N1900=05-0160 Proposal to Add Date-Time to the C++ Standard Library 0.75

Jeff Garland (jeff-at-crystalclearsoftware.com)

Table of Contents

Table of Contents

Motivation and Scope

The representation of dates and times is a recurring problem for software developers. Almost all non-trivial programs have the need to represent dates or times for one or more purposes. This proposal describes additions and changes to the C++ standard library to facilitate programming with dates and times. Most elements of the proposal are currently implemented in the Boost Date-Time Library [http://www.boost.org/libs/date_time/index.html].

Target Audience

The primary audience for this proposal is C++ application and library developers. The proposal provides a foundational library to simplify application development that needs to manipulate dates and times. In addition, by providing efficient value types, like `date`, the library provides an excellent foundation for building higher level interfaces. For example, a socket library that provides a timeout value can use the `milliseconds` type to provide a cleaner interface than is possible with a primitive type.

Due to the pervasive need for date-time programming, many libraries have been written to support date-time programming. Unfortunately, the date-time domain appears simple and trivial, but is deceptively complex. Thus, many libraries fail to address a broad range of applications by failing to solve some of the difficult date-time problems, or, providing inflexible and inefficient implementations. For example, very few libraries support flexible localized input-output capabilities. The lack of more complete standard library support means the re-invention of various date-time capabilities and a lack of portable capabilities to support C++ programmers. The Boost Date-Time Library [http://www.boost.org/libs/date_time/index.html] and this proposal are both attempts to remedy this condition by building on other standards (such as ISO 8601 and Posix 1003.1) and building a best of breed solution to support a broad range of applications.

Applications and General Functionality

A wide variety of modern software applications need to manipulate dates and times. Like numeric values, the span of application types that use dates and times include business, scientific, communications, and many others. The following are some fairly typical uses of dates and times in applications:

- recording business transaction dates
- presenting calendars and schedules
- calculation of elapsed times
- logging time of an event
- creating a schedule or plan for one or more activities
- calculate times in multiple time zones
- finding the date of an event (eg: third Thursday in March)

Consider an Estimate class that records an estimate for services rendered. To design this class at least two temporal values need to be recorded:

- the day the estimate is made, and
- the number of days the estimate is valid

One core function of the Estimate class is to determine if the estimate is still valid. The following is a sketch of how this logic might be implemented using the Boost Date-Time Library [http://www.boost.org/libs/date_time/index.html]:

```
//Example usage of a date temporal types
class Estimate {
 //...
 bool is_valid() const
 {
    date_period valid_period(date_of_estimate, valid_days);
    date today = day_clock::local_time(); //read the computer clock
    return valid_period.contains(today);
 }
 date last_valid_day() const
 {
    return (date_of_estimate + valid_days);
 }
private:
  std::string estimate_id;
  date        date_of_estimate;
  days        valid_days;  //number of days estimate is valid
  //...
};
```

In this small example, date, days, and date_period demonstrate the three core temporal types: time points, durations, and time periods. These are described in more detail in the concepts section.

Main Capabilities

Overall, the main capabilities needed for a standard date-time library include:

- Provide temporal types for representing dates and times
- Support calculation with dates and times
- Measurement of times from clocks
- Input and output of dates and times
- Local time adjustments

Temporal Types

Just as programmers are provided with `int` and `double` for development of programs involving basic mathematics, temporal types provide concise representation and calculation with dates and times. Clearly defined temporal types enhance coding practice by facilitating the creation of more precise functional interfaces. As with other value types, programmers expect temporal types to support basic value concepts such as assignability, comparability, and streamability.

The temporal type concepts are described in Points, Durations, and Periods. The list of temporal types provided in the proposal can be found in Summary of Temporal Types.

Calculation with Dates and Times

Like `int` and `double` the temporal types provide the framework for time-based arithmetic. The following are some examples:

```
date d(2004,Jan,1);
d += year(1);
d += months(3);
d += days(10);

date_time t(d, hours(5));
t+= minutes(3) - seconds(2);

milliseconds ms_count = hours(3) + milliseconds(100);
```

The design of these features is described in more detail in Arithmetic Operations on Temporal Types.

Measurement of Times From Clocks

Computer applications frequently need to determine the current time or date. To determine the time, applications read a hardware device that provides a representation of the current time. Usually, this representation is a counter that represents a duration offset from a well defined epoch time. For example, `std::gmtime` is a clock interface that retrieves the number of seconds since `1970-Jan-1 00:00:00` (the epoch) from the local computer clock. The `gmtime` call provides the time based on the standard Universal Coordinated Time (UTC). Note that many of the clock APIs also embed the concept of local time adjustment. This typically depends on the time zone settings of the computer.

Reading a clock results in the construction of a time point. For example:

```
//construct UTC time based
date_time t1 = clock::universal_time();

//construct localized time based on time zone setting of computer
date d = day_clock::local_time();
date_time t2 = second_clock::local_time();
date_time t3 = microsecond_clock::local_time();


//construct localized time based on time zone specification
posix_time_zone tz("EST-05EDT,M4.1.0,M10.5.0");
date_time t4 = second_clock::local_time(tz);
```

Input and Output

People have invented a seemingly infinite set of combinations for representing dates and times, making good input output support quite challenging. The ISO 8601 standard [http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=26780] provides a specification for formatting of dates and times. However, most applications have requirements that extend beyond ISO 8601 including the need for localized and custom formats. Some of these requirements include the need to support customized strings for elements of a time representation such as the month name. To support this variety of representations requires a relatively sophisticated input/output capability. The Boost Date-Time Library [http://www.boost.org/libs/date_time/index.html] provides a set of facets that use format strings and interoperate with the temporal types. Based on the current C++ formatting facets and integrated with standard streams they provide for customization of all aspects of input and output. For example:

```
date d(2005,Jun,25);
cout << pt << endl; // "2005-Jun-25"
//example to customize output to be "LongWeekday LongMonthname day, year"
//                                  "%A %b %d, %Y"
date_facet* facet(new date_facet("%A %B %d, %Y"));
cout.imbue(locale(cout.getloc(), facet));
cout << d << endl;
// "Saturday June 25, 2005"

stringstream ss;
ss.str("Saturday June 25, 2005");
date_input_facet* input_facet(new date_input_facet("%A %B %d, %Y");
ss.imbue(locale(ss.getloc(), input_facet));
date d2; //not_a_date_time
ss >> d2;
```

The section Extensions to time_put Format Flags describes the additional formatting flags provided in this proposal. The new flags support concepts like fractional seconds formatting which are needed to support new concepts provided by this proposal. The classes `date_facet`, `time_facet`, `date_input_facet`, and `time_input_facet` provide support for localization and customization of input/output.

Local Time Adjustments

Local time adjustment is an example of particularly difficult domain logic that is simplified by the proposed library. The rules associated with these adjustments are needed in many applications and overall the library support is quite poor. Consider, for example, an application that calculates arrival times for an airline. For the end user, it is important to see the arrival time of the flight in the arriving time zone not the departing time zone. Thus the application needs to manage the logic associated with time zone transitions and daylight savings time transitions. Other than that, the core application comes down to adding the length of the flight to the starting time of the flight.

One core element of the local time adjustment is the representation of the time zone. A time zone is quite complex to specify completely since it needs to include everything from output strings to rules that define the start and end of daylight savings time. The most broadly used solution to representing time zones is provided by the POSIX 1003.1 standard for representation of timezones. The `posix_time_zone` class provides direct support for this standard.

Here's a snippet of the core of this application as written using the proposed library extension including `posix_time_zone`:

```
//Red-eye flight from Arizona to New York that transits over
//a daylight savings time (DST) transition boundary.  Az
//doesn't ever shift to DST while New York does.  Thus at
//2:00 (in the middle of the flight the eastern US shifts
//it's clocks forward an hour.  Luckily the application
//developer doesn't need to know these details.

//setup the timezones
posix_time_zone nyc_tz("EST-05EDT,M4.1.0,M10.5.0");
posix_time_zone phx_tz("MST-07:00:00");

//construct the departure time in phx local time
date_time departure_time_phx(date(2004, Oct, 30), hours(23));

//convert to utc time
date_time departure_time_utc = departure_time_phx.to_utc(phx_tz);

//calculate the arrival time in utc
minutes flight_length = hours(4) + minutes(30);
date_time arrival_time_utc = departure_time_utc + flight_length;

//now adjust the arrival time which is in the Phoenix timezone to NY
date_time arrival_time_nyc = arrival_time.to_local(nyc_tz);
```

Additional Capabilities

Some other significant capabilities that simplify programming with dates and times include:

- Calendrical algorithms
- Compatibility with standard library
- Flexible time resolutions
- Special value support

The following sections describe these capabilities in more detail.

Calendrical Algorithms

Calendrical algorithms, or date generators, are tools for generating other dates or schedules of dates. These generator algorithms allow the representation of concepts such as "The first Sunday in February". They are useful in performing tasks such as calculating holidays. In this proposal these algorithms are incorporated into the interface of the date class. For example, the following constructors incorporate these algorithms:

```
//Construct the date for the Last Sunday in January
```

```
date d1(2004, Jan, Last_Week, Sunday);

//Construct a date for the Sunday in the 50th week of the year
date d2(2004, 50, Sunday);
```

Some additional algorithms included in the date class allow for calendar navigation and calculations. For example:

```
date d2 = d1.next_weekday(Tuesday);
date d3 = d1.previous_weekday(Friday);

days day_count1 = d1.days_until_weekday(Thursday);
days day_count2 = d1.days_before_weekday(Monday);
```

Compatibility with Standard Library

Compatibility with the current standard library is an important requirement for any new addition to the library. In this proposal there are several aspect of compatibility including:

- compatibility with collections
- compatibility with current input-output library
- compatibility with C types `time_t` and `tm`
- build on current capabilities where possible

Since the temporal types in this proposal are value types, they can be used as values in standard library collection classes. In addition, since the temporal types support comparison operations they can be used as a key in a map or as an element of a set.

Input and output using the standard library streaming and facet capabilities is another key aspect of standard library compatibility.

The proposal integrates with `time_t` and `tm` by allowing dates and times to construct from these types.

Flexible Time Resolutions

Some of the most difficult issues with programming dates and times is selecting an appropriate epoch, resolution, and size of internal representation. The choices are classic time and space tradeoffs. For this reason, providing a framework for users to provide customization of these elements greatly expands the utility of the library.

This is discussed in more detail in Underlying representations for dates and times.

Special Value Support

Special values provide the ability to represent 'logical values' with the various temporal types. For example, the ability to represent not-a-date-time or infinities is helpful in many circumstances. This is similar to how floating point types have values for not-a-number (NAN) and infinity. Infinities are useful in applications that need to represent concepts like 'a-long-time-ago' or 'forever'.

Special values can apply to both time points and time durations and thus alter the rules of calculation. Here are some code examples using special values:

```
date d; //default construct to not_a_date_time
```

```
if (d.as_special() == NOT_A_DATE_TIME) { //true
  //...
}

date_time inf(POSITIVE_INFINITY);
date_time t(date(2005, Jan, 1), hours(3));
if (t < inf) { //always true unless t == positive infinity
```

Related Capabilities Not Included

There are other date-time related functionalities that are not included in this proposal. These decisions are primarily a reflection of the author's experience of useful capabilities in the domain, while keeping the size of the proposal manageable. If a consensus of the committee believes any of these items should be incorporated, the author is willing to incorporate them.

These include:

- Recurring Intervals
- Timezone Database
- Local Time Types
- Full support for leap seconds
- Iterators in time
- Timers

Recurring Intervals

Recurring intervals are a form of generator that are useful in scheduling and other applications. A recurring interval is typically defined by an initial interval and a recurrence duration. Conceptually this is the idea of "schedule the meeting for 9-10 every Monday starting on July 11, 2004". There are many options and variations on this theme. While these could be included the use of recurring intervals is somewhat specialized and can be built on top of the existing foundation.

Timezone Database

The timezone database capabilities of Boost Date-Time Library [http://www.boost.org/libs/date_time/index.html] are highly useful. The primary capability is to support construction of time zones from a regional specification as follows:

```
tz_database tz_db;
tz_db.load_from_file("date_time_zonespec.csv");

boost::shared_ptr<time_zone_base> nyc_tz =
       tz_db.time_zone_from_region("America/New_York");
boost::shared_ptr<time_zone_base> phx_tz =
       tz_db.time_zone_from_region("America/Phoenix");
```

However, this capability introduces the issue of maintaining a set of data associated with the world timezones. This data changes frequently as governments change time zone rules. The management of the data makes this feature inappropriate for standardization.

Local Time Types

Boost Date-Time Library [http://www.boost.org/libs/date_time/index.html] has a Time Point type that holds a time zone as well as the time value. While useful, this class is not included in the proposal as it adds significant complexity to the design. For example, a dependency on shared_ptr is introduced to allow for efficient management of the time zone types.

In addition, the overall benefit of this integrated class is somewhat limited. All the local time conversions and other desired operations can be performed without this class.

Iterators in Time

Iterators provide the ability to generate a set of dates or times based on some starting conditions and an ending point. Based on the foundation of the core temporal types bi-directional iterators can be provided that enable the simple generation of calendars and other time-related constructs.

```
//print a series of dates
day_iterator start(date(2005,Jul,7));
day_iterator end(date(2005, Jul, 10));
std::copy(start, end, std::ostream_iterator<date>(std::cout, " \n"));

//make a list of 3 dates
std::list<date> dl;
std::copy(start, end, std::back_inserter(dl));
```

The Boost Date-Time Library [http://www.boost.org/libs/date_time/index.html] provides the following iterator types: `day_iterator`, `week_iterator`, `month_iterator`, `year_iterator`, `time_iterator`, and `local_time_iterator`.

Timers

There are two primary types of timers:

- Elapsed Timers
- Countdown Timers

Elapsed timers are useful for measuring the duration of activities. For example:

```
//micro timer reads clock at microsecond resolution
micro_timer mt; //automatically starts timer
cout << mt.elapsed() << endl;
sleep(1);
//elapsed will be about a second - something like: 00:00:01.000123
cout << mt.elapsed() << endl;
sleep(1);
st.pause(); //stop timing
//...
st.resume(); //continue timing
```

Countdown timers provide the opposite interface from elapsed timers. Starting with a fixed time duration, they gradually subtract time until they reach zero. For example, measuring the time left in a basketball game is an application for a countdown timer.

Concepts Overview

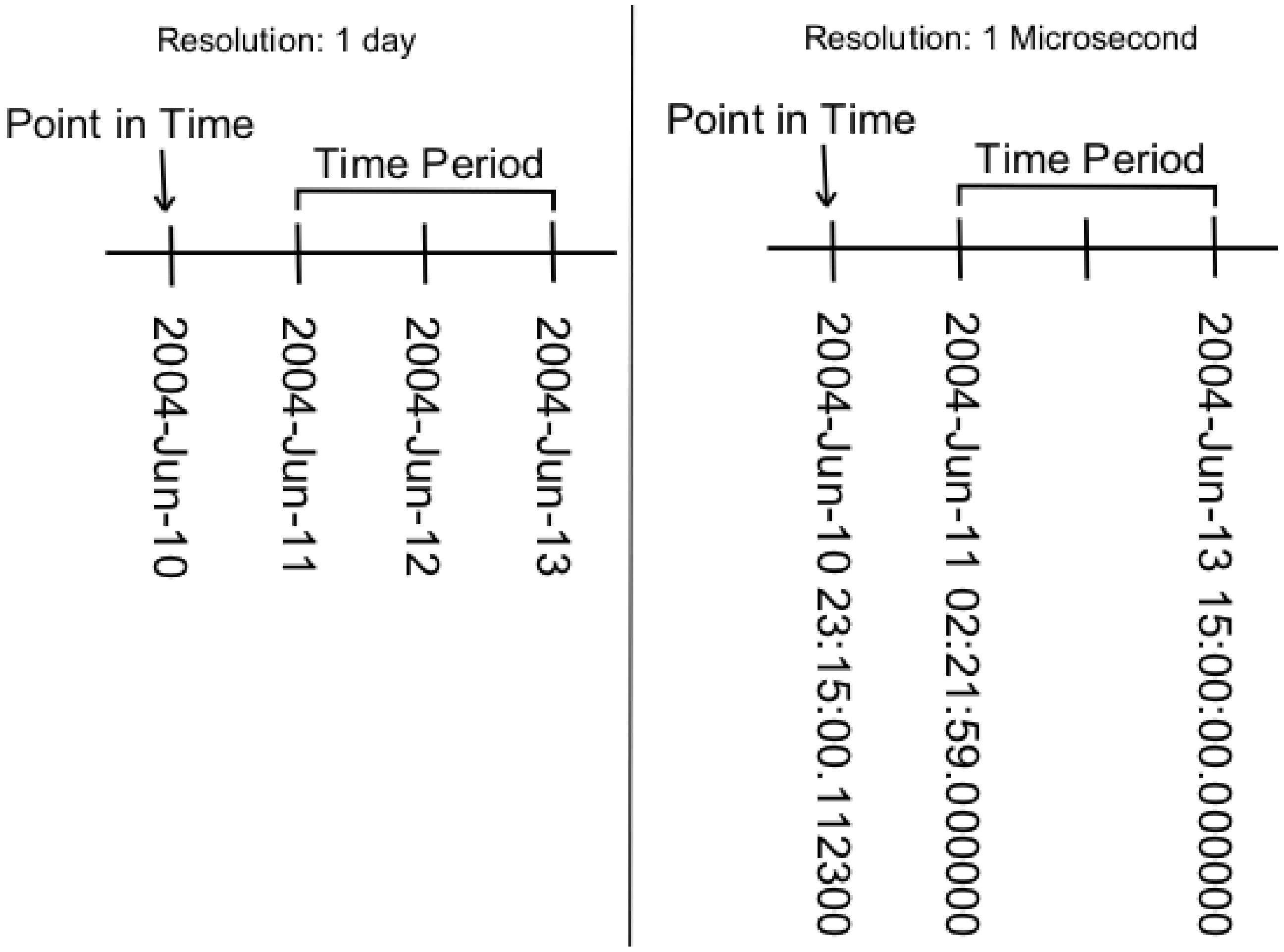
The date time domain is rich in terminology and problems. The following is a brief introduction to the concepts reflected in this proposal.

Points, Durations and Periods

There are 3 basic concepts that serve as the foundation for the representing times and dates:

- Time Point - an instant in the time continuum (dimensionless)
- Time Duration - a length of time unattached to a any time point
- Time Period - a length of time between two time points

as illustrated in the figure below.



Durations can be used to measure how long something takes. Consider the following:

- the meeting took 2 hours
- the flight will take 45 minutes
- the program execution took 1.0020 seconds

- the congress took 4 days to pass the bill
- current time is -02:25:14 from launch of the rocket
- timeout should fire in 20 milli-seconds

The examples above illustrate how resolution plays into durations. The resolution of interest depends very much on the application domain. And a single application may make broad use of many different resolutions. Also note that durations can have negative or positive values. This is useful in many applications and fits into the calculation rules described later.

Time points describe when an event has or will occur. Consider the following:

- the meeting will be on June 10, 2004
- the baby was born on June 10, 2004 at 1:00 pm
- the rocket will launch on 2004-Jun-10 15:00:25.030 EST

A single time point may be described by one or more 'labels' of varying precision. For example,

```
2004-Jun-10 15:00:00     == 2004-Jun-10 3 pm
//UTC is Universal Time Coordinated - EDT is Eastern Daylight Time
2004-Jun-10 15:00:00 UTC == 2004-Jun-10 19:00:00 EDT
```

Note that these labels belie the complexities of what time they actually represent. All these examples assume we are using a gregorian calendar (discussed more later). In addition, the first example puts aside the issue of local time adjustments by leaving the time zone unspecified.

Time periods express a range of time. Consider the following:

- the meeting will start on June 10, 3 pm and last 2 hours
- the trip will start June 10th and last 4 days
- the rocket burn went from 2004-Jun-10 15:00:00.0030 to 2004-Jun-10 15:00:01.0050
- we will arrive between June 1st and June 3rd.

These examples of time periods illustrate that a time period is simply a starting time and an ending time or a starting time and a time duration. Periods are essential to the simplification of logic associated with planning and scheduling systems. To accomplish this, periods can be shifted, intersected, merged, and combined in various ways. In addition, they can be tested for adjacency, containment, and relative location.

These same concepts are discussed and reflected in "Patterns for things that change with time" [http://martinfowler.com/ap2/timeNarrative.html] by Martin Fowler. More recently, these same concepts have also been recognized in the ISO 8601 standard [http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=26780] for representations of dates and times. Unlike dimensionless numbers, these different concepts serve different roles in application development. My personal background has included the development of a planning and scheduling system used to schedule thousands of daily events for a major satellite phone system. This experience, as well as development of other applications, led me to many of the same concepts described by Fowler.

Resolution

Each of these temporal types has a 'Resolution' which is defined by the smallest representable duration. Thus, a 'date' is a 'time point' with a resolution of 1 day. For representing a 'time' the resolution of the duration must increased. For example, time_t is a time-point with a resolution of 1 second. The ptime [http://www.boost.org/doc/html/date_time/posix_time.html#date_time.posix_time.ptime_class] class in the Boost Date-Time Library [http://www.boost.org/libs/date_time/index.html] has an adjustable resolution typically set at 1 micro-second or 1 nano-second.

The smallest duration representable within `date`, `days`, and `date_period` is one day. Thus these types have a **resolution** of one day. Other temporal types such as `hours` and `milliseconds` provide higher resolutions.

Calendar and Time Systems

Calendars describe the rules for for mapping the observable solar cycles into patterns such as days, weeks, months, and years. The Gregorian system, adopted in the sixteenth century, is the most widely used calendar system today. It defines the sequence of days and months in a year, as well as adjustments (such as leap years). The proposal uses a 'proleptic Gregorian' calendar which extends the Gregorian system back in time prior to it's adoption.

The ISO Calendar, defined in the ISO 8601 standard [http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=26780], maps to the Gregorian calendar, but has a unique technique for calculating week numbers. This proposal provides direct support for calculation of ISO week numbers. The ISO Week Data Calendar [http://personal.ecu.edu/mccartyr/isowdcal.html] illustrates the iso week information.

There are many other calendar systems such as the Chinese, Mayan, Islamic, and Hebrew. This proposal does not attempt to provide direct extensibility to all other calendar systems. However, the concepts reflected in the proposal can be applied to other calendar systems which differ in the details of how days are labeled and how concepts such as years and months are treated.

A Time system provides all these categories of temporal types as well as the rules for labeling and calculating with time points. These systems may include additional adjustment rules such as 'leap seconds'.

Epoch Time

For the calculation of dates and times it is convenient to calculate using a count of days, seconds, or some other time unit starting from a particular point in the time continuum. The starting point for this count is called the Epoch Time. The Epoch time for `time_t` is 1/1/1900 00:00:00.

Leap Seconds

Like leap years, leap seconds help keep a clock measured time in alignment with observed solar times. Leap years follow a regular pattern while leap seconds are declared as the need arises due to wobbles in the earth's rotation. The irregular nature of leap seconds makes them difficult to incorporate into computer based time systems.

More information on leap seconds is described at the US Navy web site [http://tycho.usno.navy.mil/leapsec.html].

Time Zones and Local Time

Most local time systems are based on Coordinated Universal Time (UTC) but are also adjusted for earth rotation so that daylight hours are similar everywhere. In addition, some local times include Daylight Savings Time (DST) adjustments to shift the daylight hours during the summer.

Coordinated Universal Time - UTC

UTC (Coordinated Universal Time) is a widely used standard based on the solar time at the Prime Meridian. Formerly known as Greenwich Mean Time (GMT) it is also often referred to as 'Zulu Time' by the military organizations.

UTC is adjusted for earth rotation at longitude 0 by the use of Leap Seconds.

Daylight Savings Time - DST

Daylight Savings Time (called Summer Time in Europe) adjusts the clock forward during summer months so that the 'active' hours match the hours of daylight. This practice became widespread during the twentieth century. Wikipedia provides details on the practice [http://en.wikipedia.org/wiki/Daylight_saving_time].

For computer programs, DST presents a challenge. In addition to the complex rules that define the start and end of DST, the 'jumping of times' creates

- ambiguous time labels
- invalid time labels

Consider the switch to daylight savings for the eastern time zone in the United States. For 2005 this happened on April 3rd at 02:00:00 local time. During this transition, the clock ticks from 01:59:59 to 03:00:00. Thus, the time labels from 02:00:00 to 02:59:59 are said to be invalid.

```
//2005-Apr-03 01:00:00 EST
//2005-Apr-03 01:30:00 EST
//2005-Apr-03 03:00:00 EDT
//2005-Apr-03 03:30:00 EDT
```

During the switch back from daylight savings time the problem of ambiguous time labels occurs. That is, a time without an adornment for daylight savings cannot be calculated to be clearly in, or out of, daylight savings time. Thus, in the eastern United States timezone, on the switch back from daylight savings, the times between 01:00:00 and 01:59:59 repeat twice.

```
//2005-Oct-30 00:30:00 EDT
//2005-Oct-30 01:00:00 EDT
//2005-Oct-30 01:30:00 EDT
//2005-Oct-30 01:00:00 EST
//2005-Oct-30 01:30:00 EST
//2005-Oct-30 02:00:00 EST
```

Clocks

A Clock Device is software component (tied to some hardware) that provides the current date or time with respect to a time system. A clock can measure the current time to a known resolution which may be higher, or lower, than a particular time representation.

Different software systems have different needs for software clocks. Many modern applications need high resolution, millisecond or higher, clock measurements. Modern computers routinely provide these higher resolution clocks. In addition, dedicated clock hardware often provides even higher level resolutions than typical computer hardware provides.

Impact on the Standard

This is a pure library proposal, it does not add any new language features. It does include changes to the existing facets to support enhanced date-time input and output.

Limitations of the Current C++ Standard

The current C/C++ standard library recognizes the need for tools to manipulate dates and times. Although useful, the types in the current standard are mostly derived from the C library and do not reflect modern practice.

Some of the key issues with the current capabilities include:

- limitations of the time ranges and resolution provide by `time_t`
- all functions limited to one second time resolution
- no distinction between durations and points in time
- lack of symmetry in `time_get` and `time_put` facets
- time range provided by `tm` is unspecified
- no way to represent only a date

The `time_t` type provided in the C standard has a beginning epoch of 1970-Jan-1 00:00:00. For any application that needs to represent times prior to 1970 `time_t` is insufficient. Because of this, simple applications like representing birthdays cannot be written using time_t. Overall, the core function of time_t is to serve as an interface to computer clocks. Unfortunately, with a resolution of one second it is insufficient for many modern applications which require fractional seconds to measure event times. These applications cannot use `time_t` or `tm` to represent times.

Another limitation with the current standard is that it does not provide elegant support for comparison and mathematical operations on time values. For example, consider the following 'typical' date manipulation code:

```
//only want to calculate with dates, but time_t
//doesn't support that...
time_t start_time(0);
start_time += 5*(60*60*24) + 2*(60*60*24*7);
```

A value object approach has some significant advantages in code clarity:

```
date d(1970, Jan, 1);
d += days(5) + weeks(2);
```

Clearly the first code example could be simplified by the use of constants and such, but the lack of a standard here means that unclear code is all to frequently the norm. Also, constants are insufficient because some time durations (months and years) are not fixed lengths of time. A month varies in length from 28 to 31 days and a year varies in length from 365 to 366 days. This issue is discussed in more detail in the section on Arithmetic Operations on Temporal Types.

The `tm` in the current standard structure is used largely for input-output using time_get and time_put. Unfortunately the `tm` structure provided is a bit ambiguous about the range of times supported. The field `tm.year` is defined to start at 1900. It is unclear if years prior to 1900 (negative values) are supported. Many applications need to support year representations prior to 1900 making this ambiguity an issue for writing portable output code based on `tm`.

Changes to the Current time_get and time_put Facets

This proposal includes provisions for advanced input and output of date-time values. To support this, additional formatting flags are introduced to the existing `time_put` facet. In addition, methods are added to the `time_get` facet to provide for format-based parsing of dates and times. This is described in more detail below.

One major issue for the library is to support symmetry of input and output operations. For example, it is reasonable to expect that a `date`, or other temporal type, can be output to a stream and then read back in. For example:

```
date d(2004, Jan, 1);
std::stringstream ss;
ss << d;  //calls `time_put` with format %Y-%b-%d
//ss.str() == "2004-Jan-01"
date d2;  //not_a_date_time
ss >> d2; //can't implement directly with current `time_get`
```

The current `time_get` facet does not currently provide full support for format-based input. This is a result of the `time_get` facet not providing the ability to set input formats while `time_put` provides for sophisticated formatting via format strings. Specifically note the signatures of `get_date` and `put`:

```
time_get<...
  iter_type get_date(iter_type s, iter_type end,
                     ios_base& f, ios_base::iostate& err,
                     tm* t) const;  //no format pattern here

time_put<...
  iter_type put(iter_type s, ios_base& f,
                char_type fill, const tm* tmb,
                const charT* pattern, const charT* pattern_end) const;
```

The `pattern` and `pattern_end` parameter provide for the flexible customization of date output. So, in essence, a new `date_get` is needed that supports the following signature:

```
time_get<...
  iter_type get_date(iter_type s, iter_type end,
                     ios_base& f, ios_base::iostate& err,
                     const charT* pattern, const charT* pat_end, //<-- added parameters
                     tm* t) const;
```

Note that the library working group already has recognized that the current state of these facets makes correct implementations difficult in [http://www.open-std.org/jtc1/sc22/wg21/docs/lwg-active.html#461 DR 461]. Adding format parameters should improve the implementability as has been demonstrated in the Boost Date-Time Library [http://www.boost.org/libs/date_time/index.html].

In addition to enhancements to support format-based input, this proposal expands the set of format flags to support additional concepts such as fractional seconds. These additional format flags are described further in Extensions to time_put Format Flags.

Additions to Clock Interfaces

The current standard provides for `std::localtime` and `std::gmtime` functions to retrieve locally adjusted and UTC times respectively. The resolution of these calls is one second. This proposal recommends providing equivalent functions that provide microsecond resolution. Many platforms have clocks supporting microsecond resolution. This higher resolution is important for many modern applications. Proprietary versions of these functions exist on common platforms. Finally, the higher resolution can be made optional if the platform does not support the higher resolution. The approach of the proposal to clocks allows for the addition other clock device implementations.

Other Standards and Related Work

Other C++ Libraries

There are so many date-time libraries for C++ and other languages it would be literally impossible to list them all here.

Some important examples include the Rogue Wave library, Recursion Software C++ Toolkit [http://www.recursionsw.com/cplus_documentation.html] (originally ObjectSpace.foundations) library, Microsoft Foundation Classes, IBM ICU.

Looking at many of these libraries there are lots of differences. However, some important recurring themes emerge:

- representation of dates only
- representation of dates and times
- representation of time durations
- support for time zones and local time adjustments

The following sections highlight some of the key classes and features of these libraries.

Rogue Wave

The Rogue Wave libraries have long provided support for temporal programming. The Rogue Wave libraries include several key types including:

- `RWDateTime` [http://www.roguewave.com/support/docs/sourcepro/toolsref/rwdatetime.html] - date and time to millisecond
- `RWZone` [http://www.roguewave.com/support/docs/sourcepro/toolsref/rwzone.html] - time zone abstraction
- `RWZoneSimple` [http://www.roguewave.com/support/docs/sourcepro/toolsref/rwzonesimple.html] - time zone class
- `RWDate` [http://www.roguewave.com/support/docs/sourcepro/toolsref/rwdate.html] - just a date
- `RWTimer` [http://www.roguewave.com/support/docs/sourcepro/toolsref/rwtimer.html] - second level resolution
- `RWClockTimer` [http://www.roguewave.com/support/docs/sourcepro/toolsref/rwclocktimer.html] - short interval timer (using clock limits to ~30 minutes on some platforms)

One significant feature of the Rogue Wave libraries is the representation of 'special values'. Called 'Sentinels' in the Rogue Wave library, they allow for the representation of special temporal values such as 'Not A Date Time' or 'Infinity'. This is similar to the special values concepts provided in this proposal.

Recursion Software Time<Toolkit>

The Recursion Software C++ Toolkit [http://www.recursionsw.com/cplus_documentation.html] library supports several key types including:

- `date` - a date
- `time` - a 24 hour time_duration to microsecond resolution
- `date_and_time` - date plus a time
- `timeperiod` - equivalent to time_duration concept in this proposal

- `stopwatch` - a timer
- `timezone` - encapsulation of daylight savings and UTC offset information

Microsoft Foundation Classes

The Microsoft Foundation Classes provide two primary classes for manipulation of dates and times: `CTime` [http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vclib/html/_MFC_CTime.asp] and `CTimeSpan` [http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vclib/html/_mfc_ctimespan.asp]. `CTime` is a point in time with a resolution of one second. `CTimeSpan` is a time duration. Essentially small value types that wrap the C library, these classes create a basic C++ interface for application developers.

ICU Library

The IBM International Components for Unicode (ICU) libraries have several date-time related capabilities. The ICU date-time user guide [http://icu.sourceforge.net/userguide/dateTime.html] provides more information on this C++ library. This library provides a class `UDate` to represent dates. In addition, ICU includes a TimeZone class [http://icu.sourceforge.net/userguide/dateTimezone.html] to provide for localization of dates.

Other Language Libraries

Most modern languages support a variety of types to support date-time programming.

JAVA Libraries

JAVA provides a number of different types for the representation of dates and times. These include

- Date [http://java.sun.com/j2se/1.5.0/docs/api/java/util/Date.html]
- Calendar [http://java.sun.com/j2se/1.5.0/docs/api/java/util/Calendar.html] <-- Gregorian Calendar [http://java.sun.com/j2se/1.5.0/docs/api/java/util/GregorianCalendar.html]
- TimeZone [http://java.sun.com/j2se/1.5.0/docs/api/java/util/TimeZone.html] <-- SimpleTimeZone [http://java.sun.com/j2se/1.5.0/docs/api/java/util/SimpleTimeZone.html]
- DateFormat [http://java.sun.com/j2se/1.5.0/docs/api/java/text/DateFormat.html] <-- SimpleDateFormat [http://java.sun.com/j2se/1.5.0/docs/api/java/text/SimpleDateFormat.html]

Confusingly, the JAVA Date [http://java.sun.com/j2se/1.5.0/docs/api/java/util/Date.html] actually represents a time to millisecond resolution. It serves as the primary temporal type provided by the JAVA foundation libraries. Calendar [http://java.sun.com/j2se/1.5.0/docs/api/java/util/Calendar.html] is the base class for Gregorian Calendar [http://java.sun.com/j2se/1.5.0/docs/api/java/util/GregorianCalendar.html] that provides arithmetic capabilities with times. Locale specific formatting and parsing is specified by DateFormat [http://java.sun.com/j2se/1.5.0/docs/api/java/text/DateFormat.html] and implemented using format flags in the implementation SimpleDateFormat [http://java.sun.com/j2se/1.5.0/docs/api/java/text/SimpleDateFormat.html]. TimeZone [http://java.sun.com/j2se/1.5.0/docs/api/java/util/TimeZone.html] provides an interface and SimpleTimeZone [http://java.sun.com/j2se/1.5.0/docs/api/java/util/SimpleTimeZone.html] provides an implementation for performing local time adjustments.

Python Libraries

The Python Date Time Module [http://www.python.org/doc/2.4.1/module-datetime.html] provides a number of different types for the representation of dates and times. These include

- date [http://www.python.org/doc/2.4.1/datetime-date.html] -- a date type
- datetime [http://www.python.org/doc/2.4.1/datetime-datetime.html] -- a date and time combined
- timedelta [http://www.python.org/doc/2.4.1/datetime-timedelta.html] -- a time duration type
- tzinfo [http://www.python.org/doc/2.4.1/datetime-tzinfo.html] - time zone information

The timedelta [http://www.python.org/doc/2.4.1/datetime-timedelta.html] class is an example of the time duration concept. datetime [http://www.python.org/doc/2.4.1/datetime-datetime.html] and date [http://www.python.org/doc/2.4.1/datetime-date.html] are time points that are similar to the `date_time` and `date` classes in this proposal.

ISO 8601 - Representation of Dates and Times

The ISO 8601 standard [http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=26780] defines an international standard for formatting of dates and times. Conceptually, ISO 8601 provides for the representation of lengths of time, points in time, and intervals of time at different resolutions. Most of the concepts of time representation in the ISO 8601 standard [http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=26780], with the exception of recurring intervals, are reflected in this proposal. In addition, the proposal provides support for input and output of dates and times as specified in ISO 8601.

ISO 8601 provides for two main format types: normal and extended. The normal representation represents a date or time using numeric values starting at the largest time division down. For example a date is represented as YYYYMM-DD where YYYY is a four digit specification of the year, MM is a two digit specification of a month and DD is a two digit specification of a day.

Some examples of Normal representation include:

```
20040301  is March 1, 2004
20040301T020559,01  is March 1, 2005 02:05:59.01
```

The extended form includes additional punctuation and thus is a bit easier for humans to read:

```
2004-03-01   is March 1, 2004
2004-03-01T02:05:59,01 is March 1, 2005 02:05:59.01
```

One nice feature of ISO 8601 is a consistent ordering of time components from larger to smaller. For example, years before months before days in the date.

This proposal provides direct support for iso 8601 formatting. For example, the facet classes provide for ISO 8601 input and output:

```
date d(2005,Jun,25);
date_facet* facet(new date_facet());
facet->set_iso_format();
cout.imbue(locale(cout.getloc(), facet));
cout << d << endl; //20050625
facet->set_iso_extended_format();
cout << d << endl; //2005-06-25
```

While ISO 8601 standard [http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=26780] provides an excellent foundation for one standard set of input and output approaches, there are limits. Specifically ISO 8601 does NOT provide for:

- using 'strings' for names of elements (eg: 'September' instead of 09)
- localization of date and time strings
- representation of time periods as open or closed ranges
- full specification of time zones

So while this proposal provide direct support for input and output in ISO formats, it provides a more sophisticated input-output capability overall.

POSIX 1003.1 Timezone Representation

The IEEE 1003.1 POSIX standard provides a textual specification for the representation of timezones. This is summarized as follow:

```
"std offset [dst [offset],start[/time],end[/time]]" (w/no spaces).
```

'std' specifies the abbreviating of the time zone name. '[' and ']' indicate optional fields. 'offset' is the offset from Universal Coordinated Time (UTC). 'dst' specifies the abbrev of the time zone during daylight savings time. The second offset is how many hours changed during Daylight Savings Time (DST). 'start' and 'end' are the dates when DST goes into, and out of, effect. 'offset' takes the form of:

```
[+|-]hh[:mm[:ss]] {h=0-23, m/s=0-59}
```

'time' and 'offset' take the same form. 'start' and 'end' can be one of three forms:

```
Mm.w.d {month=1-12, week=1-5 (5 is always last), day=0-6}
Jn {n=1-365 Feb29 is never counted}
n {n=0-365 Feb29 is counted in leap years}
```

The following is an example defining the timezone for the West Coast of the United States.

```
PST-08PDT,45/02,310/02
```

Breaking this down:

```
PST-08PDT,45/02,310/02
^^^   ^^^
 |     |
 |     When Daylight Savings Time (DST) is in effect the abbreviation is PDT (Pacific Daylight Tim
 |
defines PST as the normal abbreviation
```

```
PST-08PDT,45/02,310/02
   ^^^    ^^^^^ ^^^^^
    |       |      Switch back from DST on day 310 02:00:00
    |       |
    |       Switch to DST is day 45 at 02:00:00 hours
    |
    Non DST offset from Universal Coordinated Time (UTC) is -8 hours
```

The Posix Time Zone section describes the class `posix_time_zone` that implements these rules.

C Standard Library

The current C standard library, via header <time.h> [http://www-ccs.ucsd.edu/c/time.html], provides types and functions that form the basis of most current time manipulation programs. This includes types such as `time_t` and `tm`, as well as functions such as `gmtime`, `mktime`, and `strftime`.

The author is unaware of any formal proposals to the C standard committee for changes to date and time. However, David Tribble has a number of proposals that may be brought to the C committee at some point in the future.

- additional constraints on the time_t type [http://david.tribble.com/text/c0xtimet.htm]
- Long Time Type [http://david.tribble.com/text/c0xlongtime.html]
- Calendar Date Functions [http://david.tribble.com/text/c0xcalendar.html]
- Timezone Functions [http://david.tribble.com/text/c0xtimezone.html]

Overall the proposals in these papers are complementary to this proposal. The suggested enhancements to the C library would serve as an excellent foundation for building the functionality provided in this proposal.

OMG/Corba Time Service

The Object Management Group (OMG) has defined two standards for timing services: Time Service [http://www.omg.org/technology/documents/formal/time_service.htm] and Enhanced Time Service [http://www.omg.org/cgi-bin/doc?formal/04-10-04]. These standards serve as another example of a standard that reflects many of the core elements of this proposal. Some of the similar features of these services include:

- time period concepts
- timepoint resolutions to 100 nano seconds
- epoch from 1582

The current proposal would provide for the basis for simple integration with these services allowing an alternative implementation of a clock from a distributed server.

Timezone Database

The Timezone Database [http://www.twinsun.com/tz/tz-link.htm] provides an open collection of timezone information and tools.

A variety of operating systems provide an extension that provides The work captured in the timezone database demonstrates much of the complexity associated with various time adjustments. Some of the interesting capabilities of the timezone database include:

- mapping of regions to timezones (eg: "america/phoenix")
- representation of historical timezone data

While the Boost Date-Time Library [http://www.boost.org/libs/date_time/index.html] provides a timezone capable of performing similar functions, that capability is NOT part of this proposal. In addition, historical time zones are not directly supported, although the base time zone and local time interactions allow for extended implementations to provide this capability.

Boost Date-Time Library

The Boost Date-Time Library [http://www.boost.org/libs/date_time/index.html] serves as the primary foundation for this proposal. Differences between the proposal and Boost date-time are outlined in Difference from Boost Date-Time.

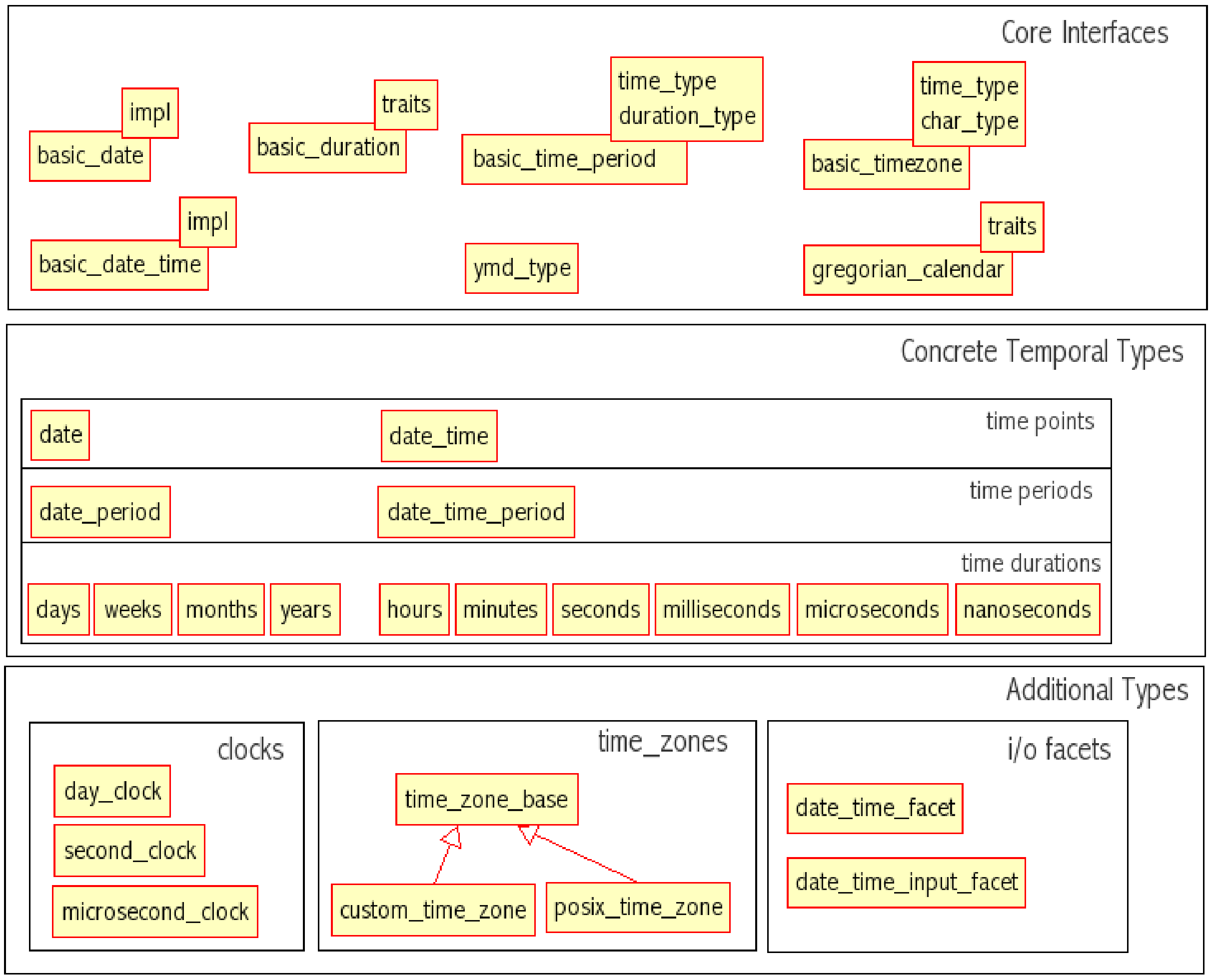
Design

Overview of Design

This proposal divides the library interface into several levels:

- Core Interfaces
- Concrete Temporal Types
- Additional Types

The core interfaces provide a templated library core that supports flexible resolutions and other capabilities similar to the way `std::basic_string` provides a core for string types. The concrete temporal types are built from the core interfaces and realize a set of types that serve as the primary programmatic interface. This is similar to how `std::string` and `std::wstring` provide concrete realizations that most C++ programmers use. The additional types provide implementation of clocks, time zones, and input output facilities. These classes provide other capabilities not offered directly in the temporal types. The figure below illustrates the various classes that make up the proposal.



Core Interfaces

The following classes make up the core interfaces of the library. The core interfaces are templates the separate aspects of the key library interfaces from implementation so that additional temporal type variations can be provided. Typical developers will not use these interfaces directly, but will use the temporal types based on these interfaces.

Type Name	**Description**
basic_date	Core interface for dates.
basic_time	Core interface for combined date and time types.
basic_duration	Core interface for duration types.
basic_time_period	Core interface for period types.
basic_time_zone	Core interface for time zones.
ymd_type	Replacement for `tm` that includes fractional seconds and resolution information.
gregorian_calendar	Core implementation of gregrorian / iso calendar.

YMD Types

To efficiently format dates and times it is convenient to have a structure that represents the 'broken down' time. This is similar to `tm` without the string values and with some additional fields to represent fractional seconds and other information.

```
//struct to represent elements of a point in time
struct timepoint {
  year_type  year;        //32 bit unsigned integer -- range depends on calendar
  week_type  month;       //short integer 1-12 -- zero flags invalid
  short      day_of_year; //short
  short      day_of_week;  //0-6  -- 0 == sunday
  short      week_number;  //1-53 - 0 indicates invalid
  short hours;             //0-24
  short minutes;           //0-59
  short seconds;           //0-60 -- 60 is leap second
  short fractional_seconds_count;
  short frac_seconds_resolution;  //increments of 10 only
};
```

For input and output this form is a useful tool.

Summary of Temporal Types

The temporal types provide the main programmer interface to the library. The temporal types are value types that provide for efficient comparison, assignment, calculation, and other operations. In general, these types are designed to approach the efficiency of a regular integer type in terms of size, comparison and calculation efficiency. Some of the design decisions impacting these types is summarized in Temporal Types - Key Design Decisions.

Type Name	Temporal Type	Description
date	point	Represent a date using gregorian calendar.
date_period	period	Represent a date period using date and days.
days	duration	Represent a count of days.
weeks	duration	Represent a count of weeks.
months	duration	Represent a count of months.
years	duration	Represent a count of years.
date_time	point	Represent a combined date and time with 1 microsecond resolution.
date_time_period	period	Represent a period using date_time.
seconds	duration	Represent a count of seconds.
milliseconds	duration	Represent a count of milliseconds.
microseconds	duration	Represent a count of microseconds.
nanoseconds	duration	Represent a count of nanoseconds.

Note that user defined temporal types can be added as needed. For example, a user can use basic_date_time to create a time point with nanosecond level resolution. To accomplish this might require expanding the size of the underlying type to 96 bits or changing the epoch times.

Arithmetic Operations on Temporal Types

These 3 temporal types form the foundation for enabling sophisticated calculations using dates and times. For example, no matter the resolution of time points and durations we can say that the following calculations apply where --> means 'results in'.

Rule --> Result Type	**Notes**
Timepoint + Duration --> Timepoint	Valid only for durations of equal or greater resolution.
Timepoint - Duration --> Timepoint	Valid only for durations of equal or greater resolution.
Timepoint - Timepoint --> Duration	Timepoints of the same resolution only.
Duration + Duration --> Duration	In mixed resolution durations, higher resolution duration must be on the left.
Duration - Duration --> Duration	In mixed resolution durations, higher resolution duration must be on the left.
Duration * Integer --> Duration	
Integer * Duration --> Duration	
Duration / Integer --> Duration	Integer Division rules.
Duration + Timepoint --> Undefined	Compilation error.
Duration - Timepoint --> Undefined	Compilation error.
Timepoint + Timepoint --> Undefined	Compilation error.

Note that the above typing rules are somewhat different from those used in Boost Date-Time Library [http://www.boost.org/libs/date_time/index.html]. After some experimenting with various solutions to handling multi-resolution arithmetic, the proposed approach seems to offer a pragmatic approach that enables clean application coding while avoiding truncation error surprises for application developers.

The effect of the above rules can be seen more clearly with some examples:

```
date d(...);
d += seconds(10); //compilation error -- dates have resolution of 1 day
d += days(1);     //ok.
d += months(2);   //ok.
d += days(3) + weeks(2)   //ok.
d += weeks(2) + days(2);  //compilation error
  date_time dt(...);     //microsecond resolution time point
dt += seconds(100);    //ok.
dt += days(2);         //ok.
dt += nanoseconds(1);  //compile error -- resolution insufficient
dt += microseconds(1); //ok.
```

In addition, to the rules above, there are additional rules associated with special value handling:

Rule --> Result Type	**Notes**
Timepoint(NADT) + Duration --> Timepoint(NADT)	
Timepoint(∞) + Duration --> Timepoint(∞)	
Timepoint + Duration(∞) --> Timepoint(∞)	
Timepoint - Duration(∞) --> Timepoint(-∞)	
Timepoint(+∞) + Duration(-∞) --> NADT	
Duration(+∞) + Duration(-∞) --> NADT	
Duration(∞) * Zero --> NADT	
Duration(∞) * Integer(Not Zero) --> Duration(∞)	
Duration(+∞) * -Integer --> Duration(-∞)	
Duration(∞) / Integer --> Duration(∞)	

Lack of Communitivity for Some Operations

While most time durations behave exactly as would be expected by normal numeric operations, years and months exhibit odd behavior because they are not of a fixed length. That is, sometimes a year is 365 days and sometimes it is 366 (leap year).

In addition to the basic capabilities above most mathematical operations on dates and times are reversible.

That is,

```
date d(2004,Jan,1);
d1 = d + months(1); //d1 == Feb 1
d1 -= months(1); //d1 == d
```

However the following is a difficulty:

```
date d(2004,Jan,31);
date d1 = d + months(1); //Feb 29
date d2 = d1 - months(1); //oops Jan 29th
d1 - months(1) == d; //false.
```

Not all temporal types follow the same rules as integer types in calculations. For example, months and years are not directly convertible to a number of days. A month is between 28 and 31 days. A year is either 365 or in a leap year 366 days. Logic that needs to 'add months' or 'add years' needs to manage this issue.

The following W3C XML reference describes the proposed arithmetic rules in more detail XML Schema Adding Durations [http://www.w3.org/TR/xmlschema-2/#adding-durations-to-dateTimes] .

Summary of Additional Types

The following are some additional classes that augment the temporal types to provide input/output and local time adjustment capabilities.

Type Name	**Description**
time_facet	Format based output facet for localization/customization of output.
time_input_facet	Format based input facet for localization/customization of input.

The library provides several clock types for measuring the current time to different levels of resolution.

Type Name	**Description**
day_clock	Measures the current time at day level resolution.
second_clock	Measures the current time to one second resolution.
microsecond_clock	Measures the current time to microsecond resolution.

The concrete time zone classes provide the ability to perform local time adjustments.

Type Name	**Description**
time_zone_base	Narrow char instantiation of `basic_time_zone` for use with `date_time`.

Type Name	Description
custom_time_zone	Class that provides for the ability to provide custom timezones in an application.
posix_time_zone	Provides implementation of Posix 1003.1 time zone specifications.

Exceptions for Errors

All errors are signaled by exceptions. While this makes the library less suitable for some contexts, exceptions are a usual part of modern C++ code. Exceptions are thrown in the following circumstances:

- Attempts to construct and invalid temporal type (eg: 2005-Feb-30)
- Using input streaming with exceptions enabled

All exceptions derive from std::out_of_range or std::logic_error and are summarized in the table below.

Type	Method	Exception	Base Type	Description
date	constructor, operator>>	bad_day_of_month	`std::out_of_range` [http://gcc.gnu.org/onlinedocs/libstdc++/latest-doxygen/classstd_1_1out__of__range.html]	Thrown if day of month is out of range (eg: Jan 32)
date	constructor, operator>>	bad_day_of_year	`std::out_of_range` [http://gcc.gnu.org/onlinedocs/libstdc++/latest-doxygen/classstd_1_1out__of__range.html]	Thrown if day of year is invalid. (note this is t.b.d. since there is no constructor)
date	constructor, operator>>	bad_month	`std::out_of_range` [http://gcc.gnu.org/onlinedocs/libstdc++/latest-doxygen/classstd_1_1out__of__range.html]	Thrown if the specified month is out of range (eg: month 13)
date	constructor, operator>>	bad_year	`std::out_of_range` [http://gcc.gnu.org/onlinedocs/libstdc++/latest-doxygen/classstd_1_1out__of__range.html]	Thrown if the year is out of range (eg: >= 10000, or < 1400)

Extensions to time_put and Format Flags

The following table describes the new flags to be supported by `time_put` facet.

Flag	**Description**
%f	Fractional seconds and separator - always used even when value is zero.
%F	Fractional seconds and separator - only output when non-zero.
%s	Seconds separator and fractional seconds
%T	Time in 24-hour notation %H:%M:%S (from strftime)
%q	ISO time zone
%Q	ISO extended time zone
%r	Time in AM/PM notation - same as '%I:%M:%S %p' (from strftime)
%V	ISO week number in range from 0 to 53 (from strftime)
%Z	Time zone name - long (eg: 'Eastern Standard Time').
%ZP	Posix time zone string (eg: EST-05EDT+01,M4.1.0/02:00,M10.5.0/02:00

In addition to adding new formatting flags this proposal recommends changing the default formatting of dates and times. The following table summarizes the proposed default i/o formats.

Temporal Type	**Format**	**Example**	**Comment**
date	%Y~~%b~~%d	2004-Jan-31	
days	None	45	No formatting or adornment required
date_period	[%Y~~%b~~%d/%Y~~%b~~%d)	[2004-Jan-31/2004-Feb-25]	
date_time	%Y~~%b~~%d %H:%M:%S.%f	2004-Jan-31 05:10:31.00000030	
time duration types	%H:%M:%S.%f	05:10:31.00000030	
date_time_period	[%Y~~%b~~%d/%Y~~%b~~%d)	[2004-Jan-31/2004-Feb-25]	

The main recommendation is that the date format use "%Y~~%b~~%d". This format is unambiguous and clear in all cases. The four digit year avoids any confusion about the century. Using short characters for the month ensures differentiation with the day of the month. For example, consider the following:

```
2004-01-02 //what's this date anyway?
```

Differences from Boost Date-Time

Some of the key differences between this proposal and the Boost Date-Time Library [http://www.boost.org/libs/date_time/index.html] include:

- Namespaces (std::tr2 versus boost::gregorian, boost::posix_time, etc)
- `ptime` is called `date_time`
- No small types for temporal types (eg: no `greg_day`, `greg_year`, etc)
- Adjusting resolutions implementation

Summary of Key Design Decisions

Advantages of the Design

Some of the key advantages of the design include:

- efficient value types for programming with dates and times
- flexible input and output
- direct support for ISO standard calendar systems
- extensibility

The extensive use of value types makes programming with dates and times easy and natural for application programmers. The lack of virtual functions and other aspects of the underlying representations for dates and times, also make the resulting code efficient and suitable in many contexts. Use of immutable types minimizes the required interfaces and simplifies the user programming model. Using exceptions for errors furthers modern error handling practices and ensures correctness.

Flexible format-based input and output integrated with iostreams dramatically eases the difficulties of handling different date-time representations and localization.

The separation of clocks and temporal types as well as the template-based design provide for extensibility.

Limitations of the Design

The lack of virtual functions means that users cannot use inheritance to extend the temporal types.

The underlying representation of dates and times combined with the always valid after construction rules mean that applications that only perform input-output conversion will incur some performance overhead to convert to the internal type and back. In practice this has not proven to be an issue with the Boost Date-Time Library [http://www.boost.org/libs/date_time/index.html], but some applications with extreme performance requirements will need to continue using approaches more optimized for input-output conversion.

There is a long list of related capabilities not included in this proposal. Thus users that need these additional capabilities will need to build them or find them.

General Principles

Extensive Use of Value Types

In the realization of the date-time concepts Boost Date-Time Library [http://www.boost.org/libs/date_time/index.html] uses a large number of value-type classes. As Kevlin Henney has defined in the from Boost.Any documentation [http://www.boost.org/doc/html/any/reference.html#any.ValueType]:

```
"Values are strongly informational objects for which
identity is not significant, i.e. the focus is principally
on their state content and any behavior organized around that.
Another distinguishing feature of values is their granularity:
normally fine-grained objects representing simple concepts in
the system such as quantities."
```

This is discussed more extensively in Objects of Value (pdf) [http://www.two-sdg.demon.co.uk/curbralan/papers/ObjectsOfValue.pdf].

Overall, the library consistently endows value types with the following properties:

- assignability with strong exception-safety
- copy constructible
- immutability
- always valid after construction
- less than comparable
- streamable

These principles are key to the overall design of the proposal.

Templates, Interface Separation, and Extensibility

One key goal of the library is to allow for user extensibility. By providing templated interfaces users can replace the internal implementation of the type allowing for customized adjustments. Most users will not need to use this capability, but for those in need of special adjustments this element of the design is essential.

Some variations that this design allows are as follows:

- removal of special value interfaces to simplify arithmetic
- special adjustments such as leap seconds
- specification of alternate time epochs
- specification of different calendar implementations

Just as other parts of the C++ standard library support extensibility, the date-time additions should be no different. Unfortunately different date-time applications have different needs for resolution support, efficiency, and special adjustments. Hence a standard with no extensibility will exclude a significant set of application developers.

The extensibility is achieved by providing a core set of interfaces upon which concrete temporal types are implemented.

Using the Gregorian Calendar

The Gregorian Calendar is the basis for the ISO 8601 standard [http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=26780] and is the mostly widely used calendar system worldwide. Of course there are many other Calendar systems which are not supported by this proposal. That said, many of the concepts in this proposal provide insight into how the new library can be extended to support other calendars.

Since this proposal supports the representation of historic dates and times a decision must be made about how to handle some of the anomalies and historic adjustments to the calendar. In addition, the proposal supports the representation of dates prior to the historic adoption of the Gregorian Calendar.

The recommendation is for the calendar to support a proleptic version of the Gregorian Calendar. The proleptic system extends the calendar into the past and assumes a monotonic representation of the calendar -- essentially ignoring historic adjustments. For most applications this representation is sufficient. However, for those requiring absolute accuracy a new set of temporal types can be created by replacing the gregorian_calendar with one that performs the required adjustments.

Separation of Date and Time Types

A different approach to the domain would eliminate the 'date' types and just use 'date_time' when dates are used. The disadvantage of this approach is that the 'date' type is cumbersome for some applications. When storing a birthday, for example, usually doesn't include the time of birth. In addition, the date type can be implemented using a smaller data type (32 bits typically) thus providing space efficiencies.

Separation of Clocks and Temporal Types

Many date-time libraries tie creation of a temporal type to the clock implementation. This proposal explicitly separates the two so that temporal types can be ported to platforms that don't supply a hardware clock or have different clock capabilities. A corollary to this is that high resolution time types (say microseconds and nanoseconds) are still available on platforms that do not have hardware that supports these resolutions. These values might be read from a file, for example. Finally, this separation makes the addition of new clocks, for example a network time clock or GPS time source, fit naturally with the library interface.

The result of this design decision is that retrieving a value from a clock device cannot be done directly from a time point. So, for example, the following code is not possible:

```
//mythical interface that ties clock implementation to time point
date_time t(NOW); //makes a call to clock
```

Instead the interface in this proposal is as follows:

```
date_time t1(microsecond_clock::universal_time()); //UTC time to microseonds
date_time t2(microsecond_clock::local_time()); //Local time to microseonds
date_time t3(second_clock::local_time()); //Local time to seconds
date_time t3(second_clock::local_time(tz)); //Local time in time zone specified by tz
```

Temporal Types - Key Design Decisions

Assignability, Comparability, and Streamability

Assignability of the types means that the constructors and operators allow for natural use of the types:

```
time t(date(2004,Jan,1));// 2004-Jan-01 00:00:00.0
//...use t and then replace value with 100 seconds from current UTC time
t = second_clock::universal_time() + seconds(100);
```

Because temporal types always construct to a valid value the assignment operator can offer the strong exception guarantee.

All the temporal types are support a full complement of comparison operators

- `operator!=(const type&) const`
- `operator==(const type&) const`
- `operator<(const type&) const`
- `operator<=(const type&) const`

- `operator>(const type&) const`
- `operator>=(const type&) const`

All temporal types are streamable supporting both input and output.

- `operator<<(std::ostream&, const type&)`
- `operator>>(std::istream&, type&)`

See Summary of Additional Types for more information on format-based streaming of temporal types.

Immutability

With only a couple exceptions, the core temporal types are immutable. That is, once constructed they cannot be modified other than being assigned from another instance of that same type. For example, there is no way to change the 'month' part of a date once it has been constructed. There are a couple reasons for this design decision. First, by keeping these types immutable the supported interfaces are kept to a minimum. For a simple 'date' class there would need to be at least 3 additional functions (setMonth, setDay, setYear) if the type was not immutable.

This is especially true when consider the exception handling required for this trivial case:

```
date d(2004, Jan, 31);
d.month(Feb); //hypothetical interface -- an exception would be generated
```

In the case of 'time' representations the interface becomes even more difficult and cumbersome. For example, what interface should be supported for setting the fractional seconds of a time?

```
time t(...);
t.setMilliSeconds(3); //a hypothetical 'set' interface
```

With a 'set interface' approach a new method is needed for every possible time resolution. The list is long and includes: setHours, setMinutes, setSeconds, setTenths, setNano, setMilli, etc. This myriad of methods in the time class makes the interface more cumbersome to understand and use. More troubling is that it is not user extensible to a new duration type. Instead of adding to the interface of the time point the use of additional duration types and operators is more natural and extensible. For example:

```
time t(date(2004,Jan,1));// 2004-Jan-01 00:00:00.0
t += milliseconds(5) + nanoseconds(10);
```

Lack of Virtual Functions

All the core temporal types avoid the use of virtual functions in the implementation to ensure efficient use of memory. In general, the goal is to make these temporal types behave as close to `time_t` in terms of run time efficiency as is practical.

Note that a major departure from this approach is made to provide local time support. While `local_time` does not have any virtual functions, it uses `time_zone` which is a pure virtual base class. In addition, `local_time` interfaces interchange `time_zone`s using `shared_ptr`s. Thus the memory costs of manipulating `local_time`s are higher.

However, this is reasonable given a need to support a variety of different approaches to providing time zones.

Always Valid After Construction

The temporal types are designed to always be valid after construction. Note that in this context, `not-a-date-time` is a valid value for a date. However, February 30th is never a valid date and will always result in a `bad_day` exception. The main advantage of this approach is program correctness is maintained and checking for invalid values in programs is minimized.

Enforcing initial construction rules means that no validation is required after the initial checking of the value. However, it does have a cost in performance for construction of many of the temporal types. Also, not that in the case of time durations checking is normally not required since they are simply a count of some amount of time (eg: 5 hours). Experience with Boost Date-Time Library [http://www.boost.org/libs/date_time/index.html] indicates that this is an acceptable cost to pay for an improved programming model.

Default Construction to not-a-date-time

Early versions of the Boost Date-Time Library [http://www.boost.org/libs/date_time/index.html] did not support default construction of temporal types such as `date`. However, users requested the addition of default construction for integration with standard library containers (eg: map keys need to be default constructable) and for input streaming.

The consensus decision was that default construction to the 'not-a-date-time' value was the most reasonable default. Some libraries make the default construction of a time point to the current time. However, this creates serious scalability issues when creating large numbers of dates and times with a default value only to be updated later.

Underlying representations for dates and times

The Boost Date-Time Library [http://www.boost.org/libs/date_time/index.html] uses integer types to represent all temporal types such as `date` and `time_duration`. There are several reasons for this decision:

- Highly efficient use of memory
- Fast comparison and calculations
- 'Lossless' calculations

Some libraries choose to use doubles to represent temporal types. The problem with double types is that they can trivially produce unexpected results with mathematical operations. This means the programmer must manage the potential error associated with all date_time calculations. With an integer type underlying all calculation the error is limited only to division operations which are not supported on most temporal types. For example, time points do not logically support division:

```
date d;
date d2 = d / 10; //what would this mean? Not allowed.
```

Durations do support division, so the programmer must be aware of the division rules:

```
//This makes sense
days day_count(10);
days day_count = dc / 10; //defined == 1
```

Special Values

Special values add significant complications to the design of the library. In particular, the logic of calculation is slower and more complex for users to understand. In addition, input and output is made more complex since strings need to be defined for the various default values.

At the same time these special values are extremely helpful when dealing with design issues related to dates and times. One of these problems is the so-called 'Null object' pattern. In this case, the 'not_a_date_time' special value. This also serves as a useful value for default construction. For more on this see always valid after construction. Infinities are useful in a number of contexts, but particularly when you want to specify the concepts like 'until further notice', 'long ago', or lasts 'forever'. Note that the Rogue Wave RWDateTime class supports a similar concept called sentinels.

In addition to infinities and not_a_date_time the special values include the minimum and maximum values. This allows for rapid construction of the minimum or maximum date or time.

While special values increase the complexity of calculation they do not increase the size of the internal representation. Boost.date_time simply uses 3 of the values of the underlying type to represent the special values.

Input-Output Design Decisions

Overall, the date-time extension for C++ should meet the following expectations:

- localizable formatting and parsing
- reasonable and unambiguous default input and output form
- parameterized support for different character types
- full user customization of strings for months, weekdays, and special values

Programmers expect all integer types to be localizable with regards to input and output streaming. Similarly, all of the temporal types supported by the library should support input and output streaming. For example:

```
//Simple example of date streaming functionality
std::stringstream ss("2004-Jan-31");
date d;
ss >> d;
std::cout << d; //prints 2004-Jan-31
```

Of course, there are other ways to specify a date which should also be parseable. For example, the 31st day of 2004 is another way to say January 31st.

The current C/C++ standard library provides for date and time input output via the time_put [http://gcc.gnu.org/onlinedocs/libstdc++/latest-doxygen/classstd_1_1time__put.html] and time_get [http://gcc.gnu.org/onlinedocs/libstdc++/latest-doxygen/classstd_1_1time__get.html] facets. The date-time extension builds on this foundation to provide full format-based stream input and output for temporal types using facets. The facet classes give the user full direct control over the strings used for both input and output.

See Extensions to time_put and Format Flags for more on the proposed default formats.

Local Time Adjustments

Local time adjustments are one of the more difficult issues for the design to address. Overall, the support for local time adjustments has been lacking in most date-time library implementations. There are many reasons for the poor support including:

- lack of standards in representation of time zones
- varying platform support
- complex rules for calculation of daylight savings rules
- complexity of input-output associated with time zones

The Posix 1003.1 specification provides key support for representation of time zones. While this standard is helpful, it does not take into account for fully for time zone representations. For example, it is not uncommon to have 4 strings to represent all the variations of the time zone: abbreviated forms for both standard and daylight adjusted times.

```
//4 representations of US eastern time zone
EDT
EST
Eastern Daylight Time
Eastern Standard Time
```

Posix 1003.1 cannot represent all these variations. In this proposal the time zone base provides for representation of these four variations. When used in conjunction with the posix_time_zone the long strings fall back to using the short abbreviations. However, the time_zone allows users to provide all these elements enabling sophisticated input and output.

>From an application development perspective there are several key adjustment scenarios:

- adjust to a local time when reading the current time
- adjust from one time zone to another
- handle any daylight savings time adjustments
- set time zones based on user input

This proposal supports these key use cases by providing an extensible set of time zone types. In addition, the clock interfaces support adjusting to both the local time as specified on the machine as well as a local time specified by an arbitrary time zone.

Proposed Text

Note that the following section is not a complete formal specification, but a rather a rough draft to use as a starting point for the final proposal.

Core Changes

```
//placeholder for details to time_put/time_get and other changes
```

Core Classes

Enumerations

```
//used for special value construction
enum special_values {negative_infinity, positive_infinity, not_a_date_time,
                     max_date_time, min_date_time, not_special };
```

Basic Duration

```
template<class duration_rep_traits>
class basic_duration
{
public:
  typedef typename duration_rep_traits::int_type  duration_rep_type;
  typedef typename duration_rep_traits::impl_type duration_rep;

  //constructors
  explicit basic_duration(duration_rep day_count);
  basic_duration(special_values);

  duration_rep get_rep()   const;
  duration_rep get_broken_down_duration()      const;

  //special value access
  bool           is_special()  const;
  special_values as_special()  const;

  static duration_type unit();

  //comparison operators
  bool operator<  (const date_duration&) const;
  bool operator<= (const date_duration&) const;
  bool operator>  (const date_duration&) const;
  bool operator>= (const date_duration&) const;
  bool operator== (const date_duration&) const;
  bool operator!= (const date_duration&) const;

  //arithmetic operations
  duration_type operator-  (const duration_type&) const;
  duration_type operator-= (const duration_type&);
  duration_type operator+  (const duration_type&) const;
  duration_type operator+= (const duration_type&);
  duration_type operator/= (int);
  duration_type operator/  (int);
  duration_type operator*  (int) const;
  duration_type operator*= (int);

  //Sign inversion
  date_duration operator-() const;

  //! return sign information (todo?)
  bool is_negative() const
};
```

Basic Gregorian Calendar

```
template<typename ymd_type_, typename date_int_type_>
class basic_gregorian_calendar {
public:
  typedef ymd_type_  ymd_type;
  typedef typename ymd_type::month_type  month_type;
  typedef typename ymd_type::day_type  day_type;
  typedef typename ymd_type::year_type  year_type;
  typedef date_int_type_ date_int_type;
  static unsigned short day_of_week    (const ymd_type& ymd);
  static int            week_number    (const ymd_type&ymd);
  static date_int_type  day_number     (const ymd_type& ymd);
  static ymd_type       from_day_number(date_int_type);

  static date_int_type  julian_day_number(const ymd_type& ymd);
  static long           modjulian_day_number(const ymd_type& ymd);
  static ymd_type       from_julian_day_number(date_int_type);
  static ymd_type       from_modjulian_day_number(long);

  static bool           is_leap_year(year_type);
  static unsigned short end_of_month_day(year_type y, month_type m);
  static ymd_type       epoch();
  static unsigned short days_in_week();

};
```

Basic Date

```
template<class date_impl>
class basic_date {
 public:
  typedef typename date_impl::calendar_type calendar_type;

  //year, month, day types
  typedef typename date_impl::year_type  year_type;
  typedef typename date_impl::month_type month_type;
  typedef typename date_impl::day_type   day_type;
  typedef typename date_impl::ymd_type   ymd_type;

  typedef typename date_impl::day_of_week_type    day_of_week_type;
  typedef typename date_impl::day_of_year_type    day_of_year_type;
  typedef typename date_impl::week_of_year_type   week_of_year_type;

  typedef typename date_impl::weekday_type        weekday_type;
  typedef typename date_impl::week_of_month_type  week_of_month_type;

  typedef typename date_impl::special_values_type special_values_type;

  typedef typename date_impl::date_duration_type  date_duration_type;

  typedef typename date_impl::date_int_type date_int_type;


  //constructors
  basic_date(year_type y, month_type m, day_type d);

  basic_date(const ymd_type& ymd);

  basic_date(year_type y, day_of_year_type doy);

  basic_date(const basic_date& rhs);
    //constructs something like: Sunday in Week 50 of year 2004
  basic_date(year_type y, week_of_year_type week_number, weekday_type wd);

  //constructs something like: 3rd Monday in Feb of 2004
  basic_date(year_type y, month_type m, week_of_month_type week_number, weekday_type wd);

  basic_date(std::time_t t);

  basic_date(const std::tm& datetm);

  //construct positive/negative infinity max or min date
  basic_date(special_values_type sv);

  //not-a-date-time
  basic_date();
  //Basic accessors
  year_type        year()           const;
  month_type       month()          const;
  day_type         day()            const;
  day_of_week_type day_of_week()    const; //eg: Sun, Mon, ...


  //additional accessors
  day_of_year_type  day_of_year()  const; //1..365 or 1..366 (for leap year)
  week_of_year_type week_number()  const; //ISO 8601 week number 1..53
  date_int_type     day_number()   const; //Return the day number since start of the epoch
  basic_date        end_of_month() const; //Return the last day of the current month

  //special value accessors
  bool is_special()                 const;
  special_values_type as_special()  const;

  //conversion accessors
  time_t         to_time_t()      const;
  ymd_type       year_month_day() const; //Get ymd structure

  // Conversion to julian calendar
  date_int_type julian_day()     const;
  long          modjulian_day()  const;

  //calculation accessors
  date_duration_type days_until  (weekday_type) const;
  date_duration_type days_before (weekday_type) const;
  basic_date         next        (weekday_type) const;
  basic_date         previous    (weekday_type) const;

  //comparison operators
  bool operator<  (const basic_date&) const;
  bool operator<= (const basic_date&) const;
  bool operator>  (const basic_date&) const;
  bool operator>= (const basic_date&) const;
  bool operator== (const basic_date&) const;
  bool operator!= (const basic_date&) const;

  //arithmetic operations
  date_duration_type operator-  (const basic_date&)     const;
  basic_date         operator-  (const date_duration_type&) const;
  basic_date         operator-= (const date_duration_type&);
  basic_date         operator+  (const date_duration_type&) const;
  basic_date         operator+= (const date_duration_type&);


};

  template<class date_type, typename string_type>
date_type from_string(const string_type& date_string,
               const string_type format_string);

template <class CharT, class TraitsT>
std::basic_ostream<CharT, TraitsT>&
operator<<(std::basic_ostream<CharT, TraitsT>& os, const date&);

template <class CharT, class Traits>
std::basic_istream<CharT, Traits>&
operator>>(std::basic_istream<CharT, Traits>& is, date&);


//Exception classes for construction of dates
class bad_day_of_month : public std::out_of_range
{};
class bad_year : public std::out_of_range
{};
class bad_month : public std::out_of_range
{};
class bad_weekday : public std::out_of_range
{};
  //Exception class for conversions of local time values
class invalid_time_label : public std::logic_error
{};
```

Clock Devices

```
template<class date_type>
class day_clock
{
public:
  typedef typename date_type::ymd_type ymd_type;
  static typename date_type::ymd_type local_day_ymd()
  static typename date_type::ymd_type universal_day_ymd()

  static date_type local_day();
  static date_type universal_day();
};
```

Time Durations

```
template<class T, typename rep_type>
class basic_time_duration
{
public:
  typedef T duration_type;  //the subclass
  typedef rep_type traits_type;
  typedef typename rep_type::day_type  day_type;
  typedef typename rep_type::hour_type hour_type;
  typedef typename rep_type::min_type  min_type;
  typedef typename rep_type::sec_type  sec_type;
  typedef typename rep_type::fractional_seconds_type fractional_seconds_type;
  typedef typename rep_type::tick_type tick_type;
  typedef typename rep_type::impl_type impl_type;

  time_duration();
  time_duration(hour_type hours,
                min_type minutes,
                sec_type seconds=0,
                fractional_seconds_type frac_sec = 0);

  time_duration(const time_duration<T, rep_type>& other)
  time_duration(special_values sv);

  static duration_type unit(); //support for periods

  hour_type hours()   const;
  min_type minutes()  const;
  sec_type seconds()  const;
  fractional_seconds_type fractional_seconds() const;

  sec_type total_seconds()       const;
  tick_type total_milliseconds() const;
  tick_type total_nanoseconds()  const;
  tick_type total_microseconds() const;

  //traits information
  static unsigned short num_fractional_digits()
  static tick_type ticks_per_second();
  static time_resolutions resolution();

  //special value accessors
  bool is_special() const
  bool is_pos_infinity() const
  bool is_neg_infinity() const
  bool is_not_a_date_time() const
  //sign inversion
  duration_type operator-() const

  bool is_negative() const

  bool operator<  (const time_duration&) const;
  bool operator<= (const time_duration&) const;
  bool operator>  (const time_duration&) const;
  bool operator>= (const time_duration&) const;
  bool operator== (const time_duration&) const;
  bool operator!= (const time_duration&) const;


  //arithmetic operations
  duration_type operator- (const duration_type& d) const
  duration_type operator-=(const duration_type& d)
  duration_type operator+ (const duration_type& d) const
  duration_type operator+=(const duration_type& d)
  duration_type operator/ (int divisor) const
  duration_type operator/=(int divisor)
  duration_type operator* (int rhs) const
  duration_type operator*=(int divisor)

  tick_type ticks() const


};

//typedef to the given resolution

//Concrete Time Duration instanciations
class hours;
class minutes;
class seconds;
class milliseconds;
class microseconds;
class nanoseconds;
```

Basic Date_Time Type

```
template <class T, class time_system>
class basic_date_time
{
public:
  typedef T time_type;
  typedef typename time_system::time_rep_type time_rep_type;
  typedef typename time_system::date_type date_type;
  typedef typename time_system::date_duration_type date_duration_type;
  typedef typename time_system::time_duration_type time_duration_type;

  basic_date_time(const date_type& day,
             const time_duration_type& td)
  {}
  base_time(special_values sv)
  base_time(const time_rep_type& rhs);

  date_type date() const
  time_duration_type time_of_day() const


  //special value accessor
  bool is_not_a_date_time()  const
  bool is_infinity()  const
  bool is_pos_infinity()  const
  bool is_neg_infinity()  const
  bool is_special() const

  //comparison operators
  bool operator<  (const time_type&) const;
  bool operator<= (const time_type&) const;
  bool operator>  (const time_type&) const;
  bool operator>= (const time_type&) const;
  bool operator== (const time_type&) const;
  bool operator!= (const time_type&) const;

  time_duration_type operator-(const time_type& rhs) const
  time_type operator+(const date_duration_type& dd) const
  time_type operator+=(const date_duration_type& dd)
  time_type operator-(const date_duration_type& dd) const
  time_type operator-=(const date_duration_type& dd)
  time_type operator+(const time_duration_type& td) const
  time_type operator+=(const time_duration_type& td)
  time_type operator-(const time_duration_type& rhs) const
  time_type operator-=(const time_duration_type& td)

};
```

Operators

```
date_time operator+ (const date_time& t, const months& m)
date_time operator+=(date_time& t, const months& m)
date_time operator- (const date_time& t, const months& m)
date_time operator-=(date_time& t, const months& m)

date_time operator+ (const date_time& t, const years& y)
date_time operator+=(date_time& t, const years& y)
date_time operator- (const date_time& t, const years& y)
date_time operator-=(date_time& t, const years& y)
```

Basic Time Period

```
template<class point_rep, class duration_rep>
class basic_time_period
{
public:
  typedef point_rep point_type;
  typedef duration_rep duration_type;

  //constructors
  basic_time_period(point_rep first_point, point_rep end_point);
  basic_time_period(point_rep first_point, duration_rep len);

  point_rep begin()     const;
  point_rep end()       const;
  point_rep last()      const;
  duration_rep length() const;
  bool is_null()        const;

  //comparison operators
  bool operator<  (const period&) const;
  bool operator<= (const period&) const;
  bool operator>  (const period&) const;
  bool operator>= (const period&) const;
  bool operator== (const period&) const;
  bool operator!= (const period&) const;

  //specialized accessors
  bool contains    (const point_rep& point) const;
  bool contains    (const period& other)    const;
  bool intersects  (const period& other)    const;
  bool is_adjacent (const period& other)    const;
  bool is_before   (const point_rep& point) const;
  bool is_after    (const point_rep& point) const;
  period intersection (const period& other) const;
  period merge        (const period& other) const;
  period span         (const period& other) const;

  void shift(const duration_rep& d);

};
```

Time Zone Base

```
template<typename time_type, typename CharT = char>
class time_zone_base  {
public:
  typedef std::basic_string<CharT> string_type;
  typedef std::basic_stringstream<CharT> stringstream_type;
  typedef typename time_type::date_type::year_type year_type;
  typedef typename time_type::time_duration_type time_duration_type;

  time_zone_base() {};
  virtual ~time_zone_base() {};

  //Time zone names information
  virtual string_type dst_zone_abbrev() const=0;
  virtual string_type std_zone_abbrev() const=0;
  virtual string_type dst_zone_name()   const=0;
  virtual string_type std_zone_name()   const=0;

  //Offsets from UTC
  virtual time_duration_type base_utc_offset()                 const=0;
  virtual time_duration_type dst_offset()                      const=0;

  //Daylight savings details
  virtual bool               has_dst()                         const=0;
  virtual time_type          dst_local_start_time(year_type y) const=0;
  virtual time_type          dst_local_end_time  (year_type y) const=0;

  //Printing function
  virtual string_type        to_posix_string()                 const=0;

};
```

Concrete Temporal Types

Date Programming

```
typedef basic_date<...>                 date;
typedef basic_duration<...>             days;
typedef basic_duration<...>             weeks;
typedef basic_duration<...>             months;
typedef basic_duration<...>             years;
typedef basic_time_period<date, days>   date_period;
```

Time Programming

```
typedef basic_date_time<...>            date_time;
typedef basic_time_duration<...>        hours;
typedef basic_time_duration<...>        minutes;
typedef basic_time_duration<...>         seconds;
typedef basic_time_duration<...>         milliseconds;
typedef basic_time_duration<...>         microseconds;
typedef basic_time_duration<...>         nanoseconds;
```

Additional Types

Time Zone

```
template<class CharT = char>
class time_zone_names_base
{
public:
  typedef std::basic_string<CharT> string_type;
  time_zone_names_base(const string_type& std_zone_name,
                       const string_type& std_zone_abbrev,
                       const string_type& dst_zone_name,
                       const string_type& dst_zone_abbrev)
  {}
  string_type dst_zone_abbrev() const
  string_type std_zone_abbrev() const
  string_type dst_zone_name() const
  string_type std_zone_name() const
};


//Timezone type used for user customization
class time_zone : public time_zone_base<date_time> {
public:
  typedef typename time_type::year_type year_type;
  typedef boost::posix_time::time_duration time_duration_type;
  typedef time_zone_base base_type;
  typedef base_type::string_type string_type;
  typedef base_type::stringstream_type stringstream_type;

  time_zone(const time_zone_names& zone_names,
            const time_duration_type& base_utc_offset,
            const dst_adjustment_offsets& dst_offset,
            boost::shared_ptr<dst_calc_rule> calc_rule) :
  {};
  virtual ~time_zone() {};
  virtual std::string dst_zone_abbrev() const
  virtual std::string std_zone_abbrev() const
  virtual std::string dst_zone_name() const
  virtual std::string std_zone_name() const
  virtual bool has_dst() const
  virtual date_time dst_local_start_time(year_type y) const
  virtual date_time dst_local_end_time(year_type y) const
  virtual time_duration_type base_utc_offset() const
  virtual time_duration_type dst_offset() const
  virtual string_type to_posix_string() const
};
```

Posix Time Zone

```
template<typename time_type, typename charT=char>
class posix_time_zone : public time_zone<time_type, charT>  {
public:
  typedef typename time_type::year_type year_type;

  posix_time_zone(const std::string& s);
  virtual ~posix_time_zone();
  //various zone name strings
  virtual std::string std_zone_abbrev() const
  virtual std::string dst_zone_abbrev() const;
  virtual std::string std_zone_name()   const;
  virtual std::string dst_zone_name()   const;

  virtual bool has_dst() const;

  //calculate start / end of dst
  virtual time_type dst_local_start_time(year_type y) const;
  virtual time_type dst_local_end_time(year_type y) const;

  virtual time_duration_type base_utc_offset() const;
  virtual time_duration_type dst_offset() const;

  //! Returns a POSIX time_zone string for this object
  virtual string_type to_posix_string() const
};
```

In addition, input and output operators are provided:

```
template <class CharT, class TraitsT>
std::basic_ostream<CharT, TraitsT>&
operator<<(std::basic_ostream<CharT, TraitsT>& os, posix_time_zone&);

template <class CharT, class Traits>
std::basic_istream<CharT, Traits>&
operator>>(std::basic_istream<CharT, Traits>& is,  posix_time_zone&);
```

Clock Types

```
template<class date_type>
class day_clock
{
public:
  //returns a time adjusted to the specified time zone
  template<class time_zone_type>
  static date_type local_time(const time_zone_type& zone_spec);

  static date_type local_time();
  static date_type universal_time();
};


template<class time_type>
class microsecond_clock
{
public:
  //returns a time adjusted to the specified time zone
  template<class time_zone_type>
  static time_type local_time(const time_zone_type& zone_spec);

  static time_type local_time();
  static time_type universal_time();
};


template<class time_type>
class second_clock
{
public:
  //returns a time adjusted to the specified time zone
  template<class time_zone_type>
  static time_type local_time(const time_zone_type& zone_spec);
  static time_type local_time();
  static time_type universal_time();
};
```

Input-Output Facets

Date Output Facet

```
/* Class that provides format based I/O facet for date types.
 *
 * This class allows the formatting of dates by using format string.
 * Format strings are:
 *
 *  - %A => long_weekday_format - Full name Ex: Tuesday
 *  - %a => short_weekday_format - Three letter abbreviation Ex: Tue
 *  - %B => long_month_format - Full name Ex: October
 *  - %b => short_month_format - Three letter abbreviation Ex: Oct
 *  - %x => standard_format_specifier - defined by the locale
 *  - %Y-%b-%d => default_date_format - YYYY-Mon-dd
 *
 * Default month format == %b
 * Default weekday format == %a
 *
 *
 */
template <class date_type,
          class CharT,
          class OutItrT = std::ostreambuf_iterator<CharT, std::char_traits<CharT> > >
class date_facet : public std::locale::facet {
public:
  typedef typename date_type::duration_type duration_type;
  typedef typename date_type::day_of_week_type day_of_week_type;
  typedef typename date_type::day_type day_type;
  typedef typename date_type::month_type month_type;
  typedef boost::date_time::period<date_type,duration_type> period_type;
  typedef std::basic_string<CharT> string_type;
  typedef CharT                    char_type;
  typedef boost::date_time::period_formatter<CharT>  period_formatter_type;
  typedef boost::date_time::special_values_formatter<CharT>  special_values_formatter_type;
  typedef std::vector<std::basic_string<CharT> > input_collection_type;

  explicit time_facet(::size_t a_ref = 0);

  explicit time_facet(const char_type* format,
                      const input_collection_type& short_month_names,
                      ::size_t ref_count = 0);

  explicit time_facet(const char_type* format,
                      period_formatter_type period_formatter = period_formatter_type(),
                      special_values_formatter_type special_values_formatter = special_values_form
                      ::size_t ref_count = 0);

  void format(const char_type* const format);
  virtual void set_iso_format();
  virtual void set_iso_extended_format();
  void month_format(const char_type* const format);
  void weekday_format(const char_type* const format);

  void period_formatter(period_formatter_type period_formatter);
  void special_values_formatter(const special_values_formatter_type& svf);
  void short_weekday_names(const input_collection_type& short_weekday_names);
  void long_weekday_names(const input_collection_type& long_weekday_names);
  void short_month_names(const input_collection_type& short_month_names);
  void long_month_names(const input_collection_type& long_month_names);
  OutItrT put(OutItrT next,  std::ios_base& a_ios,
      char_type fill_char, const date_type& d) const ;
  OutItrT put(OutItrT next,  std::ios_base& a_ios,
             char_type fill_char, const duration_type& dd) const ;
  OutItrT put(OutItrT next,  std::ios_base& a_ios,
              char_type fill_char, const month_type& m) const;
  // puts the day of month
  OutItrT put(OutItrT next,  std::ios_base& a_ios,
              char_type fill_char, const day_type& day) const;
  OutItrT put(OutItrT next,  std::ios_base& a_ios,
              char_type fill_char, const day_of_week_type& dow) const;
  OutItrT put(OutItrT next,  std::ios_base& a_ios,
              char_type fill_char, const period_type& p) const

};
```

Date Input Facet

```
// Input facet that uses format strings to parse dates,
// date durations, and date periods. Like the output facet allows
// for customization of all strings associated with date type
// including days of week, month names, and special values.
template <class date_type,
          class CharT,
          class InItrT = std::istreambuf_iterator<CharT, std::char_traits<CharT> > >
class date_input_facet : public std::locale::facet {
public:
  typedef typename date_type::duration_type duration_type;
  typedef typename date_type::day_of_week_type day_of_week_type;
  typedef typename date_type::day_type day_type;
  typedef typename date_type::month_type month_type;
  typedef typename date_type::year_type year_type;
  typedef basic_time_period<date_type,duration_type> period_type;
  typedef std::basic_string<CharT> string_type;
  typedef CharT                    char_type;
  typedef boost::date_time::period_parser<date_type, CharT>  period_parser_type;
  typedef special_values_parser<date_type,CharT> special_values_parser_type;
  typedef std::vector<std::basic_string<CharT> > input_collection_type;
  typedef format_date_parser<date_type, CharT> format_date_parser_type;


  explicit date_input_facet(::size_t a_ref = 0);

  explicit date_input_facet(const string_type& format,
                            ::size_t a_ref = 0);

  explicit date_input_facet(const string_type& format,
                            const format_date_parser_type& date_parser,
                            const special_values_parser_type& sv_parser,
                            const period_parser_type& per_parser,
                            const date_gen_parser_type& date_gen_parser,
                            ::size_t ref_count = 0);


  void format(const char_type* const format);
  virtual void set_iso_format();
  virtual void set_iso_extended_format();
  void month_format(const char_type* const format);
  void weekday_format(const char_type* const format);
  void year_format(const char_type* const format);

  //set the various strings associated with dates
  void short_weekday_names(const input_collection_type& weekday_names);
  void long_weekday_names(const input_collection_type& weekday_names);
  void short_month_names(const input_collection_type& month_names);
  void long_month_names(const input_collection_type& month_names);

  //allow user specialization of period and special value handling
  void period_parser(period_parser_type period_parser);
  void special_values_parser(special_values_parser_type sv_parser);

  InItrT get(InItrT& from,
             InItrT& to,
             std::ios_base& /*a_ios*/,
             date_type& d) const;
  InItrT get(InItrT& from,
             InItrT& to,
             std::ios_base& /*a_ios*/,
             month_type& m) const;
  InItrT get(InItrT& from,
             InItrT& to,
             std::ios_base& /*a_ios*/,
             day_of_week_type& wd) const;
  InItrT get(InItrT& from,
             InItrT& to,
             std::ios_base& /*a_ios*/,
             day_type& d) const;
  InItrT get(InItrT& from,
             InItrT& to,
             std::ios_base& /*a_ios*/,
             year_type& y) const;
  InItrT get(InItrT& from,
             InItrT& to,
             std::ios_base& a_ios,
             duration_type& dd) const;
  InItrT get(InItrT& from,
             InItrT& to,
             std::ios_base& a_ios,
             period_type& p) const;
};
```

Time Output Facet

```
/* Facet used for format-based output of date_time types
 * This class provides for the use of format strings to output times.  In addition
 * to the flags for formatting date elements, the following are the allowed format flags:
 *  - %x %X => default format - enables addition of more flags to default (ie. "%x %X %z")
 *  - %f => fractional seconds ".123456"
 *  - %F => fractional seconds or none: like frac sec but empty if frac sec == 0
 *  - %s => seconds w/ fractional sec "02.123" (this is the same as "%S%f)
 *  - %S => seconds "02"
 *  - %z => abbreviated time zone "EDT"
 *  - %Z => full time zone name "Eastern Daylight Time"
 */
template <class time_type,
          class CharT,
          class OutItrT = std::ostreambuf_iterator<CharT, std::char_traits<CharT> > >
class time_facet :
  public date_facet<typename time_type::date_type , CharT, OutItrT> {
 public:
  typedef typename time_type::date_type date_type;
  typedef typename time_type::time_duration_type time_duration_type;
  typedef basic_time_period<time_type,time_duration_type> period_type;
  typedef date_facet<typename time_type::date_type, CharT, OutItrT> base_type;
  typedef typename base_type::string_type string_type;
  typedef typename base_type::char_type   char_type;
  typedef typename base_type::period_formatter_type period_formatter_type;
  typedef typename base_type::special_values_formatter_type special_values_formatter_type;

  explicit time_facet(::size_t a_ref = 0);
  explicit time_facet(const char_type* a_format,
                      period_formatter_type period_formatter = period_formatter_type(),
                      const special_values_formatter_type& special_value_formatter = special_value
                       ::size_t a_ref = 0);

  // Changes format for time_duration
  void time_duration_format(const char_type* const format);
  virtual void set_iso_format();
  virtual void set_iso_extended_format();

  //write out a date_time time point
  OutItrT put(OutItrT a_next,
              std::ios_base& a_ios,
              char_type a_fill,
              const time_type& a_time) const;
  //write out a duration
  OutItrT put(OutItrT a_next,
              std::ios_base& a_ios,
              char_type a_fill,
              const time_duration_type& a_time_dur) const;

  //wirte out a date_time period
  OutItrT put(OutItrT next, std::ios_base& a_ios,
              char_type fill, const period_type& p) const;


};
```

Time Input Facet

```
// Facet for format-based input of date_time types
//default date_time format is YYYY-Mon-DD HH:MM:SS[.fff...][ zzz]
//default time_duration format is %H:%M:%S%F HH:MM:SS[.fff...]
template <class time_type,
          class CharT,
          class InItrT = std::istreambuf_iterator<CharT, std::char_traits<CharT> > >
class time_input_facet :
  public date_input_facet<typename time_type::date_type , CharT, InItrT> {
  public:
    typedef typename time_type::date_type date_type;
    typedef typename time_type::time_duration_type time_duration_type;
    typedef typename time_duration_type::fractional_seconds_type fracional_seconds_type;
    typedef basic_time_period<time_type,time_duration_type> period_type;
    typedef date_input_facet<typename time_type::date_type, CharT, InItrT> base_type;
    typedef typename base_type::duration_type date_duration_type;
    typedef typename base_type::year_type year_type;
    typedef typename base_type::month_type month_type;
    typedef typename base_type::day_type day_type;
    typedef typename base_type::string_type string_type;
    typedef typename string_type::const_iterator const_itr;
    typedef typename base_type::char_type   char_type;
    typedef typename base_type::format_date_parser_type format_date_parser_type;
    typedef typename base_type::period_parser_type period_parser_type;
    typedef typename base_type::special_values_parser_type special_values_parser_type;
    typedef typename base_type::date_gen_parser_type date_gen_parser_type;
    typedef typename base_type::special_values_parser_type::match_results match_results;

    explicit time_input_facet(const string_type& format, ::size_t a_ref = 0);

    explicit time_input_facet(const string_type& format,
                              const format_date_parser_type& date_parser,
                              const special_values_parser_type& sv_parser,
                              const period_parser_type& per_parser,
                              const date_gen_parser_type& date_gen_parser,
                              ::size_t a_ref = 0);

    InItrT get(InItrT& sitr,
               InItrT& stream_end,
               std::ios_base& a_ios,
               time_duration_type& td) const;

    InItrT get(InItrT& sitr,
               InItrT& stream_end,
               std::ios_base& a_ios,
               time_type& t) const;

    InItrT get_local_time(InItrT& sitr,
                          InItrT& stream_end,
                          std::ios_base& a_ios,
                          time_type& t,
                          string_type& tz_str) const;

    InItrT get(InItrT& sitr,
               InItrT& stream_end,
               std::ios_base& a_ios,
               time_type& t,
               string_type& tz_str,
               bool time_is_local) const;

};
```

Open Issues

Timezones and char type

What should be done with char types and timezones? Should there be a wposix_time_zone? If so then how do we deal with this impact on the local_date_time? This needs to be templatized by char type. But all the timezone designations currently use narrow strings. Is this complication worth it?

Acknowledgments

First thanks goes to the Boost Community for all the constructive suggestions for evolving Boost Date-Time Library [http://www.boost.org/libs/date_time/index.html] into a great C++ date-time library. Special thanks goes to my family for allowing me to work on this.

References

- Fowler, Martin "Patterns for things that change with time" [http://martinfowler.com/ap2/timeNarrative.html] .
- Boost Date-Time Library [http://www.boost.org/libs/date_time/index.html]
- Network Time Protocol [http://www.ntp.org]
- RFC 822 Date-Time Specification [http://www.w3.org/Protocols/rfc822#z28]
- IETF Timezone Draft [http://mirrors.isc.org/pub/www.watersprings.org/pub/id/draft-ietf-dhc-timezone-03.txt]
- ISO 8601 standard [http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=26780]
- Another IETF DHCP Extension Draft [http://www.join.uni-muenster.de/Dokumente/drafts/draft-ietf-dhc-v6exts-08.txt]
- IETF Timezone Draft [http://mirrors.isc.org/pub/www.watersprings.org/pub/id/draft-ietf-dhc-timezone-03.txt]
- Henney, Kevlin Object of Value (pdf) [http://www.two-sdg.demon.co.uk/curbralan/papers/ObjectsOfValue.pdf].
- Langer on C++ internationalization [http://www.langer.camelot.de/Articles/Cuj/Internationalization/I18N.html]
- Leap Second Discussion at US Navy Website. [http://tycho.usno.navy.mil/leapsec.html].