Document Number: P1686R1
Date: 2019-10-07
Reply to: Jeff Garland
CrystalClear Software
jeff@crystalclearsoftware.com

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> is Cpp17LessThanComparable" 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 27.8.6.2
- time.cal.wdidx.members 27.8.7.2
- time.cal.wdlast.members 27.8.8.2
- time.cal.md.overview 27.8.9.1
- time.cal.mdlast 27.8.10
- time.cal.mwd.members 27.8.11.2
- time.cal.mwdlast.members 27.8.12.2
- time.cal.ym.overview 27.8.13.1
- time.cal.ym.members 27.8.13.2
- time.cal.ymd.overview 27.8.14.1
- time.cal.ymd.members 27.8.14.2
- time.cal.ymwd.members 27.8.16.2
- time.cal.ymdlast.overview 27.8.15.1
- time.cal.ymdlast.members 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 [time]

27.1 General [time.general]

1 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 [tab:time.summary]

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

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

27.2 Header `<chrono>` synopsis [time.syn]

```
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<duration<Rep1, Period1>, duration<Rep2, Period2>>
        operator+(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
```

```
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 Rep2, class Period>
  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<Rep1, Rep2>
    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 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);
template<class Rep1, class Period1, class Rep2, class Period2>
  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>
  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, three_way_comparable_with<Duration1> 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, class Duration>
  basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const sys_time<Duration>& tp);

template<class charT, class traits>
  basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const sys_days& dp);

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,
                sys_time<Duration>& tp,
                basic_string<charT, traits, Alloc>* abbrev = nullptr,
                minutes* offset = nullptr);

// 27.7.2, class utc_clock
class utc_clock;

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

template<class charT, class traits, class Duration>
  basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const utc_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,
                utc_time<Duration>& tp,
                basic_string<charT, traits, Alloc>* abbrev = nullptr,
                minutes* offset = nullptr);

struct leap_second_info;

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

// 27.7.3, class tai_clock
class tai_clock;

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

template<class charT, class traits, class Duration>
  basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const tai_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,
                tai_time<Duration>& tp,
                basic_string<charT, traits, Alloc>* abbrev = nullptr,
```

```
                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;

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

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

template<class charT, class traits>
  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;

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

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

template<class charT, class traits>
  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;

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

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

template<class charT, class traits>
  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 weekday operator-(const weekday& x, const days&    y) noexcept;
constexpr days    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;

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

template<class charT, class traits>
  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,
```

```
                basic_string<charT, traits, Alloc>* abbrev = nullptr,
                minutes* offset = nullptr);

  // 27.8.15, class year_month_day_last
  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& ymwdi);

  // 27.8.17, class year_month_weekday_last
  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& ymwdl, const months& dm) noexcept;
  constexpr year_month_weekday_last
    operator+(const months& dm, const year_month_weekday_last& ymwdl) noexcept;
  constexpr year_month_weekday_last
    operator+(const year_month_weekday_last& ymwdl, const years& dy) noexcept;
```

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

template<class charT, class traits>
  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 year& y, const month_day& md) noexcept;
constexpr year_month_day
  operator/(int y, const month_day& md) noexcept;
constexpr year_month_day
  operator/(const month_day& md, const year& y) noexcept;
constexpr year_month_day
  operator/(const month_day& md, int y) noexcept;
constexpr year_month_day_last
  operator/(const year_month& ym, last_spec) noexcept;
```

```
constexpr year_month_day_last
  operator/(const year& y, const month_day_last& mdl) noexcept;
constexpr year_month_day_last
  operator/(int y, const month_day_last& mdl) noexcept;
constexpr year_month_day_last
  operator/(const month_day_last& mdl, const year& y) noexcept;
constexpr year_month_day_last
  operator/(const month_day_last& mdl, int y) noexcept;
constexpr year_month_weekday
  operator/(const year_month& ym, const weekday_indexed& wdi) noexcept;
constexpr year_month_weekday
  operator/(const year& y, const month_weekday& mwd) noexcept;
constexpr year_month_weekday
  operator/(int y, const month_weekday& mwd) noexcept;
constexpr year_month_weekday
  operator/(const month_weekday& mwd, const year& y) noexcept;
constexpr year_month_weekday
  operator/(const month_weekday& mwd, int y) noexcept;
constexpr year_month_weekday_last
  operator/(const year_month& ym, const weekday_last& wdl) noexcept;
constexpr year_month_weekday_last
  operator/(const year& y, const month_weekday_last& mwdl) noexcept;
constexpr year_month_weekday_last
  operator/(int y, const month_weekday_last& mwdl) noexcept;
constexpr year_month_weekday_last
  operator/(const month_weekday_last& mwdl, const year& y) noexcept;
constexpr year_month_weekday_last
  operator/(const month_weekday_last& mwdl, int y) noexcept;

// 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<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-t;        // exposition only
  template<class Duration>
    local-time-format-t<Duration>
      local_time_format(local_time<Duration> time, const string* abbrev = nullptr,
                        const seconds* offset_sec = nullptr);
}

template<class Rep, class Period, class charT>
  struct formatter<chrono::duration<Rep, Period>, charT>;
template<class Duration, class charT>
  struct formatter<chrono::sys_time<Duration>, charT>;
template<class Duration, class charT>
  struct formatter<chrono::utc_time<Duration>, charT>;
template<class Duration, class charT>
  struct formatter<chrono::tai_time<Duration>, charT>;
template<class Duration, class charT>
  struct formatter<chrono::gps_time<Duration>, charT>;
template<class Duration, class charT>
  struct formatter<chrono::file_time<Duration>, charT>;
template<class Duration, class charT>
  struct formatter<chrono::local_time<Duration>, charT>;
template<class Duration, class charT>
  struct formatter<chrono::local-time-format-t<Duration>, charT>;
template<class charT> struct formatter<chrono::day, charT>;
template<class charT> struct formatter<chrono::month, charT>;
template<class charT> struct formatter<chrono::year, charT>;
template<class charT> struct formatter<chrono::weekday, charT>;
template<class charT> struct formatter<chrono::weekday_indexed, charT>;
template<class charT> struct formatter<chrono::weekday_last, charT>;
template<class charT> struct formatter<chrono::month_day, charT>;
template<class charT> struct formatter<chrono::month_day_last, charT>;
template<class charT> struct formatter<chrono::month_weekday, charT>;
template<class charT> struct formatter<chrono::month_weekday_last, charT>;
template<class charT> struct formatter<chrono::year_month, charT>;
template<class charT> struct formatter<chrono::year_month_day, charT>;
template<class charT> struct formatter<chrono::year_month_day_last, charT>;
template<class charT> struct formatter<chrono::year_month_weekday, charT>;
template<class charT> struct formatter<chrono::year_month_weekday_last, charT>;
template<class Rep, class Period, class charT>
  struct formatter<chrono::hh_mm_ss<duration<Rep, Period>>, charT>;
template<class charT> struct formatter<chrono::sys_info, charT>;
template<class charT> struct formatter<chrono::local_info, charT>;
template<class Duration, class TimeZonePtr, class charT>
  struct formatter<chrono::zoned_time<Duration, TimeZonePtr>, charT>;

namespace chrono {
  // 27.13, parsing
  template<class charT, class traits, class Alloc, class Parsable>
    unspecified
      parse(const basic_string<charT, traits, Alloc>& format, Parsable& tp);

  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);

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

```
    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 [time.clock.req]

1 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.

2 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 [tab:time.clock]

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

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*]

4 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) — lvalues 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 [time.traits]

27.4.1 `treat_as_floating_point` [time.traits.is.fp]

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

1 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` [time.traits.duration.values]

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

1 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;
```

2 *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*]

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

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

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

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

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

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

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

27.4.3 Specializations of `common_type` [time.traits.specializations]

```
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 [*Note*: 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 `duration`s.

27.4.4 Class template `is_clock` [time.traits.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:

(1.1) — the *qualified-id*s `T::rep`, `T::period`, `T::duration`, and `T::time_point` are valid and each denotes a type (**??**),

(1.2) — the expression `T::is_steady` is well-formed when treated as an unevaluated operand,

(1.3) — the expression `T::now()` is well-formed when treated as an unevaluated operand.

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

27.5 Class template `duration` [time.duration]

1 A `duration` type measures time between two points in time (`time_point`s). 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`.

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

    // 27.5.2, observer
    constexpr rep count() const;

    // 27.5.3, arithmetic
    constexpr common_type_t<duration> operator+() const;
    constexpr common_type_t<duration> operator-() const;
    constexpr duration& operator++();
    constexpr duration  operator++(int);
    constexpr duration& operator--();
    constexpr duration  operator--(int);

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

    constexpr duration& operator*=(const rep& rhs);
    constexpr duration& operator/=(const rep& rhs);
    constexpr duration& operator%=(const rep& rhs);
    constexpr duration& operator%=(const duration& rhs);

    // 27.5.4, special values
    static constexpr duration zero() noexcept;
    static constexpr duration min() noexcept;
    static constexpr duration max() noexcept;
  };
}
```

2 `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.

3 If `Period` is not a specialization of `ratio`, the program is ill-formed. If `Period::num` is not positive, the program is ill-formed.

4 Members of `duration` shall not throw exceptions other than those thrown by the indicated operations on their representations.

5 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.

6 [*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 1/30 of a second
                                     // (30 Hz) using a double
```

— *end example*]

27.5.1 Constructors [time.duration.cons]

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

1 ~~*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*]

2 ~~*Effects:* Constructs an object of type `duration`~~.

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

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

4 ~~*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);
duration<int, micro> us = ms;       // OK
duration<int, milli> ms2 = us;      // error
```

— *end example*]

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

27.5.2 Observer [time.duration.observer]

```
constexpr rep count() const;
```

1 *Returns:* `rep_`.

27.5.3 Arithmetic [time.duration.arithmetic]

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

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

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

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

```
constexpr duration& operator++();
```

3 *Effects:* As if by `++rep_`.

4 *Returns:* `*this`.

```
constexpr duration operator++(int);
```

5 *Returns:* `duration(rep_++)`.

```
constexpr duration& operator--();
```

6 *Effects:* As if by `--rep_`.

7 *Returns:* `*this`.

```
constexpr duration operator--(int);
```

8 *Returns:* `duration(rep_--)`.

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

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

10 *Returns:* `*this`.

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

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

12 *Returns:* `*this`.

```
constexpr duration& operator*=(const rep& rhs);
```

13 *Effects:* As if by: `rep_ *= rhs;`

14 *Returns:* `*this`.

```
constexpr duration& operator/=(const rep& rhs);
```

15 *Effects:* As if by: `rep_ /= rhs;`

16 *Returns:* `*this`.

```
constexpr duration& operator%=(const rep& rhs);
```

17 *Effects:* As if by: `rep_ %= rhs;`

18 *Returns:* `*this`.

```
constexpr duration& operator%=(const duration& rhs);
```

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

20 *Returns:* `*this`.

27.5.4 Special values [time.duration.special]

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

1 *Returns:* `duration(duration_values<rep>::zero())`.

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

2 *Returns:* `duration(duration_values<rep>::min())`.

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

3 *Returns:* `duration(duration_values<rep>::max())`.

27.5.5 Non-member arithmetic [time.duration.nonmember]

1 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);
```

2 *Returns:* `CD(CD(lhs).count() + CD(rhs).count())`.

```
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);
```

3 *Returns:* `CD(CD(lhs).count() - CD(rhs).count())`.

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

4 ~~*Remarks:*~~ *Constraints:* ~~This operator shall not participate in overload resolution unless `Rep2` is implicitly convertible to `common_type_t<Rep1, Rep2>`~~`is_convertible_v<Rep2, common_type_t<Rep1, Rep2»` is true.

5 *Returns:* `CD(CD(d).count() * s)`.

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

6 ~~*Remarks:*~~ *Constraints:* ~~This operator shall not participate in overload resolution unless `Rep1` is implicitly convertible to `common_type_t<Rep1, Rep2>`~~ `is_convertible_v<Rep1, common_type_t<Rep1, Rep2»` is true.

7 *Returns:* `d * s`.

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

8 ~~*Remarks:*~~ *Constraints:* ~~This operator shall not participate in overload resolution unless `Rep2` is implicitly convertible to `common_type_t<Rep1, Rep2>`~~ is_convertible_v<Rep2, common_type_t<Rep1, Rep2» is true and `Rep2` is not a specialization of `duration`.

9 *Returns:* `CD(CD(d).count() / s)`.

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

10 Let `CD` be `common_type_t<duration<Rep1, Period1>, duration<Rep2, Period2>>`.

11 *Returns:* `CD(lhs).count() / CD(rhs).count()`.

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

12 ~~*Remarks:*~~ *Constraints:* ~~This operator shall not participate in overload resolution unless `Rep2` is implicitly convertible to `common_type_t<Rep1, Rep2>`~~ is_convertible_v<Rep2, common_type_t<Rep1, Rep2» is true and `Rep2` is not a specialization of `duration`.

13 *Returns:* `CD(CD(d).count() % 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);
```

14 *Returns:* `CD(CD(lhs).count() % CD(rhs).count())`.

27.5.6 Comparisons [time.duration.comparisons]

1 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);
```

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

2 *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)))
```

3 [*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);
```

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

5 *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);
```

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

7 *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);
```

8 ~~*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`.

9 *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 [time.duration.literals]

1 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.

2 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.

3 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.

4 [*Example*: The following code shows some duration literals.

```
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);
```

5 *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);
```

6 *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);
```

7 *Returns:* A `duration` literal representing `sec` seconds.

8 [*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);
```

9 *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);
```

10 *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`.

```
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` [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 [time.point.cons]

```
constexpr time_point();
```

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

```
constexpr explicit time_point(const duration& d);
```

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

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

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

4 *Effects:* ~~Constructs an object of type `time_point`, initializing~~Initializes `d_` with `t.time_since_epoch()`.

27.6.2 Observer [time.point.observer]

```
constexpr duration time_since_epoch() const;
```

1 *Returns:* `d_`.

27.6.3 Arithmetic [time.point.arithmetic]

```
constexpr time_point& operator++();
```

1 *Effects:* `++d_`.

2 *Returns:* `*this`.

```
constexpr time_point operator++(int);
```

3 *Returns:* `time_point{d_++}`.

```
constexpr time_point& operator--();
```

4 *Effects:* `--d_`.

5 *Returns:* `*this`.

```
constexpr time_point operator--(int);
```

6 *Returns:* `time_point{d_--}`.

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

7 *Effects:* As if by: `d_ += d;`

8 *Returns:* `*this`.

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

9 *Effects:* As if by: `d_ -= d;`

10 *Returns:* `*this`.

27.6.4 Special values [time.point.special]

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

1 *Returns:* `time_point(duration::min())`.

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

2 *Returns:* `time_point(duration::max())`.

27.6.5 Non-member arithmetic [time.point.nonmember]

```
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);
```

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

```
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);
```

2 *Returns:* `rhs + lhs`.

```
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);
```

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

```
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);
```

4 *Returns:* `lhs.time_since_epoch() - rhs.time_since_epoch()`.

27.6.6 Comparisons [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);
```

1 *Returns:* `lhs.time_since_epoch() == rhs.time_since_epoch()`.

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

2 *Returns:* `lhs.time_since_epoch() < rhs.time_since_epoch()`.

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

3 *Returns:* `rhs < lhs`.

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

4 *Returns:* `!(rhs < lhs)`.

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

5 *Returns:* `!(lhs < rhs)`.

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

6 *Returns:* `lhs.time_since_epoch() <=> rhs.time_since_epoch()`.

27.6.7 `time_point_cast` [time.point.cast]

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

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

2 *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);
```

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

4 *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);
```

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

6 *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);
```

7 ~~*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`.

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

27.7 Clocks [time.clock]

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

27.7.1 Class `system_clock` [time.clock.system]

27.7.1.1 Overview [time.clock.system.overview]

```
namespace std::chrono {
  class system_clock {
  public:
    using rep        = see below;
    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;
  };
}
```

1 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 [time.clock.system.members]

```
using system_clock::rep = unspecified;
```

1 ~~*Requires:*~~ *Constraints:* `system_clock::duration::min() < system_clock::duration::zero()` ~~shall be~~is `true`.

[*Note*: This implies that `rep` is a signed type. — *end note*]

```
static time_t to_time_t(const time_point& t) noexcept;
```

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

```
static time_point from_time_t(time_t t) noexcept;
```

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

27.7.1.3 Non-member functions [time.clock.system.nonmembers]

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

1 ~~*Remarks:*~~ *Constraints:* ~~This operator shall not participate in overload resolution if~~ `treat_as_floating_point_v<typename Duration::rep>` is ~~`false` or, if~~`true` and `Duration{1}` ~~`>=`~~`<=` `days{1}`.

2 *Effects:* Equivalent to:

```
auto const dp = floor<days>(tp);
return os << format(os.getloc(), STATICALLY-WIDEN<charT>("{} {}"),
                    year_month_day{dp}, hh_mm_ss{tp-dp});
```

3 [*Example*:

```
cout << sys_seconds{0s} << '\n';                // 1970-01-01 00:00:00
cout << sys_seconds{946'684'800s} << '\n';      // 2000-01-01 00:00:00
cout << sys_seconds{946'688'523s} << '\n';      // 2000-01-01 01:02:03
```

— *end example*]

```
template<class charT, class traits>
  basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const sys_days& dp);
```

4 *Effects:* `os << year_month_day{dp}`.

5 *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,
                sys_time<Duration>& tp, basic_string<charT, traits, Alloc>* abbrev = nullptr,
                minutes* offset = nullptr);
```

6 *Effects:* Attempts to parse the input stream `is` into the `sys_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`.

7 *Returns:* `is`.

27.7.2 Class `utc_clock` [time.clock.utc]

27.7.2.1 Overview [time.clock.utc.overview]

```
namespace std::chrono {
  class utc_clock {
  public:
    using rep                       = a signed arithmetic type;
    using period                    = ratio<unspecified, unspecified>;
    using duration                  = chrono::duration<rep, period>;
    using time_point                = chrono::time_point<utc_clock>;
```

```
    static constexpr bool is_steady = unspecified;

    static time_point now();

    template<class Duration>
      static sys_time<common_type_t<Duration, seconds>>
        to_sys(const utc_time<Duration>& t);
    template<class Duration>
      static utc_time<common_type_t<Duration, seconds>>
        from_sys(const sys_time<Duration>& t);
  };
}
```

1 In contrast to `sys_time`, which does not take leap seconds into account, `utc_clock` and its associated `time_point`, `utc_time`, count time, including leap seconds, since 1970-01-01 00:00:00 UTC. [*Example*: `clock_cast<utc_clock>(sys_seconds{sys_days{1970y/January/1}}).time_since_epoch()` is `0s`. `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*]

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

27.7.2.2 Member functions [time.clock.utc.members]

```
static time_point now();
```

1 *Returns:* `from_sys(system_clock::now())`, or a more accurate value of `utc_time`.

```
template<class Duration>
  static sys_time<common_type_t<Duration, seconds>>
    to_sys(const utc_time<Duration>& u);
```

2 *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.

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

3 *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.

[*Example*:

```
auto t = sys_days{July/1/2015} - 2ns;
auto u = utc_clock::from_sys(t);
assert(u.time_since_epoch() - t.time_since_epoch() == 25s);
t += 1ns;
u = utc_clock::from_sys(t);
assert(u.time_since_epoch() - t.time_since_epoch() == 25s);
t += 1ns;
u = utc_clock::from_sys(t);
assert(u.time_since_epoch() - t.time_since_epoch() == 26s);
t += 1ns;
u = utc_clock::from_sys(t);
assert(u.time_since_epoch() - t.time_since_epoch() == 26s);
```

— *end example*]

27.7.2.3 Non-member functions [time.clock.utc.nonmembers]

```
template<class charT, class traits, class Duration>
  basic_ostream<charT, traits>&
```

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

1 *Effects:* Equivalent to:

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

2 [*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\n";
```

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

3 *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`.

4 *Returns:* `is`.

```
struct leap_second_info {
  bool    is_leap_second;
  seconds elapsed;
};
```

5 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);
```

6 *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` [time.clock.tai]

27.7.3.1 Overview [time.clock.tai.overview]

```
namespace std::chrono {
  class tai_clock {
  public:
    using rep                       = a signed arithmetic type;
    using period                    = ratio<unspecified, unspecified>;
    using duration                  = chrono::duration<rep, period>;
    using time_point                = chrono::time_point<tai_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 tai_time<Duration>&) noexcept;
    template<class Duration>
      static tai_time<common_type_t<Duration, seconds>>
        from_utc(const utc_time<Duration>&) 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(utc_clock::now()))` is `false`. — *end note*]

27.7.3.2 Member functions [time.clock.tai.members]

```
static time_point now();
```

1 *Returns:* `from_utc(utc_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;
```

2 *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;
```

3 *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 [time.clock.tai.nonmembers]

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

1 *Effects:* Equivalent to:

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

2 [*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);
```

3 *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`.

4 *Returns:* `is`.

27.7.4 Class `gps_clock` [time.clock.gps]

27.7.4.1 Overview [time.clock.gps.overview]

```
namespace std::chrono {
  class gps_clock {
  public:
    using rep                       = a signed arithmetic type;
    using period                    = ratio<unspecified, unspecified>;
    using duration                  = chrono::duration<rep, period>;
    using time_point                = 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 between 1970 and 1980.

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 [time.clock.gps.members]

```
static time_point now();
```

1 *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;
```

2 *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;
```

3 *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 [time.clock.gps.nonmembers]

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

1 *Effects:* Equivalent to:

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

2 [*Example*:

```
auto st = sys_days{2000y/January/1};
auto gt = clock_cast<gps_clock>(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);
```

3 *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`.

4 *Returns:* `is`.

27.7.5 Type `file_clock` [time.clock.file]

27.7.5.1 Overview [time.clock.file.overview]

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

1 `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_time_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 [time.clock.file.members]

1 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 [time.clock.file.nonmembers]

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

1 *Effects:* Equivalent to:

```
return os << format(STATICALLY-WIDEN<charT>("{:%F %T}"), 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,
                file_time<Duration>& tp, basic_string<charT, traits, Alloc>* abbrev = nullptr,
                minutes* offset = nullptr);
```

2 *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`.

3 *Returns:* `is`.

27.7.6 Class `steady_clock` [time.clock.steady]

```
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` [time.clock.hires]

```
namespace std::chrono {
  class high_resolution_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 = 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 [time.clock.local]

1 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 [time.clock.cast]

27.7.9.1 Class template `clock_time_conversion` [time.clock.conv]

```
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 [time.clock.cast.id]

```
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;
```

1 *Returns:* `t`.

```
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;
```

2 *Returns:* `t`.

```
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;
```

3 *Returns:* `t`.

27.7.9.3 Conversions between `system_clock` and `utc_clock` [time.clock.cast.sys.utc]

```
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;
```

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

```
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;
```

2 *Returns:* `utc_clock::to_sys(t)`.

27.7.9.4 Conversions between system_clock and other clocks [time.clock.cast.sys]

```
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));
```

1 ~~*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~~returns `sys_time<Duration>`, where `Duration` is a valid `chrono::duration` specialization~~, the program is ill-formed~~.

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

```
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));
```

3 ~~*Remarks:*~~ *Constraints:* ~~This function does not participate in overload resolution unless~~ `DestClock::from_sys(t)` is well-formed. ~~If~~

4 *Mandates:* `DestClock::from_sys(t)` ~~does not return~~returns `time_point<DestClock, Duration>`, where `Duration` is a valid `chrono::duration` specialization~~, the program is ill-formed~~.

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

27.7.9.5 Conversions between utc_clock and other clocks [time.clock.cast.utc]

```
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));
```

1 ~~*Remarks:*~~ *Constraints:* ~~This function does not participate in overload resolution unless~~ `SourceClock::to_utc(t)` is well-formed. ~~If~~

2 *Mandates:* `SourceClock::to_utc(t)` ~~does not~~ returns `utc_time<Duration>`, where `Duration` is a valid `chrono::duration` specialization~~, the program is ill-formed~~.

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

```
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));
```

4 ~~*Remarks:*~~ *Constraints:* ~~This function does not participate in overload resolution unless~~ `DestClock::from_utc(t)` is well-formed. ~~If~~

5 *Mandates:* `DestClock::from_utc(t)` ~~does not~~ returns `time_point<DestClock, Duration>`, where `Duration` is a valid `chrono::duration` specialization~~, the program is ill-formed~~.

6 *Returns:* `DestClock::from_utc(t)`.

27.7.9.6 Function template `clock_cast` [time.clock.cast.fn]

```
template<class DestClock, class SourceClock, class Duration>
  auto clock_cast(const time_point<SourceClock, Duration>& t);
```

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

(1.1) — `clock_time_conversion<DestClock, SourceClock>{}(t)`

(1.2) — `clock_time_conversion<DestClock, system_clock>{}(clock_time_conversion<system_clock, SourceClock>{}(t))`

(1.3) — `clock_time_conversion<DestClock, utc_clock>{}(clock_time_conversion<utc_clock, SourceClock>{}(t))`

(1.4) — `clock_time_conversion<DestClock, utc_clock>{}(clock_time_conversion<utc_clock, system_clock>{}(clock_time_conversion<system_clock, SourceClock>{}(t)))`

(1.5) — `clock_time_conversion<DestClock, system_clock>{}(clock_time_conversion<system_clock, utc_clock>{}(clock_time_conversion<utc_clock, SourceClock>{}(t)))`

A clock time conversion expression is considered better than another clock time conversion expression if it involves fewer `operator()` calls on `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 `clock_cast` is ambiguous and the program is ill-formed.

2 *Returns:* The best well-formed clock time conversion expression in the above list.

27.8 The civil calendar [time.cal]

27.8.1 In general [time.cal.general]

1 The types in 27.8 describe the civil (Gregorian) calendar and its relationship to `sys_days` and `local_days`.

27.8.2 Class `last_spec` [time.cal.last]

```
namespace std::chrono {
  struct last_spec {
    explicit last_spec() = default;
  };
}
```

1 The type `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.

27.8.3 Class `day` [time.cal.day]

27.8.3.1 Overview [time.cal.day.overview]

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

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

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

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` ~~is~~ meets the *Cpp17EqualityComparable* (Table **??**) and *Cpp17LessThanComparable* (Table **??**) requirements, and participates in basic arithmetic with `days` objects, which represent a difference between two `day` objects.

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

27.8.3.2 Member functions [time.cal.day.members]

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

1 *Effects:* ~~Constructs an object of type `day` by initializing~~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 [time.cal.day.nonmembers]

```
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;
```

4 *Returns:* `y + x`.

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

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

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

6 *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);
```

7 *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);
```

8 *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.

9 *Returns:* `is`.

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

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

27.8.4 Class `month` [time.cal.month]

27.8.4.1 Overview [time.cal.month.overview]

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

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

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

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

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` ~~is~~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 [time.cal.month.members]

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

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

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

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

3 *Returns:* `*this`.

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

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

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

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

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

7 *Returns:* `*this`.

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

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

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

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

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

11 *Returns:* `*this`.

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

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

13 *Returns:* `*this`.

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

14 *Returns:* `m_`.

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

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

27.8.4.3 Non-member functions [time.cal.month.nonmembers]

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

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

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

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

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

3 *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*]

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

4 *Returns:* `y + x`.

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

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

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

6 *Returns:* If `x.ok() == true` and `y.ok() == true`, returns a value `m` in the range `[months{0}, 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);
```

7 *Effects:* Equivalent to:

```
return os << (m.ok() ?
  format(os.getloc(), STATICALLY-WIDEN<charT>("{:%b}"), m) :
  format(os.getloc(), STATICALLY-WIDEN<charT>("{} is not a valid month"),
         static_cast<unsigned>(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);
```

8 *Effects:* Attempts to parse the input stream `is` into the `month` `m` using the format flags given in the NTCTS `fmt` as specified in 27.13. If the parse fails to decode a valid month, `is.setstate(ios_base::failbit)` shall be called and `m` 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.

9 *Returns:* `is`.

27.8.5 Class `year` [time.cal.year]

27.8.5.1 Overview [time.cal.year.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;
  };
}
```

1 `year` represents a year in the civil calendar. It can represent values in the range `[min(), max()]`. It can be constructed with any `int` value, which will be subsequently truncated to fit into `year`'s unspecified internal

storage. `year` ~~is~~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 [time.cal.year.members]

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

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

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

2 *Effects:* `++y_`.

3 *Returns:* `*this`.

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

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

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

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

6 *Effects:* `--y_`.

7 *Returns:* `*this`.

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

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

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

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

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

11 *Returns:* `*this`.

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

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

13 *Returns:* `*this`.

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

14 *Returns:* `*this`.

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

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

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

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

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

17 *Returns:* `y_`.

```
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 [time.cal.year.nonmembers]

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

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

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

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

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

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

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

4 *Returns:* `y + x`.

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

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

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

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

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

7 *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);
```

8 *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.

9 *Returns:* `is`.

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

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

27.8.6 Class `weekday` [time.cal.wd]

27.8.6.1 Overview [time.cal.wd.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;
```

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

    constexpr unsigned c_encoding() const noexcept;
    constexpr unsigned iso_encoding() const noexcept;
    constexpr bool ok() const noexcept;

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

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` ~~is~~meets the *Cpp17EqualityComparable* (Table **??**) 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*]

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

27.8.6.2 Member functions [time.cal.wd.members]

`constexpr explicit weekday(unsigned wd) noexcept;`

1 *Effects:* ~~Constructs an object of type `weekday` by initializing~~Initializes `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;`

2 *Effects:* ~~Constructs an object of type `weekday` by computing~~Computes what day of the week corresponds to the `sys_days` `dp`, and ~~representing~~initializes that day of the week in `wd_`.

3 [*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;`

4 *Effects:* ~~Constructs an object of type `weekday` by computing~~Computes what day of the week corresponds to the `local_days` `dp`, and ~~representing~~initializes that day of the week in `wd_`.

5 ~~*Remarks:*~~ *Ensures:* The value ~~after construction~~ is identical to that constructed from `sys_days{dp.time_since_epoch()}`.

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

6 *Effects:* `*this += days{1}`.

7 *Returns:* `*this`.

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

8 *Effects:* `++(*this)`.

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

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

10 *Effects:* `*this -= days{1}`.

11 *Returns:* `*this`.

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

12 *Effects:* `--(*this)`.

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

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

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

15 *Returns:* `*this`.

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

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

17 *Returns:* `*this`.

```
constexpr unsigned c_encoding() const noexcept;
```

18 *Returns:* `wd_`.

```
constexpr unsigned iso_encoding() const noexcept;
```

19 *Returns:* `wd_ == 0u ? 7u : wd_`.

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

20 *Returns:* `wd_ <= 6`.

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

21 *Returns:* `{*this, index}`.

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

22 *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;
```

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

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

2 *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;
```

3 *Returns:* `y + x`.

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

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

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

5 *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);
```

6 *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` [time.cal.wdidx]

27.8.7.1 Overview [time.cal.wdidx.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 [time.cal.wdidx.members]

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

1 *Effects:* ~~Constructs an object of type `weekday_indexed` by initializing~~Initializes `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 [time.cal.wdidx.nonmembers]

```
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);
```

2 *Effects:* 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` [time.cal.wdlast]

27.8.8.1 Overview [time.cal.wdlast.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 [time.cal.wdlast.members]

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

1 *Effects:* ~~Constructs an object of type `weekday_last` by initializing~~Initializes `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 [time.cal.wdlast.nonmembers]

```
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` [time.cal.md]

27.8.9.1 Overview [time.cal.md.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;
```

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

1 `month_day` represents a specific day of a specific month, but with an unspecified year. `month_day` ~~is~~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 [time.cal.md.members]

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

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

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

2 *Returns:* `m_`.

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

3 *Returns:* `d_`.

```
constexpr bool ok() 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 [time.cal.md.nonmembers]

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

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

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

2 *Effects:* Equivalent to:

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

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

3 *Effects:* Equivalent to:

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

```
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);
```

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` [time.cal.mdlast]

```
namespace std::chrono {
  class month_day_last {
    chrono::month m_;                    // exposition only
```

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

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

1 `month_day_last` represents the last day of a month.

2 [*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*]

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

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

4 *Effects:* ~~Constructs an object of type `month_day_last` by initializing~~Initializes `m_` with `m`.

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

5 *Returns:* `m_`.

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

6 *Returns:* `m_.ok()`.

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

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

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

8 *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);
```

9 *Effects:* Equivalent to:

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

27.8.11 Class `month_weekday` [time.cal.mwd]

27.8.11.1 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 chrono::month           month()           const noexcept;
    constexpr chrono::weekday_indexed weekday_indexed() const noexcept;
    constexpr bool ok() const noexcept;
  };
}
```

1 `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`.

2 [*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]);
```

— *end example*]

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

27.8.11.2 Member functions [time.cal.mwd.members]

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

1 *Effects:* ~~Constructs an object of type `month_weekday` by initializing~~Initializes `m_` with `m`, and `wdi_` with `wdi`.

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

2 *Returns:* `m_`.

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

3 *Returns:* `wdi_`.

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

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

27.8.11.3 Non-member functions [time.cal.mwd.nonmembers]

```
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:

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

27.8.12 Class `month_weekday_last` [time.cal.mwdlast]

27.8.12.1 Overview [time.cal.mwdlast.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 [time.cal.mwdlast.members]

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

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

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

2 *Returns:* `m_`.

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

3 *Returns:* `wdl_`.

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

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

27.8.12.3 Non-member functions [time.cal.mwdlast.nonmembers]

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

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

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

2 *Effects:* Equivalent to:

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

27.8.13 Class `year_month` [time.cal.ym]

27.8.13.1 Overview [time.cal.ym.overview]

```
namespace std::chrono {
  class year_month {
    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;
  };
}
```

1 `year_month` represents a specific month of a specific year, but with an unspecified day. `year_month` ~~is~~meets the *Cpp17EqualityComparable* (Table **??**) and *Cpp17LessThanComparable* (Table **??**) requirements.

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

27.8.13.2 Member functions [time.cal.ym.members]

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

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

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

2 *Returns:* `y_`.

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

3 *Returns:* `m_`.

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

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

5 *Returns:* `*this`.

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

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

7 *Returns:* `*this`.

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

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

9 *Returns:* `*this`.

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

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

11 *Returns:* `*this`.

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

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

27.8.13.3 Non-member functions [time.cal.ym.nonmembers]

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

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

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

2 *Effects:* Equivalent to:

```
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;
```

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

4 *Complexity:* $\mathcal{O}(1)$ with respect to the value of `dm`.

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

5 *Returns:* `ym + dm`.

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

6 *Returns:* `ym + -dm`.

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

7 *Returns:*

```
x.year() - y.year() + months{static_cast<int>(unsigned{x.month()}) -
                             static_cast<int>(unsigned{y.month()})}
```

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

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

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

9 *Returns:* `ym + dy`.

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

10 *Returns:* `ym + -dy`.

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

11 *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);
```

12 *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.

13 *Returns:* `is`.

27.8.14 Class `year_month_day` [time.cal.ymd]

27.8.14.1 Overview [time.cal.ymd.overview]

```
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;
  };
}
```

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` ~~is~~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 [time.cal.ymd.members]

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

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

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

2 *Effects:* ~~Constructs an object of type `year_month_day` by initializing~~Initializes `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*]

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

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

4 ~~*Remarks:*~~ *Ensures:* For any value `ymd` of type `year_month_day` for which `ymd.ok()` is `true`, `ymd == year_month_day{sys_days{ymd}}` is `true`.

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

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

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

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

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

8 *Returns:* `*this`.

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

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

10 *Returns:* `*this`.

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

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

12 *Returns:* `*this`.

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

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

14 *Returns:* `*this`.

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

15 *Returns:* `y_`.

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

16 *Returns:* `m_`.

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

17 *Returns:* `d_`.

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

18 *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.

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

20 [*Example*:

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

— *end example*]

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

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

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

22 *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 [time.cal.ymd.nonmembers]

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

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

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

2 *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();
```

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

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

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

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

5 *Returns:* `ymd + dm`.

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

6 *Returns:* `ymd + (-dm)`.

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

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

8 [*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*]

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

9 *Returns:* `ymd + dy`.

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

10 *Returns:* `ymd + (-dy)`.

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

11 *Effects:* Equivalent to:

```
return os << (ymd.ok() ?
  format(STATICALLY-WIDEN<charT>("{:%F}"), ymd) :
  format(STATICALLY-WIDEN<charT>("{:%F} is not a valid date"), 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, basic_string<charT, traits, Alloc>* abbrev = nullptr,
```

```
            minutes* offset = nullptr);
```

12 *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.

13 *Returns:* `is`.

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

27.8.15.1 Overview [time.cal.ymdlast.overview]

```
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;
  };
}
```

1 `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` ~~is~~meets the *Cpp17EqualityComparable* (Table **??**) and *Cpp17LessThanComparable* (Table **??**) requirements,

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

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

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

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

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

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

3 *Returns:* `*this`.

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

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

5 *Returns:* `*this`.

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

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

7 *Returns:* `*this`.

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

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

9 *Returns:* `*this`.

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

10 *Returns:* `y_`.

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

11 *Returns:* `mdl_.month()`.

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

12 *Returns:* `mdl_`.

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

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

14 [*Note*: This value may be computed on demand. — *end note*]

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

15 *Returns:* `sys_days{year()/month()/day()}`.

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

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

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

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

27.8.15.3 Non-member functions [time.cal.ymdlast.nonmembers]

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

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

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

2 *Effects:* Equivalent to:

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

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

3 *Returns:* `(ymdl.year() / ymdl.month() + dm) / last`.

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

4 *Returns:* `ymdl + dm`.

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

5 *Returns:* `ymdl + (-dm)`.

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

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

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

7 *Returns:* `ymdl + dy`.

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

8 *Returns:* `ymdl + (-dy)`.

```
template<class charT, class traits>
  basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const year_month_day_last& ymdl);
```

9 *Effects:* Equivalent to:

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

27.8.16 Class `year_month_weekday` [time.cal.ymwd]

27.8.16.1 Overview [time.cal.ymwd.overview]

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

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

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

    constexpr 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_weekday` represents a specific year, month, and n^{th} weekday of the month. `year_month_weekday` is a field-based time point with a resolution of `days`. [*Note*: `year_month_weekday` supports `years`- and `months`-oriented arithmetic, but not `days`-oriented arithmetic. For the latter, there is a conversion to `sys_days`, which efficiently supports `days`-oriented arithmetic. — *end note*] `year_month_weekday` is *Cpp17EqualityComparable* (Table **??**).

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

27.8.16.2 Member functions [time.cal.ymwd.members]

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

1 *Effects:* ~~Constructs an object of type `year_month_weekday` by initializing~~Initializes `y_` with `y`, `m_` with `m`, and `wdi_` with `wdi`.

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

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

3 *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;`

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

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

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

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

7 *Returns:* `*this`.

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

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

9 *Returns:* `*this`.

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

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

11 *Returns:* `*this`.

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

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

13 *Returns:* `*this`.

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

14 *Returns:* `y_`.

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

15 *Returns:* `m_`.

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

16 *Returns:* `wdi_.weekday()`.

`constexpr unsigned index() const noexcept;`

17 *Returns:* `wdi_.index()`.

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

18 *Returns:* `wdi_`.

`constexpr operator sys_days() const noexcept;`

19 *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;`

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

`constexpr bool ok() const noexcept;`

21 *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 [time.cal.ymwd.nonmembers]

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

1 *Returns:*

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

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

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

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

3 *Returns:* `ymwd + dm`.

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

4 *Returns:* `ymwd + (-dm)`.

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

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

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

6 *Returns:* `ymwd + dy`.

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

7 *Returns:* `ymwd + (-dy)`.

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

8 *Effects:* Equivalent to:

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

27.8.17 Class `year_month_weekday_last` [time.cal.ymwdlast]

27.8.17.1 Overview [time.cal.ymwdlast.overview]

```
namespace std::chrono {
  class year_month_weekday_last {
    chrono::year         y_;    // exposition only
    chrono::month        m_;    // exposition only
    chrono::weekday_last wdl_;  // exposition only

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

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

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

    constexpr          operator sys_days()   const noexcept;
    constexpr explicit operator local_days() const noexcept;
    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*: `year_month_weekday_last` supports `years`- and `months`-oriented arithmetic, but not `days`-oriented arithmetic. For the latter, there is a conversion to `sys_days`, which efficiently supports `days`-oriented arithmetic. — *end note*] `year_month_weekday_last` is *Cpp17EqualityComparable* (Table **??**).

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

27.8.17.2 Member functions [time.cal.ymwdlast.members]

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

1 *Effects:* ~~Constructs an object of type `year_month_weekday_last` by initializing~~Initializes `y_` with `y`, `m_` with `m`, and `wdl_` with `wdl`.

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

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

3 *Returns:* `*this`.

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

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

5 *Returns:* `*this`.

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

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

7 *Returns:* `*this`.

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

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

9 *Returns:* `*this`.

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

10 *Returns:* `y_`.

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

11 *Returns:* `m_`.

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

12 *Returns:* `wdl_.weekday()`.

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

13 *Returns:* `wdl_`.

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

14 *Returns:* If `ok() == true`, returns a `sys_days` that represents the last `weekday()` of `year()/month()`. Otherwise the returned value is unspecified.

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

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

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

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

27.8.17.3 Non-member functions [time.cal.ymwdlast.nonmembers]

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

1 *Returns:*

```
x.year() == y.year() && x.month() == y.month() && x.weekday_last() == y.weekday_last()
```

```
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 [time.cal.operators]

1 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*:

```
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;    // error: no viable operator/ for first /
auto d = 2015/April/4;      // error: no viable operator/ for first /
```

— *end example*]

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

5 *Returns:* `{y, m}`.

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

6 *Returns:* `y / month(m)`.

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

7 *Returns:* `{m, d}`.

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

8 *Returns:* `m / day(d)`.

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

9 *Returns:* `month(m) / d`.

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

10 *Returns:* `m / d`.

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

11 *Returns:* `month(m) / d`.

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

12 *Returns:* `month_day_last{m}`.

```
constexpr month_day_last
  operator/(int m, last_spec) noexcept;
```

13 *Returns:* `month(m) / last`.

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

14 *Returns:* `m / last`.

```
constexpr month_day_last
  operator/(last_spec, int m) noexcept;
```

15 *Returns:* `month(m) / last`.

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

16 *Returns:* `{m, wdi}`.

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

17 *Returns:* `month(m) / wdi`.

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

18 *Returns:* `m / wdi`.

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

19 *Returns:* `month(m) / wdi`.

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

20 *Returns:* `{m, wdl}`.

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

21 *Returns:* `month(m) / wdl`.

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

22 *Returns:* `m / wdl`.

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

23 *Returns:* `month(m) / wdl`.

```
constexpr year_month_day
  operator/(const year_month& ym, const day& d) noexcept;
```

24 *Returns:* `{ym.year(), ym.month(), d}`.

```
constexpr year_month_day
  operator/(const year_month& ym, int d) noexcept;
```

25 *Returns:* `ym / day(d)`.

```
constexpr year_month_day
  operator/(const year& y, const month_day& md) noexcept;
```

26 *Returns:* `y / md.month() / md.day()`.

```
constexpr year_month_day
  operator/(int y, const month_day& md) noexcept;
```

27 *Returns:* `year(y) / md`.

```
constexpr year_month_day
  operator/(const month_day& md, const year& y) noexcept;
```

28 *Returns:* `y / md`.

```
constexpr year_month_day
  operator/(const month_day& md, int y) noexcept;
```

29 *Returns:* `year(y) / md`.

```
constexpr year_month_day_last
  operator/(const year_month& ym, last_spec) noexcept;
```

30 *Returns:* `{ym.year(), month_day_last{ym.month()}}`.

```
constexpr year_month_day_last
  operator/(const year& y, const month_day_last& mdl) noexcept;
```

31 *Returns:* `{y, mdl}`.

```
constexpr year_month_day_last
  operator/(int y, const month_day_last& mdl) noexcept;
```

32 *Returns:* `year(y) / mdl`.

```
constexpr year_month_day_last
  operator/(const month_day_last& mdl, const year& y) noexcept;
```

33 *Returns:* `y / mdl`.

```
constexpr year_month_day_last
  operator/(const month_day_last& mdl, int y) noexcept;
```

34 *Returns:* `year(y) / mdl`.

```
constexpr year_month_weekday
  operator/(const year_month& ym, const weekday_indexed& wdi) noexcept;
```

35 *Returns:* `{ym.year(), ym.month(), wdi}`.

```
constexpr year_month_weekday
  operator/(const year& y, const month_weekday& mwd) noexcept;
```

36 *Returns:* `{y, mwd.month(), mwd.weekday_indexed()}`.

```
constexpr year_month_weekday
  operator/(int y, const month_weekday& mwd) noexcept;
```

37 *Returns:* `year(y) / mwd`.

```
constexpr year_month_weekday
  operator/(const month_weekday& mwd, const year& y) noexcept;
```

38 *Returns:* `y / mwd`.

```
constexpr year_month_weekday
  operator/(const month_weekday& mwd, int y) noexcept;
```

39 *Returns:* `year(y) / mwd`.

```
constexpr year_month_weekday_last
  operator/(const year_month& ym, const weekday_last& wdl) noexcept;
```

40 *Returns:* `{ym.year(), ym.month(), wdl}`.

```
constexpr year_month_weekday_last
  operator/(const year& y, const month_weekday_last& mwdl) noexcept;
```

41 *Returns:* `{y, mwdl.month(), mwdl.weekday_last()}`.

```
constexpr year_month_weekday_last
  operator/(int y, const month_weekday_last& mwdl) noexcept;
```

42 *Returns:* `year(y) / mwdl`.

```
constexpr year_month_weekday_last
  operator/(const month_weekday_last& mwdl, const year& y) noexcept;
```

43 *Returns:* `y / mwdl`.

```
constexpr year_month_weekday_last
  operator/(const month_weekday_last& mwdl, int y) noexcept;
```

44 *Returns:* `year(y) / mwdl`.

27.9 Class template `hh_mm_ss` [time.hms]

27.9.1 Overview [time.hms.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;
```

```
  private:
    bool            is_neg;     // exposition only
    chrono::hours   h;          // exposition only
    chrono::minutes m;          // exposition only
    chrono::seconds s;          // exposition only
    precision       ss;         // exposition only
  };
}
```

1 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 [time.hms.members]

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

1 `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 87: Examples for `fractional_width` [tab:time.hms.width]

Duration	`fractional_width`	Formatted fractional second output
`hours`, `minutes`, and `seconds`	0	
`milliseconds`	3	0.001
`microseconds`	6	0.000001
`nanoseconds`	9	0.000000001
`duration<int, ratio<1, 2>>`	1	0.5
`duration<int, ratio<1, 3>>`	6	0.333333
`duration<int, ratio<1, 4>>`	2	0.25
`duration<int, ratio<1, 5>>`	1	0.2
`duration<int, ratio<1, 6>>`	6	0.166666
`duration<int, ratio<1, 7>>`	6	0.142857
`duration<int, ratio<1, 8>>`	3	0.125
`duration<int, ratio<1, 9>>`	6	0.111111
`duration<int, ratio<1, 10>>`	1	0.1
`duration<int, ratio<756, 625>>`	4	0.2096

— *end example*]

```
using precision = see below;
```

2 `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())`.

(3.5) — 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*]

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

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

5 *Returns:* `is_neg`.

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

6 *Returns:* `h`.

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

7 *Returns:* `m`.

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

8 *Returns:* `s`.

```
constexpr precision subseconds() const noexcept;
```

9 *Returns:* `ss`.

```
constexpr precision to_duration() const noexcept;
```

10 *Returns:* If `is_neg`, returns `-(h + m + s + ss)`, otherwise returns `h + m + s + ss`.

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

11 *Returns:* `to_duration()`.

27.9.3 Non-members [time.hms.nonmembers]

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

1 *Effects:* Equivalent to:

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

2 [*Example*:

```
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 [time.12]

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

```
constexpr bool is_am(const hours& h) noexcept;
```

2 *Returns:* `0h <= h && h <= 11h`.

```
constexpr bool is_pm(const hours& h) noexcept;
```

3 *Returns:* `12h <= h && h <= 23h`.

```
constexpr hours make12(const hours& h) noexcept;
```

4 *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;
```

5 *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 [time.zone]

27.11.1 In general [time.zone.general]

1 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 [time.zone.db]

27.11.2.1 Class `tzdb` [time.zone.db.tzdb]

```
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;
  };
}
```

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

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

2 *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*]

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

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

27.11.2.2 Class `tzdb_list` [time.zone.db.list]

```
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.

2 `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;
```

3 [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*]

4 *Returns:* A reference to the first `tzdb` in the container.

```
const_iterator erase_after(const_iterator p);
```

5 *Requires:* The iterator following `p` is dereferenceable.

6 *Effects:* Erases the `tzdb` referred to by the iterator following `p`.

7 *Returns:* An iterator pointing to the element following the one that was erased, or `end()` if no such element exists.

8 ~~*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*]

9 *Throws:* Nothing.

```
const_iterator begin() const noexcept;
```

10 *Returns:* An iterator referring to the first `tzdb` in the container.

```
const_iterator end() const noexcept;
```

11 *Returns:* An iterator referring to the position one past the last `tzdb` in the container.

```
const_iterator cbegin() const noexcept;
```

12 *Returns:* `begin()`.

```
const_iterator cend() const noexcept;
```

13 *Returns:* `end()`.

27.11.2.3 Time zone database access [time.zone.db.access]

```
tzdb_list& get_tzdb_list();
```

1 *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`.

2 [Editor's note: reordered synchronization before Returns]

~~*Remarks:*~~ Synchronization: It is safe to call this function from multiple threads at one time.

3 *Returns:* A reference to the database.

4 *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();
```

5 *Returns:* `get_tzdb_list().front()`.

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

6 *Returns:* `get_tzdb().locate_zone(tz_name)`.

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

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

8 *Returns:* `get_tzdb().current_zone()`.

27.11.2.4 Remote time zone database support [time.zone.db.remote]

1 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();
```

2 *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()`.

3 [Editor's note: reordered synchronization before Returns]

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

4 [Editor's note: reordered ensures before Returns]

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

5 *Returns:* `get_tzdb_list().front()`.

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

```
string remote_version();
```

7 *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 [time.zone.exception]

27.11.3.1 Class `nonexistent_local_time` [time.zone.exception.nonexist]

```
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);
  };
}
```

1 `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);
```

2 *Requires:* `i.result == local_info::nonexistent`.

3 *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:

```
ostringstream os;
os << tp << " is in a gap between\n"
   << local_seconds{i.first.end.time_since_epoch()} + i.first.offset << ' '
```

```
    << i.first.abbrev << " and\n"
    << local_seconds{i.second.begin.time_since_epoch()} + i.second.offset << ' '
    << i.second.abbrev
    << " which are both equivalent to\n"
    << 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` [time.zone.exception.ambig]

```
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~~Initializes 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\n"
   << tp << ' ' << i.first.abbrev << " == "
   << tp - i.first.offset << " UTC or\n"
   << tp << ' ' << 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[1]/November/2016} + 1h + 30min};
  } catch (const ambiguous_local_time& e) {
    std::cout << e.what() << '\n';
```

```
    }
  }
```

Produces the output:

```
2016-11-06 01:30:00 is ambiguous.  It could be
2016-11-06 01:30:00 EDT == 2016-11-06 05:30:00 UTC or
2016-11-06 01:30:00 EST == 2016-11-06 06:30:00 UTC
```

— *end example*]

27.11.4 Information classes [time.zone.info]

27.11.4.1 Class sys_info [time.zone.info.sys]

```
namespace std::chrono {
  struct sys_info {
    sys_seconds   begin;
    sys_seconds   end;
    seconds       offset;
    minutes       save;
    string        abbrev;
  };
}
```

1 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`.

2 [*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*]

3 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`.

4 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:

```
offset = local_time - sys_time
```

5 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`).

6 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.

```
template<class charT, class traits>
  basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const sys_info& r);
```

7 *Effects:* Streams out the `sys_info` object `r` in an unspecified format.

8 *Returns:* `os`.

27.11.4.2 Class local_info [time.zone.info.local]

```
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;
```

```
  };
}
```

1 [*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*]

2 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);
```

3 *Effects:* Streams out the `local_info` object `r` in an unspecified format.

4 *Returns:* `os`.

27.11.5 Class `time_zone` [time.zone.timezone]

27.11.5.1 Overview [time.zone.overview]

```
namespace std::chrono {
  class time_zone {
  public:
    time_zone(time_zone&&) = default;
    time_zone& operator=(time_zone&&) = default;

    // unspecified additional constructors

    string_view name() const noexcept;

    template<class Duration> sys_info   get_info(const sys_time<Duration>& st)   const;
    template<class Duration> local_info get_info(const local_time<Duration>& tp) const;

    template<class Duration>
      sys_time<common_type_t<Duration, seconds>>
        to_sys(const local_time<Duration>& tp) const;

    template<class Duration>
      sys_time<common_type_t<Duration, seconds>>
        to_sys(const local_time<Duration>& tp, choose z) const;

    template<class Duration>
      local_time<common_type_t<Duration, seconds>>
        to_local(const sys_time<Duration>& tp) const;
  };
}
```

1 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*]

27.11.5.2 Member functions [time.zone.members]

```
string_view name() const noexcept;
```

1 *Returns:* The name of the `time_zone`.

2 [*Example*: `"America/New_York"`. — *end example*]

```
template<class Duration>
  sys_info get_info(const sys_time<Duration>& st) const;
```

3 *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;
```

4 *Returns:* A `local_info` for `tp`.

```
template<class Duration>
  sys_time<common_type_t<Duration, seconds>>
    to_sys(const local_time<Duration>& tp) const;
```

5 *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`.

6 *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;
```

7 *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;
```

8 *Returns:* The `local_time` associated with `tp` and this `time_zone`.

27.11.5.3 Non-member functions [time.zone.nonmembers]

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

1 *Returns:* `x.name() == y.name()`.

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

2 *Returns:* `x.name() <=> y.name()`.

27.11.6 Class template `zoned_traits` [time.zone.zonedtraits]

```
namespace std::chrono {
  template<class T> struct zoned_traits {};
}
```

1 `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();
```

2 *Returns:* `std::chrono::locate_zone("UTC")`.

```
static const time_zone* locate_zone(string_view name);
```

3 *Returns:* `std::chrono::locate_zone(name)`.

27.11.7 Class template `zoned_time` [time.zone.zonedtime]

27.11.7.1 Overview [time.zone.zonedtime.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>
    zoned_time(string_view, local_time<Duration>, choose = choose::earliest)
      -> 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>;
}
```

1 `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.

2 If `Duration` is not a specialization of `chrono::duration`, the program is ill-formed.

27.11.7.2 Constructors [time.zone.zonedtime.ctor]

```
zoned_time();
```

1 ~~*Remarks:*~~ *Constraints:* ~~This constructor does not participate in overload resolution unless~~ `traits::default_zone()` is a well-formed expression.

2 *Effects:* Constructs a `zoned_time` by initializing `zone_` with `traits::default_zone()` and default constructing `tp_`.

```
zoned_time(const sys_time<Duration>& st);
```

3 ~~*Remarks:*~~ *Constraints:* ~~This constructor does not participate in overload resolution unless~~ `traits::default_zone()` is a well-formed expression.

4 *Effects:* Constructs a `zoned_time` by initializing `zone_` with `traits::default_zone()` and `tp_` with `st`.

```
explicit zoned_time(TimeZonePtr z);
```

5 ~~*Requires:*~~ *Expects:* `z` refers to a time zone.

6 *Effects:* Constructs a `zoned_time` by initializing `zone_` with `std::move(z)`.

```
explicit zoned_time(string_view name);
```

7 ~~*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{})`.

8 *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;
```

9 ~~*Remarks:*~~ *Constraints:* ~~Does not participate in overload resolution unless sys_time<Duration2> is implicitly convertible to sys_time<Duration>~~ `is_convertible_v<sys_time<Duration2>, sys_time<Duration»` is true.

10 *Effects:* Constructs a `zoned_time` by initializing `zone_` with `y.zone_` and `tp_` with `y.tp_`.

`zoned_time(TimeZonePtr z, const sys_time<Duration>& st);`

11 ~~*Requires:*~~ *Expects:* `z` refers to a time zone.

12 *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);`

13 ~~*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`.

14 *Effects:* Equivalent to construction with `{traits::locate_zone(name), st}`.

`zoned_time(TimeZonePtr z, const local_time<Duration>& tp);`

15 ~~*Requires:*~~ *Expects:* `z` refers to a time zone.

16 ~~*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>`.

17 *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);`

18 ~~*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 `tp`.

19 *Effects:* Equivalent to construction with `{traits::locate_zone(name), tp}`.

`zoned_time(TimeZonePtr z, const local_time<Duration>& tp, choose c);`

20 ~~*Requires:*~~ *Expects:* `z` refers to a time zone.

21 ~~*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>`.

22 *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);`

23 ~~*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`.

24 *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);
```

25 ~~*Remarks:*~~ *Constraints:* ~~Does not participate in overload resolution unless `sys_time<Duration2>` is implicitly convertible to `sys_time<Duration>`~~ `is_convertible_v<sys_time<Duration2>, sys_time<Duration»` is true.

26 ~~*Requires:*~~ *Expects:* `z` refers to a valid time zone.

27 *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);
```

28 ~~*Remarks:*~~ *Constraints:* ~~Does not participate in overload resolution unless `sys_time<Duration2>` is implicitly convertible to `sys_time<Duration>`~~ `is_convertible_v<sys_time<Duration2>, sys_time<Duration»` is true.

29 ~~*Requires:*~~ *Expects:* `z` refers to a valid time zone.

30 *Effects:* Equivalent to construction with `{z, y}`.

31 [*Note*: The `choose` parameter has no effect. — *end note*]

```
zoned_time(string_view name, const zoned_time<Duration>& y);
```

32 ~~*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`.

33 *Effects:* Equivalent to construction with `{traits::locate_zone(name), y}`.

```
zoned_time(string_view name, const zoned_time<Duration>& y, choose c);
```

34 ~~*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`.

35 *Effects:* Equivalent to construction with `{traits::locate_zone(name), y, c}`.

36 [*Note*: The `choose` parameter has no effect. — *end note*]

27.11.7.3 Member functions [time.zone.zonedtime.members]

```
zoned_time& operator=(const sys_time<Duration>& st);
```

1 *Effects:* After assignment, `get_sys_time() == st`. This assignment has no effect on the return value of `get_time_zone()`.

2 *Returns:* `*this`.

```
zoned_time& operator=(const local_time<Duration>& lt);
```

3 *Effects:* After assignment, `get_local_time() == lt`. This assignment has no effect on the return value of `get_time_zone()`.

4 *Returns:* `*this`.

```
operator sys_time<duration>() const;
```

5 *Returns:* `get_sys_time()`.

```
explicit operator local_time<duration>() const;
```

6 *Returns:* `get_local_time()`.

```
TimeZonePtr get_time_zone() const;
```

7 *Returns:* `zone_`.

```
local_time<duration> get_local_time() const;
```

8 *Returns:* `zone_->to_local(tp_)`.

```
sys_time<duration> get_sys_time() const;
```

9 *Returns:* `tp_`.

```
sys_info get_info() const;
```

10 *Returns:* `zone_->get_info(tp_)`.

27.11.7.4 Non-member functions [time.zone.zonedtime.nonmembers]

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

1 *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);
```

2 *Effects:* Streams the value returned from `t.get_local_time()` to `os` using the format `"%F %T %Z"`.

3 *Returns:* `os`.

27.11.8 Class `leap` [time.zone.leap]

27.11.8.1 Overview [time.zone.leap.overview]

```
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*:

```
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
1985-07-01 00:00:00
1988-01-01 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
1996-01-01 00:00:00
1997-07-01 00:00:00
1999-01-01 00:00:00
2006-01-01 00:00:00
2009-01-01 00:00:00
2012-07-01 00:00:00
2015-07-01 00:00:00
2017-01-01 00:00:00
```

— *end example*]

27.11.8.2 Member functions [time.zone.leap.members]

```
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]

```
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;
```

2 *Returns:* `x.date() <=> y.date()`.

```
template<class Duration>
  constexpr bool operator==(const leap& x, const sys_time<Duration>& y) noexcept;
```

3 *Returns:* `x.date() == y`.

```
template<class Duration>
  constexpr bool operator<(const leap& x, const sys_time<Duration>& y) noexcept;
```

4 *Returns:* `x.date() < y`.

```
template<class Duration>
  constexpr bool operator<(const sys_time<Duration>& x, const leap& y) noexcept;
```

5 *Returns:* `x < y.date()`.

```
template<class Duration>
  constexpr bool operator>(const leap& x, const sys_time<Duration>& y) noexcept;
```

6 *Returns:* `y < x`.

```
template<class Duration>
  constexpr bool operator>(const sys_time<Duration>& x, const leap& y) noexcept;
```

7 *Returns:* `y < x`.

```
template<class Duration>
  constexpr bool operator<=(const leap& x, const sys_time<Duration>& y) noexcept;
```

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

```
template<class Duration>
  constexpr bool operator<=(const sys_time<Duration>& x, const leap& y) noexcept;
```

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

```
template<class Duration>
  constexpr bool operator>=(const leap& x, const sys_time<Duration>& y) noexcept;
```

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

```
template<class Duration>
  constexpr bool operator>=(const sys_time<Duration>& x, const leap& y) noexcept;
```

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

```
template<three_way_comparable_with<sys_seconds> Duration>
  constexpr auto operator<=>(const leap& x, const sys_time<Duration>& y) noexcept;
```

12 *Returns:* `x.date() <=> y`.

27.11.9 Class `link` [time.zone.link]

27.11.9.1 Overview [time.zone.link.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;
  };
}
```

1 A `link` specifies an alternative name for a `time_zone`. `link`s are constructed when the time zone database is initialized.

27.11.9.2 Member functions [time.zone.link.members]

```
string_view name() const noexcept;
```

1 *Returns:* The alternative name for the time zone.

```
string_view target() 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 [time.zone.link.nonmembers]

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

1 *Returns:* `x.name() == y.name()`.

```
strong_ordering operator<=>(const link& x, const link& y) noexcept;
```

2 *Returns:* `x.name() <=> y.name()`.

27.12 Formatting [time.format]

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:

chrono-format-spec:
fill-and-align$_{opt}$ *width*$_{opt}$ *precision*$_{opt}$ *chrono-specs*$_{opt}$

chrono-specs:
conversion-spec
chrono-specs conversion-spec
chrono-specs literal-char

literal-char:
any character other than `{` or `}`

conversion-spec:
`%` *modifier*$_{opt}$ *type*

modifier: one of
`E O`

type: one of
`a A b B c C d D e F g G h H I j m M n`
`p r R S t 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 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 [tab:time.format.spec]

Specifier	Replacement
`%a`	The locale's abbreviated weekday name. If the value does not contain a valid weekday, an exception of type `format_error` is thrown.
`%A`	The locale's full weekday name. If the value does not contain a valid weekday, an exception of type `format_error` is thrown.

Table 88: Meaning of conversion specifiers (continued)

Specifier	Replacement
%b	The locale's abbreviated month name. If the value does not contain a valid month, an exception of type format_error is thrown.
%B	The locale's full month name. If the value does not contain a valid month, an exception of type format_error is thrown.
%c	The locale's date and time representation. The modified command %Ec produces the locale's alternate date and time representation.
%C	The year divided by 100 using floored division. If the result is a single decimal digit, it is prefixed with 0. The modified command %EC produces the locale's alternative representation of the century.
%d	The day of month as a decimal number. If the result is a single decimal digit, it is prefixed with 0. The modified command %Od produces the locale's alternative representation.
%D	Equivalent to %m/%d/%y.
%e	The day of month as a decimal number. If the result is a single decimal digit, it is prefixed with a space. The modified command %Oe produces the locale's alternative representation.
%F	Equivalent to %Y-%m-%d.
%g	The last two decimal digits of the ISO week-based year. If the result is a single digit it is prefixed by 0.
%G	The ISO week-based year as a decimal number. If the result is less than four digits it is left-padded with 0 to four digits.
%h	Equivalent to %b.
%H	The hour (24-hour clock) as a decimal number. If the result is a single digit, it is prefixed with 0. The modified command %OH produces the locale's alternative representation.
%I	The hour (12-hour clock) as a decimal number. If the result is a single digit, it is prefixed with 0. The modified command %OI produces the locale's alternative representation.
%j	The day of the year as a decimal number. Jan 1 is 001. If the result is less than three digits, it is left-padded with 0 to three digits.
%m	The month as a decimal number. Jan is 01. If the result is a single digit, it is prefixed with 0. The modified command %Om produces the locale's alternative representation.
%M	The minute as a decimal number. If the result is a single digit, it is prefixed with 0. The modified command %OM produces the locale's alternative representation.
%n	A new-line character.
%p	The locale's equivalent of the AM/PM designations associated with a 12-hour clock.
%q	The duration's unit suffix as specified in 27.5.10.
%Q	The duration's numeric value (as if extracted via .count()).
%r	The locale's 12-hour clock time.
%R	Equivalent to %H:%M.
%S	Seconds as a decimal number. If the number of seconds is less than 10, the result is prefixed with 0. If the precision of the input 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 %OS produces the locale's alternative representation.
%t	A horizontal-tab character.
%T	Equivalent to %H:%M:%S.
%u	The ISO weekday as a decimal number (1-7), where Monday is 1. The modified command %Ou produces the locale's alternative representation.
%U	The week number of the year as a decimal number. The first Sunday of the year is the first day of week 01. Days of the same year prior to that are in week 00. If the result is a single digit, it is prefixed with 0. The modified command %OU produces the locale's alternative representation.

Table 88: Meaning of conversion specifiers (continued)

Specifier	Replacement
%V	The ISO week-based week number as a decimal number. If the result is a single digit, it is prefixed with 0. The modified command %OV produces the locale's alternative representation.
%w	The weekday as a decimal number (0-6), where Sunday is 0. The modified command %Ow produces the locale's alternative representation.
%W	The week number of the year as a decimal number. The first Monday of the year is the first day of week 01. Days of the same year prior to that are in week 00. If the result is a single digit, it is prefixed with 0. The modified command %OW produces the locale's alternative representation.
%x	The locale's date representation. The modified command %Ex produces the locale's alternate date representation.
%X	The locale's time representation. The modified command %EX produces the locale's alternate time representation.
%y	The last two decimal digits of the year. If the result is a single digit it is prefixed by 0.
%Y	The year as a decimal number. If the result is less than four digits it is left-padded with 0 to four digits.
%z	The offset from UTC in the ISO 8601 format. For example -0430 refers to 4 hours 30 minutes behind UTC. If the offset is zero, +0000 is used. The modified commands %Ez and %Oz insert a : between the hours and minutes: -04:30. If the offset information is not available, an exception of type format_error is thrown.
%Z	The time zone abbreviation. If the time zone abbreviation is not available, an exception of type format_error is thrown.
%%	A % character.

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*]

```
template<class Duration, class charT>
  struct formatter<chrono::sys_time<Duration>, charT>;
```

5 *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.

```
template<class Duration, class charT>
  struct formatter<chrono::utc_time<Duration>, charT>;
```

6 *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").

```
template<class Duration, class charT>
  struct formatter<chrono::tai_time<Duration>, charT>;
```

7 *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})
```

```
template<class Duration, class charT>
  struct formatter<chrono::gps_time<Duration>, charT>;
```

8 *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

```
      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>;
```

9 *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. The date and time formatted are equivalent to those formatted by a `sys_time` initialized with `clock_cast<system_clock>(t)`, or by a `utc_time` initialized with `clock_cast<utc_clock>(t)`, where `t` is the first argument to `format`.

```
template<class Duration, class charT>
  struct formatter<chrono::local_time<Duration>, charT>;
```

10 *Remarks:* If `%Z`, `%z`, or a modified version of `%z` is used, an exception of type `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;                                   // exposition only
  };

template<class Duration>
  local-time-format-t<Duration>
    local_time_format(local_time<Duration> time, const string* abbrev = nullptr,
                      const seconds* offset_sec = nullptr);
```

11 *Returns:* `{time, abbrev, offset_sec}`.

```
template<class Duration, class charT>
  struct formatter<chrono::local-time-format-t<Duration>, charT>;
```

12 Let `f` be a *local-time-format-t*`<Duration>` object passed to `formatter::format`.

13 *Remarks:* If `%Z` is used, it is replaced with `*f.abbrev` if `f.abbrev` is not a null pointer value. If `%Z` is used and `f.abbrev` is a null pointer value, an exception of type `format_error` is thrown. If `%z` (or a modified variant of `%z`) is used, it is formatted with the value of `*f.offset_sec` if `f.offset_sec` is not a null pointer value. If `%z` (or a modified variant of `%z`) is used and `f.offset_sec` is a null pointer value, then an exception of type `format_error` is thrown.

```
  template<class Duration, class TimeZonePtr, 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);
  };

template<class FormatContext>
  typename FormatContext::iterator
    format(const chrono::zoned_time<Duration, TimeZonePtr>& tp, FormatContext& ctx);
```

14 *Effects:* Equivalent to:

```
      sys_info info = tp.get_info();
      return formatter<chrono::local-time-format-t<Duration>, charT>::
               format({tp.get_local_time(), &info.abbrev, &info.offset}, ctx);
```

27.13 Parsing [time.parse]

1 Each `parse` overload specified in this subclause calls `from_stream` unqualified, so as to enable argument dependent lookup (**??**).

```
template<class charT, class traits, class Alloc, class Parsable>
  unspecified
    parse(const basic_string<charT, traits, Alloc>& fmt, Parsable& tp);
```

2 ~~*Remarks:*~~ *Constraints:* ~~This function shall not participate in overload resolution unless~~

```
      from_stream(declval<basic_istream<charT, traits>&>(), fmt.c_str(), tp)
```

is a valid expression.

3 *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);
```

4 ~~*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.

5 *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);
```

6

~~*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.

7 *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);
```

8 ~~*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.

9 *Returns:* A manipulator that, when extracted from a `basic_istream<charT, traits> is`, calls `from_stream(is, fmt.c_str(), tp, addressof(abbrev), &offset)`.

10 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.

11 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 89: Meaning of `parse` flags [tab:time.parse.spec]

Flag	Parsed value
`%a`	The locale's full or abbreviated case-insensitive weekday name.
`%A`	Equivalent to `%a`.
`%b`	The locale's full or abbreviated case-insensitive month name.
`%B`	Equivalent to `%b`.

Table 89: Meaning of `parse` flags (continued)

Flag	Parsed value
`%c`	The locale's date and time representation. The modified command `%Ec` interprets the locale's alternate date and time representation.
`%C`	The century as a decimal number. The modified command `%`*N*`C` 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.
`%d`	The day of the month as a decimal number. The modified command `%`*N*`d` 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.
`%D`	Equivalent to `%m/%d/%y`.
`%e`	Equivalent to `%d` and can be modified like `%d`.
`%F`	Equivalent to `%Y-%m-%d`. If modified with a width *N*, the width is applied to only `%Y`.
`%g`	The last two decimal digits of the ISO week-based year. The modified command `%`*N*`g` specifies the maximum number of characters to read. If *N* is not specified, the default is 2. Leading zeroes are permitted but not required.
`%G`	The ISO week-based year as a decimal number. The modified command `%`*N*`G` specifies the maximum number of characters to read. If *N* is not specified, the default is 4. Leading zeroes are permitted but not required.
`%h`	Equivalent to `%b`.
`%H`	The hour (24-hour clock) as a decimal number. The modified command `%`*N*`H` 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.
`%I`	The hour (12-hour clock) as a decimal number. The modified command `%`*N*`I` specifies the maximum number of characters to read. If *N* is not specified, the default is 2. Leading zeroes are permitted but not required.
`%j`	The day of the year as a decimal number. Jan 1 is `1`. The modified command `%`*N*`j` specifies the maximum number of characters to read. If *N* is not specified, the default is 3. Leading zeroes are permitted but not required.
`%m`	The month as a decimal number. Jan is `1`. The modified command `%`*N*`m` 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.
`%M`	The minutes as a decimal number. The modified command `%`*N*`M` 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.
`%n`	Matches one white space character. [*Note*: `%n`, `%t`, and a space can be combined to match a wide range of white-space patterns. For example, `"%n "` matches one or more white space characters, and `"%n%t%t"` matches one to three white space characters. — *end note*]
`%p`	The locale's equivalent of the AM/PM designations associated with a 12-hour clock. The command `%I` must precede `%p` in the format string.
`%r`	The locale's 12-hour clock time.
`%R`	Equivalent to `%H:%M`.
`%S`	The seconds as a decimal number. The modified command `%`*N*`S` 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.
`%t`	Matches zero or one white space characters.
`%T`	Equivalent to `%H:%M:%S`.

Table 89: Meaning of `parse` flags (continued)

Flag	Parsed value
`%u`	The ISO weekday as a decimal number (`1`-`7`), where Monday is `1`. The modified command `%`*N*`u` 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 `%Ou` interprets the locale's alternative representation.
`%U`	The week number of the year as a decimal number. The first Sunday of the year is the first day of week `01`. Days of the same year prior to that are in week `00`. The modified command `%`*N*`U` specifies the maximum number of characters to read. If *N* is not specified, the default is 2. Leading zeroes are permitted but not required.
`%V`	The ISO week-based week number as a decimal number. The modified command `%`*N*`V` specifies the maximum number of characters to read. If *N* is not specified, the default is 2. Leading zeroes are permitted but not required.
`%w`	The weekday as a decimal number (`0`-`6`), where Sunday is `0`. The modified command `%`*N*`w` 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 `%Ow` interprets the locale's alternative representation.
`%W`	The week number of the year as a decimal number. The first Monday of the year is the first day of week `01`. Days of the same year prior to that are in week `00`. The modified command `%`*N*`W` specifies the maximum number of characters to read. If *N* is not specified, the default is 2. Leading zeroes are permitted but not required.
`%x`	The locale's date representation. The modified command `%Ex` interprets the locale's alternate date representation.
`%X`	The locale's time representation. The modified command `%EX` interprets the locale's alternate time representation.
`%y`	The last two decimal digits of the year. If the century is not otherwise specified (e.g. with `%C`), values in the range `[69, 99]` are presumed to refer to the years 1969 to 1999, and values in the range `[00, 68]` are presumed to refer to the years 2000 to 2068. The modified command `%`*N*`y` 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 `%Oy` interpret the locale's alternative representation.
`%Y`	The year as a decimal number. The modified command `%`*N*`Y` 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.
`%z`	The offset from UTC in the format `[+|-]hh[mm]`. For example `-0430` refers to 4 hours 30 minutes behind UTC, and `04` refers to 4 hours ahead of UTC. The modified commands `%Ez` and `%Oz` parse a `:` between the hours and minutes and render leading zeroes on the hour field optional: `[+|-]h[h][:mm]`. For example `-04:30` refers to 4 hours 30 minutes behind UTC, and `4` refers to 4 hours ahead of UTC.
`%Z`	The time zone abbreviation or name. A single word is parsed. This word can only contain characters from the basic source character set (**??**) that are alphanumeric, or one of `'_'`, `'/'`, `'-'`, or `'+'`.
`%%`	A `%` character is extracted.

27.14 Header <ctime> synopsis [ctime.syn]

```
#define NULL see ??
#define CLOCKS_PER_SEC see below
#define TIME_UTC see below

namespace std {
  using size_t = see ??;
  using clock_t = see below;
  using time_t = see below;

  struct timespec;
  struct tm;
```

```
  clock_t clock();
  double difftime(time_t time1, time_t time0);
  time_t mktime(struct tm* timeptr);
  time_t time(time_t* timer);
  int timespec_get(timespec* ts, int base);
  char* asctime(const struct tm* timeptr);
  char* ctime(const time_t* timer);
  struct tm* gmtime(const time_t* timer);
  struct tm* localtime(const time_t* timer);
  size_t strftime(char* s, size_t maxsize, const char* format, const struct tm* timeptr);
}
```

1 The contents of the header `<ctime>` are the same as the C standard library header `<time.h>`.[259]

2 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`, `G`, `h`, `r`, `R`, `t`, `T`, `u`, `V`, and `z`, and the modifiers `E` and `O`.