%g</code></td>
<td>The last two decimal digits of the ISO week-based year. If the result is a single digit it is prefixed by 0.</td>
</tr>
<tr>
<td><code>%G</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>%h</code></td>
<td>Equivalent to <code>%b</code>.</td>
</tr>
<tr>
<td><code>%H</code></td>
<td>The hour (24-hour clock) as a decimal number. If the result is a single digit, it is prefixed with 0. The modified command <code>%OH</code> produces the locale’s alternative representation.</td>
</tr>
<tr>
<td><code>%I</code></td>
<td>The hour (12-hour clock) as a decimal number. If the result is a single digit, it is prefixed with 0. The modified command <code>%OI</code> produces the locale’s alternative representation.</td>
</tr>
<tr>
<td><code>%j</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>%m</code></td>
<td>The month as a decimal number. Jan is 01. If the result is a single digit, it is prefixed with 0. The modified command <code>%Om</code> produces the locale’s alternative representation.</td>
</tr>
<tr>
<td><code>%M</code></td>
<td>The minute as a decimal number. If the result is a single digit, it is prefixed with 0. The modified command <code>%OM</code> produces the locale’s alternative representation.</td>
</tr>
<tr>
<td><code>%n</code></td>
<td>A new-line character.</td>
</tr>
<tr>
<td><code>%p</code></td>
<td>The locale’s equivalent of the AM/PM designations associated with a 12-hour clock.</td>
</tr>
<tr>
<td><code>%q</code></td>
<td>The duration’s unit suffix as specified in 27.5.10.</td>
</tr>
<tr>
<td><code>%Q</code></td>
<td>The duration’s numeric value (as if extracted via <code>.count()</code>).</td>
</tr>
<tr>
<td><code>%r</code></td>
<td>The locale’s 12-hour clock time.</td>
</tr>
<tr>
<td><code>%R</code></td>
<td>Equivalent to <code>%H:%M</code>.</td>
</tr>
<tr>
<td><code>%s</code></td>
<td>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 cannot 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 cannot be made within 18 fractional digits). The character for the decimal point is localized according to the locale. The modified command <code>%Os</code> produces the locale’s alternative representation.</td>
</tr>
<tr>
<td><code>%t</code></td>
<td>A horizontal-tab character.</td>
</tr>
<tr>
<td><code>%T</code></td>
<td>Equivalent to <code>%H:%M:%S</code>.</td>
</tr>
<tr>
<td><code>%u</code></td>
<td>The ISO weekday as a decimal number (1-7), where Monday is 1. The modified command <code>%Ou</code> produces the locale’s alternative representation.</td>
</tr>
<tr>
<td><code>%U</code></td>
<td>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 <code>%OU</code> produces the locale’s alternative representation.</td>
</tr>
</tbody>
</table>
Table 88: Meaning of conversion specifiers (continued)

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>%v</td>
<td>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.</td>
</tr>
<tr>
<td>%w</td>
<td>The weekday as a decimal number (0-6), where Sunday is 0. The modified command %Ow produces the locale’s alternative representation.</td>
</tr>
<tr>
<td>%W</td>
<td>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.</td>
</tr>
<tr>
<td>%x</td>
<td>The locale’s date representation. The modified command %Ex produces the locale’s alternate date representation.</td>
</tr>
<tr>
<td>%X</td>
<td>The locale’s time representation. The modified command %EX produces the locale’s alternate time representation.</td>
</tr>
<tr>
<td>%y</td>
<td>The last two decimal digits of the year. If the result is a single digit it is prefixed by 0.</td>
</tr>
<tr>
<td>%Y</td>
<td>The year as a decimal number. If the result is less than four digits it is left-padded with 0 to four digits.</td>
</tr>
<tr>
<td>%z</td>
<td>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, an exception of type format_error is thrown.</td>
</tr>
<tr>
<td>%Z</td>
<td>The time zone abbreviation. If the time zone abbreviation is not available, an exception of type format_error is thrown.</td>
</tr>
<tr>
<td>%%</td>
<td>A % character.</td>
</tr>
</tbody>
</table>

4 If the chrono-specs is omitted, the chrono object is formatted as if by streaming it to std::ostringstream os and copying os.str() through the output iterator of the context with additional padding and adjustments as specified by the format specifiers. [Example:
string s = format("{{:=>8}}", 42ms); // value of s is "====42ms"
— end example]

5 template<class Duration, class charT>
struct formatter<chrono::sys_time<Duration>, charT>;

Remarks: If %Z is used, it is replaced with STATICALLY-WIDEN<charT>("UTC"). If %z (or a modified variant of %z) is used, an offset of 0min is formatted.

6 template<class Duration, class charT>
struct formatter<chrono::utc_time<Duration>, charT>;

Remarks: If %Z is used, it is replaced with STATICALLY-WIDEN<charT>("UTC"). If %z (or a modified variant of %z) is used, an offset of 0min is formatted. If the argument represents a time during a leap second insertion, and if a seconds field is formatted, the integral portion of that format is STATICALLY-WIDEN<charT>("60").

7 template<class Duration, class charT>
struct formatter<chrono::tai_time<Duration>, charT>;

Remarks: If %Z is used, it is replaced with STATICALLY-WIDEN<charT>("TAI"). If %z (or a modified variant of %z) is used, an offset of 0min is formatted. The date and time formatted are equivalent to those formatted by a sys_time initialized with
 sys_time<Duration>(tp.time_since_epoch()) -
(sys_days{1970y/January/1} - sys_days{1958y/January/1})

8 template<class Duration, class charT>
struct formatter<chrono::gps_time<Duration>, charT>;

Remarks: If %Z is used, it is replaced with STATICALLY-WIDEN<charT>("GPS"). If %z (or a modified variant of %z) is used, an offset of 0min is formatted. The date and time formatted are equivalent to those formatted by a sys_time initialized with

§ 27.12

84
sys_time<Duration>{tp.time_since_epoch()} +
(sys_days{1980y/January/Sunday[1]} - sys_days{1970y/January/1})

template<class Duration, class charT>
struct formatter<chrono::file_time<Duration>, charT>;

Remarks: If %Z is used, it is replaced with \texttt{STATICALLY-WIDEN<charT>("UTC")}. If %z (or a modified variant of %Z) is used, an offset of 0min is formatted. The date and time formatted are equivalent to those formatted by a \texttt{sys_time} initialized with \texttt{clock_cast<system_clock>(t)} or by a \texttt{utc_time} initialized with \texttt{clock_cast<utc_clock>(t)} where t is the first argument to \texttt{format}.

template<class Duration, class charT>
struct formatter<chrono::local_time<Duration>, charT>;

Remarks: If %Z, %z, or a modified version of %Z is used, an exception of type \texttt{format_error} is thrown.

template<class Duration> struct local-time-format-t {
  // exposition only
  local_time<Duration> time;
  // exposition only
  const string* abbrev;
  // exposition only
  const seconds* offset_sec;
};

template<class Duration>
local-time-format-t<Duration>
local_time_format(local_time<Duration> time, const string* abbrev = nullptr,
                  const seconds* offset_sec = nullptr);

Returns: \{time, abbrev, offset_sec\}.

template<class Duration, class charT>
struct formatter<chrono::zoned_time<Duration, TimeZonePtr>, charT>:
  formatter<chrono::local-time-format-t<Duration>, charT> {
  template<class FormatContext>
  typename FormatContext::iterator
  format(const chrono::zoned_time<Duration, TimeZonePtr>& tp, FormatContext& ctx);
};

27.13 Parsing [time.parse]

1 Each \texttt{parse} overload specified in this subclause calls \texttt{from_stream} unqualified, so as to enable argument dependent lookup (??).

template<class charT, class traits, class Alloc, class Parsable>
unspecifed
parse(const basic_string<charT, traits, Alloc>& fmt, Parsable& tp);

Remarks: Constraints: This function shall not participate in overload resolution unless
\texttt{from_stream(declval<charT, traits>& fmt, c_str(), tp)}
is a valid expression.

**Returns:** A manipulator that, when extracted from a `basic_istream<charT, traits>` is, calls `from_stream(is, fmt.c_str(), tp).

template<class charT, class traits, class Alloc, class Parsable>
unspecified
parse(const basic_string<charT, traits, Alloc>& fmt, Parsable& tp, basic_string<charT, traits, Alloc>&& abbrev);

**Remarks:** Constraints: This function shall not participate in overload resolution unless
`from_stream(declval<basic_istream<charT, traits>&>(), fmt.c_str(), tp, addressof(abbrev))`

is a valid expression.

**Returns:** A manipulator that, when extracted from a `basic_istream<charT, traits>` is, calls `from_stream(is, fmt.c_str(), tp, addressof(abbrev)).`

template<class charT, class traits, class Alloc, class Parsable>
unspecified
parse(const basic_string<charT, traits, Alloc>& fmt, Parsable& tp, minutes& offset);

**Remarks:** Constraints: This function shall not participate in overload resolution unless
`from_stream(declval<basic_istream<charT, traits>&>(), fmt.c_str(), tp, nullptr, &offset)`

is a valid expression.

**Returns:** A manipulator that, when extracted from a `basic_istream<charT, traits>` is, calls `from_stream(is, fmt.c_str(), tp, nullptr, &offset).`

template<class charT, class traits, class Alloc, class Parsable>
unspecified
parse(const basic_string<charT, traits, Alloc>& fmt, Parsable& tp, basic_string<charT, traits, Alloc>&& abbrev, minutes& offset);

**Remarks:** Constraints: This function shall not participate in overload resolution unless
`from_stream(declval<basic_istream<charT, traits>&>(), fmt.c_str(), tp, addressof(abbrev), &offset)`

is a valid expression.

**Returns:** A manipulator that, when extracted from a `basic_istream<charT, traits>` is, calls `from_stream(is, fmt.c_str(), tp, addressof(abbrev), &offset).`

All `from_stream` overloads behave as unformatted input functions, except that they have an unspecified
effect on the value returned by subsequent calls to `basic_istream::gcount()`. 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 types according to Table 89. Some flags can be modified by a width parameter given as a positive
decimal integer called out as N below which governs how many characters are parsed from the stream in
interpreting the flag. All characters in the format string that are not represented in Table 89, 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.

If the `from_stream` overload fails to parse everything specified by the format string, or if insufficient
information is parsed to specify a complete duration, time point, or calendrical data structure, `setstate(ios_base::failbit)` is called on the `basic_istream`.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Parsed value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%a</td>
<td>The locale's full or abbreviated case-insensitive weekday name.</td>
</tr>
<tr>
<td>%A</td>
<td>Equivalent to %a.</td>
</tr>
<tr>
<td>%b</td>
<td>The locale's full or abbreviated case-insensitive month name.</td>
</tr>
<tr>
<td>%B</td>
<td>Equivalent to %b.</td>
</tr>
</tbody>
</table>
Table 89: Meaning of parse flags (continued)

<table>
<thead>
<tr>
<th>Flag</th>
<th>Parsed value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%c</td>
<td>The locale’s date and time representation. The modified command %Ec interprets the locale’s alternate date and time representation.</td>
</tr>
<tr>
<td>%c</td>
<td>The century as a decimal number. The modified command %NC specifies the maximum number of characters to read. If N is 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.</td>
</tr>
<tr>
<td>%C</td>
<td>The day of the month as a decimal number. The modified command %Nd specifies the maximum number of characters to read. If N is not specified, the default is 2. Leading zeroes are permitted but not required. The modified command %Ed interprets the locale’s alternative representation of the day of the month.</td>
</tr>
<tr>
<td>%D</td>
<td>Equivalent to %m/%d/%y.</td>
</tr>
<tr>
<td>%d</td>
<td>Equivalent to %d and can be modified like %d.</td>
</tr>
<tr>
<td>%F</td>
<td>Equivalent to %Y-%m-%d. If modified with a width N, the width is applied to only %Y.</td>
</tr>
<tr>
<td>%g</td>
<td>The last two decimal digits of the ISO week-based year. The modified command %Ng specifies the maximum number of characters to read. If N is not specified, the default is 2. Leading zeroes are permitted but not required.</td>
</tr>
<tr>
<td>%G</td>
<td>The ISO week-based year as a decimal number. The modified command %Ng specifies the maximum number of characters to read. If N is not specified, the default is 4. Leading zeroes are permitted but not required.</td>
</tr>
<tr>
<td>%h</td>
<td>Equivalent to %b.</td>
</tr>
<tr>
<td>%H</td>
<td>The hour (24-hour clock) as a decimal number. The modified command %NH specifies the maximum number of characters to read. If N is not specified, the default is 2. Leading zeroes are permitted but not required. The modified command %OH interprets the locale’s alternative representation.</td>
</tr>
<tr>
<td>%I</td>
<td>The hour (12-hour clock) as a decimal number. The modified command %NI specifies the maximum number of characters to read. If N is not specified, the default is 2. Leading zeroes are permitted but not required.</td>
</tr>
<tr>
<td>%j</td>
<td>The day of the year as a decimal number. Jan 1 is 1. The modified command %Nj specifies the maximum number of characters to read. If N is not specified, the default is 3. Leading zeroes are permitted but not required.</td>
</tr>
<tr>
<td>%m</td>
<td>The month as a decimal number. Jan is 1. The modified command %Nm specifies the maximum number of characters to read. If N is not specified, the default is 2. Leading zeroes are permitted but not required. The modified command %Om interprets the locale’s alternative representation.</td>
</tr>
<tr>
<td>%M</td>
<td>The minutes as a decimal number. The modified command %Nm specifies the maximum number of characters to read. If N is not specified, the default is 2. Leading zeroes are permitted but not required. The modified command %OM interprets the locale’s alternative representation.</td>
</tr>
<tr>
<td>%n</td>
<td>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, &quot;%n &quot; matches one or more white space characters, and &quot;%n%t%t&quot; matches one to three white space characters. — end note]</td>
</tr>
<tr>
<td>%p</td>
<td>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.</td>
</tr>
<tr>
<td>%r</td>
<td>The locale’s 12-hour clock time.</td>
</tr>
<tr>
<td>%R</td>
<td>Equivalent to %H:%M.</td>
</tr>
<tr>
<td>%S</td>
<td>The seconds as a decimal number. The modified command %NS specifies the maximum number of characters to read. If N is 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.</td>
</tr>
<tr>
<td>%s</td>
<td>Matches zero or one white space characters.</td>
</tr>
<tr>
<td>%t</td>
<td>Equivalent to %H:%M:%S.</td>
</tr>
</tbody>
</table>
Table 89: Meaning of parse flags (continued)

<table>
<thead>
<tr>
<th>Flag</th>
<th>Parsed value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%u</td>
<td>The ISO weekday as a decimal number (1-7), where Monday is 1. The modified command %Nu specifies the maximum number of characters to read. If N is not specified, the default is 1. Leading zeroes are permitted but not required. The modified command %0u interprets the locale’s alternative representation.</td>
</tr>
<tr>
<td>%U</td>
<td>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 specifies the maximum number of characters to read. If N is not specified, the default is 2. Leading zeroes are permitted but not required.</td>
</tr>
<tr>
<td>%V</td>
<td>The ISO week-based week number as a decimal number. The modified command %NV specifies the maximum number of characters to read. If N is not specified, the default is 2. Leading zeroes are permitted but not required.</td>
</tr>
<tr>
<td>%w</td>
<td>The weekday as a decimal number (0-6), where Sunday is 0. The modified command %Nw specifies the maximum number of characters to read. If N is not specified, the default is 1. Leading zeroes are permitted but not required. The modified command %0w interprets the locale’s alternative representation.</td>
</tr>
<tr>
<td>%W</td>
<td>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 specifies the maximum number of characters to read. If N is not specified, the default is 2. Leading zeroes are permitted but not required.</td>
</tr>
<tr>
<td>%x</td>
<td>The locale’s date representation. The modified command %Ex interprets the locale’s alternate date representation.</td>
</tr>
<tr>
<td>%X</td>
<td>The locale’s time representation. The modified command %EX interprets the locale’s alternate time representation.</td>
</tr>
<tr>
<td>%y</td>
<td>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 to 1999, and values in the range [00, 68] are presumed to refer to the years 2000 to 2068. The modified command %Ny specifies the maximum number of characters to read. If N is not specified, the default is 2. Leading zeroes are permitted but not required. The modified commands %Ey and %0y interpret the locale’s alternative representation.</td>
</tr>
<tr>
<td>%Y</td>
<td>The year as a decimal number. The modified command %NY specifies the maximum number of characters to read. If N is not specified, the default is 4. Leading zeroes are permitted but not required. The modified command %EY interprets the locale’s alternative representation.</td>
</tr>
<tr>
<td>%z</td>
<td>The offset from UTC in the format [+</td>
</tr>
<tr>
<td>%Z</td>
<td>The time zone abbreviation or name. A single word is parsed. This word can only contain characters from the basic source character set (??) that are alphanumeric, or one of ‘_’, ‘/’, ‘-’, or ‘+’.</td>
</tr>
<tr>
<td>%%</td>
<td>A % character is extracted.</td>
</tr>
</tbody>
</table>
The contents of the header `<ctime>` are the same as the C standard library header `<time.h>`.

The functions `asctime`, `ctime`, `gmtime`, and `localtime` are not required to avoid data races (??). See also: ISO C 7.27

---

259) `strftime` supports the C conversion specifiers C, D, E, F, G, h, x, a, t, U, v, and z, and the modifiers E and D.