
Boost.Date_Time

Jeff Garland

Copyright © 2001-2005 CrystalClear Software, Inc

Subject to the Boost Software License, Version 1.0. (See accompanying file `LICENSE_1_0.txt` or copy at http://www.boost.org/LICENSE_1_0.txt)

Table of Contents

Conceptual	3
Motivation	3
Domain Concepts	3
Design Concepts	4
General Usage Examples	5
Gregorian	7
Date	7
Date Duration (aka Days)	15
Date Period	19
Date Iterators	26
Date Generators/Algorithms	29
Gregorian Calendar	32
Posix Time	34
Ptime	34
Time Duration	44
Time Period	54
Time Iterators	61
Local Time	64
Time Zone (abstract)	64
Posix Time Zone	66
Time Zone Database	71
Custom Time Zone	74
Local Date Time	81
Local Time Period	89
Date Time Input/Output	97
Format Flags	97
Date Facet	106
Date Input Facet	111
Time Facet	116
Time Input Facet	119
Date Time Formatter/Parser Objects	121
Date Time IO Tutorial	140
Serialization	147
Details	149
Calculations	149
Design Goals	151
Tradeoffs: Stability, Predictability, and Approximations	153
Terminology	155
References	156
Build-Compiler Information	158
Tests	160
Change History	161
Acknowledgements	175
Examples	176

Dates as Strings	176
Days Alive	177
Days Between New Years	177
Last Day of the Months	178
Localization Demonstration	179
Date Period Calculations	181
Print Holidays	183
Print Month	185
Month Adding	186
Time Math	187
Print Hours	188
Local to UTC Conversion	189
Time Periods	191
Simple Time Zones	192
Daylight Savings Calc Rules	194
Flight Time Example	195
Seconds Since Epoch	196
Library Reference	198
Date Time Reference	198
Gregorian Reference	364
Posix Time Reference	389
Local Time Reference	404

Introduction

A set of date-time libraries based on generic programming concepts.

This documentation is also available in PDF format. It can be found at:

http://www.crystalclearsoftware.com/libraries/date_time/date_time.pdf

In addition, a full doxygen reference can be found at:

http://www.crystalclearsoftware.com/libraries/date_time/ref_guide/index.html

The most current version of the documentation can be found at:

http://www.crystalclearsoftware.com/libraries/date_time/index.html

Conceptual

Motivation

The motivation for this library comes from working with and helping build several date-time libraries on several projects. Date-time libraries provide fundamental infrastructure for most development projects. However, most of them have limitations in their ability to calculate, format, convert, or perform some other functionality. For example, most libraries do not correctly handle leap seconds, provide concepts such as infinity, or provide the ability to use high resolution or network time sources. These libraries also tend to be rigid in their representation of dates and times. Thus customized policies for a project or subproject are not possible.

Programming with dates and times should be almost as simple and natural as programming with strings and integers. Applications with lots of temporal logic can be radically simplified by having a robust set of operators and calculation capabilities. Classes should provide the ability to compare dates and times, add lengths or time durations, retrieve dates and times from clocks, and work naturally with date and time intervals.

Another motivation for development of the library was to apply modern C++ library design techniques to the date-time domain. Really to build a framework for the construction of building temporal types. For example, by providing iterators and traits classes to control fundamental properties of the library. To the authors knowledge this library is the only substantial attempt to apply modern C++ to a date-time library.

Domain Concepts

The date time domain is rich in terminology and problems. The following is a brief introduction to the concepts you will find reflected in the library.

The library supports 3 basic temporal types:

- **Time Point** -- Specifier for a location in the time continuum.
- **Time Duration** -- A length of time unattached to any point on the time continuum.
- **Time Interval** -- A duration of time attached to a specific point in the time continuum. Also known as a time period.

Each of these temporal types has a **Resolution** which is defined by the smallest representable duration. A **Time system** provides all these categories of temporal types as well as the rules for labeling and calculating with time points. **Calendar Systems** are simply time systems with a maximum resolution of one day. The **Gregorian** system is the most widely used calendar system today (the ISO system is basically a derivative of this). However, there are many other calendar systems as well. **UTC (Coordinated Universal Time)** is a widely used civil time system. UTC is adjusted for earth rotation at longitude 0 by the use of leap seconds (This is not predictable, only as necessary). Most **local time** systems are based on 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.

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.

The library provides support for calculating with dates and times. However, time calculations are not quite the same as calculating with integers. If you are serious about the accuracy of your time calculations need to read about [Stability, Predictability, and Approximations](#).

- [Basic Terminology](#)
- [Calculations](#)
- [Stability, Predictability, and Approximations](#)
- [References](#)

Design Concepts

A large part of the genesis of this library has been the observation that few date-time libraries are built in a fashion that allows customization and extension. A typical example, the calendar logic is built directly into the date class. Or the clock retrieval functions are built directly into the time class. These design decisions usually make it impossible to extend or change the library behavior. At a more fundamental level, there are usually assumptions about the resolution of time representation or the gregorian calendar.

Often times, the result is that a project must settle for a less than complete library because of a requirement for high resolution time representation or other assumptions that do not match the implementation of the library. This is extremely unfortunate because development of a library of this sort is far from a trivial task.

While the design is far from perfect the current design is far more flexible than any date-time library the author is aware of. It is expected that the various aspects of extensibility will be better documented in future versions. Information about the design goals of the library is [summarized here](#).

General Usage Examples

The following provides some sample usage of dates. See [Date Programming](#) for more details.

```
using namespace boost::gregorian;
date weekstart(2002, Feb, 1);
date weekend = weekstart + weeks(1);
date d2 = d1 + days(5);
date today = day_clock::local_day();
if (d2 >= today) {} //date comparison operators

date_period thisWeek(d1, d2);
if (thisWeek.contains(today)) {} //do something

//iterate and print the week
day_iterator itr(weekstart);
while (itr <= weekend) {
    std::cout << (*itr) << std::endl;
    ++itr;
}
//input streaming
std::stringstream ss("2004-Jan-1");
ss >> d3;

//date generator functions
date d5 = next_weekday(d4, Sunday); //calculate Sunday following d4

//US labor day is first Monday in Sept
typedef nth_day_of_the_week_in_month nth_dow;
nth_dow labor_day(nth_dow::first, Monday, Sep);
//calculate a specific date for 2004 from functor
date d6 = labor_day.get_date(2004);
┘
```

The following provides some example code using times. See [Time Programming](#) for more details.

```
using namespace boost::posix_time;
date d(2002, Feb, 1); //an arbitrary date
ptime t1(d, hours(5)+nanosec(100)); //date + time of day offset
ptime t2 = t1 - minutes(4)+seconds(2);
ptime now = second_clock::local_time(); //use the clock
date today = now.date(); //Get the date part out of the time
date tomorrow = today + date_duration(1);
ptime tomorrow_start(tomorrow); //midnight

//input streaming
std::stringstream ss("2004-Jan-1 05:21:33.20");
ss >> t2;

//starting at current time iterator adds by one hour
time_iterator titr(now, hours(1));
for (; titr < tomorrow_start; ++titr) {
    std::cout << (*titr) << std::endl;
}
┘
```

The following provides some example code using times. See [Local Time Programming](#) for more details.

```
using namespace boost::local_time;
//setup some timezones for creating and adjusting times
//first time zone uses the time zone file for regional timezone definitions
tz_database tz_db;
tz_db.load_from_file("date_time_zonespec.csv");
time_zone_ptr nyc_tz = tz_db.time_zone_from_region("America/New_York");
//This timezone uses a posix time zone string definition to create a time zone
time_zone_ptr phx_tz(new posix_time_zone("MST-07:00:00"));

//local departure time in phoenix is 11 pm on April 2 2005
// Note that New York changes to daylight savings on Apr 3 at 2 am)
local_date_time phx_departure(date(2005, Apr, 2), hours(23), phx_tz,
                               local_date_time::NOT_DATE_TIME_ON_ERROR);

time_duration flight_length = hours(4) + minutes(30);
local_date_time phx_arrival = phx_departure + flight_length;
//convert the phx time to a nyz time
local_date_time nyc_arrival = phx_arrival.local_time_in(nyc_tz);

//2005-Apr-03 06:30:00 EDT
std::cout << nyc_arrival << std::endl;
┘
```

Gregorian

Gregorian Date System

[Introduction](#) -- [Usage Examples](#)

Introduction

The gregorian date system provides a date programming system based the Gregorian Calendar. The first introduction of the Gregorian calendar was in 1582 to fix an error in the Julian Calendar. However, many local jurisdictions did not adopt this change until much later. Thus there is potential confusion with historical dates.

The implemented calendar is a "proleptic Gregorian calendar" which extends dates back prior to the Gregorian Calendar's first adoption in 1582. The current implementation supports dates in the range 1400-Jan-01 to 9999-Dec-31. Many references will represent dates prior to 1582 using the Julian Calendar, so caution is in order if accurate calculations are required on historic dates. See [Calendrical Calculations](#) by Reingold & Dershowitz for more details. Date information from Calendrical Calculations has been used to cross-test the correctness of the Gregorian calendar implementation.

All types for the gregorian system are found in namespace `boost::gregorian`. The library supports a convenience header `boost/date_time/gregorian/gregorian_types.hpp` that will include all the classes of the library with no input/output dependency. Another header `boost/date_time/gregorian/gregorian.hpp` will include the types and the input/output code.

The class `boost::gregorian::date` is the primary temporal type for users. If you are interested in learning about writing programs that do specialized date calculations such as finding the "first sunday in april" see the date [generators and algorithms page](#).

Usage Examples

Example	Description
Days Alive Days Between New Years	Simple date arithmetic. Retrieve current day from clock.
Dates as strings	Simple parsing and formatting of dates from/to strings
Date Period Calculations	See if a date is in a set of date periods (eg: is it a holiday/week-end)
Print a month	Small utility program which prints out all the days in a month from command line. Need to know if 1999-Jan-1 was a Friday or a Saturday? This program shows how to do it.
Print Holidays	Uses date generators to convert abstract specification into concrete set of dates.

Date

[Introduction](#) -- [Header](#) -- [Construction](#) -- [Construct from String](#) -- [Construct from Clock](#) -- [Accessors](#) -- [Convert to String](#) -- [Operators](#) -- [Struct tm Functions](#)

Introduction

The class `boost::gregorian::date` is the primary interface for date programming. In general, the date class is immutable once constructed although it does allow assignment from another date. Techniques for creating dates include reading the [current date from the clock](#), using [date iterators](#), and [date algorithms or generators](#).

Internally `boost::gregorian::date` is stored as a 32 bit integer type. The class is specifically designed to NOT contain virtual functions. This design allows for efficient calculation and memory usage with large collections of dates.

The construction of a date validates all input so that it is not possible to construct an 'invalid' date. That is 2001-Feb-29 cannot be constructed as a date. Various exceptions derived from `std::out_of_range` are thrown to indicate which aspect of the date input is invalid. Note that the special value `not-a-date-time` can be used as 'invalid' or 'null' date if so desired.

Header

```
#include "boost/date_time/gregorian/gregorian.hpp" //include all types plus i/o
or
#include "boost/date_time/gregorian/gregorian_types.hpp" //no i/o just types
```

Construction

Syntax	Description
	Example
<code>date(greg_year, greg_month, greg_day)</code>	Construct from parts of date. Throws <code>bad_year</code> , <code>bad_day_of_month</code> , or <code>bad_day_month</code> (derivatives of <code>std::out_of_range</code>) if the year, month or day are out of range.
	<pre>date d(2002,Jan,10);</pre>
<code>date(date d)</code>	Copy constructor
	<pre>date d1(d);</pre>
<code>date(special_values sv)</code>	Constructor for infinities, not-a-date-time, <code>max_date_time</code> , and <code>min_date_time</code>
	<pre>date d1(neg_infin); date d2(pos_infin); date d3(not_a_date_time); date d4(max_date_time); date d5(min_date_time);</pre>
<code>date()</code>	Default constructor. Creates a date object initialized to <code>not_a_date_time</code> . NOTE: this constructor can be disabled by defining <code>DATE_TIME_NO_DEFAULT_CONSTRUCTOR</code> (see <code>compiler_config.hpp</code>)
	<pre>date d; // d => not_a_date_time</pre>

Construct from String

Syntax	Description
	Example
<pre>date from_string(std::string)</pre>	<p>From delimited date string where with order year-month-day eg: 2002-1-25</p> <pre>std::string ds("2002/1/25"); date d(from_string(ds));</pre>
<pre>date from_undelimited_string(std::string)</pre>	<p>From iso type date string where with order year-month-day eg: 20020125</p> <pre>std::string ds("20020125"); date d(from_undelimited_string(ds));</pre>

Construct from Clock

Syntax	Description
	Example
<pre>day_clock::local_day()</pre>	<p>Get the local day based on the time zone settings of the computer.</p> <pre>date d(day_clock::local_day());</pre>
<pre>day_clock::universal_day()</pre>	<p>Get the UTC day.</p> <pre>date d(day_clock::universal_day());</pre>

Accessors

Syntax	Description
	Example
<code>greg_year year() const</code>	<p>Get the year part of the date.</p> <pre>date d(2002,Jan,10); d.year(); // --> 2002</pre>
<code>greg_month month() const</code>	<p>Get the month part of the date.</p> <pre>date d(2002,Jan,10); d.month(); // --> 1</pre>
<code>greg_day day() const</code>	<p>Get the day part of the date.</p> <pre>date d(2002,Jan,10); d.day(); // --> 10</pre>
<code>greg_ymd year_month_day() const</code>	<p>Return a <code>year_month_day</code> struct. More efficient when all 3 parts of the date are needed.</p> <pre>date d(2002,Jan,10); date::ymd_type ymd = d.year_month_day(); // ymd.year --> 2002, // ymd.month --> 1, // ymd.day --> 10</pre>
<code>greg_day_of_week day_of_week() const</code>	<p>Get the day of the week (Sunday, Monday, etc.)</p> <pre>date d(2002,Jan,10); d.day(); // --> Thursday</pre>
<code>greg_day_of_year day_of_year() const</code>	<p>Get the day of the year. Number from 1 to 366</p> <pre>date d(2000,Jan,10); d.day_of_year(); // --> 10</pre>
<code>date end_of_month() const</code>	<p>Returns a <code>date</code> object set to the last day of the calling objects current month.</p> <pre>date d(2000,Jan,10); d.end_of_month(); // --> 2000-Jan-31</pre>

Syntax	Description
	Example
<code>bool is_infinity() const</code>	<p>Returns true if date is either positive or negative infinity</p> <pre>date d(pos_infin); d.is_infinity(); // --> true</pre>
<code>bool is_neg_infinity() const</code>	<p>Returns true if date is negative infinity</p> <pre>date d(neg_infin); d.is_neg_infinity(); // --> true</pre>
<code>bool is_pos_infinity() const</code>	<p>Returns true if date is positive infinity</p> <pre>date d(neg_infin); d.is_pos_infinity(); // --> true</pre>
<code>bool is_not_a_date() const</code>	<p>Returns true if value is not a date</p> <pre>date d(not_a_date_time); d.is_not_a_date(); // --> true</pre>
<code>bool is_special() const</code>	<p>Returns true if date is any special_value</p> <pre>date d(pos_infin); date d2(not_a_date_time); date d3(2005,Mar,1); d.is_special(); // --> true d2.is_special(); // --> true d3.is_special(); // --> false</pre>
<code>special_value as_special() const</code>	<p>Returns represented special_value or not_special if the represented date is a normal date.</p> <pre></pre>
<code>long modjulian_day() const</code>	<p>Returns the modified julian day for the date.</p> <pre></pre>
<code>long julian_day() const</code>	<p>Returns the julian day for the date.</p> <pre></pre>

Syntax	Description
	Example
<code>int week_number() const</code>	Returns the ISO 8601 week number for the date.
<code>date end_of_month() const</code>	Returns the last day of the month for the date. <pre>date d(2000, Feb, 1); //gets Feb 29 -- 2000 was leap year date eom = d.end_of_month();</pre>

Convert to String

Syntax	Description
	Example
<code>std::string to_simple_string(date d)</code>	To YYYY-mm-DD string where mm is a 3 char month name. <pre>"2002-Jan-01"</pre>
<code>std::string to_iso_string(date d)</code>	To YYYYMMDD where all components are integers. <pre>"20020131"</pre>
<code>std::string to_iso_extended_string(date d)</code>	To YYYY-MM-DD where all components are integers. <pre>"2002-01-31"</pre>

Operators

Syntax	Description
	Example
<code>operator<<</code>	Stream output operator <pre>date d(2002,Jan,1); std::cout << d << std::endl;</pre>
<code>operator>></code>	Stream input operator. Note: As of version 1.33, streaming operations have been greatly improved. See Date Time IO System for details on exceptions and error conditions. <pre>date d(not_a_date_time); stringstream ss("2002-Jan-01"); ss >> d;</pre>
<code>operator==, operator!=, operator>, operator<, operator>=, operator<=</code>	A full complement of comparison operators <pre>d1 == d2, etc</pre>
<code>date operator+(date_duration) const</code>	Return a date adding a day offset <pre>date d(2002,Jan,1); date_duration dd(1); date d2 = d + dd;</pre>
<code>date operator-(date_duration) const</code>	Return a date by subtracting a day offset <pre>date d(2002,Jan,1); date_duration dd(1); date d2 = d - dd;</pre>
<code>date_duration operator-(date) const</code>	Return a date_duration by subtracting two dates <pre>date d1(2002,Jan,1); date d2(2002,Jan,2); date_duration dd = d2-d1;</pre>

Struct tm Functions

Functions for converting a date object to, and from, a tm struct are provided.

Syntax	Description
<pre>tm to_tm(date)</pre>	Example
	<p>A function for converting a date object to a tm struct. The fields: tm_hour, tm_min, and tm_sec are set to zero. The tm_isdst field is set to -1.</p> <pre> date d(2005,Jan,1); tm d_tm = to_tm(d); /* tm_year => 105 tm_mon => 0 tm_mday => 1 tm_wday => 6 (Saturday) tm_yday => 0 tm_hour => 0 tm_min => 0 tm_sec => 0 tm_isdst => -1 */ </pre>
<pre>date date_from_tm(tm datetm)</pre>	<p>A function for converting a tm struct to a date object. The fields: tm_wday ,tm_yday ,tm_hour,tm_min,tm_sec, and tm_isdst are ignored.</p>
	<pre> tm d_tm; d_tm.tm_year = 105; d_tm.tm_mon = 0; d_tm.tm_mday = 1; date d = date_from_tm(d_tm); // d => 2005-Jan-01 </pre>

Date Duration (aka Days)

[Introduction](#) -- [Header](#) -- [Construction](#) -- [Accessors](#) -- [Operators](#) -- [Additional Duration Types](#)

Introduction

The class `boost::gregorian::date_duration` is a simple day count used for arithmetic with [gregorian::date](#). A duration can be either positive or negative.

As of version 1_32 the `date_duration` class has been typedef'd as `days` in the `boost::gregorian` namespace. Throughout the examples you will find `days` used instead of `date_duration`.

Header

```

#include "boost/date_time/gregorian/gregorian.hpp" //include all types plus i/o
or
#include "boost/date_time/gregorian/gregorian_types.hpp" //no i/o just types

```

Construction

Syntax	Description
	Example
<code>date_duration(long)</code>	Create a duration count. <pre>date_duration dd(3); // 3 days</pre>
<code>days(special_values sv)</code>	Constructor for infinities, not-a-date-time, max_date_time, and min_date_time <pre>days dd1(neg_infin); days dd2(pos_infin); days dd3(not_a_date_time); days dd4(max_date_time); days dd5(min_date_time);</pre>

Accessors

Syntax	Description
	Example
<code>long days() const</code>	Get the day count. <pre>date_duration dd(3); dd.days() --> 3</pre>
<code>bool is_negative() const</code>	True if number of days is less than zero. <pre>date_duration dd(-1); dd.is_negative() --> 1 true</pre>
<code>static date_duration unit()</code>	Return smallest possible unit of duration type. <pre>date_duration::unit() --> date_duration(1)</pre>
<code>bool is_special() const</code>	Returns true if days is any special_value <pre>days dd(pos_infin); days dd2(not_a_date_time); days dd3(25); dd.is_special(); // --> true dd2.is_special(); // --> true dd3.is_special(); // --> false</pre>

Operators

Syntax	Description
	Example
<code>operator<<, operator>></code>	Streaming operators. Note: As of version 1.33, streaming operations have been greatly improved. See Date Time IO System for more details (including exceptions and error conditions). <pre>date d(not_a_date_time); stringstream ss("2002-Jan-01"); ss >> d; std::cout << d; // "2002-Jan-01"</pre>
<code>operator==, operator!=, operator>, operator<, operator>=, operator<=</code>	A full complement of comparison operators <pre>dd1 == dd2, etc</pre>
<code>date_duration operator+(date_duration) const</code>	Add date durations. <pre>date_duration dd1(3); date_duration dd2(5); date_duration dd3 = dd1 + dd2;</pre>
<code>date_duration operator-(date_duration) const</code>	Subtract durations. <pre>date_duration dd1(3); date_duration dd2(5); date_duration dd3 = dd1 - dd2;</pre>

Additional Duration Types

These additional types are logical representations of spans of days.

Syntax	Description
<pre>months(int num_of_months)</pre>	<p>Example</p> <p>A logical month representation. Depending on the usage, this months object may cover a span of 28 to 31 days. The objects also use a snap to end-of-month behavior when used in conjunction with a date that is the last day of a given month. WARNING: this behavior may lead to unexpected results. See: Reversibility of Operations Pitfall for complete details and alternatives.</p> <pre>months single(1); date leap_year(2004,Jan,31); date norm_year(2005,Jan,31); leap_year + single; // => 2004-Feb-29 norm_year + single; // => 2005-Feb-28 date(2005,Jan,1) + single; // => 2005-Feb-01 date(2005,Jan,31) + single; // => 2005-Feb-01 date(2005,Jan,1) + single; // => 2005-Mar-01 date(2005,Jan,28) + single; // => 2005-Mar-31</pre>
<pre>years(int num_of_years)</pre>	<p>A logical representation of a year. The years object has the same behavior as the months objects with regards to the end-of-the-month.</p> <pre>years single(1); date(2003,Jan,28) + single; // results in => 2004-Feb-29 date(2004,Jan,29) + single; // results in => 2005-Feb-28</pre>
<pre>weeks(int num_of_weeks)</pre>	<p>A duration type representing a number of $n * 7$ days.</p> <pre>weeks single(1); date(2005,Jan,1) + single; // => 2005-Jan-08</pre>

Reversibility of Operations Pitfall

A natural expectation when adding a number of months to a date, and then subtracting the same number of months, is to end up exactly where you started. This is most often the result the `date_time` library provides but there is one significant exception: The snap-to-end-of-month behavior implemented by the `months` duration type. The `months` duration type may provide unexpected results when the starting day is the 28th, 29th, or 30th in a 31 day month. The `month_iterator` is not affected by this issue and is therefore included in the examples to illustrate a possible alternative.

When the starting date is in the middle of a month, adding or subtracting any number of months will result in a date that is the same day of month (e.g. if you start on the 15th, you will end on the 15th). When a date is the last day of the month, adding or subtracting any number of months will give a result that is also the last day of the month (e.g if you start on Jan 31st, you will land on: Feb 28th, Mar 31st, etc).

```
// using months duration type
date d(2005, Nov, 30); // last day of November
d + months(1); // result is last day of December "2005-Dec-31"
d - months(1); // result is last day of October "2005-Oct-31"

// using month_iterator
month_iterator itr(d); // last day of November
++itr; // result is last day of December "2005-Dec-31"
--itr; // back to original starting point "2005-Nov-30"
--itr; // last day of October "2005-Oct-31"
└
```

If the start date is the 28th, 29th, or 30th in a 31 day month, the result of adding or subtracting a month may result in the snap-to-end-of-month behavior kicking in unexpectedly. This would cause the final result to be different than the starting date.

```
// using months duration type
date d(2005, Nov, 29);
d += months(1); // "2005-Dec-29"
d += months(1); // "2006-Jan-29"
d += months(1); // "2006-Feb-28" --> snap-to-end-of-month behavior kicks in
d += months(1); // "2006-Mar-31" --> unexpected result
d -= months(4); // "2005-Nov-30" --> unexpected result, not where we started

// using month_iterator
month_iterator itr(date(2005, Dec, 30));
++itr; // "2006-Jan-30" --> ok
++itr; // "2006-Feb-28" --> snap-to DOES NOT kick in
++itr; // "2006-Mar-30" --> ok
--itr; // "2006-Feb-28" --> ok
--itr; // "2006-Jan-30" --> ok
--itr; // "2005-Dec-30" --> ok, back where we started
└
```

The additional duration types (months, years, and weeks) are provided as a convenience and can be easily removed to insure this pitfall never occurs. To remove these types simply undefine `BOOST_DATE_TIME_OPTIONAL_GREGORIAN_TYPES`.

Date Period

[Introduction](#) -- [Header](#) -- [Construction](#) -- [Mutators](#) -- [Accessors](#) -- [Convert to String](#) -- [Operators](#)

Introduction

The class `boost::gregorian::date_period` provides direct representation for ranges between two dates. Periods provide the ability to simplify some types of calculations by simplifying the conditional logic of the program. For example, testing if a date is within an irregular schedule such as a weekend or holiday can be accomplished using collections of date periods. This is facilitated by several methods that allow evaluation if a `date_period` intersects with another date period, and to generate the period resulting from the intersection. The [date period calculation example](#) provides an example of this.

A period that is created with beginning and end points being equal, or with a duration of zero, is known as a zero length period. Zero length periods are considered invalid (it is perfectly legal to construct an invalid period). For these periods, the `last` point will always be one unit less than the `begin` point.

Date periods used in combination with infinity values have the ability to represent complex concepts such as 'until further notice'.

Header

```
#include "boost/date_time/gregorian/gregorian.hpp" //include all types plus i/o
or
#include "boost/date_time/gregorian/gregorian_types.hpp" //no i/o just types
```

Construction

Syntax	Description
	Example
<pre>date_period(date, date)</pre>	<p>Create a period as [begin, end). If end is <= begin then the period will be invalid.</p> <pre>date_period dp(date(2002,Jan,10), date(2002,Jan,12));</pre>
<pre>date_period(date, days)</pre>	<p>Create a period as [begin, begin+len) where end point would be begin+len. If len is <= zero then the period will be defined as invalid.</p> <pre>date_period dp(date(2002,Jan,10), days(2));</pre>
<pre>date_period(date_period)</pre>	<p>Copy constructor</p> <pre>date_period dp1(dp);</pre>

Mutators

Syntax	Description
	Example
<pre>date_period shift(days)</pre>	<p>Add duration to both begin and end.</p> <pre>date_period dp(date(2005,Jan,1), days(3)); dp.shift(days(3)); // dp == 2005-Jan-04 to 2005-Jan-07 └┐</pre>
<pre>date_period expand(days)</pre>	<p>Subtract duration from begin and add duration to end.</p> <pre>date_period dp(date(2005,Jan,2), days(2)); dp.expand(days(1)); // dp == 2005-Jan-01 to 2005-Jan-04 └┐</pre>

Accessors

Syntax	Description
	Example
<code>date begin()</code>	<p>Return first day of period.</p> <pre>date_period dp(date(2002,Jan,1), date(2002,Jan,10)); dp.begin() --> 2002-Jan-01</pre>
<code>date last()</code>	<p>Return last date in the period</p> <pre>date_period dp(date(2002,Jan,1), date(2002,Jan,10)); dp.last() --> 2002-Jan-09</pre>
<code>date end()</code>	<p>Return one past the last in period</p> <pre>date_period dp(date(2002,Jan,1), date(2002,Jan,10)); dp.end() --> 2002-Jan-10</pre>
<code>days length()</code>	<p>Return the length of the date_period</p> <pre>date_period dp(date(2002,Jan,1), days(2)); dp.length() --> 2</pre>
<code>bool is_null()</code>	<p>True if period is not well formed. eg: end less than or equal to begin.</p> <pre>date_period dp(date(2002,Jan,10), date(2002,Jan,1)); dp.begin() --> true</pre>
<code>bool contains(date)</code>	<p>True if date is within the period. Zero length periods cannot contain any points</p> <pre>date d(2002,Jan,1); date_period dp(d, date(2002,Jan,10)); dp.contains(date(2002,Jan,2)); // true date_period dp2(d, d); dp2.contains(date(2002,Jan,1)); // false</pre>

Syntax	Description
	Example
<code>bool contains(date_period)</code>	<p>True if date period is within the period</p> <pre> date_period dp1(date(2002,Jan,1), date(2002,Jan,10)); date_period dp2(date(2002,Jan,2), date(2002,Jan,3)); dp1.contains(dp2) --> true dp2.contains(dp1) --> false </pre>
<code>bool intersects(date_period)</code>	<p>True if periods overlap</p> <pre> date_period dp1(date(2002,Jan,1), date(2002,Jan,10)); date_period dp2(date(2002,Jan,2), date(2002,Jan,3)); dp2.intersects(dp1) --> true </pre>
<code>date_period intersection(date_period)</code>	<p>Calculate the intersection of 2 periods. Null if no intersection.</p> <pre> date_period dp1(date(2002,Jan,1), date(2002,Jan,10)); date_period dp2(date(2002,Jan,2), date(2002,Jan,3)); dp2.intersection(dp1) --> dp2 </pre>
<code>date_period is_adjacent(date_period)</code>	<p>Check if two periods are adjacent, but not overlapping.</p> <pre> date_period dp1(date(2002,Jan,1), date(2002,Jan,3)); date_period dp2(date(2002,Jan,3), date(2002,Jan,10)); dp2.is_adjacent(dp1) --> true </pre>
<code>date_period is_after(date)</code>	<p>Determine the period is after a given date.</p> <pre> date_period dp1(date(2002,Jan,10), date(2002,Jan,30)); date d(2002,Jan,3); dp1.is_after(d) --> true </pre>
<code>date_period is_before(date)</code>	<p>Determine the period is before a given date.</p> <pre> date_period dp1(date(2002,Jan,1), date(2002,Jan,3)); date d(2002,Jan,10); dp1.is_before(d) --> true </pre>

Syntax	Description
	Example
<code>date_period merge(date_period)</code>	<p>Returns union of two periods. Null if no intersection.</p> <pre> date_period dp1(date(2002,Jan,1), date(2002,Jan,10)); date_period dp2(date(2002,Jan,9), date(2002,Jan,31)); dp2.merge(dp1) // 2002-Jan-01/2002-Jan-31 </pre>
<code>date_period span(date_period)</code>	<p>Combines two periods and any gap between them such that begin = min(p1.begin, p2.begin) and end = max(p1.end , p2.end)</p> <pre> date_period dp1(date(2002,Jan,1), date(2002,Jan,5)); date_period dp2(date(2002,Jan,9), date(2002,Jan,31)); dp2.span(dp1); // 2002-Jan-01/2002-Jan-31 </pre>
<code>date_period shift(days)</code>	<p>Add duration to both begin and end.</p> <pre> date_period dp1(date(2002,Jan,1), date(2002,Jan,10)); dp1.shift(days(1)); // 2002-Jan-02/2002-Jan-11 </pre>
<code>date_period expand(days)</code>	<p>Subtract duration from begin and add duration to end.</p> <pre> date_period dp1(date(2002,Jan,4), date(2002,Jan,10)); dp1.expand(days(2)); // 2002-Jan-02/2002-Jan-12 </pre>

Convert to String

Syntax	Description
	Example
<code>std::string to_simple_string(date_period dp)</code>	<p>To [YYYY-mm-dd/YYYY-mm-dd] string where mm is 3 char month name.</p> <pre>[2002-Jan-01/2002-Jan-31]</pre>

Operators

Syntax	Description
	Example
<code>operator<<</code>	ostream operator for <code>date_period</code> . Uses facet to format time points. Typical output: [2002-Jan-01/2002-Jan-31]. <pre>std::cout << dp << std::endl;</pre>
<code>operator>></code>	istream operator for <code>date_period</code> . Uses facet to parse time points. <pre>"[2002-Jan-01/2002-Jan-31]"</pre>
<code>operator==, operator!=, operator>, operator<</code>	A full complement of comparison operators <pre>dp1 == dp2, etc</pre>
<code>operator<</code>	True if <code>dp1.end()</code> less than <code>dp2.begin()</code> <pre>dp1 < dp2, etc</pre>
<code>operator></code>	True if <code>dp1.begin()</code> greater than <code>dp2.end()</code> <pre>dp1 > dp2, etc</pre>

Date Iterators

[Introduction](#) -- [Header](#) -- [Overview](#)

Introduction

Date iterators provide a standard mechanism for iteration through dates. Date iterators are a model of [Bidirectional Iterator](#) and can be used to populate collections with dates and other date generation tasks. For example, the [print month](#) example iterates through all the days in a month and prints them.

All of the iterators here derive from `boost::gregorian::date_iterator`.

Header

```
#include "boost/date_time/gregorian/gregorian.hpp" //include all types plus i/o
or
#include "boost/date_time/gregorian/gregorian_types.hpp" //no i/o just types
```

Overview

Syntax	Description
<pre>date_iterator</pre>	Example
	<p>Common (abstract) base class for all day level iterators.</p> <pre></pre>
<pre>day_iterator(date start_date, int day_count=1)</pre>	<p>Iterate <code>day_count</code> days at a time. This iterator does not provide postfix increment/decrement operators. Only prefix operators are provided.</p>
	<pre>day_iterator day_itr(date(2005,Jan,1)); ++d_itr; // 2005-Jan-02 day_iterator 2day_itr(date(2005,Feb,1),2); ++2d_itr; // 2005-Feb-03</pre>
<pre>week_iterator(...) Parameters: date start_date int week_offset (defaults to 1)</pre>	<p>Iterate <code>week_offset</code> weeks at a time. This iterator does not provide postfix increment/decrement operators. Only prefix operators are provided.</p>
	<pre>week_iterator wk_itr(date(2005,Jan,1)); ++wk_itr; // 2005-Jan-08 week_iterator 2wk_itr(date(2005,Jan,1),2); ++2wk_itr; // 2005-Feb-15</pre>
<pre>month_iterator(...) Parameters: date start_date int month_offset (defaults to 1)</pre>	<p>Iterate <code>month_offset</code> months. There are special rules for handling the end of the month. These are: if start date is last day of the month, always adjust to last day of the month. If date is beyond the end of the month (e.g. Jan 31 + 1 month) adjust back to end of month (for more details and examples of this, see Reversibility of Operations Pitfall. NOTE: the <code>month_iterator</code> is not effected by this pitfall.) This iterator does not provide postfix increment/decrement operators. Only prefix operators are provided.</p>
	<pre>month_iterator m_itr(date(2005,Jan,1)); ++m_itr; // 2005-Feb-01 month_iterator 2m_itr(date(2005,Feb,1),2); ++2m_itr; // 2005-Apr-01</pre>

Syntax	Description
<pre>year_iterator(...) Parameters: date start_date int year_offset (defaults to 1)</pre>	Example
	<p>Iterate year_offset years. The year_iterator will always land on the day of the date parameter except when date is Feb 28 in a non-leap year. In this case the iterator will return Feb 29 for leap years (eg: 2003-Feb-28, 2004-Feb-29, 2005-Feb-28). This iterator does not provide postfix increment/decrement operators. Only prefix operators are provided.</p> <pre>year_iterator y_itr(date(2005,Jan,1)); ++y_itr; // 2006-Jan-01 year_iterator 2y_itr(date(2005,Feb,1),2); ++2y_itr; // 2007-Feb-01</pre>

Date Generators/Algorithms

Date Generators/Algorithms

[Introduction](#) -- [Header](#) -- [Class Overview](#) -- [Function Overview](#)

Introduction

Date algorithms or generators are tools for generating other dates or schedules of dates. A generator function starts with some part of a date such as a month and day and is supplied another part to then generate a concrete date. This allows the programmer to represent concepts such as "The first Sunday in February" and then create a concrete set of dates when provided with one or more years. *Note:* As of boost version 1_31_0, date generator names have been changed. Old names are still available but are no longer documented and may someday be deprecated

Also provided are stand-alone functions for generating a date, or calculation a duration of days. These functions take a date object and a weekday object as parameters.

All date generator classes and functions are in the `boost::gregorian` namespace.

The [print holidays](#) example shows a detailed usage example.

Header

```
#include "boost/date_time/gregorian/gregorian.hpp"
```

Overview

Class and get_date Parameter	Description
<pre>year_based_generator date get_date(greg_year year)</pre>	<p>Example</p> <p>A unifying (abstract) date_generator base type for: partial_date, nth_day_of_the_week_in_month, first_day_of_the_week_in_month, and last_day_of_the_week_in_month.</p> <p>The print holidays example shows a detailed usage example.</p>
<pre>last_day_of_the_week_in_month(greg_weekday, greg_month) date get_date(greg_year year)</pre>	<p>Calculate something like last Monday of January</p> <pre>last_day_of_the_week_in_month ↵ lwdm(Monday, Jan); date d = lwdm.get_date(2002); //2002-Jan-28</pre>
<pre>first_day_of_the_week_in_month(greg_weekday, greg_month) date get_date(greg_year year)</pre>	<p>Calculate something like first Monday of January</p> <pre>first_day_of_the_week_in_month ↵ fdm(Monday, Jan); date d = fdm.get_date(2002); //2002-Jan-07</pre>
<pre>nth_day_of_the_week_in_month(week_num, greg_weekday, greg_month) date get_date(greg_year year)</pre>	<p>week_num is a public enum member of nth_day_of_the_week_in_month. Calculate something like first Monday of January, second Tuesday of March, Third Sunday of December, etc. (first through fifth are provided, fifth is the equivalent of last)</p> <pre>typedef nth_day_of_the_week_in_month nth_dow; nth_dow ndm(nth_dow::third, Monday, Jan); date d = ndm.get_date(2002); //2002-Jan-21</pre>
<pre>partial_date(greg_day, greg_month) date get_date(greg_year year)</pre>	<p>Generates a date by applying the year to the given month and day.</p> <pre>partial_date pd(1, Jan); date d = pd.get_date(2002); //2002-Jan-01</pre>
<pre>first_day_of_the_week_after(greg_weekday) date get_date(date d)</pre>	<p>Calculate something like First Sunday after Jan 1, 2002</p> <pre>first_day_of_the_week_after fdaf(Monday); date d = fdaf.get_date(date(2002, Jan, 1)); //2002-Jan-07</pre>

Class and get_date Parameter	Description
	Example
<pre>first_day_of_the_week_before(greg_weekday) date get_date(date d)</pre>	<p>Calculate something like First Monday before Feb 1,2002</p> <pre>first_day_of_the_week_before fdbf(Monday); date d = fdbf.get_date(date(2002, Feb, 1)); //2002-Jan-28</pre>

Function Overview

Function Prototype	Description
	Example
<pre>days days_until_weekday date, greg_weekday)</pre>	<p>Calculates the number of days from given date until given weekday.</p> <pre>date d(2004, Jun, 1); // Tuesday greg_weekday gw(Friday); days_until_weekday(d, gw); // 3 days</pre>
<pre>days days_before_weekday(date, greg_weekday)</pre>	<p>Calculates the number of day from given date to previous given weekday.</p> <pre>date d(2004, Jun, 1); // Tuesday greg_weekday gw(Friday); days_before_weekday(d, gw); // 4 days</pre>
<pre>date next_weekday(date, greg_weekday)</pre>	<p>Generates a date object representing the date of the following weekday from the given date.</p> <pre>date d(2004, Jun, 1); // Tuesday greg_weekday gw(Friday); next_weekday(d, gw); // 2004-Jun-4</pre>
<pre>date previous_weekday(date, greg_weekday)</pre>	<p>Generates a date object representing the date of the previous weekday from the given date.</p> <pre>date d(2004, Jun, 1); // Tuesday greg_weekday gw(Friday); previous_weekday(d, gw); // 2004-May-28</pre>

Gregorian Calendar

[Introduction](#) -- [Header](#) -- [Functions](#)

Introduction

The class `boost::gregorian::gregorian_calendar` implements the functions necessary to create the gregorian date system. It converts to the year-month-day form of a date to a day number representation and back.

For most purposes this class is simply accessed by [gregorian::date](#) and is not used directly by the user. However, there are useful functions that might be of use such as the `end_of_month_day` function.

The [print month](#) example demonstrates this.

Header

```
#include "boost/date_time/gregorian/gregorian.hpp" //include all types plus i/o
or
#include "boost/date_time/gregorian/gregorian_types.hpp" //no i/o just types
```

Functions

Syntax	Description
	Example
<code>static short day_of_week(ymd_type)</code>	Return the day of the week (0==Sunday, 1==Monday, etc) See also gregorian::date <code>day_of_week</code>
<code>static date_int_type day_number(ymd_type)</code>	Convert a <code>ymd_type</code> into a day number. The day number is an absolute number of days since the epoch start.
<code>static short end_of_month_day(year_type, month_type)</code>	Given a year and month determine the last day of the month.
<code>static ymd_type from_day_number(date_int_type)</code>	Convert a day number to a ymd struct.
<code>static bool is_leap_year(year_type)</code>	Returns true if specified year is a leap year. <pre>gregorian_calendar::is_leap_year(2000) //--> true</pre>

Posix Time

Posix Time System

[Introduction](#) -- [Usage Examples](#)

Introduction

Defines a non-adjusted time system with nano-second/micro-second resolution and stable calculation properties. The nano-second resolution option uses 96 bits of underlying storage for each ptime while the micro-second resolution uses 64 bits per ptime (see [Build Options](#) for details). This time system uses the Gregorian calendar to implement the date portion of the time representation.

Usage Examples

Example	Description
Time Math	A few simple calculations using ptime and time_durations.
Print Hours	Retrieve time from clock, use a time_iterator.
Local to UTC Conversion	Demonstrates a couple different ways to convert a local to UTC time including daylight savings rules.
Time Periods	Some simple examples of intersection and display of time periods.

Ptime

[Introduction](#) -- [Header](#) -- [Construction](#) -- [Construct from String](#) -- [Construct from Clock](#) -- [Construct using Conversion functions](#) -- [Accessors](#) -- [Conversion To String](#) -- [Operators](#) -- [Struct tm, time_t, and FILETIME Functions](#)

Introduction

The class `boost::posix_time::ptime` is the primary interface for time point manipulation. In general, the ptime class is immutable once constructed although it does allow assignment.

Class ptime is dependent on [gregorian::date](#) for the interface to the date portion of a time point.

Other techniques for creating times include [time iterators](#).

Header

```
#include "boost/date_time/posix_time/posix_time.hpp" //include all types plus i/o
or
#include "boost/date_time/posix_time/posix_time_types.hpp" //no i/o just types
```

Construction

Syntax	Description
	Example
<code>ptime(date,time_duration)</code>	<p>Construct from a date and offset</p> <pre>ptime t1(date(2002,Jan,10), time_duration(1,2,3)); ptime t2(date(2002,Jan,10), hours(1)+nanosec(5));</pre>
<code>ptime(ptime)</code>	<p>Copy constructor</p> <pre>ptime t3(t1)</pre>
<code>ptime(special_values sv)</code>	<p>Constructor for infinities, not-a-date-time, max_date_time, and min_date_time</p> <pre>ptime d1(neg_infin); ptime d2(pos_infin); ptime d3(not_a_date_time); ptime d4(max_date_time); ptime d5(min_date_time);</pre>
<code>ptime;</code>	<p>Default constructor. Creates a ptime object initialized to not_a_date_time. NOTE: this constructor can be disabled by defining <code>DATE_TIME_NO_DEFAULT_CONSTRUCTOR</code> (see <code>compiler_config.hpp</code>)</p> <pre>ptime p; // p => not_a_date_time</pre>

Construct from String

Syntax	Description
<pre>ptime time_from_string(std::string)</pre>	<p>Example</p> <p>From delimited string. NOTE: Excess digits in fractional seconds will be dropped. Ex: "1:02:03.123456999" => 1:02:03.123456. This behavior is affected by the precision the library is compiled with (see Build-Compiler Information).</p> <pre>std::string ts("2002-01-20 23:59:59.000"); ptime t(time_from_string(ts))</pre>
<pre>ptime from_iso_string(std::string)</pre>	<p>From non delimited iso form string.</p> <pre>std::string ts("20020131T235959"); ptime t(from_iso_string(ts))</pre>

Construct from Clock

Syntax	Description
	Example
<code>ptime second_clock::local_time()</code>	Get the local time, second level resolution, based on the time zone settings of the computer.
	<code>ptime t(second_clock::local_time());</code>
<code>ptime second_clock::universal_time()</code>	Get the UTC time.
	<code>ptime t(second_clock::universal_time());</code>
<code>ptime microsec_clock::local_time()</code>	Get the local time using a sub second resolution clock. On Unix systems this is implemented using <code>GetTimeOfDay</code> . On most Win32 platforms it is implemented using <code>ftime</code> . Win32 systems often do not achieve microsecond resolution via this API. If higher resolution is critical to your application test your platform to see the achieved resolution.
	<code>ptime t(microsec_clock::local_time());</code>
<code>ptime microsec_clock::universal_time()</code>	Get the UTC time using a sub second resolution clock. On Unix systems this is implemented using <code>GetTimeOfDay</code> . On most Win32 platforms it is implemented using <code>ftime</code> . Win32 systems often do not achieve microsecond resolution via this API. If higher resolution is critical to your application test your platform to see the achieved resolution.
	<code>ptime t(microsec_clock::universal_time());</code>

Construct using Conversion Functions

Syntax	Description
	Example
<code>ptime from_time_t(time_t t);</code>	Converts a <code>time_t</code> into a <code>ptime</code> .
	<code>ptime t = from_time_t(tt);</code>
<code>ptime from_ftime<ptime>(FILETIME ft);</code>	Creates a <code>ptime</code> object from a <code>FILETIME</code> structure.
	<code>ptime t = from_ftime<ptime>(ft);</code>

Accessors

Syntax	Description
	Example
<code>date date()</code>	<p>Get the date part of a time.</p> <pre>date d(2002,Jan,10); ptime t(d, hour(1)); t.date() --> 2002-Jan-10;</pre>
<code>time_duration time_of_day()</code>	<p>Get the time offset in the day.</p> <pre>date d(2002,Jan,10); ptime t(d, hour(1)); t.time_of_day() --> 01:00:00;</pre>
<code>bool is_infinity() const</code>	<p>Returns true if ptime is either positive or negative infinity</p> <pre>ptime pt(pos_infin); pt.is_infinity(); // --> true</pre>
<code>bool is_neg_infinity() const</code>	<p>Returns true if ptime is negative infinity</p> <pre>ptime pt(neg_infin); pt.is_neg_infinity(); // --> true</pre>
<code>bool is_pos_infinity() const</code>	<p>Returns true if ptime is positive infinity</p> <pre>ptime pt(neg_infin); pt.is_pos_infinity(); // --> true</pre>
<code>bool is_not_a_date_time() const</code>	<p>Returns true if value is not a ptime</p> <pre>ptime pt(not_a_date_time); pt.is_not_a_date_time(); // --> true</pre>
<code>bool is_special() const</code>	<p>Returns true if ptime is any special_value</p> <pre>ptime pt(pos_infin); ptime pt2(not_a_date_time); ptime pt3(date(2005,Mar,1), hours(10)); pt.is_special(); // --> true pt2.is_special(); // --> true pt3.is_special(); // --> false</pre>

Conversion to String

Syntax	Description
	Example
<code>std::string to_simple_string(ptime)</code>	<p>To YYYY-mm-dd HH:MM:SS.ffffffff string where mm is 3 char month name. Fractional seconds only included if non-zero.</p> <pre>2002-Jan-01 10:00:01.123456789</pre>
<code>std::string to_iso_string(ptime)</code>	<p>Convert to form YYYYMMDDTHHMMSS,ffffffff where T is the date-time separator</p> <pre>20020131T100001,123456789</pre>
<code>std::string to_iso_extended_string(ptime)</code>	<p>Convert to form YYYY-MM-DDTHH:MM:SS,ffffffff where T is the date-time separator</p> <pre>2002-01-31T10:00:01,123456789</pre>

Operators

Syntax	Description
<pre>operator<<, operator>></pre>	Example
	<p>Streaming operators. Note: As of version 1.33, streaming operations have been greatly improved. See Date Time IO System for more details (including exceptions and error conditions).</p> <pre>ptime pt(not_a_date_time); stringstream ss("2002-Jan-01 14:23:11"); ss >> pt; std::cout << pt; // "2002-Jan-01 14:23:11" └─</pre>
<pre>operator==, operator!=, operator>, operator<, operator>=, operator<=</pre>	<p>A full complement of comparison operators</p> <pre>t1 == t2, etc</pre>
<pre>ptime operator+(days)</pre>	<p>Return a ptime adding a day offset</p> <pre>date d(2002,Jan,1); ptime t(d,minutes(5)); days dd(1); ptime t2 = t + dd;</pre>
<pre>ptime operator-(days)</pre>	<p>Return a ptime subtracting a day offset</p> <pre>date d(2002,Jan,1); ptime t(d,minutes(5)); days dd(1); ptime t2 = t - dd;</pre>
<pre>ptime operator+(time_duration)</pre>	<p>Return a ptime adding a time duration</p> <pre>date d(2002,Jan,1); ptime t(d,minutes(5)); ptime t2 = t + hours(1) + minutes(2);</pre>
<pre>ptime operator-(time_duration)</pre>	<p>Return a ptime subtracting a time duration</p> <pre>date d(2002,Jan,1); ptime t(d,minutes(5)); ptime t2 = t - minutes(2);</pre>

Syntax	Description
	Example
<code>time_duration operator-(ptime)</code>	Take the difference between two times. <pre>date d(2002,Jan,1); ptime t1(d,minutes(5)); ptime t2(d,seconds(5)); time_duration t3 = t2 - t1;//negative result</pre>

Struct tm, time_t, and FILETIME Functions

Functions for converting posix_time objects to, and from, tm structs are provided as well as conversion from time_t and FILETIME.

Syntax	Description
	Example
<pre>tm to_tm(ptime)</pre>	<p>A function for converting a ptime object to a tm struct. The tm_isdst field is set to -1.</p> <pre>ptime pt(date(2005,Jan,1), time_duration(1,2,3)); tm pt_tm = to_tm(pt); /* tm_year => 105 tm_mon => 0 tm_mday => 1 tm_wday => 6 (Saturday) tm_yday => 0 tm_hour => 1 tm_min => 2 tm_sec => 3 tm_isdst => -1 */</pre>
<pre>date date_from_tm(tm datetm)</pre>	<p>A function for converting a tm struct to a date object. The fields: tm_wday , tm_yday , and tm_isdst are ignored.</p> <pre>tm pt_tm; pt_tm.tm_year = 105; pt_tm.tm_mon = 0; pt_tm.tm_mday = 1; pt_tm.tm_hour = 1; pt_tm.tm_min = 2; pt_tm.tm_sec = 3; ptime pt = ptime_from_tm(pt_tm); // pt => 2005-Jan-01 01:02:03</pre>
<pre>tm to_tm(time_duration)</pre>	<p>A function for converting a time_duration object to a tm struct. The fields: tm_year, tm_mon, tm_mday, tm_wday, tm_yday are set to zero. The tm_isdst field is set to -1.</p> <pre>time_duration td(1,2,3); tm td_tm = to_tm(td); /* tm_year => 0 tm_mon => 0 tm_mday => 0 tm_wday => 0 tm_yday => 0 tm_hour => 1 tm_min => 2 tm_sec => 3 tm_isdst => -1 */</pre>

Syntax	Description
<pre>ptime from_time_t(std::time_t)</pre>	Example
	<p>Creates a ptime from the time_t parameter. The seconds held in the time_t are added to a time point of 1970-Jan-01.</p> <pre>ptime pt(not_a_date_time); std::time_t t; t = 1118158776; pt = from_time_t(t); // pt => 2005-Jun-07 15:39:36</pre>
<pre>ptime from_ftime<ptime>(FILETIME)</pre>	Description
	<p>A template function that constructs a ptime from a FILETIME struct.</p> <pre>FILETIME ft; ft.dwHighDateTime = 29715317; ft.dwLowDateTime = 3865122988UL; ptime pt = from_ftime<ptime>(ft); // pt => 2005-Jun-07 15:30:57.039582000</pre>

Time Duration

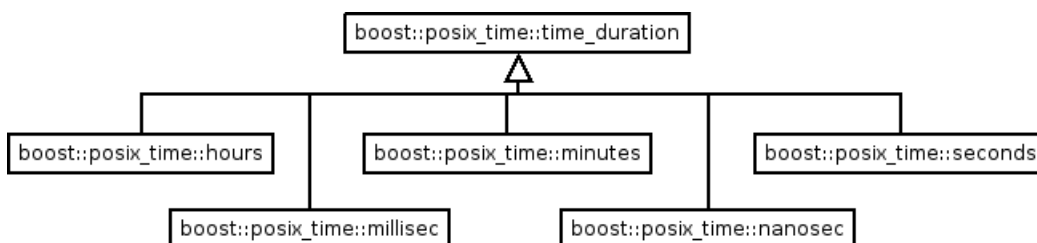
[Introduction](#) -- [Header](#) -- [Construction](#) -- [Count Based Construction](#) -- [Construct from String](#) -- [Accessors](#) -- [Conversion To String](#) -- [Operators](#) -- [Struct tm Functions](#)

Introduction

The class `boost::posix_time::time_duration` is the base type responsible for representing a length of time. A duration can be either positive or negative. The general `time_duration` class provides a constructor that takes a count of the number of hours, minutes, seconds, and fractional seconds count as shown in the code fragment below. The resolution of the `time_duration` is configurable at compile time. See [Build-Compiler Information](#) for more information.

```
using namespace boost::posix_time;
time_duration td(1,2,3,4); //01:02:03.000000004 when resolution is nano seconds
time_duration td(1,2,3,4); //01:02:03.000004 when resolution is micro seconds
```

Several small helper classes that derive from a base `time_duration`, as shown below, to adjust for different resolutions. These classes can shorten code and make the intent clearer.



As an example:

```
using namespace boost::posix_time;

time_duration td = hours(1) + seconds(10); //01:00:01
td = hours(1) + nanoseconds(5); //01:00:00.000000005
```

Note that the existence of the higher resolution classes (eg: nanoseconds) depends on the installation of the library. See [Build-Compiler Information](#) for more information.

Another way to handle this is to utilize the `ticks_per_second()` method of `time_duration` to write code that is portable no matter how the library is compiled. The general equation for calculating a resolution independent count is as follows:

```
count*(time_duration_ticks_per_second / count_ticks_per_second)
└┐
```

For example, let's suppose we want to construct using a count that represents tenths of a second. That is, each tick is 0.1 second.

```
int number_of_tenths = 5;
//create a resolution independent count -- divide by 10 since there are
//10 tenths in a second.
int count = number_of_tenths*(time_duration::ticks_per_second()/10);
time_duration td(1,2,3,count); //01:02:03.5 //no matter the resolution settings
└┐
```

Header

```
#include "boost/date_time/posix_time/posix_time.hpp" //include all types plus i/o
or
#include "boost/date_time/posix_time/posix_time_types.hpp" //no i/o just types
```

Construction

Syntax	Description
<pre>time_duration(hours, minutes, seconds, fractional_seconds)</pre>	<p>Example</p> <p>Construct a duration from the counts. The fractional_second parameter is a number of units and is therefore affected by the resolution the application is compiled with (see Build-Compiler Information). If the fractional_seconds argument exceeds the limit of the compiled precision, the excess value will be "carried over" into the seconds field. See above for techniques to creating a resolution independent count.</p> <pre>time_duration td(1,2,3,9); //1 hr 2 min 3 sec 9 nanoseconds time_duration td2(1,2,3,123456789); time_duration td3(1,2,3,1000); // with microsecond resolution (6 digits) // td2 => "01:04:06.456789" // td3 => "01:02:03.001000" // with nanosecond resolution (9 digits) // td2 => "01:02:03.123456789" // td3 => "01:02:03.000001000"</pre>
<pre>time_duration(special_value sv)</pre>	<p>Special values constructor. Important note: When a time_duration is a special value, either by construction or other means, the following accessor functions will give unpredictable results:</p> <pre>hours(), minutes(), seconds(), ticks(), fractional_seconds(), total_nanoseconds(), total_microseconds(), total_milliseconds(), total_seconds()</pre> <p>The remaining accessor functions will work as expected.</p>

Count Based Construction

Syntax	Description
	Example
<code>hours(long)</code>	Number of hours <pre>time_duration td = hours(3);</pre>
<code>minutes(long)</code>	Number of minutes <pre>time_duration td = minutes(3);</pre>
<code>seconds(long)</code>	Number of seconds <pre>time_duration td = seconds(3);</pre>
<code>milliseconds(long)</code>	Number of milliseconds. <pre>time_duration td = milliseconds(3);</pre>
<code>microseconds(long)</code>	Number of microseconds. <pre>time_duration td = microseconds(3);</pre>
<code>nanoseconds(long)</code>	Number of nanoseconds. <pre>time_duration td = nanoseconds(3);</pre>

Construct from String

Syntax	Description
	Example
<pre>time_duration duration_from_string(std::string)</pre>	From delimited string. NOTE: Excess digits in fractional seconds will be dropped. Ex: "1:02:03.123456999" => 1:02:03.123456. This behavior is affected by the precision the library is compiled with (see Build-Compiler Information). <pre>std::string ts("23:59:59.000"); time_duration td(duration_from_string(ts));</pre>

Accessors

Syntax	Description
	Example
<pre>long hours()</pre>	<p>Get the number of normalized hours (will give unpredictable results if calling time_duration is a special_value).</p> <pre>time_duration td(1,2,3); time_duration neg_td(-1,2,3); td.hours(); // --> 1 neg_td.hours(); // --> -1</pre>
<pre>long minutes()</pre>	<p>Get the number of minutes normalized +/- (0.59) (will give unpredictable results if calling time_duration is a special_value).</p> <pre>time_duration td(1,2,3); time_duration neg_td(-1,2,3); td.minutes(); // --> 2 neg_td.minutes(); // --> -2</pre>
<pre>long seconds()</pre>	<p>Get the normalized number of second +/- (0.59) (will give unpredictable results if calling time_duration is a special_value).</p> <pre>time_duration td(1,2,3); time_duration neg_td(-1,2,3); td.seconds(); // --> 3 neg_td.seconds(); // --> -3</pre>
<pre>long total_seconds()</pre>	<p>Get the total number of seconds truncating any fractional seconds (will give unpredictable results if calling time_duration is a special_value).</p> <pre>time_duration td(1,2,3,10); td.total_seconds(); // --> (1*3600) + (2*60) + 3 == 3723</pre>
<pre>long total_milliseconds()</pre>	<p>Get the total number of milliseconds truncating any remaining digits (will give unpredictable results if calling time_duration is a special_value).</p> <pre>time_duration td(1,2,3,123456789); td.total_milliseconds(); // HMS --> (1*3600) + (2*60) + 3 == 3723 ↴ seconds // milliseconds is 3 decimal places // (3723 * 1000) + 123 == 3723123</pre>

Syntax	Description
	Example
<code>long total_microseconds()</code>	<p>Get the total number of microseconds truncating any remaining digits (will give unpredictable results if calling <code>time_duration</code> is a <code>special_value</code>).</p> <pre>time_duration td(1,2,3,123456789); td.total_microseconds(); // HMS --> (1*3600) + (2*60) + 3 == 3723 ↴ seconds // microseconds is 6 decimal places // (3723 * 1000000) + 123456 == 3723123456</pre>
<code>long total_nanoseconds()</code>	<p>Get the total number of nanoseconds truncating any remaining digits (will give unpredictable results if calling <code>time_duration</code> is a <code>special_value</code>).</p> <pre>time_duration td(1,2,3,123456789); td.total_nanoseconds(); // HMS --> (1*3600) + (2*60) + 3 == 3723 ↴ seconds // nanoseconds is 9 decimal places // (3723 * 1000000000) + 123456789 // == ↴ 3723123456789</pre>
<code>long fractional_seconds()</code>	<p>Get the number of fractional seconds (will give unpredictable results if calling <code>time_duration</code> is a <code>special_value</code>).</p> <pre>time_duration td(1,2,3, 1000); td.fractional_seconds(); // --> 1000</pre>
<code>bool is_negative()</code>	<p>True if duration is negative.</p> <pre>time_duration td(-1,0,0); td.is_negative(); // --> true</pre>
<code>time_duration invert_sign()</code>	<p>Generate a new duration with the sign inverted/</p> <pre>time_duration td(-1,0,0); td.invert_sign(); // --> 01:00:00</pre>
<code>date_time::time_resolutions resolution()</code>	<p>Describes the resolution capability of the <code>time_duration</code> class. <code>time_resolutions</code> is an enum of resolution possibilities ranging from seconds to nanoseconds.</p> <pre>time_duration::resolution() --> nano</pre>

Syntax	Description
	Example
<code>time_duration::num_fractional_digits()</code>	<p>Returns an unsigned short holding the number of fractional digits the time resolution has.</p> <pre> unsigned short secs; secs = time_duration::num_fractional_digits(); // 9 for nano, 6 for micro, etc. </pre>
<code>time_duration::ticks_per_second()</code>	<p>Return the number of ticks in a second. For example, if the duration supports nanoseconds then the returned result will be 1,000,000,000 (1e+9).</p> <pre> std::cout << time_duration::ticks_per_second(); </pre>
<code>boost::int64_t ticks()</code>	<p>Return the raw count of the duration type (will give unpredictable results if calling time_duration is a special_value).</p> <pre> time_duration td(0,0,0, 1000); td.ticks() // --> 1000 </pre>
<code>time_duration unit()</code>	<p>Return smallest possible unit of duration type (1 nanosecond).</p> <pre> time_duration::unit() --> time_duration(0,0,0,1) </pre>
<code>bool is_neg_infinity() const</code>	<p>Returns true if time_duration is negative infinity</p> <pre> time_duration td(neg_infin); td.is_neg_infinity(); // --> true </pre>
<code>bool is_pos_infinity() const</code>	<p>Returns true if time_duration is positive infinity</p> <pre> time_duration td(neg_infin); td.is_pos_infinity(); // --> true </pre>
<code>bool is_not_a_date_time() const</code>	<p>Returns true if value is not a time</p> <pre> time_duration td(not_a_date_time); td.is_not_a_date_time(); // --> true </pre>

Syntax	Description
	Example
<code>bool is_special() const</code>	Returns true if time_duration is any special_value
	<pre>time_duration td(pos_infin); time_duration td2(not_a_date_time); time_duration td3(2,5,10); td.is_special(); // --> true td2.is_special(); // --> true td3.is_special(); // --> false</pre>

Conversion To String

Syntax	Description
	Example
<code>std::string to_simple_string(time_duration)</code>	To HH:MM:SS.fffffffff were fff is fractional seconds that are only included if non-zero.
	10:00:01.123456789
<code>std::string to_iso_string(time_duration)</code>	Convert to form HHMMSS,fffffffff.
	100001,123456789

Operators

Syntax	Description
	Example
<code>operator<<, operator>></code>	Streaming operators. Note: As of version 1.33, streaming operations have been greatly improved. See Date Time IO System for more details (including exceptions and error conditions). <pre>time_duration td(0,0,0); stringstream ss("14:23:11.345678"); ss >> td; std::cout << td; // "14:23:11.345678" └─┘</pre>
<code>operator==, operator!=, operator>, operator<, operator>=, operator<=</code>	A full complement of comparison operators <pre>ddl == dd2, etc</pre>
<code>time_duration operator+(time_duration)</code>	Add durations. <pre>time_duration td1(hours(1)+minutes(2)); time_duration td2(seconds(10)); time_duration td3 = td1 + td2;</pre>
<code>time_duration operator-(time_duration)</code>	Subtract durations. <pre>time_duration td1(hours(1)+nanoseconds(2)); time_duration td2 = td1 - minutes(1);</pre>
<code>time_duration operator/(int)</code>	Divide the length of a duration by an integer value. Discards any remainder. <pre>hours(3)/2 == time_duration(1,30,0); nanosecond(3)/2 == nanosecond(1);</pre>
<code>time_duration operator*(int)</code>	Multiply the length of a duration by an integer value. <pre>hours(3)*2 == hours(6);</pre>

Struct tm, time_t, and FILETIME Functions

Function for converting a `time_duration` to a `tm` struct is provided.

Syntax	Description
<pre>tm to_tm(time_duration)</pre>	Example
	<p>A function for converting a <code>time_duration</code> object to a <code>tm</code> struct. The fields: <code>tm_year</code>, <code>tm_mon</code>, <code>tm_mday</code>, <code>tm_wday</code>, <code>tm_yday</code> are set to zero. The <code>tm_isdst</code> field is set to -1.</p> <pre>time_duration td(1,2,3); tm td_tm = to_tm(td); /* tm_year => 0 tm_mon => 0 tm_mday => 0 tm_wday => 0 tm_yday => 0 tm_hour => 1 tm_min => 2 tm_sec => 3 tm_isdst => -1 */</pre>

Time Period

[Introduction](#) -- [Header](#) -- [Construction](#) -- [Mutators](#) -- [Accessors](#) -- [Conversion To String](#) -- [Operators](#)

Introduction

The class `boost::posix_time::time_period` provides direct representation for ranges between two times. Periods provide the ability to simplify some types of calculations by simplifying the conditional logic of the program.

A period that is created with beginning and end points being equal, or with a duration of zero, is known as a zero length period. Zero length periods are considered invalid (it is perfectly legal to construct an invalid period). For these periods, the `last` point will always be one unit less than the `begin` point.

The [time periods example](#) provides an example of using time periods.

Header

```
#include "boost/date_time/posix_time/posix_time.hpp" //include all types plus i/o
or
#include "boost/date_time/posix_time/posix_time_types.hpp" //no i/o just types
```

Construction

Syntax	Description
	Example
<pre>time_period(ptime, ptime)</pre>	<p>Create a period as [begin, end). If end is \leq begin then the period will be defined as invalid.</p> <pre>date d(2002,Jan,01); ptime t1(d, seconds(10)); //10 sec after mid- night ptime t2(d, hours(10)); //10 hours after mid- night time_period tp(t1, t2);</pre>
<pre>time_period(ptime, time_duration)</pre>	<p>Create a period as [begin, begin+len) where end would be begin+len. If len is \leq zero then the period will be defined as invalid.</p> <pre>date d(2002,Jan,01); ptime t(d, seconds(10)); //10 sec after mid- night time_period tp(t, hours(3));</pre>
<pre>time_period(time_period rhs)</pre>	<p>Copy constructor</p> <pre>time_period tp1(tp);</pre>

Mutators

Syntax	Description
	Example
<pre>time_period shift(time_duration)</pre>	<p>Add duration to both begin and end.</p> <pre>time_period ↵ tp(ptime(date(2005,Jan,1),hours(1)), ↵ hours(2)); tp.shift(minutes(5)); // tp == 2005-Jan-01 01:05:00 to 2005-Jan- 01 03:05:00 ↵</pre>
<pre>time_period expand(days)</pre>	<p>Subtract duration from begin and add duration to end.</p> <pre>time_period ↵ tp(ptime(date(2005,Jan,1),hours(1)), ↵ hours(2)); tp.expand(minutes(5)); // tp == 2005-Jan-01 00:55:00 to 2005-Jan- 01 03:05:00 ↵</pre>

Accessors

Syntax	Description
	Example
<pre>ptime begin()</pre>	<p>Return first time of period.</p> <pre> date d(2002,Jan,01); ptime t1(d, seconds(10)); //10 sec after mid- night ptime t2(d, hours(10)); //10 hours after mid- night time_period tp(t1, t2); tp.begin(); // --> 2002-Jan-01 00:00:10 </pre>
<pre>ptime last()</pre>	<p>Return last time in the period</p> <pre> date d(2002,Jan,01); ptime t1(d, seconds(10)); //10 sec after mid- night ptime t2(d, hours(10)); //10 hours after mid- night time_period tp(t1, t2); tp.last();// --> 2002-Jan-01 0 9:59:59.999999999 </pre>
<pre>ptime end()</pre>	<p>Return one past the last in period</p> <pre> date d(2002,Jan,01); ptime t1(d, seconds(10)); //10 sec after mid- night ptime t2(d, hours(10)); //10 hours after mid- night time_period tp(t1, t2); tp.last(); // --> 2002-Jan-01 10:00:00 </pre>
<pre>time_duration length()</pre>	<p>Return the length of the time period.</p> <pre> date d(2002,Jan,01); ptime t1(d); //midnight time_period tp(t1, hours(1)); tp.length() --> 1 hour </pre>
<pre>bool is_null()</pre>	<p>True if period is not well formed. eg: end is less than or equal to begin.</p> <pre> date d(2002,Jan,01); ptime t1(d, hours(12)); // noon on Jan 1st ptime t2(d, hours(9)); // 9am on Jan 1st time_period tp(t1, t2); tp.is_null(); // true </pre>

Syntax	Description
	Example
<pre>bool contains(ptime)</pre>	<p>True if ptime is within the period. Zero length periods cannot contain any points.</p> <pre> date d(2002,Jan,01); ptime t1(d, seconds(10)); //10 sec after mid- night ptime t2(d, hours(10)); //10 hours after mid- night ptime t3(d, hours(2)); //2 hours after mid- night time_period tp(t1, t2); tp.contains(t3); // true time_period tp2(t1, t1); tp2.contains(t1); // false </pre>
<pre>bool contains(time_period)</pre>	<p>True if period is within the period</p> <pre> time_period tp1(ptime(d,hours(1)), ptime(d,hours(12))); time_period tp2(ptime(d,hours(2)), ptime(d,hours(4))); tp1.contains(tp2); // --> true tp2.contains(tp1); // --> false </pre>
<pre>bool intersects(time_period)</pre>	<p>True if periods overlap</p> <pre> time_period tp1(ptime(d,hours(1)), ptime(d,hours(12))); time_period tp2(ptime(d,hours(2)), ptime(d,hours(4))); tp2.intersects(tp1); // --> true </pre>
<pre>time_period intersection(time_period)</pre>	<p>Calculate the intersection of 2 periods. Null if no intersection.</p>
<pre>time_period merge(time_period)</pre>	<p>Returns union of two periods. Null if no intersection.</p>
<pre>time_period span(time_period)</pre>	<p>Combines two periods and any gap between them such that begin = min(p1.begin, p2.begin) and end = max(p1.end , p2.end).</p>

Conversion To String

Syntax	Description
	Example
<pre>std::string to_simple_string(time_period dp)</pre>	<p>To [YYYY-mm-DD hh:mm:ss.fffffffff/YYYY-mm-DD hh:mm:ss.fffffffff] string where mm is 3 char month name.</p> <pre>[2002-Jan-01 01:25:10.000000001/ 2002-Jan-31 ↵ 01:25:10.123456789] // string occupies one line</pre>

Operators

Syntax	Description
	Example
<code>operator<<</code>	<p>Output streaming operator for time duration. Uses facet to output [date time_of_day/date time_of_day]. The default is format is [YYYY-mm-DD hh:mm:ss.ffffffffff/YYYY-mm-DD hh:mm:ss.ffffffffff] string where mmm is 3 char month name and the fractional seconds are left out when zero.</p> <pre>[2002-Jan-01 01:25:10.000000001/ \ 2002-Jan-31 01:25:10.123456789]</pre>
<code>operator>></code>	<p>Input streaming operator for time duration. Uses facet to read [date time_of_day/date time_of_day]. The default is format is [YYYY-mm-DD hh:mm:ss.ffffffffff/YYYY-mm-DD hh:mm:ss.ffffffffff] string where mmm is 3 char month name and the fractional seconds are left out when zero.</p> <pre>[2002-Jan-01 01:25:10.000000001/ \ 2002-Jan-31 01:25:10.123456789]</pre>
<code>operator==, operator!=</code>	<p>Equality operators. Periods are equal if p1.begin == p2.begin && p1.last == p2.last</p> <pre>if (tp1 == tp2) {...</pre>
<code>operator<</code>	<p>Ordering with no overlap. True if tp1.end() less than tp2.begin()</p> <pre>if (tp1 < tp2) {...</pre>
<code>operator></code>	<p>Ordering with no overlap. True if tp1.begin() greater than tp2.end()</p> <pre>if (tp1 > tp2) {... etc</pre>
<code>operator<=, operator>=</code>	<p>Defined in terms of the other operators.</p>

Time Iterators

[Introduction](#) -- [Header](#) -- [Overview](#) -- [Operators](#)

Introduction

Time iterators provide a mechanism for iteration through times. Time iterators are similar to [Bidirectional Iterators](#). However, `time_iterators` are different than standard iterators in that there is no underlying sequence, just a calculation function. In addition, `time_iterators` are directly comparable against instances of [class `ptime`](#). Thus a second iterator for the end point of the iteration is not required, but rather a point in time can be used directly. For example, the following code iterates using a 15 minute iteration interval. The [print hours](#) example also illustrates the use of the `time_iterator`.

```
#include "boost/date_time/posix_time/posix_time.hpp"
#include <iostream>

int
main()
{
    using namespace boost::gregorian;
    using namespace boost::posix_time;
    date d(2000,Jan,20);
    ptime start(d);
    ptime end = start + hours(1);
    time_iterator titr(start,minutes(15)); //increment by 15 minutes
    //produces 00:00:00, 00:15:00, 00:30:00, 00:45:00
    while (titr < end) {
        std::cout << to_simple_string(*titr) << std::endl;
        ++titr;
    }
    std::cout << "Now backward" << std::endl;
    //produces 01:00:00, 00:45:00, 00:30:00, 00:15:00
    while (titr > start) {
        std::cout << to_simple_string(*titr) << std::endl;
        --titr;
    }
}
```

Header

```
#include "boost/date_time/posix_time/posix_time.hpp" //include all types plus i/o
or
#include "boost/date_time/posix_time/posix_time_types.hpp" //no i/o just types
```

Overview

Class	Description
	Construction Parameters
time_iterator	Iterate incrementing by the specified duration.
	ptime start_time, time_duration increment

Operators

Syntax	Description
<pre>operator==(const ptime& rhs), operator!=(const ptime& rhs), operator>, operator<, operator>=, operator<=</pre>	<p data-bbox="810 342 916 371">Example</p> <p data-bbox="810 405 1283 434">A full complement of comparison operators</p> <pre data-bbox="826 488 1481 748">date d(2002,Jan,1); ptime start_time(d, hours(1)); //increment by 10 minutes time_iterator titr(start_time, minutes(10)); ptime end_time = start_time + hours(2); if (titr == end_time) // false if (titr != end_time) // true if (titr >= end_time) // false if (titr <= end_time) // true</pre>
<pre>prefix increment</pre>	<p data-bbox="810 817 1321 846">Increment the iterator by the specified duration.</p> <pre data-bbox="826 900 1481 1016">//increment by 10 milli seconds time_iterator titr(start_time, milli- seconds(10)); ++titr; // == start_time + 10 milliseconds</pre>
<pre>prefix decrement</pre>	<p data-bbox="810 1086 1385 1115">Decrement the iterator by the specified time duration.</p> <pre data-bbox="826 1169 1481 1249">time_duration td(1,2,3); time_iterator titr(start_time, td); --titr; // == start_time - 01:02:03</pre>

Local Time

Local Time System

[Introduction](#) -- [Usage Examples](#)

Introduction

The library supports 4 main extensions for the management of local times. This includes

`local_date_time` -- locally adjusted time point
`posix_time_zone` -- time zone defined by posix string (eg: "EST10EDT,M10.5.0,M3.5.0/03")
`time_zone_database` -- get time zones by region from .csv file (eg: America/New York)
`time_zone` -- abstract time zone interface

Together, these extensions define a time system adjusted for recording times related to a specific earth location. This time system utilizes all the features and benefits of the `posix_time` system (see [posix_time](#) for full details). It uses a `time_zone` object which contains all the necessary data/rules to enable adjustments to and from various time zones. The `time_zone` objects used in `date_time` are handled via a `boost::shared_ptr<boost::local_time::time_zone>`.

The phrase "wall-clock" refers to the time that would be shown on a wall clock in a particular time zone at any point in time. `Local_time` uses a time zone object to account for differences in time zones and daylight savings adjustments. For example: While 5:00 pm, October 10, 2004 in Sydney Australia occurs at exactly the same instant as 3:00 am, October 10, 2004 in New York USA, it is a 14 hour difference in wall-clock times. However, a point in time just one day later will result in a 16 hour difference in wall-clock time due to daylight savings adjustments in both time zones. The `local_time` system tracks these by means of a time point, stored as UTC, and `time_zone` objects that contain all the necessary data to correctly calculate wall-clock times.

Usage Examples

Example	Description
Simple Time Zone	Side by side examples of Time Zone usage. Both <code>custom_time_zone</code> and <code>posix_time_zone</code> are shown.
Daylight Savings Calc Rules	Simple example showing the creation of all five <code>dst_calc_rule</code> types.
Seconds Since Epoch	Example that calculates the total seconds elapsed since the epoch (1970-Jan-01) utilizing <code>local_date_time</code> .

Time Zone (abstract)

[Introduction](#) -- [Header](#) -- [Construction](#) -- [Accessors](#)

Introduction

The `time_zone_base` class is an abstract base class template for representing time zones. Time zones are a set of data and rules that provide information about a time zone. The `date_time` library handles `time_zones` by means of a `boost::shared_ptr<time_zone_base>`. A user's custom time zone class will work in the `date_time` library by means of this `shared_ptr`.

For convenience, the `time_zone_base` class is typedef'd as `time_zone`. All references in the documentation to `time_zone`, are referring to this typedef.

Header

The `time_zone_base` class is defined in the header:


```
#include "boost/date_time/time_zone_base.hpp"
↵
```

Construction

A default constructor is provided in the `time_zone_base` class. There are no private data members in this base class to initialize.

Template parameters are `time_type` (typically `posix_time::ptime`) and `CharT` (defaults to `char`).

Accessors

All of the accessors listed here are pure virtual functions.

Syntax	Description
<pre>string_type dst_zone_abbrev();</pre>	Returns the daylight savings abbreviation for the represented time zone.
<pre>string_type std_zone_abbrev();</pre>	Returns the standard abbreviation for the represented time zone.
<pre>string_type dst_zone_name();</pre>	Returns the daylight savings name for the represented time zone.
<pre>string_type std_zone_name();</pre>	Returns the standard name for the represented time zone.
<pre>bool has_dst();</pre>	Returns true if this time zone does not make a daylight savings shift.
<pre>time_type dst_local_start_time(year_type);</pre>	The date and time daylight savings time begins in given year.
<pre>time_type dst_local_end_time(year_type);</pre>	The date and time daylight savings time ends in given year.
<pre>time_duration_type base_utc_offset();</pre>	The amount of time offset from UTC (typically in hours).
<pre>time_duration_type dst_offset();</pre>	The amount of time shifted during daylight savings.
<pre>std::string to_posix_string();</pre>	Returns a posix time zone string representation of this time_zone_base object. For a detailed description of a posix time zone string see posix_time_zone .

Posix Time Zone

[Introduction](#) -- [Important Notes](#) -- [Header](#) -- [Construction](#) -- [Accessors](#)

Introduction

A `posix_time_zone` object is a set of data and rules that provide information about a time zone. Information such as the offset from UTC, it's name and abbreviation, as well as daylight savings rules, called [dst_calc_rules](#). These rules are stored as a `boost::shared_ptr<dst_calc_rules>`.

As a convenience, a typedef for `shared_ptr<dst_calc_rules>` is provided.

```
typedef boost::shared_ptr<dst_calc_rules> local_time::dst_calc_rule_ptr;
```

A `posix_time_zone` is unique in that the object is created from a Posix time zone string (IEEE Std 1003.1). A POSIX time zone string takes the form of:

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

'std' specifies the abbrev of the time zone. 'offset' is the offset from UTC. 'dst' specifies the abbrev of the time zone during daylight savings time. The second offset is how many hours changed during 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}

Exceptions will be thrown under the following conditions:

- An exception will be thrown for an invalid date spec (see [date class](#)).
- A `boost::local_time::bad_offset` exception will be thrown for:
- A DST start or end offset that is negative or more than 24 hours.
- A UTC zone that is greater than +14 or less than -12 hours.
- A `boost::local_time::bad_adjustment` exception will be thrown for a DST adjustment that is 24 hours or more (positive or negative)

As stated above, the 'offset' and '/time' portions of the string are not required. If they are not given they default to 01:00 for 'offset', and 02:00 for both occurrences of '/time'.

Some examples are:

"PST-8PDT01:00:00,M4.1.0/02:00:00,M10.1.0/02:00:00"

"PST-8PDT,M4.1.0,M10.1.0"

These two are actually the same specification (defaults were used in the second string). This zone lies eight hours west of GMT and makes a one hour shift forward during daylight savings time. Daylight savings for this zone starts on the first Sunday of April at 2am, and ends on the first Sunday of October at 2am.

"MST-7"

This zone is as simple as it gets. This zone lies seven hours west of GMT and has no daylight savings.

"EST10EDT,M10.5.0,M3.5.0/03"

This string describes the time zone for Sydney Australia. It lies ten hours east of GMT and makes a one hour shift forward during daylight savings. Being located in the southern hemisphere, daylight savings begins on the last Sunday in October at 2am and ends on the last Sunday in March at 3am.

"FST+3FDT02:00,J60/00,J304/02"

This specification describes a fictitious zone that lies three hours east of GMT. It makes a two hour shift forward for daylight savings which begins on March 1st at midnight, and ends on October 31st at 2am. The 'J' designation in the start/end specs signifies that counting starts at one and February 29th is never counted.

"FST+3FDT,59,304"

This specification is significant because of the '59'. The lack of 'J' for the start and end dates, indicates that the Julian day-count begins at zero and ends at 365. If you do the math, you'll see that allows for a total of 366 days. This is fine in leap years, but in non-leap years '59' (Feb-29) does not exist. This will construct a valid `posix_time_zone` object but an exception will be thrown if the date of '59' is accessed in a non-leap year. Ex:

```
posix_time_zone leap_day(std::string("FST+3FDT,59,304"));
leap_day.dst_local_start_time(2004); // ok
leap_day.dst_local_start_time(2003); // Exception thrown
```

The `posix_time_zone` objects are used via a `boost::shared_ptr<local_time::time_zone_base>`. As a convenience, a typedef for `boost::shared_ptr<local_time::time_zone_base>` is provided:

```
typedef boost::shared_ptr<time_zone_base> local_time::time_zone_ptr;
```

See [Simple time zone](#) for a side by side example of `time_zone` and `posix_time_zone` usage.

Important Notes

- `posix_time_zone` objects use the standard and daylight savings abbreviations in place of the full names (see [Accessors](#) below).
- 'Jn' and 'n' date specifications can not be mixed in a specification string. Ex: "FST+3FDT,59,J304"
- 'n' date specification of 59 represents Feb-29. Do not attempt to access in a non-leap year or an exception will be thrown.

Header

The inclusion of a single header will bring in all `boost::local_time` types, functions, and IO operators.

```
#include "boost/date_time/local_time/local_time.hpp"
```

Construction

Syntax	Example
<code>posix_time_zone(std::string)</code>	<pre>std::string nyc("EST-5EDT,M4.1.0,M10.5.0"); time_zone_ptr zone(new posix_time_zone(nyc));</pre>

Accessors

Syntax	Description
	Example
<code>std::string dst_zone_abbrev()</code>	Returns the daylight savings abbreviation for the represented time zone. <pre>nyc_zone_sh_ptr->dst_zone_abbrev(); // "EDT"</pre>
<code>std::string std_zone_abbrev()</code>	Returns the standard abbreviation for the represented time zone. <pre>nyc_zone_sh_ptr->std_zone_abbrev(); // "EST"</pre>
<code>std::string dst_zone_name()</code>	Returns the daylight savings ABBREVIATION for the represented time zone. <pre>nyc_zone_sh_ptr->dst_zone_name(); // "EDT"</pre>
<code>std::string std_zone_name()</code>	Returns the standard ABBREVIATION for the represented time zone. <pre>nyc_zone_sh_ptr->std_zone_name(); // "EST"</pre>
<code>bool has_dst()</code>	Returns true when time_zone's shared_ptr to dst_calc_rules is not NULL. <pre>nyc_zone_sh_ptr->has_dst(); // true phx_zone_sh_ptr->has_dst(); // false</pre>
<code>ptime dst_local_start_time(greg_year)</code>	The date and time daylight savings time begins in given year. Returns not_a_date_time if this zone has no daylight savings. <pre>nyc_zone_sh_ptr->dst_local_start_time(2004); // 2004-Apr-04 02:00</pre>
<code>ptime dst_local_end_time(greg_year)</code>	The date and time daylight savings time ends in given year. Returns not_a_date_time if this zone has no daylight savings. <pre>nyc_zone_sh_ptr->dst_local_end_time(2004); // 2004-Oct-31 02:00</pre>
<code>time_duration base_utc_offset()</code>	The amount of time offset from UTC (typically in hours). <pre>nyc_zone_sh_ptr->base_utc_offset(); // -05:00</pre>

Syntax	Description
<pre>posix_time::time_duration dst_offset()</pre>	Example
	<p>The amount of time shifted during daylight savings.</p> <pre>nyc_zone_sh_ptr->dst_offset(); // 01:00</pre>
<pre>std::string to_posix_string()</pre>	<p>Returns a posix time zone string representation of this <code>time_zone_base</code> object. Depending on how the <code>time_zone</code> object was created, the date-spec format of the string will be in either 'M' notation or 'n' notation. Every possible date-spec that can be represented in 'J' notation can also be represented in 'n' notation. The reverse is not true so only 'n' notation is used for these types of date-specs. For a detailed description of a posix time zone string see posix_time_zone.</p>
	<pre>nyc_zone_sh_ptr->to_posix_string(); // "EST-05EDT+01,M4.1.0/02:00,M10.5.0/02:00" phx_zone_sh_ptr->to_posix_string(); // "MST-07"</pre>

Time Zone Database

[Introduction](#) -- [Header](#) -- [Construction](#) -- [Accessors](#) -- [Data File Details](#)

Introduction

The `local_time` system depends on the ability to store time zone information. Our Time Zone Database (`tz_database`) is a means of permanently storing that data. The specifications for many time zones (377 at this time) are provided. These specifications are based on data found in the [zoneinfo database](#). The specifications are stored in the file:

```
libs/date_time/data/date_time_zonespec.csv
```

. While this file already contains specifications for many time zones, it's real intent is for the user to modify it by adding (or removing) time zones to fit their application. See [Data File Details](#) to learn how this is accomplished.

Header

The inclusion of a single header will bring in all `boost::local_time` types, functions, and IO operators.

```
#include "boost/date_time/local_time/local_time.hpp"
└─
```

Construction

The only constructor takes no arguments and creates an empty database. It is up to the user to populate the database. This is typically achieved by loading the desired datafile, but can also be accomplished by means of the `add_record(...)` function (see the [Accessors table](#)). A `local_time::data_not_accessible` exception will be thrown if given zonespec file cannot be found. `local_time::bad_field_count` exception will be thrown if the number of fields in given zonespec file is incorrect.

Syntax	Description
<code>tz_database()</code>	<p>Constructor creates an empty database.</p> <pre>tz_database tz_db;</pre>
<code>bool load_from_file(std::string)</code>	<p>Parameter is path to a time zone spec csv file (see Data File Details for details on the contents of this file). This function populates the database with time zone records found in the zone spec file. A <code>local_time::data_not_accessible</code> exception will be thrown if given zonespec file cannot be found. <code>local_time::bad_field_count</code> exception will be thrown if the number of fields in given zonespec file is incorrect.</p> <pre>tz_database tz_db; tz_db.load_from_file("./date_time_zonespec.csv");</pre>

Accessors

Syntax	Description
	Example
<code>bool tz_db.add_record(std::string id, time_zone_ptr tz);</code>	<p>Adds a <code>time_zone</code>, or a <code>posix_time_zone</code>, to the database. ID is the region name for this zone (Ex: "America/Phoenix").</p> <pre>time_zone_ptr zone(new posix_time_zone("PST-8PDT,M4.1.0,M10.1.0")); std::string id("America/West_Coast"); tz_db.add_record(id, zone);</pre>
<code>time_zone_ptr tz_db.time_zone_from_region(string id);</code>	<p>Returns a <code>time_zone</code>, via a <code>time_zone_ptr</code>, that matches the region listed in the data file. A null pointer is returned if no match is found.</p> <pre>time_zone_ptr nyc = tz_db.time_zone_from_region("America/New_York");</pre>
<code>vector<string> tz_db.region_list();</code>	<p>Returns a vector of strings that holds all the region ID strings from the database.</p> <pre>std::vector<std::string> regions; regions = tz_db.region_list();</pre>

Data File Details

Field Description/Details

The csv file containing the zone_specs used by the boost::local_time::tz_database is intended to be customized by the library user. When customizing this file (or creating your own) the file must follow a specific format.

This first line is expected to contain column headings and is therefore not processed by the tz_database.

Each record (line) must have eleven fields. Some of those fields can be empty. Every field (even empty ones) must be enclosed in double-quotes.

Ex:

```
"America/Phoenix" <- string enclosed in quotes  
"" <- empty field
```

Some fields represent a length of time. The format of these fields must be:

```
"{+|-}hh:mm[:ss]" <- length-of-time format
```

Where the plus or minus is mandatory and the seconds are optional.

Since some time zones do not use daylight savings it is not always necessary for every field in a zone_spec to contain a value. All zone_specs must have at least ID and GMT offset. Zones that use daylight savings must have all fields filled except: STD ABBR, STD NAME, DST NAME. You should take note that DST ABBR is mandatory for zones that use daylight savings (see field descriptions for further details).

Field Description/Details

- ID
Contains the identifying string for the zone_spec. Any string will do as long as it's unique. No two ID's can be the same.
 - STD ABBR
 - STD NAME
 - DST ABBR
 - DST NAME
These four are all the names and abbreviations used by the time zone being described. While any string will do in these fields, care should be taken. These fields hold the strings that will be used in the output of many of the local_time classes.
 - GMT offset
This is the number of hours added to utc to get the local time before any daylight savings adjustments are made. Some examples are: America/New_York offset -5 hours, and Africa/Cairo offset +2 hours. The format must follow the length-of-time format described above.
 - DST adjustment
The amount of time added to gmt_offset when daylight savings is in effect. The format must follow the length-of-time format described above.
- NOTE: more rule capabilities are needed - this portion of the tz_database is incomplete
- DST Start Date rule
This is a specially formatted string that describes the day of year in which the transition take place. It holds three fields of it's own, separated by semicolons.

1. The first field indicates the "nth" weekday of the month. The possible values are: 1 (first), 2 (second), 3 (third), 4 (fourth), 5 (fifth), and -1 (last).

2. The second field indicates the day-of-week from 0-6 (Sun=0).

3. The third field indicates the month from 1-12 (Jan=1).

Examples are: "-1;5;9"="Last Friday of September", "2;1;3"="Second Monday of March"

- Start time

Start time is the number of hours past midnight, on the day of the start transition, the transition takes place. More simply put, the time of day the transition is made (in 24 hours format). The format must follow the length-of-time format described above with the exception that it must always be positive.

- DST End date rule

See DST Start date rule. The difference here is this is the day daylight savings ends (transition to STD).

- End time

Same as Start time.

Custom Time Zone

[Introduction](#) -- [Header](#) -- [Construction](#) -- [Accessors](#) -- [Dependent Types](#)

Introduction

A `custom_time_zone` object is a set of data and rules that provide information about a time zone. Information such as the offset from UTC, it's name and abbreviation, as well as daylight savings rules, called [dst_calc_rules](#). These rules are handled via a `boost::shared_ptr<dst_calc_rules>`. Not all time zones utilize daylight savings, therefore, `time_zone` objects can be used with a NULL-assigned `shared_ptr`.

As a convenience, a typedef for `shared_ptr<dst_calc_rules>` is provided.

```
typedef boost::shared_ptr<dst_calc_rules> local_time::dst_calc_rule_ptr;
```

The `time_zone` objects are used via a `boost::shared_ptr<local_time::time_zone>`. As a convenience, a typedef for `boost::shared_ptr<local_time::time_zone>` is provided:

```
typedef boost::shared_ptr<time_zone> local_time::time_zone_ptr;
```

Header

The inclusion of a single header will bring in all `boost::local_time` types, functions, and IO operators.

```
#include "boost/date_time/local_time/local_time.hpp"
```

Construction

Construction of a `custom_time_zone` is dependent on four objects: a [time_duration](#), a [time_zone_names](#), a [dst_adjustment_offsets](#), and a `shared_ptr` to a [dst_calc_rule](#).

Syntax	Example
<pre>custom_time_zone(...) Parameters: names, gmt_offset, dst_offsets, dst_rules ↵</pre>	See simple_time_zone example for time_zone usage

Accessors

Syntax	Description
<pre>std::string dst_zone_abbrev()</pre>	Example
	<p>Returns the daylight savings abbreviation for the represented time zone.</p> <pre>nyc_zone_sh_ptr->dst_zone_abbrev(); // "EDT"</pre>
<pre>std::string std_zone_abbrev()</pre>	Returns the standard abbreviation for the represented time zone.
	<pre>nyc_zone_sh_ptr->std_zone_abbrev(); // "EST"</pre>
<pre>std::string dst_zone_name()</pre>	Returns the daylight savings name for the represented time zone.
	<pre>nyc_zone_sh_ptr->dst_zone_name(); // "Eastern Daylight Time"</pre>
<pre>std::string std_zone_name()</pre>	Returns the standard name for the represented time zone.
	<pre>nyc_zone_sh_ptr->std_zone_name(); // "Eastern Standard Time"</pre>
<pre>bool has_dst()</pre>	Returns true when custom_time_zone's shared_ptr to dst_calc_rules is not NULL.
	<pre>nyc_zone_sh_ptr->has_dst(); // true phx_zone_sh_ptr->has_dst(); // false</pre>
<pre>dst_local_start_time(...) Return Type: ptime Parameter: greg_year</pre>	The date and time daylight savings time begins in given year. Returns not_a_date_time if this zone has no daylight savings.
	<pre>nyc_ptr->dst_local_start_time(2004); // 2004-Apr-04 02:00</pre>
<pre>dst_local_end_time(...) Return Type: ptime Parameter: greg_year</pre>	The date and time daylight savings time ends in given year. Returns not_a_date_time if this zone has no daylight savings.
	<pre>nyc_ptr->dst_local_end_time(2004); // 2004-Oct-31 02:00</pre>

Syntax	Description
	Example
<code>time_duration base_utc_offset()</code>	<p>The amount of time offset from UTC (typically in hours).</p> <pre>nyc_ptr->base_utc_offset(); // -05:00</pre>
<code>time_duration dst_offset()</code>	<p>The amount of time shifted during daylight savings.</p> <pre>nyc_zone_sh_ptr->dst_offset(); // 01:00</pre>
<code>std::string to_posix_string()</code>	<p>Returns a posix time zone string representation of this <code>time_zone</code> object. Depending on how the <code>time_zone</code> object was created, the date-spec format of the string will be in either 'M' notation or 'n' notation. Every possible date-spec that can be represented in 'J' notation can also be represented in 'n' notation. The reverse is not true so only 'n' notation is used for these types of date-specs. For a detailed description of a posix time zone string see posix_time_zone.</p> <pre>nyc_ptr->to_posix_string(); // "EST-05EDT+01,M4.1.0/02:00,M10.5.0/02:00" phx_ptr->to_posix_string(); // "MST-07" └─┘</pre>

Dependent Types

[Time Zone Names](#) -- [Dst Adjustment Offsets](#) -- [Daylight Savings Calc Rules](#)

Time Zone Names

The `time_zone_names_base` type is an immutable template class of four strings. One each for the name and abbreviation in standard time and daylight savings time. The `time_zone_names` type is a typedef of `time_zone_names_base<char>`.

Syntax	Description
<pre>time_zone_names(...) Parameters: string std_name string std_abbrev string dst_name string dst_abbrev</pre>	<p data-bbox="810 271 1487 304">Example</p> <p data-bbox="810 333 1487 367">The only constructor, all four strings must be provided.</p> <pre data-bbox="815 405 1482 600">string sn("Eastern Standard Time"); string sa("EST"); string dn("Eastern Daylight Time"); string da("EDT"); time_zone_names nyc_names(sn, sa, dn, da);</pre>
<pre>std::string std_zone_name()</pre>	<p data-bbox="810 656 1487 689">Returns the standard zone name</p> <pre data-bbox="815 728 1482 808">nyc_names.std_zone_name(); // "Eastern Standard Time"</pre>
<pre>std::string std_zone_abbrev()</pre>	<p data-bbox="810 857 1487 891">Returns the standard zone abbreviation</p> <pre data-bbox="815 929 1482 1010">nyc_names.std_zone_abbrev(); // "EST"</pre>
<pre>std::string dst_zone_name()</pre>	<p data-bbox="810 1059 1487 1093">Returns the daylight savings zone name</p> <pre data-bbox="815 1131 1482 1211">nyc_names.std_zone_name(); // "Eastern Daylight Time"</pre>
<pre>std::string dst_zone_abbrev()</pre>	<p data-bbox="810 1261 1487 1294">Returns the daylight savings zone abbreviation</p> <pre data-bbox="815 1332 1482 1413">nyc_names.std_zone_abbrev(); // "EDT"</pre>

Dst Adjustment Offsets

The `dst_adjustment_offsets` type is a collection of three [time_duration](#) objects.

Syntax	Description
	Example
<pre>dst_adjustment_offsets(...) Parameters: time_duration dst_adjust time_duration start_offset time_duration end_offset</pre>	<p>The first <code>time_duration</code> is the daylight savings adjustment. The second is the time which daylight savings starts on the start day. The third is the time daylight savings ends on the ending day.</p> <pre>dst_adjustment_offsets(hours(1), hours(2), hours(2));</pre>

Daylight Savings Calc Rules

Daylight savings calc rules, named `dst_calc_rules`, are a series of objects that group appropriate [date_generators](#) together to form rule sets. The individual rules objects are used via `dst_calc_rule_ptr`.

For a complete example of all five `dst_calc_rule` types, see: [calc_rules example](#).

Syntax	Description
<pre>partial_date_dst_rule(...) Parameters: start_rule end_rule</pre>	Both the start and end rules are of type <code>gregorian::partial_date</code> .
<pre>first_last_dst_rule(...) Parameters: start_rule end_rule</pre>	The DST start rule is of type <code>gregorian::first_day_of_the_week_in_month</code> and the end rule is of type <code>gregorian::last_day_of_the_week_in_month</code> .
<pre>last_last_dst_rule(...) Parameters: start_rule end_rule</pre>	Both the start and end rules are of type <code>gregorian::last_day_of_the_week_in_month</code> .
<pre>nth_last_dst_rule(...) Parameters: start_rule end_rule</pre>	The DST start rule is of type <code>gregorian::nth_day_of_the_week_in_month</code> and the end rule is of type <code>gregorian::last_day_of_the_week_in_month</code> .
<pre>nth_kday_dst_rule(...) Parameters: start_rule end_rule (see note* below)</pre>	Both rules are of type <code>gregorian::nth_day_of_the_week_in_month</code> .

* Note: The name "nth_kday_dst_rule" is a bit cryptic. Therefore, a more descriptive name, "nth_day_of_the_week_in_month_dst_rule", is also provided.

Local Date Time

[Introduction](#) -- [Header](#) -- [Construct From Clock](#) -- [Construction](#) -- [Accessors](#) -- [Operators](#) -- [Struct tm Functions](#)

Introduction

A `local_date_time` object is a point in time and an associated time zone. The time is represented internally as UTC.

Header

The inclusion of a single header will bring in all `boost::local_time` types, functions, and IO operators.

```
#include "boost/date_time/local_time/local_time.hpp"
└
```

Construct From Clock

Creation of a `local_date_time` object from clock is possible with either second, or sub second resolution.

Syntax	Example
<pre>local_microsec_clock(...) Return Type: local_date_time Parameter: time_zone_ptr</pre>	<pre>time_zone_ptr zone(new posix_time_zone("MST-07")); local_date_time ldt = local_microsec_clock::local_time(zone);</pre>
<pre>local_sec_clock(...) Return Type: local_date_time Parameter: time_zone_ptr</pre>	<pre>time_zone_ptr zone(new posix_time_zone("MST-07")); local_date_time ldt = local_sec_clock::local_time(zone);</pre>

Construction

Construction of a `local_date_time` object can be done with a `ptime` and a `time_zone_ptr` where the `ptime` represents UTC time. Construction with a wall-clock representation takes the form of a date, a `time_duration`, a `time_zone_ptr`, and a fourth parameter that addresses the following complication.

Construction from a wall-clock rep may result in differing shifts for a particular time zone, depending on daylight savings rules for that zone. This means it is also possible to create a `local_date_time` with a non-existent, or duplicated, UTC representation. These cases occur during the forward shift in time that is the transition into daylight savings and during the backward shift that is the transition out of daylight savings. The user has two options for handling these cases: a bool flag that states if the time is daylight savings, or an enum that states what to do when either of these cases are encountered.

The bool flag is ignored when the given time_zone has no daylight savings specification. When the daylight savings status of a given time label is calculated and it does not match the flag, a `local_time::dst_not_valid` exception is thrown. If a time label is invalid (does not exist), a `local_time::time_label_invalid` exception is thrown.

There are two elements in the `local_date_time::DST_CALC_OPTIONS` enum: `EXCEPTION_ON_ERROR` and `NOT_DATE_TIME_ON_ERROR`. The possible exceptions thrown are a `local_time::ambiguous_result` or a `local_time::time_label_invalid`. The `NOT_DATE_TIME_ON_ERROR` sets the time value to the special value `local_time::not_a_date_time` in the event of either a invalid or an ambiguous time label.

Syntax	Description
<pre>local_date_time(...) Parameters: posix_time::ptime time_zone_ptr</pre>	<p data-bbox="810 271 1487 300">Example</p> <p data-bbox="810 333 1487 425">The given time is expected to be UTC. Therefore, the given time will be adjusted according to the offset described in the time zone.</p> <pre data-bbox="826 504 1471 678"> // 3am, 2004-Nov-05 local time ptime pt(date(2004,Nov,5), hours(10)); time_zone_ptr zone(new posix_time_zone("MST-07")); local_date_time az(pt, zone); </pre>
<pre>local_date_time(...) Parameters: date time_duration time_zone_ptr bool</pre>	<p data-bbox="810 750 1487 873">The passed time information understood to be in the passed tz. The DST flag must be passed to indicate whether the time is in daylight savings or not. May throw a <code>dst_not_valid</code> or <code>time_label_invalid</code> exception.</p> <pre data-bbox="826 925 1471 1122"> date d(2004,Nov,5); time_duration td(5,0,0,0); string z("PST-8PDT,M4.1.0,M10.1.0") time_zone_ptr zone(new posix_time_zone(z)); local_date_time nyc(d, td, zone, false); </pre>
<pre>local_date_time(...) Parameters: date time_duration time_zone_ptr DST_CALC_OPTIONS</pre>	<p data-bbox="810 1198 1487 1321">The passed time information understood to be in the passed tz. The DST flag is calculated according to the specified rule. May throw a <code>ambiguous_result</code> or <code>time_label_invalid</code> exception.</p> <pre data-bbox="826 1373 1471 1570"> date d(2004,Nov,5); time_duration td(5,0,0,0); string z("PST-8PDT,M4.1.0,M10.1.0") time_zone_ptr zone(new posix_time_zone(z)); local_date_time nyc(d, td, zone, NOT_DATE_TIME_ON_ERROR); </pre>
<pre>local_date_time(local_date_time);</pre>	<p data-bbox="810 1646 1008 1675">Copy Constructor.</p> <pre data-bbox="826 1720 1471 1749"> local_date_time az_2(az); </pre>

Syntax	Description
	Example
<pre>local_date_time(...) Parameters: special_values time_zone_ptr</pre>	<p>Special Values constructor.</p> <pre>time_zone_ptr zone(new posix_time_zone("MST-07")); local_date_time nadt(not_a_date_time, zone); // default NULL time_zone_ptr local_date_time nadt(pos_infin);</pre>

Accessors

Syntax	Description
	Example
<code>time_zone_ptr zone()</code>	Returns associated <code>time_zone</code> object via a <code>time_zone_ptr</code>
<code>bool is_dst()</code>	Determines if time value is in DST for associated zone.
<code>ptime utc_time()</code>	Converts the local time value to a UTC value. <pre>ptime pt(date(2004,Nov,5), hours(10)); time_zone_ptr zone(new posix_time_zone("MST-07")); local_date_time az(pt, zone); az.utc_time(); // 10am 2004-Nov-5</pre>
<code>ptime local_time()</code>	Returns the local time for this object (Wall-clock). <pre>ptime pt(date(2004,Nov,5), hours(10)); time_zone_ptr zone(new posix_time_zone("MST-07")); local_date_time az(pt, zone); az.utc_time(); // 10am 2004-Nov-5 az.local_time(); // 3am 2004-Nov-5</pre>
<pre>local_time_in(...) Return Type: local_date_time Parameters: time_zone_ptr time_duration</pre>	Returns a <code>local_date_time</code> representing the same UTC time as calling object, plus optional <code>time_duration</code> , with given time zone. <pre>local_date_time nyc = az.local_time_in(nyc_zone); // nyc == 7am 2004-Nov-5</pre>
<code>bool is_infinity() const</code>	Returns true if <code>local_date_time</code> is either positive or negative infinity <pre>local_date_time ldt(pos_infin); ldt.is_infinity(); // --> true</pre>
<code>bool is_neg_infinity() const</code>	Returns true if <code>local_date_time</code> is negative infinity <pre>local_date_time ldt(neg_infin); ldt.is_neg_infinity(); // --> true</pre>

Syntax	Description
	Example
<code>bool is_pos_infinity() const</code>	<p>Returns true if local_date_time is positive infinity</p> <pre>local_date_time ldt(neg_infin); ldt.is_pos_infinity(); // --> true</pre>
<code>bool is_not_a_date_time() const</code>	<p>Returns true if value is not a date</p> <pre>local_date_time ldt(not_a_date_time); ldt.is_not_a_date_time(); // --> true</pre>
<code>bool is_special() const</code>	<p>Returns true if local_date_time is any special_value</p> <pre>local_date_time ldt(pos_infin); local_date_time ldt2(not_a_date_time); time_zone_ptr mst(new posix_time_zone("MST-07")); local_date_time ldt3(local_sec_clock::local_time(mst)); ldt.is_special(); // --> true ldt2.is_special(); // --> true ldt3.is_special(); // --> false</pre>

Operators

Syntax	Description
	Example
<code>operator<<</code>	<p>Output streaming operator. This operator is part of the v1.33 IO addition to <code>date_time</code>. For complete details on this feature see Date Time IO. The default output is shown in this example.</p> <pre>time_zone_ptr zone(new posix_time_zone("MST-07")); local_date_time ldt(date(2005,Jul,4), hours(20), false); std::cout << ldt << std::endl; // "2005-Jul-04 20:00:00 MST"</pre>
<code>operator>></code>	<p>Input streaming operator. This operator is part of the v1.33 IO addition to <code>date_time</code>. For complete details on this feature see Date Time IO. At this time, <code>local_date_time</code> objects can only be streamed in with a Posix Time Zone string. A complete description of a Posix Time Zone string can be found in the documentation for the posix_time_zone class.</p> <pre>stringstream ss; ss.str("2005-Jul-04 20:00:00 MST-07"); ss >> ldt;</pre>
<code>operator==, operator!=, operator>, operator<, operator>=, operator<=</code>	<p>A full complement of comparison operators</p> <pre>ldt1 == ldt2, etc</pre>
<code>operator+, operator+=, operator-, operator-=</code>	<p>Addition, subtraction, and shortcut operators for <code>local_date_time</code> and date duration types. These include: days, months, and years.</p> <pre>ldt + days(5), etc</pre>
<code>operator+, operator+=, operator-, operator-=</code>	<p>Addition, subtraction, and shortcut operators for <code>local_date_time</code> and <code>time_duration</code>.</p> <pre>ldt + hours(5), etc</pre>

Struct tm Functions

Function for converting a `local_date_time` object to a `tm` struct is provided.

Syntax	Description
	Example
<pre>tm to_tm(local_date_time)</pre>	<p>A function for converting a <code>local_date_time</code> object to a <code>tm</code> struct.</p> <pre>// 6am, 2005-Jul-05 local time std::string z("EST-05EDT,M4.1.0,M10.1.0"); ptime pt(date(2005,Jul,5), hours(10)); time_zone_ptr zone(new posix_time_zone(z)); local_date_time ldt(pt, zone); tm ldt_tm = to_tm(ldt); /* tm_year => 105 tm_mon => 6 tm_mday => 5 tm_wday => 2 (Tuesday) tm_yday => 185 tm_hour => 6 tm_min => 0 tm_sec => 0 tm_isdst => 1 */</pre>

Local Time Period

[Introduction](#) -- [Header](#) -- [Construction](#) -- [Accessors](#) -- [Operators](#)

Introduction

The class `boost::local_time::local_time_period` provides direct representation for ranges between two local times. Periods provide the ability to simplify some types of calculations by simplifying the conditional logic of the program.

A period that is created with beginning and end points being equal, or with a duration of zero, is known as a zero length period. Zero length periods are considered invalid (it is perfectly legal to construct an invalid period). For these periods, the `last` point will always be one unit less than the `begin` point.

Header

```
#include "boost/date_time/local_time/local_time.hpp" //include all types plus i/o
or
#include "boost/date_time/local_time/local_time_types.hpp" //no i/o just types
```

Construction

Syntax	Description
<pre>local_time_period(...) Parameters: local_date_time beginning local_date_time end</pre>	<p>Example</p> <p>Create a period as [begin, end). If end is \leq begin then the period will be defined as invalid.</p> <pre>time_zone_ptr zone(new posix_time_zone("MST-07")); local_date_time beg(ptime(date(2005,Jan,1),hours(0)), zone); local_date_time end(ptime(date(2005,Feb,1),hours(0)), zone); // period for the entire month of Jan 2005 local_time_period ltp(beg, end);</pre>
<pre>local_time_period(...) Parameters: local_date_time beginning time_duration length</pre>	<p>Create a period as [begin, begin+len) where end would be begin+len. If len is \leq zero then the period will be defined as invalid.</p> <pre>time_zone_ptr zone(new posix_time_zone("MST-07")); local_date_time beg(ptime(date(2005,Jan,1),hours(0)), zone); // period for the whole day of 2005-Jan-01 local_time_period ltp(beg, hours(24));</pre>
<pre>local_time_period(local_time_period rhs)</pre>	<p>Copy constructor</p> <pre>local_time_period ltp1(ltp);</pre>

Accessors

Syntax	Description
	Example
<pre>local_date_time begin()</pre>	<p>Return first local_date_time of the period.</p> <pre>time_zone_ptr zone(new posix_time_zone("MST-07")); local_date_time ldt((ptime(date(2005,Jan,1)),hours(0)), ↓ zone); local_time_period ltp(ldt, hours(2)); ltp.begin(); // => 2005-Jan-01 00:00:00</pre>
<pre>local_date_time last()</pre>	<p>Return last local_date_time in the period</p> <pre>time_zone_ptr zone(new posix_time_zone("MST-07")); local_date_time ldt((ptime(date(2005,Jan,1),hours(0))), ↓ zone); local_time_period ltp(ldt, hours(2)); ltp.last(); // => 2005-Jan-01 ↓ 01:59:59.999999999</pre>
<pre>local_date_time end()</pre>	<p>Return one past the last in period</p> <pre>time_zone_ptr zone(new posix_time_zone("MST-07")); local_date_time ldt((ptime(date(2005,Jan,1),hours(0))), ↓ zone); local_time_period ltp(ldt, hours(2)); ltp.end(); // => 2005-Jan-01 02:00:00</pre>
<pre>time_duration length()</pre>	<p>Return the length of the local_time period.</p> <pre>time_zone_ptr zone(new posix_time_zone("MST-07")); local_date_time ldt((ptime(date(2005,Jan,1),hours(0))), ↓ zone); local_time_period ltp(ldt, hours(2)); ltp.length(); // => 02:00:00</pre>

Syntax	Description
	Example
<pre>bool is_null()</pre>	<p>True if period is not well formed. eg: end less than or equal to begin.</p> <pre>time_zone_ptr zone(new posix_time_zone("MST-07")); local_date_time beg((ptime(date(2005, Feb, 1), hours(0))), ↵ zone); local_date_time end((ptime(date(2005, Jan, 1), hours(0))), ↵ zone); local_time_period ltp(beg, end); ltp.is_null(); // => true</pre>
<pre>bool contains(local_date_time)</pre>	<p>True if local_date_time is within the period. Zero length periods cannot contain any points</p> <pre>time_zone_ptr zone(new posix_time_zone("MST-07")); local_date_time beg((ptime(date(2005, Jan, 1), hours(0))), ↵ zone); local_date_time end((ptime(date(2005, Feb, 1), hours(0))), ↵ zone); local_time_period jan_mst(beg, end); local_date_time ldt((ptime(date(2005, Jan, 15), hours(12))), ↵ zone); jan_mst.contains(ldt); // => true local_time_period zero(beg, beg); zero.contains(beg); // false</pre>
<pre>bool contains(local_time_period)</pre>	<p>True if period is within the period</p> <pre>// using jan_mst period from previous example local_date_time beg((ptime(date(2005, Jan, 7), hours(0))), ↵ zone); local_time_period ltp(beg, hours(24)); jan_mst.contains(ltp); // => true</pre>

Syntax	Description
<pre>bool intersects(local_time_period)</pre>	<p data-bbox="801 253 1497 315">Example</p> <p data-bbox="801 315 1497 383">True if periods overlap</p> <pre data-bbox="801 383 1497 786"> // using jan_mst period from previous example local_date_time beg((ptime(date(2005,Jan,7),hours(0))), ↓ zone); local_date_time end((ptime(date(2005,Feb,7),hours(0))), ↓ zone); local_time_period ltp(beg, end); jan_mst.intersects(ltp); // => true </pre>
<pre>local_time_period intersection(local_time_peri↓ od)</pre>	<p data-bbox="801 786 1497 853">Calculate the intersection of 2 periods. Null if no intersection.</p> <pre data-bbox="801 853 1497 1341"> // using jan_mst period from previous example local_date_time beg((ptime(date(2005,Jan,7),hours(0))), ↓ zone); local_date_time end((ptime(date(2005,Feb,7),hours(0))), ↓ zone); local_time_period ltp(beg, end); local_time_period res(jan_mst.intersec↓ tion(ltp)); // res => 2005-Jan-07 00:00:00 through // 2005-Jan-31 23:59:59.999999999 (inclusive) </pre>
<pre>local_time_period merge(local_time_period)</pre>	<p data-bbox="801 1341 1497 1408">Returns union of two periods. Null if no intersection.</p> <pre data-bbox="801 1408 1497 1854"> // using jan_mst period from previous example local_date_time beg((ptime(date(2005,Jan,7),hours(0))), ↓ zone); local_date_time end((ptime(date(2005,Feb,7),hours(0))), ↓ zone); local_time_period ltp(beg, end); local_time_period res(jan_mst.merge(ltp)); // res => 2005-Jan-07 00:00:00 through // 2005-Feb-06 23:59:59.999999999 (inclusive) </pre>

Syntax	Description
<pre>local_time_period span(local_time_period)</pre>	<p data-bbox="810 271 1487 302">Example</p> <p data-bbox="810 333 1487 396">Combines two periods and any gap between them such that begin = min(p1.begin, p2.begin) and end = max(p1.end , p2.end).</p> <pre data-bbox="826 445 1471 817"> // using jan_mst period from previous example local_date_time beg((ptime(date(2005,Mar,1),hours(0))), ↓ zone); local_date_time end((ptime(date(2005,Apr,1),hours(0))), ↓ zone); local_time_period mar_mst(beg, end); local_time_period res(jan_mst.span(mar_mst)); // res => 2005-Jan-01 00:00:00 through // 2005-Mar-31 23:59:59.999999999 (inclusive) </pre>
<pre>void shift(time_duration)</pre>	<p data-bbox="810 893 1198 925">Add duration to both begin and end.</p> <pre data-bbox="826 974 1471 1283"> local_date_time beg((ptime(date(2005,Mar,1),hours(0))), ↓ zone); local_date_time end((ptime(date(2005,Apr,1),hours(0))), ↓ zone); local_time_period mar_mst(beg, end); mar_mst.shift(hours(48)); // mar_mst => 2005-Mar-03 00:00:00 through // 2005-Apr-02 23:59:59.999999999 (inclusive) </pre>

Operators

Syntax	Description
	Example
<code>operator==, operator!=</code>	<p>Equality operators. Periods are equal if <code>ltp1.begin == ltp2.begin</code> && <code>ltp1.last == ltp2.last</code></p> <pre>if (ltp1 == ltp2) {...</pre>
<code>operator<</code>	<p>Ordering with no overlap. True if <code>ltp1.end()</code> less than <code>ltp2.begin()</code></p> <pre>if (ltp1 < ltp2) {...</pre>
<code>operator></code>	<p>Ordering with no overlap. True if <code>ltp1.begin()</code> greater than <code>ltp2.end()</code></p> <pre>if (ltp1 > ltp2) {... etc</pre>
<code>operator<=, operator>=</code>	<p>Defined in terms of the other operators.</p>

Date Time Input/Output

Date Time IO System

Exception Handling on Streams

As of version 1.33, the `date_time` library utilizes a new IO streaming system. This new system gives the user great control over how dates and times can be represented. The customization options can be broken down into two groups: format flags and string elements. Format flags provide flexibility in the order of the date elements as well as the type. Customizing the string elements allows the replacement of built in strings from month names, weekday names, and other strings used in the IO.

The output system is based on a `date_facet` (derived from `std::facet`), while the input system is based on a `date_input_facet` (also derived from `std::facet`). The time and local_time facets are derived from these base types. The output system utilizes three formatter objects, whereas the input system uses four parser objects. These formatter and parser objects are also customizable.

It is important to note, that while all the examples shown here use narrow streams, there are wide stream facets available as well (see [IO Objects](#) for a complete list).

It should be further noted that not all compilers are capable of using this IO system. For those compilers the IO system used in previous `date_time` versions is still available. The "legacy IO" is automatically selected for these compilers, however, the legacy IO system can be manually selected by defining `USE_DATE_TIME_PRE_1_33_FACET_IO`. See the [Build-Compiler Information](#) for more information.

Exception Handling on Streams

When an error occurs during the input streaming process, the `std::ios_base::failbit` will (always) be set on the stream. It is also possible to have exceptions thrown when an error occurs. To "turn on" these exceptions, call the stream's `exceptions` function with a parameter of `std::ios_base::failbit`.

```
// "Turning on" exceptions
date d(not_a_date_time);
std::stringstream ss;
ss.exceptions(std::ios_base::failbit);
ss.str("204-Jan-01");
ss >> d; // throws bad_year exception AND sets failbit on stream
```

A simple example of this new system:

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

Format Flags

Many of the format flags this new system uses for output are those used by `strftime(...)`, but not all. Some new flags have been added, and others overridden. The input system supports only specific flags, therefore, not all flags that work for output will work with input (we are currently working to correct this situation).

The following tables list the all the flags available for both `date_time` IO as well as `strftime`. Format flags marked with a single asterisk (*) have a behavior unique to `date_time`. Those flags marked with an exclamation point (!) are not usable for input (at this time). The flags marked with a hash sign (#) are implemented by system locale and are known to be missing on some platforms. The first table is for dates, and the second table is for times.

Date Facet Format Flags

Format Specifier	Description
	Example
%a	Abbreviated weekday name "Mon" => Monday
%A	Long weekday name "Monday"
%b	Abbreviated month name "Feb" => February
%B	Full month name "February"
%c !	The preferred date and time representation for the current locale.
%C !#	The century number (year/100) as a 2-digit integer.
%d	Day of the month as decimal 01 to 31
%D !#	Equivalent to %m/%d/%y
%e #	Like %d, the day of the month as a decimal number, but a leading zero is replaced by a space
%G !	This has the same format and value as %y, except that if the ISO week number belongs to the previous or next year, that year is used instead.

Format Specifier	Description
	Example
%g !	Like %G, but without century.
%h !#	Equivalent to %b
%j	Day of year as decimal from 001 to 366 for leap years, 001 - 365 for non-leap years. "060" => Feb-29
%m	Month name as a decimal 01 to 12 "01" => January
%u !	The day of the week as a decimal, range 1 to 7, Monday being 1.
%U	The week number of the current year as a decimal number, range 00 to 53, starting with the first Sunday as the first day of week 01. In 2005, Jan 1st falls on a Saturday, so therefore it falls within week 00 of 2005 (week 00 spans 2004-Dec-26 to 2005-Jan-01. This also happens to be week 53 of 2004). <pre> date d(2005, Jan, 1); // Saturday // with format %U ss << d; // "00" d += day(1); // Sunday ss << d; // "01" beginning of week 1 </pre>
%V !#	The ISO 8601:1988 week number of the current year as a decimal number, range 01 to 53, where week 1 is the first week that has at least 4 days in the current year, and with Monday as the first day of the week.
%w	Weekday as decimal number 0 to 6 "0" => Sunday

Format Specifier	Description
	Example
%W	<p>Week number 00 to 53 where Monday is first day of week 1</p> <pre>date d(2005, Jan, 2); // Sunday // with format %W ss << d; // "00" d += day(1); // Monday ss << d; // "01" beginning of week 1</pre>
%x	<p>Implementation defined date format from the locale.</p> <pre>date d(2005,Oct,31); date_facet* f = new date_facet("%x"); locale loc = locale(locale("en_US"), f); cout.imbue(loc); cout << d; // "10/31/2005" loc = locale(locale("de_DE"), f); cout.imbue(loc); cout << d; // "31.10.2005"</pre>
%y	<p>Two digit year</p> <pre>"05" => 2005</pre>
%Y	<p>Four digit year</p> <pre>"2005"</pre>
%Y-%b-%d	<p>Default date format</p> <pre>"2005-Apr-01"</pre>
%Y%m%d	<p>ISO format</p> <pre>"20050401"</pre>
%Y-%m-%d	<p>ISO extended format</p> <pre>"2005-04-01"</pre>

Time Facet Format Flags

Format Specifier	Description
	Example
%- *!	Placeholder for the sign of a duration. Only displays when the duration is negative. "-13:15:16"
%+ *!	Placeholder for the sign of a duration. Always displays for both positive and negative. "+13:15:16"
%f	Fractional seconds are always used, even when their value is zero "13:15:16.000000"
%F *	Fractional seconds are used only when their value is not zero. "13:15:16" "05:04:03.001234"
%H	The hour as a decimal number using a 24-hour clock (range 00 to 23).
%I !	The hour as a decimal number using a 12-hour clock (range 01 to 12).
%k !	The hour (24-hour clock) as a decimal number (range 0 to 23); single digits are preceded by a blank.
%l !	The hour (12-hour clock) as a decimal number (range 1 to 12); single digits are preceded by a blank.
%M	The minute as a decimal number (range 00 to 59).

Format Specifier	Description
	Example
%O	The number of hours in a time duration as a decimal number (range 0 to max. representable duration); single digits are preceded by a zero.
%p !	Either `AM' or `PM' according to the given time value, or the corresponding strings for the current locale.
%P !#	Like %p but in lowercase: `am' or `pm' or a corresponding string for the current locale.
%r !#	The time in a.m. or p.m. notation. In the POSIX locale this is equivalent to `%I:%M:%S %p'
%R !	The time in 24-hour notation (%H:%M)
%S *	Seconds with fractional seconds.
	"59.000000"
%S	Seconds only
	"59"
%T !	The time in 24-hour notation (%H:%M:%S)
%q	ISO time zone (output only). This flag is ignored when using the time_facet with a ptime.
	"-0700" // Mountain Standard Time
%Q	ISO extended time zone (output only). This flag is ignored when using the time_facet with a ptime.
	"-05:00" // Eastern Standard Time

Format Specifier	Description
	Example
<code>%z *!</code>	Abbreviated time zone (output only). This flag is ignored when using the <code>time_facet</code> with a <code>ptime</code> . "MST" // Mountain Standard Time
<code>%Z *!</code>	Full time zone name (output only). This flag is ignored when using the <code>time_facet</code> with a <code>ptime</code> . "EDT" // Eastern Daylight Time
<code>%ZP *</code>	Posix time zone string (available to both input and output). This flag is ignored when using the <code>time_facet</code> with a <code>ptime</code> . For complete details on posix time zone strings, see posix_time_zone class . "EST-05EDT+01,M4.1.0/02:00,M10.5.0/02:00"
<code>%x %X</code>	Implementation defined date/time format from the locale. <pre> date d(2005,Oct,31); ptime pt(d, hours(20)); time_facet* f = new time_facet("%x %X"); locale loc = locale(locale("en_US"), f); cout.imbue(loc); cout << pt; // "10/31/2005 08:00:00 PM" loc = locale(locale("de_DE"), f); cout.imbue(loc); cout << pt; // "31.10.2005 20:00:00" </pre>
<code>%Y%m%dT%H%M%S%F%q</code>	ISO format "20051015T131211-0700" // Oct 15, 2005 ↴ 13:12:11 MST
<code>%Y-%m-%d %H:%M:%S%F%Q</code>	Extended ISO format "2005-10-15 13:12:11-07:00"
<code>%Y-%b-%d %H:%M:%S%F %z</code>	Default format used when outputting <code>ptime</code> and <code>local_date_time</code> . "2005-Oct-15 13:12:11 MST"

Format Specifier	Description
	Example
<code>%Y-%b-%d %H:%M:%S%F %ZP</code>	Default format used when inputting ptime and local_date_time. <code>"2005-Oct-15 13:12:11 MST-07"</code>
<code>%-H:%M:%S%F !</code>	Default time_duration format for output. Sign will only be displayed for negative durations. <code>"-13:14:15.003400"</code>
<code>%H:%M:%S%F</code>	Default time_duration format for input. <code>"13:14:15.003400"</code>

* Signifies flags that have a behavior unique to date_time.

Signifies flags that have a platform-dependent behavior. These may not be supported everywhere.

! Signifies flags that currently do not work for input.

The following table lists the available facets.

IO Objects

Output	Input
date_facet	date_input_facet
wdate_facet	wdate_input_facet
time_facet	time_input_facet
wtime_facet	wtime_input_facet
local_time_facet *	local_time_input_facet *
wlocal_time_facet *	wlocal_time_input_facet *

* These links lead to the [time_facet](#) and [time_input_facet](#) reference sections. They are not actual classes but typedefs.

Formatter/Parser Objects

To implement the new i/o facets the date-time library uses a number of new parsers and formatters. These classes are available for users that want to implement specialized input/output routines.

Output	Input
period_formatter	period_parser
date_generator_formatter	date_generator_parser
special_values_formatter	special_values_parser
	format_date_parser

Date Facet

[Introduction](#) - [Construction](#) - [Accessors](#)

Introduction

The `boost::date_time::date_facet` enables users to have significant control over the output streaming of dates (and other gregorian objects). The `date_facet` is typedef'd in the `gregorian` namespace as `date_facet` and `wdate_facet`.

Construction

Syntax	Description
<code>date_facet()</code>	Default constructor
<pre>date_facet(...) Parameters: char_type* format input_collection_type</pre>	Format given will be used for date output. All other formats will use their defaults. Collection is the set of short names to be used for months. All other name collections will use their defaults.
<pre>date_facet(...) Parameters: char_type* format period_formatter_type special_values_formatter_type date_gen_formatter_type</pre>	Format given will be used for date output. The remaining parameters are formatter objects. Further details on these objects can be found here . This constructor also provides default arguments for all parameters except the format. Therefore, <code>date_facet("%m %d %Y")</code> will work.

Accessors

Syntax	Description
	Example
<pre>void format(char_type*)</pre>	<p>Set the format for dates.</p> <pre>date_facet* f = new date_facet(); f->format("%m %d %Y");</pre>
<pre>void set_iso_format()</pre>	<p>Sets the date format to ISO</p> <pre>f->set_iso_format(); // "%Y%m%d"</pre>
<pre>void set_iso_extended_format()</pre>	<p>Sets the date format to ISO Extended</p> <pre>f->set_iso_extended_format(); // "%Y-%m-%d"</pre>
<pre>void month_format(char_type*)</pre>	<p>Set the format for months when they are 'put' individually.</p> <pre>f->month_format("%B"); ss << greg_month(12); // "December"</pre>
<pre>void weekday_format(char_type*)</pre>	<p>Set the format for weekdays when they are 'put' individually.</p> <pre>f->weekday_format("%a"); ss << greg_weekday(2); // "Tue"</pre>
<pre>void period_formatter(...) Parameter: period_formatter_type</pre>	<p>Replaces the period formatter object with a user created one.</p> <p>see the tutorial for a complete example.</p>
<pre>void special_values_formatter(...) Parameter: special_values_formatter_type</pre>	<p>Replaces the special_values formatter object with a user created one.</p> <p>see the tutorial for a complete example.</p>
<pre>void date_gen_phrase_strings(...) Parameters: input_collection_type date_gen_formatter_type:: phrase_elements</pre>	<p>Sets new date generator phrase strings in date_gen_formatter. The input collection is a vector of strings (for details on these strings see date generator formatter/parser documentation). The phrase_elements parameter is an enum, defined in the date_generator_formatter object, that has a default value of 'first'. It is used to indicate what the position of the first string in the collection will be.</p>

Syntax	Description
	Example
<pre>void short_weekday_names(...) Parameter: input_collection_type</pre>	<p>Replace strings used when 'putting' short weekdays.</p> <p>see the tutorial for a complete example.</p>
<pre>void long_weekday_names(...) Parameter: input_collection_type</pre>	<p>Replace strings used when 'putting' long weekdays.</p> <p>see the tutorial for a complete example.</p>
<pre>void short_month_names(...) Parameter: input_collection_type</pre>	<p>Replace strings used when 'putting' short months.</p> <p>see the tutorial for a complete example.</p>
<pre>void long_month_names(...) Parameter: input_collection_type</pre>	<p>Replace strings used when 'putting' long months.</p> <p>see the tutorial for a complete example.</p>
<pre>OutItrT put(...) Common parameters for all 'put' functions: OutItrT ios_base char_type Unique parameter for 'put' funcs: gregorian object</pre>	<p>There are 12 put functions in the <code>date_facet</code>. The common parameters are: an iterator pointing to the next item in the stream, an <code>ios_base</code> object, and the fill character. Each unique gregorian object has it's own put function. Each unique put function is described below.</p>
<pre>OutItrT put(..., date)</pre>	<p>Puts a date object into the stream using the format set by <code>format(...)</code> or the default.</p>
<pre>OutItrT put(..., days)</pre>	<p>Puts a days object into the stream as a number.</p>
<pre>OutItrT put(..., month)</pre>	<p>Puts a month object into the stream using the format set by <code>month_format(...)</code> or the default.</p>
<pre>OutItrT put(..., day)</pre>	<p>Puts a day of month object into the stream as a two digit number.</p> <pre>"01" // January 1st</pre>

Syntax	Description
<pre>OutItrT put(..., day_of_week)</pre>	<p>Example</p> <p>Puts a day of week object into the stream using the format set by <code>weekday_format(...)</code> or the default.</p> <pre></pre>
<pre>OutItrT put(..., date_period)</pre>	<p>Puts a <code>date_period</code> into the stream. The format of the dates will use the format set by <code>format(...)</code> or the default date format. The type of period (open or closed range) and the delimiters used are those used by the <code>period_formatter</code>.</p> <pre></pre>
<pre>OutItrT put(..., partial_date)</pre>	<p>Puts a <code>partial_date</code> <code>date_generator</code> object into the stream. The month format used is set by <code>month_format(...)</code> or the default. The day of month is represented as a two digit number.</p> <pre>"01 Jan" // default formats "01 January" // long month format</pre>
<pre>OutItrT put(..., date_generator) Date Generator Type: nth_day_of_the_week_in_month</pre>	<p>Puts a <code>nth_day_of_the_week_in_month</code> object into the stream. The month format is set by <code>month_format(...)</code> or the default. The weekday format is set by <code>weekday_format(...)</code> or the default. The remaining phrase elements are set in the date_generator_formatter.</p> <pre>"third Fri in May" // defaults</pre>
<pre>OutItrT put(..., date_generator) Date Generator Type: first_day_of_the_week_in_month</pre>	<p>Puts a <code>first_day_of_the_week_in_month</code> object into the stream. The month format is set by <code>month_format(...)</code> or the default. The weekday format is set by <code>weekday_format(...)</code> or the default. The remaining phrase elements are set in the date_generator_formatter.</p> <pre>"first Wed of Jun" // defaults</pre>
<pre>OutItrT put(..., date_generator) Date Generator Type: last_day_of_the_week_in_month</pre>	<p>Puts a <code>last_day_of_the_week_in_month</code> object into the stream. The month format is set by <code>month_format(...)</code> or the default. The weekday format is set by <code>weekday_format(...)</code> or the default. The remaining phrase elements are set in the date_generator_formatter.</p> <pre>"last Tue of Mar" // defaults</pre>

Syntax	Description
	Example
<pre>OutItrT put(..., date_generator) Date Generator Type: first_day_of_the_week_after</pre>	<p>Puts a <code>first_day_of_the_week_after</code> object into the stream. The weekday format is set by <code>weekday_format(...)</code> or the default. The remaining phrase elements are set in the date_generator_formatter.</p> <pre>"first Sat after" // defaults</pre>
<pre>OutItrT put(..., date_generator) Date Generator Type: first_day_of_the_week_before</pre>	<p>Puts a <code>first_day_of_the_week_before</code> object into the stream. The weekday format is set by <code>weekday_format(...)</code> or the default. The remaining phrase elements are set in the date_generator_formatter.</p> <pre>"first Mon before" // defaults</pre>

Date Input Facet

[Introduction](#) - [Construction](#) - [Accessors](#)

Introduction

The `boost::date_time::date_input_facet` enables users to have significant control how dates (and other gregorian objects) are streamed in. The `date_input_facet` is typedef'd in the `gregorian` namespace as `date_input_facet` and `wdate_input_facet`.

Construction

Syntax	Description
<pre>date_input_facet()</pre>	Default constructor
<pre>date_input_facet(string_type format)</pre>	Format given will be used for date input. All other formats will use their defaults.
<pre>date_input_facet(...) Parameters: string_type format format_date_parser_type special_values_parser_type period_parser_type date_gen_parser_type</pre>	Format given will be used for date input. The remaining parameters are parser objects. Further details on these objects can be found here .

Accessors

Syntax	Description
	Example
<code>void format(char_type*)</code>	<p>Set the format for dates.</p> <pre>date_input_facet* f = new date_input_facet(); f->format("%m %d %Y");</pre>
<code>void set_iso_format()</code>	<p>Sets the date format to ISO</p> <pre>f->set_iso_format(); // "%Y%m%d"</pre>
<code>void set_iso_extended_format()</code>	<p>Sets the date format to ISO Extended</p> <pre>f->set_iso_extended_format(); // "%Y-%m-%d"</pre>
<code>void month_format(char_type*)</code>	<p>Set the format when 'get'ing months individually.</p> <pre>f->month_format("%B"); ss.str("March"); ss >> m; // March</pre>
<code>void weekday_format(char_type*)</code>	<p>Set the format when 'get'ing weekdays individually.</p> <pre>f->weekday_format("%a"); ss.str("Sun"); ss >> wd; // Sunday</pre>
<code>void year_format(char_type*)</code>	<p>Set the format when 'get'ing years individually.</p> <pre>f->weekday_format("%y"); ss.str("04"); ss >> year; // 2004</pre>
<code>void period_parser(...)</code> Parameter: <code>period_parser_type</code>	<p>Replaces the period parser object with a user created one.</p> <p>see the tutorial for a complete example.</p>
<code>void special_values_parser(...)</code> Parameter: <code>special_values_parser_type</code>	<p>Replaces the special_values parser object with a user created one.</p> <p>see the tutorial for a complete example.</p>

Syntax	Description
	Example
<pre>void date_gen_phrase_strings(...) Parameters: input_collection_type</pre>	<p>Sets new date generator phrase strings in <code>date_gen_parser</code>. The input collection is a vector of strings (for details on these strings see date generator formatter/parser documentation).</p>
<pre>void short_weekday_names(...) Parameter: input_collection_type</pre>	<p>Replace strings used when 'getting' short weekdays.</p> <p>see the tutorial for a complete example.</p>
<pre>void long_weekday_names(...) Parameter: input_collection_type</pre>	<p>Replace strings used when 'getting' long weekdays.</p> <p>see the tutorial for a complete example.</p>
<pre>void short_month_names(...) Parameter: input_collection_type</pre>	<p>Replace strings used when 'getting' short months.</p> <p>see the tutorial for a complete example.</p>
<pre>void long_month_names(...) Parameter: input_collection_type</pre>	<p>Replace strings used when 'getting' long months.</p> <p>see the tutorial for a complete example.</p>
<pre>InItrT get(...) Common parameters for all 'get' functions: InItrT from InItrT to ios_base Unique parameter for 'get' funcs: gregorian object</pre>	<p>There are 13 get functions in the <code>date_input_facet</code>. The common parameters are: an iterator pointing to the beginning of the stream, an iterator pointing to the end of the stream, and an <code>ios_base</code> object. Each unique gregorian object has it's own get function. Each unique get function is described below.</p>
<pre>InItrT get(..., date)</pre>	<p>Gets a date object from the stream using the format set by <code>format(...)</code> or the default.</p> <pre>ss.str("2005-Jan-01"); ss >> d; // default format</pre>
<pre>InItrT get(..., month)</pre>	<p>Gets a month object from the stream using the format set by <code>month_format(...)</code> or the default.</p> <pre>ss.str("Feb"); ss >> m; // default format</pre>

Syntax	Description
<pre>InItrT get(..., day_of_week)</pre>	<p>Example</p> <p>Gets a day of week object from the stream using the format set by <code>weekday_format(...)</code> or the default.</p> <pre>ss.str("Sun"); ss >> dow; // default format</pre>
<pre>InItrT get(..., day)</pre>	<p>Gets a day of month object from the stream as a two digit number.</p> <pre>"01" // January 1st</pre>
<pre>InItrT get(..., year)</pre>	<p>Gets a year object from the stream as a number. The number of expected digits depends on the year format.</p> <pre>ss/str("2005"); ss >> y; // default format</pre>
<pre>InItrT get(..., days)</pre>	<p>Gets a days object from the stream as a number.</p> <pre>ss.str("356"); ss >> dys; // a full year</pre>
<pre>InItrT get(..., date_period)</pre>	<p>Gets a <code>date_period</code> from the stream. The format of the dates will use the format set by <code>format(...)</code> or the default date format. The type of period (open or closed range) and the delimiters used are those used by the <code>period_parser</code>.</p> <p>see the tutorial for a complete example.</p>
<pre>InItrT get(..., partial_date)</pre>	<p>Gets a <code>partial_date</code> <code>date_generator</code> object from the stream. The month format used is set by <code>month_format(...)</code> or the default. The day of month is represented as a two digit number.</p> <pre>"01 Jan" // default formats "01 January" // long month format</pre>
<pre>InItrT get(..., date_generator) Date Generator Type: nth_day_of_the_week_in_month</pre>	<p>Gets a <code>nth_day_of_the_week_in_month</code> object from the stream. The month format is set by <code>month_format(...)</code> or the default. The weekday format is set by <code>weekday_format(...)</code> or the default. The remaining phrase elements are set in the date_generator_parser.</p> <pre>"third Fri in May" // defaults</pre>

Syntax	Description
<pre>InItrT get(..., date_generator) Date Generator Type: first_day_of_the_week_in_month</pre>	<p>Example</p> <p>Gets a <code>first_day_of_the_week_in_month</code> object from the stream. The month format is set by <code>month_format(...)</code> or the default. The weekday format is set by <code>weekday_format(...)</code> or the default. The remaining phrase elements are set in the date_generator_parser.</p> <pre>"first Wed of Jun" // defaults</pre>
<pre>InItrT get(..., date_generator) Date Generator Type: last_day_of_the_week_in_month</pre>	<p>Gets a <code>last_day_of_the_week_in_month</code> object from the stream. The month format is set by <code>month_format(...)</code> or the default. The weekday format is set by <code>weekday_format(...)</code> or the default. The remaining phrase elements are set in the date_generator_parser.</p> <pre>"last Tue of Mar" // defaults</pre>
<pre>InItrT get(..., date_generator) Date Generator Type: first_day_of_the_week_after</pre>	<p>Gets a <code>first_day_of_the_week_after</code> object from the stream. The weekday format is set by <code>weekday_format(...)</code> or the default. The remaining phrase elements are set in the date_generator_parser.</p> <pre>"first Sat after" // defaults</pre>
<pre>InItrT get(..., date_generator) Date Generator Type: first_day_of_the_week_before</pre>	<p>Gets a <code>first_day_of_the_week_before</code> object from the stream. The weekday format is set by <code>weekday_format(...)</code> or the default. The remaining phrase elements are set in the date_generator_parser.</p> <pre>"first Mon before" // defaults</pre>

Time Facet

[Introduction](#) - [Construction](#) - [Accessors](#)

Introduction

The `boost::date_time::time_facet` is an extension of the `boost::date_time::date_facet`. The `time_facet` is typedef'd in the `posix_time` namespace as `time_facet` and `wtime_facet`. It is typedef'd in the `local_time` namespace as `local_time_facet` and `wlocal_time_facet`.

Construction

Syntax	Description
<code>time_facet()</code>	Default constructor
<pre>time_facet(...) Parameters: char_type* format period_formatter_type special_values_formatter_type date_gen_formatter_type</pre>	Format given will be used for time output. The remaining parameters are formatter objects. Further details on these objects can be found here . This constructor also provides default arguments for all parameters except the format. Therefore, <code>time_facet("%H:%M:S %m %d %Y")</code> will work.

Accessors

The `time_facet` inherits all the public `date_facet` methods. Therefore, the `date_facet` methods are not listed here. Instead, they can be found by following [this](#) link.

Syntax	Description
<pre>void time_duration_format(...) Parameter: char_type*</pre>	<p>Example</p> <p>Sets the time_duration format. The time_duration format has the ability to display the sign of the duration. The '+' flag will always display the sign. The '-' will only display if the sign is negative. Currently the '-' and '+' characters are used to denote the sign.</p> <pre>f->time_duration_format("%+H%M"); // hours and minutes only w/ sign always displayed time_duration td1(3, 15, 56); time_duration td2(-12, 25, 32); ss << td1; // "+03:15:56" ss << td2; // "-12:25:56"</pre>
<pre>void set_iso_format()</pre>	<p>Sets the date and time format to ISO.</p> <pre>f->set_iso_format(); // "%Y%m%dT%H%M%S%F%Q"</pre>
<pre>void set_iso_extended_format()</pre>	<p>Sets the date and time format to ISO Extended</p> <pre>f->set_iso_extended_format(); // "%Y-%m-%d %H:%M:%S%F%Q"</pre>
<pre>OutItrT put(...) Common parameters for all 'put' functions: OutItrT ios_base char_type Unique parameter for 'put' funcs: posix_time object</pre>	<p>There are 3 put functions in the time_facet. The common parameters are: an iterator pointing to the next item in the stream, an ios_base object, and the fill character. Each unique posix_time object has it's own put function. Each unique put function is described below.</p>
<pre>OutItrT put(..., ptime)</pre>	<p>Puts a ptime object into the stream using the format set by format(...) or the default.</p>
<pre>OutItrT put(..., time_duration)</pre>	<p>Puts a time_duration object into the stream using the format set by time_duration_format(...) or the default.</p>

Syntax	Description
	Example
<pre>OutItrT put(..., time_period)</pre>	<p>Puts a <code>time_period</code> into the stream. The format of the dates and times will use the format set by <code>format(...)</code> or the default date/time format. The type of period (open or closed range) and the delimiters used are those used by the <code>period_formatter</code>.</p>

Time Input Facet

[Introduction](#) - [Construction](#) - [Accessors](#)

Introduction

The `boost::date_time::time_input_facet` is an extension of the `date_input_facet`. It is typedef'd in the `boost::posix_time` namespace as `time_input_facet` and `wtime_input_facet`. It is typedef'd in the `boost::local_time` namespace as `local_time_input_facet` and `wlocal_time_input_facet`.

Construction

Syntax	Description
<pre>time_input_facet()</pre>	Default constructor
<pre>time_input_facet(string_type)</pre>	Format given will be used for date/time input. All other formats will use their defaults.
<pre>time_input_facet(...) Parameters: string_type format format_date_parser_type special_values_parser_type period_parser_type date_gen_parser_type</pre>	Format given will be used for date/time input. The remaining parameters are parser objects. Further details on these objects can be found here .

Accessors

The `time_input_facet` inherits all the public `date_input_facet` methods. Therefore, the `date_input_facet` methods are not listed here. Instead, they can be found by following [this](#) link.

Syntax	Description
	Example
<pre>void set_iso_format()</pre>	<p>Sets the time format to ISO</p> <pre>f->set_iso_format(); // "%Y%m%dT%H%M%S%F%q" "20051225T132536.789-0700"</pre>
<pre>void set_iso_extended_format()</pre>	<p>Sets the date format to ISO Extended</p> <pre>f->set_iso_extended_format(); // "%Y-%m-%d %H:%M:%S%F %Q" "2005-12-25 13:25:36.789 -07:00"</pre>
<pre>void time_duration_format(...) Parameter: char_type*</pre>	<p>Sets the time_duration format.</p> <pre>f->time_duration_format("%H:%M"); // hours and minutes only</pre>
<pre>InItrT get(...) Common parameters for all 'get' functions: InItrT from InItrT to ios_base Unique parameter for 'get' funcs: gregorian object</pre>	<p>There are 3 get functions in the time_input_facet. The common parameters are: an iterator pointing to the beginning of the stream, an iterator pointing to the end of the stream, and an ios_base object. Each unique gregorian object has it's own get function. Each unique get function is described below.</p>
<pre>InItrT get(..., ptime)</pre>	<p>Gets a ptime object from the stream using the format set by format(...) or the default.</p> <pre>ss.str("2005-Jan-01 13:12:01"); ss >> pt; // default format</pre>
<pre>InItrT get(..., time_duration)</pre>	<p>Gets a time_duration object from the stream using the format set by time_duration_format(...) or the default.</p> <pre>ss.str("01:25:15.000123000"); ss >> td; // default format</pre>
<pre>InItrT get(..., time_period)</pre>	<p>Gets a time_period from the stream. The format of the dates/times will use the format set by format(...) or the default date and time format. The type of period (open or closed range) and the delimiters used are those used by the period_parser.</p> <p>see the tutorial for a complete example.</p>

Date Time Formatter/Parser Objects

Date Time Formatter/Parser Objects

[Periods](#) | [Date Generators](#) | [Special Values](#) | [Format Date Parser](#)

Periods

The `period_formatter` and `period_parser` provide a uniform interface for the input and output of `date_periods`, `time_periods`, and in a future release, `local_date_time_periods`. The user has control over the delimiters, formats of the date/time components, and the form the period takes. The format of the date/time components is controlled via the `date_time` input and output facets.

Period Form

Periods are constructed with open ranged parameters. The first value is the starting point, and is included in the period. The end value is not included but immediately follows the last value: `[begin/end)`. However, a period can be streamed as either an open range or a closed range.

```
[2003-Jan-01/2003-Dec-31] <-- period holding 365 days
[2003-Jan-01/2004-Jan-01) <-- period holding 365 days
```

Delimiters

There are four delimiters. The default values are

```
"\" - separator
"[" - start delimiter
")" - open range end delimiter
"]" - closed range end delimiter
```

A user can provide a custom set of delimiters. Custom delimiters may contain spaces.

Customization

The period form and delimiters can be set as construction parameters or by means of accessor functions. A custom period parser/formatter can then be used as a construction parameter to a new facet, or can be set in an existing facet via an accessor function.

Period Formatter/Parser Reference

The complete class reference can be found here: [Period Formatter Doxygen Reference](#) and here: [Period Parser Doxygen Reference](#)

Period Formatter Construction

Syntax	Description
<pre>period_formatter(...) Parameters: range_display_options char_type* char_type* char_type* char_type*</pre>	<p>NOTE: All five construction parameters have default values so this constructor also doubles as the default constructor. The <code>range_display_options</code> is a public type enum of the <code>period_formatter</code> class. The possible choices are <code>AS_OPEN_RANGE</code> or <code>AS_CLOSED_RANGE</code>. The closed range is the default. A period has three significant points: the beginning, the last, and the end. A closed range period takes the form <code>[begin,end)</code>, where an open range period takes the form <code>[begin,last]</code>. The four <code>char_type*</code> parameters are: the period separator, the start delimiter, the open range end delimiter, and the closed range end delimiter.</p>

Period Formatter Accessors

Syntax	Description
<pre>range_display_options range_option()</pre>	<p>Example</p> <p>Returns the current setting for the range display (either AS_OPEN_RANGE or AS_CLOSED_RANGE).</p> <pre></pre>
<pre>void range_option(...) Parameter: range_display_options</pre>	<p>Sets the option for range display (either AS_OPEN_RANGE or AS_CLOSED_RANGE).</p> <pre></pre>
<pre>void delimiter_strings(...) Parameters: string_type string_type string_type string_type</pre>	<p>Set new delimiter strings in the formatter.</p> <pre>string beg("-> "); string sep(" "); string opn(" -> "); string clo(" <-"); pf.delimiter_strings(beg, sep, opn, clo);</pre>
<pre>put_period_start_delimeter(...) Return Type: OutItrT Parameter: OutItrT</pre>	<p>Puts the start delimiter into the stream at position pointed to by OutItrT parameter.</p> <pre></pre>
<pre>put_period_sepatator(...) Return Type: OutItrT Parameter: OutItrT</pre>	<p>Puts the separator into the stream at position pointed to by OutItrT parameter.</p> <pre></pre>
<pre>put_period_end_delimeter(...) Return Type: OutItrT Parameter: OutItrT</pre>	<p>Puts the end delimiter into the stream at position pointed to by OutItrT parameter.</p> <pre></pre>
<pre>OutItrT put_period(...) Parameters: OutItrT ios_base char_type period_type facet_type</pre>	<p>Puts a period into the stream using the set values for delimiters, separator, and range display. The facet parameter is used to put the date (or time) objects of the period.</p> <pre></pre>

Period Parser Construction

Syntax	Description
<pre>period_parser(...) Parameters: period_range_option char_type* char_type* char_type* char_type*</pre>	<p>NOTE: All five construction parameters have default values so this constructor also doubles as the default constructor. The <code>period_range_option</code> is a public type enum of the <code>period_parser</code> class. The possible choices are <code>AS_OPEN_RANGE</code> or <code>AS_CLOSED_RANGE</code>. The closed range is the default. A period has three significant points: the beginning, the last, and the end. A closed range period takes the form <code>[begin,end)</code>, where an open range period takes the form <code>[begin,last]</code>. The four <code>char_type*</code> parameters are: the period separator, the start delimiter, the open range end delimiter, and the closed range end delimiter.</p>
<pre>period_parser(period_parser)</pre>	Copy constructor

Period Parser Accessors

Syntax	Description
	Example
<code>period_range_option range_option()</code>	Returns the current setting for the period range (either AS_OPEN_RANGE or AS_CLOSED_RANGE).
<code>void range_option(...)</code> Parameter: <code>period_range_option</code>	Sets the option for period range (either AS_OPEN_RANGE or AS_CLOSED_RANGE).
<code>void delimiter_strings(...)</code> Parameters: <code>string_type</code> <code>string_type</code> <code>string_type</code> <code>string_type</code>	Set new delimiter strings in the parser.
<code>collection_type delimiter_strings()</code>	Returns the set of delimiter strings currently held in the parser.
<code>period_type get_period(...)</code> Parameters: <code>stream_itr_type</code> <code>stream_itr_type</code> <code>ios_base</code> <code>period_type</code> <code>duration_type</code> <code>facet_type</code>	Parses a period from the stream. The iterator parameters point to the beginning and end of the stream. The duration_type is relevant to the period type, for example: A date_period would use days as a duration_type. The period will be parsed according to the formats and strings found in the facet parameter.

Date Generators

The date_generator formatter and parser provide flexibility by allowing the user to use custom "phrase elements". These phrase elements are the "in-between" words in the date_generators. For example, in the date_generator "Second Monday of March", "Second" and "of" are the phrase elements, where "Monday" and "March" are the date elements. Customization of the date elements is done with the facet. The order of the date and phrase elements cannot be changed. When parsing, all elements of the date_generator phrase must parse correctly or an ios_base::failure exception will be thrown.

Customization

The default "phrase_strings" are:

"first" "second" "third" "fourth" "fifth" "last" "before" "after" "of"

A custom set of phrase_strings must maintain this order of occurrence (Ex: "1st", "2nd", "3rd", "4th", "5th", "last", "prior", "past", "in").

Examples using default phrase_strings and default facet formats for weekday & month:

```
"first Tue of Mar"
```

And using custom phrase_strings:

```
"1st Tue in Mar"
```

The custom set of phrase elements can be set as construction parameters or through an accessor function. A custom date_generator parser/formatter can then be used as a construction parameter to a new facet, or can be set in an existing facet via an accessor function.

IMPORTANT NOTE: Prior to 1.33, partial_date was output as "1 Jan" with a single *or* double digit number for the day. The new behavior is to *always* place a double digit number for the day - "01 Jan".

Date Generator Reference

The complete class references can be found here: [Date Generator Formatter Doxygen Reference](#) and here: [Date Generator Parser Doxygen Reference](#)

Date Generator Formatter Construction

Syntax	Description
<pre>date_generator_formatter()</pre>	Uses the default date generator elements.
<pre>date_generator_formatter(...) Parameters: string_type first_element string_type second_element string_type third_element string_type fourth_element string_type fifth_element string_type last_element string_type before_element string_type after_element string_type of_element</pre>	Constructs a date_generator_formatter using the given element strings.

Date Generator Formatter Accessors

Syntax	Description
<pre>void elements(...) Parameters: collection_type phrase_elements</pre>	<p>Example</p> <p>Replace the current phrase elements with a collection of new ones. The <code>phrase_elements</code> parameter is an enum that indicates what the first element in the new collection is (defaults to first).</p> <pre>// col is a collection holding // "final", "prior", "following", // and "in" typedef date_generator_formatter dgf; dgf formatter(); formatter.elements(col, dgf::last); // complete elements in dgf are now: "first", "second", "third", "fourth", "fifth", "final", "prior", "following", and "in"</pre>
<pre>put_partial_date(...) Return Type: facet_type::OutItrT Parameters: OutItrT next ios_base char_type fill partial_date facet_type</pre>	<p>A put function for <code>partial_date</code>. This is a templated function that takes a <code>facet_type</code> as a parameter.</p> <p>Put a <code>partial_date</code> => "dd Month".</p>
<pre>put_nth_kday(...) Return Type: facet_type::OutItrT Parameters: OutItrT next ios_base char_type fill nth_kday_type facet_type</pre>	<p>A put function for <code>nth_kday_type</code>. This is a templated function that takes a <code>facet_type</code> as a parameter.</p> <p>Put an <code>nth_day_of_the_week_in_month</code> => "nth weekday of month".</p>
<pre>put_first_kday(...) Return Type: facet_type::OutItrT Parameters: OutItrT next ios_base char_type fill first_kday_type facet_type</pre>	<p>A put function for <code>first_kday_type</code>. This is a templated function that takes a <code>facet_type</code> as a parameter.</p> <p>Put a <code>first_day_of_the_week_in_month</code> => "first weekday of month".</p>

Syntax	Description
	Example
<pre> put_last_kday(...) Return Type: facet_type::OutItrT Parameters: OutItrT next ios_base char_type fill last_kday_type facet_type </pre>	<p>A put function for last_kday_type. This is a templated function that takes a facet_type as a parameter.</p>
	<p>Put a last_day_of_the_week_in_month => "last weekday of month".</p>
<pre> put_kday_before(...) Return Type: facet_type::OutItrT Parameters: OutItrT next ios_base char_type fill kday_before_type facet_type </pre>	<p>A put function for kday_before_type. This is a templated function that takes a facet_type as a parameter.</p>
	<p>Put a first_day_of_the_week_before => "weekday before"</p>
<pre> put_kday_after(...) Return Type: facet_type::OutItrT Parameters: OutItrT next ios_base char_type fill kday_after_type facet_type </pre>	<p>A put function for kday_after_type. This is a templated function that takes a facet_type as a parameter.</p>
	<p>Put a first_day_of_the_week_after => "weekday after".</p>

Date Generator Parser Construction

Syntax	Description
<pre>date_generator_parser()</pre>	Uses the default date generator elements.
<pre>date_generator_parser(...) Parameter: date_generator_parser</pre>	Copy Constructor
<pre>date_generator_parser(...) Parameters: string_type first_element string_type second_element string_type third_element string_type fourth_element string_type fifth_element string_type last_element string_type before_element string_type after_element string_type of_element</pre>	Constructs a date_generator_parser using the given element strings.

Date Generator Parser Accessors

Syntax	Description
	Example
<pre>void element_strings(...) Parameter: collection_type</pre>	<p>Replace the set of date generator element string with a new set.</p> <div></div>
<pre>void element_strings(...) Parameters: string_type first string_type second string_type third string_type fourth string_type fifth string_type last string_type before string_type after string_type of</pre>	<p>Replace the set of date generator elements with new values.</p> <div></div>
<pre>get_partial_date_type(...) Return Type: facet_type::partial_date_type Parameters: stream_itr_type next stream_itr_type str_end ios_base facet_type</pre>	<p>A templated function that parses a date_generator from the stream.</p> <p>Parses a partial_date => "dd Month".</p>
<pre>get_nth_kday_type(...) Return Type: facet_type::nth_kday_type Parameters: stream_itr_type next stream_itr_type str_end ios_base facet_type</pre>	<p>A templated function that parses a date_generator from the stream.</p> <p>Parses an nth_day_of_the_week_in_month => "nth weekday of month".</p>
<pre>get_first_kday_type(...) Return Type: facet_type::first_kday_type Parameters: stream_itr_type next stream_itr_type str_end ios_base facet_type</pre>	<p>A templated function that parses a date_generator from the stream.</p> <p>Parses a first_day_of_the_week_in_month => "first weekday of month".</p>

Syntax	Description
	Example
<pre>get_last_kday_type(...) Return Type: facet_type::last_kday_type Parameters: stream_itr_type next stream_itr_type str_end ios_base facet_type</pre>	<p>A templated function that parses a date_generator from the stream.</p> <p>Parses a last_day_of_the_week_in_month => "last weekday of month".</p>
<pre>get_kday_before_type(...) Return Type: facet_type::kday_before_type Parameters: stream_itr_type next stream_itr_type str_end ios_base facet_type</pre>	<p>A templated function that parses a date_generator from the stream.</p> <p>Parses a first_day_of_the_week_before => "weekday before"</p>
<pre>get_kday_after_type(...) Return Type: facet_type::kday_after_type Parameters: stream_itr_type next stream_itr_type str_end ios_base facet_type</pre>	<p>A templated function that parses a date_generator from the stream.</p> <p>Parses a first_day_of_the_week_after => "weekday after".</p>

Special Values

The date_time library uses five special_values. They are:

not_a_date_time neg_infin pos_infin min_date_time max_date_time

The default set of strings used to represent these types are: "not-a-date-time", "-infinity", "+infinity", "minimum-date-time", "maximum-date-time". When output, the min_date_time and max_date_time appear as normal date/time representations: "1400-Jan-01" and "9999-Dec-31" respectively.

Customization

The special values parser/formatter allows the user to set custom strings for these special values. These strings can be set as construction parameters to a new facet, or can be set in an existing facet via an accessor function.

Special Values Formatter/Parser Reference

The complete class references can be found here: [Special Values Formatter Doxygen Reference](#) and here: [Special Values Parser Doxygen Reference](#)

Special Values Formatter Constructor

Syntax	Description
<code>special_values_formatter()</code>	Constructor uses defaults for special value strings.
<pre>special_values_formatter(...) Parameters: collection_type::iterator collection_type::iterator</pre>	Constructs using values in collection. NOTE: Only the first three strings of the collection will be used. Strings for <code>minimum_date_time</code> and <code>maximum_date_time</code> are ignored as those special values are output as normal dates/times.
<pre>special_values_formatter(...) Parameters: char_type* char_type*</pre>	Constructs special values formatter from an array of strings.

Special Values Formatter Accessors

Syntax	Description
	Example
<pre>OutItrT put_special(...) Parameters: OutItrT next special_values value</pre>	<p>Puts the given special value into the stream.</p> <pre>date d1(not_a_date_time); date d2(minimum_date_time); special_values_formatter formatter; formatter.put_special(itr, d1); // Puts: "not-a-date-time" formatter.put_special(itr, d2); // Puts: "1400-Jan-01"</pre>

Special Values Parser Constructor

Syntax	Description
<code>special_values_parser()</code>	
<pre>special_values_parser(...) Parameters: collection_type::iterator collection_type::iterator</pre>	Constructs a special values parser using the strings in the collection.
<pre>special_values_parser(...) Parameter: special_values_parser</pre>	Copy constructor.
<pre>special_values_parser(...) Parameters: string_type nadt_str string_type neg_inf_str string_type pos_inf_str string_type min_dt_str string_type max_dt_str</pre>	Constructs a special values parser using the supplied strings.

Special Values Parser Accessors

Syntax	Description
<pre>void sv_strings(...) Parameters: string_type nadt_str string_type neg_inf_str string_type pos_inf_str string_type min_dt_str string_type max_dt_str</pre>	<p>Example</p> <p>Replace the set of special value strings with the given ones.</p> <div data-bbox="810 461 1481 495" style="border: 1px solid #ccc; height: 15px; width: 100%;"></div>
<pre>bool match(...) Parameters: stream_itr_type beg stream_itr_type end match_results</pre>	<p>Returns true if parse was successful. Upon a successful parse, <code>mr.current_match</code> will be set an int values corresponding to the equivalent special_value.</p> <div data-bbox="810 813 1481 1160" style="border: 1px solid #ccc; padding: 10px;"> <pre>// stream holds "maximum_date_time" typedef special_values_parser svp; svp parser; svp::match_results mr; if(parser.match(itr, str_end, mr)) { d = date(static_cast<special_values>(mr.match_results)) } else { // error, failed parse } // d == "9999-Dec-31"</pre> </div>

Format Date Parser

The format date parser is the object that holds the strings for months and weekday names, as well as their abbreviations. Custom sets of strings can be set at construction time, or, the strings in an existing `format_date_parser` can be replaced through accessor functions. Both the constructor and the accessor functions take a vector of strings as their arguments.

Format Date Parser Reference

The complete class reference can be found here: [Doxygen Reference](#)

Format Date Parser Constructor

Syntax	Description
<pre>format_date_parser(...) Parameters: string_type format std::locale</pre>	Creates a parser that uses the given format for parsing dates (in those functions where there is no format parameter). The names and abbreviations used are extracted from the given locale.
<pre>format_date_parser(...) Parameters: string_type format input_collection_type input_collection_type input_collection_type input_collection_type</pre>	Creates a parser from using the given components. The input_collection_type parameters are for: short month names, long month names, short weekday names, and long weekday names (in that order). These collections must contain values for every month and every weekday (beginning with January and Sunday).
<pre>format_date_parser(...) Parameters: format_date_parser</pre>	Copy Constructor

Format Date Parser Accessors

Syntax	Description
	Example
<pre>string_type format()</pre>	Returns the format that will be used when parsing dates in those functions where there is no format parameter. <div></div>
<pre>void format(string_type)</pre>	Sets the format that will be used when parsing dates in those functions where there is no format parameter. <div></div>
<pre>void short_month_names(...) Parameter: input_collection_type names</pre>	Replace the short month names used by the parser. The collection must contain values for each month, starting with January. <div></div>
<pre>void long_month_names(...) Parameter: input_collection_type names</pre>	Replace the long month names used by the parser. The collection must contain values for each month, starting with January. <div></div>
<pre>void short_weekday_names(...) Parameter: input_collection_type names</pre>	Replace the short weekday names used by the parser. The collection must contain values for each weekday, starting with Sunday. <div></div>
<pre>void long_weekday_names(...) Parameter: input_collection_type names</pre>	Replace the long weekday names used by the parser. The collection must contain values for each weekday, starting with Sunday. <div></div>
<pre>date_type parse_date(...) Parameters: string_type input string_type format special_values_parser</pre>	Parse a date from the given input using the given format. <div><pre>string inp("2005-Apr-15"); string format("%Y-%b-%d"); date d; d = parser.parse_date(inp, format, svp); // d == 2005-Apr-15</pre></div>
<pre>date_type parse_date(...) Parameters: istreambuf_iterator input istreambuf_iterator str_end special_values_parser</pre>	Parse a date from stream using the parser's format. <div></div>

Syntax	Description
<pre>date_type parse_date(...) Parameters: istreambuf_iterator input istreambuf_iterator str_end string_type format special_values_parser</pre>	<p>Example</p> <p>Parse a date from stream using the given format.</p> <pre>// stream holds "2005-04-15" string format("%Y-%m-%d"); date d; d = parser.parse_date(itr, str_end, format, svp); // d == 2005-Apr-15</pre>
<pre>month_type parse_month(...) Parameters: istreambuf_iterator input istreambuf_iterator str_end string_type format</pre>	<p>Parses a month from stream using given format. Throws <code>bad_month</code> if unable to parse.</p> <pre>// stream holds "March" string format("%B"); greg_month m; m = parser.parse_month(itr, str_end, format); // m == March</pre>
<pre>day_type parse_day_of_month(...) Parameters: istreambuf_iterator input istreambuf_iterator str_end</pre>	<p>Parses a <code>day_of_month</code> from stream. The day must appear as a two digit number (01-31), or a <code>bad_day_of_month</code> will be thrown.</p> <pre>// stream holds "01" greg_day d; d = parser.parse_day_of_month(itr, str_end); // d == 1st</pre>
<pre>day_type parse_var_day_of_month(...) Parameters: istreambuf_iterator input istreambuf_iterator str_end</pre>	<p>Parses a <code>day_of_month</code> from stream. The day must appear as a one or two digit number (1-31), or a <code>bad_day_of_month</code> will be thrown.</p> <pre>// stream holds "1" greg_day d; d = parser.parse_var_day_of_month(itr, str_end); // d == 1st</pre>

Syntax	Description
<pre> day_of_week_type parse_weekday(...) Parameters: istreambuf_iterator input istreambuf_iterator str_end string_type format </pre>	<p>Example</p> <p>Parse a weekday from stream according to the given format. Throws a bad_weekday if unable to parse.</p> <pre> // stream holds "Tue" string format("%a"); greg_weekday wd; wd = parser.parse_weekday(itr, str_end, format); // wd == Tuesday </pre>
<pre> year_type parse_year(...) Parameters: istreambuf_iterator input istreambuf_iterator str_end string_type format </pre>	<p>Parse a year from stream according to given format. Throws bad year if unable to parse.</p> <pre> // stream holds "98" string format("%y"); greg_year y; y = parser.parse_year(itr, str_end, format); // y == 1998 </pre>

Date Time IO Tutorial

Date Time IO Tutorial

[Basic Use](#) | [Format Strings](#) | [Content Strings](#) | [Special Values](#) | [Date/Time Periods](#) | [Date Generators](#)

Basic Use

Facets are automatically imbued when operators '>>' and '<<' are called. The list of date_time objects that can be streamed are:

Gregorian

date, days, date_period, greg_month, greg_weekday, greg_year, partial_date, nth_day_of_the_week_in_month, first_day_of_the_week_in_month, last_day_of_the_week_in_month, first_day_of_the_week_after, first_day_of_the_week_before

Posix_time

ptime, time_period, time_duration

Local_time

local_date_time

The following example is of the basic use of the new IO code, utilizing all the defaults. (this example can be found in the `libs/date_time/examples/tutorial` directory)

```

date d(2004, Feb, 29);
time_duration td(12,34,56,789);
stringstream ss;
ss << d << ' ' << td;
ptime pt(not_a_date_time);
cout << pt << endl; // "not-a-date-time"
ss >> pt;
cout << pt << endl; // "2004-Feb-29 12:34:56.000789"
ss.str("");
ss << pt << " EDT-05EDT,M4.1.0,M10.5.0";
local_date_time ldt(not_a_date_time);
ss >> ldt;
cout << ldt << endl; // "2004-Feb-29 12:34:56.000789 EDT"
└─

```

This example used the default settings for the input and output facets. The default formats are such that interoperability like that shown in the example is possible. NOTE: Input streaming of `local_date_time` can only be done with a [posix time zone string](#). The default output format uses a time zone abbreviation. The format can be changed so out and in match (as we will see later in this tutorial).

Format Strings

The format strings control the order, type, and style of the date/time elements used. The facets provide some predefined formats (`iso_format_specifier`, `iso_format_extended_specifier`, and `default_date_format`) but the user can easily create their own. (continued from previous example)

```

local_time_facet* output_facet = new local_time_facet();
local_time_input_facet* input_facet = new local_time_input_facet();
ss.imbue(locale(locale::classic(), output_facet));
ss.imbue(locale(ss.getloc(), input_facet));

output_facet->format("%a %b %d, %H:%M %z");
ss.str("");
ss << ldt;
cout << ss.str() << endl; // "Sun Feb 29, 12:34 EDT"

output_facet->format(local_time_facet::iso_time_format_specifier);
ss.str("");
ss << ldt;
cout << ss.str() << endl; // "20040229T123456.000789-0500"

output_facet->format(local_time_facet::iso_time_format_extended_specifier);
ss.str("");
ss << ldt;
cout << ss.str() << endl; // "2004-02-29 12:34:56.000789-05:00"
└─

```

Format strings are not limited to date/time elements. Extra verbiage can be placed in a format string. NOTE: When extra verbiage is present in an input format, the data being input must also contain the exact verbiage. (continued from previous example)

```
// extra words in format
string my_format("The extended ordinal time %Y-%jT%H:%M can also be \
represented as %A %B %d, %Y");
output_facet->format(my_format.c_str());
input_facet->format(my_format.c_str());
ss.str("");
ss << ldt;
cout << ss.str() << endl;

// matching extra words in input
ss.str("The extended ordinal time 2005-128T12:15 can also be \
represented as Sunday May 08, 2005");
ss >> ldt;
cout << ldt << endl;

┘
```

Content Strings

So far we've shown how a user can achieve a great deal of customization with very little effort by using formats. Further customization can be achieved through user defined elements (ie strings). The elements that can be customized are: Special value names, month names, month abbreviations, weekday names, weekday abbreviations, delimiters of the date/time periods, and the phrase elements of the `date_generators`.

The default values for these are as follows:

Special values

not-a-date-time, -infinity, +infinity, minimum-date-time, maximum-date-time

Months

English calendar and three letter abbreviations

Weekdays

English calendar and three letter abbreviations

Date generator phrase elements

first, second, third, fourth, fifth, last, before, after, of

NOTE: We've shown earlier that the components of a date/time representation can be re-ordered via the format string. This is not the case with `date_generators`. The elements themselves can be customized but their order cannot be changed.

Content Strings

To illustrate the customization possibilities we will use custom strings for months and weekdays (we will only use long names, is all lowercase, for this example).

(continued from previous example)

```

// set up the collections of custom strings.
// only the full names are altered for the sake of brevity
string month_names[12] = { "january", "february", "march",
                           "april", "may", "june",
                           "july", "august", "september",
                           "october", "november", "december" };
vector<string> long_months(&month_names[0], &month_names[12]);
string day_names[7] = { "sunday", "monday", "tuesday", "wednesday",
                        "thursday", "friday", "saturday" };
vector<string> long_days(&day_names[0], &day_names[7]);

// create date_facet and date_input_facet using all defaults
date_facet* date_output = new date_facet();
date_input_facet* date_input = new date_input_facet();
ss.imbue(locale(ss.getloc(), date_output));
ss.imbue(locale(ss.getloc(), date_input));

// replace names in the output facet
date_output->long_month_names(long_months);
date_output->long_weekday_names(long_days);

// replace names in the input facet
date_input->long_month_names(long_months);
date_input->long_weekday_names(long_days);

// customize month, weekday and date formats
date_output->format("%Y-%B-%d");
date_input->format("%Y-%B-%d");
date_output->month_format("%B"); // full name
date_input->month_format("%B"); // full name
date_output->weekday_format("%A"); // full name
date_input->weekday_format("%A"); // full name

ss.str("");
ss << greg_month(3);
cout << ss.str() << endl; // "march"
ss.str("");
ss << greg_weekday(3);
cout << ss.str() << endl; // "tuesday"
ss.str("");
ss << date(2005, Jul, 4);
cout << ss.str() << endl; // "2005-july-04"

┘

```

Special Values

Customizing the input and output of special values is best done by creating a new `special_values_parser` and `special_values_formatter`. The new strings can be set at construction time (as in the example below).

(continued from previous example)

```
// reset the formats to defaults
output_facet->format(local_time_facet::default_time_format);
input_facet->format(local_time_input_facet::default_time_input_format);

// create custom special_values parser and formatter objects
// and add them to the facets
string sv[5] = {"nadt", "neg_inf", "pos_inf", "min_dt", "max_dt" };
vector<string> sv_names(&sv[0], &sv[5]);
special_values_parser sv_parser(sv_names.begin(), sv_names.end());
special_values_formatter sv_formatter(sv_names.begin(), sv_names.end());
output_facet->special_values_formatter(sv_formatter);
input_facet->special_values_parser(sv_parser);

ss.str("");
ldt = local_date_time(not_a_date_time);
ss << ldt;
cout << ss.str() << endl; // "nadt"

ss.str("min_dt");
ss >> ldt;
ss.str("");
ss << ldt;
cout << ss.str() << endl; // "1400-Jan-01 00:00:00 UTC"

└
```

NOTE: even though we sent in strings for min and max to the formatter, they are ignored because those special values construct to actual dates (as shown above).

Date/Time Periods

Customizing the input and output of periods is best done by creating a new `period_parser` and `period_formatter`. The new strings can be set at construction time (as in the example below).

(continued from previous example)


```
// all formats set back to defaults (not shown for brevity)

// create our date_period
date_period dp(date(2005,Mar,1), days(31)); // month of march

// custom period formatter and parser
period_formatter per_formatter(period_formatter::AS_OPEN_RANGE,
                               " to ", "from ", " exclusive", " inclusive" );
period_parser per_parser(period_parser::AS_OPEN_RANGE,
                          " to ", "from ", " exclusive" , "inclusive" );

// default output
ss.str("");
ss << dp;
cout << ss.str() << endl; // "[2005-Mar-01/2005-Mar-31]"

// add out custom parser and formatter to the facets
date_output->period_formatter(per_formatter);
date_input->period_parser(per_parser);

// custom output
ss.str("");
ss << dp;
cout << ss.str() << endl; // "from 2005-Feb-01 to 2005-Apr-01 exclusive"

└
```

Date Generators

Customizing the input and output of date_generators is done by replacing the existing strings (in the facet) with new strings.

NOTE: We've shown earlier that the components of a date/time representation can be re-ordered via the format string. This is not the case with date_generators. The elements themselves can be customized but their order cannot be changed.

(continued from previous example)

```
// custom date_generator phrases
string dg_phrases[9] = { "1st", "2nd", "3rd", "4th", "5th",
                        "final", "prior to", "following", "in" };
vector<string> phrases(&dg_phrases[0], &dg_phrases[9]);

// create our date_generator
first_day_of_the_week_before d_gen(Monday);

// default output
ss.str("");
ss << d_gen;
cout << ss.str() << endl; // "Mon before"

// add our custom strings to the date facets
date_output->date_gen_phrase_strings(phrases);
date_input->date_gen_element_strings(phrases);

// custom output
ss.str("");
ss << d_gen;
cout << ss.str() << endl; // "Mon prior to"

└
```

Serialization

The `boost::date_time` library is compatible with the `boost::serialization` library's text and xml archives. The list of classes that are serializable are:

`boost::gregorian`

date	date_duration	date_period
partial_date	nth_day_of_week_in_month	first_day_of_week_in_month
last_day_of_week_in_month	first_day_of_week_before	first_day_of_week_after
greg_month	greg_day	greg_weekday

`boost::posix_time`

ptime	time_duration	time_period
-----------------------	-------------------------------	-----------------------------

No extra steps are required to build the `date_time` library for serialization use.

NOTE: due to a change in the serialization library interface, it is now required that all streamable objects be `const` prior to writing to the archive. The following template function will allow for this (and is used in the `date_time` tests). At this time no special steps are necessary to read from an archive.

```
template<class archive_type, class temporal_type>
void save_to(archive_type& ar, const temporal_type& tt)
{
    ar << tt;
}
↵
```

Example `text_archive` usage:

```
using namespace boost::posix_time;
using namespace boost::gregorian;
ptime pt(date(2002, Feb, 14)), hours(10), pt2(not_a_date_time);
std::ofstream ofs("tmp_file");
archive::test_oarchive oa(ofs);
save_to(oa, pt); // NOTE: no macro
ofs.close();
std::ifstream ifs("tmp_file");
archive::text_iarchive ia(ifs);
ia >> pt2; // NOTE: no macro
ifs.close();
pt == pt2; // true
```

Example `xml_archive` usage:

```
using namespace boost::gregorian;
date d(2002, Feb, 14), d2(not_a_date_time);
std::ofstream ofs("tmp_file");
archive::xml_oarchive oa(ofs);
save_to(oa, BOOST_SERIALIZATION_NVP(d)); // macro required for xml_archive
ofs.close();
std::ifstream ifs("tmp_file");
archive::xml_iarchive ia(ifs);
ia >> BOOST_SERIALIZATION_NVP(d2);      // macro required for xml_archive
ifs.close();
d == d2; // true
```

To use the date_time serialization code, the proper header files must be explicitly included. The header files are:

```
boost/date_time/gregorian/greg_serialize.hpp
```

and

```
boost/date_time/posix_time/time_serialize.hpp
```

Details

Calculations

[Timepoints](#) -- [Durations](#) -- [Intervals \(Periods\)](#) -- [Special Value Handling](#)

Timepoints

This section describes some of basic arithmetic rules that can be performed with timepoints. In general, Timepoints support basic arithmetic in conjunction with Durations as follows:

```
Timepoint + Duration --> Timepoint
Timepoint - Duration --> Timepoint
Timepoint - Timepoint --> Duration
└┐
```

Unlike regular numeric types, the following operations are undefined:

```
Duration + Timepoint --> Undefined
Duration - Timepoint --> Undefined
Timepoint + Timepoint --> Undefined
└┐
```

Durations

Durations represent a length of time and can have positive and negative values. It is frequently useful to be able to perform calculations with other durations and with simple integral values. The following describes these calculations:

```
Duration + Duration --> Duration
Duration - Duration --> Duration

Duration * Integer --> Duration
Integer * Duration --> Duration
Duration / Integer --> Duration (Integer Division rules)
└┐
```

Intervals (Periods)

Interval logic is extremely useful for simplifying many 'calculations' for dates and times. The following describes the operations provided by periods which are based on half-open range. The following operations calculate new time periods based on two input time periods:

```
Timeperiod intersection Timeperiod --> Timeperiod
  (null interval if no intersection)
Timeperiod merge Timeperiod --> Timeperiod
  (null interval if no intersection)
Timeperiod shift Duration --> Timeperiod
  (shift start and end by duration amount)
└┐
```

In addition, periods support various queries that calculate boolean results. The first set is calculations with other time periods:

```
Timeperiod == Timeperiod          --> bool
Timeperiod < Timeperiod           --> bool (true if lhs.last <= rhs.begin)
Timeperiod intersects Timeperiod  --> bool
Timeperiod contains Timeperiod    --> bool
Timeperiod is_adjacent Timeperiod --> bool
└
```

The following calculations are performed versus the Timepoint.

```
Timeperiod contains Timepoint      --> bool
Timeperiod is_before Timepoint     --> bool
Timeperiod is_after Timepoint      --> bool
└
```

Special Value Handling

For many temporal problems it is useful for Duration and Timepoint types to support special values such as Not A Date Time (NADT) and infinity. In general special values such as Not A Date Time (NADT) and infinity should follow rules like floating point values. Note that it should be possible to configure NADT based systems to throw an exception instead of result in NADT.

```
Timepoint(NADT) + Duration --> Timepoint(NADT)
Timepoint(∞) + Duration    --> Timepoint(∞)
Timepoint + Duration(∞)    --> Timepoint(∞)
Timepoint - Duration(∞)    --> Timepoint(-∞)
└
```

When performing operations on both positive and negative infinities, the library will produce results consistent with the following.

```
Timepoint(+∞) + Duration(-∞) --> NADT
Duration(+∞) + Duration(-∞)  --> NADT
Duration(±∞) * Zero          --> NADT

Duration(∞) * Integer(Not Zero) --> Duration(∞)
Duration(+∞) * -Integer         --> Duration(-∞)
Duration(∞) / Integer          --> Duration(∞)
└
```

Design Goals

Category	Description
	Functions
Interfaces	Provide concrete classes for manipulation of dates and times
	<ul style="list-style-type: none"> • date, time, date_duration, time_duration, date_period, time_period, etc • support for infinity - positive infinity, negative infinity • iterators over time and date ranges • allow date and time implementations to be separate as much as possible
Calculation	Provide a basis for performing efficient time calculations
	<ul style="list-style-type: none"> • days between dates • durations of times • durations of dates and times together
Representation Flexibility	Provide the maximum possible reusability and flexibility
	<ul style="list-style-type: none"> • traits based customization of internal representations for size versus resolution control • Allowing the use of different epochs and resolution (eg: seconds versus microseconds, dates starting at the year 2000 versus dates starting in 1700) • Options for configuring unique calendar representations (Gregorian + others) • the use of Julian Day number and the conversion between this and the Gregorian/Julian calendar date • Allow for flexible adjustments including leap seconds
Date Calculations	Provide tools for date calculations
	<ul style="list-style-type: none"> • provide basis for calculation of complex event specs like holidays • calendar to calendar conversions • provide for ability to extend to new calendar systems
Time Calculations	Provide concrete classes for manipulation of time
	<ul style="list-style-type: none"> • provide the ability to handle cross time-zone issues • provide adjustments for daylight savings time (summer time)

Category	Description
	Functions
Clock Interfaces	Provide classes for retrieving time current time
	<ul style="list-style-type: none">• access to a network / high resolution time sources• retrieving the current date time information to populate classes
I/O Interfaces	Provide input and output for time including
	<ul style="list-style-type: none">• multi-lingual support• provide ISO8601 compliant time facet• use I/O facets for different local behavior

Tradeoffs: Stability, Predictability, and Approximations

Unavoidable Trade-offs

The library does its best to provide everything a user could want, but there are certain inherent constraints that limit what *any* temporal library can do. Specifically, a user must choose which two of the following three capabilities are desired in any particular application:

- exact agreement with wall-clock time
- accurate math, e.g. duration calculations
- ability to handle timepoints in the future

Some libraries may implicitly promise to deliver all three, but if you actually put them to the test, only two can be true at once. This limitation is not a deficiency in the design or implementation of any particular library; rather it is a consequence of the way different time systems are defined by international standards. Let's look at each of the three cases:

If you want exact agreement with wall-clock time, you must use either UTC or local time. If you compute a duration by subtracting one UTC time from another and you want an answer accurate to the second, the two times must not be too far in the future because leap seconds affect the count but are only determined about 6 months in advance. With local times a future duration calculation could be off by an entire hour, since legislatures can and do change DST rules at will.

If you want to handle wall-clock times in the future, you won't be able (in the general case) to calculate exact durations, for the same reasons described above.

If you want accurate calculations with future times, you will have to use TAI or an equivalent, but the mapping from TAI to UTC or local time depends on leap seconds, so you will not have exact agreement with wall-clock time.

Stability, Predictability, and Approximations

Here is some underlying theory that helps to explain what's going on. Remember that a temporal type, like any abstract data type (ADT), is a set of values together with operations on those values.

Stability

The representation of a type is *stable* if the bit pattern associated with a given value does not change over time. A type with an unstable representation is unlikely to be of much use to anyone, so we will insist that any temporal library use only stable representations.

An operation on a type is stable if the result of applying the operation to a particular operand(s) does not change over time.

Predictability

Sets are most often classified into two categories: well-defined and ill-defined. Since a type is a set, we can extend these definitions to cover types. For any type T , there must be a predicate $is_member(x)$ which determines whether a value x is a member of type T . This predicate must return *true*, *false*, or *dont_know*.

If for all x , $is_member(x)$ returns either true or false, we say the set T is *well-defined*.

If for any x , $is_member(x)$ returns *dont_know*, we say the set T is *ill-defined*.

Those are the rules normally used in math. However, because of the special characteristics of temporal types, it is useful to refine this view and create a third category as follows:

For any temporal type T , there must be a predicate $is_member(x, t)$ which determines whether a value x is a member of T . The parameter t represents the time when the predicate is evaluated. For each x_i , there must be a time t_i and a value v such that:

- $v = \text{true}$ or $v = \text{false}$, and
- for all $t < t_i$, $is_member(x_i, t)$ returns *dont_know*, and
- for all $t \geq t_i$, $is_member(x_i, t)$ returns v .

t_i is thus the time when we "find out" whether x_i is a member of T . Now we can define three categories of temporal types:

If for all x_i , $t_i = \text{negative infinity}$, we say the type T is *predictable*.

If for some x_i , $t_i = \text{positive infinity}$, we say the type T is *ill-formed*.

Otherwise we say the type T is *unpredictable* (this implies that for some x_i , t_i is finite).

Ill-formed sets are not of much practical use, so we will not discuss them further. In plain english the above simply says that all the values of a predictable type are known ahead of time, but some values of an unpredictable type are not known until some particular time.

Stability of Operations

Predictable types have a couple of important properties:

- there is an order-preserving mapping from their elements onto a set of consecutive integers, and
- duration operations on their values are stable

The practical effect of this is that duration calculations can be implemented with simple integer subtraction. Examples of predictable types are TAI timepoints and Gregorian dates.

Unpredictable types have exactly the opposite properties:

- there is no order-preserving mapping from their elements onto a set of consecutive integers, and
- duration operations on their values are not stable.

Examples of unpredictable types are UTC timepoints and Local Time timepoints.

We can refine this a little by saying that a range within an unpredictable type can be predictable, and operations performed entirely on values within that range will be stable. For example, the range of UTC timepoints from 1970-01-01 through the present is predictable, so calculations of durations within that range will be stable.

Approximations

These limitations are problematical, because important temporal types like UTC and Local Time are in fact unpredictable, and therefore operations on them are sometimes unstable. Yet as a practical matter we often want to perform this kind of operation, such as computing the duration between two timepoints in the future that are specified in Local Time.

The best the library can do is to provide an approximation, which is generally possible and for most purposes will be good enough. Of course the documentation must specify when an answer will be approximate (and thus unstable) and how big the error may be. In many respects calculating with unpredictable sets is analogous to the use of floating point numbers, for which results are expected to only be approximately correct. Calculating with predictable sets would then be analogous to the user of integers, where results are expected to be exact.

For situations where exact answers are required or instability cannot be tolerated, the user must be able to specify this, and then the library should throw an exception if the user requests a computation for which an exact, stable answer is not possible.

Terminology

The following are a number of terms relevant to the date-time domain.

A taxonomy of temporal types:

- Timepoint -- Specifier for a location in the time continuum. Similar to a number on a ruler.
- Timelength -- A duration of time unattached to any point on the time continuum.
- Timeinterval -- A duration of time attached to a specific point in the time continuum.

And some other terms:

- Accuracy -- A measure of error, the difference between the reading of a clock and the true time.
- Calendar System -- A system for labeling time points with day level resolution.
- Clock Device -- A software component (tied to some hardware) that provides the current date or time with respect to a calendar or clock system.
- Precision -- A measure of repeatability of a clock.
- Resolution -- A specification of the smallest representable duration (eg: 1 second, 1 century) for a clock/calendar system or temporal type.
- Stability -- The property of a class which says that the underlying representation (implementation) associated with a particular (abstract) value will never change.
- Time System -- A system for labeling time points with higher resolution than day-level.

Some standard date-time terminology:

- Epoch -- Starting time point of a calendar or clock system.
- DST -- Daylight savings time - a local time adjustment made in some regions during the summer to shift the clock time of the daylight hours
- Time zone -- A region of the earth that provides for a 'local time' defined by DST rules and UT offset.

- UTC Time -- Coordinated Universal Time - Civil time system as measured at longitude zero. Kept adjusted to earth rotation by use of leap seconds. Also known as Zulu Time. Replaced the similar system known as Greenwich Mean Time. For more see <http://aa.usno.navy.mil/faq/docs/UT.html>
- TAI Time -- A high-accuracy monotonic (need better term) time system measured to .1 microsecond resolution by atomic clocks around the world. Not adjusted to earth rotation. For more see http://www.bipm.fr/enus/5_Scientific/c_time/time_server.html

Some more experimental ones:

- Local Time -- A time measured in a specific location of the universe.
- Time Label -- A tuple that either completely or partially specifies a specific date-time with respect to a calendar or clock system. This is the year-month-day representation.
- Adjusting Time Length -- A duration that represents varying physical durations depending on the moment in time. For example, a 1 month duration is typically not a fixed number of days and it depends on the date it is measured from to determine the actual length.

These are design sorts of terms:

- Generation function -- A function that generates a specific set of time points, lengths, or intervals based on one or more parameters.

References

The design of the library is currently being evolved using Wiki and email discussions. You can find more information at: [Boost Wiki GDTL Start Page](#).

- [Date References](#)
- [Time References](#)
- [Other C/C++ Libraries](#)
- [JAVA Date-Time Libraries](#)
- [Scripting Language Libraries](#)
- [Related Commercial and Fanciful Pages](#)
- [Resolution, Precision, and Accuracy](#)

Date Calendar References

- ISO 8601 date time standard -- [Summary by Markus Kuhn](#)
- [Calendrical Calculations](#) book by Reingold & Dershowitz
- [Calendar FAQ by Claus Tønndering](#)
- Calendar zone <http://www.calendarzone.com>
- [XML schema for date time](#)
- Will Linden's [Calendar Links](#)
- [XMAS calendar melt](#)

Time

- Martin Folwer on time patterns

- [Recurring Events for Calendars](#)
- Patterns for things that [Change with time](#)
- US National Institute of Standards and Technology [Time Exhibits](#)
- Network Time Protocol at [NTP.org](#)
- US Navy [Systems of Time](#)
- [International Atomic Time](#)
- [Date-Time type PostgreSQL](#) User Guide

Other C/C++ Libraries

- [ctime C](#) Standard library reference at [cplusplus.com](#)
- [XTime C extension](#) proposal
- [Another C library extension proposal](#) by David Tribble
- [libTAI](#) is a C based time library
- [Time Zone Database](#) C library for managing timezones/places
- International Components for Unicode by IBM (open source)
 - [Calendar Class](#)
 - [Date Time Services](#)
 - [Time Zone Class](#)
 - [Date-Time Formatting](#)
- [Julian Library in C](#) by Mark Showalter -- NASA

JAVA Date & Time Library Quick Reference

- [Calendar class](#)
- [Gregorian calendar](#)
- [Date class](#)
- [sql.time class](#)
- [Date format symbols](#)
- [Date format](#)
- [Simple Date Format](#)
- [Simple Time Zone](#)

Scripting Language Libraries

- A python date library [MX Date Time](#)
- Perl date-time

- [Date-Time packages at CPAN](#)
- [Date::Calc](#) at CPAN
- [Date::Convert](#) calendar conversions at CPAN

Related Commercial and Fanciful Pages

- [Eastern Standard Tribe](#) -- Cory Doctorow science fiction novel with time themes.
- [Leapsecond.com](#) time page
- [World Time Server / TZ database](#)
- [10000 year clock](#) at Long Now Foundation
- [Timezones for PCs](#)

Resolution, Precision, and Accuracy

- Definitions with pictures from [Agilent Technologies](#)
- Definitions from [Southampton Institute](#)

Build-Compiler Information

[Overview](#) -- [Compilation Options](#) -- [Compiler/Portability Notes](#) -- [Directory Structure](#) -- [Required Boost Libraries](#)

Overview

The library has a few functions that require the creation of a library file (mostly `to_string`, `from_string` functions). Most library users can make effective use of the library WITHOUT building the library, but simply including the required headers. If the library is needed, the Jamfile in the build directory will produce a "static" library (`libboost_date_time`) and a "dynamic/shared" library (`boost_date_time`) that contains these functions.

Compilation Options

By default the `posix_time` system uses a single 64 bit integer internally to provide a microsecond level resolution. As an alternative, a combination of a 64 bit integer and a 32 bit integer (96 bit resolution) can be used to provide nano-second level resolutions. The default implementation may provide better performance and more compact memory usage for many applications that do not require nano-second resolutions.

To use the alternate resolution (96 bit nanosecond) the variable `BOOST_DATE_TIME_POSIX_TIME_STD_CONFIG` must be defined in the library users project files (ie Makefile, Jamfile, etc). This macro is not used by the Gregorian system and therefore has no effect when building the library.

As of version 1.33, the `date_time` library introduced a new IO streaming system. Some compilers are not capable of utilizing this new system. For those compilers the earlier ("legacy") IO system is still available. Non-supported compilers will select the legacy system automatically but the user can force the usage of the legacy system by defining `USE_DATE_TIME_PRE_1_33_FACET_IO`.

As a convenience, `date_time` has provided some [additional duration types](#). Use of these types may have unexpected results due to the snap-to-end-of-month behavior (see [Reversibility of Operations Pitfall](#) for complete details and examples). These types are enabled by default. To disable these types, simply undefine `BOOST_DATE_TIME_OPTIONAL_GREGORIAN_TYPES` in your project file.

Another convenience is the default constructors for [date](#), and [ptime](#). These constructors are enabled by default. To disable them, simply define `DATE_TIME_NO_DEFAULT_CONSTRUCTOR` in your project file.

Compiler/Portability Notes

The Boost Date-Time library has been built and tested with many compilers and platforms. However, some compilers and standard libraries have issues. While some of these issues can be worked around, others are difficult to work around. The following compilers are known to fully support all aspects of the library:

- Codewarrior 9.4 Windows
- GCC 3.2 - 3.4, 4.x on Linux
- GCC 3.3, 4.x on Darwin
- GCC 3.3 - 3.4, 4.x on Solaris
- GCC 3.3, 4.x on HP-UX
- QCC 3.3.5 on QNX
- MSVC 7.1 Windows
- Intel 8.1-9.x Linux and Windows

Unfortunately, the VC8 compiler has some issues with date-time code. The most serious issue is a memory leak which was introduced into the VC8 standard library `basic_stream` code. Date-time code has been changed to avoid this as much as possible, but if you are using the legacy IO option (NOT the default with VC8) then the issue can still arise. See the [mailing list archive](#) for more details.

In addition to the problem above, some versions of the VC8 library have limited the range of allowed values in the `std::tm` structure to positive values. This was a new restriction added in the VC8. The effect is that dates prior to the year 1900 will cause exceptions. There is, unfortunately, no easy workaround for this issue. Note that the new 64bit version of the VC8 compiler does not appear to have this limitation.

These compilers support all aspects of the library except `wstring/wstream` output.

- MinGW 3.2, 3.4, 3.5 *
- GCC 3.2 (cygwin) *

In particular, a lack of support for standard locales limits the ability of the library to support `iostream` based input output. For these compilers a set of more limited string based input-output is provided. Some compilers/standard libraries with this limitation include:

- Borland 5.6

Official support for some older compilers has now been dropped. This includes:

- GCC 2.9x
- Borland 5.1.1
- MSVC 7.0 and 6 SP5

Visual Studio & STLPort

There is a known issue with Visual Studio (7.0 & 7.1) and STLPort. The build errors typically make reference to a type issue or 'no acceptable conversion' and are attempting to instantiate a template with `wchar_t`. The default build of STLPort does not support `wchar_t`. There are two possible workarounds for this issue. The simplest is the user can build `date_time` with no wide stream/string etc. The other is to rebuild STLPort with `wchar_t` support.

To build `date_time` with no wide stream/string etc, execute the following command from `$BOOST_ROOT`:

```
bjam -a "-sTOOLS=vc-7_1-stlport" "-sSTLPORT_PATH=..." \  
      "-sBUILD=<define>BOOST_NO_STD_WSTRING"              \  
      --stagedir=... --with-date_time stage
```

(replace the ellipsis with the correct paths for the build system and adjust the `TOOLS` to the proper toolset if necessary)

Rebuilding STLPort with `wchar_t` support involves placing `/Zc:wchar_t` in the STLPort makefile. Date_time should then be built with the following command from `$BOOST_ROOT`:

```
bjam -a "-sTOOLS=vc-7_1-stlport" "-sSTLPORT_PATH=..." \  
      "-sBUILD=&native-wchar_t>on"              \  
      --stagedir=... --with-date_time stage
```

(replace the ellipsis with the correct paths for the build system and adjust the `TOOLS` to the proper toolset if necessary)

Directory Structure

The directory tree has the following structure:

<code>/boost/date_time</code>	-- common headers and template code
<code>/boost/date_time/gregorian</code>	-- Gregorian date system header files
<code>/boost/date_time/posix_time</code>	-- Posix time system headers
<code>/boost/date_time/local_time</code>	-- Local time system headers
<code>/libs/date_time/build</code>	-- build files and output directory
<code>/libs/date_time/test</code>	-- test battery for generic code
<code>/libs/date_time/test/gregorian</code>	-- test battery for the Gregorian system
<code>/libs/date_time/test/posix_time</code>	-- test battery for the posix_time system
<code>/libs/date_time/test/local_time</code>	-- test battery for the local_time system
<code>/libs/date_time/examples/gregorian</code>	-- example programs for dates
<code>/libs/date_time/examples/posix_time</code>	-- time example programs
<code>/libs/date_time/examples/local_time</code>	-- nifty example programs
<code>/libs/date_time/src/gregorian</code>	-- cpp files for libboost_date_time
<code>/libs/date_time/src/posix_time</code>	-- empty (one file, but no source code...)

Required Boost Libraries

Various parts of date-time depend on other boost libraries. These include:

- [boost.tokenizer](#)
- [boost.integer\(cstdint\)](#)
- [boost.operators](#)
- [boost.lexical_cast](#)
- [boost.smart_ptr](#) (local time only)
- [boost::string_algorithms](#)
- [boost::serialize](#) (serialization code only)

so these libraries need to be installed.

Tests

The library provides a large number of tests in the


```
libs/date_time/test
libs/date_time/test/gregorian
libs/date_time/test/posix_time
libs/date_time/test/local_time
└─
```

directories. Building and executing these tests assures that the installation is correct and that the library is functioning correctly. In addition, these tests facilitate the porting to new compilers. Finally, the tests provide examples of many functions not explicitly described in the usage examples.

Change History

Changes from Boost 1.41 to 1.44 (date_time 1.08 to 1.09)

Type	Description
Bug fix	The "%T" and "%R" format specifiers are now processed by the library rather than underlying standard facet. This fixes the cases when the placeholders are not supported by the facet (#3876).

Changes from Boost 1.40 to 1.41 (date_time 1.07 to 1.08)

Type	Description
Change	The default format for time durations is now "%- %O:%M:%S%F" instead of "%- %H:%M:%S%F" that was used previously. In order to retain the old behavior, the format string has to be specified explicitly during the time IO facet construction (#1861).
Bug fix	Gregorian dates now use 32-bit integer type internally on 64-bit platforms (#3308).
Bug fix	Adjusted UTC time zone offset boundaries in order to allow offsets up to +14 hours (#2213).

Changes from Boost 1.38 to 1.40 (date_time 1.06 to 1.07)

Type	Description
Bug fix	Minor bug fixes (#2809 , #2824 , #3015 , #3105 and others).

Changes from Boost 1.34 to 1.38 (date_time 1.05 to 1.06)

Type	Description
Feature	Added support for formatting and reading time durations longer than 24 hours. A new formatter %O is used indicate such long durations in the format string. The old %H format specifier is thus restricted to represent durations that fit into two characters, in order to retain support for reading durations in ISO format. In case if it is detected that the %H format specifier is used with longer durations, the results are not specified (an assertion in debug builds is raised).
Bug fix	Added support for GCC 4.3. Several compilation issues were solved, as well as compiler warnings were taken care of.
Bug fix	Added missing streaming operators for the <code>local_time_period</code> class.
Bug fix	Added several missing includes in different places. Some includes that are not needed in some configurations were made conditional.
Bug fix	Solved compilation problem that was caused by not finding streaming operators for <code>gregorian::date_duration</code> via ADL. The type is now made actually a class rather a typedef for the <code>date_time::date_duration</code> template. The similar change was done for <code>gregorian::weeks</code> .
Bug fix	Added a correctly spelled <code>date_time::hundredth</code> time resolution enum value. The old one <code>date_time::hundreth</code> is considered deprecated and to be removed in future releases.
Bug fix	Fixed compilation error in <code>format_date_parser.hpp</code> because of incorrect stream type being used.
Bug fix	On Windows platform made inclusion of <code>windows.h</code> optional. The header is only used when the <code>BOOST_USE_WINDOWS_H</code> macro is defined. Otherwise (and by default), the library uses internal definitions of symbols from this header.
Bug fix	On Windows platform function <code>from_fptime</code> could return incorrect time if the <code>FILETIME</code> that is being passed to the function contained dates before 1970-Jan-01.
Bug fix	Fixed a possible crash in <code>gregorian::special_value_from_string</code> if the string did not represent a valid special value.
Bug fix	Removed the <code>testfrmwk.hpp</code> file from the public include directory. This file was internal for the library tests and was not documented.
Bug fix	Fixed missing include in <code>filetime_functions.hpp</code> (#2688).
Bug fix	Fixed dereferencing end string iterators in different places of code, which could cause crashes on MSVC (#2698).

Changes from Boost 1.33 to 1.34 (date_time 1.04 to 1.05)

Type	Description
Feature	Updated the data in the <code>date_time_zonespec.csv</code> file to reflect new US/Canada daylight savings time rules for 2007. If you upgrade to the new file, be aware that the library will only give correct answers for current/future date conversions. So if you are converting dates from earlier years the answers will reflect current time zone rules not past rules. The library doesn't support historic timezone rules presently.
Feature	Two other dst calculation features have also been update to reflect the new US/Canada timzone rules. This is the <code>boost::date_time::us_dst_rules</code> and <code>dst_calc_engine</code> . While the <code>us_dst_rules</code> is officially deprecated, a patch by Graham Bennett has been applied which allows this class to work correctly for both historical and future dates. The <code>dst_calc_engine</code> was updated to also work for historical and future times. This allows the various <code>local_adjustor</code> classes to work correctly. There was an interface change for classes using the <code>dst_calc_engine</code> with custom dst traits classes. The traits classes signatures changed to take a 'year' parameter on most of the methods such as <code>end_month</code> . In addition, 2 new functions are needed on the traits classes: <code>static date_type local_dst_start_day(year_type year)</code> and <code>static date_type local_dst_end_day(year_type year)</code> . Implementers should see <code>date_time/local_timezone_defs.hpp</code> for examples.
Bug Fix	Fix DST traits for Australia (sf# 1672139) to set end of DST at 3:00 am instead of 2:00 am.
Bug Fix	Fix a problem with potential linking error with multiple definitions due to I/O code.
Bug Fix	Changed serialization code in both <code>greg_serialize.hpp</code> and <code>time_serialize.hpp</code> to eliminate warnings due to unused variables for <code>version</code> and <code>file_version</code> . Thanks to Caleb Epstein for the patch suggestion.
Bug Fix	Fix regression errors that showed up under FreeBSD with GCC and the <code>LANG</code> environment set to <code>russian</code> -- changed parser to use classic locale instead of blank locale.
Bug Fix	Changes for tracker issue 1178092 -- change in <code>convert_to_lower</code> to make local a const static and speed up parsing.
Bug Fix	Patches from Ulrich Eckhardt to fix support for EVC++ 4.
Feature	Reduce the usage of <code>basic_stringstream</code> as much a possible to work around a bug in the VC8 standard library. See mailing list archive for more information.

Changes from Boost 1.32 to 1.33 (date_time 1.03 to 1.04)

Type	Description
Bug Fix	Period lengths, when beginning and end points are the same, or are consecutive, were being incorrectly calculated. The corrected behavior, where end and beginning points are equal, or a period is created with a zero duration, now return a length of zero. A period where beginning and end points are consecutive will return a length of one.
Bug Fix	Time_input_facet was missing functions to set iso formats. It also failed to parse time values that did not use a separator (%H%M%S). Both these bugs have been corrected.
Feature	Preliminary names of ptime_facet and ptime_input_facet changed to simply time_facet and time_input_facet. The ptime_* versions have been removed all together.
Feature	The from_iso_string function failed to parse fractional digits. We added code that correctly parses when input has more digits, or too few digits, that the compiled library precision. Ptimes with only a decimal are also correctly parsed.
Bug Fix	The parsing mechanism in the new IO would consume the next character after a match was made. This bug presented itself when attempting to parse a period that had special value for it's beginning point.
Bug Fix	The new IO system failed to provide the ability for the user to "turn on" exceptions on the stream. The failbit was also not set when parsing failed. Both of these problems have been fixed.
Bug Fix	Parsing of special values, by means of from_*_string functions, has been fixed. This also effects the libraries ability to serialize special values. Time_duration now serializes as either a string or individual fields (depending on is_special()).
Bug Fix	Previously, output streaming of partial_date would display the day as either a single or double digit integer (ie '1', or '12'). This has been corrected to always display a double digit integer (ie '01').
Feature	Major new features related to management of local times. This includes the introduction of a series of new classes to represent time zones and local times (see Date Time Local Time for complete details).
Feature	Input and output facets have been re-written to support format-based redefinition of formats (see Date Time IO for complete details).
Feature	Functions have been added to facilitate conversions between tm structs for date, ptime, time_duration, and local_date_time. Functions for converting FILETIME, and time_t to ptime are also provided. See the individual sections for details.

Type	Description
Feature	A <code>universal_time</code> function has been added to the <code>micro-sec_time_clock</code> (full details of this function can be found here).
Feature	Functions have been added to facilitate conversions between <code>tm</code> structs for <code>date</code> , <code>ptime</code> , <code>time_duration</code> , and <code>local_date_time</code> . Functions for converting <code>FILETIME</code> , and <code>time_t</code> to <code>ptime</code> are also provided. See the individual sections for details.
Feature	A <code>universal_time</code> function has been added to the <code>micro-sec_time_clock</code> (full details of this function can be found here).
Feature	Date-time now uses reentrant POSIX functions on those platforms that support them when <code>BOOST_HAS_THREADS</code> is defined.
Bug Fix	Fixed a bug in serialization code where special values (not-a-date-time, infinities, etc) for <code>ptime</code> , <code>time_duration</code> would not read back correctly from an archive. The output serialization code wrote subfields such as <code>time_duration.seconds()</code> which are invalid for special values and thus undefined values. Thus when read back the values could cause strange behavior including exceptions on construction.
Bug Fix	Fixed multiple warnings generated with various platforms/compilers.
Bug Fix	Construction of a <code>ptime</code> with a <code>time_duration</code> beyond the range of 00:00 to 23:59:59.9... now adjusts the date and time to make the <code>time_duration</code> fall within this range (ie <code>ptime(date(2005,2,1), hours(-5))</code> -> "2005-Jan-31 19:00:00" & <code>ptime(date(2005,2,1), hours(35))</code> -> "2005-Feb-02 11:00:00").
Bug Fix	Time parsing now correctly handles excessive digits for fractional seconds. Leading zeros are dropped ("000100" -> 100 <code>frac_sec</code>), and excessive digits are truncated at the proper place ("123456789876" -> 123456 or 123456789 depending on what precision the library was compiled with).
Bug Fix	<p>Changes to the <code>boost::serialization</code> interface broke serialization compatibility for <code>date_time</code>. The user must provide a function to insure <code>date_time</code> objects are <code>const</code> before they are serialized. The function should be similar to:</p> <pre> template<class archive_type, class temporal_type> void save_to(archive_type& ar, const temporal_type& tt) { ar << tt; } </pre>

Type	Description
Bug Fix	Use of the deprecated <code>boost::tokenizer</code> interface has been updated to the current interface. This fixes compiler errors on some older compilers.
Bug Fix	Templatized formatters in the legacy IO system to accept char type. Also removed calls to <code>boost::lexical_cast</code> .

Changes from Boost 1.31 to 1.32 (date_time 1.02 to 1.03)

Type	Description																		
Bug Fix	Snap to end of month behavior corrected for year_functor. Previously, starting from 2000-Feb-28 (leap year and not end of month) and iterating through the next leap year would result in 2004-Feb-29 instead of 2004-Feb-28. This behavior has been corrected to produce the correct result of 2004-Feb-28. Thanks to Bart Garst for this change.																		
Feature	Free function for creating a ptime object from a FILETIME struct. This function is only available on platforms that define BOOST_HAS_FTIME.																		
Feature	Microsecond time clock is now available on most windows compilers as well as Unix.																		
Feature	Use of the boost::serialization library is now available with most of the date_time classes. Classes capable of serialization are: date_generator classes, date, days, date_period, greg_month, greg_weekday, greg_day, ptime, time_duration, and time_period. Thanks to Bart Garst for this change.																		
Feature	Functions added to convert date and time classes to wstring. The library now provides to_*_wstring as well as to_*_string functions for: simple, iso, iso_extended, and sql for dates and compilers that support wstrings. Thanks to Bart Garst for this change.																		
Feature	Period classes now handle zero length and NULL periods correctly. A NULL period is a period with a negative length. Thanks to Frank Wolf and Bart Garst for this change.																		
Feature	Added end_of_month function to gregorian::date to return the last day of the current month represented by the date. Result is undefined for not_a_date_time or infinities.																		
Bug Fix	Removed incorrect usage of BOOST_NO_CWCHAR macro throughout library.																		
Feature	<div>New names added for some date classes. Original names are still valid but may some day be deprecated. Changes are:</div> <table><tr><td>date_duration</td><td>is now</td><td>days</td></tr><tr><td>nth_kday_of_month</td><td>is now</td><td>nth day of the week in month</td></tr><tr><td>first_kday_of_month</td><td>is now</td><td>first day of the week in month</td></tr><tr><td>last_kday_of_month</td><td>is now</td><td>last day of the week in month</td></tr><tr><td>first_kday_after</td><td>is now</td><td>first day of the week after</td></tr><tr><td>first_kday_before</td><td>is now</td><td>first day of the week before</td></tr></table>	date_duration	is now	days	nth_kday_of_month	is now	nth day of the week in month	first_kday_of_month	is now	first day of the week in month	last_kday_of_month	is now	last day of the week in month	first_kday_after	is now	first day of the week after	first_kday_before	is now	first day of the week before
date_duration	is now	days																	
nth_kday_of_month	is now	nth day of the week in month																	
first_kday_of_month	is now	first day of the week in month																	
last_kday_of_month	is now	last day of the week in month																	
first_kday_after	is now	first day of the week after																	
first_kday_before	is now	first day of the week before																	

Type	Description
Feature	<p>Free functions for date generators added. Functions are: <code>days_until_weekday</code>, <code>days_before_weekday</code>, <code>next_weekday</code>, and <code>previous_weekday</code>.</p> <pre>days days_until_weekday(date, greg_weekday); days days_before_weekday(date, greg_weekday); date next_weekday(date, greg_weekday); date previous_weekday(date, greg_weekday);</pre> <p>Thanks to Bart Garst for this change.</p>
Feature	<p>New experimental duration types added for months, years, and weeks. These classes also provide mathematical operators for use with date and time classes. Be aware that adding of months or years a time or date past the 28th of a month may show non-normal mathematical properties. This is a result of 'end-of-month' snapping used in the calculation. The last example below illustrates the issue.</p> <pre>months m(12); years y(1); m == y; // true days d(7); weeks w(1); d == w; // true ptime t(...); t += months(3); date d(2004, Jan, 30); d += months(1); // 2004-Feb-29 d -= months(1); // 2004-Jan-29</pre> <p>Input streaming is not yet implemented for these types. Thanks to Bart Garst for this change.</p>
Feature	<p>Unifying base class for date_generators brought in to gregorian namespace. See Print Holidays Example.</p>
Feature	<p>Added constructors for date and ptime that allow for default construction (both) and special values construction (ptime, both now support this). Default constructors initialize the objects to <code>not_a_date_time</code> (NADT).</p> <pre>ptime p_nadt(not_a_date_time); ptime p_posinf(pos_infin); ptime p; // p == NADT date d; // d == NADT</pre> <p>Thanks to Bart Garst for this change.</p>

Type	Description
Feature	<p>Output streaming now supports wide stream output on compiler / standard library combinations that support wide streams. This allows code like:</p> <pre>std::wstringstream wss; date d(2003,Aug,21); wss << d;</pre> <p>Thanks to Bart Garst for this change.</p>
Feature	<p>Input streaming for date and time types is now supported on both wide and narrow streams:</p> <pre>date d(not_a_date_time); std::stringstream ss("2000-FEB-29"); ss >> d; //Feb 29th, 2000 std::wstringstream ws("2000-FEB-29"); ws >> d; //Feb 29th, 2000</pre> <p>Thanks to Bart Garst for this change.</p>
Bug Fix	<p>Fixed bug in duration_from_string() where a string formatted with less than full amount of fractional digits created an incorrect time_duration. With microsecond resolution for time durations the string "1:01:01.010" created a time duration of 01:01:01.000010 instead of 01:01:01.010000</p>
Bug Fix	<p>Fixed the special value constructor for gregorian::date and posix_time::ptime when constructing with min_date_time or max_date_time. The wrong value was constructed for these.</p>

Changes from Boost 1.30 to 1.31 (date_time 1.01 to 1.02)

Type	Description
Bug Fix	Build configuration updated so dll, statically, and dynamically linkable library files are now produced with MSVC compilers. See Build/Compiler Information for more details.
Bug Fix	Time_duration from_string is now correctly constructed from a negative value. (ie "-0:39:00.000") Code provided by Bart Garst.
Bug Fix	Fixed many MSVC compiler warnings when compiled with warning level 4.
Feature	Added prefix decrement operator (-- for date and time iterators. See Time Iterators and Date Iterators for more details. Code provided by Bart Garst.
Feature	Special_values functionality added for date_duration, time_duration and time classes. Code provided by Bart Garst.
Bug Fix	Fixed time_duration_traits calculation bug which was causing time duration to be limited to 32bit range even when 64 bits were available. Thanks to Joe de Guzman for tracking this down.
Bug Fix	Provided additional operators for duration types (eg: date_duration, time_duration). This includes dividable by integer and fixes to allow +=, -= operators. Thanks to Bart Garst for writing this code. Also, the documentation of Calculations has been improved.
Bug Fix	Added typedefs to boost::gregorian gregorian_types.hpp various date_generator function classes.
Feature	Added from_time_t function to convert time_t to a ptime.
Feature	Added a span function for combining periods. See date period and time period docs.
Feature	<p>Added a function to time_duration to get the total number of seconds in a duration truncating any fractional seconds. In addition, other resolutions were added to allow for easy conversions. For example</p> <pre>seconds(1).total_milliseconds() == 1000 seconds(1).total_microseconds() == 1000000 hours(1).total_milliseconds() == 3600*1000 ↵ //3600 sec/hour seconds(1).total_nanoseconds() == 1000000000</pre>
Feature	Added output streaming operators for the date generator classes - partial_date, first_kday_after, first_kday_before, etc. Thanks to Bart Garst for this work.

Type	Description
Feature	<p>Added unary- operators for durations for reversing the sign of a time duration. For example:</p> <pre>time_duration td(5,0,0); //5 hours td = -td; //-5 hours</pre> <p>Thanks to Bart Garst for this work.</p>
Feature	<p>Added support for parsing strings with 'month names'. Thus creating a date object from string now accepts multiple formats ("2003-10-31", "2003-Oct-31", and "2003-October-31"). Thus, date d = from_simple_string("2003-Feb-27") is now allowed. A bad month name string (from_simple_string("2003-Some-BogusMonthName-27")) will cause a bad_month exception. On most compilers the string compare is case insensitive. Thanks to Bart Garst for this work.</p>
Feature	<p>In addition to support for month names or numbers, functions have been added to create date objects from multi-ordered date strings. Ex: "January-21-2002", "2002-Jan-21", and "21-Jan-2003". See Date Class for more details.</p>
Bug-Fix	<p>Various documentation fixes. Thanks to Bart Garst for updates.</p>

Changes from Boost 1.29 to 1.30 (date_time 1.00 to 1.01)

Notice: The interface to the `partial_date` class (see [date_algorithms](#)) was changed. The order of construction parameters was changed which will cause some code to fail execution. This change was made to facilitate more generic local time adjustment code. Thus instead of specifying `partial_date pd(Dec,25)` the code needs to be changed to `partial_date pd(25, Dec)`;

Type	Description
Bug Fix	Added new experimental feature for Daylight Savings Time calculations. This allows traits based specification of dst rules.
Feature	Added new interfaces to calculate julian day and modified julian day to the gregorian date class. See boost::gregorian::date .
Feature	Add new interface to calculate iso 8601 week number for a date. See boost::gregorian::date .
Feature	Add an iso 8601 time date-time format (eg: YYYYMMDDTH-HHMMSS) parsing function. See Class ptime for more information.
Feature	Added a length function to the period template so that both date_periods and time_periods will now support this function.
Bug Fix	Split Jamfiles so that libs/date_time/build/Jamfile only builds library and /libs/date_time/libs/test/Jamfile which runs tests.
Bug Fix	Fixed many minor documentation issues.
Bug Fix	Removed the DATE_TIME_INLINE macro which was causing link errors. This macro is no longer needed in projects using the library.
Bug Fix	Added missing typedef for year_iterator to gregorian_types.hpp
Bug Fix	Fixed problem with gregorian ostream operators that prevented the use of wide streams.
Bug-Fix	Tighten error handling for dates so that date(2002, 2, 29) will throw a bad_day_of_month exception. Previously the date would be incorrectly constructed. Reported by sourceforge bug: 628054 among others.

Acknowledgements

Many people have contributed to the development of this library. In particular Hugo Duncan and Joel de Guzman for help with porting to various compilers. For initial development of concepts and design Corwin Joy and Michael Kenniston deserve special thanks. Also extra thanks to Michael for writing up the theory and tradeoffs part of the documentation. Dave Zumbro for initial inspiration and sage thoughts. Many thanks to boost reviewers and users including: William Seymour, Kjell Elster, Beman Dawes, Gary Powell, Andrew Maclean, William Kempf, Peter Dimov, Chris Little, David Moore, Darin Adler, Gennadiy Rozental, Joachim Achtzehnter, Paul Bristow, Jan Langer, Mark Rodgers, Glen Knowles, Matthew Denman, and George Heintzelman.

Examples

Dates as Strings

Various parsing and output of strings.

```

/* The following is a simple example that shows conversion of dates
 * to and from a std::string.
 *
 * Expected output:
 * 2001-Oct-09
 * 2001-10-09
 * Tuesday October 9, 2001
 * An expected exception is next:
 * Exception: Month number is out of range 1..12
 */

#include "boost/date_time/gregorian/gregorian.hpp"
#include <iostream>
#include <string>

int
main()
{
    using namespace boost::gregorian;

    try {
        // The following date is in ISO 8601 extended format (CCYY-MM-DD)
        std::string s("2001-10-9"); //2001-October-09
        date d(from_simple_string(s));
        std::cout << to_simple_string(d) << std::endl;

        //Read ISO Standard(CCYYMMDD) and output ISO Extended
        std::string ud("20011009"); //2001-Oct-09
        date dl(from_undelimited_string(ud));
        std::cout << to_iso_extended_string(dl) << std::endl;

        //Output the parts of the date - Tuesday October 9, 2001
        date::ymd_type ymd = dl.year_month_day();
        greg_weekday wd = dl.day_of_week();
        std::cout << wd.as_long_string() << " "
                  << ymd.month.as_long_string() << " "
                  << ymd.day << ", " << ymd.year
                  << std::endl;

        //Let's send in month 25 by accident and create an exception
        std::string bad_date("20012509"); //2001-??-09
        std::cout << "An expected exception is next: " << std::endl;
        date wont_construct(from_undelimited_string(bad_date));
        //use wont_construct so compiler doesn't complain, but you wont get here!
        std::cout << "oh oh, you shouldn't reach this line: "
                  << to_iso_string(wont_construct) << std::endl;
    }
    catch(std::exception& e) {
        std::cout << " Exception: " << e.what() << std::endl;
    }
}

```



```
    return 0;
}
```

```
└─
```

Days Alive

Calculate the number of days you have been living using durations and dates.

```
/* Short example that calculates the number of days since user was born.
 * Demonstrates comparisons of durations, use of the day_clock,
 * and parsing a date from a string.
 */

#include "boost/date_time/gregorian/gregorian.hpp"
#include <iostream>

int
main()
{
    using namespace boost::gregorian;
    std::string s;
    std::cout << "Enter birth day YYYY-MM-DD (eg: 2002-02-01): ";
    std::cin >> s;
    try {
        date birthday(from_simple_string(s));
        date today = day_clock::local_day();
        days days_alive = today - birthday;
        days one_day(1);
        if (days_alive == one_day) {
            std::cout << "Born yesterday, very funny" << std::endl;
        }
        else if (days_alive < days(0)) {
            std::cout << "Not born yet, hmm: " << days_alive.days()
                << " days" << std::endl;
        }
        else {
            std::cout << "Days alive: " << days_alive.days() << std::endl;
        }
    }
    catch(...) {
        std::cout << "Bad date entered: " << s << std::endl;
    }
    return 0;
}
```

Days Between New Years

Calculate the number of days till new years

```
/* Provides a simple example of using a date_generator, and simple
 * mathematical operations, to calculate the days since
 * New Years day of this year, and days until next New Years day.
 *
 * Expected results:
 * Adding together both durations will produce 366 (365 in a leap year).
 */
#include <iostream>
#include "boost/date_time/gregorian/gregorian.hpp"

int
main()
{
    using namespace boost::gregorian;

    date today = day_clock::local_day();
    partial_date new_years_day(1,Jan);
    //Subtract two dates to get a duration
    days days_since_year_start = today - new_years_day.get_date(today.year());
    std::cout << "Days since Jan 1: " << days_since_year_start.days()
               << std::endl;

    days days_until_year_start = new_years_day.get_date(today.year()+1) - today;
    std::cout << "Days until next Jan 1: " << days_until_year_start.days()
               << std::endl;
    return 0;
};

└
```

Last Day of the Months

Example that gets a month and a year from the user and finds the last day of each remaining month of that year.

```
/* Simple program that finds the last day of the given month,
 * then displays the last day of every month left in the given year.
 */

#include "boost/date_time/gregorian/gregorian.hpp"
#include <iostream>

int
main()
{
    using namespace boost::gregorian;

    greg_year year(1400);
    greg_month month(1);

    // get a month and a year from the user
    try {
        int y, m;
        std::cout << "    Enter Year(ex: 2002): ";
        std::cin >> y;
        year = greg_year(y);
        std::cout << "    Enter Month(1..12): ";
        std::cin >> m;
        month = greg_month(m);
    }
    catch(bad_year by) {
        std::cout << "Invalid Year Entered: " << by.what() << '\n'
            << "Using minimum values for month and year." << std::endl;
    }
    catch(bad_month bm) {
        std::cout << "Invalid Month Entered" << bm.what() << '\n'
            << "Using minimum value for month. " << std::endl;
    }

    date start_of_next_year(year+1, Jan, 1);
    date d(year, month, 1);

    // add another month to d until we enter the next year.
    while (d < start_of_next_year){
        std::cout << to_simple_string(d.end_of_month()) << std::endl;
        d += months(1);
    }

    return 0;
}
```

↵

Localization Demonstration

The `boost::date_time` library provides the ability to create customized locale facets. Date ordering, language, separators, and abbreviations can be customized.

```

/* The following shows the creation of a facet for the output of
 * dates in German (please forgive me for any errors in my German --
 * I'm not a native speaker).
 */

#include "boost/date_time/gregorian/gregorian.hpp"
#include <iostream>
#include <algorithm>

/* Define a series of char arrays for short and long name strings
 * to be associated with German date output (US names will be
 * retrieved from the locale). */
const char* const de_short_month_names[] =
{
    "Jan", "Feb", "Mar", "Apr", "Mai", "Jun",
    "Jul", "Aug", "Sep", "Okt", "Nov", "Dez", "NAM"
};
const char* const de_long_month_names[] =
{
    "Januar", "Februar", "Marz", "April", "Mai",
    "Juni", "Juli", "August", "September", "Oktober",
    "November", "Dezember", "NichtDerMonat"
};
const char* const de_long_weekday_names[] =
{
    "Sonntag", "Montag", "Dienstag", "Mittwoch",
    "Donnerstag", "Freitag", "Samstag"
};
const char* const de_short_weekday_names[] =
{
    "Son", "Mon", "Die", "Mit", "Don", "Fre", "Sam"
};

int main()
{
    using namespace boost::gregorian;

    // create some gregorian objects to output
    date d1(2002, Oct, 1);
    greg_month m = d1.month();
    greg_weekday wd = d1.day_of_week();

    // create a facet and a locale for German dates
    date_facet* german_facet = new date_facet();
    std::cout.imbue(std::locale(std::locale::classic(), german_facet));

    // create the German name collections
    date_facet::input_collection_type short_months, long_months,
                                     short_weekdays, long_weekdays;
    std::copy(&de_short_month_names[0], &de_short_month_names[11],
              std::back_inserter(short_months));
    std::copy(&de_long_month_names[0], &de_long_month_names[11],
              std::back_inserter(long_months));
    std::copy(&de_short_weekday_names[0], &de_short_weekday_names[6],
              std::back_inserter(short_weekdays));
    std::copy(&de_long_weekday_names[0], &de_long_weekday_names[6],
              std::back_inserter(long_weekdays));

    // replace the default names with ours
    // NOTE: date_generators and special_values were not replaced as

```

```
// they are not used in this example
german_facet->short_month_names(short_months);
german_facet->long_month_names(long_months);
german_facet->short_weekday_names(short_weekdays);
german_facet->long_weekday_names(long_weekdays);

// output the date in German using short month names
german_facet->format("%d.%m.%Y");
std::cout << dl << std::endl; //01.10.2002

german_facet->month_format("%B");
std::cout << m << std::endl; //Oktober

german_facet->weekday_format("%A");
std::cout << wd << std::endl; //Dienstag

// Output the same gregorian objects using US names
date_facet* us_facet = new date_facet();
std::cout.imbue(std::locale(std::locale::classic(), us_facet));

us_facet->format("%m/%d/%Y");
std::cout << dl << std::endl; // 10/01/2002

// English names, iso order (year-month-day), '-' separator
us_facet->format("%Y-%b-%d");
std::cout << dl << std::endl; // 2002-Oct-01

return 0;
}
└
```

Date Period Calculations

Calculates if a date is in an 'irregular' collection of periods using period calculation functions.

```

/*
This example demonstrates a simple use of periods for the calculation
of date information.

The example calculates if a given date is a weekend or holiday
given an exclusion set. That is, each weekend or holiday is
entered into the set as a time interval. Then if a given date
is contained within any of the intervals it is considered to
be within the exclusion set and hence is a offtime.

Output:
Number Excluded Periods: 5
20020202/20020203
20020209/20020210
20020212/20020212
20020216/20020217
In Exclusion Period: 20020216 --> 20020216/20020217
20020223/20020224

*/

#include "boost/date_time/gregorian/gregorian.hpp"
#include <set>
#include <algorithm>
#include <iostream>

typedef std::set<boost::gregorian::date_period> date_period_set;

//Simple population of the exclusion set
date_period_set
generateExclusion()
{
    using namespace boost::gregorian;
    date_period periods_array[] =
        { date_period(date(2002, Feb, 2), date(2002, Feb, 4)), //weekend of 2nd-3rd
          date_period(date(2002, Feb, 9), date(2002, Feb, 11)),
          date_period(date(2002, Feb, 16), date(2002, Feb, 18)),
          date_period(date(2002, Feb, 23), date(2002, Feb, 25)),
          date_period(date(2002, Feb, 12), date(2002, Feb, 13)) //a random holiday 2-12
        };
    const int num_periods = sizeof(periods_array)/sizeof(date_period);

    date_period_set ps;
    //insert the periods in the set
    std::insert_iterator<date_period_set> itr(ps, ps.begin());
    std::copy(periods_array, periods_array+num_periods, itr );
    return ps;
}

int main()
{
    using namespace boost::gregorian;

    date_period_set ps = generateExclusion();
    std::cout << "Number Excluded Periods: " << ps.size() << std::endl;

    date d(2002, Feb, 16);
    date_period_set::const_iterator i = ps.begin();

```

```
//print the periods, check for containment
for (;i != ps.end(); i++) {
    std::cout << to_iso_string(*i) << std::endl;
    //if date is in exclusion period then print it
    if (i->contains(d)) {
        std::cout << "In Exclusion Period: "
            << to_iso_string(d) << " --> " << to_iso_string(*i)
            << std::endl;
    }
}

return 0;

}
```

└─

Print Holidays

This is an example of using functors to define a holiday schedule

```

/* Generate a set of dates using a collection of date generators
 * Output looks like:
 * Enter Year: 2002
 * 2002-Jan-01 [Tue]
 * 2002-Jan-21 [Mon]
 * 2002-Feb-12 [Tue]
 * 2002-Jul-04 [Thu]
 * 2002-Sep-02 [Mon]
 * 2002-Nov-28 [Thu]
 * 2002-Dec-25 [Wed]
 * Number Holidays: 7
 */

#include "boost/date_time/gregorian/gregorian.hpp"
#include <algorithm>
#include <functional>
#include <vector>
#include <iostream>
#include <set>

void
print_date(boost::gregorian::date d)
{
    using namespace boost::gregorian;
#ifdef BOOST_DATE_TIME_NO_LOCALE
    std::cout << to_simple_string(d) << " [" << d.day_of_week() << "]\n";
#else
    std::cout << d << " [" << d.day_of_week() << "]\n";
#endif
}

int
main() {

    std::cout << "Enter Year: ";
    int year;
    std::cin >> year;

    using namespace boost::gregorian;

    //define a collection of holidays fixed by month and day
    std::vector<year_based_generator*> holidays;
    holidays.push_back(new partial_date(1,Jan)); //Western New Year
    holidays.push_back(new partial_date(4,Jul)); //US Independence Day
    holidays.push_back(new partial_date(25, Dec)); //Christmas day

    //define a shorthand for the nth_day_of_the_week_in_month function object
    typedef nth_day_of_the_week_in_month nth_dow;

    //US labor day
    holidays.push_back(new nth_dow(nth_dow::first, Monday, Sep));
    //MLK Day
    holidays.push_back(new nth_dow(nth_dow::third, Monday, Jan));
    //Pres day
    holidays.push_back(new nth_dow(nth_dow::second, Tuesday, Feb));
    //Thanksgiving
    holidays.push_back(new nth_dow(nth_dow::fourth, Thursday, Nov));

```



```
typedef std::set<date> date_set;
date_set all_holidays;

for(std::vector<year_based_generator*>::iterator it = holidays.begin();
    it != holidays.end(); ++it)
{
    all_holidays.insert((*it)->get_date(year));
}

//print the holidays to the screen
std::for_each(all_holidays.begin(), all_holidays.end(), print_date);
std::cout << "Number Holidays: " << all_holidays.size() << std::endl;

return 0;
}
```

└─

Print Month

Simple utility to print out days of the month with the days of a month. Demonstrates date iteration (`date_time::date_itr`).

```

/* This example prints all the dates in a month. It demonstrates
 * the use of iterators as well as functions of the gregorian_calendar
 *
 * Output:
 * Enter Year: 2002
 * Enter Month(1..12): 2
 * 2002-Feb-01 [Fri]
 * 2002-Feb-02 [Sat]
 * 2002-Feb-03 [Sun]
 * 2002-Feb-04 [Mon]
 * 2002-Feb-05 [Tue]
 * 2002-Feb-06 [Wed]
 * 2002-Feb-07 [Thu]
 */

#include "boost/date_time/gregorian/gregorian.hpp"
#include <iostream>

int
main()
{
    std::cout << "Enter Year: ";
    int year, month;
    std::cin >> year;
    std::cout << "Enter Month(1..12): ";
    std::cin >> month;

    using namespace boost::gregorian;
    try {
        //Use the calendar to get the last day of the month
        int eom_day = gregorian_calendar::end_of_month_day(year,month);
        date endOfMonth(year,month,eom_day);

        //construct an iterator starting with first day of the month
        day_iterator ditr(date(year,month,1));
        //loop thru the days and print each one
        for (; ditr <= endOfMonth; ++ditr) {
            #if defined(BOOST_DATE_TIME_NO_LOCALE)
                std::cout << to_simple_string(*ditr) << " ["
            #else
                std::cout << *ditr << " ["
            #endif
                << ditr->day_of_week() << "]"
                << std::endl;
        }
    }
    catch(std::exception& e) {
        std::cout << "Error bad date, check your entry: \n"
            << "  Details: " << e.what() << std::endl;
    }
    return 0;
}

```

Month Adding

Adding a month to a day without the use of iterators.

```
/* Simple program that uses the gregorian calendar to progress by exactly
 * one month, irregardless of how many days are in that month.
 *
 * This method can be used as an alternative to iterators
 */

#include "boost/date_time/gregorian/gregorian.hpp"
#include <iostream>

int
main()
{
    using namespace boost::gregorian;

    date d = day_clock::local_day();
    add_month mf(1);
    date d2 = d + mf.get_offset(d);
    std::cout << "Today is: " << to_simple_string(d) << ".\n"
        << "One month from today will be: " << to_simple_string(d2)
        << std::endl;

    return 0;
}
└
```

Time Math

Various types of calculations with times and time durations.

```
/* Some simple examples of constructing and calculating with times
 * Output:
 * 2002-Feb-01 00:00:00 - 2002-Feb-01 05:04:02.001000000 = -5:04:02.001000000
 */

#include "boost/date_time/posix_time/posix_time.hpp"
#include <iostream>

int
main()
{
    using namespace boost::posix_time;
    using namespace boost::gregorian;

    date d(2002, Feb, 1); //an arbitrary date
    //construct a time by adding up some durations
    ptime t1(d, hours(5)+minutes(4)+seconds(2)+millisec(1));
    //construct a new time by subtracting some times
    ptime t2 = t1 - hours(5)- minutes(4)- seconds(2)- millisec(1);
    //construct a duration by taking the difference between times
    time_duration td = t2 - t1;

    std::cout << to_simple_string(t2) << " - "
               << to_simple_string(t1) << " = "
               << to_simple_string(td) << std::endl;

    return 0;
}
```

Print Hours

Demonstrate time iteration, clock retrieval, and simple calculation.

```
/* Print the remaining hours of the day
 * Uses the clock to get the local time
 * Use an iterator to iterate over the remaining hours
 * Retrieve the date part from a time
 *
 * Expected Output something like:
 *
 * 2002-Mar-08 16:30:59
 * 2002-Mar-08 17:30:59
 * 2002-Mar-08 18:30:59
 * 2002-Mar-08 19:30:59
 * 2002-Mar-08 20:30:59
 * 2002-Mar-08 21:30:59
 * 2002-Mar-08 22:30:59
 * 2002-Mar-08 23:30:59
 * Time left till midnight: 07:29:01
 */

#include "boost/date_time/posix_time/posix_time.hpp"
#include <iostream>

int
main()
{
    using namespace boost::posix_time;
    using namespace boost::gregorian;

    //get the current time from the clock -- one second resolution
    ptime now = second_clock::local_time();
    //Get the date part out of the time
    date today = now.date();
    date tomorrow = today + days(1);
    ptime tomorrow_start(tomorrow); //midnight

    //iterator adds by one hour
    time_iterator titr(now, hours(1));
    for (; titr < tomorrow_start; ++titr) {
        std::cout << to_simple_string(*titr) << std::endl;
    }

    time_duration remaining = tomorrow_start - now;
    std::cout << "Time left till midnight: "
              << to_simple_string(remaining) << std::endl;
    return 0;
}
```

↵

Local to UTC Conversion

Demonstrate utc to local and local to utc calculations including dst.

```

/* Demonstrate conversions between a local time and utc
 * Output:
 *
 * UTC <--> New York while DST is NOT active (5 hours)
 * 2001-Dec-31 19:00:00 in New York is 2002-Jan-01 00:00:00 UTC time
 * 2002-Jan-01 00:00:00 UTC is 2001-Dec-31 19:00:00 New York time
 *
 * UTC <--> New York while DST is active (4 hours)
 * 2002-May-31 20:00:00 in New York is 2002-Jun-01 00:00:00 UTC time
 * 2002-Jun-01 00:00:00 UTC is 2002-May-31 20:00:00 New York time
 *
 * UTC <--> Arizona (7 hours)
 * 2002-May-31 17:00:00 in Arizona is 2002-Jun-01 00:00:00 UTC time
 */

#include "boost/date_time/posix_time/posix_time.hpp"
#include "boost/date_time/local_time_adjustor.hpp"
#include "boost/date_time/c_local_time_adjustor.hpp"
#include <iostream>

int
main()
{
    using namespace boost::posix_time;
    using namespace boost::gregorian;

    //This local adjustor depends on the machine TZ settings-- highly dangerous!
    typedef boost::date_time::c_local_adjustor<ptime> local_adj;
    ptime t10(date(2002,Jan,1), hours(7));
    ptime t11 = local_adj::utc_to_local(t10);
    std::cout << "UTC <--> Zone base on TZ setting" << std::endl;
    std::cout << to_simple_string(t11) << " in your TZ is "
              << to_simple_string(t10) << " UTC time "
              << std::endl;
    time_duration td = t11 - t10;
    std::cout << "A difference of: "
              << to_simple_string(td) << std::endl;

    //eastern timezone is utc-5
    typedef boost::date_time::local_adjustor<ptime, -5, us_dst> us_eastern;

    ptime t1(date(2001,Dec,31), hours(19)); //5 hours b/f midnight NY time

    std::cout << "\nUTC <--> New York while DST is NOT active (5 hours)"
              << std::endl;
    ptime t2 = us_eastern::local_to_utc(t1);
    std::cout << to_simple_string(t1) << " in New York is "
              << to_simple_string(t2) << " UTC time "
              << std::endl;

    ptime t3 = us_eastern::utc_to_local(t2); //back should be the same
    std::cout << to_simple_string(t2) << " UTC is "
              << to_simple_string(t3) << " New York time "
              << "\n\n";

    ptime t4(date(2002,May,31), hours(20)); //4 hours b/f midnight NY time
    std::cout << "UTC <--> New York while DST is active (4 hours)" << std::endl;
    ptime t5 = us_eastern::local_to_utc(t4);
    std::cout << to_simple_string(t4) << " in New York is "

```

```
        << to_simple_string(t5) << " UTC time "
        << std::endl;

ptime t6 = us_eastern::utc_to_local(t5); //back should be the same
std::cout << to_simple_string(t5) << " UTC is "
          << to_simple_string(t6) << " New York time "
          << "\n" << std::endl;

//Arizona timezone is utc-7 with no dst
typedef boost::date_time::local_adjustor<ptime, -7, no_dst> us_arizona;

ptime t7(date(2002,May,31), hours(17));
std::cout << "UTC <--> Arizona (7 hours)" << std::endl;
ptime t8 = us_arizona::local_to_utc(t7);
std::cout << to_simple_string(t7) << " in Arizona is "
          << to_simple_string(t8) << " UTC time "
          << std::endl;

return 0;
}

└
```

Time Periods

Demonstrate some simple uses of time periods.

```

/* Some simple examples of constructing and calculating with times
 * Returns:
 * [2002-Feb-01 00:00:00/2002-Feb-01 23:59:59.999999999]
 * contains 2002-Feb-01 03:00:05
 * [2002-Feb-01 00:00:00/2002-Feb-01 23:59:59.999999999]
 * intersected with
 * [2002-Feb-01 00:00:00/2002-Feb-01 03:00:04.999999999]
 * is
 * [2002-Feb-01 00:00:00/2002-Feb-01 03:00:04.999999999]
 */

#include "boost/date_time/posix_time/posix_time.hpp"
#include <iostream>

using namespace boost::posix_time;
using namespace boost::gregorian;

//Create a simple period class to contain all the times in a day
class day_period : public time_period
{
public:
    day_period(date d) : time_period(ptime(d), //midnight
                                   ptime(d, hours(24)))
    {}
};

int
main()
{
    date d(2002, Feb, 1); //an arbitrary date
    //a period that represents a day
    day_period dp(d);
    ptime t(d, hours(3)+seconds(5)); //an arbitray time on that day
    if (dp.contains(t)) {
        std::cout << to_simple_string(dp) << " contains "
                  << to_simple_string(t) << std::endl;
    }
    //a period that represents part of the day
    time_period part_of_day(ptime(d, hours(0)), t);
    //intersect the 2 periods and print the results
    if (part_of_day.intersects(dp)) {
        time_period result = part_of_day.intersection(dp);
        std::cout << to_simple_string(dp) << " intersected with\n"
                  << to_simple_string(part_of_day) << " is \n"
                  << to_simple_string(result) << std::endl;
    }

    return 0;
}

```

Simple Time Zones

Example usage of custom_time_zone as well as posix_time_zone.


```

/* A simple example for using a custom_time_zone and a posix_time_zone.
*/

#include "boost/date_time/local_time/local_time.hpp"
#include <iostream>

int
main()
{
    using namespace boost;
    using namespace local_time;
    using namespace gregorian;
    using posix_time::time_duration;

    /***** custom_time_zone *****/

    // create the dependent objects for a custom_time_zone
    time_zone_names tzn("Eastern Standard Time", "EST",
                        "Eastern Daylight Time", "EDT");
    time_duration utc_offset(-5,0,0);
    dst_adjustment_offsets adj_offsets(time_duration(1,0,0),
                                       time_duration(2,0,0),
                                       time_duration(2,0,0));

    // rules for this zone are:
    // start on first Sunday of April at 2 am
    // end on last Sunday of October at 2 am
    // so we use a first_last_dst_rule
    first_day_of_the_week_in_month start_rule(Sunday, Apr);
    last_day_of_the_week_in_month end_rule(Sunday, Oct);
    shared_ptr<dst_calc_rule> nyc_rules(new first_last_dst_rule(start_rule,
                                                                end_rule));

    // create more dependent objects for a non-dst custom_time_zone
    time_zone_names tzn2("Mountain Standard Time", "MST",
                        "", ""); // no dst means empty dst strings
    time_duration utc_offset2(-7,0,0);
    dst_adjustment_offsets adj_offsets2(time_duration(0,0,0),
                                       time_duration(0,0,0),
                                       time_duration(0,0,0));

    // no dst means we need a null pointer to the rules
    shared_ptr<dst_calc_rule> phx_rules;

    // create the custom_time_zones
    time_zone_ptr nyc_1(new custom_time_zone(tzn, utc_offset,
                                             adj_offsets, nyc_rules));
    time_zone_ptr phx_1(new custom_time_zone(tzn2, utc_offset2,
                                             adj_offsets2, phx_rules));

    /***** posix_time_zone *****/

    // create posix_time_zones that are the duplicates of the
    // custom_time_zones created above. See posix_time_zone documentation
    // for details on full zone names.
    std::string nyc_string, phx_string;
    nyc_string = "EST-05:00:00EDT+01:00:00,M4.1.0/02:00:00,M10.5.0/02:00:00";
    // nyc_string = "EST-05EDT,M4.1.0,M10.5.0"; // shorter when defaults used
    phx_string = "MST-07"; // no-dst
    time_zone_ptr nyc_2(new posix_time_zone(nyc_string));
    time_zone_ptr phx_2(new posix_time_zone(phx_string));

    /***** show the sets are equal *****/

```

```
std::cout << "The first zone is in daylight savings from:\n "  
  << nyc_1->dst_local_start_time(2004) << " through "  
  << nyc_1->dst_local_end_time(2004) << std::endl;  
  
std::cout << "The second zone is in daylight savings from:\n "  
  << nyc_2->dst_local_start_time(2004) << " through "  
  << nyc_2->dst_local_end_time(2004) << std::endl;  
  
std::cout << "The third zone (no daylight savings):\n "  
  << phx_1->std_zone_abbrev() << " and "  
  << phx_1->base_utc_offset() << std::endl;  
  
std::cout << "The fourth zone (no daylight savings):\n "  
  << phx_2->std_zone_abbrev() << " and "  
  << phx_2->base_utc_offset() << std::endl;  
  
return 0;  
}
```

Daylight Savings Calc Rules

Example of creating various Daylight Savings Calc Rule objects.

```
/* A simple example for creating various dst_calc_rule instances
 */

#include "boost/date_time/gregorian/gregorian.hpp"
#include "boost/date_time/local_time/local_time.hpp"
#include <iostream>

int
main()
{
    using namespace boost;
    using namespace local_time;
    using namespace gregorian;

    /***** create the necessary date_generator objects *****/
    // starting generators
    first_day_of_the_week_in_month fd_start(Sunday, May);
    last_day_of_the_week_in_month ld_start(Sunday, May);
    nth_day_of_the_week_in_month nkd_start(nth_day_of_the_week_in_month::third,
                                           Sunday, May);

    partial_date pd_start(1, May);
    // ending generators
    first_day_of_the_week_in_month fd_end(Sunday, Oct);
    last_day_of_the_week_in_month ld_end(Sunday, Oct);
    nth_day_of_the_week_in_month nkd_end(nth_day_of_the_week_in_month::third,
                                           Sunday, Oct);

    partial_date pd_end(31, Oct);

    /***** create the various dst_calc_rule objects *****/
    dst_calc_rule_ptr pdr(new partial_date_dst_rule(pd_start, pd_end));
    dst_calc_rule_ptr flr(new first_last_dst_rule(fd_start, ld_end));
    dst_calc_rule_ptr llr(new last_last_dst_rule(ld_start, ld_end));
    dst_calc_rule_ptr nlr(new nth_last_dst_rule(nkd_start, ld_end));
    dst_calc_rule_ptr ndr(new nth_day_of_the_week_in_month_dst_rule(nkd_start,
                                                                    nkd_end));

    return 0;
}

└
```

Flight Time Example

This example shows a program that calculates the arrival time of a plane that flies from Phoenix to New York. During the flight New York shifts into daylight savings time (Phoenix doesn't because Arizona doesn't use dst).

```
#include "boost/date_time/local_time/local_time.hpp"
#include <iostream>

/* This example shows a program that calculates the arrival time of a plane
 * that flies from Phoenix to New York. During the flight New York shifts
 * into daylight savings time (Phoenix doesn't because Arizona doesn't use
 * dst).
 *
 *
 */

int main()
{
    using namespace boost::gregorian;
    using namespace boost::local_time;
    using namespace boost::posix_time;

    //setup some timezones for creating and adjusting local times
    //This user editable file can be found in libs/date_time/data.
    tz_database tz_db;
    tz_db.load_from_file("date_time_zonespec.csv");
    time_zone_ptr nyc_tz = tz_db.time_zone_from_region("America/New_York");
    //Use a
    time_zone_ptr phx_tz(new posix_time_zone("MST-07:00:00"));

    //local departure time in phoenix is 11 pm on april 2 2005
    // (ny changes to dst on apr 3 at 2 am)
    local_date_time phx_departure(date(2005, Apr, 2), hours(23),
                                   phx_tz,
                                   local_date_time::NOT_DATE_TIME_ON_ERROR);

    time_duration flight_length = hours(4) + minutes(30);
    local_date_time phx_arrival = phx_departure + flight_length;
    local_date_time nyc_arrival = phx_arrival.local_time_in(nyc_tz);

    std::cout << "departure phx time: " << phx_departure << std::endl;
    std::cout << "arrival phx time:   " << phx_arrival << std::endl;
    std::cout << "arrival nyc time:    " << nyc_arrival << std::endl;
}
```

Seconds Since Epoch

Example of calculating seconds elapsed since epoch (1970-Jan-1) using local_date_time.

```

/* This example demonstrates the use of the time zone database and
 * local time to calculate the number of seconds since the UTC
 * time_t epoch 1970-01-01 00:00:00. Note that the selected timezone
 * could be any timezone supported in the time zone database file which
 * can be modified and updated as needed by the user.
 *
 * To solve this problem the following steps are required:
 * 1) Get a timezone from the tz database for the local time
 * 2) Construct a local time using the timezone
 * 3) Construct a posix_time::ptime for the time_t epoch time
 * 4) Convert the local_time to utc and subtract the epoch time
 *
 */

#include "boost/date_time/local_time/local_time.hpp"
#include <iostream>

int main()
{
    using namespace boost::gregorian;
    using namespace boost::local_time;
    using namespace boost::posix_time;

    tz_database tz_db;
    try {
        tz_db.load_from_file("../data/date_time_zonespec.csv");
    } catch(data_not_accessible dna) {
        std::cerr << "Error with time zone data file: " << dna.what() << std::endl;
        exit(EXIT_FAILURE);
    } catch(bad_field_count bfc) {
        std::cerr << "Error with time zone data file: " << bfc.what() << std::endl;
        exit(EXIT_FAILURE);
    }

    time_zone_ptr nyc_tz = tz_db.time_zone_from_region("America/New_York");
    date in_date(2004,10,04);
    time_duration td(12,14,32);
    // construct with local time value
    // create not-a-date-time if invalid (eg: in dst transition)
    local_date_time nyc_time(in_date,
                             td,
                             nyc_tz,
                             local_date_time::NOT_DATE_TIME_ON_ERROR);

    std::cout << nyc_time << std::endl;

    ptime time_t_epoch(date(1970,1,1));
    std::cout << time_t_epoch << std::endl;

    // first convert nyc_time to utc via the utc_time()
    // call and subtract the ptime.
    time_duration diff = nyc_time.utc_time() - time_t_epoch;

    //Expected 1096906472
    std::cout << "Seconds diff: " << diff.total_seconds() << std::endl;
}

```

Library Reference

The following is a detailed reference of the `date_time` library. A click on any of the reference links will take you to a list of the header files found in that section. Following one of those links will take you to a list of the items declared in that header file. Further sublinks take you to detailed descriptions of each individual item.

Date Time Reference

Header [`<boost/date_time/adjust_functors.hpp>`](#)

```
namespace boost {
  namespace date_time {
    template<typename date_type> class day_functor;
    template<typename date_type> class month_functor;
    template<typename date_type> class week_functor;
    template<typename date_type> class year_functor;
  }
}
```

Class template `day_functor`

`boost::date_time::day_functor` — Functor to iterate a fixed number of days.

Synopsis

```
// In header: <boost/date_time/adjust_functors.hpp>

template<typename date_type>
class day_functor {
public:
    // types
    typedef date_type::duration_type duration_type;

    // construct/copy/destruct
    day_functor(int);

    // public member functions
    duration_type get_offset(const date_type &) const;
    duration_type get_neg_offset(const date_type &) const;
};
```

Description

`day_functor` public construct/copy/destruct

1. `day_functor(int f);`

`day_functor` public member functions

1. `duration_type get_offset(const date_type & d) const;`

2. `duration_type get_neg_offset(const date_type & d) const;`

Class template month_functor

boost::date_time::month_functor — Provides calculation to find next nth month given a date.

Synopsis

```
// In header: <boost/date_time/adjust_functors.hpp>

template<typename date_type>
class month_functor {
public:
    // types
    typedef date_type::duration_type duration_type;
    typedef date_type::calendar_type cal_type;
    typedef cal_type::ymd_type ymd_type;
    typedef cal_type::day_type day_type;

    // construct/copy/destruct
    month_functor(int);

    // public member functions
    duration_type get_offset(const date_type &) const;
    duration_type get_neg_offset(const date_type &) const;
};
```

Description

This adjustment function provides the logic for 'month-based' advancement on a ymd based calendar. The policy it uses to handle the non existant end of month days is to back up to the last day of the month. Also, if the starting date is the last day of a month, this functor will attempt to adjust to the end of the month.

month_functor public construct/copy/destruct

1. `month_functor(int f);`

month_functor public member functions

1. `duration_type get_offset(const date_type & d) const;`
2. `duration_type get_neg_offset(const date_type & d) const;`

Returns a negative duration_type.

Class template week_functor

boost::date_time::week_functor — Functor to iterate a over weeks.

Synopsis

```
// In header: <boost/date_time/adjust_functors.hpp>

template<typename date_type>
class week_functor {
public:
    // types
    typedef date_type::duration_type duration_type;
    typedef date_type::calendar_type calendar_type;

    // construct/copy/destruct
    week_functor(int);

    // public member functions
    duration_type get_offset(const date_type &) const;
    duration_type get_neg_offset(const date_type &) const;
};
```

Description

week_functor public construct/copy/destruct

1. `week_functor(int f);`

week_functor public member functions

1. `duration_type get_offset(const date_type & d) const;`
2. `duration_type get_neg_offset(const date_type & d) const;`

Class template year_functor

boost::date_time::year_functor — Functor to iterate by a year adjusting for leap years.

Synopsis

```
// In header: <boost/date_time/adjust_functors.hpp>

template<typename date_type>
class year_functor {
public:
    // types
    typedef date_type::duration_type duration_type;

    // construct/copy/destruct
    year_functor(int);

    // public member functions
    duration_type get_offset(const date_type &) const;
    duration_type get_neg_offset(const date_type &) const;
};
```


Description

year_functor public construct/copy/destruct

```
1. year_functor(int f);
```

year_functor public member functions

```
1. duration_type get_offset(const date_type & d) const;
```

```
2. duration_type get_neg_offset(const date_type & d) const;
```

Header <boost/date_time/c_local_time_adjustor.hpp>

Time adjustment calculations based on machine

```
namespace boost {  
    namespace date_time {  
        template<typename time_type> class c_local_adjustor;  
    }  
}
```

Class template c_local_adjustor

boost::date_time::c_local_adjustor — Adjust to / from utc using the C API.

Synopsis

```
// In header: <boost/date_time/c_local_time_adjustor.hpp>  
  
template<typename time_type>  
class c_local_adjustor {  
public:  
    // types  
    typedef time_type::time_duration_type time_duration_type;  
    typedef time_type::date_type          date_type;  
    typedef date_type::duration_type      date_duration_type;  
  
    // public static functions  
    static time_type utc_to_local(const time_type &);  
};
```

Description

Warning!!! This class assumes that timezone settings of the machine are correct. This can be a very dangerous assumption.

c_local_adjustor public static functions

```
1. static time_type utc_to_local(const time_type & t);
```

Convert a utc time to local time.

Header `<boost/date_time/c_time.hpp>`

Provide workarounds related to the ctime header

```
namespace std {
}namespace boost {
    namespace date_time {
        struct c_time;
    }
}
```

Struct `c_time`

`boost::date_time::c_time` — Provides a uniform interface to some 'ctime' functions.

Synopsis

```
// In header: <boost/date_time/c_time.hpp>

struct c_time {

    // public static functions
    static std::tm * localtime(const std::time_t *, std::tm *);
    static std::tm * gmtime(const std::time_t *, std::tm *);
    static std::tm * localtime(const std::time_t *, std::tm *);
    static std::tm * gmtime(const std::time_t *, std::tm *);
};
```

Description

Provides a uniform interface to some ctime functions and their '_r' counterparts. The '_r' functions require a pointer to a user created `std::tm` struct whereas the regular functions use a statically created struct and return a pointer to that. These wrapper functions require the user to create a `std::tm` struct and send in a pointer to it. This struct may be used to store the resulting time. The returned pointer may or may not point to this struct, however, it will point to the result of the corresponding function. All functions do proper checking of the C function results and throw exceptions on error. Therefore the functions will never return NULL.

`c_time` public static functions

1.

```
static std::tm * localtime(const std::time_t * t, std::tm * result);
```

requires a pointer to a user created `std::tm` struct

2.

```
static std::tm * gmtime(const std::time_t * t, std::tm * result);
```

requires a pointer to a user created `std::tm` struct

3.

```
static std::tm * localtime(const std::time_t * t, std::tm * result);
```

requires a pointer to a user created `std::tm` struct

4.

```
static std::tm * gmtime(const std::time_t * t, std::tm * result);
```

requires a pointer to a user created `std::tm` struct

Header <boost/date_time/compiler_config.hpp>

Header <boost/date_time/constrained_value.hpp>

```

namespace boost {
    namespace CV {
        template<typename value_policies> class constrained_value;
        template<typename rep_type, rep_type min_value, rep_type max_value,
                typename exception_type>
            class simple_exception_policy;

        // Represent a min or max violation type.
        enum violation_enum { min_violation, max_violation };
    }
}

```

Class template constrained_value

boost::CV::constrained_value — A template to specify a constrained basic value type.

Synopsis

```

// In header: <boost/date_time/constrained_value.hpp>

template<typename value_policies>
class constrained_value {
public:
    // types
    typedef value_policies::value_type value_type;

    // construct/copy/destroy
    constrained_value(value_type);
    constrained_value& operator=(value_type);

    // public member functions
    operator value_type() const;

    // public static functions
    static value_type max BOOST_PREVENT_MACRO_SUBSTITUTION();
    static value_type min BOOST_PREVENT_MACRO_SUBSTITUTION();

    // private member functions
    void assign(value_type);
};

```

Description

This template provides a quick way to generate an integer type with a constrained range. The type provides for the ability to specify the min, max, and and error handling policy.

value policies A class that provides the range limits via the min and max functions as well as a function on_error that determines how errors are handled. A common strategy would be to assert or throw an exception. The on_error is passed both the current value and the new value that is in error.

constrained_value public construct/copy/destroy

1. `constrained_value(value_type value);`

2. `constrained_value& operator=(value_type v);`

constrained_value public member functions

1. `operator value_type() const;`

Coerce into the representation type.

constrained_value public static functions

1. `static value_type max BOOST_PREVENT_MACRO_SUBSTITUTION();`

Return the max allowed value (traits method)

2. `static value_type min BOOST_PREVENT_MACRO_SUBSTITUTION();`

Return the min allowed value (traits method)

constrained_value private member functions

1. `void assign(value_type value);`

Class template simple_exception_policy

boost::CV::simple_exception_policy — Template to shortcut the [constrained_value](#) policy creation process.

Synopsis

```
// In header: <boost/date_time/constrained_value.hpp>

template<typename rep_type, rep_type min_value, rep_type max_value,
        typename exception_type>
class simple_exception_policy {
public:
    // types
    typedef rep_type value_type;

    // member classes/structs/unions

    struct exception_wrapper {

        // public member functions
        operator std::out_of_range() const;
    };

    // public static functions
    static rep_type min BOOST_PREVENT_MACRO_SUBSTITUTION();
    static rep_type max BOOST_PREVENT_MACRO_SUBSTITUTION();
    static void on_error(rep_type, rep_type, violation_enum);
};
```

Description

`simple_exception_policy` public static functions

1.

```
static rep_type min BOOST_PREVENT_MACRO_SUBSTITUTION();
```
2.

```
static rep_type max BOOST_PREVENT_MACRO_SUBSTITUTION();
```
3.

```
static void on_error(rep_type, rep_type, violation_enum);
```

Struct `exception_wrapper`

`boost::CV::simple_exception_policy::exception_wrapper`

Synopsis

```
// In header: <boost/date_time/constrained_value.hpp>

struct exception_wrapper {

    // public member functions
    operator std::out_of_range() const;
};
```

Description

`exception_wrapper` public member functions

1.

```
operator std::out_of_range() const;
```

Header `<boost/date_time/date.hpp>`

```
namespace boost {
    namespace date_time {
        template<typename T, typename calendar, typename duration_type_> class date;
    }
}
```

Class template `date`

`boost::date_time::date` — Representation of timepoint at the one day level resolution.

Synopsis

```
// In header: <boost/date_time/date.hpp>

template<typename T, typename calendar, typename duration_type_>
class date {
public:
    // types
    typedef T date_type;
    typedef calendar calendar_type;
    typedef calendar::date_traits_type traits_type;
    typedef duration_type_ duration_type;
    typedef calendar::year_type year_type;
    typedef calendar::month_type month_type;
    typedef calendar::day_type day_type;
    typedef calendar::ymd_type ymd_type;
    typedef calendar::date_rep_type date_rep_type;
    typedef calendar::date_int_type date_int_type;
    typedef calendar::day_of_week_type day_of_week_type;

    // construct/copy/destruct
    date(year_type, month_type, day_type);
    date(const ymd_type &);
    explicit date(date_int_type);
    explicit date(date_rep_type);

    // public member functions
    year_type year() const;
    month_type month() const;
    day_type day() const;
    day_of_week_type day_of_week() const;
    ymd_type year_month_day() const;
    bool operator<(const date_type &) const;
    bool operator==(const date_type &) const;
    bool is_special() const;
    bool is_not_a_date() const;
    bool is_infinity() const;
    bool is_pos_infinity() const;
    bool is_neg_infinity() const;
    special_values as_special() const;
    duration_type operator-(const date_type &) const;
    date_type operator-(const duration_type &) const;
    date_type operator+=(const duration_type &);
    date_rep_type day_count() const;
    date_type operator+(const duration_type &) const;
    date_type operator+=(const duration_type &);
};
```

Description

The date template represents an interface shell for a date class that is based on a year-month-day system such as the gregorian or iso systems. It provides basic operations to enable calculation and comparisons.

Theory

This date representation fundamentally departs from the C tm struct approach. The goal for this type is to provide efficient date operations (add, subtract) and storage (minimize space to represent) in a concrete class. Thus, the date uses a count internally to represent a particular date. The calendar parameter defines the policies for converting the the year-month-day and internal counted form here. Applications that need to perform heavy formatting of the same date repeatedly will perform better by using the year-month-day representation.

Internally the date uses a day number to represent the date. This is a monotonic time representation. This representation allows for fast comparison as well as simplifying the creation of writing numeric operations. Essentially, the internal day number is like adjusted julian day. The adjustment is determined by the Epoch date which is represented as day 1 of the calendar. Day 0 is reserved for negative infinity so that any actual date is automatically greater than negative infinity. When a date is constructed from a date or formatted for output, the appropriate conversions are applied to create the year, month, day representations.

date public construct/copy/destruct

1. `date(year_type y, month_type m, day_type d);`

2. `date(const ymd_type & ymd);`

3. `explicit date(date_int_type days);`

This is a private constructor which allows for the creation of new dates. It is not exposed to users since that would require class users to understand the inner workings of the date class.

4. `explicit date(date_rep_type days);`

date public member functions

1. `year_type year() const;`

2. `month_type month() const;`

3. `day_type day() const;`

4. `day_of_week_type day_of_week() const;`

5. `ymd_type year_month_day() const;`

6. `bool operator<(const date_type & rhs) const;`

7. `bool operator==(const date_type & rhs) const;`

8. `bool is_special() const;`

check to see if date is a special value

9. `bool is_not_a_date() const;`

check to see if date is not a value

10. `bool is_infinity() const;`

check to see if date is one of the infinity values

11. `bool is_pos_infinity() const;`

check to see if date is greater than all possible dates

12. `bool is_neg_infinity() const;`

check to see if date is greater than all possible dates

13. `special_values as_special() const;`

return as a special value or a not_special if a normal date

14. `duration_type operator-(const date_type & d) const;`

15. `date_type operator-(const duration_type & dd) const;`

16. `date_type operator-=(const duration_type & dd);`

17. `date_rep_type day_count() const;`

18. `date_type operator+(const duration_type & dd) const;`

19. `date_type operator+=(const duration_type & dd);`

Header `<boost/date_time/date_clock_device.hpp>`

```
namespace boost {  
    namespace date_time {  
        template<typename date_type> class day_clock;  
    }  
}
```

Class template `day_clock`

`boost::date_time::day_clock` — A clock providing day level services based on C `time_t` capabilities.

Synopsis

```
// In header: <boost/date_time/date_clock_device.hpp>

template<typename date_type>
class day_clock {
public:
    // types
    typedef date_type::ymd_type ymd_type;

    // public static functions
    static date_type local_day();
    static date_type::ymd_type local_day_ymd();
    static date_type::ymd_type universal_day_ymd();
    static date_type universal_day();

    // private static functions
    static ::std::tm * get_local_time(std::tm &);
    static ::std::tm * get_universal_time(std::tm &);
};
```

Description

This clock uses Posix interfaces as its implementation and hence uses the timezone settings of the operating system. Incorrect user settings will result in incorrect results for the calls to `local_day`.

`day_clock` public static functions

1. `static date_type local_day();`

Get the local day as a date type.

2. `static date_type::ymd_type local_day_ymd();`

Get the local day as a `ymd_type`.

3. `static date_type::ymd_type universal_day_ymd();`

Get the current day in universal date as a `ymd_type`.

4. `static date_type universal_day();`

Get the UTC day as a date type.

`day_clock` private static functions

1. `static ::std::tm * get_local_time(std::tm & result);`

2. `static ::std::tm * get_universal_time(std::tm & result);`

Header <boost/date_time/date_defs.hpp>

```
namespace boost {
    namespace date_time {

        // An enumeration of weekday names.
        enum weekdays { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,
                        Saturday };

        // Simple enum to allow for nice programming with Jan, Feb, etc.
        enum months_of_year { Jan = 1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep,
                             Oct, Nov, Dec, NotAMonth, NumMonths };

    }
}
```

Header <boost/date_time/date_duration.hpp>

```
namespace boost {
    namespace date_time {
        template<typename duration_rep_traits> class date_duration;

        struct duration_traits_long;
        struct duration_traits_adapted;
    }
}
```

Class template date_duration

boost::date_time::date_duration — Duration type with date level resolution.

Synopsis

```
// In header: <boost/date_time/date_duration.hpp>

template<typename duration_rep_traits>
class date_duration : private boost::less_than_comparable< date_duration< duration_rep_traits > >, boost::equality_comparable< date_duration< duration_rep_traits > >, boost::addable< date_duration< duration_rep_traits > >, boost::subtractable< date_duration< duration_rep_traits > >, boost::dividable2< date_duration< duration_rep_traits > >, int > > > >
{
public:
    // types
    typedef duration_rep_traits::int_type    duration_rep_type;
    typedef duration_rep_traits::impl_type   duration_rep;

    // construct/copy/destruct
    explicit date_duration(duration_rep);
    date_duration(special_values);
    date_duration(const date_duration< duration_rep_traits > &);

    // public member functions
    duration_rep get_rep() const;
    bool is_special() const;
    duration_rep_type days() const;
    bool operator==(const date_duration &) const;
    bool operator<(const date_duration &) const;
    date_duration & operator-=(const date_duration &);
    date_duration & operator+=(const date_duration &);
    date_duration operator-() const;
    date_duration & operator/=(int);
    bool is_negative() const;

    // public static functions
    static date_duration unit();
};
```

Description

date_duration public construct/copy/destruct

1. `explicit date_duration(duration_rep day_count);`

Construct from a day count.

2. `date_duration(special_values sv);`

construct from special_values - only works when instantiated with `duration_traits_adapted`

3. `date_duration(const date_duration< duration_rep_traits > & other);`

Construct from another `date_duration` (Copy Constructor)

date_duration public member functions

1. `duration_rep get_rep() const;`

returns days_ as it's instantiated type - used for streaming

2. `bool is_special() const;`

3. `duration_rep_type days() const;`

returns days as value, not object.

4. `bool operator==(const date_duration & rhs) const;`

Equality.

5. `bool operator<(const date_duration & rhs) const;`

Less.

6. `date_duration & operator--(const date_duration & rhs);`

Subtract another duration -- result is signed.

7. `date_duration & operator+=(const date_duration & rhs);`

Add a duration -- result is signed.

8. `date_duration operator-() const;`

unary- Allows for `dd = -date_duration(2);` -> `dd == -2`

9. `date_duration & operator/=(int divisor);`

Division operations on a duration with an integer.

10. `bool is_negative() const;`

return sign information

date_duration public static functions

1. `static date_duration unit();`

Returns the smallest duration -- used by to calculate 'end'.

Struct duration_traits_long

`boost::date_time::duration_traits_long`

Synopsis

```
// In header: <boost/date_time/date_duration.hpp>

struct duration_traits_long {
    // types
    typedef long int_type;
    typedef long impl_type;

    // public static functions
    static int_type as_number(impl_type);
};
```

Description

Struct for instantiating `date_duration` with **NO** special values functionality. Allows for transparent implementation of either `date_duration<long>` or `date_duration<int_adapter<long>>`

duration_traits_long public static functions

1.

```
static int_type as_number(impl_type i);
```

Struct duration_traits_adapted

boost::date_time::duration_traits_adapted

Synopsis

```
// In header: <boost/date_time/date_duration.hpp>

struct duration_traits_adapted {
    // types
    typedef long int_type;
    typedef boost::date_time::int_adapter< long > impl_type;

    // public static functions
    static int_type as_number(impl_type);
};
```

Description

Struct for instantiating `date_duration` **WITH** special values functionality. Allows for transparent implementation of either `date_duration<long>` or `date_duration<int_adapter<long>>`

duration_traits_adapted public static functions

1.

```
static int_type as_number(impl_type i);
```

Header `<boost/date_time/date_duration_types.hpp>`

```
namespace boost {
    namespace date_time {
        template<typename duration_config> class weeks_duration;
        template<typename base_config> class months_duration;
        template<typename base_config> class years_duration;
    }
}
```

Class template `weeks_duration`

`boost::date_time::weeks_duration` — Additional duration type that represents a number of $n \cdot 7$ days.

Synopsis

```
// In header: <boost/date_time/date_duration_types.hpp>

template<typename duration_config>
class weeks_duration :
    public boost::date_time::date_duration< duration_config >
{
public:
    // construct/copy/destroy
    weeks_duration(typename duration_config::impl_type);
    weeks_duration(special_values);
};
```

Description

`weeks_duration` public construct/copy/destroy

1. `weeks_duration(typename duration_config::impl_type w);`
2. `weeks_duration(special_values sv);`

Class template `months_duration`

`boost::date_time::months_duration` — additional duration type that represents a logical month

Synopsis

```
// In header: <boost/date_time/date_duration_types.hpp>

template<typename base_config>
class months_duration {
public:
    // construct/copy/destroy
    months_duration(int_rep);
    months_duration(special_values);

    // public member functions
    int_rep number_of_months() const;
    duration_type get_neg_offset(const date_type &) const;
    duration_type get_offset(const date_type &) const;
    bool operator==(const months_type &) const;
    bool operator!=(const months_type &) const;
    months_type operator+(const months_type &) const;
    months_type & operator+=(const months_type &);
    months_type operator-(const months_type &) const;
    months_type & operator-=(const months_type &);
    months_type operator*(const int_type) const;
    months_type & operator*=(const int_type);
    months_type operator/(const int_type) const;
    months_type & operator/=(const int_type);
    months_type operator+(const years_type &) const;
    months_type & operator+=(const years_type &);
    months_type operator-(const years_type &) const;
    months_type & operator-=(const years_type &);
};
```

Description

A logical month enables things like: "date(2002,Mar,2) + months(2) -> 2002-May2". If the date is a last day-of-the-month, the result will also be a last-day-of-the-month.

months_duration public construct/copy/destroy

1. `months_duration(int_rep num);`
2. `months_duration(special_values sv);`

months_duration public member functions

1. `int_rep number_of_months() const;`
2. `duration_type get_neg_offset(const date_type & d) const;`
returns a negative duration
3. `duration_type get_offset(const date_type & d) const;`

4.

```
bool operator==(const months_type & rhs) const;
```
5.

```
bool operator!=(const months_type & rhs) const;
```
6.

```
months_type operator+(const months_type & rhs) const;
```
7.

```
months_type & operator+=(const months_type & rhs);
```
8.

```
months_type operator-(const months_type & rhs) const;
```
9.

```
months_type & operator-=(const months_type & rhs);
```
10.

```
months_type operator*(const int_type rhs) const;
```
11.

```
months_type & operator*=(const int_type rhs);
```
12.

```
months_type operator/(const int_type rhs) const;
```
13.

```
months_type & operator/=(const int_type rhs);
```
14.

```
months_type operator+(const years_type & y) const;
```
15.

```
months_type & operator+=(const years_type & y);
```
16.

```
months_type operator-(const years_type & y) const;
```
17.

```
months_type & operator-=(const years_type & y);
```

Class template years_duration

boost::date_time::years_duration — additional duration type that represents a logical year

Synopsis

```
// In header: <boost/date_time/date_duration_types.hpp>

template<typename base_config>
class years_duration {
public:
    // construct/copy/destruct
    years_duration(int_rep);
    years_duration(special_values);

    // public member functions
    int_rep number_of_years() const;
    duration_type get_neg_offset(const date_type &) const;
    duration_type get_offset(const date_type &) const;
    bool operator==(const years_type &) const;
    bool operator!=(const years_type &) const;
    years_type operator+(const years_type &) const;
    years_type & operator+=(const years_type &);
    years_type operator-(const years_type &) const;
    years_type & operator-=(const years_type &);
    years_type operator*(const int_type) const;
    years_type & operator*=(const int_type);
    years_type operator/(const int_type) const;
    years_type & operator/=(const int_type);
    months_type operator+(const months_type &) const;
    months_type operator-(const months_type &) const;
};
```

Description

A logical year enables things like: "date(2002,Mar,2) + years(2) -> 2004-Mar-2". If the date is a last day-of-the-month, the result will also be a last-day-of-the-month (ie date(2001-Feb-28) + years(3) -> 2004-Feb-29).

years_duration public construct/copy/destruct

1. `years_duration(int_rep num);`
2. `years_duration(special_values sv);`

years_duration public member functions

1. `int_rep number_of_years() const;`
2. `duration_type get_neg_offset(const date_type & d) const;`
returns a negative duration
3. `duration_type get_offset(const date_type & d) const;`
4. `bool operator==(const years_type & rhs) const;`

5. `bool operator!=(const years_type & rhs) const;`
6. `years_type operator+(const years_type & rhs) const;`
7. `years_type & operator+=(const years_type & rhs);`
8. `years_type operator-(const years_type & rhs) const;`
9. `years_type & operator-=(const years_type & rhs);`
10. `years_type operator*(const int_type rhs) const;`
11. `years_type & operator*=(const int_type rhs);`
12. `years_type operator/(const int_type rhs) const;`
13. `years_type & operator/=(const int_type rhs);`
14. `months_type operator+(const months_type & m) const;`
15. `months_type operator-(const months_type & m) const;`

Header `<boost/date_time/date_facet.hpp>`

```
namespace boost {  
    namespace date_time {  
        template<typename date_type, typename CharT,  
                typename OutItrT = std::ostreambuf_iterator<CharT, std::char_traits<CharT> > >  
            class date_facet;  
        template<typename date_type, typename CharT,  
                typename InItrT = std::istreambuf_iterator<CharT, std::char_traits<CharT> > >  
            class date_input_facet;  
    }  
}
```

Class template `date_facet`

`boost::date_time::date_facet`

Synopsis

```
// In header: <boost/date_time/date_facet.hpp>

template<typename date_type, typename CharT,
        typename OutItrT = std::ostreambuf_iterator<CharT, std::char_traits<CharT> > >
class date_facet : public facet {
public:
    // types
    typedef date_type::duration_type          duration_type;
    typedef date_type::day_of_week_type       day_of_week_type;
    typedef date_type::day_type               day_type;
    typedef 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;
    typedef date_generator_formatter< date_type, CharT > date_gen_formatter_type;
    typedef partial_date< date_type >         partial_date_type;
    typedef nth_kday_of_month< date_type >     nth_kday_type;
    typedef first_kday_of_month< date_type >   first_kday_type;
    typedef last_kday_of_month< date_type >    last_kday_type;
    typedef first_kday_after< date_type >      kday_after_type;
    typedef first_kday_before< date_type >     kday_before_type;

    // construct/copy/destruct
    explicit date_facet(::size_t = 0);
    explicit date_facet(const char_type *, const input_collection_type &,
                        ::size_t = 0);
    explicit date_facet(const char_type *,
                        period_formatter_type = period_formatter_type(),
                        special_values_formatter_type = special_values_formatter_type(),
                        date_gen_formatter_type = date_gen_formatter_type(),
                        ::size_t = 0);

    // public member functions
    std::locale::id & __get_id(void) const;
    void format(const char_type *const);
    void set_iso_format();
    void set_iso_extended_format();
    void month_format(const char_type *const);
    void weekday_format(const char_type *const);
    void period_formatter(period_formatter_type);
    void special_values_formatter(const special_values_formatter_type &);
    void short_weekday_names(const input_collection_type &);
    void long_weekday_names(const input_collection_type &);
    void short_month_names(const input_collection_type &);
    void long_month_names(const input_collection_type &);
    void date_gen_phrase_strings(const input_collection_type &,
                                typename date_gen_formatter_type::phrase_elements = date_gen_formatter_type::first);
    OutItrT put(OutItrT, std::ios_base &, char_type, const date_type &) const;
    OutItrT put(OutItrT, std::ios_base &, char_type, const duration_type &) const;
    OutItrT put(OutItrT, std::ios_base &, char_type, const month_type &) const;
    OutItrT put(OutItrT, std::ios_base &, char_type, const day_type &) const;
    OutItrT put(OutItrT, std::ios_base &, char_type, const day_of_week_type &) const;
    OutItrT put(OutItrT, std::ios_base &, char_type, const period_type &) const;
    OutItrT put(OutItrT, std::ios_base &, char_type, const partial_date_type &) const;
    OutItrT put(OutItrT, std::ios_base &, char_type, const nth_kday_type &) const;
    OutItrT put(OutItrT, std::ios_base &, char_type, const first_kday_type &) const;
```

```

OutItrT put(OutItrT, std::ios_base &, char_type, const last_kday_type &) const;
OutItrT put(OutItrT, std::ios_base &, char_type, const kday_before_type &) const;
OutItrT put(OutItrT, std::ios_base &, char_type, const kday_after_type &) const;

// protected member functions
OutItrT do_put_special(OutItrT, std::ios_base &, char_type,
                      const boost::date_time::special_values) const;
OutItrT do_put_tm(OutItrT, std::ios_base &, char_type, const tm &,
                  string_type) const;

// public data members
static const char_type long_weekday_format;
static const char_type short_weekday_format;
static const char_type long_month_format;
static const char_type short_month_format;
static const char_type default_period_separator;
static const char_type standard_format_specifier;
static const char_type iso_format_specifier;
static const char_type iso_format_extended_specifier;
static const char_type default_date_format;
static std::locale::id id;
};

```

Description

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

date_facet public construct/copy/destruct

1.

```
explicit date_facet(::size_t a_ref = 0);
```
2.

```
explicit date_facet(const char_type * format_str,
                  const input_collection_type & short_names,
                  ::size_t ref_count = 0);
```
3.

```
explicit date_facet(const char_type * format_str,
                  period_formatter_type per_formatter = period_formatter_type(),
                  special_values_formatter_type sv_formatter = special_values_formatter_type(),
                  date_gen_formatter_type dg_formatter = date_gen_formatter_type(),
                  ::size_t ref_count = 0);
```

date_facet public member functions

1. `std::locale::id & __get_id(void) const;`
2. `void format(const char_type *const format_str);`
3. `void set_iso_format();`
4. `void set_iso_extended_format();`
5. `void month_format(const char_type *const format_str);`
6. `void weekday_format(const char_type *const format_str);`
7. `void period_formatter(period_formatter_type per_formatter);`
8. `void special_values_formatter(const special_values_formatter_type & svf);`
9. `void short_weekday_names(const input_collection_type & short_names);`
10. `void long_weekday_names(const input_collection_type & long_names);`
11. `void short_month_names(const input_collection_type & short_names);`
12. `void long_month_names(const input_collection_type & long_names);`
13. `void date_gen_phrase_strings(const input_collection_type & new_strings,
 typename date_gen_formatter_type::phrase_elements beg_pos =
 date_gen_formatter_type::first);`
14. `OutItrT put(OutItrT next, std::ios_base & a_ios, char_type fill_char,
 const date_type & d) const;`
15. `OutItrT put(OutItrT next, std::ios_base & a_ios, char_type fill_char,
 const duration_type & dd) const;`

16. `OutItrT put(OutItrT next, std::ios_base & a_ios, char_type fill_char,
const month_type & m) const;`

17. `OutItrT put(OutItrT next, std::ios_base & a_ios, char_type fill_char,
const day_type & day) const;`

puts the day of month

18. `OutItrT put(OutItrT next, std::ios_base & a_ios, char_type fill_char,
const day_of_week_type & dow) const;`

19. `OutItrT put(OutItrT next, std::ios_base & a_ios, char_type fill_char,
const period_type & p) const;`

20. `OutItrT put(OutItrT next, std::ios_base & a_ios, char_type fill_char,
const partial_date_type & pd) const;`

21. `OutItrT put(OutItrT next, std::ios_base & a_ios, char_type fill_char,
const nth_kday_type & nkd) const;`

22. `OutItrT put(OutItrT next, std::ios_base & a_ios, char_type fill_char,
const first_kday_type & fkd) const;`

23. `OutItrT put(OutItrT next, std::ios_base & a_ios, char_type fill_char,
const last_kday_type & lkd) const;`

24. `OutItrT put(OutItrT next, std::ios_base & a_ios, char_type fill_char,
const kday_before_type & fkb) const;`

25. `OutItrT put(OutItrT next, std::ios_base & a_ios, char_type fill_char,
const kday_after_type & fka) const;`

date_facet protected member functions

1. `OutItrT do_put_special(OutItrT next, std::ios_base &, char_type,
const boost::date_time::special_values sv) const;`

2. `OutItrT do_put_tm(OutItrT next, std::ios_base & a_ios, char_type fill_char,
const tm & tm_value, string_type a_format) const;`

Class template date_input_facet

boost::date_time::date_input_facet — Input facet.

Synopsis

```
// In header: <boost/date_time/date_facet.hpp>

template<typename date_type, typename CharT,
        typename InItrT = std::istreambuf_iterator<CharT, std::char_traits<CharT> > >
class date_input_facet : public facet {
public:
    // types
    typedef date_type::duration_type          duration_type;          ↵

    typedef date_type::day_of_week_type       day_of_week_type;       ↵

    typedef date_type::day_type               day_type;               ↵

    typedef date_type::month_type             month_type;             ↵

    typedef date_type::year_type              year_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_parser< date_type, CharT >        period_parser_type;  ↵

    typedef boost::date_time::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;  ↵

    typedef date_generator_parser< date_type, CharT >                date_gen_parser_type;    ↵

    typedef partial_date< date_type >                                partial_date_type;      ↵

    typedef nth_kday_of_month< date_type >                            nth_kday_type;        ↵

    typedef first_kday_of_month< date_type >                          first_kday_type;      ↵

    typedef last_kday_of_month< date_type >                           last_kday_type;       ↵

    typedef first_kday_after< date_type >                             kday_after_type;      ↵

    typedef first_kday_before< date_type >                            kday_before_type;     ↵

    // construct/copy/destruct
    explicit date_input_facet(::size_t = 0);
    explicit date_input_facet(const string_type &, ::size_t = 0);
    explicit date_input_facet(const string_type &,
                              const format_date_parser_type &,
                              const special_values_parser_type &,
                              const period_parser_type &,
                              const date_gen_parser_type &, ::size_t = 0);

    // public member functions
    void format(const char_type *const);
```

```

void set_iso_format();
void set_iso_extended_format();
void month_format(const char_type *const);
void weekday_format(const char_type *const);
void year_format(const char_type *const);
void period_parser(period_parser_type);
void short_weekday_names(const input_collection_type &);
void long_weekday_names(const input_collection_type &);
void short_month_names(const input_collection_type &);
void long_month_names(const input_collection_type &);
void date_gen_element_strings(const input_collection_type &);
void date_gen_element_strings(const string_type &, const string_type &,
                             const string_type &, const string_type &,
                             const string_type &, const string_type &,
                             const string_type &);
void special_values_parser(special_values_parser_type);
InItrT get(InItrT &, InItrT &, std::ios_base &, date_type &) const;
InItrT get(InItrT &, InItrT &, std::ios_base &, month_type &) const;
InItrT get(InItrT &, InItrT &, std::ios_base &, day_of_week_type &) const;
InItrT get(InItrT &, InItrT &, std::ios_base &, day_type &) const;
InItrT get(InItrT &, InItrT &, std::ios_base &, year_type &) const;
InItrT get(InItrT &, InItrT &, std::ios_base &, duration_type &) const;
InItrT get(InItrT &, InItrT &, std::ios_base &, period_type &) const;
InItrT get(InItrT &, InItrT &, std::ios_base &, nth_kday_type &) const;
InItrT get(InItrT &, InItrT &, std::ios_base &, partial_date_type &) const;
InItrT get(InItrT &, InItrT &, std::ios_base &, first_kday_type &) const;
InItrT get(InItrT &, InItrT &, std::ios_base &, last_kday_type &) const;
InItrT get(InItrT &, InItrT &, std::ios_base &, kday_before_type &) const;
InItrT get(InItrT &, InItrT &, std::ios_base &, kday_after_type &) const;

// public data members
static const char_type long_weekday_format;
static const char_type short_weekday_format;
static const char_type long_month_format;
static const char_type short_month_format;
static const char_type four_digit_year_format;
static const char_type two_digit_year_format;
static const char_type default_period_separator;
static const char_type standard_format_specifier;
static const char_type iso_format_specifier;
static const char_type iso_format_extended_specifier;
static const char_type default_date_format;
static std::locale::id id;
};

```

Description

date_input_facet public construct/copy/destruct

1. `explicit date_input_facet(::size_t a_ref = 0);`
2. `explicit date_input_facet(const string_type & format_str, ::size_t a_ref = 0);`


```
3. explicit date_input_facet(const string_type & format_str,
                           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);
```

date_input_facet public member functions

```
1. void format(const char_type *const format_str);
```

```
2. void set_iso_format();
```

```
3. void set_iso_extended_format();
```

```
4. void month_format(const char_type *const format_str);
```

```
5. void weekday_format(const char_type *const format_str);
```

```
6. void year_format(const char_type *const format_str);
```

```
7. void period_parser(period_parser_type per_parser);
```

```
8. void short_weekday_names(const input_collection_type & weekday_names);
```

```
9. void long_weekday_names(const input_collection_type & weekday_names);
```

```
10. void short_month_names(const input_collection_type & month_names);
```

```
11. void long_month_names(const input_collection_type & month_names);
```

```
12. void date_gen_element_strings(const input_collection_type & col);
```

13.

```
void date_gen_element_strings(const string_type & first,
                             const string_type & second,
                             const string_type & third,
                             const string_type & fourth,
                             const string_type & fifth,
                             const string_type & last,
                             const string_type & before,
                             const string_type & after,
                             const string_type & of);
```
14.

```
void special_values_parser(special_values_parser_type sv_parser);
```
15.

```
InItrT get(InItrT & from, InItrT & to, std::ios_base &, date_type & d) const;
```
16.

```
InItrT get(InItrT & from, InItrT & to, std::ios_base &, month_type & m) const;
```
17.

```
InItrT get(InItrT & from, InItrT & to, std::ios_base &, day_of_week_type & wd) const;
```
18.

```
InItrT get(InItrT & from, InItrT & to, std::ios_base &, day_type & d) const;
```

Expects 1 or 2 digit day range: 1-31.
19.

```
InItrT get(InItrT & from, InItrT & to, std::ios_base &, year_type & y) const;
```
20.

```
InItrT get(InItrT & from, InItrT & to, std::ios_base & a_ios,
           duration_type & dd) const;
```
21.

```
InItrT get(InItrT & from, InItrT & to, std::ios_base & a_ios, period_type & p) const;
```
22.

```
InItrT get(InItrT & from, InItrT & to, std::ios_base & a_ios,
           nth_kday_type & nkd) const;
```
23.

```
InItrT get(InItrT & from, InItrT & to, std::ios_base & a_ios,
           partial_date_type & pd) const;
```
24.

```
InItrT get(InItrT & from, InItrT & to, std::ios_base & a_ios,
           first_kday_type & fkd) const;
```
25.

```
InItrT get(InItrT & from, InItrT & to, std::ios_base & a_ios,
           last_kday_type & lkd) const;
```

```
26. InItrT get(InItrT & from, InItrT & to, std::ios_base & a_ios,
           kday_before_type & fkb) const;
```

```
27. InItrT get(InItrT & from, InItrT & to, std::ios_base & a_ios,
           kday_after_type & fka) const;
```

Header <boost/date_time/date_format_simple.hpp>

```
namespace boost {
    namespace date_time {
        template<typename charT> class simple_format;

        template<> class simple_format<wchar_t>;
    }
}
```

Class template simple_format

boost::date_time::simple_format — Class to provide simple basic formatting rules.

Synopsis

```
// In header: <boost/date_time/date_format_simple.hpp>

template<typename charT>
class simple_format {
public:

    // public static functions
    static const charT * not_a_date();
    static const charT * pos_infinity();
    static const charT * neg_infinity();
    static month_format_spec month_format();
    static ymd_order_spec date_order();
    static bool has_date_sep_chars();
    static charT year_sep_char();
    static charT month_sep_char();
    static charT day_sep_char();
    static charT hour_sep_char();
    static charT minute_sep_char();
    static charT second_sep_char();
};
```

Description

simple_format public static functions

```
1. static const charT * not_a_date();
```

String used printed is date is invalid.

```
2. static const charT * pos_infinity();
```

String used to for positive infinity value.

3.

```
static const charT * neg_infinity();
```

String used to for positive infinity value.

4.

```
static month_format_spec month_format();
```

Describe month format.

5.

```
static ymd_order_spec date_order();
```

6.

```
static bool has_date_sep_chars();
```

This format uses '-' to separate date elements.

7.

```
static charT year_sep_char();
```

Char to sep?

8.

```
static charT month_sep_char();
```

char between year-month

9.

```
static charT day_sep_char();
```

Char to separate month-day.

10.

```
static charT hour_sep_char();
```

char between date-hours

11.

```
static charT minute_sep_char();
```

char between hour and minute

12.

```
static charT second_sep_char();
```

char for second

Specializations

- [Class simple_format<wchar_t>](#)

Class simple_format<wchar_t>

boost::date_time::simple_format<wchar_t> — Specialization of formmating rules for wchar_t.

Synopsis

```
// In header: <boost/date_time/date_format_simple.hpp>
```

```
class simple_format<wchar_t> {
public:

    // public static functions
    static const wchar_t * not_a_date();
    static const wchar_t * pos_infinity();
    static const wchar_t * neg_infinity();
    static month_format_spec month_format();
    static ymd_order_spec date_order();
    static bool has_date_sep_chars();
    static wchar_t year_sep_char();
    static wchar_t month_sep_char();
    static wchar_t day_sep_char();
    static wchar_t hour_sep_char();
    static wchar_t minute_sep_char();
    static wchar_t second_sep_char();
};
```

Description

simple_format public static functions

1. `static const wchar_t * not_a_date();`

String used printed is date is invalid.

2. `static const wchar_t * pos_infinity();`

String used to for positive infinity value.

3. `static const wchar_t * neg_infinity();`

String used to for positive infinity value.

4. `static month_format_spec month_format();`

Describe month format.

5. `static ymd_order_spec date_order();`

6. `static bool has_date_sep_chars();`

This format uses '-' to separate date elements.

7. `static wchar_t year_sep_char();`

Char to sep?

8.

```
static wchar_t month_sep_char();
```

char between year-month

9.

```
static wchar_t day_sep_char();
```

Char to separate month-day.

10.

```
static wchar_t hour_sep_char();
```

char between date-hours

11.

```
static wchar_t minute_sep_char();
```

char between hour and minute

12.

```
static wchar_t second_sep_char();
```

char for second

Header **<boost/date_time/date_formatting.hpp>**

```
namespace boost {  
    namespace date_time {  
        template<typename month_type, typename format_type, typename charT = char>  
            class month_formatter;  
        template<typename ymd_type, typename format_type, typename charT = char>  
            class ymd_formatter;  
        template<typename date_type, typename format_type, typename charT = char>  
            class date_formatter;  
    }  
}
```

Class template month_formatter

boost::date_time::month_formatter — Formats a month as as string into an ostream.

Synopsis

```
// In header: <boost/date_time/date_formatting.hpp>  
  
template<typename month_type, typename format_type, typename charT = char>  
class month_formatter {  
public:  
  
    // public static functions  
    static ostream_type & format_month(const month_type &, ostream_type &);  
    static std::ostream & format_month(const month_type &, std::ostream &);  
};
```

Description

`month_formatter` public static functions

1.

```
static ostream_type &
format_month(const month_type & month, ostream_type & os);
```

Formats a month as as string into an ostream.

This function demands that `month_type` provide functions for converting to short and long strings if that capability is used.

2.

```
static std::ostream &
format_month(const month_type & month, std::ostream & os);
```

Formats a month as as string into an ostream.

This function demands that `month_type` provide functions for converting to short and long strings if that capability is used.

Class template `ymd_formatter`

`boost::date_time::ymd_formatter` — Convert ymd to a standard string formatting policies.

Synopsis

```
// In header: <boost/date_time/date_formatting.hpp>

template<typename ymd_type, typename format_type, typename charT = char>
class ymd_formatter {
public:

    // public static functions
    static std::basic_string< charT > ymd_to_string(ymd_type);
    static std::string ymd_to_string(ymd_type);
};
```

Description

`ymd_formatter` public static functions

1.

```
static std::basic_string< charT > ymd_to_string(ymd_type ymd);
```

Convert ymd to a standard string formatting policies.

This is standard code for handling date formatting with year-month-day based date information. This function uses the `format_type` to control whether the string will contain separator characters, and if so what the character will be. In addition, it can format the month as either an integer or a string as controlled by the formatting policy

2.

```
static std::string ymd_to_string(ymd_type ymd);
```

Convert ymd to a standard string formatting policies.

This is standard code for handling date formatting with year-month-day based date information. This function uses the `format_type` to control whether the string will contain separator characters, and if so what the character will be. In addition, it can format the month as either an integer or a string as controlled by the formatting policy

Class template `date_formatter`

`boost::date_time::date_formatter` — Convert a date to string using format policies.

Synopsis

```
// In header: <boost/date_time/date_formatting.hpp>

template<typename date_type, typename format_type, typename charT = char>
class date_formatter {
public:
    // types
    typedef std::basic_string< charT > string_type;

    // public static functions
    static string_type date_to_string(date_type);
    static std::string date_to_string(date_type);
};
```

Description

`date_formatter` public static functions

1. `static string_type date_to_string(date_type d);`

Convert to a date to standard string using format policies.

2. `static std::string date_to_string(date_type d);`

Convert to a date to standard string using format policies.

Header `<boost/date_time/date_formatting_limited.hpp>`

Header `<boost/date_time/date_formatting_locales.hpp>`

```
namespace boost {
    namespace date_time {
        template<typename facet_type, typename charT = char>
            class ostream_month_formatter;
        template<typename weekday_type, typename facet_type,
            typename charT = char>
            class ostream_weekday_formatter;
        template<typename ymd_type, typename facet_type, typename charT = char>
            class ostream_ymd_formatter;
        template<typename date_type, typename facet_type, typename charT = char>
            class ostream_date_formatter;
    }
}
```

Class template `ostream_month_formatter`

`boost::date_time::ostream_month_formatter` — Formats a month as a string into an ostream.

Synopsis

```
// In header: <boost/date_time/date_formatting_locales.hpp>

template<typename facet_type, typename charT = char>
class ostream_month_formatter {
public:
    // types
    typedef facet_type::month_type      month_type;
    typedef std::basic_ostream< charT > ostream_type;

    // public static functions
    static void format_month(const month_type &, ostream_type &,
                           const facet_type &);
};
```

Description

ostream_month_formatter public static functions

1.

```
static void format_month(const month_type & month, ostream_type & os,
                        const facet_type & f);
```

Formats a month as as string into an output iterator.

Class template ostream_weekday_formatter

boost::date_time::ostream_weekday_formatter — Formats a weekday.

Synopsis

```
// In header: <boost/date_time/date_formatting_locales.hpp>

template<typename weekday_type, typename facet_type, typename charT = char>
class ostream_weekday_formatter {
public:
    // types
    typedef facet_type::month_type      month_type;
    typedef std::basic_ostream< charT > ostream_type;

    // public static functions
    static void format_weekday(const weekday_type &, ostream_type &,
                              const facet_type &, bool);
};
```

Description

ostream_weekday_formatter public static functions

1.

```
static void format_weekday(const weekday_type & wd, ostream_type & os,
                          const facet_type & f, bool as_long_string);
```

Formats a month as as string into an output iterator.

Class template ostream_ymd_formatter

boost::date_time::ostream_ymd_formatter — Convert ymd to a standard string formatting policies.

Synopsis

```
// In header: <boost/date_time/date_formatting_locales.hpp>

template<typename ymd_type, typename facet_type, typename charT = char>
class ostream_ymd_formatter {
public:
    // types
    typedef ymd_type::month_type          month_type;
    typedef ostream_month_formatter< facet_type, charT > month_formatter_type;
    typedef std::basic_ostream< charT >    ostream_type;
    typedef std::basic_string< charT >     foo_type;

    // public static functions
    static void ymd_put(ymd_type, ostream_type &, const facet_type &);
};
```

Description

ostream_ymd_formatter public static functions

1.

```
static void ymd_put(ymd_type ymd, ostream_type & os, const facet_type & f);
```

Convert ymd to a standard string formatting policies.

This is standard code for handling date formatting with year-month-day based date information. This function uses the format_type to control whether the string will contain separator characters, and if so what the character will be. In addition, it can format the month as either an integer or a string as controlled by the formatting policy

Class template ostream_date_formatter

boost::date_time::ostream_date_formatter — Convert a date to string using format policies.

Synopsis

```
// In header: <boost/date_time/date_formatting_locales.hpp>

template<typename date_type, typename facet_type, typename charT = char>
class ostream_date_formatter {
public:
    // types
    typedef std::basic_ostream< charT > ostream_type;
    typedef date_type::ymd_type        ymd_type;

    // public static functions
    static void date_put(const date_type &, ostream_type &, const facet_type &);
    static void date_put(const date_type &, ostream_type &);
};
```

Description

`ostream_date_formatter` public static functions

1.

```
static void date_put(const date_type & d, ostream_type & os,
                    const facet_type & f);
```

Put date into an ostream.

2.

```
static void date_put(const date_type & d, ostream_type & os);
```

Put date into an ostream.

Header `<boost/date_time/date_generator_formatter.hpp>`

```
namespace boost {
    namespace date_time {
        template<typename date_type, typename CharT,
                typename OutItrT = std::ostreambuf_iterator<CharT, std::char_traits<CharT> > >
                class date_generator_formatter;
    }
}
```

Class template `date_generator_formatter`

`boost::date_time::date_generator_formatter` — Formats date_generators for output.

Synopsis

```
// In header: <boost/date_time/date_generator_formatter.hpp>

template<typename date_type, typename CharT,
        typename OutItrT = std::ostreambuf_iterator<CharT, std::char_traits<CharT> > >
class date_generator_formatter {
public:
    // types
    typedef partial_date< date_type >      partial_date_type;
    typedef nth_kday_of_month< date_type >  nth_kday_type;
    typedef first_kday_of_month< date_type > first_kday_type;
    typedef last_kday_of_month< date_type > last_kday_type;
    typedef first_kday_after< date_type >   kday_after_type;
    typedef first_kday_before< date_type >  kday_before_type;
    typedef CharT                           char_type;
    typedef std::basic_string< char_type >   string_type;
    typedef std::vector< string_type >      collection_type;

    enum phrase_elements { first = 0, second, third, fourth, fifth, last,
                          before, after, of, number_of_phrase_elements };

    // construct/copy/destruct
    date_generator_formatter();
    date_generator_formatter(const string_type &, const string_type &,
                           const string_type &, const string_type &,
                           const string_type &, const string_type &,
                           const string_type &);

    // public member functions
    void elements(const collection_type &, phrase_elements = first);
    template<typename facet_type>
        OutItrT put_partial_date(OutItrT, std::ios_base &, CharT,
                                const partial_date_type &, const facet_type &) const;
    template<typename facet_type>
        OutItrT put_nth_kday(OutItrT, std::ios_base &, CharT,
                              const nth_kday_type &, const facet_type &) const;
    template<typename facet_type>
        OutItrT put_first_kday(OutItrT, std::ios_base &, CharT,
                                const first_kday_type &, const facet_type &) const;
    template<typename facet_type>
        OutItrT put_last_kday(OutItrT, std::ios_base &, CharT,
                               const last_kday_type &, const facet_type &) const;
    template<typename facet_type>
        OutItrT put_kday_before(OutItrT, std::ios_base &, CharT,
                                 const kday_before_type &, const facet_type &) const;
    template<typename facet_type>
        OutItrT put_kday_after(OutItrT, std::ios_base &, CharT,
                                const kday_after_type &, const facet_type &) const;

    // private member functions
    OutItrT put_string(OutItrT, const string_type &) const;

    // public data members
    static const char_type first_string;
    static const char_type second_string;
    static const char_type third_string;
    static const char_type fourth_string;
```

```
static const char_type fifth_string;
static const char_type last_string;
static const char_type before_string;
static const char_type after_string;
static const char_type of_string;
};
```

Description

Formatting of date_generators follows specific orders for the various types of date_generators.

- `partial_date` => "dd Month"
- `nth_day_of_the_week_in_month` => "nth weekday of month"
- `first_day_of_the_week_in_month` => "first weekday of month"
- `last_day_of_the_week_in_month` => "last weekday of month"
- `first_day_of_the_week_after` => "weekday after"
- `first_day_of_the_week_before` => "weekday before" While the order of the elements in these phrases cannot be changed, the elements themselves can be. Weekday and Month get their formats and names from the [date_facet](#). The remaining elements are stored in the [date_generator_formatter](#) and can be customized upon construction or via a member function. The default elements are those shown in the examples above.

`date_generator_formatter` public construct/copy/destruct

1.

```
date_generator_formatter();
```

Default format elements used.

2.

```
date_generator_formatter(const string_type & first_str,
                        const string_type & second_str,
                        const string_type & third_str,
                        const string_type & fourth_str,
                        const string_type & fifth_str,
                        const string_type & last_str,
                        const string_type & before_str,
                        const string_type & after_str,
                        const string_type & of_str);
```

Constructor that allows for a custom set of phrase elements.

`date_generator_formatter` public member functions

1.

```
void elements(const collection_type & new_strings,
             phrase_elements beg_pos = first);
```

Replace the set of phrase elements with those contained in new_strings.

The order of the strings in the given collection is important. They must follow:

- first, second, third, fourth, fifth, last, before, after, of.

It is not necessary to send in a complete set if only a few elements are to be replaced as long as the correct beg_pos is used.

Ex: To keep the default first through fifth elements, but replace the rest with a collection of:

- "final", "prior", "following", "in". The beg_pos of date_generator_formatter::last would be used.

```
2. template<typename facet_type>
    OutItrT put_partial_date(OutItrT next, std::ios_base & a_ios, CharT a_fill,
                           const partial_date_type & pd,
                           const facet_type & facet) const;
```

Put a `partial_date` => "dd Month".

```
3. template<typename facet_type>
    OutItrT put_nth_kday(OutItrT next, std::ios_base & a_ios, CharT a_fill,
                       const nth_kday_type & nkd, const facet_type & facet) const;
```

Put an `nth_day_of_the_week_in_month` => "nth weekday of month".

```
4. template<typename facet_type>
    OutItrT put_first_kday(OutItrT next, std::ios_base & a_ios, CharT a_fill,
                          const first_kday_type & fkd,
                          const facet_type & facet) const;
```

Put a `first_day_of_the_week_in_month` => "first weekday of month".

```
5. template<typename facet_type>
    OutItrT put_last_kday(OutItrT next, std::ios_base & a_ios, CharT a_fill,
                        const last_kday_type & lkd, const facet_type & facet) const;
```

Put a `last_day_of_the_week_in_month` => "last weekday of month".

```
6. template<typename facet_type>
    OutItrT put_kday_before(OutItrT next, std::ios_base & a_ios, CharT a_fill,
                          const kday_before_type & fkb,
                          const facet_type & facet) const;
```

Put a `first_day_of_the_week_before` => "weekday before".

```
7. template<typename facet_type>
    OutItrT put_kday_after(OutItrT next, std::ios_base & a_ios, CharT a_fill,
                        const kday_after_type & fka,
                        const facet_type & facet) const;
```

Put a `first_day_of_the_week_after` => "weekday after".

date_generator_formatter private member functions

```
1. OutItrT put_string(OutItrT next, const string_type & str) const;
```

helper function to put the various member string into stream

Header <boost/date_time/date_generator_parser.hpp>

```
namespace boost {
    namespace date_time {
        template<typename date_type, typename charT> class date_generator_parser;
    }
}
```

Class template date_generator_parser

boost::date_time::date_generator_parser — Class for date_generator parsing.

Synopsis

```
// In header: <boost/date_time/date_generator_parser.hpp>

template<typename date_type, typename charT>
class date_generator_parser {
public:
    // types
    typedef std::basic_string< charT >                string_type;
    typedef std::istreambuf_iterator< charT >          stream_itr_type;
    typedef date_type::month_type                      month_type;
    typedef date_type::day_of_week_type                 day_of_week_type;
    typedef date_type::day_type                        day_type;
    typedef string_parse_tree< charT >                 parse_tree_type;
    typedef parse_tree_type::parse_match_result_type    match_results;
    typedef std::vector< std::basic_string< charT > >    collection_type;
    typedef partial_date< date_type >                  partial_date_type;
    typedef nth_kday_of_month< date_type >              nth_kday_type;
    typedef first_kday_of_month< date_type >            first_kday_type;
    typedef last_kday_of_month< date_type >             last_kday_type;
    typedef first_kday_after< date_type >              kday_after_type;
    typedef first_kday_before< date_type >             kday_before_type;
    typedef charT                                       char_type;

    enum phrase_elements { first = 0, second, third, fourth, fifth, last,
                          before, after, of, number_of_phrase_elements };

    // construct/copy/destruct
    date_generator_parser();
    date_generator_parser(const string_type &, const string_type &,
                          const string_type &, const string_type &,
                          const string_type &, const string_type &,
                          const string_type &);

    // public member functions
    void element_strings(const string_type &, const string_type &,
                        const string_type &, const string_type &,
                        const string_type &, const string_type &,
                        const string_type &);
    void element_strings(const collection_type &);
    template<typename facet_type>
        partial_date_type
        get_partial_date_type(stream_itr_type &, stream_itr_type &,
                              std::ios_base &, const facet_type &) const;
    template<typename facet_type>
        nth_kday_type
        get_nth_kday_type(stream_itr_type &, stream_itr_type &, std::ios_base &,
                          const facet_type &) const;
    template<typename facet_type>
        first_kday_type
        get_first_kday_type(stream_itr_type &, stream_itr_type &, std::ios_base &,
                            const facet_type &) const;
    template<typename facet_type>
        last_kday_type
        get_last_kday_type(stream_itr_type &, stream_itr_type &, std::ios_base &,
                           const facet_type &) const;
```

```

template<typename facet_type>
    kday_before_type
    get_kday_before_type(stream_itr_type &, stream_itr_type &,
        std::ios_base &, const facet_type &) const;
template<typename facet_type>
    kday_after_type
    get_kday_after_type(stream_itr_type &, stream_itr_type &, std::ios_base &,
        const facet_type &) const;

// private member functions
void extract_element(stream_itr_type &, stream_itr_type &,
    typename date_generator_parser::phrase_elements) const;

// public data members
static const char_type first_string;
static const char_type second_string;
static const char_type third_string;
static const char_type fourth_string;
static const char_type fifth_string;
static const char_type last_string;
static const char_type before_string;
static const char_type after_string;
static const char_type of_string;
};

```

Description

The elements of a `date_generator` "phrase" are parsed from the input stream in a particular order. All elements are required and the order in which they appear cannot change, however, the elements themselves can be changed. The default elements and their order are as follows:

- `partial_date` => "dd Month"
- `nth_day_of_the_week_in_month` => "nth weekday of month"
- `first_day_of_the_week_in_month` => "first weekday of month"
- `last_day_of_the_week_in_month` => "last weekday of month"
- `first_day_of_the_week_after` => "weekday after"
- `first_day_of_the_week_before` => "weekday before"

Weekday and Month names and formats are handled via the [date_input_facet](#).

`date_generator_parser` public construct/copy/destruct

1. `date_generator_parser();`

Creates a `date_generator_parser` with the default set of "element_strings".

2. `date_generator_parser(const string_type & first_str,
 const string_type & second_str,
 const string_type & third_str,
 const string_type & fourth_str,
 const string_type & fifth_str,
 const string_type & last_str,
 const string_type & before_str,
 const string_type & after_str,
 const string_type & of_str);`

Creates a `date_generator_parser` using a user defined set of element strings.

`date_generator_parser` public member functions

```
1. void element_strings(const string_type & first_str,
                      const string_type & second_str,
                      const string_type & third_str,
                      const string_type & fourth_str,
                      const string_type & fifth_str,
                      const string_type & last_str,
                      const string_type & before_str,
                      const string_type & after_str,
                      const string_type & of_str);
```

Replace strings that determine nth week for generator.

```
2. void element_strings(const collection_type & col);
```

```
3. template<typename facet_type>
    partial_date_type
    get_partial_date_type(stream_itr_type & sitr, stream_itr_type & stream_end,
                        std::ios_base & a_ios, const facet_type & facet) const;
```

returns `partial_date` parsed from stream

```
4. template<typename facet_type>
    nth_kday_type
    get_nth_kday_type(stream_itr_type & sitr, stream_itr_type & stream_end,
                    std::ios_base & a_ios, const facet_type & facet) const;
```

returns `nth_kday_of_week` parsed from stream

```
5. template<typename facet_type>
    first_kday_type
    get_first_kday_type(stream_itr_type & sitr, stream_itr_type & stream_end,
                      std::ios_base & a_ios, const facet_type & facet) const;
```

returns `first_kday_of_week` parsed from stream

```
6. template<typename facet_type>
    last_kday_type
    get_last_kday_type(stream_itr_type & sitr, stream_itr_type & stream_end,
                     std::ios_base & a_ios, const facet_type & facet) const;
```

returns `last_kday_of_week` parsed from stream

```
7. template<typename facet_type>
    kday_before_type
    get_kday_before_type(stream_itr_type & sitr, stream_itr_type & stream_end,
                       std::ios_base & a_ios, const facet_type & facet) const;
```

returns `first_kday_of_week` parsed from stream

```
8. template<typename facet_type>
    kday_after_type
    get_kday_after_type(stream_itr_type & sitr, stream_itr_type & stream_end,
                        std::ios_base & a_ios, const facet_type & facet) const;
```

returns first_kday_of_week parsed from stream

date_generator_parser private member functions

```
1. void extract_element(stream_itr_type & sitr, stream_itr_type & stream_end,
                       typename date_generator_parser::phrase_elements ele) const;
```

Extracts phrase element from input. Throws ios_base::failure on error.

Header <boost/date_time/date_generators.hpp>

Definition and implementation of date algorithm templates

```
namespace boost {
    namespace date_time {
        template<typename date_type> class year_based_generator;
        template<typename date_type> class partial_date;
        template<typename date_type> class nth_kday_of_month;
        template<typename date_type> class first_kday_of_month;
        template<typename date_type> class last_kday_of_month;
        template<typename date_type> class first_kday_after;
        template<typename date_type> class first_kday_before;

        // Returns nth arg as string. 1 -> "first", 2 -> "second", max is 5.
        BOOST_DATE_TIME_DECL const char * nth_as_str(int n);
        template<typename date_type, typename weekday_type>
            date_type::duration_type
            days_until_weekday(const date_type &, const weekday_type &);
        template<typename date_type, typename weekday_type>
            date_type::duration_type
            days_before_weekday(const date_type &, const weekday_type &);
        template<typename date_type, typename weekday_type>
            date_type next_weekday(const date_type &, const weekday_type &);
        template<typename date_type, typename weekday_type>
            date_type previous_weekday(const date_type &, const weekday_type &);
    }
}
```

Class template year_based_generator

boost::date_time::year_based_generator — Base class for all generators that take a year and produce a date.

Synopsis

```
// In header: <boost/date_time/date_generators.hpp>

template<typename date_type>
class year_based_generator {
public:
    // types
    typedef date_type::calendar_type calendar_type;
    typedef calendar_type::year_type year_type;

    // construct/copy/destruct
    year_based_generator();
    ~year_based_generator();

    // public member functions
    date_type get_date(year_type) const;
    std::string to_string() const;
};
```

Description

This class is a base class for polymorphic function objects that take a year and produce a concrete date.

year_based_generator public construct/copy/destruct

1. `year_based_generator();`
2. `~year_based_generator();`

year_based_generator public member functions

1. `date_type get_date(year_type y) const;`
2. `std::string to_string() const;`

Returns a string for use in a POSIX time_zone string.

Class template partial_date

`boost::date_time::partial_date` — Generates a date by applying the year to the given month and day.

Synopsis

```
// In header: <boost/date_time/date_generators.hpp>

template<typename date_type>
class partial_date :
public boost::date_time::year_based_generator< date_type >
{
public:
    // types
    typedef date_type::calendar_type    calendar_type;
    typedef calendar_type::day_type     day_type;
    typedef calendar_type::month_type   month_type;
    typedef calendar_type::year_type    year_type;
    typedef date_type::duration_type    duration_type;
    typedef duration_type::duration_rep duration_rep;

    // construct/copy/destruct
    partial_date(day_type, month_type);
    partial_date(duration_rep);

    // public member functions
    date_type get_date(year_type) const;
    date_type operator()(year_type) const;
    bool operator==(const partial_date &) const;
    bool operator<(const partial_date &) const;
    month_type month() const;
    day_type day() const;
    std::string to_string() const;
};
```

Description

Example usage:

```
partial_date pd(1, Jan);
partial_date pd2(70);
date d = pd.get_date(2002); //2002-Jan-01
date d2 = pd2.get_date(2002); //2002-Mar-10
```

partial_date public construct/copy/destruct

1. `partial_date(day_type d, month_type m);`
2. `partial_date(duration_rep days);`

Partial date created from number of days into year. Range 1-366.

Allowable values range from 1 to 366. 1=Jan1, 366=Dec31. If argument exceeds range, `partial_date` will be created with closest in-range value. 60 will always be Feb29, if `get_date()` is called with a non-leap year an exception will be thrown

partial_date public member functions

1. `date_type get_date(year_type y) const;`

Return a concrete date when provided with a year specific year.

Will throw an 'invalid_argument' exception if a `partial_date` object, instantiated with Feb-29, has `get_date` called with a non-leap year. Example:

```
partial_date pd(29, Feb);
pd.get_date(2003); // throws invalid_argument exception
pg.get_date(2000); // returns 2000-2-29
```

2. `date_type operator()(year_type y) const;`
3. `bool operator==(const partial_date & rhs) const;`
4. `bool operator<(const partial_date & rhs) const;`
5. `month_type month() const;`
6. `day_type day() const;`
7. `std::string to_string() const;`

Returns string suitable for use in POSIX time zone string.

Returns string formatted with up to 3 digits: Jan-01 == "0" Feb-29 == "58" Dec-31 == "365"

Class template `nth_kday_of_month`

`boost::date_time::nth_kday_of_month` — Useful generator functor for finding holidays.

Synopsis

```
// In header: <boost/date_time/date_generators.hpp>

template<typename date_type>
class nth_kday_of_month :
public boost::date_time::year_based_generator< date_type >
{
public:
    // types
    typedef date_type::calendar_type      calendar_type;
    typedef calendar_type::day_of_week_type day_of_week_type;
    typedef calendar_type::month_type      month_type;
    typedef calendar_type::year_type       year_type;
    typedef date_type::duration_type       duration_type;

    enum week_num { first = 1, second, third, fourth, fifth };

    // construct/copy/destruct
    nth_kday_of_month(week_num, day_of_week_type, month_type);

    // public member functions
    date_type get_date(year_type) const;
    month_type month() const;
    week_num nth_week() const;
    day_of_week_type day_of_week() const;
    const char * nth_week_as_str() const;
    std::string to_string() const;
};
```

Description

Based on the idea in Cal. Calc. for finding holidays that are the 'first Monday of September'. When instantiated with 'fifth' kday of month, the result will be the last kday of month which can be the fourth or fifth depending on the structure of the month.

The algorithm here basically guesses for the first day of the month. Then finds the first day of the correct type. That is, if the first of the month is a Tuesday and it needs Wenesday then we simply increment by a day and then we can add the length of a week until we get to the 'nth kday'. There are probably more efficient algorithms based on using a mod 7, but this one works reasonably well for basic applications.

nth_kday_of_month public construct/copy/destruct

1. `nth_kday_of_month(week_num week_no, day_of_week_type dow, month_type m);`

nth_kday_of_month public member functions

1. `date_type get_date(year_type y) const;`

Return a concrete date when provided with a year specific year.

2. `month_type month() const;`

3. `week_num nth_week() const;`

4. `day_of_week_type day_of_week() const;`

5. `const char * nth_week_as_str() const;`

6. `std::string to_string() const;`

Returns string suitable for use in POSIX time zone string.

Returns a string formatted as "M4.3.0" ==> 3rd Sunday in April.

Class template first_kday_of_month

`boost::date_time::first_kday_of_month` — Useful generator functor for finding holidays and daylight savings.

Synopsis

```
// In header: <boost/date_time/date_generators.hpp>

template<typename date_type>
class first_kday_of_month :
    public boost::date_time::year_based_generator< date_type >
{
public:
    // types
    typedef date_type::calendar_type      calendar_type;
    typedef calendar_type::day_of_week_type day_of_week_type;
    typedef calendar_type::month_type      month_type;
    typedef calendar_type::year_type       year_type;
    typedef date_type::duration_type       duration_type;

    // construct/copy/destruct
    first_kday_of_month(day_of_week_type, month_type);

    // public member functions
    date_type get_date(year_type) const;
    month_type month() const;
    day_of_week_type day_of_week() const;
    std::string to_string() const;
};
```

Description

Similar to `nth_kday_of_month`, but requires less paramters

first_kday_of_month public construct/copy/destruct

1. `first_kday_of_month(day_of_week_type dow, month_type m);`

Specify the first 'Sunday' in 'April' spec.

Parameters:	dow	The day of week, eg: Sunday, Monday, etc
	m	The month of the year, eg: Jan, Feb, Mar, etc

first_kday_of_month public member functions

1. `date_type get_date(year_type year) const;`

Return a concrete date when provided with a year specific year.

2. `month_type month() const;`

3. `day_of_week_type day_of_week() const;`

4. `std::string to_string() const;`

Returns string suitable for use in POSIX time zone string.

Returns a string formatted as "M4.1.0" ==> 1st Sunday in April.

Class template last_kday_of_month

boost::date_time::last_kday_of_month — Calculate something like Last Sunday of January.

Synopsis

```
// In header: <boost/date_time/date_generators.hpp>

template<typename date_type>
class last_kday_of_month :
    public boost::date_time::year_based_generator< date_type >
{
public:
    // types
    typedef date_type::calendar_type      calendar_type;
    typedef calendar_type::day_of_week_type day_of_week_type;
    typedef calendar_type::month_type      month_type;
    typedef calendar_type::year_type       year_type;
    typedef date_type::duration_type       duration_type;

    // construct/copy/destruct
    last_kday_of_month(day_of_week_type, month_type);

    // public member functions
    date_type get_date(year_type) const;
    month_type month() const;
    day_of_week_type day_of_week() const;
    std::string to_string() const;
};
```

Description

Useful generator functor for finding holidays and daylight savings Get the last day of the month and then calculate the difference to the last previous day.

last_kday_of_month public construct/copy/destruct

1. `last_kday_of_month(day_of_week_type dow, month_type m);`

Specify the date spec like last 'Sunday' in 'April' spec.

Parameters: dow The day of week, eg: Sunday, Monday, etc
 m The month of the year, eg: Jan, Feb, Mar, etc

last_kday_of_month public member functions

1. `date_type get_date(year_type year) const;`

Return a concrete date when provided with a year specific year.

2. `month_type month() const;`

3. `day_of_week_type day_of_week() const;`

4. `std::string to_string() const;`

Returns string suitable for use in POSIX time zone string.

Returns a string formatted as "M4.5.0" ==> last Sunday in April.

Class template first_kday_after

boost::date_time::first_kday_after — Calculate something like "First Sunday after Jan 1,2002.

Synopsis

```
// In header: <boost/date_time/date_generators.hpp>

template<typename date_type>
class first_kday_after {
public:
    // types
    typedef date_type::calendar_type      calendar_type;
    typedef calendar_type::day_of_week_type day_of_week_type;
    typedef date_type::duration_type      duration_type;

    // construct/copy/destruct
    first_kday_after(day_of_week_type);

    // public member functions
    date_type get_date(date_type) const;
    day_of_week_type day_of_week() const;
};
```

Description

Date generator that takes a date and finds kday after

```
typedef boost::date_time::first_kday_after<date> firstkdayafter;
firstkdayafter fkaf(Monday);
fkaf.get_date(date(2002, Feb, 1));
```

first_kday_after public construct/copy/destruct

```
1. first_kday_after(day_of_week_type dow);
```

first_kday_after public member functions

```
1. date_type get_date(date_type start_day) const;
```

Return next kday given.

```
2. day_of_week_type day_of_week() const;
```

Class template first_kday_before

boost::date_time::first_kday_before — Calculate something like "First Sunday before Jan 1,2002."

Synopsis

```
// In header: <boost/date_time/date_generators.hpp>

template<typename date_type>
class first_kday_before {
public:
    // types
    typedef date_type::calendar_type      calendar_type;
    typedef calendar_type::day_of_week_type day_of_week_type;
    typedef date_type::duration_type      duration_type;

    // construct/copy/destruct
    first_kday_before(day_of_week_type);

    // public member functions
    date_type get_date(date_type) const;
    day_of_week_type day_of_week() const;
};
```

Description

Date generator that takes a date and finds kday after

```
typedef boost::date_time::first_kday_before<date> firstkdaybefore;
firstkdaybefore fkbfb(Monday);
fkbfb.get_date(date(2002, Feb, 1));
```

first_kday_before public construct/copy/destruct

```
1. first_kday_before(day_of_week_type dow);
```

first_kday_before public member functions

```
1. date_type get_date(date_type start_day) const;
```

Return next kday given.

2. `day_of_week_type day_of_week() const;`

Function template `days_until_weekday`

`boost::date_time::days_until_weekday` — Calculates the number of days until the next weekday.

Synopsis

```
// In header: <boost/date_time/date_generators.hpp>

template<typename date_type, typename weekday_type>
date_type::duration_type
days_until_weekday(const date_type & d, const weekday_type & wd);
```

Description

Calculates the number of days until the next weekday. If the date given falls on a Sunday and the given weekday is Tuesday the result will be 2 days

Function template `days_before_weekday`

`boost::date_time::days_before_weekday` — Calculates the number of days since the previous weekday.

Synopsis

```
// In header: <boost/date_time/date_generators.hpp>

template<typename date_type, typename weekday_type>
date_type::duration_type
days_before_weekday(const date_type & d, const weekday_type & wd);
```

Description

Calculates the number of days since the previous weekday. If the date given falls on a Sunday and the given weekday is Tuesday the result will be 5 days. The answer will be a positive number because Tuesday is 5 days before Sunday, not -5 days before.

Function template `next_weekday`

`boost::date_time::next_weekday` — Generates a date object representing the date of the following weekday from the given date.

Synopsis

```
// In header: <boost/date_time/date_generators.hpp>

template<typename date_type, typename weekday_type>
date_type next_weekday(const date_type & d, const weekday_type & wd);
```

Description

Generates a date object representing the date of the following weekday from the given date. If the date given is 2004-May-9 (a Sunday) and the given weekday is Tuesday then the resulting date will be 2004-May-11.

Function template `previous_weekday`

`boost::date_time::previous_weekday` — Generates a date object representing the date of the previous weekday from the given date.

Synopsis

```
// In header: <boost/date_time/date_generators.hpp>

template<typename date_type, typename weekday_type>
date_type previous_weekday(const date_type & d, const weekday_type & wd);
```

Description

Generates a date object representing the date of the previous weekday from the given date. If the date given is 2004-May-9 (a Sunday) and the given weekday is Tuesday then the resulting date will be 2004-May-4.

Header `<boost/date_time/date_iterator.hpp>`

```
namespace boost {
    namespace date_time {
        template<typename date_type> class date_itr_base;
        template<typename offset_functor, typename date_type> class date_itr;

        // An iterator over dates with varying resolution (day, week, month, year, etc)
        enum date_resolutions { day, week, months, year, decade, century,
                                NumDateResolutions };
    }
}
```

Class template `date_itr_base`

`boost::date_time::date_itr_base` — Base date iterator type.

Synopsis

```
// In header: <boost/date_time/date_iterator.hpp>

template<typename date_type>
class date_itr_base {
public:
    // types
    typedef date_type::duration_type duration_type;
    typedef date_type value_type;
    typedef std::input_iterator_tag iterator_category;

    // construct/copy/destruct
    date_itr_base(date_type);
    ~date_itr_base();

    // public member functions
    date_itr_base & operator++();
    date_itr_base & operator--();
    duration_type get_offset(const date_type &) const;
    duration_type get_neg_offset(const date_type &) const;
    date_type operator*();
    date_type * operator->();
    bool operator<(const date_type &);
    bool operator<=(const date_type &);
    bool operator>(const date_type &);
    bool operator>=(const date_type &);
    bool operator==(const date_type &);
    bool operator!=(const date_type &);
};
```

Description

This class provides the skeleton for the creation of iterators. New and interesting iterators can be created by plugging in a new function that derives the next value from the current state, generation of various types of -based information.

Template Parameters

date_type

The date_type is a concrete date_type. The date_type must define a duration_type and a calendar_type.

date_itr_base public construct/copy/destruct

1. `date_itr_base(date_type d);`

2. `~date_itr_base();`

date_itr_base public member functions

1. `date_itr_base & operator++();`

2. `date_itr_base & operator--();`

3. `duration_type get_offset(const date_type & current) const;`
4. `duration_type get_neg_offset(const date_type & current) const;`
5. `date_type operator*();`
6. `date_type * operator->();`
7. `bool operator<(const date_type & d);`
8. `bool operator<=(const date_type & d);`
9. `bool operator>(const date_type & d);`
10. `bool operator>=(const date_type & d);`
11. `bool operator==(const date_type & d);`
12. `bool operator!=(const date_type & d);`

Class template date_itr

boost::date_time::date_itr — Overrides the base date iterator providing hook for functors.

Synopsis

```
// In header: <boost/date_time/date_iterator.hpp>

template<typename offset_functor, typename date_type>
class date_itr : public boost::date_time::date_itr_base< date_type > {
public:
    // types
    typedef date_type::duration_type duration_type;

    // construct/copy/destruct
    date_itr(date_type, int = 1);

    // private member functions
    duration_type get_offset(const date_type &) const;
    duration_type get_neg_offset(const date_type &) const;
};
```

Description

`date_itr` public construct/copy/destruct

```
1. date_itr(date_type d, int factor = 1);
```

`date_itr` private member functions

```
1. duration_type get_offset(const date_type & current) const;
```

```
2. duration_type get_neg_offset(const date_type & current) const;
```

Header `<boost/date_time/date_names_put.hpp>`

```
namespace boost {  
    namespace date_time {  
        template<typename Config, typename charT = char,  
                typename OutputIterator = std::ostreambuf_iterator<charT> >  
            class date_names_put;  
        template<typename Config, typename charT = char,  
                typename OutputIterator = std::ostreambuf_iterator<charT> >  
            class all_date_names_put;  
    }  
}
```

Class template `date_names_put`

`boost::date_time::date_names_put` — Output facet base class for gregorian dates.

Synopsis

```
// In header: <boost/date_time/date_names_put.hpp>

template<typename Config, typename charT = char,
        typename OutputIterator = std::ostreambuf_iterator<charT> >
class date_names_put : public facet {
public:
    // types
    typedef OutputIterator          iter_type;
    typedef Config::month_type      month_type;
    typedef Config::month_enum      month_enum;
    typedef Config::weekday_enum    weekday_enum;
    typedef Config::special_value_enum special_value_enum;
    typedef std::basic_string< charT > string_type;
    typedef charT                   char_type;

    // construct/copy/destruct
    date_names_put();

    // public member functions
    std::locale::id & __get_id(void) const;
    void put_special_value(iter_type &, special_value_enum) const;
    void put_month_short(iter_type &, month_enum) const;
    void put_month_long(iter_type &, month_enum) const;
    void put_weekday_short(iter_type &, weekday_enum) const;
    void put_weekday_long(iter_type &, weekday_enum) const;
    bool has_date_sep_chars() const;
    void year_sep_char(iter_type &) const;
    void month_sep_char(iter_type &) const;
    void day_sep_char(iter_type &) const;
    ymd_order_spec date_order() const;
    month_format_spec month_format() const;

    // protected member functions
    void do_put_month_short(iter_type &, month_enum) const;
    void do_put_month_long(iter_type &, month_enum) const;
    void do_put_special_value(iter_type &, special_value_enum) const;
    void do_put_weekday_short(iter_type &, weekday_enum) const;
    void do_put_weekday_long(iter_type &, weekday_enum) const;
    bool do_has_date_sep_chars() const;
    void do_year_sep_char(iter_type &) const;
    void do_month_sep_char(iter_type &) const;
    void do_day_sep_char(iter_type &) const;
    ymd_order_spec do_date_order() const;
    month_format_spec do_month_format() const;
    void put_string(iter_type &, const charT *const) const;
    void put_string(iter_type &, const string_type &) const;

    // public data members
    static const char_type default_special_value_names;
    static const char_type separator;
    static std::locale::id id; // Generate storage location for a std::locale::id.
};
```

Description

This class is a base class for date facets used to localize the names of months and the names of days in the week.

Requirements of Config

- define an enumeration month_enum that enumerates the months. The enumeration should be '1' based eg: Jan==1

- define as_short_string and as_long_string

(see langer & kreft p334).

date_names_put public construct/copy/destruct

1. `date_names_put();`

date_names_put public member functions

1. `std::locale::id & __get_id(void) const;`

2. `void put_special_value(iter_type & oitr, special_value_enum sv) const;`

3. `void put_month_short(iter_type & oitr, month_enum moy) const;`

4. `void put_month_long(iter_type & oitr, month_enum moy) const;`

5. `void put_weekday_short(iter_type & oitr, weekday_enum wd) const;`

6. `void put_weekday_long(iter_type & oitr, weekday_enum wd) const;`

7. `bool has_date_sep_chars() const;`

8. `void year_sep_char(iter_type & oitr) const;`

9. `void month_sep_char(iter_type & oitr) const;`

char between year-month

10. `void day_sep_char(iter_type & oitr) const;`

Char to separate month-day.

11. `ymd_order_spec date_order() const;`

Determines the order to put the date elements.

12. `month_format_spec month_format() const;`

Determines if month is displayed as integer, short or long string.

date_names_put protected member functions

1. `void do_put_month_short(iter_type & oitr, month_enum moy) const;`

Default facet implementation uses month_type defaults.

2. `void do_put_month_long(iter_type & oitr, month_enum moy) const;`

Default facet implementation uses month_type defaults.

3. `void do_put_special_value(iter_type & oitr, special_value_enum sv) const;`

Default facet implementation for special value types.

4. `void do_put_weekday_short(iter_type &, weekday_enum) const;`

5. `void do_put_weekday_long(iter_type &, weekday_enum) const;`

6. `bool do_has_date_sep_chars() const;`

7. `void do_year_sep_char(iter_type & oitr) const;`

8. `void do_month_sep_char(iter_type & oitr) const;`

char between year-month

9. `void do_day_sep_char(iter_type & oitr) const;`

Char to separate month-day.

10. `ymd_order_spec do_date_order() const;`

Default for date order.

11. `month_format_spec do_month_format() const;`

Default month format.

12. `void put_string(iter_type & oi, const charT *const s) const;`

13. `void put_string(iter_type & oi, const string_type & s1) const;`

Class template all_date_names_put

boost::date_time::all_date_names_put — A date name output facet that takes an array of char* to define strings.

Synopsis

```
// In header: <boost/date_time/date_names_put.hpp>

template<typename Config, typename charT = char,
        typename OutputIterator = std::ostreambuf_iterator<charT> >
class all_date_names_put :
    public boost::date_time::date_names_put< Config, charT, OutputIterator >
{
public:
    // types
    typedef OutputIterator          iter_type;
    typedef Config::month_enum      month_enum;
    typedef Config::weekday_enum    weekday_enum;
    typedef Config::special_value_enum special_value_enum;

    // construct/copy/destruct
    all_date_names_put(const charT *const, const charT *const,
                      const charT *const, const charT *const,
                      const charT *const, charT = '-',
                      ymd_order_spec = ymd_order_iso,
                      month_format_spec = month_as_short_string);

    // public member functions
    const charT *const get_short_month_names() const;
    const charT *const get_long_month_names() const;
    const charT *const get_special_value_names() const;
    const charT *const get_short_weekday_names() const;
    const charT *const get_long_weekday_names() const;

    // protected member functions
    void do_put_month_short(iter_type &, month_enum) const;
    void do_put_month_long(iter_type &, month_enum) const;
    void do_put_special_value(iter_type &, special_value_enum) const;
    void do_put_weekday_short(iter_type &, weekday_enum) const;
    void do_put_weekday_long(iter_type &, weekday_enum) const;
    void do_month_sep_char(iter_type &) const;
    void do_day_sep_char(iter_type &) const;
    ymd_order_spec do_date_order() const;
    month_format_spec do_month_format() const;
};
```

Description

all_date_names_put public construct/copy/destruct

1.

```
all_date_names_put(const charT *const month_short_names,
                  const charT *const month_long_names,
                  const charT *const special_value_names,
                  const charT *const weekday_short_names,
                  const charT *const weekday_long_names,
                  charT separator_char = '-',
                  ymd_order_spec order_spec = ymd_order_iso,
                  month_format_spec month_format = month_as_short_string);
```

all_date_names_put public member functions

1. `const charT *const get_short_month_names() const;`
2. `const charT *const get_long_month_names() const;`
3. `const charT *const get_special_value_names() const;`
4. `const charT *const get_short_weekday_names() const;`
5. `const charT *const get_long_weekday_names() const;`

all_date_names_put protected member functions

1. `void do_put_month_short(iter_type & oitr, month_enum moy) const;`

Generic facet that takes array of chars.

2. `void do_put_month_long(iter_type & oitr, month_enum moy) const;`

Long month names.

3. `void do_put_special_value(iter_type & oitr, special_value_enum sv) const;`

Special values names.

4. `void do_put_weekday_short(iter_type & oitr, weekday_enum wd) const;`

5. `void do_put_weekday_long(iter_type & oitr, weekday_enum wd) const;`

6. `void do_month_sep_char(iter_type & oitr) const;`

char between year-month

7. `void do_day_sep_char(iter_type & oitr) const;`

Char to separate month-day.

8. `ymd_order_spec do_date_order() const;`

Set the date ordering.

9. `month_format_spec do_month_format() const;`

Set the date ordering.

Header `<boost/date_time/date_parsing.hpp>`

```
namespace boost {
namespace date_time {
    std::string convert_to_lower(std::string);

    // Helper function for parse_date.
    template<typename month_type>
        unsigned short month_str_to_ushort(std::string const & s);
    template<typename charT>
        short find_match(const charT *const *, const charT *const *, short,
                        const std::basic_string< charT > &);
    template<typename date_type>
        date_type parse_date(const std::string &, int = ymd_order_iso);

    // Generic function to parse un delimited date (eg: 20020201)
    template<typename date_type>
        date_type parse_un delimited_date(const std::string & s);
    template<typename date_type, typename iterator_type>
        date_type from_stream_type(iterator_type &, iterator_type const &, char);
    template<typename date_type, typename iterator_type>
        date_type from_stream_type(iterator_type &, iterator_type const &,
                                    std::string const &);
    template<typename date_type, typename iterator_type>
        date_type from_stream_type(iterator_type &, iterator_type const &,
                                    wchar_t);
    template<typename date_type, typename iterator_type>
        date_type from_stream_type(iterator_type &, iterator_type const &,
                                    std::wstring const &);

    // function called by wrapper functions: date_period_from_(w)string()
    template<typename date_type, typename charT>
        period< date_type, typename date_type::duration_type >
        from_simple_string_type(const std::basic_string< charT > & s);
}
}
```

Function `convert_to_lower`

`boost::date_time::convert_to_lower` — A function to replace the `std::transform(, , tolower)` construct.

Synopsis

```
// In header: <boost/date_time/date_parsing.hpp>

std::string convert_to_lower(std::string inp);
```

Description

This function simply takes a string, and changes all the characters in that string to lowercase (according to the default system locale). In the event that a compiler does not support locales, the old C style `tolower()` is used.

Function template `find_match`

`boost::date_time::find_match` — Find index of a string in either of 2 arrays.

Synopsis

```
// In header: <boost/date_time/date_parsing.hpp>

template<typename charT>
short find_match(const charT *const * short_names,
                const charT *const * long_names, short size,
                const std::basic_string< charT > & s);
```

Description

find_match searches both arrays for a match to 's'. Both arrays must contain 'size' elements. The index of the match is returned. If no match is found, 'size' is returned. Ex. "Jan" returns 0, "Dec" returns 11, "Tue" returns 2. 'size' can be sent in with: (greg_month::max)() (which 12), (greg_weekday::max)() + 1 (which is 7) or date_time::NumSpecialValues

Function template parse_date

boost::date_time::parse_date — Generic function to parse a delimited date (eg: 2002-02-10)

Synopsis

```
// In header: <boost/date_time/date_parsing.hpp>

template<typename date_type>
date_type parse_date(const std::string & s, int order_spec = ymd_order_iso);
```

Description

Accepted formats are: "2003-02-10" or " 2003-Feb-10" or "2003-Februry-10" The order in which the Month, Day, & Year appear in the argument string can be accomodated by passing in the appropriate ymd_order_spec

Function template from_stream_type

boost::date_time::from_stream_type — Helper function for 'date gregorian::from_stream()'.

Synopsis

```
// In header: <boost/date_time/date_parsing.hpp>

template<typename date_type, typename iterator_type>
date_type from_stream_type(iterator_type & beg, iterator_type const & end,
                          char);
```

Description

Creates a string from the iterators that reference the begining & end of a char[] or string. All elements are used in output string

Function template from_stream_type

boost::date_time::from_stream_type — Helper function for 'date gregorian::from_stream()'.

Synopsis

```
// In header: <boost/date_time/date_parsing.hpp>

template<typename date_type, typename iterator_type>
date_type from_stream_type(iterator_type & beg, iterator_type const &,
                           std::string const &);
```

Description

Returns the first string found in the stream referenced by the beginning & end iterators

Function template from_stream_type

boost::date_time::from_stream_type — Helper function for 'date gregorian::from_stream()'.

Synopsis

```
// In header: <boost/date_time/date_parsing.hpp>

template<typename date_type, typename iterator_type>
date_type from_stream_type(iterator_type & beg, iterator_type const & end,
                           wchar_t);
```

Description

Creates a string from the iterators that reference the beginning & end of a wstring. All elements are used in output string

Function template from_stream_type

boost::date_time::from_stream_type — Helper function for 'date gregorian::from_stream()'.

Synopsis

```
// In header: <boost/date_time/date_parsing.hpp>

template<typename date_type, typename iterator_type>
date_type from_stream_type(iterator_type & beg, iterator_type const &,
                           std::wstring const &);
```

Description

Creates a string from the first wstring found in the stream referenced by the beginning & end iterators

Header <boost/date_time/dst_rules.hpp>

Contains template class to provide static dst rule calculations

```

namespace boost {
    namespace date_time {
        template<typename date_type_, typename time_duration_type_>
            class dst_calculator;
        template<typename date_type_, typename time_duration_type_,
                typename dst_traits>
            class dst_calc_engine;
        template<typename date_type_, typename time_duration_type_,
                unsigned int dst_start_offset_minutes = 120,
                short dst_length_minutes = 60>
            class us_dst_rules;
        template<typename date_type_, typename time_duration_type_>
            class null_dst_rules;

        enum time_is_dst_result { is_not_in_dst, is_in_dst, ambiguous,
                                invalid_time_label };
    }
}

```

Class template dst_calculator

boost::date_time::dst_calculator — Dynamic class used to calculate dst transition information.

Synopsis

```

// In header: <boost/date_time/dst_rules.hpp>

template<typename date_type_, typename time_duration_type_>
class dst_calculator {
public:
    // types
    typedef time_duration_type_ time_duration_type;
    typedef date_type_          date_type;

    // public static functions
    static time_is_dst_result
    process_local_dst_start_day(const time_duration_type &, unsigned int, long);
    static time_is_dst_result
    process_local_dst_end_day(const time_duration_type &, unsigned int, long);
    static time_is_dst_result
    local_is_dst(const date_type &, const time_duration_type &,
                const date_type &, const time_duration_type &,
                const date_type &, const time_duration_type &,
                const time_duration_type &);
    static time_is_dst_result
    local_is_dst(const date_type &, const time_duration_type &,
                const date_type &, unsigned int, const date_type &,
                unsigned int, long);
};

```

Description

dst_calculator public static functions

1.

```
static time_is_dst_result
process_local_dst_start_day(const time_duration_type & time_of_day,
                           unsigned int dst_start_offset_minutes,
                           long dst_length_minutes);
```


Check the local time offset when on dst start day.

On this dst transition, the time label between the transition boundary and the boudary + the offset are invalid times. If before the boundary then still not in dst.

Parameters:	dst_length_minutes	Number of minutes to adjust clock forward
	dst_start_offset_minutes	Local day offset for start of dst
	time_of_day	Time offset in the day for the local time

2.

```
static time_is_dst_result
process_local_dst_end_day(const time_duration_type & time_of_day,
                          unsigned int dst_end_offset_minutes,
                          long dst_length_minutes);
```

Check the local time offset when on the last day of dst.

This is the calculation for the DST end day. On that day times prior to the conversion time - dst_length (1 am in US) are still in dst. Times between the above and the switch time are ambiguous. Times after the start_offset are not in dst.

Parameters:	dst_end_offset_minutes	Local time of day for end of dst
	time_of_day	Time offset in the day for the local time

3.

```
static time_is_dst_result
local_is_dst(const date_type & current_day,
             const time_duration_type & time_of_day,
             const date_type & dst_start_day,
             const time_duration_type & dst_start_offset,
             const date_type & dst_end_day,
             const time_duration_type & dst_end_offset,
             const time_duration_type & dst_length_minutes);
```

Calculates if the given local time is dst or not.

Determines if the time is really in DST or not. Also checks for invalid and ambiguous.

Parameters:	current_day	The day to check for dst
	dst_end_day	Ending day of dst for the given locality
	dst_end_offset	Time offset within day given in dst for dst boundary
	dst_start_day	Starting day of dst for the given locality
	dst_start_offset	Time offset within day for dst boundary
	time_of_day	Time offset within the day to check

4.

```
static time_is_dst_result
local_is_dst(const date_type & current_day,
             const time_duration_type & time_of_day,
             const date_type & dst_start_day,
             unsigned int dst_start_offset_minutes,
             const date_type & dst_end_day,
             unsigned int dst_end_offset_minutes, long dst_length_minutes);
```

Calculates if the given local time is dst or not.

Determines if the time is really in DST or not. Also checks for invalid and ambiguous.

Parameters:	current_day	The day to check for dst
	dst_end_day	Ending day of dst for the given locality
	dst_end_offset_minutes	Offset within day given in dst for dst boundary (eg 120 for US which is 02:00:00)
	dst_length_minutes	Length of dst adjusment (eg: 60 for US)
	dst_start_day	Starting day of dst for the given locality
	dst_start_offset_minutes	Offset within day for dst boundary (eg 120 for US which is 02:00:00)
	time_of_day	Time offset within the day to check

Class template `dst_calc_engine`

`boost::date_time::dst_calc_engine` — Compile-time configurable daylight savings time calculation engine.

Synopsis

```
// In header: <boost/date_time/dst_rules.hpp>

template<typename date_type, typename time_duration_type, typename dst_traits>
class dst_calc_engine {
public:
    // types
    typedef date_type::year_type          year_type;
    typedef date_type::calendar_type      calendar_type;
    typedef dst_calculator< date_type, time_duration_type > dstcalc;

    // public static functions
    static time_is_dst_result
    local_is_dst(const date_type &, const time_duration_type &);
    static bool is_dst_boundary_day(date_type);
    static time_duration_type dst_offset();
    static date_type local_dst_start_day(year_type);
    static date_type local_dst_end_day(year_type);
};
```

Description

`dst_calc_engine` public static functions

1.

```
static time_is_dst_result
local_is_dst(const date_type & d, const time_duration_type & td);
```

Calculates if the given local time is dst or not.

Determines if the time is really in DST or not. Also checks for invalid and ambiguous.

2.

```
static bool is_dst_boundary_day(date_type d);
```

3.

```
static time_duration_type dst_offset();
```

The time of day for the dst transition (eg: typically 01:00:00 or 02:00:00)

4.

```
static date_type local_dst_start_day(year_type year);
```

5.

```
static date_type local_dst_end_day(year_type year);
```

Class template `us_dst_rules`

`boost::date_time::us_dst_rules` — Deprecated: Class to calculate dst boundaries for US time zones.

Synopsis

```
// In header: <boost/date_time/dst_rules.hpp>

template<typename date_type_, typename time_duration_type_,
         unsigned int dst_start_offset_minutes = 120,
         short dst_length_minutes = 60>
class us_dst_rules {
public:
    // types
    typedef time_duration_type_          time_duration_type;
    typedef date_type_                  date_type;
    typedef date_type::year_type        year_type;
    typedef date_type::calendar_type    calendar_type;
    typedef date_type::last_kday_of_month< date_type >    lkday;
    typedef date_type::first_kday_of_month< date_type >   fkday;
    typedef date_type::nth_kday_of_month< date_type >     nkday;
    typedef dst_calculator< date_type, time_duration_type > dstcalc;

    // public static functions
    static time_is_dst_result
    local_is_dst(const date_type &, const time_duration_type &);
    static bool is_dst_boundary_day(date_type);
    static date_type local_dst_start_day(year_type);
    static date_type local_dst_end_day(year_type);
    static time_duration_type dst_offset();
};
```

Description

us_dst_rules public static functions

1.

```
static time_is_dst_result
local_is_dst(const date_type & d, const time_duration_type & td);
```

Calculates if the given local time is dst or not.

Determines if the time is really in DST or not. Also checks for invalid and ambiguous.

2.

```
static bool is_dst_boundary_day(date_type d);
```
3.

```
static date_type local_dst_start_day(year_type year);
```
4.

```
static date_type local_dst_end_day(year_type year);
```
5.

```
static time_duration_type dst_offset();
```

Class template null_dst_rules

boost::date_time::null_dst_rules — Used for local time adjustments in places that don't use dst.

Synopsis

```
// In header: <boost/date_time/dst_rules.hpp>

template<typename date_type_, typename time_duration_type_>
class null_dst_rules {
public:
    // types
    typedef time_duration_type_ time_duration_type;
    typedef date_type_          date_type;

    // public static functions
    static time_is_dst_result
    local_is_dst(const date_type &, const time_duration_type &);
    static time_is_dst_result
    utc_is_dst(const date_type &, const time_duration_type &);
    static bool is_dst_boundary_day(date_type);
    static time_duration_type dst_offset();
};
```

Description

null_dst_rules public static functions

1.

```
static time_is_dst_result
local_is_dst(const date_type &, const time_duration_type &);
```

Calculates if the given local time is dst or not.

2.

```
static time_is_dst_result
utc_is_dst(const date_type &, const time_duration_type &);
```

Calculates if the given utc time is in dst.

3.

```
static bool is_dst_boundary_day(date_type d);
```

4.

```
static time_duration_type dst_offset();
```

Header **<boost/date_time/dst_transition_generators.hpp>**

```
namespace boost {
    namespace date_time {
        template<typename date_type> class dst_day_calc_rule;
        template<typename spec> class day_calc_dst_rule;
    }
}
```

Class template dst_day_calc_rule

boost::date_time::dst_day_calc_rule — Defines base interface for calculating start and end date of daylight savings.

Synopsis

```
// In header: <boost/date_time/dst_transition_generators.hpp>

template<typename date_type>
class dst_day_calc_rule {
public:
    // types
    typedef date_type::year_type year_type;

    // construct/copy/destruct
    ~dst_day_calc_rule();

    // public member functions
    date_type start_day(year_type) const;
    std::string start_rule_as_string() const;
    date_type end_day(year_type) const;
    std::string end_rule_as_string() const;
};
```

Description

dst_day_calc_rule public construct/copy/destruct

1. `~dst_day_calc_rule();`

dst_day_calc_rule public member functions

1. `date_type start_day(year_type y) const;`
2. `std::string start_rule_as_string() const;`
3. `date_type end_day(year_type y) const;`
4. `std::string end_rule_as_string() const;`

Class template day_calc_dst_rule

`boost::date_time::day_calc_dst_rule` — Canonical form for a class that provides day rule calculation.

Synopsis

```
// In header: <boost/date_time/dst_transition_generators.hpp>

template<typename spec>
class day_calc_dst_rule :
{
public boost::date_time::dst_day_calc_rule< spec::date_type >
{
public:
    // types
    typedef spec::date_type      date_type;
    typedef date_type::year_type year_type;
    typedef spec::start_rule     start_rule;
    typedef spec::end_rule       end_rule;

    // construct/copy/destruct
    day_calc_dst_rule(start_rule, end_rule);

    // public member functions
    date_type start_day(year_type) const;
    std::string start_rule_as_string() const;
    date_type end_day(year_type) const;
    std::string end_rule_as_string() const;
};
```

Description

This class is used to generate specific sets of dst rules

day_calc_dst_rule public construct/copy/destruct

1. `day_calc_dst_rule(start_rule dst_start, end_rule dst_end);`

day_calc_dst_rule public member functions

1. `date_type start_day(year_type y) const;`
2. `std::string start_rule_as_string() const;`
3. `date_type end_day(year_type y) const;`
4. `std::string end_rule_as_string() const;`

Header `<boost/date_time/filetime_functions.hpp>`

Function(s) for converting between a FILETIME structure and a time object. This file is only available on systems that have BOOST_HAS_FTIME defined.

```
namespace boost {
  namespace date_time {
    template<typename TimeT, typename FileTimeT>
      TimeT time_from_ftime(const FileTimeT &);
    namespace winapi {
      struct FILETIME;
      struct SYSTEMTIME;

      typedef FILETIME file_time;
      typedef SYSTEMTIME system_time;

      FILETIME * lpLocalFileTime;
      FILETIME * lpFileTime;
      __declspec(dllimport);
      void get_system_time_as_file_time(file_time & ft);
      template<typename FileTimeT>
        boost::uint64_t file_time_to_microseconds(FileTimeT const &);
    }
  }
}
```

Struct FILETIME

boost::date_time::winapi::FILETIME

Synopsis

```
// In header: <boost/date_time/filetime_functions.hpp>

struct FILETIME {

  // public data members
  boost::uint32_t dwLowDateTime;
  boost::uint32_t dwHighDateTime;
};
```

Struct SYSTEMTIME

boost::date_time::winapi::SYSTEMTIME

Synopsis

```
// In header: <boost/date_time/filetime_functions.hpp>

struct SYSTEMTIME {

    // public data members
    boost::uint16_t wYear;
    boost::uint16_t wMonth;
    boost::uint16_t wDayOfWeek;
    boost::uint16_t wDay;
    boost::uint16_t wHour;
    boost::uint16_t wMinute;
    boost::uint16_t wSecond;
    boost::uint16_t wMilliseconds;
};
```

Global lpLocalFileTime

boost::date_time::winapi::lpLocalFileTime

Synopsis

```
// In header: <boost/date_time/filetime_functions.hpp>

FILETIME * lpLocalFileTime;
```

Global lpFileTime

boost::date_time::winapi::lpFileTime

Synopsis

```
// In header: <boost/date_time/filetime_functions.hpp>

FILETIME * lpFileTime;
```

Function template file_time_to_microseconds

boost::date_time::winapi::file_time_to_microseconds

Synopsis

```
// In header: <boost/date_time/filetime_functions.hpp>

template<typename FileTimeT>
    boost::uint64_t file_time_to_microseconds(FileTimeT const & ft);
```

Description

The function converts file_time into number of microseconds elapsed since 1970-Jan-01



Note

Only dates after 1970-Jan-01 are supported. Dates before will be wrapped.

The function is templated on the `FILETIME` type, so that it can be used with both native `FILETIME` and the ad-hoc `boost::date_time::winapi::file_time` type.

Function template `time_from_ftime`

`boost::date_time::time_from_ftime` — Create a time object from an initialized `FILETIME` struct.

Synopsis

```
// In header: <boost/date_time/filetime_functions.hpp>
```

```
template<typename TimeT, typename FileTimeT>
TimeT time_from_ftime(const FileTimeT & ft);
```

Description

Create a time object from an initialized `FILETIME` struct. A `FILETIME` struct holds 100-nanosecond units (0.0000001). When built with microsecond resolution the `file_time`'s sub second value will be truncated. Nanosecond resolution has no truncation.



Note

The function is templated on the `FILETIME` type, so that it can be used with both native `FILETIME` and the ad-hoc `boost::date_time::winapi::file_time` type.

Header `<boost/date_time/format_date_parser.hpp>`

```
namespace std {
} namespace boost {
    namespace date_time {
        template<typename date_type, typename charT> class format_date_parser;
        template<typename int_type, typename charT>
            int_type fixed_string_to_int(std::istreambuf_iterator< charT > &,
                                        std::istreambuf_iterator< charT > &,
                                        parse_match_result< charT > &,
                                        unsigned int, const charT &);
        template<typename int_type, typename charT>
            int_type fixed_string_to_int(std::istreambuf_iterator< charT > &,
                                        std::istreambuf_iterator< charT > &,
                                        parse_match_result< charT > &,
                                        unsigned int);
        template<typename int_type, typename charT>
            int_type var_string_to_int(std::istreambuf_iterator< charT > &,
                                      const std::istreambuf_iterator< charT > &,
                                      unsigned int);
    }
}
```

Class template `format_date_parser`

`boost::date_time::format_date_parser` — Class with generic date parsing using a format string.

Synopsis

```
// In header: <boost/date_time/format_date_parser.hpp>

template<typename date_type, typename charT>
class format_date_parser {
public:
    // types
    typedef std::basic_string< charT >                string_type;
    typedef std::basic_istream< charT >                stringstream_type;
    typedef std::istreambuf_iterator< charT >          stream_itr_type;
    typedef string_type::const_iterator               const_itr;
    typedef date_type::year_type                     year_type;
    typedef date_type::month_type                     month_type;
    typedef date_type::day_type                       day_type;
    typedef date_type::duration_type                  duration_type;
    typedef date_type::day_of_week_type               day_of_week_type;
    typedef date_type::day_of_year_type               day_of_year_type;
    typedef string_parse_tree< charT >                parse_tree_type;
    typedef parse_tree_type::parse_match_result_type  match_results;
    typedef std::vector< std::basic_string< charT > > input_collection_type;

    // construct/copy/destruct
    format_date_parser(const string_type &, const input_collection_type &,
                      const input_collection_type &,
                      const input_collection_type &);
    format_date_parser(const string_type &, const std::locale &);
    format_date_parser(const format_date_parser< date_type, charT > &);

    // public member functions
    string_type format() const;
    void format(string_type);
    void short_month_names(const input_collection_type &);
    void long_month_names(const input_collection_type &);
    void short_weekday_names(const input_collection_type &);
    void long_weekday_names(const input_collection_type &);
    date_type parse_date(const string_type &, const string_type &,
                        const special_values_parser< date_type, charT > &) const;
    date_type parse_date(std::istreambuf_iterator< charT > &,
                        std::istreambuf_iterator< charT > &,
                        const special_values_parser< date_type, charT > &) const;
    date_type parse_date(std::istreambuf_iterator< charT > &,
                        std::istreambuf_iterator< charT > &, string_type,
                        const special_values_parser< date_type, charT > &) const;
    month_type parse_month(std::istreambuf_iterator< charT > &,
                          std::istreambuf_iterator< charT > &, string_type) const;
    month_type parse_month(std::istreambuf_iterator< charT > &,
                          std::istreambuf_iterator< charT > &, string_type,
                          match_results &) const;
    day_type parse_var_day_of_month(std::istreambuf_iterator< charT > &,
                                   std::istreambuf_iterator< charT > &) const;
    day_type parse_day_of_month(std::istreambuf_iterator< charT > &,
                                std::istreambuf_iterator< charT > &) const;
    day_of_week_type
    parse_weekday(std::istreambuf_iterator< charT > &,
                 std::istreambuf_iterator< charT > &, string_type) const;
    day_of_week_type
    parse_weekday(std::istreambuf_iterator< charT > &,
                 std::istreambuf_iterator< charT > &, string_type,
                 match_results &) const;
```

```

year_type parse_year(std::istreambuf_iterator< charT > &,
                    std::istreambuf_iterator< charT > &, string_type) const;
year_type parse_year(std::istreambuf_iterator< charT > &,
                    std::istreambuf_iterator< charT > &, string_type,
                    match_results &) const;
};

```

Description

The following is the set of recognized format specifiers

- a - Short weekday name
- A - Long weekday name
- b - Abbreviated month name
- B - Full month name
- d - Day of the month as decimal 01 to 31
- j - Day of year as decimal from 001 to 366
- m - Month name as a decimal 01 to 12
- U - Week number 00 to 53 with first Sunday as the first day of week 1?
- w - Weekday as decimal number 0 to 6 where Sunday == 0
- W - Week number 00 to 53 where Monday is first day of week 1
- x - facet default date representation
- y - Year without the century - eg: 04 for 2004
- Y - Year with century

The weekday specifiers (a and A) do not add to the date construction, but they provide a way to skip over the weekday names for formats that provide them.

todo -- Another interesting feature that this approach could provide is an option to fill in any missing fields with the current values from the clock. So if you have m-d the parser would detect the missing year value and fill it in using the clock.

todo -- What to do with the x. x in the classic facet is just bad...

format_date_parser public construct/copy/destruct

1.

```
format_date_parser(const string_type & format_str,
                  const input_collection_type & month_short_names,
                  const input_collection_type & month_long_names,
                  const input_collection_type & weekday_short_names,
                  const input_collection_type & weekday_long_names);
```
2.

```
format_date_parser(const string_type & format_str, const std::locale & locale);
```
3.

```
format_date_parser(const format_date_parser< date_type, charT > & fdp);
```

format_date_parser public member functions

1.

```
string_type format() const;
```
2.

```
void format(string_type format_str);
```
3.

```
void short_month_names(const input_collection_type & month_names);
```
4.

```
void long_month_names(const input_collection_type & month_names);
```
5.

```
void short_weekday_names(const input_collection_type & weekday_names);
```
6.

```
void long_weekday_names(const input_collection_type & weekday_names);
```
7.

```
date_type parse_date(const string_type & value,  
                    const string_type & format_str,  
                    const special_values_parser< date_type, charT > & sv_parser) const;
```
8.

```
date_type parse_date(std::istreambuf_iterator< charT > & sitr,  
                    std::istreambuf_iterator< charT > & stream_end,  
                    const special_values_parser< date_type, charT > & sv_parser) const;
```
9.

```
date_type parse_date(std::istreambuf_iterator< charT > & sitr,  
                    std::istreambuf_iterator< charT > & stream_end,  
                    string_type format_str,  
                    const special_values_parser< date_type, charT > & sv_parser) const;
```

Of all the objects that the `format_date_parser` can parse, only a date can be a special value. Therefore, only `parse_date` checks for special_values.

10.

```
month_type parse_month(std::istreambuf_iterator< charT > & sitr,  
                      std::istreambuf_iterator< charT > & stream_end,  
                      string_type format_str) const;
```

Throws `bad_month` if unable to parse.

11.

```
month_type parse_month(std::istreambuf_iterator< charT > & sitr,  
                      std::istreambuf_iterator< charT > & stream_end,  
                      string_type format_str, match_results & mr) const;
```

Throws `bad_month` if unable to parse.

12.

```
day_type parse_var_day_of_month(std::istreambuf_iterator< charT > & sitr,  
                               std::istreambuf_iterator< charT > & stream_end) const;
```

Expects 1 or 2 digits 1-31. Throws `bad_day_of_month` if unable to parse.

```
13. day_type parse_day_of_month(std::istreambuf_iterator< charT > & sitr,
                             std::istreambuf_iterator< charT > & stream_end) const;
```

Expects 2 digits 01-31. Throws `bad_day_of_month` if unable to parse.

```
14. day_of_week_type
    parse_weekday(std::istreambuf_iterator< charT > & sitr,
                  std::istreambuf_iterator< charT > & stream_end,
                  string_type format_str) const;
```

```
15. day_of_week_type
    parse_weekday(std::istreambuf_iterator< charT > & sitr,
                  std::istreambuf_iterator< charT > & stream_end,
                  string_type format_str, match_results & mr) const;
```

```
16. year_type parse_year(std::istreambuf_iterator< charT > & sitr,
                        std::istreambuf_iterator< charT > & stream_end,
                        string_type format_str) const;
```

throws `bad_year` if unable to parse

```
17. year_type parse_year(std::istreambuf_iterator< charT > & sitr,
                        std::istreambuf_iterator< charT > & stream_end,
                        string_type format_str, match_results & mr) const;
```

throws `bad_year` if unable to parse

Function template `fixed_string_to_int`

`boost::date_time::fixed_string_to_int` — Helper function for parsing fixed length strings into integers.

Synopsis

```
// In header: <boost/date_time/format_date_parser.hpp>

template<typename int_type, typename charT>
int_type fixed_string_to_int(std::istreambuf_iterator< charT > & itr,
                           std::istreambuf_iterator< charT > & stream_end,
                           parse_match_result< charT > & mr,
                           unsigned int length, const charT & fill_char);
```

Description

Will consume 'length' number of characters from stream. Consumed character are transferred to `parse_match_result` struct. Returns `-1` if no number can be parsed or incorrect number of digits in stream.

Function template `fixed_string_to_int`

`boost::date_time::fixed_string_to_int` — Helper function for parsing fixed length strings into integers.

Synopsis

```
// In header: <boost/date_time/format_date_parser.hpp>

template<typename int_type, typename charT>
int_type fixed_string_to_int(std::istreambuf_iterator< charT > & itr,
                           std::istreambuf_iterator< charT > & stream_end,
                           parse_match_result< charT > & mr,
                           unsigned int length);
```

Description

Will consume 'length' number of characters from stream. Consumed character are transfered to `parse_match_result` struct. Returns '-1' if no number can be parsed or incorrect number of digits in stream.

Function template `var_string_to_int`

`boost::date_time::var_string_to_int` — Helper function for parsing varied length strings into integers.

Synopsis

```
// In header: <boost/date_time/format_date_parser.hpp>

template<typename int_type, typename charT>
int_type var_string_to_int(std::istreambuf_iterator< charT > & itr,
                          const std::istreambuf_iterator< charT > & stream_end,
                          unsigned int max_length);
```

Description

Will consume 'max_length' characters from stream only if those characters are digits. Returns '-1' if no number can be parsed. Will not parse a number preceeded by a '+' or '-'.

Header `<boost/date_time/gregorian_calendar.hpp>`

```
namespace boost {
    namespace date_time {
        template<typename ymd_type_, typename date_int_type_>
        class gregorian_calendar_base;
    }
}
```

Class template `gregorian_calendar_base`

`boost::date_time::gregorian_calendar_base` — An implementation of the Gregorian calendar.

Synopsis

```
// In header: <boost/date_time/gregorian_calendar.hpp>

template<typename ymd_type_, typename date_int_type_>
class gregorian_calendar_base {
public:
    // types
    typedef ymd_type_          ymd_type;          // define a type a date split into components
    typedef ymd_type::month_type month_type;       // define a type for representing months
    typedef ymd_type::day_type  day_type;          // define a type for representing days
    typedef ymd_type::year_type year_type;         // Type to hold a stand alone year value (eg: 2002)
    typedef date_int_type_      date_int_type;     // Define the integer type to use for internal calculations.

    // public static functions
    static unsigned short day_of_week(const ymd_type &);
    static int week_number(const ymd_type &);
    static date_int_type day_number(const ymd_type &);
    static date_int_type julian_day_number(const ymd_type &);
    static date_int_type modjulian_day_number(const ymd_type &);
    static ymd_type from_day_number(date_int_type);
    static ymd_type from_julian_day_number(date_int_type);
    static ymd_type from_modjulian_day_number(date_int_type);
    static bool is_leap_year(year_type);
    static unsigned short end_of_month_day(year_type, month_type);
    static ymd_type epoch();
    static unsigned short days_in_week();
};
```

Description

This is a parameterized implementation of a proleptic Gregorian Calendar that can be used in the creation of date systems or just to perform calculations. All the methods of this class are static functions, so the intent is to never create instances of this class.

gregorian_calendar_base public static functions

1. `static unsigned short day_of_week(const ymd_type & ymd);`
2. `static int week_number(const ymd_type & ymd);`
3. `static date_int_type day_number(const ymd_type & ymd);`
4. `static date_int_type julian_day_number(const ymd_type & ymd);`
5. `static date_int_type modjulian_day_number(const ymd_type & ymd);`
6. `static ymd_type from_day_number(date_int_type);`

7.

```
static ymd_type from_julian_day_number(date_int_type);
```
8.

```
static ymd_type from_modjulian_day_number(date_int_type);
```
9.

```
static bool is_leap_year(year_type);
```
10.

```
static unsigned short end_of_month_day(year_type y, month_type m);
```
11.

```
static ymd_type epoch();
```
12.

```
static unsigned short days_in_week();
```

Header **<boost/date_time/int_adapter.hpp>**

```
namespace boost {  
    namespace date_time {  
        template<typename int_type_> class int_adapter;  
        template<typename charT, typename traits, typename int_type>  
            std::basic_ostream< charT, traits > &  
            operator<< (std::basic_ostream< charT, traits > &,  
                        const int_adapter< int_type > &);  
    }  
}
```

Class template **int_adapter**

boost::date_time::int_adapter — Adapter to create integer types with +-infinity, and not a value.

Synopsis

```
// In header: <boost/date_time/int_adapter.hpp>

template<typename int_type_>
class int_adapter {
public:
    // types
    typedef int_type_ int_type;

    // construct/copy/destruct
    int_adapter(int_type);

    // public member functions
    bool is_infinity() const;
    bool is_pos_infinity() const;
    bool is_neg_infinity() const;
    bool is_nan() const;
    bool is_special() const;
    bool operator==(const int_adapter &) const;
    bool operator==(const int &) const;
    bool operator!=(const int_adapter &) const;
    bool operator!=(const int &) const;
    bool operator<(const int_adapter &) const;
    bool operator<(const int &) const;
    bool operator>(const int_adapter &) const;
    int_type as_number() const;
    special_values as_special() const;
    template<typename rhs_type>
        int_adapter operator+(const int_adapter< rhs_type > &) const;
    int_adapter operator+(const int_type) const;
    template<typename rhs_type>
        int_adapter operator-(const int_adapter< rhs_type > &) const;
    int_adapter operator-(const int_type) const;
    int_adapter operator*(const int_adapter &) const;
    int_adapter operator*(const int) const;
    int_adapter operator/(const int_adapter &) const;
    int_adapter operator/(const int) const;
    int_adapter operator%(const int_adapter &) const;
    int_adapter operator%(const int) const;

    // public static functions
    static bool has_infinity();
    static const int_adapter pos_infinity();
    static const int_adapter neg_infinity();
    static const int_adapter not_a_number();
    static int_adapter max BOOST_PREVENT_MACRO_SUBSTITUTION();
    static int_adapter min BOOST_PREVENT_MACRO_SUBSTITUTION();
    static int_adapter from_special(special_values);
    static bool is_inf(int_type);
    static bool is_neg_inf(int_type);
    static bool is_pos_inf(int_type);
    static bool is_not_a_number(int_type);
    static special_values to_special(int_type);
    static int_type maxcount();

    // private member functions
    int compare(const int_adapter &) const;
    int_adapter mult_div_specials(const int_adapter &) const;
    int_adapter mult_div_specials(const int &) const;
};
```

Description

This class is used internally in counted date/time representations. It adds the floating point like features of infinities and not a number. It also provides mathematical operations with consideration to special values following these rules:

```
+infinity - infinity == Not A Number (NAN)
infinity * non-zero == infinity
infinity * zero      == NAN
+infinity * -integer == -infinity
infinity / infinity  == NAN
infinity * infinity  == infinity
*
```

`int_adapter` public construct/copy/destruct

```
1. int_adapter(int_type v);
```

`int_adapter` public member functions

```
1. bool is_infinity() const;
```

```
2. bool is_pos_infinity() const;
```

```
3. bool is_neg_infinity() const;
```

```
4. bool is_nan() const;
```

```
5. bool is_special() const;
```

```
6. bool operator==(const int_adapter & rhs) const;
```

```
7. bool operator==(const int & rhs) const;
```

```
8. bool operator!=(const int_adapter & rhs) const;
```

```
9. bool operator!=(const int & rhs) const;
```

```
10. bool operator<(const int_adapter & rhs) const;
```

```
11. bool operator<(const int & rhs) const;
```

12. `bool operator>(const int_adapter & rhs) const;`

13. `int_type as_number() const;`

14. `special_values as_special() const;`

Returns either special value type or is_not_special.

15. `template<typename rhs_type>
int_adapter operator+(const int_adapter< rhs_type > & rhs) const;`

Operator allows for adding dissimilar `int_adapter` types. The return type will match that of the the calling object's type

16. `int_adapter operator+(const int_type rhs) const;`

17. `template<typename rhs_type>
int_adapter operator-(const int_adapter< rhs_type > & rhs) const;`

Operator allows for subtracting dissimilar `int_adapter` types. The return type will match that of the the calling object's type

18. `int_adapter operator-(const int_type rhs) const;`

19. `int_adapter operator*(const int_adapter & rhs) const;`

20. `int_adapter operator*(const int rhs) const;`

Provided for cases when automatic conversion from 'int' to 'int_adapter' causes incorrect results.

21. `int_adapter operator/(const int_adapter & rhs) const;`

22. `int_adapter operator/(const int rhs) const;`

Provided for cases when automatic conversion from 'int' to 'int_adapter' causes incorrect results.

23. `int_adapter operator%(const int_adapter & rhs) const;`

24. `int_adapter operator%(const int rhs) const;`

Provided for cases when automatic conversion from 'int' to 'int_adapter' causes incorrect results.

`int_adapter` public static functions

1. `static bool has_infinity();`

2.

```
static const int_adapter pos_infinity();
```
3.

```
static const int_adapter neg_infinity();
```
4.

```
static const int_adapter not_a_number();
```
5.

```
static int_adapter max BOOST_PREVENT_MACRO_SUBSTITUTION();
```
6.

```
static int_adapter min BOOST_PREVENT_MACRO_SUBSTITUTION();
```
7.

```
static int_adapter from_special(special_values sv);
```
8.

```
static bool is_inf(int_type v);
```
9.

```
static bool is_neg_inf(int_type v);
```
10.

```
static bool is_pos_inf(int_type v);
```
11.

```
static bool is_not_a_number(int_type v);
```
12.

```
static special_values to_special(int_type v);
```

Returns either special value type or is_not_special.
13.

```
static int_type maxcount();
```

int_adapter private member functions

1.

```
int compare(const int_adapter & rhs) const;
```

returns -1, 0, 1, or 2 if 'this' is <, ==, >, or 'nan comparison' rhs
2.

```
int_adapter mult_div_specials(const int_adapter & rhs) const;
```

Assumes at least 'this' or 'rhs' is a special value.
3.

```
int_adapter mult_div_specials(const int & rhs) const;
```

Assumes 'this' is a special value.

Function template operator<<

boost::date_time::operator<<

Synopsis

```
// In header: <boost/date_time/int_adapter.hpp>

template<typename charT, typename traits, typename int_type>
std::basic_ostream< charT, traits > &
operator<<(std::basic_ostream< charT, traits > & os,
          const int_adapter< int_type > & ia);
```

Description

Expected output is either a numeric representation or a special values representation.

Ex. "12", "+infinity", "not-a-number", etc.

Header <boost/date_time/iso_format.hpp>

```
namespace boost {
  namespace date_time {
    template<typename charT> class iso_format_base;

    template<> class iso_format_base<wchar_t>;

    template<typename charT> class iso_format;
    template<typename charT> class iso_extended_format;
  }
}
```

Class template iso_format_base

boost::date_time::iso_format_base — Class to provide common iso formatting spec.

Synopsis

```
// In header: <boost/date_time/iso_format.hpp>

template<typename charT>
class iso_format_base {
public:

    // public static functions
    static month_format_spec month_format();
    static const charT * not_a_date();
    static const charT * pos_infinity();
    static const charT * neg_infinity();
    static charT year_sep_char();
    static charT month_sep_char();
    static charT day_sep_char();
    static charT hour_sep_char();
    static charT minute_sep_char();
    static charT second_sep_char();
    static charT period_start_char();
    static charT time_start_char();
    static charT week_start_char();
    static charT period_sep_char();
    static charT time_sep_char();
    static charT fractional_time_sep_char();
    static bool is_component_sep(charT);
    static bool is_fractional_time_sep(charT);
    static bool is_timezone_sep(charT);
    static charT element_sep_char();
};
```

Description

iso_format_base public static functions

1. `static month_format_spec month_format();`

Describe month format -- its an integer in iso format.

2. `static const charT * not_a_date();`

String used printed is date is invalid.

3. `static const charT * pos_infinity();`

String used to for positive infinity value.

4. `static const charT * neg_infinity();`

String used to for positive infinity value.

5. `static charT year_sep_char();`

ISO char for a year -- used in durations.

6. `static charT month_sep_char();`

ISO char for a month.

7.

```
static charT day_sep_char();
```

ISO char for a day.

8.

```
static charT hour_sep_char();
```

char for minute

9.

```
static charT minute_sep_char();
```

char for minute

10.

```
static charT second_sep_char();
```

char for second

11.

```
static charT period_start_char();
```

ISO char for a period.

12.

```
static charT time_start_char();
```

Used in time in mixed strings to set start of time.

13.

```
static charT week_start_char();
```

Used in mixed strings to identify start of a week number.

14.

```
static charT period_sep_char();
```

Separators for periods.

15.

```
static charT time_sep_char();
```

Separator for hh:mm:ss.

16.

```
static charT fractional_time_sep_char();
```

Preferred Separator for hh:mm:ss,decimal_fraction.

17.

```
static bool is_component_sep(charT sep);
```

18.

```
static bool is_fractional_time_sep(charT sep);
```

19.

```
static bool is_timezone_sep(charT sep);
```

```
20. static charT element_sep_char();
```

Specializations

- [Class iso_format_base<wchar_t>](#)

Class iso_format_base<wchar_t>

boost::date_time::iso_format_base<wchar_t> — Class to provide common iso formatting spec.

Synopsis

```
// In header: <boost/date_time/iso_format.hpp>

class iso_format_base<wchar_t> {
public:

    // public static functions
    static month_format_spec month_format();
    static const wchar_t * not_a_date();
    static const wchar_t * pos_infinity();
    static const wchar_t * neg_infinity();
    static wchar_t year_sep_char();
    static wchar_t month_sep_char();
    static wchar_t day_sep_char();
    static wchar_t hour_sep_char();
    static wchar_t minute_sep_char();
    static wchar_t second_sep_char();
    static wchar_t period_start_char();
    static wchar_t time_start_char();
    static wchar_t week_start_char();
    static wchar_t period_sep_char();
    static wchar_t time_sep_char();
    static wchar_t fractional_time_sep_char();
    static bool is_component_sep(wchar_t);
    static bool is_fractional_time_sep(wchar_t);
    static bool is_timezone_sep(wchar_t);
    static wchar_t element_sep_char();
};
```

Description

iso_format_base public static functions

1.

```
static month_format_spec month_format();
```

Describe month format -- its an integer in iso format.

2.

```
static const wchar_t * not_a_date();
```

String used printed is date is invalid.

3.

```
static const wchar_t * pos_infinity();
```

String used to for positive infinity value.

4.

```
static const wchar_t * neg_infinity();
```

String used to for positive infinity value.

5.

```
static wchar_t year_sep_char();
```

ISO char for a year -- used in durations.

6.

```
static wchar_t month_sep_char();
```

ISO char for a month.

7.

```
static wchar_t day_sep_char();
```

ISO char for a day.

8.

```
static wchar_t hour_sep_char();
```

char for minute

9.

```
static wchar_t minute_sep_char();
```

char for minute

10.

```
static wchar_t second_sep_char();
```

char for second

11.

```
static wchar_t period_start_char();
```

ISO char for a period.

12.

```
static wchar_t time_start_char();
```

Used in time in mixed strings to set start of time.

13.

```
static wchar_t week_start_char();
```

Used in mixed strings to identify start of a week number.

14.

```
static wchar_t period_sep_char();
```

Separators for periods.

15.

```
static wchar_t time_sep_char();
```

Separator for hh:mm:ss.

16.

```
static wchar_t fractional_time_sep_char();
```

Preferred Separator for hh:mm:ss,decimal_fraction.

- ```
17. static bool is_component_sep(wchar_t sep);

18. static bool is_fractional_time_sep(wchar_t sep);

19. static bool is_timezone_sep(wchar_t sep);

20. static wchar_t element_sep_char();
```

## Class template iso\_format

boost::date\_time::iso\_format — Format description for iso normal YYYYMMDD.

## Synopsis

```
// In header: <boost/date_time/iso_format.hpp>

template<typename charT>
class iso_format : public boost::date_time::iso_format_base< charT > {
public:

 // public static functions
 static bool has_date_sep_chars();
};
```

## Description

### iso\_format public static functions

- ```
1. static bool has_date_sep_chars();
```

The ios standard format doesn't use char separators.

Class template iso_extended_format

boost::date_time::iso_extended_format — Extended format uses separators YYYY-MM-DD.

Synopsis

```
// In header: <boost/date_time/iso_format.hpp>

template<typename charT>
class iso_extended_format : public boost::date_time::iso_format_base< charT > {
public:

    // public static functions
    static bool has_date_sep_chars();
};
```

Description

iso_extended_format public static functions

1.

```
static bool has_date_sep_chars();
```

Extended format needs char separators.

Header **<boost/date_time/local_time_adjustor.hpp>**

Time adjustment calculations for local times

```
namespace boost {
    namespace date_time {
        template<typename time_duration_type, short hours,
                unsigned short minutes = 0>
            class utc_adjustment;
        template<typename time_type, typename dst_rules>
            class dynamic_local_time_adjustor;
        template<typename time_type, typename dst_rules,
                typename utc_offset_rules>
            class static_local_time_adjustor;
        template<typename time_type, short utc_offset, typename dst_rule>
            class local_adjustor;
        void dummy_to_prevent_msvc6_ice();
    }
}
```

Class template **utc_adjustment**

boost::date_time::utc_adjustment — Provides a base offset adjustment from utc.

Synopsis

```
// In header: <boost/date_time/local_time_adjustor.hpp>

template<typename time_duration_type, short hours, unsigned short minutes = 0>
class utc_adjustment {
public:

    // public static functions
    static time_duration_type local_to_utc_base_offset();
    static time_duration_type utc_to_local_base_offset();
};
```

Description

utc_adjustment public static functions

1.

```
static time_duration_type local_to_utc_base_offset();
```
2.

```
static time_duration_type utc_to_local_base_offset();
```

Class template `dynamic_local_time_adjustor`

`boost::date_time::dynamic_local_time_adjustor` — Allow sliding utc adjustment with fixed dst rules.

Synopsis

```
// In header: <boost/date_time/local_time_adjustor.hpp>

template<typename time_type, typename dst_rules>
class dynamic_local_time_adjustor {
public:
    // types
    typedef time_type::time_duration_type time_duration_type;
    typedef time_type::date_type          date_type;

    // construct/copy/destroy
    dynamic_local_time_adjustor(time_duration_type);

    // public member functions
    time_duration_type utc_offset(bool);
};
```

Description

`dynamic_local_time_adjustor` public construct/copy/destroy

1. `dynamic_local_time_adjustor(time_duration_type utc_offset);`

`dynamic_local_time_adjustor` public member functions

1. `time_duration_type utc_offset(bool is_dst);`

Presumes local time.

Class template `static_local_time_adjustor`

`boost::date_time::static_local_time_adjustor` — Embed the rules for local time adjustments at compile time.

Synopsis

```
// In header: <boost/date_time/local_time_adjustor.hpp>

template<typename time_type, typename dst_rules, typename utc_offset_rules>
class static_local_time_adjustor {
public:
    // types
    typedef time_type::time_duration_type time_duration_type;
    typedef time_type::date_type          date_type;

    // public static functions
    static time_duration_type utc_to_local_offset(const time_type &);
    static time_duration_type local_to_utc_offset(const time_type &,
                                                  date_time::dst_flags = date_time::calculate);
};
```

Description

static_local_time_adjustor public static functions

1.

```
static time_duration_type utc_to_local_offset(const time_type & t);
```

Calculates the offset from a utc time to local based on dst and utc offset.

The logic is as follows. Starting with UTC time use the offset to create a label for a non-dst adjusted local time. Then call `dst_rules::local_is_dst` with the non adjust local time. The results of this function will either unambiguously decide that the initial local time is in dst or return an illegal or ambiguous result. An illegal result only occurs at the end of dst (where labels are skipped) and indicates that dst has ended. An ambiguous result means that we need to recheck by making a dst adjustment and then rechecking. If the dst offset is added to the utc time and the recheck proves non-ambiguous then we are past the boundary. If it is still ambiguous then we are ahead of the boundary and dst is still in effect.

TODO -- check if all dst offsets are positive. If not then the algorithm needs to check for this and reverse the illegal/ambiguous logic.

Parameters: `t` UTC time to calculate offset to local time This adjustment depends on the following observations about the workings of the DST boundary offset. Since UTC time labels are monotonically increasing we can determine if a given local time is in DST or not and therefore adjust the offset appropriately.

2.

```
static time_duration_type
local_to_utc_offset(const time_type & t,
                  date_time::dst_flags dst = date_time::calculate);
```

Get the offset to UTC given a local time.

Class template local_adjustor

`boost::date_time::local_adjustor` — Template that simplifies the creation of local time calculator.

Synopsis

```
// In header: <boost/date_time/local_time_adjustor.hpp>

template<typename time_type, short utc_offset, typename dst_rule>
class local_adjustor {
public:
    // types
    typedef time_type::time_duration_type
        time_duration_type;
    typedef time_type::date_type
        date_type;
    typedef static_local_time_adjustor< time_type, dst_rule, utc_adjustment< time_dura-
tion_type, utc_offset > > dst_adjustor;

    // public static functions
    static time_type utc_to_local(const time_type &);
    static time_type
    local_to_utc(const time_type &, date_time::dst_flags = date_time::calculate);
};
```

Description

Use this template to create the timezone to utc convertors as required.

This class will also work for other regions that don't use dst and have a utc offset which is an integral number of hours.

Template Parameters -time_type -- Time class to use -utc_offset -- Number hours local time is adjust from utc -use_dst -- true (default) if region uses dst, false otherwise For example:

```
//eastern timezone is utc-5
typedef date_time::local_adjustor<ptime, -5, us_dst> us_eastern;
typedef date_time::local_adjustor<ptime, -6, us_dst> us_central;
typedef date_time::local_adjustor<ptime, -7, us_dst> us_mountain;
typedef date_time::local_adjustor<ptime, -8, us_dst> us_pacific;
typedef date_time::local_adjustor<ptime, -7, no_dst> us_arizona;
```

local_adjustor public static functions

1.

```
static time_type utc_to_local(const time_type & t);
```

Convert a utc time to local time.

2.

```
static time_type
local_to_utc(const time_type & t,
            date_time::dst_flags dst = date_time::calculate);
```

Convert a local time to utc.

Header **<boost/date_time/local_timezone_defs.hpp>**

```
namespace boost {
    namespace date_time {
        template<typename date_type> struct us_dst_trait;
        template<typename date_type> struct eu_dst_trait;
        template<typename date_type> struct uk_dst_trait;
        template<typename date_type> struct acst_dst_trait;
    }
}
```

Struct template us_dst_trait

boost::date_time::us_dst_trait — Specification for daylight savings start rules in US.

Synopsis

```
// In header: <boost/date_time/local_timezone_defs.hpp>

template<typename date_type>
struct us_dst_trait {
    // types
    typedef date_type::day_of_week_type      day_of_week_type;
    typedef date_type::month_type            month_type;
    typedef date_type::year_type             year_type;
    typedef date_type::nth_kday_of_month< date_type > start_rule_functor;
    typedef date_type::first_kday_of_month< date_type > end_rule_functor;
    typedef date_type::first_kday_of_month< date_type > start_rule_functor_pre2007;
    typedef date_type::last_kday_of_month< date_type > end_rule_functor_pre2007;

    // public static functions
    static day_of_week_type start_day(year_type);
    static month_type start_month(year_type);
    static day_of_week_type end_day(year_type);
    static month_type end_month(year_type);
    static date_type local_dst_start_day(year_type);
    static date_type local_dst_end_day(year_type);
    static int dst_start_offset_minutes();
    static int dst_end_offset_minutes();
    static int dst_shift_length_minutes();
};
```

Description

This class is used to configure [dst_calc_engine](#) template typically as follows:

```
using namespace boost::gregorian;
using namespace boost::posix_time;
typedef us_dst_trait<date> us_dst_traits;
typedef boost::date_time::dst_calc_engine<date, time_duration,
                                         us_dst_traits>
                                         us_dst_calc;

//calculate the 2002 transition day of USA April 7 2002
date dst_start = us_dst_calc::local_dst_start_day(2002);

//calculate the 2002 transition day of USA Oct 27 2002
date dst_end = us_dst_calc::local_dst_end_day(2002);

//check if a local time is in dst or not -- possible answers
//are yes, no, invalid time label, ambiguous
ptime t(...some time...);
if (us_dst::local_is_dst(t.date(), t.time_of_day())
    == boost::date_time::is_not_in_dst)
{
}
```

This generates a type suitable for the calculation of dst transitions for the United States. Of course other templates can be used for other locales.

us_dst_trait public static functions

1. `static day_of_week_type start_day(year_type);`

2.

```
static month_type start_month(year_type y);
```
3.

```
static day_of_week_type end_day(year_type);
```
4.

```
static month_type end_month(year_type y);
```
5.

```
static date_type local_dst_start_day(year_type year);
```
6.

```
static date_type local_dst_end_day(year_type year);
```
7.

```
static int dst_start_offset_minutes();
```
8.

```
static int dst_end_offset_minutes();
```
9.

```
static int dst_shift_length_minutes();
```

Struct template eu_dst_trait

boost::date_time::eu_dst_trait — Rules for daylight savings start in the EU (Last Sun in Mar)

Synopsis

```
// In header: <boost/date_time/local_timezone_defs.hpp>

template<typename date_type>
struct eu_dst_trait {
    // types
    typedef date_type::day_of_week_type    day_of_week_type;
    typedef date_type::month_type          month_type;
    typedef date_type::year_type            year_type;
    typedef date_time::last_kday_of_month< date_type > start_rule_functor;
    typedef date_time::last_kday_of_month< date_type > end_rule_functor;

    // public static functions
    static day_of_week_type start_day(year_type);
    static month_type start_month(year_type);
    static day_of_week_type end_day(year_type);
    static month_type end_month(year_type);
    static int dst_start_offset_minutes();
    static int dst_end_offset_minutes();
    static int dst_shift_length_minutes();
    static date_type local_dst_start_day(year_type);
    static date_type local_dst_end_day(year_type);
};
```


Description

These amount to the following:

- Start of dst day is last Sunday in March
- End day of dst is last Sunday in Oct
- Going forward switch time is 2:00 am (offset 120 minutes)
- Going back switch time is 3:00 am (off set 180 minutes)
- Shift duration is one hour (60 minutes)

eu_dst_trait public static functions

1.

```
static day_of_week_type start_day(year_type);
```
2.

```
static month_type start_month(year_type);
```
3.

```
static day_of_week_type end_day(year_type);
```
4.

```
static month_type end_month(year_type);
```
5.

```
static int dst_start_offset_minutes();
```
6.

```
static int dst_end_offset_minutes();
```
7.

```
static int dst_shift_length_minutes();
```
8.

```
static date_type local_dst_start_day(year_type year);
```
9.

```
static date_type local_dst_end_day(year_type year);
```

Struct template **uk_dst_trait**

boost::date_time::uk_dst_trait — Alternative dst traits for some parts of the United Kingdom.

Synopsis

```
// In header: <boost/date_time/local_timezone_defs.hpp>

template<typename date_type>
struct uk_dst_trait : public boost::date_time::eu_dst_trait< date_type > {

    // public static functions
    static int dst_start_offset_minutes();
    static int dst_end_offset_minutes();
    static int dst_shift_length_minutes();
};
```

Description

uk_dst_trait public static functions

1. `static int dst_start_offset_minutes();`
2. `static int dst_end_offset_minutes();`
3. `static int dst_shift_length_minutes();`

Struct template acst_dst_trait

boost::date_time::acst_dst_trait

Synopsis

```
// In header: <boost/date_time/local_timezone_defs.hpp>

template<typename date_type>
struct acst_dst_trait {
    // types
    typedef date_type::day_of_week_type      day_of_week_type;
    typedef date_type::month_type            month_type;
    typedef date_type::year_type             year_type;
    typedef date_time::last_kday_of_month< date_type > start_rule_functor;
    typedef date_time::last_kday_of_month< date_type > end_rule_functor;

    // public static functions
    static day_of_week_type start_day(year_type);
    static month_type start_month(year_type);
    static day_of_week_type end_day(year_type);
    static month_type end_month(year_type);
    static int dst_start_offset_minutes();
    static int dst_end_offset_minutes();
    static int dst_shift_length_minutes();
    static date_type local_dst_start_day(year_type);
    static date_type local_dst_end_day(year_type);
};
```

Description

acst_dst_trait public static functions

1.

```
static day_of_week_type start_day(year_type);
```
2.

```
static month_type start_month(year_type);
```
3.

```
static day_of_week_type end_day(year_type);
```
4.

```
static month_type end_month(year_type);
```
5.

```
static int dst_start_offset_minutes();
```
6.

```
static int dst_end_offset_minutes();
```
7.

```
static int dst_shift_length_minutes();
```
8.

```
static date_type local_dst_start_day(year_type year);
```
9.

```
static date_type local_dst_end_day(year_type year);
```

Header <boost/date_time/microsec_time_clock.hpp>

This file contains a high resolution time clock implementation.

```
namespace boost {  
    namespace date_time {  
        template<typename time_type> class microsec_clock;  
    }  
}
```

Class template microsec_clock

boost::date_time::microsec_clock — A clock providing microsecond level resolution.

Synopsis

```
// In header: <boost/date_time/microsec_time_clock.hpp>

template<typename time_type>
class microsec_clock {
public:
    // types
    typedef time_type::date_type      date_type;
    typedef time_type::time_duration_type time_duration_type;
    typedef time_duration_type::rep_type resolution_traits_type;

    // public static functions
    template<typename time_zone_type>
        static time_type local_time(shared_ptr< time_zone_type >);
    static time_type local_time();
    static time_type universal_time();

    // private static functions
    static time_type create_time(time_converter);
};
```

Description

A high precision clock that measures the local time at a resolution up to microseconds and adjusts to the resolution of the time system. For example, for the a library configuration with nano second resolution, the last 3 places of the fractional seconds will always be 000 since there are 1000 nano-seconds in a micro second.

microsec_clock public static functions

1.

```
template<typename time_zone_type>
    static time_type local_time(shared_ptr< time_zone_type > tz_ptr);
```

return a local time object for the given zone, based on computer clock

2.

```
static time_type local_time();
```

Returns the local time based on computer clock settings.

3.

```
static time_type universal_time();
```

Returns the UTC time based on computer settings.

microsec_clock private static functions

1.

```
static time_type create_time(time_converter converter);
```

Header `<boost/date_time/parse_format_base.hpp>`

```
namespace boost {
    namespace date_time {

        // Enum for distinguishing parsing and formatting options.
        enum month_format_spec { month_as_integer, month_as_short_string,
                                month_as_long_string };

        enum ymd_order_spec;

    }
}
```

Type `ymd_order_spec`

`boost::date_time::ymd_order_spec` — Enum for distinguishing the order of Month, Day, & Year.

Synopsis

```
// In header: <boost/date_time/parse_format_base.hpp>

enum ymd_order_spec { ymd_order_iso, ymd_order_dmy, ymd_order_us };
```

Description

Enum for distinguishing the order in which Month, Day, & Year will appear in a date string

Header `<boost/date_time/period.hpp>`

This file contain the implementation of the period abstraction. This is basically the same idea as a range. Although this class is intended for use in the time library, it is pretty close to general enough for other numeric uses.

```
namespace boost {
    namespace date_time {
        template<typename point_rep, typename duration_rep> class period;
    }
}
```

Class template `period`

`boost::date_time::period` — Provides generalized period type useful in date-time systems.

Synopsis

```
// In header: <boost/date_time/period.hpp>

template<typename point_rep, typename duration_rep>
class period {
public:
    // types
    typedef point_rep    point_type;
    typedef duration_rep duration_type;

    // construct/copy/destruct
    period(point_rep, point_rep);
    period(point_rep, duration_rep);

    // public member functions
    point_rep begin() const;
    point_rep end() const;
    point_rep last() const;
    duration_rep length() const;
    bool is_null() const;
    bool operator==(const period &) const;
    bool operator<(const period &) const;
    void shift(const duration_rep &);
    void expand(const duration_rep &);
    bool contains(const point_rep &) const;
    bool contains(const period &) const;
    bool intersects(const period &) const;
    bool is_adjacent(const period &) const;
    bool is_before(const point_rep &) const;
    bool is_after(const point_rep &) const;
    period intersection(const period &) const;
    period merge(const period &) const;
    period span(const period &) const;
};
```

Description

This template uses a class to represent a time point within the period and another class to represent a duration. As a result, this class is not appropriate for use when the number and duration representation are the same (eg: in the regular number domain).

A period can be specified by providing either the beginning point and a duration or the beginning point and the end point(end is NOT part of the period but 1 unit past it. A period will be "invalid" if either end_point <= begin_point or the given duration is <= 0. Any valid period will return false for is_null().

Zero length periods are also considered invalid. Zero length periods are periods where the beginning and end points are the same, or, the given duration is zero. For a zero length period, the last point will be one unit less than the beginning point.

In the case that the begin and last are the same, the period has a length of one unit.

The best way to handle periods is usually to provide a beginning point and a duration. So, day1 + 7 days is a week period which includes all of the first day and 6 more days (eg: Sun to Sat).

period public construct/copy/destruct

1. `period(point_rep first_point, point_rep end_point);`

create a period from begin to last eg: [begin,end)

If end <= begin then the period will be invalid

2. `period(point_rep first_point, duration_rep len);`

create a period as [begin, begin+len)

If len is ≤ 0 then the period will be invalid

period public member functions

1. `point_rep begin() const;`

Return the first element in the period.

2. `point_rep end() const;`

Return one past the last element.

3. `point_rep last() const;`

Return the last item in the period.

4. `duration_rep length() const;`

Return the length of the period.

5. `bool is_null() const;`

True if period is ill formed (length is zero or less)

6. `bool operator==(const period & rhs) const;`

Equality operator.

7. `bool operator<(const period & rhs) const;`

Strict as defined by `rhs.last <= lhs.last`.

8. `void shift(const duration_rep & d);`

Shift the start and end by the specified amount.

9. `void expand(const duration_rep & d);`

Expands the size of the period by the duration on both ends.

So before expand

```

      [-----]
    ^   ^   ^   ^   ^   ^   ^
    1   2   3   4   5   6   7
    *

```

After expand(2)

```

[-----]
^   ^   ^   ^   ^   ^   ^
1   2   3   4   5   6   7
    *

```

10. `bool contains(const point_rep & point) const;`

True if the point is inside the period, zero length periods contain no points.

11. `bool contains(const period & other) const;`

True if this period fully contains (or equals) the other period.

12. `bool intersects(const period & other) const;`

True if the periods overlap in any way.

13. `bool is_adjacent(const period & other) const;`

True if periods are next to each other without a gap.

14. `bool is_before(const point_rep & point) const;`

True if all of the period is prior to the passed point or $\text{end} \leq t$.

15. `bool is_after(const point_rep & point) const;`

True if all of the period is prior or $t < \text{start}$.

16. `period intersection(const period & other) const;`

Returns the period of intersection or invalid range no intersection.

17. `period merge(const period & other) const;`

Returns the union of intersecting periods -- or null period.

18. `period span(const period & other) const;`

Combine two periods with earliest start and latest end.

Combines two periods and any gap between them such that $\text{start} = \min(p1.\text{start}, p2.\text{start})$ $\text{end} = \max(p1.\text{end}, p2.\text{end})$

```

      [---p1---)           [---p2---)
result:
      [-----p3-----)
    *

```


Header `<boost/date_time/period_formatter.hpp>`

```
namespace boost {
    namespace date_time {
        template<typename CharT,
                typename OutItrT = std::ostreambuf_iterator<CharT, std::char_traits<CharT> > >
                class period_formatter;
    }
}
```

Class template `period_formatter`

`boost::date_time::period_formatter` — Not a facet, but a class used to specify and control period formats.

Synopsis

```
// In header: <boost/date_time/period_formatter.hpp>

template<typename CharT,
        typename OutItrT = std::ostreambuf_iterator<CharT, std::char_traits<CharT> > >
class period_formatter {
public:
    // types
    typedef std::basic_string< CharT >                string_type;
    typedef CharT                                     char_type;
    typedef std::basic_string< char_type >::const_iterator const_itr_type;
    typedef std::vector< std::basic_string< CharT > >    collection_type;

    enum range_display_options { AS_OPEN_RANGE, AS_CLOSED_RANGE };

    // construct/copy/destruct
    period_formatter(range_display_options = AS_CLOSED_RANGE,
                    const char_type *const = default_period_separator,
                    const char_type *const = default_period_start_delimiter,
                    const char_type *const = default_period_open_range_end_delimiter,
                    const char_type *const = default_period_closed_range_end_delimiter);

    // public member functions
    OutItrT put_period_separator(OutItrT &) const;
    OutItrT put_period_start_delimeter(OutItrT &) const;
    OutItrT put_period_end_delimeter(OutItrT &) const;
    range_display_options range_option() const;
    void range_option(range_display_options) const;
    void delimiter_strings(const string_type &, const string_type &,
                          const string_type &, const string_type &);
    template<typename period_type, typename facet_type>
        OutItrT put_period(OutItrT, std::ios_base &, char_type,
                          const period_type &, const facet_type &) const;

    // public data members
    static const char_type default_period_separator;
    static const char_type default_period_start_delimeter;
    static const char_type default_period_open_range_end_delimeter;
    static const char_type default_period_closed_range_end_delimeter;
};
```

Description

Provides settings for the following:

- `period_separator` -- default `'/'`
- `period_open_start_delimiter` -- default `'['`
- `period_open_range_end_delimiter` -- default `']'`
- `period_closed_range_end_delimiter` -- default `']'`
- `display_as_open_range`, `display_as_closed_range` -- default `closed_range`

Thus the default formatting for a period is as follows:

```
[period.start()/period.last()]
*
```

So for a typical date_period this would be

```
[ 2004-Jan-04/2004-Feb-01 ]
*
```

where the date formatting is controlled by the date facet

period_formatter public construct/copy/destruct

1.

```
period_formatter(range_display_options range_option_in = AS_CLOSED_RANGE,
                 const char_type *const period_separator = default_period_separator,
                 const char_type *const period_start_delimiter = default_period_start_delimiter,
                 const char_type *const period_open_range_end_delimiter = default_period_open_range_end_delimiter,
                 const char_type *const period_closed_range_end_delimiter = default_period_closed_range_end_delimiter);
```

Constructor that sets up period formatter options -- default should suffice most cases.

period_formatter public member functions

1.

```
OutItrT put_period_separator(OutItrT & oitr) const;
```

Puts the characters between period elements into stream -- default is `./`.

2.

```
OutItrT put_period_start_delimiter(OutItrT & oitr) const;
```

Puts the period start characters into stream -- default is `[`.

3.

```
OutItrT put_period_end_delimiter(OutItrT & oitr) const;
```

Puts the period end characters into stream as controlled by open/closed range setting.

4.

```
range_display_options range_option() const;
```

5.

```
void range_option(range_display_options option) const;
```

Reset the range_option control.

6.

```
void delimiter_strings(const string_type & separator,
                     const string_type & start_delim,
                     const string_type & open_end_delim,
                     const string_type & closed_end_delim);
```
7.

```
template<typename period_type, typename facet_type>
    OutItrT put_period(OutItrT next, std::ios_base & a_ios, char_type a_fill,
                      const period_type & p, const facet_type & facet) const;
```

Generic code to output a period -- no matter the period type.

This generic code will output any period using a facet to output the 'elements'. For example, in the case of a date_period the elements will be instances of a date which will be formatted according to the setup in the passed facet parameter.

The steps for formatting a period are always the same:

- put the start delimiter
- put start element
- put the separator
- put either last or end element depending on range settings
- put end delimiter depending on range settings

Thus for a typical date period the result might look like this:

```
[March 01, 2004/June 07, 2004]    <-- closed range
[March 01, 2004/June 08, 2004)    <-- open range

*
```

Header <boost/date_time/period_parser.hpp>

```
namespace boost {
    namespace date_time {
        template<typename date_type, typename CharT> class period_parser;
    }
}
```

Class template period_parser

boost::date_time::period_parser — Not a facet, but a class used to specify and control period parsing.

Synopsis

```
// In header: <boost/date_time/period_parser.hpp>

template<typename date_type, typename CharT>
class period_parser {
public:
    // types
    typedef std::basic_string< CharT >          string_type;
    typedef CharT                               char_type;
    typedef std::istreambuf_iterator< CharT >    stream_itr_type;
    typedef string_parse_tree< CharT >          parse_tree_type;
    typedef parse_tree_type::parse_match_result_type match_results;
    typedef std::vector< std::basic_string< CharT > > collection_type;

    enum period_range_option { AS_OPEN_RANGE, AS_CLOSED_RANGE };

    // construct/copy/destruct
    period_parser(period_range_option = AS_CLOSED_RANGE,
                  const char_type *const = default_period_separator,
                  const char_type *const = default_period_start_delimiter,
                  const char_type *const = default_period_open_range_end_delimiter,
                  const char_type *const = default_period_closed_range_end_delimiter);
    period_parser(const period_parser< date_type, CharT > &);

    // public member functions
    period_range_option range_option() const;
    void range_option(period_range_option);
    collection_type delimiter_strings() const;
    void delimiter_strings(const string_type &, const string_type &,
                          const string_type &, const string_type &);
    template<typename period_type, typename duration_type, typename facet_type>
        period_type get_period(stream_itr_type &, stream_itr_type &,
                               std::ios_base &, const period_type &,
                               const duration_type &, const facet_type &) const;

    // private member functions
    void consume_delim(stream_itr_type &, stream_itr_type &,
                      const string_type &) const;

    // public data members
    static const char_type default_period_separator;
    static const char_type default_period_start_delimiter;
    static const char_type default_period_open_range_end_delimiter;
    static const char_type default_period_closed_range_end_delimiter;
};
```

Description

Provides settings for the following:

- period_separator -- default '/'
- period_open_start_delimiter -- default '['
- period_open_range_end_delimiter -- default ')
- period_closed_range_end_delimiter -- default ']'
- display_as_open_range, display_as_closed_range -- default closed_range

For a typical date_period, the contents of the input stream would be

```
[ 2004-Jan-04 / 2004-Feb-01 ]
*
```

where the date format is controlled by the date facet

period_parser public construct/copy/destruct

1.

```
period_parser(period_range_option range_opt = AS_CLOSED_RANGE,
               const char_type *const period_separator = default_period_separator,
               const char_type *const period_start_delimiter = default_period_start_delimiter,

               const char_type *const period_open_range_end_delimiter = default_peri↵
               od_open_range_end_delimiter,
               const char_type *const period_closed_range_end_delimiter = default_peri↵
               od_closed_range_end_delimiter);
```

Constructor that sets up period parser options.

2.

```
period_parser(const period_parser< date_type, CharT > & p_parser);
```

period_parser public member functions

1.

```
period_range_option range_option() const;
```
2.

```
void range_option(period_range_option option);
```
3.

```
collection_type delimiter_strings() const;
```
4.

```
void delimiter_strings(const string_type & separator,
                       const string_type & start_delim,
                       const string_type & open_end_delim,
                       const string_type & closed_end_delim);
```
5.

```
template<typename period_type, typename duration_type, typename facet_type>
period_type get_period(stream_itr_type & sitr, stream_itr_type & stream_end,
                       std::ios_base & a_ios, const period_type &,
                       const duration_type & dur_unit,
                       const facet_type & facet) const;
```

Generic code to parse a period -- no matter the period type.

This generic code will parse any period using a facet to to get the 'elements'. For example, in the case of a date_period the elements will be instances of a date which will be parsed according the to setup in the passed facet parameter.

The steps for parsing a period are always the same:

- consume the start delimiter
- get start element
- consume the separator

- get either last or end element depending on range settings
- consume the end delimiter depending on range settings

Thus for a typical date period the contents of the input stream might look like this:

```
[March 01, 2004/June 07, 2004]    <-- closed range
[March 01, 2004/June 08, 2004)    <-- open range

*
```

period_parser private member functions

```
1. void consume_delim(stream_itr_type & sitr, stream_itr_type & stream_end,
    const string_type & delim) const;
```

throws ios_base::failure if delimiter and parsed data do not match

Header <boost/date_time/special_defs.hpp>

```
namespace boost {
    namespace date_time {

        enum special_values { not_a_date_time, neg_infin, pos_infin,
                               min_date_time, max_date_time, not_special,
                               NumSpecialValues };

    }
}
```

Header <boost/date_time/special_values_formatter.hpp>

```
namespace boost {
    namespace date_time {
        template<typename CharT,
                typename OutItrT = std::ostreambuf_iterator<CharT, std::char_traits<CharT> > >
                class special_values_formatter;
    }
}
```

Class template special_values_formatter

boost::date_time::special_values_formatter — Class that provides generic formatting ostream formatting for special values.

Synopsis

```
// In header: <boost/date_time/special_values_formatter.hpp>

template<typename CharT,
        typename OutItrT = std::ostreambuf_iterator<CharT, std::char_traits<CharT> > >
class special_values_formatter {
public:
    // types
    typedef std::basic_string< CharT > string_type;
    typedef CharT char_type;
    typedef std::vector< string_type > collection_type;

    // construct/copy/destruct
    special_values_formatter();
    special_values_formatter(const char_type *const *, const char_type *const *);
    special_values_formatter(typename collection_type::iterator,
                            typename collection_type::iterator);

    // public member functions
    OutItrT put_special(OutItrT, const boost::date_time::special_values &) const;

    // public data members
    static const char_type default_special_value_names; // Storage for the strings used to indicate special values.
};
```

Description

This class provides for the formatting of special values to an output stream. In particular, it produces strings for the values of negative and positive infinity as well as not_a_date_time.

While not a facet, this class is used by the date and time facets for formatting special value types.

special_values_formatter public construct/copy/destruct

1.

```
special_values_formatter();
```

Construct special values formatter using default strings.

Default strings are not-a-date-time -infinity +infinity

2.

```
special_values_formatter(const char_type *const * begin,
                        const char_type *const * end);
```

Construct special values formatter from array of strings.

This constructor will take pair of iterators from an array of strings that represent the special values and copy them for use in formatting special values.

```
const char* const special_value_names[]={ "nadt", "-inf", "+inf" };

special_value_formatter svf(&special_value_names[0], &special_value_names[3]);
*
```

3.

```
special_values_formatter(typename collection_type::iterator beg,
                        typename collection_type::iterator end);
```

special_values_formatter public member functions

1.

```
OutItrT put_special(OutItrT next,
                    const boost::date_time::special_values & value) const;
```

Header <boost/date_time/special_values_parser.hpp>

```
namespace boost {
  namespace date_time {
    template<typename date_type, typename charT> class special_values_parser;
  }
}
```

Class template special_values_parser

boost::date_time::special_values_parser — Class for special_value parsing.

Synopsis

```
// In header: <boost/date_time/special_values_parser.hpp>

template<typename date_type, typename charT>
class special_values_parser {
public:
  // types
  typedef std::basic_string< charT > string_type;
  typedef std::istreambuf_iterator< charT > stream_itr_type;
  typedef date_type::duration_type duration_type;
  typedef string_parse_tree< charT > parse_tree_type;
  typedef parse_tree_type::parse_match_result_type match_results;
  typedef std::vector< std::basic_string< charT > > collection_type;
  typedef charT char_type;

  // construct/copy/destruct
  special_values_parser();
  special_values_parser(const string_type &, const string_type &,
                      const string_type &, const string_type &,
                      const string_type &);
  special_values_parser(typename collection_type::iterator,
                      typename collection_type::iterator);
  special_values_parser(const special_values_parser< date_type, charT > &);

  // public member functions
  void sv_strings(const string_type &, const string_type &,
                const string_type &, const string_type &,
                const string_type &);
  bool match(stream_itr_type &, stream_itr_type &, match_results &) const;

  // public data members
  static const char_type nadt_string;
  static const char_type neg_inf_string;
  static const char_type pos_inf_string;
  static const char_type min_date_time_string;
  static const char_type max_date_time_string;
};
```


Description

TODO: add doc-comments for which elements can be changed Parses input stream for strings representing special_values. Special values parsed are:

- not_a_date_time
- neg_infin
- pod_infin
- min_date_time
- max_date_time

special_values_parser public construct/copy/destruct

1.

```
special_values_parser();
```

Creates a `special_values_parser` with the default set of "sv_strings".

2.

```
special_values_parser(const string_type & nadt_str,
                     const string_type & neg_inf_str,
                     const string_type & pos_inf_str,
                     const string_type & min_dt_str,
                     const string_type & max_dt_str);
```

Creates a `special_values_parser` using a user defined set of element strings.

3.

```
special_values_parser(typename collection_type::iterator beg,
                     typename collection_type::iterator end);
```

4.

```
special_values_parser(const special_values_parser< date_type, charT > & svp);
```

special_values_parser public member functions

1.

```
void sv_strings(const string_type & nadt_str, const string_type & neg_inf_str,
               const string_type & pos_inf_str,
               const string_type & min_dt_str,
               const string_type & max_dt_str);
```

Replace special value strings.

2.

```
bool match(stream_itr_type & sitr, stream_itr_type & str_end,
           match_results & mr) const;
```

Sets `match_results.current_match` to the corresponding special_value or -1.

Header `<boost/date_time/string_convert.hpp>`

```
namespace boost {
    namespace date_time {
        template<typename InputT, typename OutputT>
            std::basic_string< OutputT >
                convert_string_type(const std::basic_string< InputT > &);
    }
}
```

Function template `convert_string_type`

`boost::date_time::convert_string_type` — Converts a string from one value_type to another.

Synopsis

```
// In header: <boost/date_time/string_convert.hpp>

template<typename InputT, typename OutputT>
    std::basic_string< OutputT >
        convert_string_type(const std::basic_string< InputT > & inp_str);
```

Description

Converts a wstring to a string (or a string to wstring). If both template parameters are of same type, a copy of the input string is returned.

Header `<boost/date_time/string_parse_tree.hpp>`

```
namespace boost {
    namespace date_time {
        template<typename charT> struct parse_match_result;
        template<typename charT> struct string_parse_tree;
        template<typename charT>
            std::basic_ostream< charT > &
                operator<<(std::basic_ostream< charT > & os,
                    parse_match_result< charT > & mr);
    }
}
```

Struct template `parse_match_result`

`boost::date_time::parse_match_result`

Synopsis

```
// In header: <boost/date_time/string_parse_tree.hpp>

template<typename charT>
struct parse_match_result {
    // types
    typedef std::basic_string< charT > string_type;

    enum PARSE_STATE { PARSE_ERROR = -1 };

    // construct/copy/destroy
    parse_match_result();

    // public member functions
    string_type remaining() const;
    charT last_char() const;
    bool has_remaining() const;

    // public data members
    string_type cache;
    unsigned short match_depth;
    short current_match;
};
```

Description

parse_match_result public construct/copy/destroy

1. `parse_match_result();`

parse_match_result public member functions

1. `string_type remaining() const;`

2. `charT last_char() const;`

3. `bool has_remaining() const;`

Returns true if more characters were parsed than was necessary.

Should be used in conjunction with `last_char()` to get the remaining character.

Struct template string_parse_tree

`boost::date_time::string_parse_tree` — Recursive data structure to allow efficient parsing of various strings.

Synopsis

```
// In header: <boost/date_time/string_parse_tree.hpp>

template<typename charT>
struct string_parse_tree {
    // types
    typedef std::multimap< charT, string_parse_tree< charT > > ptree_coll;
    typedef std::multimap< charT, string_parse_tree > ptree_coll;
    typedef ptree_coll::value_type value_type;
    typedef ptree_coll::iterator iterator;
    typedef ptree_coll::const_iterator const_iterator;
    typedef std::basic_string< charT > string_type;
    typedef std::vector< std::basic_string< charT > > collection_type;
    typedef parse_match_result< charT > parse_match_result_type;

    // construct/copy/destruct
    string_parse_tree(collection_type, unsigned int = 0);
    string_parse_tree(short = -1);

    // public member functions
    void insert(const string_type &, unsigned short);
    short match(std::istreambuf_iterator< charT > &,
                std::istreambuf_iterator< charT > &, parse_match_result_type &,
                unsigned int &) const;
    parse_match_result_type
    match(std::istreambuf_iterator< charT > &,
          std::istreambuf_iterator< charT > &) const;
    void printme(std::ostream &, int &);
    void print(std::ostream &);
    void printmatch(std::ostream &, charT);

    // public data members
    ptree_coll m_next_chars;
    short m_value;
};
```

Description

This class provides a quick lookup by building what amounts to a tree data structure. It also features a match function which can handle nasty input iterators by caching values as it recurses the tree so that it can backtrack as needed.

string_parse_tree public construct/copy/destruct

1. `string_parse_tree(collection_type names, unsigned int starting_point = 0);`

Parameter "starting_point" designates where the numbering begins. A starting_point of zero will start the numbering at zero (Sun=0, Mon=1, ...) were a starting_point of one starts the numbering at one (Jan=1, Feb=2, ...). The default is zero, negative vaules are not allowed

2. `string_parse_tree(short value = -1);`

string_parse_tree public member functions

1. `void insert(const string_type & s, unsigned short value);`

```
2. short match(std::istreambuf_iterator< charT > & sitr,
              std::istreambuf_iterator< charT > & stream_end,
              parse_match_result_type & result, unsigned int & level) const;
```

Recursive function that finds a matching string in the tree.

Must check `match_results::has_remaining()` after `match()` is called. This is required so the user can determine if stream iterator is already pointing to the expected character or not (`match()` might advance `sitr` to next char in stream).

A `parse_match_result` that has been returned from a failed match attempt can be sent in to the `match` function of a different `string_parse_tree` to attempt a match there. Use the iterators for the partially consumed stream, the `parse_match_result` object, and '0' for the level parameter.

```
3. parse_match_result_type
match(std::istreambuf_iterator< charT > & sitr,
      std::istreambuf_iterator< charT > & stream_end) const;
```

Must check `match_results::has_remaining()` after `match()` is called. This is required so the user can determine if stream iterator is already pointing to the expected character or not (`match()` might advance `sitr` to next char in stream).

```
4. void printme(std::ostream & os, int & level);
```

```
5. void print(std::ostream & os);
```

```
6. void printmatch(std::ostream & os, charT c);
```

Header `<boost/date_time/strings_from_facet.hpp>`

```
namespace boost {
  namespace date_time {
    template<typename charT>
      std::vector< std::basic_string< charT > >
      gather_month_strings(const std::locale &, bool = true);
    template<typename charT>
      std::vector< std::basic_string< charT > >
      gather_weekday_strings(const std::locale &, bool = true);
  }
}
```

Function template `gather_month_strings`

`boost::date_time::gather_month_strings` — This function gathers up all the month strings from a `std::locale`.

Synopsis

```
// In header: <boost/date_time/strings_from_facet.hpp>

template<typename charT>
  std::vector< std::basic_string< charT > >
  gather_month_strings(const std::locale & locale, bool short_strings = true);
```

Description

Using the `time_put` facet, this function creates a collection of all the month strings from a locale. This is handy when building custom date parsers or formatters that need to be localized.

Parameters: `locale` The locale to use when gathering the strings
 `short_strings` True(default) to gather short strings, false for long strings.
Returns: A vector of strings containing the strings in order. eg: Jan, Feb, Mar, etc.

Function template `gather_weekday_strings`

`boost::date_time::gather_weekday_strings` — This function gathers up all the weekday strings from a `std::locale`.

Synopsis

```
// In header: <boost/date_time/strings_from_facet.hpp>

template<typename charT>
    std::vector< std::basic_string< charT > >
    gather_weekday_strings(const std::locale & locale,
                          bool short_strings = true);
```

Description

Using the `time_put` facet, this function creates a collection of all the weekday strings from a locale starting with the string for 'Sunday'. This is handy when building custom date parsers or formatters that need to be localized.

Parameters: `locale` The locale to use when gathering the strings
 `short_strings` True(default) to gather short strings, false for long strings.
Returns: A vector of strings containing the weekdays in order. eg: Sun, Mon, Tue, Wed, Thu, Fri, Sat

Header `<boost/date_time/time.hpp>`

This file contains the interface for the time associated classes.

```
namespace boost {
    namespace date_time {
        template<typename T, typename time_system> class base_time;
    }
}
```

Class template `base_time`

`boost::date_time::base_time` — Representation of a precise moment in time, including the date.

Synopsis

```
// In header: <boost/date_time/time.hpp>

template<typename T, typename time_system>
class base_time {
public:
    // types
    typedef T time_type;
    typedef time_system::time_rep_type time_rep_type;
    typedef time_system::date_type date_type;
    typedef time_system::date_duration_type date_duration_type;
    typedef time_system::time_duration_type time_duration_type;

    // construct/copy/destruct
    base_time(const date_type &, const time_duration_type &,
              dst_flags = not_dst);
    base_time(special_values);
    base_time(const time_rep_type &);

    // public member functions
    date_type date() const;
    time_duration_type time_of_day() const;
    std::string zone_name(bool = false) const;
    std::string zone_abbrev(bool = false) const;
    std::string zone_as_posix_string() const;
    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;
    bool operator==(const time_type &) const;
    bool operator<(const time_type &) const;
    time_duration_type operator-(const time_type &) const;
    time_type operator+(const date_duration_type &) const;
    time_type operator+=(const date_duration_type &);
    time_type operator-(const date_duration_type &) const;
    time_type operator-=(const date_duration_type &);
    time_type operator+(const time_duration_type &) const;
    time_type operator+=(const time_duration_type &);
    time_type operator-(const time_duration_type &) const;
    time_type operator-=(const time_duration_type &);
};
```

Description

This class is a skeleton for the interface of a temporal type with a resolution that is higher than a day. It is intended that this class be the base class and that the actual time class be derived using the BN pattern. In this way, the derived class can make decisions such as 'should there be a default constructor' and what should it set its value to, should there be optional constructors say allowing only an time_durations that generate a time from a clock, etc. So, in fact multiple time types can be created for a time_system with different construction policies, and all of them can perform basic operations by only writing a copy constructor. Finally, compiler errors are also shorter.

The real behavior of the time class is provided by the time_system template parameter. This class must provide all the logic for addition, subtraction, as well as define all the interface types.

base_time public construct/copy/destruct

1.

```
base_time(const date_type & day, const time_duration_type & td,
          dst_flags dst = not_dst);
```

2. `base_time(special_values sv);`

3. `base_time(const time_rep_type & rhs);`

base_time public member functions

1. `date_type date() const;`

2. `time_duration_type time_of_day() const;`

3. `std::string zone_name(bool = false) const;`

Optional bool parameter will return time zone as an offset (ie "+07:00"). Empty string is returned for classes that do not use a time_zone

4. `std::string zone_abbrev(bool = false) const;`

Optional bool parameter will return time zone as an offset (ie "+07:00"). Empty string is returned for classes that do not use a time_zone

5. `std::string zone_as_posix_string() const;`

An empty string is returned for classes that do not use a time_zone.

6. `bool is_not_a_date_time() const;`

check to see if date is not a value

7. `bool is_infinity() const;`

check to see if date is one of the infinity values

8. `bool is_pos_infinity() const;`

check to see if date is greater than all possible dates

9. `bool is_neg_infinity() const;`

check to see if date is greater than all possible dates

10. `bool is_special() const;`

check to see if time is a special value

11. `bool operator==(const time_type & rhs) const;`

Equality operator -- others generated by boost::equality_comparable.


```
12 bool operator<(const time_type & rhs) const;
```

Equality operator -- others generated by boost::less_than_comparable.

```
13 time_duration_type operator-(const time_type & rhs) const;
```

difference between two times

```
14 time_type operator+(const date_duration_type & dd) const;
```

add date durations

```
15 time_type operator+=(const date_duration_type & dd);
```

```
16 time_type operator-(const date_duration_type & dd) const;
```

subtract date durations

```
17 time_type operator-=(const date_duration_type & dd);
```

```
18 time_type operator+(const time_duration_type & td) const;
```

add time durations

```
19 time_type operator+=(const time_duration_type & td);
```

```
20 time_type operator-(const time_duration_type & rhs) const;
```

subtract time durations

```
21 time_type operator-=(const time_duration_type & td);
```

Header <boost/date_time/time_clock.hpp>

This file contains the interface for clock devices.

```
namespace boost {  
    namespace date_time {  
        template<typename time_type> class second_clock;  
    }  
}
```

Class template second_clock

boost::date_time::second_clock — A clock providing time level services based on C time_t capabilities.

Synopsis

```
// In header: <boost/date_time/time_clock.hpp>

template<typename time_type>
class second_clock {
public:
    // types
    typedef time_type::date_type          date_type;
    typedef time_type::time_duration_type time_duration_type;

    // public static functions
    static time_type local_time();
    static time_type universal_time();
    template<typename time_zone_type>
        static time_type local_time(boost::shared_ptr< time_zone_type >);

    // private static functions
    static time_type create_time(::std::tm *);
};
```

Description

This clock provides resolution to the 1 second level

second_clock public static functions

1. `static time_type local_time();`

2. `static time_type universal_time();`

Get the current day in universal date as a ymd_type.

3. `template<typename time_zone_type>`
`static time_type local_time(boost::shared_ptr< time_zone_type > tz_ptr);`

second_clock private static functions

1. `static time_type create_time(::std::tm * current);`

Header **<boost/date_time/time_defs.hpp>**

This file contains nice definitions for handling the resolution of various time representations.

```
namespace boost {
    namespace date_time {

        // Defines some nice types for handling time level resolutions.
        enum time_resolutions { sec, tenth, hundreth, hundredth = hundreth,
                                milli, ten_thousandth, micro, nano,
                                NumResolutions };

        // Flags for daylight savings or summer time.
        enum dst_flags { not_dst, is_dst, calculate };

    }
}
```

Header <boost/date_time/time_duration.hpp>

```
namespace boost {
    namespace date_time {
        template<typename T, typename rep_type> class time_duration;
        template<typename base_duration, boost::int64_t frac_of_second>
            class subsecond_duration;
    }
}
```

Class template time_duration

boost::date_time::time_duration — Represents some amount of elapsed time measure to a given resolution.

Synopsis

```
// In header: <boost/date_time/time_duration.hpp>

template<typename T, typename rep_type>
class time_duration {
public:
    // types
    typedef T duration_type;
    typedef rep_type traits_type;
    typedef rep_type::day_type day_type;
    typedef rep_type::hour_type hour_type;
    typedef rep_type::min_type min_type;
    typedef rep_type::sec_type sec_type;
    typedef rep_type::fractional_seconds_type fractional_seconds_type;
    typedef rep_type::tick_type tick_type;
    typedef rep_type::impl_type impl_type;

    // construct/copy/destruct
    time_duration();
    time_duration(hour_type, min_type, sec_type = 0,
                  fractional_seconds_type = 0);
    time_duration(const time_duration< T, rep_type > &);
    time_duration(special_values);
    explicit time_duration(impl_type);

    // public member functions
    hour_type hours() const;
    min_type minutes() const;
    sec_type seconds() const;
    sec_type total_seconds() const;
    tick_type total_milliseconds() const;
    tick_type total_nanoseconds() const;
    tick_type total_microseconds() const;
    fractional_seconds_type fractional_seconds() const;
    duration_type invert_sign() const;
    bool is_negative() const;
    bool operator<(const time_duration &) const;
    bool operator==(const time_duration &) const;
    duration_type operator-() const;
    duration_type operator-(const duration_type &) const;
    duration_type operator+(const duration_type &) const;
    duration_type operator/(int) const;
    duration_type operator-=(const duration_type &);
    duration_type operator+=(const duration_type &);
    duration_type operator/=(int);
    duration_type operator*(int) const;
    duration_type operator*=(int);
    tick_type ticks() const;
    bool is_special() const;
    bool is_pos_infinity() const;
    bool is_neg_infinity() const;
    bool is_not_a_date_time() const;
    impl_type get_rep() const;

    // public static functions
    static duration_type unit();
    static tick_type ticks_per_second();
    static time_resolutions resolution();
    static unsigned short num_fractional_digits();
};
```

Description

This class represents a standard set of capabilities for all counted time durations. Time duration implementations should derive from this class passing their type as the first template parameter. This design allows the subclass duration types to provide custom construction policies or other custom features not provided here.

`time_duration` public construct/copy/destroy

1.

```
time_duration();
```
2.

```
time_duration(hour_type hours_in, min_type minutes_in,  
              sec_type seconds_in = 0,  
              fractional_seconds_type frac_sec_in = 0);
```
3.

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

Construct from another `time_duration` (Copy constructor)

4.

```
time_duration(special_values sv);
```

Construct from `special_values`.

5.

```
explicit time_duration(impl_type in);
```

`time_duration` public member functions

1.

```
hour_type hours() const;
```

Returns number of hours in the duration.

2.

```
min_type minutes() const;
```

Returns normalized number of minutes.

3.

```
sec_type seconds() const;
```

Returns normalized number of seconds (0..60)

4.

```
sec_type total_seconds() const;
```

Returns total number of seconds truncating any fractional seconds.

5.

```
tick_type total_milliseconds() const;
```

Returns total number of milliseconds truncating any fractional seconds.

6.

```
tick_type total_nanoseconds() const;
```

Returns total number of nanoseconds truncating any sub millisecond values.

7. `tick_type total_microseconds() const;`

Returns total number of microseconds truncating any sub microsecond values.

8. `fractional_seconds_type fractional_seconds() const;`

Returns count of fractional seconds at given resolution.

9. `duration_type invert_sign() const;`

10. `bool is_negative() const;`

11. `bool operator<(const time_duration & rhs) const;`

12. `bool operator==(const time_duration & rhs) const;`

13. `duration_type operator-() const;`

unary- Allows for `time_duration` `td = -td1`

14. `duration_type operator-(const duration_type & d) const;`

15. `duration_type operator+(const duration_type & d) const;`

16. `duration_type operator/(int divisor) const;`

17. `duration_type operator--(const duration_type & d);`

18. `duration_type operator+=(const duration_type & d);`

19. `duration_type operator/=(int divisor);`

Division operations on a duration with an integer.

20. `duration_type operator*(int rhs) const;`

Multiplication operations on a duration with an integer.

21. `duration_type operator*=(int divisor);`

22. `tick_type ticks() const;`

23. `bool is_special() const;`

Is ticks_ a special value?

24. `bool is_pos_infinity() const;`

Is duration pos-infinity.

25. `bool is_neg_infinity() const;`

Is duration neg-infinity.

26. `bool is_not_a_date_time() const;`

Is duration not-a-date-time.

27. `impl_type get_rep() const;`

Used for special_values output.

time_duration public static functions

1. `static duration_type unit();`

Returns smallest representable duration.

2. `static tick_type ticks_per_second();`

Return the number of ticks in a second.

3. `static time_resolutions resolution();`

Provide the resolution of this duration type.

4. `static unsigned short num_fractional_digits();`

Returns number of possible digits in fractional seconds.

Class template subsecond_duration

`boost::date_time::subsecond_duration` — Template for instantiating derived adjusting durations.

Synopsis

```
// In header: <boost/date_time/time_duration.hpp>

template<typename base_duration, boost::int64_t frac_of_second>
class subsecond_duration {
public:
    // types
    typedef base_duration::traits_type traits_type;

    // construct/copy/destruct
    explicit subsecond_duration(boost::int64_t);
};
```

Description

subsecond_duration public construct/copy/destruct

1. `explicit subsecond_duration(boost::int64_t ss);`

Header **<boost/date_time/time_facet.hpp>**

```
namespace boost {
    namespace date_time {
        template<typename CharT> struct time_formats;

        template<typename time_type, typename CharT,
                typename OutItrT = std::ostreambuf_iterator<CharT, std::char_traits<CharT> > >
                class time_facet;
        template<typename time_type, typename CharT,
                typename InItrT = std::istreambuf_iterator<CharT, std::char_traits<CharT> > >
                class time_input_facet;
    }
}
```

Struct template **time_formats**

boost::date_time::time_formats

Synopsis

```
// In header: <boost/date_time/time_facet.hpp>

template<typename CharT>
struct time_formats {
    // types
    typedef CharT char_type;

    // public data members
    static const char_type fractional_seconds_format;
    static const char_type fractional_seconds_or_none_format;
    static const char_type seconds_with_fractional_seconds_format;
    static const char_type seconds_format;
    static const char_type hours_format;
    static const char_type unrestricted_hours_format;
    static const char_type full_24_hour_time_format;
    static const char_type full_24_hour_time_expanded_format;
    static const char_type short_24_hour_time_format;
    static const char_type short_24_hour_time_expanded_format;
    static const char_type standard_format;
    static const char_type zone_abbrev_format;
    static const char_type zone_name_format;
    static const char_type zone_iso_format;
    static const char_type zone_iso_extended_format;
    static const char_type posix_zone_string_format;
    static const char_type duration_sign_negative_only;
    static const char_type duration_sign_always;
    static const char_type duration_seperator;
    static const char_type negative_sign;
    static const char_type positive_sign;
    static const char_type iso_time_format_specifier;
    static const char_type iso_time_format_extended_specifier;
    static const char_type default_time_format;
    static const char_type default_time_input_format;
    static const char_type default_time_duration_format;
};
```

Class template time_facet

boost::date_time::time_facet

Synopsis

```
// In header: <boost/date_time/time_facet.hpp>

template<typename time_type, typename CharT,
        typename OutItrT = std::ostreambuf_iterator<CharT, std::char_traits<CharT> > >
class time_facet :
    public boost::date_time::date_facet< time_type::date_type, CharT, OutItrT >
{
public:
    // types
    typedef time_type::date_type                                date_type; ↵

    typedef time_type::time_duration_type                       time_dur↵
ation_type;
    typedef boost::date_time::period< time_type, time_duration_type >    peri↵
od_type;
    typedef boost::date_time::date_facet< typename time_type::date_type, CharT, OutItrT > base_type; ↵

    typedef base_type::string_type                                ↵
string_type;
    typedef base_type::char_type                                char_type; ↵

    typedef base_type::period_formatter_type                     peri↵
od_formatter_type;
    typedef base_type::special_values_formatter_type            spe↵
cial_values_formatter_type;
    typedef base_type::date_gen_formatter_type                  ↵
date_gen_formatter_type;

    // construct/copy/destruct
    explicit time_facet(::size_t = 0);
    explicit time_facet(const char_type *,
                        period_formatter_type = period_formatter_type(),
                        const special_values_formatter_type & = special_values_formatter_type(),
                        date_gen_formatter_type = date_gen_formatter_type(),
                        ::size_t = 0);

    // public member functions
    std::locale::id & __get_id(void) const;
    void time_duration_format(const char_type *const);
    void set_iso_format();
    void set_iso_extended_format();
    OutItrT put(OutItrT, std::ios_base &, char_type, const time_type &) const;
    OutItrT put(OutItrT, std::ios_base &, char_type, const time_duration_type &) const;
    OutItrT put(OutItrT, std::ios_base &, char_type, const period_type &) const;

    // protected static functions
    static string_type
    fractional_seconds_as_string(const time_duration_type &, bool);
    static string_type hours_as_string(const time_duration_type &, int = 2);
    template<typename IntT> static string_type integral_as_string(IntT, int = 2);

    // public data members
    static const char_type * fractional_seconds_format;
    static const char_type * fractional_seconds_or_none_format;
    static const char_type * seconds_with_fractional_seconds_format;
    static const char_type * seconds_format;
    static const char_type * hours_format;
    static const char_type * unrestricted_hours_format;
    static const char_type * standard_format;
    static const char_type * zone_abbrev_format;
```

```

static const char_type * zone_name_format;
static const char_type * zone_iso_format;
static const char_type * zone_iso_extended_format;
static const char_type * posix_zone_string_format;
static const char_type * duration_seperator;
static const char_type * duration_sign_always;
static const char_type * duration_sign_negative_only;
static const char_type * negative_sign;
static const char_type * positive_sign;
static const char_type * iso_time_format_specifier;
static const char_type * iso_time_format_extended_specifier;
static const char_type * default_time_format;
static const char_type * default_time_duration_format;
static std::locale::id id;
};

```

Description

Facet used for format-based output of 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"

time_facet public construct/copy/destruct

1. `explicit time_facet(::size_t ref_arg = 0);`

sets default formats for ptime, local_date_time, and time_duration

2. `explicit time_facet(const char_type * format_arg,
period_formatter_type period_formatter_arg = period_formatter_type(),
const special_values_formatter_type & special_value_formatter = special_val-
ues_formatter_type(),
date_gen_formatter_type dg_formatter = date_gen_formatter_type(),
::size_t ref_arg = 0);`

Construct the facet with an explicitly specified format.

time_facet public member functions

1. `std::locale::id & __get_id(void) const;`

2. `void time_duration_format(const char_type *const format);`

Changes format for time_duration.

3. `void set_iso_format();`

4.

```
void set_iso_extended_format();
```
5.

```
OutItrT put(OutItrT next_arg, std::ios_base & ios_arg, char_type fill_arg,  
            const time_type & time_arg) const;
```
6.

```
OutItrT put(OutItrT next_arg, std::ios_base & ios_arg, char_type fill_arg,  
            const time_duration_type & time_dur_arg) const;
```

put function for `time_duration`
7.

```
OutItrT put(OutItrT next, std::ios_base & ios_arg, char_type fill,  
            const period_type & p) const;
```

time_facet protected static functions

1.

```
static string_type  
fractional_seconds_as_string(const time_duration_type & time_arg,  
                             bool null_when_zero);
```
2.

```
static string_type  
hours_as_string(const time_duration_type & time_arg, int width = 2);
```
3.

```
template<typename IntT>  
static string_type integral_as_string(IntT val, int width = 2);
```

Class template time_input_facet

`boost::date_time::time_input_facet` — Facet for format-based input.

Synopsis

```
// In header: <boost/date_time/time_facet.hpp>

template<typename time_type, typename CharT,
        typename InItrT = std::istreambuf_iterator<CharT, std::char_traits<CharT> > >
class time_input_facet : public boost::date_time::date_input_facet< time_type::date_type, CharT, InItrT >
{
public:
    // types
    typedef time_type::date_type date_type;
    typedef time_type::time_duration_type time_duration_type;
    typedef time_duration_type::fractional_seconds_type fractional_seconds_type;
    typedef boost::date_time::period< time_type, time_duration_type > period_type;
    typedef boost::date_time::date_input_facet< time_type::date_type, CharT, InItrT > base_type;
    typedef base_type::duration_type date_duration_type;
    typedef base_type::year_type year_type;
    typedef base_type::month_type month_type;
    typedef base_type::day_type day_type;
    typedef base_type::string_type string_type;
    typedef string_type::const_iterator const_itr;
    typedef base_type::char_type char_type;
    typedef base_type::format_date_parser_type format_date_parser_type;
    typedef base_type::period_parser_type period_parser_type;
    typedef base_type::special_values_parser_type special_values_parser_type;
    typedef base_type::date_gen_parser_type date_gen_parser_type;
    typedef base_type::special_values_parser_type::match_results match_results;

    // construct/copy/destruct
    explicit time_input_facet(const string_type &, ::size_t = 0);
    explicit time_input_facet(const string_type &,
                             const format_date_parser_type &,
                             const special_values_parser_type &,
                             const period_parser_type &,
                             const date_gen_parser_type &, ::size_t = 0);
    explicit time_input_facet(::size_t = 0);

    // public member functions
    void time_duration_format(const char_type *const);
    void set_iso_format();
    void set_iso_extended_format();
    InItrT get(InItrT &, InItrT &, std::ios_base &, period_type &) const;
    InItrT get(InItrT &, InItrT &, std::ios_base &, time_duration_type &) const;
    InItrT get(InItrT &, InItrT &, std::ios_base &, time_type &) const;
```

```

InItrT get_local_time(InItrT &, InItrT &, std::ios_base &, time_type &,
                     string_type &) const;

// protected member functions
InItrT get(InItrT &, InItrT &, std::ios_base &, time_type &, string_type &,
           bool) const;
template<typename temporal_type>
    InItrT check_special_value(InItrT &, InItrT &, temporal_type &,
                              char_type = '\\0') const;
void parse_frac_type(InItrT &, InItrT &, fractional_seconds_type &) const;

// private member functions
template<typename int_type>
    int_type decimal_adjust(int_type, const unsigned short) const;

// public data members
static const char_type * fractional_seconds_format;
static const char_type * fractional_seconds_or_none_format;
static const char_type * seconds_with_fractional_seconds_format;
static const char_type * seconds_format;
static const char_type * standard_format;
static const char_type * zone_abbrev_format;
static const char_type * zone_name_format;
static const char_type * zone_iso_format;
static const char_type * zone_iso_extended_format;
static const char_type * duration_seperator;
static const char_type * iso_time_format_specifier;
static const char_type * iso_time_format_extended_specifier;
static const char_type * default_time_input_format;
static const char_type * default_time_duration_format;
static std::locale::id id;
};

```

Description

time_input_facet public construct/copy/destruct

1. `explicit time_input_facet(const string_type & format, ::size_t ref_arg = 0);`

Constructor that takes a format string for a ptime.

2. `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 ref_arg = 0);`

3. `explicit time_input_facet(::size_t ref_arg = 0);`

sets default formats for ptime, local_date_time, and `time_duration`

time_input_facet public member functions

1. `void time_duration_format(const char_type *const format);`

Set the format for `time_duration`.

2.

```
void set_iso_format();
```
3.

```
void set_iso_extended_format();
```
4.

```
InItrT get(InItrT & sitr, InItrT & stream_end, std::ios_base & ios_arg,
           period_type & p) const;
```
5.

```
InItrT get(InItrT & sitr, InItrT & stream_end, std::ios_base & ios_arg,
           time_duration_type & td) const;
```
6.

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

Parses a time object from the input stream.

7.

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

Expects a time_zone in the input stream.

time_input_facet protected member functions

1.

```
InItrT get(InItrT & sitr, InItrT & stream_end, std::ios_base & ios_arg,
           time_type & t, string_type & tz_str, bool time_is_local) const;
```
2.

```
template<typename temporal_type>
InItrT check_special_value(InItrT & sitr, InItrT & stream_end,
                          temporal_type & tt, char_type c = '\\0') const;
```

Helper function to check for special_value.

First character may have been consumed during original parse attempt. Parameter 'c' should be a copy of that character. Throws ios_base::failure if parse fails.

3.

```
void parse_frac_type(InItrT & sitr, InItrT & stream_end,
                    fractional_seconds_type & frac) const;
```

Helper function for parsing a fractional second type from the stream.

time_input_facet private member functions

1.

```
template<typename int_type>
int_type decimal_adjust(int_type val, const unsigned short places) const;
```

Helper function to adjust trailing zeros when parsing fractional digits.

Header `<boost/date_time/time_formatting_streams.hpp>`

```
namespace boost {
    namespace date_time {
        template<typename time_duration_type, typename charT = char>
            class ostream_time_duration_formatter;
        template<typename time_type, typename charT = char>
            class ostream_time_formatter;
        template<typename time_period_type, typename charT = char>
            class ostream_time_period_formatter;
    }
}
```

Class template `ostream_time_duration_formatter`

`boost::date_time::ostream_time_duration_formatter` — Put a time type into a stream using appropriate facets.

Synopsis

```
// In header: <boost/date_time/time_formatting_streams.hpp>

template<typename time_duration_type, typename charT = char>
class ostream_time_duration_formatter {
public:
    // types
    typedef std::basic_ostream< charT > ostream_type;
    typedef time_duration_type::fractional_seconds_type fractional_seconds_type;

    // public static functions
    static void duration_put(const time_duration_type &, ostream_type &);
};
```

Description

`ostream_time_duration_formatter` public static functions

1. `static void duration_put(const time_duration_type & td, ostream_type & os);`

Put time into an ostream.

Class template `ostream_time_formatter`

`boost::date_time::ostream_time_formatter` — Put a time type into a stream using appropriate facets.

Synopsis

```
// In header: <boost/date_time/time_formatting_streams.hpp>

template<typename time_type, typename charT = char>
class ostream_time_formatter {
public:
    // types
    typedef std::basic_ostream< charT >          ostream_type;
    typedef time_type::date_type                 date_type;
    typedef time_type::time_duration_type        time_duration_type;
    typedef ostream_time_duration_formatter< time_duration_type, charT > duration_formatter;

    // public static functions
    static void time_put(const time_type &, ostream_type &);
};
```

Description

ostream_time_formatter public static functions

1. `static void time_put(const time_type & t, ostream_type & os);`

Put time into an ostream.

Class template ostream_time_period_formatter

boost::date_time::ostream_time_period_formatter — Put a time period into a stream using appropriate facets.

Synopsis

```
// In header: <boost/date_time/time_formatting_streams.hpp>

template<typename time_period_type, typename charT = char>
class ostream_time_period_formatter {
public:
    // types
    typedef std::basic_ostream< charT >          ostream_type;
    typedef time_period_type::point_type        time_type;
    typedef ostream_time_formatter< time_type, charT > time_formatter;

    // public static functions
    static void period_put(const time_period_type &, ostream_type &);
};
```

Description

ostream_time_period_formatter public static functions

1. `static void period_put(const time_period_type & tp, ostream_type & os);`

Put time into an ostream.

Header `<boost/date_time/time_iterator.hpp>`

```
namespace boost {
    namespace date_time {
        template<typename time_type> class time_itr;
    }
}
```

Class template `time_itr`

`boost::date_time::time_itr` — Simple time iterator skeleton class.

Synopsis

```
// In header: <boost/date_time/time_iterator.hpp>

template<typename time_type>
class time_itr {
public:
    // types
    typedef time_type::time_duration_type time_duration_type;

    // construct/copy/destruct
    time_itr(time_type, time_duration_type);

    // public member functions
    time_itr & operator++();
    time_itr & operator--();
    time_type operator*();
    time_type * operator->();
    bool operator<(const time_type &);
    bool operator<=(const time_type &);
    bool operator!=(const time_type &);
    bool operator==(const time_type &);
    bool operator>(const time_type &);
    bool operator>=(const time_type &);
};
```

Description

`time_itr` public construct/copy/destruct

1. `time_itr(time_type t, time_duration_type d);`

`time_itr` public member functions

1. `time_itr & operator++();`
2. `time_itr & operator--();`
3. `time_type operator*();`

4. `time_type * operator->();`
5. `bool operator<(const time_type & t);`
6. `bool operator<=(const time_type & t);`
7. `bool operator!=(const time_type & t);`
8. `bool operator==(const time_type & t);`
9. `bool operator>(const time_type & t);`
10. `bool operator>=(const time_type & t);`

Header `<boost/date_time/time_parsing.hpp>`

```
namespace boost {
    namespace date_time {

        // computes exponential math like 2^8 => 256, only works with positive integers
        template<typename int_type>
        int_type power(int_type base, int_type exponent);
        template<typename time_duration, typename char_type>
        time_duration
        str_from_delimited_time_duration(const std::basic_string< char_type > &);
        template<typename time_duration>
        time_duration parse_delimited_time_duration(const std::string &);

        // Utility function to split appart string.
        bool split(const std::string & s, char sep, std::string & first,
                  std::string & second);
        template<typename time_type>
        time_type parse_delimited_time(const std::string & s, char sep);

        // Parse time duration part of an iso time of form: [-]hhmmss[.fff...] (eg: 120259.123 is 12
        // hours, 2 min, 59 seconds, 123000 microseconds)
        template<typename time_duration>
        time_duration parse_undelimited_time_duration(const std::string & s);

        // Parse time string of form YYYYMMDDThhmmss where T is delimiter between date and time.
        template<typename time_type>
        time_type parse_iso_time(const std::string & s, char sep);
    }
}
```

Function template `str_from_delimited_time_duration`

`boost::date_time::str_from_delimited_time_duration` — Creates a [time_duration](#) object from a delimited string.

Synopsis

```
// In header: <boost/date_time/time_parsing.hpp>

template<typename time_duration, typename char_type>
    time_duration
    str_from_delimited_time_duration(const std::basic_string< char_type > & s);
```

Description

Expected format for string is "[~]h[h][:mm][:ss][.fff]". If the number of fractional digits provided is greater than the precision of the time duration type then the extra digits are truncated.

A negative duration will be created if the first character in string is a '-', all other '-' will be treated as delimiters. Accepted delimiters are "-:.,".

Function template `parse_delimited_time_duration`

`boost::date_time::parse_delimited_time_duration` — Creates a [time_duration](#) object from a delimited string.

Synopsis

```
// In header: <boost/date_time/time_parsing.hpp>

template<typename time_duration>
    time_duration parse_delimited_time_duration(const std::string & s);
```

Description

Expected format for string is "[~]h[h][:mm][:ss][.fff]". If the number of fractional digits provided is greater than the precision of the time duration type then the extra digits are truncated.

A negative duration will be created if the first character in string is a '-', all other '-' will be treated as delimiters. Accepted delimiters are "-:.,".

Header `<boost/date_time/time_resolution_traits.hpp>`

```
namespace boost {
    namespace date_time {
        struct time_resolution_traits_bi32_impl;
        struct time_resolution_traits_adapted32_impl;
        struct time_resolution_traits_bi64_impl;
        struct time_resolution_traits_adapted64_impl;

        template<typename frac_sec_type, time_resolutions res,
                #if defined(BOOST_MSVC) && (_MSC_VER < 1300) boost::int64_t resolution_adjust,
                #else typename frac_sec_type::int_type resolution_adjust,
                #endif unsigned short frac_digits,
                typename v_type = boost::int32_t>
        class time_resolution_traits;

        typedef time_resolution_traits< time_resolution_traits_adapted32_impl, milli, 1000, 3 > milli_res;
        typedef time_resolution_traits< time_resolution_traits_adapted64_impl, micro, 1000000, 6 > micro_res;
        typedef time_resolution_traits< time_resolution_traits_adapted64_impl, nano, 1000000000, 9 > nano_res;

        // Simple function to calculate absolute value of a numeric type.
        template<typename T> T absolute_value(T x);
    }
}
```

Struct `time_resolution_traits_bi32_impl`

`boost::date_time::time_resolution_traits_bi32_impl` — traits struct for `time_resolution_traits` implementation type

Synopsis

```
// In header: <boost/date_time/time_resolution_traits.hpp>

struct time_resolution_traits_bi32_impl {
    // types
    typedef boost::int32_t int_type;
    typedef boost::int32_t impl_type;

    // public static functions
    static int_type as_number(impl_type);
    static bool is_adapted();
};
```

Description

`time_resolution_traits_bi32_impl` public static functions

1. `static int_type as_number(impl_type i);`

2. `static bool is_adapted();`

Used to determine if implemented type is `int_adapter` or `int`.

Struct `time_resolution_traits_adapted32_impl`

`boost::date_time::time_resolution_traits_adapted32_impl` — traits struct for [time_resolution_traits](#) implementation type

Synopsis

```
// In header: <boost/date_time/time_resolution_traits.hpp>

struct time_resolution_traits_adapted32_impl {
    // types
    typedef boost::int32_t          int_type;
    typedef boost::date_time::int_adapter< boost::int32_t > impl_type;

    // public static functions
    static int_type as_number(impl_type);
    static bool is_adapted();
};
```

Description

`time_resolution_traits_adapted32_impl` public static functions

1. `static int_type as_number(impl_type i);`
2. `static bool is_adapted();`

Used to determine if implemented type is [int_adapter](#) or `int`.

Struct `time_resolution_traits_bi64_impl`

`boost::date_time::time_resolution_traits_bi64_impl` — traits struct for [time_resolution_traits](#) implementation type

Synopsis

```
// In header: <boost/date_time/time_resolution_traits.hpp>

struct time_resolution_traits_bi64_impl {
    // types
    typedef boost::int64_t int_type;
    typedef boost::int64_t impl_type;

    // public static functions
    static int_type as_number(impl_type);
    static bool is_adapted();
};
```

Description

`time_resolution_traits_bi64_impl` public static functions

1. `static int_type as_number(impl_type i);`

2. `static bool is_adapted();`

Used to determine if implemented type is [int_adapter](#) or `int`.

Struct `time_resolution_traits_adapted64_impl`

`boost::date_time::time_resolution_traits_adapted64_impl` — traits struct for [time_resolution_traits](#) implementation type

Synopsis

```
// In header: <boost/date_time/time_resolution_traits.hpp>

struct time_resolution_traits_adapted64_impl {
    // types
    typedef boost::int64_t                int_type;
    typedef boost::date_time::int_adapter< boost::int64_t > impl_type;

    // public static functions
    static int_type as_number(impl_type);
    static bool is_adapted();
};
```

Description

`time_resolution_traits_adapted64_impl` public static functions

1. `static int_type as_number(impl_type i);`

2. `static bool is_adapted();`

Used to determine if implemented type is [int_adapter](#) or `int`.

Class template `time_resolution_traits`

`boost::date_time::time_resolution_traits`

Synopsis

```
// In header: <boost/date_time/time_resolution_traits.hpp>

template<typename frac_sec_type, time_resolutions res,
        #if defined(BOOST_MSVC) && (_MSC_VER < 1300) boost::int64_t resolution_adjust,
        #else typename frac_sec_type::int_type resolution_adjust,
        #endif unsigned short frac_digits, typename v_type = boost::int32_t>
class time_resolution_traits {
public:
    // types
    typedef frac_sec_type::int_type    fractional_seconds_type;
    typedef frac_sec_type::int_type    tick_type;
    typedef frac_sec_type::impl_type    impl_type;
    typedef v_type                     day_type;
    typedef v_type                     hour_type;
    typedef v_type                     min_type;
    typedef v_type                     sec_type;

    // public static functions
    static fractional_seconds_type as_number(impl_type i);
    static bool is_adapted();
    static time_resolutions resolution();
    static unsigned short num_fractional_digits();
    static fractional_seconds_type res_adjust();
    static tick_type
    to_tick_count(hour_type hours, min_type minutes, sec_type seconds, fractional_seconds_type fs);

    // public member functions
    BOOST_STATIC_CONSTANT(boost::int64_t, ticks_per_second = resolution_adjust);
    BOOST_STATIC_CONSTANT(fractional_seconds_type,
                          ticks_per_second = resolution_adjust);
};
```

Description

time_resolution_traits public static functions

1. `static fractional_seconds_type as_number(impl_type i);`
2. `static bool is_adapted();`
3. `static time_resolutions resolution();`
4. `static unsigned short num_fractional_digits();`
5. `static fractional_seconds_type res_adjust();`
6. `static tick_type
to_tick_count(hour_type hours, min_type minutes, sec_type seconds,
 fractional_seconds_type fs);`

Any negative argument results in a negative tick_count.

time_resolution_traits public member functions

1.

```
BOOST_STATIC_CONSTANT(boost::int64_t, ticks_per_second = resolution_adjust);
```
2.

```
BOOST_STATIC_CONSTANT(fractional_seconds_type,  
                      ticks_per_second = resolution_adjust);
```

Header <boost/date_time/time_system_counted.hpp>

```
namespace boost {  
    namespace date_time {  
        template<typename config> struct counted_time_rep;  
  
        template<typename time_rep> class counted_time_system;  
    }  
}
```

Struct template counted_time_rep

boost::date_time::counted_time_rep — Time representation that uses a single integer count.

Synopsis

```
// In header: <boost/date_time/time_system_counted.hpp>

template<typename config>
struct counted_time_rep {
    // types
    typedef config::int_type          int_type;
    typedef config::date_type         date_type;
    typedef config::impl_type         impl_type;
    typedef date_type::duration_type  date_duration_type;
    typedef date_type::calendar_type  calendar_type;
    typedef date_type::ymd_type       ymd_type;
    typedef config::time_duration_type time_duration_type;
    typedef config::resolution_traits resolution_traits;

    // construct/copy/destroy
    counted_time_rep(const date_type &, const time_duration_type &);
    explicit counted_time_rep(int_type);
    explicit counted_time_rep(impl_type);

    // public member functions
    date_type date() const;
    unsigned long day_count() const;
    int_type time_count() const;
    int_type tod() const;
    bool is_pos_infinity() const;
    bool is_neg_infinity() const;
    bool is_not_a_date_time() const;
    bool is_special() const;
    impl_type get_rep() const;

    // public static functions
    static int_type frac_sec_per_day();
};
```

Description

counted_time_rep public construct/copy/destroy

1. `counted_time_rep(const date_type & d, const time_duration_type & time_of_day);`
2. `explicit counted_time_rep(int_type count);`
3. `explicit counted_time_rep(impl_type count);`

counted_time_rep public member functions

1. `date_type date() const;`
2. `unsigned long day_count() const;`

3. `int_type time_count() const;`

4. `int_type tod() const;`

5. `bool is_pos_infinity() const;`

6. `bool is_neg_infinity() const;`

7. `bool is_not_a_date_time() const;`

8. `bool is_special() const;`

9. `impl_type get_rep() const;`

counted_time_rep public static functions

1. `static int_type frac_sec_per_day();`

Class template counted_time_system

boost::date_time::counted_time_system — An unadjusted time system implementation.

Synopsis

```
// In header: <boost/date_time/time_system_counted.hpp>

template<typename time_rep>
class counted_time_system {
public:
    // types
    typedef time_rep                time_rep_type;
    typedef time_rep_type::impl_type impl_type;
    typedef time_rep_type::time_duration_type time_duration_type;
    typedef time_duration_type::fractional_seconds_type fractional_seconds_type;
    typedef time_rep_type::date_type date_type;
    typedef time_rep_type::date_duration_type date_duration_type;

    // public static functions
    template<typename T> static void unused_var(const T &);
    static time_rep_type
    get_time_rep(const date_type &, const time_duration_type &,
                 date_time::dst_flags = not_dst);
    static time_rep_type get_time_rep(special_values);
    static date_type get_date(const time_rep_type &);
    static time_duration_type get_time_of_day(const time_rep_type &);
    static std::string zone_name(const time_rep_type &);
    static bool is_equal(const time_rep_type &, const time_rep_type &);
    static bool is_less(const time_rep_type &, const time_rep_type &);
    static time_rep_type
    add_days(const time_rep_type &, const date_duration_type &);
    static time_rep_type
    subtract_days(const time_rep_type &, const date_duration_type &);
    static time_rep_type
    subtract_time_duration(const time_rep_type &, const time_duration_type &);
    static time_rep_type
    add_time_duration(const time_rep_type &, time_duration_type);
    static time_duration_type
    subtract_times(const time_rep_type &, const time_rep_type &);
};
```

Description

counted_time_system public static functions

1.

```
template<typename T> static void unused_var(const T &);
```
2.

```
static time_rep_type
get_time_rep(const date_type & day, const time_duration_type & tod,
             date_time::dst_flags dst = not_dst);
```
3.

```
static time_rep_type get_time_rep(special_values sv);
```
4.

```
static date_type get_date(const time_rep_type & val);
```
5.

```
static time_duration_type get_time_of_day(const time_rep_type & val);
```

6.

```
static std::string zone_name(const time_rep_type &);
```
7.

```
static bool is_equal(const time_rep_type & lhs, const time_rep_type & rhs);
```
8.

```
static bool is_less(const time_rep_type & lhs, const time_rep_type & rhs);
```
9.

```
static time_rep_type  
add_days(const time_rep_type & base, const date_duration_type & dd);
```
10.

```
static time_rep_type  
subtract_days(const time_rep_type & base, const date_duration_type & dd);
```
11.

```
static time_rep_type  
subtract_time_duration(const time_rep_type & base,  
                       const time_duration_type & td);
```
12.

```
static time_rep_type  
add_time_duration(const time_rep_type & base, time_duration_type td);
```
13.

```
static time_duration_type  
subtract_times(const time_rep_type & lhs, const time_rep_type & rhs);
```

Header <boost/date_time/time_system_split.hpp>

```
namespace boost {  
    namespace date_time {  
        template<typename config, boost::int32_t ticks_per_second>  
            class split_timedate_system;  
    }  
}
```

Class template split_timedate_system

boost::date_time::split_timedate_system — An unadjusted time system implementation.

Synopsis

```
// In header: <boost/date_time/time_system_split.hpp>

template<typename config, boost::int32_t ticks_per_second>
class split_timedate_system {
public:
    // types
    typedef config::time_rep_type          time_rep_type;
    typedef config::date_type              date_type;
    typedef config::time_duration_type     time_duration_type;
    typedef config::date_duration_type     date_duration_type;
    typedef config::int_type               int_type;
    typedef config::resolution_traits      resolution_traits;
    typedef date_time::wrapping_int< int_type, INT64_C(86400)*ticks_per_second >
        wrap_int_type;
    typedef date_time::wrapping_int< split_timedate_system::int_type, split_timedate_sys-
tem::ticks_per_day > wrap_int_type;
    typedef date_time::wrapping_int< int_type, ticks_per_day >
        wrap_int_type;

    // private member functions
    BOOST_STATIC_CONSTANT(int_type,
        ticks_per_day = INT64_C(86400)*config::tick_per_second);

    // public static functions
    static time_rep_type get_time_rep(special_values);
    static time_rep_type
    get_time_rep(const date_type &, const time_duration_type &,
        date_time::dst_flags = not_dst);
    static date_type get_date(const time_rep_type &);
    static time_duration_type get_time_of_day(const time_rep_type &);
    static std::string zone_name(const time_rep_type &);
    static bool is_equal(const time_rep_type &, const time_rep_type &);
    static bool is_less(const time_rep_type &, const time_rep_type &);
    static time_rep_type
    add_days(const time_rep_type &, const date_duration_type &);
    static time_rep_type
    subtract_days(const time_rep_type &, const date_duration_type &);
    static time_rep_type
    subtract_time_duration(const time_rep_type &, const time_duration_type &);
    static time_rep_type
    add_time_duration(const time_rep_type &, time_duration_type);
    static time_duration_type
    subtract_times(const time_rep_type &, const time_rep_type &);
};
```

Description

split_timedate_system private member functions

1. BOOST_STATIC_CONSTANT(int_type,
ticks_per_day = INT64_C(86400)*config::tick_per_second);

split_timedate_system public static functions

1.

```
static time_rep_type get_time_rep(special_values sv);
```
2.

```
static time_rep_type  
get_time_rep(const date_type & day, const time_duration_type & tod,  
             date_time::dst_flags = not_dst);
```
3.

```
static date_type get_date(const time_rep_type & val);
```
4.

```
static time_duration_type get_time_of_day(const time_rep_type & val);
```
5.

```
static std::string zone_name(const time_rep_type &);
```
6.

```
static bool is_equal(const time_rep_type & lhs, const time_rep_type & rhs);
```
7.

```
static bool is_less(const time_rep_type & lhs, const time_rep_type & rhs);
```
8.

```
static time_rep_type  
add_days(const time_rep_type & base, const date_duration_type & dd);
```
9.

```
static time_rep_type  
subtract_days(const time_rep_type & base, const date_duration_type & dd);
```
10.

```
static time_rep_type  
subtract_time_duration(const time_rep_type & base,  
                      const time_duration_type & td);
```
11.

```
static time_rep_type  
add_time_duration(const time_rep_type & base, time_duration_type td);
```
12.

```
static time_duration_type  
subtract_times(const time_rep_type & lhs, const time_rep_type & rhs);
```

Header `<boost/date_time/time_zone_base.hpp>`

```
namespace boost {
    namespace date_time {
        template<typename time_type, typename CharT> class time_zone_base;
        template<typename time_duration_type> class dst_adjustment_offsets;
    }
}
```

Class template `time_zone_base`

`boost::date_time::time_zone_base` — Interface class for dynamic time zones.

Synopsis

```
// In header: <boost/date_time/time_zone_base.hpp>

template<typename time_type, typename CharT>
class time_zone_base {
public:
    // types
    typedef CharT char_type;
    typedef std::basic_string< CharT > string_type;
    typedef std::basic_ostringstream< CharT > stringstream_type;
    typedef time_type::date_type::year_type year_type;
    typedef time_type::time_duration_type time_duration_type;

    // construct/copy/destruct
    time_zone_base();
    ~time_zone_base();

    // public member functions
    string_type dst_zone_abbrev() const;
    string_type std_zone_abbrev() const;
    string_type dst_zone_name() const;
    string_type std_zone_name() const;
    bool has_dst() const;
    time_type dst_local_start_time(year_type) const;
    time_type dst_local_end_time(year_type) const;
    time_duration_type base_utc_offset() const;
    time_duration_type dst_offset() const;
    string_type to_posix_string() const;
};
```

Description

This class represents the base interface for all timezone representations. Subclasses may provide different systems for identifying a particular zone. For example some may provide a geographical based zone construction while others may specify the offset from GMT. Another possible implementation would be to convert from POSIX timezone strings. Regardless of the construction technique, this is the interface that these time zone types must provide.

Note that this class is intended to be used as a shared resource (hence the derivation from `boost::counted_base`).

`time_zone_base` public construct/copy/destruct

1. `time_zone_base();`

2. `~time_zone_base();`

time_zone_base public member functions

1. `string_type dst_zone_abbrev() const;`

String for the timezone when in daylight savings (eg: EDT)

2. `string_type std_zone_abbrev() const;`

String for the zone when not in daylight savings (eg: EST)

3. `string_type dst_zone_name() const;`

String for the timezone when in daylight savings (eg: Eastern Daylight Time)

4. `string_type std_zone_name() const;`

String for the zone when not in daylight savings (eg: Eastern Standard Time)

5. `bool has_dst() const;`

True if zone uses daylight savings adjustments otherwise false.

6. `time_type dst_local_start_time(year_type y) const;`

Local time that DST starts -- undefined if has_dst is false.

7. `time_type dst_local_end_time(year_type y) const;`

Local time that DST ends -- undefined if has_dst is false.

8. `time_duration_type base_utc_offset() const;`

Base offset from UTC for zone (eg: -07:30:00)

9. `time_duration_type dst_offset() const;`

Adjustment forward or back made while DST is in effect.

10. `string_type to_posix_string() const;`

Returns a POSIX time_zone string for this object.

Class template dst_adjustment_offsets

boost::date_time::dst_adjustment_offsets — Structure which holds the time offsets associated with daylight savings time.

Synopsis

```
// In header: <boost/date_time/time_zone_base.hpp>

template<typename time_duration_type>
class dst_adjustment_offsets {
public:
    // construct/copy/destroy
    dst_adjustment_offsets(const time_duration_type &,
                          const time_duration_type &,
                          const time_duration_type &);

    // public data members
    time_duration_type dst_adjust_; // Amount DST adjusts the clock eg: plus one hour.
    time_duration_type dst_start_offset_; // Time past midnight on start transition day that dst
    starts.
    time_duration_type dst_end_offset_; // Time past midnight on end transition day that dst ends.
};
```

Description

dst_adjustment_offsets public construct/copy/destroy

1.

```
dst_adjustment_offsets(const time_duration_type & dst_adjust,
                      const time_duration_type & dst_start_offset,
                      const time_duration_type & dst_end_offset);
```

Header <boost/date_time/time_zone_names.hpp>

```
namespace boost {
    namespace date_time {
        template<typename CharT> struct default_zone_names;

        template<typename CharT> class time_zone_names_base;
    }
}
```

Struct template default_zone_names

boost::date_time::default_zone_names

Synopsis

```
// In header: <boost/date_time/time_zone_names.hpp>

template<typename CharT>
struct default_zone_names {
    // types
    typedef CharT char_type;

    // public data members
    static const char_type standard_name;
    static const char_type standard_abbrev;
    static const char_type non_dst_identifier;
};
```

Class template `time_zone_names_base`

`boost::date_time::time_zone_names_base` — Base type that holds various string names for timezone output.

Synopsis

```
// In header: <boost/date_time/time_zone_names.hpp>

template<typename CharT>
class time_zone_names_base {
public:
    // types
    typedef std::basic_string< CharT > string_type;

    // construct/copy/destruct
    time_zone_names_base();
    time_zone_names_base(const string_type &, const string_type &,
                        const string_type &, const string_type &);

    // public member functions
    string_type dst_zone_abbrev() const;
    string_type std_zone_abbrev() const;
    string_type dst_zone_name() const;
    string_type std_zone_name() const;
};
```

Description

Class that holds various types of strings used for timezones. For example, for the western United States there is the full name: Pacific Standard Time and the abbreviated name: PST. During daylight savings there are additional names: Pacific Daylight Time and PDT. CharT Allows class to support different character types

`time_zone_names_base` public construct/copy/destruct

1. `time_zone_names_base();`
2. `time_zone_names_base(const string_type & std_zone_name_str,
 const string_type & std_zone_abbrev_str,
 const string_type & dst_zone_name_str,
 const string_type & dst_zone_abbrev_str);`

`time_zone_names_base` public member functions

1. `string_type dst_zone_abbrev() const;`
2. `string_type std_zone_abbrev() const;`
3. `string_type dst_zone_name() const;`
4. `string_type std_zone_name() const;`

Header `<boost/date_time/tz_db_base.hpp>`

```
namespace boost {
    namespace date_time {
        class data_not_accessible;
        class bad_field_count;
        template<typename time_zone_type, typename rule_type> class tz_db_base;
    }
}
```

Class `data_not_accessible`

`boost::date_time::data_not_accessible` — Exception thrown when tz database cannot locate requested data file.

Synopsis

```
// In header: <boost/date_time/tz_db_base.hpp>

class data_not_accessible {
public:
    // construct/copy/destroy
    data_not_accessible();
    data_not_accessible(const std::string &);
};
```

Description

`data_not_accessible` **public construct/copy/destroy**

1. `data_not_accessible();`
2. `data_not_accessible(const std::string & filespec);`

Class `bad_field_count`

`boost::date_time::bad_field_count` — Exception thrown when tz database locates incorrect field structure in data file.

Synopsis

```
// In header: <boost/date_time/tz_db_base.hpp>

class bad_field_count {
public:
    // construct/copy/destroy
    bad_field_count(const std::string &);
};
```

Description

`bad_field_count` **public construct/copy/destruct**

```
1. bad_field_count(const std::string & s);
```

Class template `tz_db_base`

`boost::date_time::tz_db_base` — Creates a database of time_zones from csv datafile.

Synopsis

```
// In header: <boost/date_time/tz_db_base.hpp>

template<typename time_zone_type, typename rule_type>
class tz_db_base {
public:
    // types
    typedef char                                char_type;
    typedef time_zone_type::base_type           time_zone_base_type;
    typedef time_zone_type::time_duration_type time_duration_type;
    typedef time_zone_names_base< char_type >    time_zone_names;
    typedef boost::date_time::dst_adjustment_offsets< time_duration_type > dst_adjustment_offsets;
    typedef std::basic_string< char_type >        string_type;

    // construct/copy/destruct
    tz_db_base();

    // public member functions
    void load_from_stream(std::istream &);
    void load_from_file(const std::string &);
    bool add_record(const string_type &,
                   boost::shared_ptr< time_zone_base_type >);
    boost::shared_ptr< time_zone_base_type >
    time_zone_from_region(const string_type &) const;
    std::vector< std::string > region_list() const;

    // private member functions
    rule_type * parse_rules(const string_type &, const string_type &) const;
    week_num get_week_num(int) const;
    void split_rule_spec(int &, int &, int &, string_type) const;
    bool parse_string(string_type &);
};
```

Description

The csv file containing the zone_specs used by the `tz_db_base` is intended to be customized by the library user. When customizing this file (or creating your own) the file must follow a specific format.

This first line is expected to contain column headings and is therefore not processed by the `tz_db_base`.

Each record (line) must have eleven fields. Some of those fields can be empty. Every field (even empty ones) must be enclosed in double-quotes. Ex:

```
"America/Phoenix" <- string enclosed in quotes
" "              <- empty field
```

Some fields represent a length of time. The format of these fields must be:

```
"{+|-}hh:mm[:ss]" <- length-of-time format
```

Where the plus or minus is mandatory and the seconds are optional.

Since some time zones do not use daylight savings it is not always necessary for every field in a zone_spec to contain a value. All zone_specs must have at least ID and GMT offset. Zones that use daylight savings must have all fields filled except: STD ABBR, STD NAME, DST NAME. You should take note that DST ABBR is mandatory for zones that use daylight savings (see field descriptions for further details).

***** Fields and their description/details *****

ID: Contains the identifying string for the zone_spec. Any string will do as long as it's unique. No two ID's can be the same.

STD ABBR: STD NAME: DST ABBR: DST NAME: These four are all the names and abbreviations used by the time zone being described. While any string will do in these fields, care should be taken. These fields hold the strings that will be used in the output of many of the local_time classes. Ex:

```
time_zone nyc = tz_db.time_zone_from_region("America/New_York");
local_time ny_time(date(2004, Aug, 30), IS_DST, nyc);
cout << ny_time.to_long_string() << endl;
// 2004-Aug-30 00:00:00 Eastern Daylight Time
cout << ny_time.to_short_string() << endl;
// 2004-Aug-30 00:00:00 EDT
```

NOTE: The exact format/function names may vary - see local_time documentation for further details.

GMT offset: This is the number of hours added to utc to get the local time before any daylight savings adjustments are made. Some examples are: America/New_York offset -5 hours, & Africa/Cairo offset +2 hours. The format must follow the length-of-time format described above.

DST adjustment: The amount of time added to gmt_offset when daylight savings is in effect. The format must follow the length-of-time format described above.

DST Start Date rule: This is a specially formatted string that describes the day of year in which the transition take place. It holds three fields of it's own, separated by semicolons. The first field indicates the "nth" weekday of the month. The possible values are: 1 (first), 2 (second), 3 (third), 4 (fourth), 5 (fifth), and -1 (last). The second field indicates the day-of-week from 0-6 (Sun=0). The third field indicates the month from 1-12 (Jan=1).

Examples are: "-1;5;9"="Last Friday of September", "2;1;3"="Second Monday of March"

Start time: Start time is the number of hours past midnight, on the day of the start transition, the transition takes place. More simply put, the time of day the transition is made (in 24 hours format). The format must follow the length-of-time format described above with the exception that it must always be positive.

DST End date rule: See DST Start date rule. The difference here is this is the day daylight savings ends (transition to STD).

End time: Same as Start time.

tz_db_base public construct/copy/destruct

```
1. tz_db_base();
```

Constructs an empty database.

tz_db_base public member functions

```
1. void load_from_stream(std::istream & in);
```

Process csv data file, may throw exceptions.

May throw `bad_field_count` exceptions

```
2. void load_from_file(const std::string & pathspec);
```

Process csv data file, may throw exceptions.

May throw `data_not_accessible`, or `bad_field_count` exceptions

```
3. bool add_record(const string_type & region,
                  boost::shared_ptr< time_zone_base_type > tz);
```

returns true if record successfully added to map

Takes a region name in the form of "America/Phoenix", and a time_zone object for that region. The id string must be a unique name that does not already exist in the database.

```
4. boost::shared_ptr< time_zone_base_type >
   time_zone_from_region(const string_type & region) const;
```

Returns a time_zone object built from the specs for the given region.

Returns a time_zone object built from the specs for the given region. If region does not exist a `local_time::record_not_found` exception will be thrown

```
5. std::vector< std::string > region_list() const;
```

Returns a vector of strings holding the time zone regions in the database.

tz_db_base private member functions

```
1. rule_type * parse_rules(const string_type & sr, const string_type & er) const;
```

parses rule specs for transition day rules

```
2. week_num get_week_num(int nth) const;
```

helper function for parse_rules()

```
3. void split_rule_spec(int & nth, int & d, int & m, string_type rule) const;
```

splits the [start|end]_date_rule string into 3 ints

```
4. bool parse_string(string_type & s);
```

Take a line from the csv, turn it into a time_zone_type.

Take a line from the csv, turn it into a time_zone_type, and add it to the map. Zone_specs in csv file are expected to have eleven fields that describe the time zone. Returns true if zone_spec successfully added to database

Header `<boost/date_time/wrapping_int.hpp>`

```
namespace boost {
    namespace date_time {
        template<typename int_type_, int_type_ wrap_val> class wrapping_int;
        template<typename int_type_, int_type_ wrap_min, int_type_ wrap_max>
            class wrapping_int2;
    }
}
```

Class template `wrapping_int`

`boost::date_time::wrapping_int` — A wrapping integer used to support time durations (WARNING: only instantiate with a signed type)

Synopsis

```
// In header: <boost/date_time/wrapping_int.hpp>

template<typename int_type_, int_type_ wrap_val>
class wrapping_int {
public:
    // types
    typedef int_type_ int_type;

    // construct/copy/destroy
    wrapping_int(int_type);

    // public static functions
    static int_type wrap_value();

    // public member functions
    int_type as_int() const;
    operator int_type() const;
    template<typename IntT> IntT add(IntT);
    template<typename IntT> IntT subtract(IntT);

    // private member functions
    template<typename IntT> IntT calculate_wrap(IntT);
};
```

Description

In composite date and time types this type is used to wrap at the day boundary. Ex: A `wrapping_int<short, 10>` will roll over after nine, and roll under below zero. This gives a range of [0,9]

NOTE: it is strongly recommended that `wrapping_int2` be used instead of `wrapping_int` as `wrapping_int` is to be deprecated at some point soon.

Also Note that warnings will occur if instantiated with an unsigned type. Only a signed type should be used!

`wrapping_int` public construct/copy/destroy

1. `wrapping_int(int_type v);`

Add, return true if wrapped.

wrapping_int public static functions

1.

```
static int_type wrap_value();
```

wrapping_int public member functions

1.

```
int_type as_int() const;
```

Explicit conversion method.

2.

```
operator int_type() const;
```

3.

```
template<typename IntT> IntT add(IntT v);
```

Add, return number of wraps performed.

The sign of the returned value will indicate which direction the wraps went. Ex: add a negative number and wrapping under could occur, this would be indicated by a negative return value. If wrapping over took place, a positive value would be returned

4.

```
template<typename IntT> IntT subtract(IntT v);
```

Subtract will return '+d' if wrapping under took place ('d' is the number of wraps)

The sign of the returned value will indicate which direction the wraps went (positive indicates wrap under, negative indicates wrap over). Ex: subtract a negative number and wrapping over could occur, this would be indicated by a negative return value. If wrapping under took place, a positive value would be returned.

wrapping_int private member functions

1.

```
template<typename IntT> IntT calculate_wrap(IntT wrap);
```

Class template wrapping_int2

boost::date_time::wrapping_int2 — A wrapping integer used to wrap around at the top (WARNING: only instantiate with a signed type)

Synopsis

```
// In header: <boost/date_time/wrapping_int.hpp>

template<typename int_type_, int_type_ wrap_min, int_type_ wrap_max>
class wrapping_int2 {
public:
    // types
    typedef int_type_ int_type;

    // construct/copy/destruct
    wrapping_int2(int_type);

    // public static functions
    static int_type wrap_value();
    static int_type min_value();

    // public member functions
    int_type as_int() const;
    operator int_type() const;
    template<typename IntT> IntT add(IntT);
    template<typename IntT> IntT subtract(IntT);

    // private member functions
    template<typename IntT> IntT calculate_wrap(IntT);
};
```

Description

Bad name, quick impl to fix a bug -- fix later!! This allows the wrap to restart at a value other than 0.

wrapping_int2 public construct/copy/destruct

1. `wrapping_int2(int_type v);`

If initializing value is out of range of [wrap_min, wrap_max], value will be initialized to closest of min or max

wrapping_int2 public static functions

1. `static int_type wrap_value();`
2. `static int_type min_value();`

wrapping_int2 public member functions

1. `int_type as_int() const;`

Explicit conversion method.

2. `operator int_type() const;`

3. `template<typename IntT> IntT add(IntT v);`

Add, return number of wraps performed.

The sign of the returned value will indicate which direction the wraps went. Ex: add a negative number and wrapping under could occur, this would be indicated by a negative return value. If wrapping over took place, a positive value would be returned

4.

```
template<typename IntT> IntT subtract(IntT v);
```

Subtract will return '-d' if wrapping under took place ('d' is the number of wraps)

The sign of the returned value will indicate which direction the wraps went. Ex: subtract a negative number and wrapping over could occur, this would be indicated by a positive return value. If wrapping under took place, a negative value would be returned

wrapping_int2 private member functions

1.

```
template<typename IntT> IntT calculate_wrap(IntT wrap);
```

Header <boost/date_time/year_month_day.hpp>

```
namespace boost {
    namespace date_time {
        template<typename YearType, typename MonthType, typename DayType>
            struct year_month_day_base;
    }
}
```

Struct template year_month_day_base

boost::date_time::year_month_day_base — Allow rapid creation of ymd triples of different types.

Synopsis

```
// In header: <boost/date_time/year_month_day.hpp>

template<typename YearType, typename MonthType, typename DayType>
struct year_month_day_base {
    // types
    typedef YearType  year_type;
    typedef MonthType month_type;
    typedef DayType   day_type;

    // construct/copy/destruct
    year_month_day_base(YearType, MonthType, DayType);

    // public data members
    YearType year;
    MonthType month;
    DayType day;
};
```

Description

year_month_day_base public construct/copy/destruct

1.

```
year_month_day_base(YearType year, MonthType month, DayType day);
```

A basic constructor.

Gregorian Reference

Header <boost/date_time/gregorian/conversion.hpp>

```
namespace boost {
    namespace gregorian {

        // Converts a date to a tm struct. Throws out_of_range exception if date is a special value.
        std::tm to_tm(const date & d);

        // Converts a tm structure into a date dropping the any time values.
        date date_from_tm(const std::tm & datetm);

    }
}
```

Header <boost/date_time/gregorian/formatters.hpp>

```
namespace boost {
    namespace gregorian {
        template<typename charT>
            std::basic_string< charT > to_simple_string_type(const date & d);

        // To YYYY-mm-DD string where mm 3 char month name. Example: 2002-Jan-01.
        std::string to_simple_string(const date & d);
        template<typename charT>
            std::basic_string< charT > to_simple_string_type(const date_period & d);

        // Convert date period to simple string. Example: [2002-Jan-01/2002-Jan-02].
        std::string to_simple_string(const date_period & d);
        template<typename charT>
            std::basic_string< charT > to_iso_string_type(const date_period & d);

        // Date period to iso standard format CCYYMMDD/CCYYMMDD. Example: 20021225/20021231.
        std::string to_iso_string(const date_period & d);
        template<typename charT>
            std::basic_string< charT > to_iso_extended_string_type(const date & d);

        // Convert to iso extended format string CCYY-MM-DD. Example 2002-12-31.
        std::string to_iso_extended_string(const date & d);
        template<typename charT>
            std::basic_string< charT > to_iso_string_type(const date & d);

        // Convert to iso standard string YYYYMMDD. Example: 20021231.
        std::string to_iso_string(const date & d);
        template<typename charT>
            std::basic_string< charT > to_sql_string_type(const date & d);
        std::string to_sql_string(const date & d);

        // Convert date period to simple string. Example: [2002-Jan-01/2002-Jan-02].
        std::wstring to_simple_wstring(const date_period & d);

        // To YYYY-mm-DD string where mm 3 char month name. Example: 2002-Jan-01.
        std::wstring to_simple_wstring(const date & d);

        // Date period to iso standard format CCYYMMDD/CCYYMMDD. Example: 20021225/20021231.
        std::wstring to_iso_wstring(const date_period & d);

        // Convert to iso extended format string CCYY-MM-DD. Example 2002-12-31.
```

```
std::wstring to_iso_extended_wstring(const date & d);

// Convert to iso standard string YYYYMMDD. Example: 20021231.
std::wstring to_iso_wstring(const date & d);
std::wstring to_sql_wstring(const date & d);
}
}
```

Header **<boost/date_time/gregorian/formatters_limited.hpp>**

Header **<boost/date_time/gregorian/greg_calendar.hpp>**

```
namespace boost {
    namespace gregorian {
        class gregorian_calendar;

        typedef date_time::int_adapter< uint32_t > fancy_date_rep; // An internal date representa-
        tion that includes infinities, not a date.
    }
}
```

Class gregorian_calendar

boost::gregorian::gregorian_calendar — Gregorian calendar for this implementation, hard work in the base.

Synopsis

```
// In header: <boost/date_time/gregorian/greg_calendar.hpp>

class gregorian_calendar {
public:
    // types
    typedef greg_weekday          day_of_week_type; // Type to hold a weekday (eg: Sunday, ↵
Monday,...)
    typedef greg_day_of_year_rep day_of_year_type; // Counter type from 1 to 366 for gregorian ↵
dates.
    typedef fancy_date_rep        date_rep_type;    // Internal date representation that handles ↵
infinity, not a date.
    typedef fancy_date_rep        date_traits_type; // Date rep implements the traits stuff as well.
};
```

Header **<boost/date_time/gregorian/greg_date.hpp>**

```
namespace boost {
    namespace gregorian {
        class date;
    }
}
```

Class date

boost::gregorian::date — A date type based on [gregorian_calendar](#).

Synopsis

```
// In header: <boost/date_time/gregorian/greg_date.hpp>

class date {
public:
    // types
    typedef gregorian_calendar::year_type      year_type;
    typedef gregorian_calendar::month_type     month_type;
    typedef gregorian_calendar::day_type       day_type;
    typedef gregorian_calendar::day_of_year_type day_of_year_type;
    typedef gregorian_calendar::ymd_type       ymd_type;
    typedef gregorian_calendar::date_rep_type  date_rep_type;
    typedef gregorian_calendar::date_int_type  date_int_type;
    typedef date_duration                      duration_type;

    // construct/copy/destruct
    date();
    date(year_type, month_type, day_type);
    explicit date(const ymd_type &);
    explicit date(const date_int_type &);
    explicit date(date_rep_type);
    explicit date(special_values);

    // public member functions
    date_int_type julian_day() const;
    day_of_year_type day_of_year() const;
    date_int_type modjulian_day() const;
    int week_number() const;
    date_int_type day_number() const;
    date end_of_month() const;
};
```

Description

This class is the primary interface for programming with gregorian dates. The is a lightweight type that can be freely passed by value. All comparison operators are supported.

date public construct/copy/destruct

1. `date();`

Default constructor constructs with not_a_date_time.

2. `date(year_type y, month_type m, day_type d);`

Main constructor with year, month, day.

3. `explicit date(const ymd_type & ymd);`

Constructor from a ymd_type structure.

4. `explicit date(const date_int_type & rhs);`

Needed copy constructor.

5. `explicit date(date_rep_type rhs);`

Needed copy constructor.

6. `explicit date(special_values sv);`

Constructor for infinities, not a date, max and min date.

date public member functions

1. `date_int_type julian_day() const;`

Return the Julian Day number for the date.

2. `day_of_year_type day_of_year() const;`

Return the day of year 1..365 or 1..366 (for leap year)

3. `date_int_type modjulian_day() const;`

Return the Modified Julian Day number for the date.

4. `int week_number() const;`

Return the iso 8601 week number 1..53.

5. `date_int_type day_number() const;`

Return the day number from the calendar.

6. `date end_of_month() const;`

Return the last day of the current month.

Header `<boost/date_time/gregorian/greg_day.hpp>`

```
namespace boost {
    namespace gregorian {
        struct bad_day_of_month;

        class greg_day;

        typedef CV::simple_exception_policy< unsigned short, 1, 31, bad_day_of_month > greg_day_policies; // Policy class that declares error handling and day of month ranges.
        typedef CV::constrained_value< greg_day_policies > greg_day_rep; // Generated representation for gregorian day of month.
    }
}
```

Struct bad_day_of_month

boost::gregorian::bad_day_of_month — Exception type for gregorian day of month (1..31)

Synopsis

```
// In header: <boost/date_time/gregorian/greg_day.hpp>

struct bad_day_of_month {
    // construct/copy/destroy
    bad_day_of_month();
    bad_day_of_month(const std::string &);
};
```

Description

bad_day_of_month public construct/copy/destroy

1. `bad_day_of_month();`
2. `bad_day_of_month(const std::string & s);`

Allow other classes to throw with unique string for bad day like Feb 29.

Class greg_day

`boost::gregorian::greg_day` — Represent a day of the month (range 1 - 31)

Synopsis

```
// In header: <boost/date_time/gregorian/greg_day.hpp>

class greg_day {
public:
    // construct/copy/destroy
    greg_day(unsigned short);

    // public member functions
    unsigned short as_number() const;
    operator unsigned short() const;
};
```

Description

This small class allows for simple conversion an integer value into a day of the month for a standard gregorian calendar. The type is automatically range checked so values outside of the range 1-31 will cause a `bad_day_of_month` exception

greg_day public construct/copy/destroy

1. `greg_day(unsigned short day_of_month);`

greg_day public member functions

1. `unsigned short as_number() const;`

2. `operator unsigned short() const;`

Header `<boost/date_time/gregorian/greg_day_of_year.hpp>`

```
namespace boost {
    namespace gregorian {
        struct bad_day_of_year;

        typedef CV::simple_exception_policy< unsigned short, 1, 366, bad_day_of_year > greg_day_of_year_policies; // A day of the year range 1..366
        typedef CV::constrained_value< greg_day_of_year_policies > greg_day_of_year_rep; // Define a range representation type for the day of the year 1..366.
    }
}
```

Struct `bad_day_of_year`

`boost::gregorian::bad_day_of_year` — Exception type for day of year (1..366)

Synopsis

```
// In header: <boost/date_time/gregorian/greg_day_of_year.hpp>

struct bad_day_of_year {
    // construct/copy/destroy
    bad_day_of_year();
};
```

Description

`bad_day_of_year` **public construct/copy/destroy**

1. `bad_day_of_year();`

Header `<boost/date_time/gregorian/greg_duration.hpp>`

```
namespace boost {
    namespace gregorian {
        class date_duration;

        typedef boost::date_time::duration_traits_adapted date_duration_rep; // An internal date representation that includes infinities, not a date.
        typedef date_duration days; // Shorthand for date_duration.
    }
}
```

Class `date_duration`

`boost::gregorian::date_duration` — Durations in days for gregorian system.

Synopsis

```
// In header: <boost/date_time/gregorian/greg_duration.hpp>
```

```
class date_duration {
public:
    // types
    typedef base_type::duration_rep duration_rep;

    // construct/copy/destruct
    explicit date_duration(duration_rep = 0);
    date_duration(date_time::special_values);
    date_duration(const date_duration &);
    date_duration(const base_type &);

    // public member functions
    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;
    date_duration & operator-=(const date_duration &);
    date_duration & operator+=(const date_duration &);
    date_duration operator-() const;
    date_duration & operator/=(int);

    // public static functions
    static date_duration unit();
};
```

Description

date_duration public construct/copy/destruct

1. `explicit date_duration(duration_rep day_count = 0);`

Construct from a day count.

2. `date_duration(date_time::special_values sv);`

construct from special_values

3. `date_duration(const date_duration & other);`

Copy constructor.

4. `date_duration(const base_type & other);`

Construct from another `date_duration`.

date_duration public member functions

1. `bool operator==(const date_duration & rhs) const;`

2. `bool operator!=(const date_duration & rhs) const;`

3. `bool operator<(const date_duration & rhs) const;`

4. `bool operator>(const date_duration & rhs) const;`

5. `bool operator<=(const date_duration & rhs) const;`

6. `bool operator>=(const date_duration & rhs) const;`

7. `date_duration & operator--(const date_duration & rhs);`

Subtract another duration -- result is signed.

8. `date_duration & operator+=(const date_duration & rhs);`

Add a duration -- result is signed.

9. `date_duration operator-() const;`

unary- Allows for `dd = -date_duration(2);` -> `dd == -2`

10. `date_duration & operator/=(int divisor);`

Division operations on a duration with an integer.

date_duration public static functions

1. `static date_duration unit();`

Returns the smallest duration -- used by to calculate 'end'.

Header <boost/date_time/gregorian/greg_duration_types.hpp>

```
namespace boost {
    namespace gregorian {
        struct greg_durations_config;

        class weeks_duration;

        typedef date_time::months_duration< greg_durations_config > months;
        typedef date_time::years_duration< greg_durations_config > years;
        typedef weeks_duration weeks;
    }
}
```

Struct greg_durations_config

boost::gregorian::greg_durations_config — config struct for additional duration types (ie months_duration<> & years_duration<>)

Synopsis

```
// In header: <boost/date_time/gregorian/greg_duration_types.hpp>

struct greg_durations_config {
    // types
    typedef date                                date_type;
    typedef date_time::int_adapter< int >        int_rep;
    typedef date_time::month_functor< date_type > month_adjustor_type;
};
```

Class weeks_duration

boost::gregorian::weeks_duration

Synopsis

```
// In header: <boost/date_time/gregorian/greg_duration_types.hpp>

class weeks_duration : public boost::gregorian::date_duration {
public:
    // construct/copy/destroy
    weeks_duration(duration_rep);
    weeks_duration(date_time::special_values);
};
```

Description

weeks_duration public construct/copy/destroy

1. `weeks_duration(duration_rep w);`
2. `weeks_duration(date_time::special_values sv);`

Header <boost/date_time/gregorian/greg_facet.hpp>

```

namespace boost {
    namespace gregorian {
        struct greg_facet_config;

        typedef boost::date_time::date_names_put< greg_facet_config > greg_base_facet; // Create the base facet type for gregorian::date.
        template<typename charT, typename traits>
            std::basic_ostream< charT, traits > &
            operator<<(std::basic_ostream< charT, traits > &, const date &);
        template<typename charT, typename traits>
            std::basic_ostream< charT, traits > &
            operator<<(std::basic_ostream< charT, traits > &, const greg_month &);
        template<typename charT, typename traits>
            std::basic_ostream< charT, traits > &
            operator<<(std::basic_ostream< charT, traits > &, const greg_weekday &);
        template<typename charT, typename traits>
            std::basic_ostream< charT, traits > &
            operator<<(std::basic_ostream< charT, traits > &, const date_period &);
        template<typename charT, typename traits>
            std::basic_ostream< charT, traits > &
            operator<<(std::basic_ostream< charT, traits > & os,
                const date_duration & dd);

        // operator<< for gregorian::partial_date. Output: "Jan 1"
        template<typename charT, typename traits>
            std::basic_ostream< charT, traits > &
            operator<<(std::basic_ostream< charT, traits > & os,
                const partial_date & pd);

        // operator<< for gregorian::nth_kday_of_month. Output: "first Mon of Jun"
        template<typename charT, typename traits>
            std::basic_ostream< charT, traits > &
            operator<<(std::basic_ostream< charT, traits > & os,
                const nth_kday_of_month & nkd);

        // operator<< for gregorian::first_kday_of_month. Output: "first Mon of Jun"
        template<typename charT, typename traits>
            std::basic_ostream< charT, traits > &
            operator<<(std::basic_ostream< charT, traits > & os,
                const first_kday_of_month & fkd);

        // operator<< for gregorian::last_kday_of_month. Output: "last Mon of Jun"
        template<typename charT, typename traits>
            std::basic_ostream< charT, traits > &
            operator<<(std::basic_ostream< charT, traits > & os,
                const last_kday_of_month & lkd);

        // operator<< for gregorian::first_kday_after. Output: "first Mon after"
        template<typename charT, typename traits>
            std::basic_ostream< charT, traits > &
            operator<<(std::basic_ostream< charT, traits > & os,
                const first_kday_after & fka);

        // operator<< for gregorian::first_kday_before. Output: "first Mon before"
        template<typename charT, typename traits>
            std::basic_ostream< charT, traits > &
            operator<<(std::basic_ostream< charT, traits > & os,
                const first_kday_before & fkb);

        // operator>> for gregorian::date
        template<typename charT>

```

```

std::basic_istream< charT > &
operator>>(std::basic_istream< charT > & is, date & d);

// operator>> for gregorian::date_duration
template<typename charT>
std::basic_istream< charT > &
operator>>(std::basic_istream< charT > & is, date_duration & dd);

// operator>> for gregorian::date_period
template<typename charT>
std::basic_istream< charT > &
operator>>(std::basic_istream< charT > & is, date_period & dp);

// generates a locale with the set of gregorian name-strings of type char*
BOOST_DATE_TIME_DECL std::locale
generate_locale(std::locale & loc, char type);

// Returns a pointer to a facet with a default set of names (English)
BOOST_DATE_TIME_DECL boost::date_time::all_date_names_put< greg_facet_config, char > *
create_facet_def(char type);

// generates a locale with the set of gregorian name-strings of type wchar_t*
BOOST_DATE_TIME_DECL std::locale
generate_locale(std::locale & loc, wchar_t type);

// Returns a pointer to a facet with a default set of names (English)
BOOST_DATE_TIME_DECL boost::date_time::all_date_names_put< greg_facet_config, wchar_t > *
create_facet_def(wchar_t type);

// operator>> for gregorian::greg_month - throws exception if invalid month given
template<typename charT>
std::basic_istream< charT > &
operator>>(std::basic_istream< charT > & is, greg_month & m);

// operator>> for gregorian::greg_weekday - throws exception if invalid weekday given
template<typename charT>
std::basic_istream< charT > &
operator>>(std::basic_istream< charT > & is, greg_weekday & wd);
}
}

```

Struct greg_facet_config

boost::gregorian::greg_facet_config — Configuration of the output facet template.

Synopsis

```

// In header: <boost/date_time/gregorian/greg_facet.hpp>

struct greg_facet_config {
    // types
    typedef boost::gregorian::greg_month      month_type;
    typedef boost::date_time::special_values  special_value_enum;
    typedef boost::gregorian::months_of_year  month_enum;
    typedef boost::date_time::weekdays       weekday_enum;
};

```

Function template operator<<

boost::gregorian::operator<< — ostream operator for [gregorian::date](#)

Synopsis

```
// In header: <boost/date_time/gregorian/greg_facet.hpp>

template<typename charT, typename traits>
    std::basic_ostream< charT, traits > &
    operator<<(std::basic_ostream< charT, traits > & os, const date & d);
```

Description

Uses the date facet to determine various output parameters including:

- string values for the month (eg: Jan, Feb, Mar) (default: English)
- string values for special values (eg: not-a-date-time) (default: English)
- selection of long, short strings, or numerical month representation (default: short string)
- month day year order (default yyyy-mmm-dd)

Function template operator<<

boost::gregorian::operator<< — operator<< for [gregorian::greg_month](#) typically streaming: Jan, Feb, Mar...

Synopsis

```
// In header: <boost/date_time/gregorian/greg_facet.hpp>

template<typename charT, typename traits>
    std::basic_ostream< charT, traits > &
    operator<<(std::basic_ostream< charT, traits > & os, const greg_month & m);
```

Description

Uses the date facet to determine output string as well as selection of long or short strings. Default if no facet is installed is to output a 2 wide numeric value for the month eg: 01 == Jan, 02 == Feb, ... 12 == Dec.

Function template operator<<

boost::gregorian::operator<< — operator<< for [gregorian::greg_weekday](#) typically streaming: Sun, Mon, Tue, ...

Synopsis

```
// In header: <boost/date_time/gregorian/greg_facet.hpp>

template<typename charT, typename traits>
    std::basic_ostream< charT, traits > &
    operator<<(std::basic_ostream< charT, traits > & os,
               const greg_weekday & wd);
```

Description

Uses the date facet to determine output string as well as selection of long or short string. Default if no facet is installed is to output a 3 char english string for the day of the week.

Function template operator<<

boost::gregorian::operator<< — operator<< for gregorian::date_period typical output: [2002-Jan-01/2002-Jan-31]

Synopsis

```
// In header: <boost/date_time/gregorian/greg_facet.hpp>

template<typename charT, typename traits>
std::basic_ostream< charT, traits > &
operator<<(std::basic_ostream< charT, traits > & os, const date_period & dp);
```

Description

Uses the date facet to determine output string as well as selection of long or short string fr dates. Default if no facet is installed is to output a 3 char english string for the day of the week.

Header <boost/date_time/gregorian/greg_month.hpp>

```
namespace boost {
    namespace gregorian {
        struct bad_month;

        class greg_month;

        typedef date_time::months_of_year months_of_year;
        typedef CV::simple_exception_policy< unsigned short, 1, 12, bad_month > greg_month_policies; ↵
        // Build a policy class for the greg_month_rep.
        typedef CV::constrained_value< greg_month_policies > greg_month_rep; // A constrained range ↵
        that implements the gregorian_month rules.
    }
}
```

Struct bad_month

boost::gregorian::bad_month — Exception thrown if a [greg_month](#) is constructed with a value out of range.

Synopsis

```
// In header: <boost/date_time/gregorian/greg_month.hpp>

struct bad_month {
    // construct/copy/destruct
    bad_month();
};
```


Description

bad_month public construct/copy/destruct

1.

```
bad_month();
```

Class greg_month

boost::gregorian::greg_month — Wrapper class to represent months in gregorian based calendar.

Synopsis

```
// In header: <boost/date_time/gregorian/greg_month.hpp>

class greg_month {
public:
    // types
    typedef date_time::months_of_year      month_enum;
    typedef std::map< std::string, unsigned short > month_map_type;
    typedef boost::shared_ptr< month_map_type > month_map_ptr_type;

    // construct/copy/destruct
    greg_month(month_enum);
    greg_month(unsigned short);

    // public member functions
    operator unsigned short() const;
    unsigned short as_number() const;
    month_enum as_enum() const;
    const char * as_short_string() const;
    const char * as_long_string() const;
    const wchar_t * as_short_wstring() const;
    const wchar_t * as_long_wstring() const;
    const char * as_short_string(char) const;
    const char * as_long_string(char) const;
    const wchar_t * as_short_string(wchar_t) const;
    const wchar_t * as_long_string(wchar_t) const;

    // public static functions
    static month_map_ptr_type get_month_map_ptr();
};
```

Description

greg_month public construct/copy/destruct

1.

```
greg_month(month_enum theMonth);
```

Construct a month from the months_of_year enumeration.

2.

```
greg_month(unsigned short theMonth);
```

Construct from a short value.

greg_month public member functions

1. `operator unsigned short() const;`

Convert the value back to a short.

2. `unsigned short as_number() const;`

Returns month as number from 1 to 12.

3. `month_enum as_enum() const;`

4. `const char * as_short_string() const;`

5. `const char * as_long_string() const;`

6. `const wchar_t * as_short_wstring() const;`

7. `const wchar_t * as_long_wstring() const;`

8. `const char * as_short_string(char) const;`

9. `const char * as_long_string(char) const;`

10. `const wchar_t * as_short_string(wchar_t) const;`

11. `const wchar_t * as_long_string(wchar_t) const;`

greg_month public static functions

1. `static month_map_ptr_type get_month_map_ptr();`

Shared pointer to a map of Month strings (Names & Abbrev) & numbers.

Header <boost/date_time/gregorian/greg_serialize.hpp>

```
BOOST_SERIALIZATION_SPLIT_FREE (::boost::gregorian::date_duration);
```

Function BOOST_SERIALIZATION_SPLIT_FREE

BOOST_SERIALIZATION_SPLIT_FREE

Synopsis

```
// In header: <boost/date_time/gregorian/greg_serialize.hpp>

BOOST_SERIALIZATION_SPLIT_FREE( ::boost::gregorian::date_duration );
```

Description

Method that does serialization for gregorian::date -- splits to load/save

Function to save gregorian::date objects using serialization lib

Dates are serialized into a string for transport and storage. While it would be more efficient to store the internal integer used to manipulate the dates, it is an unstable solution.

Function to load gregorian::date objects using serialization lib

Dates are serialized into a string for transport and storage. While it would be more efficient to store the internal integer used to manipulate the dates, it is an unstable solution.

override needed b/c no default constructor

Function to save gregorian::date_duration objects using serialization lib

Function to load gregorian::date_duration objects using serialization lib

override needed b/c no default constructor

helper unction to save date_duration objects using serialization lib

helper function to load date_duration objects using serialization lib

override needed b/c no default constructor

Function to save gregorian::date_period objects using serialization lib

date_period objects are broken down into 2 parts for serialization: the begining date object and the end date object

Function to load gregorian::date_period objects using serialization lib

date_period objects are broken down into 2 parts for serialization: the begining date object and the end date object

override needed b/c no default constructor

Function to save gregorian::greg_month objects using serialization lib

Function to load gregorian::greg_month objects using serialization lib

override needed b/c no default constructor

Function to save gregorian::greg_day objects using serialization lib

Function to load gregorian::greg_day objects using serialization lib

override needed b/c no default constructor

Function to save `gregorian::greg_weekday` objects using serialization lib

Function to load `gregorian::greg_weekday` objects using serialization lib

override needed b/c no default constructor

Function to save `gregorian::partial_date` objects using serialization lib

`partial_date` objects are broken down into 2 parts for serialization: the day (typically `greg_day`) and month (typically `greg_month`) objects

Function to load `gregorian::partial_date` objects using serialization lib

`partial_date` objects are broken down into 2 parts for serialization: the day (`greg_day`) and month (`greg_month`) objects

override needed b/c no default constructor

Function to save `nth_day_of_the_week_in_month` objects using serialization lib

`nth_day_of_the_week_in_month` objects are broken down into 3 parts for serialization: the week number, the day of the week, and the month

Function to load `nth_day_of_the_week_in_month` objects using serialization lib

`nth_day_of_the_week_in_month` objects are broken down into 3 parts for serialization: the week number, the day of the week, and the month

override needed b/c no default constructor

Function to save `first_day_of_the_week_in_month` objects using serialization lib

`first_day_of_the_week_in_month` objects are broken down into 2 parts for serialization: the day of the week, and the month

Function to load `first_day_of_the_week_in_month` objects using serialization lib

`first_day_of_the_week_in_month` objects are broken down into 2 parts for serialization: the day of the week, and the month

override needed b/c no default constructor

Function to save `last_day_of_the_week_in_month` objects using serialization lib

`last_day_of_the_week_in_month` objects are broken down into 2 parts for serialization: the day of the week, and the month

Function to load `last_day_of_the_week_in_month` objects using serialization lib

`last_day_of_the_week_in_month` objects are broken down into 2 parts for serialization: the day of the week, and the month

override needed b/c no default constructor

Function to save `first_day_of_the_week_before` objects using serialization lib

Function to load `first_day_of_the_week_before` objects using serialization lib

override needed b/c no default constructor

Function to save `first_day_of_the_week_after` objects using serialization lib

Function to load `first_day_of_the_week_after` objects using serialization lib

override needed b/c no default constructor

Header `<boost/date_time/gregorian/greg_weekday.hpp>`

```
namespace boost {
    namespace gregorian {
        struct bad_weekday;

        class greg_weekday;

        typedef CV::simple_exception_policy< unsigned short, 0, 6, bad_weekday > greg_weekday_policies;
        typedef CV::constrained_value< greg_weekday_policies > greg_weekday_rep;
    }
}
```

Struct `bad_weekday`

`boost::gregorian::bad_weekday` — Exception that flags that a weekday number is incorrect.

Synopsis

```
// In header: <boost/date_time/gregorian/greg_weekday.hpp>

struct bad_weekday {
    // construct/copy/destruct
    bad_weekday();
};
```

Description

`bad_weekday` **public construct/copy/destruct**

1. `bad_weekday();`

Class `greg_weekday`

`boost::gregorian::greg_weekday` — Represent a day within a week (range 0==Sun to 6==Sat)

Synopsis

```
// In header: <boost/date_time/gregorian/greg_weekday.hpp>

class greg_weekday {
public:
    // types
    typedef boost::date_time::weekdays weekday_enum;

    // construct/copy/destruct
    greg_weekday(unsigned short);

    // public member functions
    unsigned short as_number() const;
    const char * as_short_string() const;
    const char * as_long_string() const;
    const wchar_t * as_short_wstring() const;
    const wchar_t * as_long_wstring() const;
    weekday_enum as_enum() const;
};
```

Description

greg_weekday public construct/copy/destruct

1. `greg_weekday(unsigned short day_of_week_num);`

greg_weekday public member functions

1. `unsigned short as_number() const;`
2. `const char * as_short_string() const;`
3. `const char * as_long_string() const;`
4. `const wchar_t * as_short_wstring() const;`
5. `const wchar_t * as_long_wstring() const;`
6. `weekday_enum as_enum() const;`

Header `<boost/date_time/gregorian/greg_year.hpp>`

```
namespace boost {
    namespace gregorian {
        struct bad_year;

        class greg_year;

        typedef CV::simple_exception_policy< unsigned short, 1400, 10000, bad_year > greg_year_policies; // Policy class that declares error handling gregorian year type.
        typedef CV::constrained_value< greg_year_policies > greg_year_rep; // Generated representation for gregorian year.
    }
}
```

Struct `bad_year`

`boost::gregorian::bad_year` — Exception type for gregorian year.

Synopsis

```
// In header: <boost/date_time/gregorian/greg_year.hpp>

struct bad_year {
    // construct/copy/destroy
    bad_year();
};
```

Description

`bad_year` **public construct/copy/destroy**

```
1. bad_year();
```

Class `greg_year`

`boost::gregorian::greg_year` — Represent a day of the month (range 1900 - 10000)

Synopsis

```
// In header: <boost/date_time/gregorian/greg_year.hpp>

class greg_year {
public:
    // construct/copy/destroy
    greg_year(unsigned short);

    // public member functions
    operator unsigned short() const;
};
```

Description

This small class allows for simple conversion an integer value into a year for the gregorian calendar. This currently only allows a range of 1900 to 10000. Both ends of the range are a bit arbitrary at the moment, but they are the limits of current testing of the library. As such they may be increased in the future.

greg_year public construct/copy/destruct

```
1. greg_year(unsigned short year);
```

greg_year public member functions

```
1. operator unsigned short() const;
```

Header <[boost/date_time/gregorian/greg_ymd.hpp](#)>

```
namespace boost {  
    namespace gregorian {  
        typedef date_time::year_month_day_base< greg_year, greg_month, greg_day > greg_year_month_day;  
    }  
}
```

Header <[boost/date_time/gregorian/gregorian.hpp](#)>

Single file header that provides overall include for all elements of the gregorian date-time system. This includes the various types defined, but also other functions for formatting and parsing.

Header <boost/date_time/gregorian/gregorian_io.hpp>

```

namespace boost {
    namespace gregorian {
        typedef boost::date_time::period_formatter< wchar_t > wperiod_formatter;
        typedef boost::date_time::period_formatter< char > period_formatter;
        typedef boost::date_time::date_facet< date, wchar_t > wdate_facet;
        typedef boost::date_time::date_facet< date, char > date_facet;
        typedef boost::date_time::period_parser< date, char > period_parser;
        typedef boost::date_time::period_parser< date, wchar_t > wperiod_parser;
        typedef boost::date_time::special_values_formatter< char > special_values_formatter;
        typedef boost::date_time::special_values_formatter< wchar_t > wspecial_values_formatter;
        typedef boost::date_time::special_values_parser< date, char > special_values_parser;
        typedef boost::date_time::special_values_parser< date, wchar_t > wspecial_values_parser;
        typedef boost::date_time::date_input_facet< date, char > date_input_facet;
        typedef boost::date_time::date_input_facet< date, wchar_t > wdate_input_facet;
        template<typename CharT, typename TraitsT>
            std::basic_ostream< CharT, TraitsT > &
            operator<<(std::basic_ostream< CharT, TraitsT > & os,
                const boost::gregorian::date & d);

        // input operator for date
        template<typename CharT, typename Traits>
            std::basic_istream< CharT, Traits > &
            operator>>(std::basic_istream< CharT, Traits > & is, date & d);
        template<typename CharT, typename TraitsT>
            std::basic_ostream< CharT, TraitsT > &
            operator<<(std::basic_ostream< CharT, TraitsT > & os,
                const boost::gregorian::date_duration & dd);

        // input operator for date_duration
        template<typename CharT, typename Traits>
            std::basic_istream< CharT, Traits > &
            operator>>(std::basic_istream< CharT, Traits > & is, date_duration & dd);
        template<typename CharT, typename TraitsT>
            std::basic_ostream< CharT, TraitsT > &
            operator<<(std::basic_ostream< CharT, TraitsT > & os,
                const boost::gregorian::date_period & dp);

        // input operator for date_period
        template<typename CharT, typename Traits>
            std::basic_istream< CharT, Traits > &
            operator>>(std::basic_istream< CharT, Traits > & is, date_period & dp);
        template<typename CharT, typename TraitsT>
            std::basic_ostream< CharT, TraitsT > &
            operator<<(std::basic_ostream< CharT, TraitsT > & os,
                const boost::gregorian::greg_month & gm);

        // input operator for greg_month
        template<typename CharT, typename Traits>
            std::basic_istream< CharT, Traits > &
            operator>>(std::basic_istream< CharT, Traits > & is, greg_month & m);
        template<typename CharT, typename TraitsT>
            std::basic_ostream< CharT, TraitsT > &
            operator<<(std::basic_ostream< CharT, TraitsT > & os,
                const boost::gregorian::greg_weekday & gw);

        // input operator for greg_weekday
        template<typename CharT, typename Traits>
            std::basic_istream< CharT, Traits > &
            operator>>(std::basic_istream< CharT, Traits > & is, greg_weekday & wd);

        // input operator for greg_day
    }
}

```

```

template<typename CharT, typename Traits>
    std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > & is, greg_day & gd);

// input operator for greg_year
template<typename CharT, typename Traits>
    std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > & is, greg_year & gy);
template<typename CharT, typename TraitsT>
    std::basic_ostream< CharT, TraitsT > &
    operator<<(std::basic_ostream< CharT, TraitsT > & os,
        const boost::gregorian::partial_date & pd);

// input operator for partial_date
template<typename CharT, typename Traits>
    std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > & is, partial_date & pd);
template<typename CharT, typename TraitsT>
    std::basic_ostream< CharT, TraitsT > &
    operator<<(std::basic_ostream< CharT, TraitsT > & os,
        const boost::gregorian::nth_day_of_the_week_in_month & nk);

// input operator for nth_day_of_the_week_in_month
template<typename CharT, typename Traits>
    std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > & is,
        nth_day_of_the_week_in_month & nday);
template<typename CharT, typename TraitsT>
    std::basic_ostream< CharT, TraitsT > &
    operator<<(std::basic_ostream< CharT, TraitsT > & os,
        const boost::gregorian::first_day_of_the_week_in_month & fkd);

// input operator for first_day_of_the_week_in_month
template<typename CharT, typename Traits>
    std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > & is,
        first_day_of_the_week_in_month & fkd);
template<typename CharT, typename TraitsT>
    std::basic_ostream< CharT, TraitsT > &
    operator<<(std::basic_ostream< CharT, TraitsT > & os,
        const boost::gregorian::last_day_of_the_week_in_month & lkd);

// input operator for last_day_of_the_week_in_month
template<typename CharT, typename Traits>
    std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > & is,
        last_day_of_the_week_in_month & lkd);
template<typename CharT, typename TraitsT>
    std::basic_ostream< CharT, TraitsT > &
    operator<<(std::basic_ostream< CharT, TraitsT > & os,
        const boost::gregorian::first_day_of_the_week_after & fda);

// input operator for first_day_of_the_week_after
template<typename CharT, typename Traits>
    std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > & is,
        first_day_of_the_week_after & fka);
template<typename CharT, typename TraitsT>
    std::basic_ostream< CharT, TraitsT > &
    operator<<(std::basic_ostream< CharT, TraitsT > & os,
        const boost::gregorian::first_day_of_the_week_before & fdb);

// input operator for first_day_of_the_week_before

```

```

template<typename CharT, typename Traits>
    std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > & is,
               first_day_of_the_week_before & fkb);
}

```

Header <boost/date_time/gregorian/gregorian_types.hpp>

Single file header that defines most of the types for the gregorian date-time system.

```

namespace boost {
    namespace gregorian {
        typedef date_time::period< date, date_duration > date_period; // Date periods for the gregorian system.
        typedef date_time::year_based_generator< date > year_based_generator;
        typedef date_time::partial_date< date > partial_date; // A date generation object type.
        typedef date_time::nth_kday_of_month< date > nth_kday_of_month;
        typedef nth_kday_of_month nth_day_of_the_week_in_month;
        typedef date_time::first_kday_of_month< date > first_kday_of_month;
        typedef first_kday_of_month first_day_of_the_week_in_month;
        typedef date_time::last_kday_of_month< date > last_kday_of_month;
        typedef last_kday_of_month last_day_of_the_week_in_month;
        typedef date_time::first_kday_after< date > first_kday_after;
        typedef first_kday_after first_day_of_the_week_after;
        typedef date_time::first_kday_before< date > first_kday_before;
        typedef first_kday_before first_day_of_the_week_before;
        typedef date_time::day_clock< date > day_clock; // A clock to get the current day from the local computer.
        typedef date_time::date_itr_base< date > date_iterator; // Base date_iterator type for gregorian types.
        typedef date_time::date_itr< date_time::day_functor< date >, date > day_iterator; // A day level iterator.
        typedef date_time::date_itr< date_time::week_functor< date >, date > week_iterator; // A week level iterator.
        typedef date_time::date_itr< date_time::month_functor< date >, date > month_iterator; // A month level iterator.
        typedef date_time::date_itr< date_time::year_functor< date >, date > year_iterator; // A year level iterator.
    }
}

```

Type definition year_based_generator

year_based_generator — A unifying date_generator base type.

Synopsis

```

// In header: <boost/date_time/gregorian/gregorian_types.hpp>

typedef date_time::year_based_generator< date > year_based_generator;

```

Description

A unifying date_generator base type for: partial_date, nth_day_of_the_week_in_month, first_day_of_the_week_in_month, and last_day_of_the_week_in_month

Header `<boost/date_time/gregorian/parsers.hpp>`

```
namespace boost {
    namespace gregorian {
        BOOST_DATE_TIME_DECL special_values
        special_value_from_string(const std::string &);

        // Deprecated: Use from_simple_string.
        date from_string(std::string s);

        // From delimited date string where with order year-month-day eg: 2002-1-25 or 2003-Jan-
        25 (full month name is also accepted)
        date from_simple_string(std::string s);

        // From delimited date string where with order year-month-day eg: 1-25-2003 or Jan-25-
        2003 (full month name is also accepted)
        date from_us_string(std::string s);

        // From delimited date string where with order day-month-year eg: 25-1-2002 or 25-Jan-
        2003 (full month name is also accepted)
        date from_uk_string(std::string s);

        // From iso type date string where with order year-month-day eg: 20020125.
        date from_undelimited_string(std::string s);

        // From iso type date string where with order year-month-day eg: 20020125.
        date date_from_iso_string(const std::string & s);

        // Stream should hold a date in the form of: 2002-1-25. Month number, abbrev, or name are
        accepted.
        template<typename iterator_type>
        date from_stream(iterator_type beg, iterator_type end);

        // Function to parse a date_period from a string (eg: [2003-Oct-31/2003-Dec-25])
        date_period date_period_from_string(const std::string & s);

        // Function to parse a date_period from a wstring (eg: [2003-Oct-31/2003-Dec-25])
        date_period date_period_from_wstring(const std::wstring & s);
    }
}
```

Function `special_value_from_string`

`boost::gregorian::special_value_from_string` — Return `special_value` from string argument.

Synopsis

```
// In header: <boost/date_time/gregorian/parsers.hpp>

BOOST_DATE_TIME_DECL special_values
special_value_from_string(const std::string & s);
```

Description

Return `special_value` from string argument. If argument is not one of the special value names (defined in `src/gregorian/names.hpp`), return `'not_special'`

Posix Time Reference

Header <boost/date_time/posix_time/conversion.hpp>

```
namespace boost {
    namespace posix_time {

        // Function that converts a time_t into a ptime.
        ptime from_time_t(std::time_t t);

        // Convert a time to a tm structure truncating any fractional seconds.
        std::tm to_tm(const boost::posix_time::ptime & t);

        // Convert a time_duration to a tm structure truncating any fractional seconds and zeroing ↓
        fields for date components.
        std::tm to_tm(const boost::posix_time::time_duration & td);

        // Convert a tm struct to a ptime ignoring is_dst flag.
        ptime ptime_from_tm(const std::tm & timetm);
        template<typename TimeT, typename FileTimeT>
        TimeT from_fptime(const FileTimeT &);
    }
}
```

Function template from_fptime

boost::posix_time::from_fptime — Function to create a time object from an initialized FILETIME struct.

Synopsis

```
// In header: <boost/date_time/posix_time/conversion.hpp>

template<typename TimeT, typename FileTimeT>
TimeT from_fptime(const FileTimeT & ft);
```

Description

Function to create a time object from an initialized FILETIME struct. A FILETIME struct holds 100-nanosecond units (0.0000001). When built with microsecond resolution the FILETIME's sub second value will be truncated. Nanosecond resolution has no truncation.



Note

FILETIME is part of the Win32 API, so it is not portable to non-windows platforms.

The function is templated on the FILETIME type, so that it can be used with both native FILETIME and the ad-hoc boost::date_time::winapi::file_time type.

Header <boost/date_time/posix_time/date_duration_operators.hpp>

Operators for ptime and optional gregorian types. Operators use snap-to-end-of-month behavior. Further details on this behavior can be found in reference for date_time/date_duration_types.hpp and documentation for month and year iterators.

```
namespace boost {  
    namespace posix_time {  
        ptime operator+(const ptime &, const boost::gregorian::months &);  
        ptime operator+=(ptime &, const boost::gregorian::months &);  
        ptime operator-(const ptime &, const boost::gregorian::months &);  
        ptime operator-=(ptime &, const boost::gregorian::months &);  
        ptime operator+(const ptime &, const boost::gregorian::years &);  
        ptime operator+=(ptime &, const boost::gregorian::years &);  
        ptime operator-(const ptime &, const boost::gregorian::years &);  
        ptime operator-=(ptime &, const boost::gregorian::years &);  
    }  
}
```

Function operator+

boost::posix_time::operator+

Synopsis

```
// In header: <boost/date_time/posix_time/date_duration_operators.hpp>  
  
ptime operator+(const ptime & t, const boost::gregorian::months & m);
```

Description

Adds a months object and a ptime. Result will be same day-of-month as ptime unless original day was the last day of month. see `date_time::months_duration` for more details

Function operator+=

boost::posix_time::operator+=

Synopsis

```
// In header: <boost/date_time/posix_time/date_duration_operators.hpp>  
  
ptime operator+=(ptime & t, const boost::gregorian::months & m);
```

Description

Adds a months object to a ptime. Result will be same day-of-month as ptime unless original day was the last day of month. see `date_time::months_duration` for more details

Function operator-

boost::posix_time::operator-

Synopsis

```
// In header: <boost/date_time/posix_time/date_duration_operators.hpp>

ptime operator-(const ptime & t, const boost::gregorian::months & m);
```

Description

Subtracts a months object and a ptime. Result will be same day-of-month as ptime unless original day was the last day of month. see `date_time::months_duration` for more details

Function operator-=

`boost::posix_time::operator-=`

Synopsis

```
// In header: <boost/date_time/posix_time/date_duration_operators.hpp>

ptime operator-=(ptime & t, const boost::gregorian::months & m);
```

Description

Subtracts a months object from a ptime. Result will be same day-of-month as ptime unless original day was the last day of month. see `date_time::months_duration` for more details

Function operator+

`boost::posix_time::operator+`

Synopsis

```
// In header: <boost/date_time/posix_time/date_duration_operators.hpp>

ptime operator+(const ptime & t, const boost::gregorian::years & y);
```

Description

Adds a years object and a ptime. Result will be same month and day-of-month as ptime unless original day was the last day of month. see `date_time::years_duration` for more details

Function operator+=

`boost::posix_time::operator+=`

Synopsis

```
// In header: <boost/date_time/posix_time/date_duration_operators.hpp>

ptime operator+=(ptime & t, const boost::gregorian::years & y);
```

Description

Adds a years object to a ptime. Result will be same month and day-of-month as ptime unless original day was the last day of month. see `date_time::years_duration` for more details

Function operator-

`boost::posix_time::operator-`

Synopsis

```
// In header: <boost/date_time/posix_time/date_duration_operators.hpp>

ptime operator-(const ptime & t, const boost::gregorian::years & y);
```

Description

Subtracts a years object and a ptime. Result will be same month and day-of-month as ptime unless original day was the last day of month. see `date_time::years_duration` for more details

Function operator-=

`boost::posix_time::operator-=`

Synopsis

```
// In header: <boost/date_time/posix_time/date_duration_operators.hpp>

ptime operator-=(ptime & t, const boost::gregorian::years & y);
```

Description

Subtracts a years object from a ptime. Result will be same month and day-of-month as ptime unless original day was the last day of month. see `date_time::years_duration` for more details

Header **<boost/date_time/posix_time/posix_time.hpp>**

Global header file to get all of posix time types

Header `<boost/date_time/posix_time/posix_time_config.hpp>`

```
namespace boost {
    namespace posix_time {
        class time_duration;

        struct simple_time_rep;

        class posix_time_system_config;
        class millisec_posix_time_system_config;

        typedef date_time::time_resolution_traits< boost::date_time::time_resolution_traits_adapted64_impl, boost::date_time::nano, 1000000000, 9 > time_res_traits;
    }
}
```

Class `time_duration`

`boost::posix_time::time_duration` — Base time duration type.

Synopsis

```
// In header: <boost/date_time/posix_time/posix_time_config.hpp>

class time_duration {
public:
    // types
    typedef time_res_traits          rep_type;
    typedef time_res_traits::day_type day_type;
    typedef time_res_traits::hour_type hour_type;
    typedef time_res_traits::min_type min_type;
    typedef time_res_traits::sec_type sec_type;
    typedef time_res_traits::fractional_seconds_type fractional_seconds_type;
    typedef time_res_traits::tick_type tick_type;
    typedef time_res_traits::impl_type impl_type;

    // construct/copy/destruct
    time_duration(hour_type hour, min_type min, sec_type sec, fractional_seconds_type fs = 0);
    time_duration();
    time_duration(boost::date_time::special_values);
    explicit time_duration(impl_type);
};
```

Description

`time_duration` public construct/copy/destruct

1. `time_duration(hour_type hour, min_type min, sec_type sec, fractional_seconds_type fs = 0);`
2. `time_duration();`
3. `time_duration(boost::date_time::special_values sv);`

Construct from special_values.

4. `explicit time_duration(impl_type tick_count);`

Struct simple_time_rep

boost::posix_time::simple_time_rep — Simple implementation for the time rep.

Synopsis

```
// In header: <boost/date_time/posix_time/posix_time_config.hpp>

struct simple_time_rep {
    // types
    typedef gregorian::date date_type;
    typedef time_duration time_duration_type;

    // construct/copy/destruct
    simple_time_rep(date_type, time_duration_type);

    // public member functions
    bool is_special() const;
    bool is_pos_infinity() const;
    bool is_neg_infinity() const;
    bool is_not_a_date_time() const;

    // public data members
    date_type day;
    time_duration_type time_of_day;
};
```

Description

simple_time_rep public construct/copy/destruct

1. `simple_time_rep(date_type d, time_duration_type tod);`

simple_time_rep public member functions

1. `bool is_special() const;`

2. `bool is_pos_infinity() const;`

3. `bool is_neg_infinity() const;`

4. `bool is_not_a_date_time() const;`

Class posix_time_system_config

boost::posix_time::posix_time_system_config

Synopsis

```
// In header: <boost/date_time/posix_time/posix_time_config.hpp>

class posix_time_system_config {
public:
    // types
    typedef simple_time_rep          time_rep_type;
    typedef gregorian::date          date_type;
    typedef gregorian::date_duration date_duration_type;
    typedef time_duration            time_duration_type;
    typedef time_res_traits::tick_type int_type;
    typedef time_res_traits          resolution_traits;

    // public member functions
    BOOST_STATIC_CONSTANT(boost::int64_t, tick_per_second = 1000000000);
};
```

Description

posix_time_system_config public member functions

1. BOOST_STATIC_CONSTANT(boost::int64_t, tick_per_second = 1000000000);

Class millisec_posix_time_system_config

boost::posix_time::millisec_posix_time_system_config

Synopsis

```
// In header: <boost/date_time/posix_time/posix_time_config.hpp>

class millisec_posix_time_system_config {
public:
    // types
    typedef boost::int64_t          time_rep_type;
    typedef gregorian::date          date_type;
    typedef gregorian::date_duration date_duration_type;
    typedef time_duration            time_duration_type;
    typedef time_res_traits::tick_type int_type;
    typedef time_res_traits::impl_type impl_type;
    typedef time_res_traits          resolution_traits;

    // public member functions
    BOOST_STATIC_CONSTANT(boost::int64_t, tick_per_second = 1000000);
};
```

Description

`millisec_posix_time_system_config` public member functions

1. `BOOST_STATIC_CONSTANT(boost::int64_t, tick_per_second = 1000000);`

Header `<boost/date_time/posix_time/posix_time_duration.hpp>`

```
namespace boost {
    namespace posix_time {
        class hours;
        class minutes;
        class seconds;

        typedef date_time::subsecond_duration< time_duration, 1000 > millisec; // Allows expression of durations as milli seconds.
        typedef date_time::subsecond_duration< time_duration, 1000 > milliseconds;
        typedef date_time::subsecond_duration< time_duration, 1000000 > microsec; // Allows expression of durations as micro seconds.
        typedef date_time::subsecond_duration< time_duration, 1000000 > microseconds;
        typedef date_time::subsecond_duration< time_duration, 1000000000 > nanosec; // Allows expression of durations as nano seconds.
        typedef date_time::subsecond_duration< time_duration, 1000000000 > nanoseconds;
    }
}
```

Class hours

`boost::posix_time::hours` — Allows expression of durations as an hour count.

Synopsis

```
// In header: <boost/date_time/posix_time/posix_time_duration.hpp>

class hours : public boost::posix_time::time_duration {
public:
    // construct/copy/destruct
    explicit hours(long);
};
```

Description

`hours` public construct/copy/destruct

1. `explicit hours(long h);`

Class minutes

`boost::posix_time::minutes` — Allows expression of durations as a minute count.

Synopsis

```
// In header: <boost/date_time/posix_time/posix_time_duration.hpp>

class minutes : public boost::posix_time::time_duration {
public:
    // construct/copy/destroy
    explicit minutes(long);
};
```

Description

minutes public construct/copy/destroy

1. `explicit minutes(long m);`

Class seconds

`boost::posix_time::seconds` — Allows expression of durations as a seconds count.

Synopsis

```
// In header: <boost/date_time/posix_time/posix_time_duration.hpp>

class seconds : public boost::posix_time::time_duration {
public:
    // construct/copy/destroy
    explicit seconds(long);
};
```

Description

seconds public construct/copy/destroy

1. `explicit seconds(long s);`

Header <boost/date_time/posix_time/posix_time_io.hpp>

```

namespace boost {
    namespace posix_time {
        typedef boost::date_time::time_facet< ptime, wchar_t > wtime_facet;
        typedef boost::date_time::time_facet< ptime, char > time_facet;
        typedef boost::date_time::time_input_facet< ptime, wchar_t > wtime_input_facet;
        typedef boost::date_time::time_input_facet< ptime, char > time_input_facet;
        template<typename CharT, typename TraitsT>
            std::basic_ostream< CharT, TraitsT > &
            operator<<(std::basic_ostream< CharT, TraitsT > & os, const ptime & p);

        // input operator for ptime
        template<typename CharT, typename Traits>
            std::basic_istream< CharT, Traits > &
            operator>>(std::basic_istream< CharT, Traits > & is, ptime & pt);
        template<typename CharT, typename TraitsT>
            std::basic_ostream< CharT, TraitsT > &
            operator<<(std::basic_ostream< CharT, TraitsT > & os,
                const boost::posix_time::time_period & p);

        // input operator for time_period
        template<typename CharT, typename Traits>
            std::basic_istream< CharT, Traits > &
            operator>>(std::basic_istream< CharT, Traits > & is, time_period & tp);

        // ostream operator for posix_time::time_duration
        template<typename CharT, typename Traits>
            std::basic_ostream< CharT, Traits > &
            operator<<(std::basic_ostream< CharT, Traits > & os,
                const time_duration & td);

        // input operator for time_duration
        template<typename CharT, typename Traits>
            std::basic_istream< CharT, Traits > &
            operator>>(std::basic_istream< CharT, Traits > & is, time_duration & td);
    }
}

```

Type definition wtime_facet

wtime_facet — wptime_facet is deprecated and will be phased out. use wtime_facet instead

Synopsis

```

// In header: <boost/date_time/posix_time/posix_time_io.hpp>

typedef boost::date_time::time_facet< ptime, wchar_t > wtime_facet;

```

Description

ptime_facet is deprecated and will be phased out. use time_facet instead wptime_input_facet is deprecated and will be phased out. use wtime_input_facet instead ptime_input_facet is deprecated and will be phased out. use time_input_facet instead

Header `<boost/date_time/posix_time/posix_time_legacy_io.hpp>`

```

namespace boost {
    namespace posix_time {

        // ostream operator for posix_time::time_duration
        template<typename charT, typename traits>
        std::basic_ostream< charT, traits > &
        operator<<(std::basic_ostream< charT, traits > & os,
            const time_duration & td);

        // ostream operator for posix_time::ptime
        template<typename charT, typename traits>
        std::basic_ostream< charT, traits > &
        operator<<(std::basic_ostream< charT, traits > & os, const ptime & t);

        // ostream operator for posix_time::time_period
        template<typename charT, typename traits>
        std::basic_ostream< charT, traits > &
        operator<<(std::basic_ostream< charT, traits > & os,
            const time_period & tp);
        template<typename charT>
        std::basic_istream< charT > &
        operator>>(std::basic_istream< charT > & is, time_duration & td);
        template<typename charT>
        std::basic_istream< charT > &
        operator>>(std::basic_istream< charT > & is, ptime & pt);
        template<typename charT>
        std::basic_istream< charT > &
        operator>>(std::basic_istream< charT > & is, time_period & tp);
    }
}

```

Function template operator>>

boost::posix_time::operator>>

Synopsis

```

// In header: <boost/date_time/posix_time/posix_time_legacy_io.hpp>

template<typename charT>
std::basic_istream< charT > &
operator>>(std::basic_istream< charT > & is, time_period & tp);

```

Description

operator>> for time_period. time_period must be in "[date time_duration/date time_duration]" format.

Header <boost/date_time/posix_time/posix_time_system.hpp>

```

namespace boost {
    namespace posix_time {
        typedef date_time::split_timedate_system< posix_time_system_config, 1000000000 > posix_time_system;
        typedef date_time::counted_time_rep< millisec_posix_time_system_config > int64_time_rep;
    }
}

```

Header <boost/date_time/posix_time/posix_time_types.hpp>

```

namespace boost {
    namespace posix_time {
        typedef date_time::time_itr< ptime > time_iterator; // Iterator over a defined time duration.
        typedef date_time::second_clock< ptime > second_clock; // A time clock that has a resolution of one second.
        typedef date_time::microsec_clock< ptime > microsec_clock; // A time clock that has a resolution of one microsecond.
        typedef date_time::null_dst_rules< ptime::date_type, time_duration > no_dst; // Define a null dst rule for the posix_time system.
        typedef date_time::us_dst_rules< ptime::date_type, time_duration > us_dst; // Define US dst rule calculator for the posix_time system.
    }
}

```

Header <boost/date_time/posix_time/ptime.hpp>

```

namespace boost {
    namespace posix_time {
        class ptime;
    }
}

```

Class ptime

boost::posix_time::ptime — Time type with no timezone or other adjustments.

Synopsis

```
// In header: <boost/date_time/posix_time/ptime.hpp>

class ptime {
public:
    // types
    typedef posix_time_system          time_system_type;
    typedef time_system_type::time_rep_type    time_rep_type;
    typedef time_system_type::time_duration_type time_duration_type;
    typedef ptime                      time_type;

    // construct/copy/destruct
    ptime(gregorian::date, time_duration_type);
    explicit ptime(gregorian::date);
    ptime(const time_rep_type &);
    ptime(const special_values);
    ptime();
};
```

Description

ptime public construct/copy/destruct

1. `ptime(gregorian::date d, time_duration_type td);`

Construct with date and offset in day.

2. `explicit ptime(gregorian::date d);`

Construct a time at start of the given day (midnight)

3. `ptime(const time_rep_type & rhs);`

Copy from time_rep.

4. `ptime(const special_values sv);`

Construct from special value.

5. `ptime();`

Header **<boost/date_time/posix_time/time_formatters.hpp>**

```

namespace boost {
    namespace posix_time {
        template<typename charT>
            std::basic_string< charT > to_simple_string_type(time_duration td);

        // Time duration to string -hh:mm:ss.ffffff. Example: 10:09:03.0123456.
        std::string to_simple_string(time_duration td);
        template<typename charT>
            std::basic_string< charT > to_iso_string_type(time_duration td);

        // Time duration in iso format -hhmmss,ffffff Example: 10:09:03,0123456.
        std::string to_iso_string(time_duration td);

        // Time to simple format CCYY-mm-dd hh:mm:ss.ffffff.
        template<typename charT>
            std::basic_string< charT > to_simple_string_type(ptime t);

        // Time to simple format CCYY-mm-dd hh:mm:ss.ffffff.
        std::string to_simple_string(ptime t);
        template<typename charT>
            std::basic_string< charT > to_simple_string_type(time_period tp);

        // Convert to string of form [YYYY-mm-DD HH:MM::SS.ffffff/YYYY-mm-DD HH:MM::SS.ffffff].
        std::string to_simple_string(time_period tp);
        template<typename charT>
            std::basic_string< charT > to_iso_string_type(ptime t);

        // Convert iso short form YYYYMMDDTHHMMSS where T is the date-time separator.
        std::string to_iso_string(ptime t);
        template<typename charT>
            std::basic_string< charT > to_iso_extended_string_type(ptime t);

        // Convert to form YYYY-MM-DDTHH:MM:SS where T is the date-time separator.
        std::string to_iso_extended_string(ptime t);

        // Time duration to wstring -hh:mm:ss.ffffff. Example: 10:09:03.0123456.
        std::wstring to_simple_wstring(time_duration td);

        // Time duration in iso format -hhmmss,ffffff Example: 10:09:03,0123456.
        std::wstring to_iso_wstring(time_duration td);
        std::wstring to_simple_wstring(ptime t);

        // Convert to wstring of form [YYYY-mm-DD HH:MM::SS.ffffff/YYYY-mm-DD HH:MM::SS.ffffff].
        std::wstring to_simple_wstring(time_period tp);

        // Convert iso short form YYYYMMDDTHHMMSS where T is the date-time separator.
        std::wstring to_iso_wstring(ptime t);

        // Convert to form YYYY-MM-DDTHH:MM:SS where T is the date-time separator.
        std::wstring to_iso_extended_wstring(ptime t);
    }
}

```

Header <[boost/date_time/posix_time/time_formatters_limited.hpp](#)>

Header <[boost/date_time/posix_time/time_parsers.hpp](#)>

```
namespace boost {
    namespace posix_time {
        time_duration duration_from_string(const std::string &);
        ptime time_from_string(const std::string & s);
        ptime from_iso_string(const std::string & s);
    }
}
```

Function `duration_from_string`

`boost::posix_time::duration_from_string` — Creates a [time_duration](#) object from a delimited string.

Synopsis

```
// In header: <boost/date_time/posix_time/time_parsers.hpp>

time_duration duration_from_string(const std::string & s);
```

Description

Expected format for string is "[-]h[h][:mm][:ss][.fff]". A negative duration will be created if the first character in string is a '-', all other '-' will be treated as delimiters. Accepted delimiters are "-:.,".

Header <[boost/date_time/posix_time/time_period.hpp](#)>

```
namespace boost {
    namespace posix_time {
        typedef date_time::period< ptime, time_duration > time_period; // Time period type.
    }
}
```

Header <[boost/date_time/posix_time/time_serialize.hpp](#)>

```
BOOST_SERIALIZATION_SPLIT_FREE(boost::posix_time::ptime);
```

Function `BOOST_SERIALIZATION_SPLIT_FREE`

`BOOST_SERIALIZATION_SPLIT_FREE`

Synopsis

```
// In header: <boost/date_time/posix_time/time_serialize.hpp>

BOOST_SERIALIZATION_SPLIT_FREE(boost::posix_time::ptime);
```

Description

Function to save `posix_time::time_duration` objects using serialization lib

`time_duration` objects are broken down into 4 parts for serialization: types are `hour_type`, `min_type`, `sec_type`, and `fractional_seconds_type` as defined in the `time_duration` class

Function to load `posix_time::time_duration` objects using serialization lib

`time_duration` objects are broken down into 4 parts for serialization: types are `hour_type`, `min_type`, `sec_type`, and `fractional_seconds_type` as defined in the `time_duration` class

Function to save `posix_time::ptime` objects using serialization lib

`ptime` objects are broken down into 2 parts for serialization: a date object and a `time_duration` object

Function to load `posix_time::ptime` objects using serialization lib

`ptime` objects are broken down into 2 parts for serialization: a date object and a `time_duration` object

override needed b/c no default constructor

Function to save `posix_time::time_period` objects using serialization lib

`time_period` objects are broken down into 2 parts for serialization: a beginning `ptime` object and an ending `ptime` object

Function to load `posix_time::time_period` objects using serialization lib

`time_period` objects are broken down into 2 parts for serialization: a beginning `ptime` object and an ending `ptime` object

override needed b/c no default constructor

Local Time Reference

Header <[boost/date_time/local_time/conversion.hpp](#)>

```
namespace boost {
    namespace local_time {

        // Function that creates a tm struct from a local_date_time.
        std::tm to_tm(const local_date_time & lt);

    }
}
```

Header <[boost/date_time/local_time/custom_time_zone.hpp](#)>

```
namespace boost {
    namespace local_time {
        template<typename CharT> class custom_time_zone_base;

        typedef boost::date_time::dst_adjustment_offsets< boost::posix_time::time_duration > dst_adjustment_offsets;
        typedef boost::shared_ptr< dst_calc_rule > dst_calc_rule_ptr;
        typedef custom_time_zone_base< char > custom_time_zone;

    }
}
```

Class template custom_time_zone_base

boost::local_time::custom_time_zone_base — A real time zone.

Synopsis

```
// In header: <boost/date_time/local_time/custom_time_zone.hpp>

template<typename CharT>
class custom_time_zone_base {
public:
    // types
    typedef boost::posix_time::time_duration                time_duration_type;
    typedef date_time::time_zone_base< posix_time::ptime, CharT > base_type;
    typedef base_type::string_type                          string_type;
    typedef base_type::stringstream_type                   stringstream_type;
    typedef date_time::time_zone_names_base< CharT >        time_zone_names;
    typedef CharT                                           char_type;

    // construct/copy/destruct
    custom_time_zone_base(const time_zone_names &, const time_duration_type &,
                        const dst_adjustment_offsets &,
                        boost::shared_ptr< dst_calc_rule >);
    ~custom_time_zone_base();

    // public member functions
    string_type dst_zone_abbrev() const;
    string_type std_zone_abbrev() const;
    string_type dst_zone_name() const;
    string_type std_zone_name() const;
    bool has_dst() const;
    posix_time::ptime dst_local_start_time(gregorian::greg_year) const;
    posix_time::ptime dst_local_end_time(gregorian::greg_year) const;
    time_duration_type base_utc_offset() const;
    time_duration_type dst_offset() const;
    string_type to_posix_string() const;
};
```

Description

custom_time_zone_base public construct/copy/destruct

1.

```
custom_time_zone_base(const time_zone_names & zone_names,
                    const time_duration_type & utc_offset,
                    const dst_adjustment_offsets & dst_shift,
                    boost::shared_ptr< dst_calc_rule > calc_rule);
```
2.

```
~custom_time_zone_base();
```

custom_time_zone_base public member functions

1.

```
string_type dst_zone_abbrev() const;
```
2.

```
string_type std_zone_abbrev() const;
```

3. `string_type dst_zone_name() const;`

4. `string_type std_zone_name() const;`

5. `bool has_dst() const;`

True if zone uses daylight savings adjustments.

6. `posix_time::ptime dst_local_start_time(gregorian::greg_year y) const;`

Local time that DST starts -- NADT if has_dst is false.

7. `posix_time::ptime dst_local_end_time(gregorian::greg_year y) const;`

Local time that DST ends -- NADT if has_dst is false.

8. `time_duration_type base_utc_offset() const;`

Base offset from UTC for zone (eg: -07:30:00)

9. `time_duration_type dst_offset() const;`

Adjustment forward or back made while DST is in effect.

10. `string_type to_posix_string() const;`

Returns a POSIX time_zone string for this object.

Header <[boost/date_time/local_time/date_duration_operators.hpp](#)>

Operators for local_date_time and optional gregorian types. Operators use snap-to-end-of-month behavior. Further details on this behavior can be found in reference for date_time/date_duration_types.hpp and documentation for month and year iterators.

```
namespace boost {
    namespace local_time {
        local_date_time
        operator+(const local_date_time &, const boost::gregorian::months &);
        local_date_time
        operator+=(local_date_time &, const boost::gregorian::months &);
        local_date_time
        operator-(const local_date_time &, const boost::gregorian::months &);
        local_date_time
        operator-=(local_date_time &, const boost::gregorian::months &);
        local_date_time
        operator+(const local_date_time &, const boost::gregorian::years &);
        local_date_time
        operator+=(local_date_time &, const boost::gregorian::years &);
        local_date_time
        operator-(const local_date_time &, const boost::gregorian::years &);
        local_date_time
        operator-=(local_date_time &, const boost::gregorian::years &);
    }
}
```

Function operator+

boost::local_time::operator+

Synopsis

```
// In header: <boost/date_time/local_time/date_duration_operators.hpp>

local_date_time
operator+(const local_date_time & t, const boost::gregorian::months & m);
```

Description

Adds a months object and a local_date_time. Result will be same day-of-month as local_date_time unless original day was the last day of month. see date_time::months_duration for more details

Function operator+=

boost::local_time::operator+=

Synopsis

```
// In header: <boost/date_time/local_time/date_duration_operators.hpp>

local_date_time
operator+=(local_date_time & t, const boost::gregorian::months & m);
```

Description

Adds a months object to a local_date_time. Result will be same day-of-month as local_date_time unless original day was the last day of month. see date_time::months_duration for more details

Function operator-

boost::local_time::operator-

Synopsis

```
// In header: <boost/date_time/local_time/date_duration_operators.hpp>

local_date_time
operator-(const local_date_time & t, const boost::gregorian::months & m);
```

Description

Subtracts a months object and a local_date_time. Result will be same day-of-month as local_date_time unless original day was the last day of month. see date_time::months_duration for more details

Function operator-=

boost::local_time::operator-=

Synopsis

```
// In header: <boost/date_time/local_time/date_duration_operators.hpp>

local_date_time
operator-=(local_date_time & t, const boost::gregorian::months & m);
```

Description

Subtracts a months object from a local_date_time. Result will be same day-of-month as local_date_time unless original day was the last day of month. see date_time::months_duration for more details

Function operator+

boost::local_time::operator+

Synopsis

```
// In header: <boost/date_time/local_time/date_duration_operators.hpp>

local_date_time
operator+(const local_date_time & t, const boost::gregorian::years & y);
```

Description

Adds a years object and a local_date_time. Result will be same month and day-of-month as local_date_time unless original day was the last day of month. see date_time::years_duration for more details

Function operator+=

boost::local_time::operator+=

Synopsis

```
// In header: <boost/date_time/local_time/date_duration_operators.hpp>

local_date_time
operator+=(local_date_time & t, const boost::gregorian::years & y);
```

Description

Adds a years object to a local_date_time. Result will be same month and day-of-month as local_date_time unless original day was the last day of month. see date_time::years_duration for more details

Function operator-

boost::local_time::operator-

Synopsis

```
// In header: <boost/date_time/local_time/date_duration_operators.hpp>

local_date_time
operator-(const local_date_time & t, const boost::gregorian::years & y);
```

Description

Subtracts a years object and a local_date_time. Result will be same month and day-of-month as local_date_time unless original day was the last day of month. see date_time::years_duration for more details

Function operator-=

boost::local_time::operator-=

Synopsis

```
// In header: <boost/date_time/local_time/date_duration_operators.hpp>

local_date_time
operator-=(local_date_time & t, const boost::gregorian::years & y);
```

Description

Subtracts a years object from a local_date_time. Result will be same month and day-of-month as local_date_time unless original day was the last day of month. see date_time::years_duration for more details

Header <boost/date_time/local_time/dst_transition_day_rules.hpp>

```

namespace boost {
    namespace local_time {
        struct partial_date_rule_spec;
        struct first_last_rule_spec;
        struct last_last_rule_spec;
        struct nth_last_rule_spec;
        struct nth_kday_rule_spec;

        typedef date_time::dst_day_calc_rule< gregorian::date > dst_calc_rule; // Provides rule of the
        the form starting Apr 30 ending Oct 21.
        typedef date_time::day_calc_dst_rule< partial_date_rule_spec > partial_date_dst_rule; // Provides
        rule of the form first Sunday in April, last Saturday in Oct.
        typedef date_time::day_calc_dst_rule< first_last_rule_spec > first_last_dst_rule; // Provides
        rule of the form first Sunday in April, last Saturday in Oct.
        typedef date_time::day_calc_dst_rule< last_last_rule_spec > last_last_dst_rule; // Provides
        rule of the form last Sunday in April, last Saturday in Oct.
        typedef date_time::day_calc_dst_rule< nth_last_rule_spec > nth_last_dst_rule; // Provides
        rule in form of [1st|2nd|3rd|4th] Sunday in April, last Sunday in Oct.
        typedef date_time::day_calc_dst_rule< nth_kday_rule_spec > nth_kday_dst_rule; // Provides
        rule in form of [1st|2nd|3rd|4th] Sunday in April/October.
        typedef
        def date_time::day_calc_dst_rule< nth_kday_rule_spec > nth_day_of_the_week_in_month_dst_rule; //
        Provides rule in form of [1st|2nd|3rd|4th] Sunday in April/October.
    }
}

```

Struct partial_date_rule_spec

boost::local_time::partial_date_rule_spec

Synopsis

```

// In header: <boost/date_time/local_time/dst_transition_day_rules.hpp>

struct partial_date_rule_spec {
    // types
    typedef gregorian::date          date_type;
    typedef gregorian::partial_date start_rule;
    typedef gregorian::partial_date end_rule;
};

```

Struct first_last_rule_spec

boost::local_time::first_last_rule_spec

Synopsis

```
// In header: <boost/date_time/local_time/dst_transition_day_rules.hpp>

struct first_last_rule_spec {
    // types
    typedef gregorian::date          date_type;
    typedef gregorian::first_kday_of_month start_rule;
    typedef gregorian::last_kday_of_month end_rule;
};
```

Struct last_last_rule_spec

boost::local_time::last_last_rule_spec

Synopsis

```
// In header: <boost/date_time/local_time/dst_transition_day_rules.hpp>

struct last_last_rule_spec {
    // types
    typedef gregorian::date          date_type;
    typedef gregorian::last_kday_of_month start_rule;
    typedef gregorian::last_kday_of_month end_rule;
};
```

Struct nth_last_rule_spec

boost::local_time::nth_last_rule_spec

Synopsis

```
// In header: <boost/date_time/local_time/dst_transition_day_rules.hpp>

struct nth_last_rule_spec {
    // types
    typedef gregorian::date          date_type;
    typedef gregorian::nth_kday_of_month start_rule;
    typedef gregorian::last_kday_of_month end_rule;
};
```

Struct nth_kday_rule_spec

boost::local_time::nth_kday_rule_spec

Synopsis

```
// In header: <boost/date_time/local_time/dst_transition_day_rules.hpp>
```

```
struct nth_kday_rule_spec {
    // types
    typedef gregorian::date          date_type;
    typedef gregorian::nth_kday_of_month start_rule;
    typedef gregorian::nth_kday_of_month end_rule;
};
```

Header <boost/date_time/local_time/local_date_time.hpp>

```
namespace boost {
    namespace local_time {
        struct ambiguous_result;
        struct time_label_invalid;
        struct dst_not_valid;

        template<typename utc_time_ = posix_time::ptime,
                 typename tz_type = date_time::time_zone_base<utc_time_,char> >
            class local_date_time_base;

        typedef local_date_time_base local_date_time; // Use the default parameters to define local_date_time.
    }
}
```

Struct ambiguous_result

boost::local_time::ambiguous_result — simple exception for reporting when STD or DST cannot be determined

Synopsis

```
// In header: <boost/date_time/local_time/local_date_time.hpp>
```

```
struct ambiguous_result {
    // construct/copy/destruct
    ambiguous_result(std::string const & = std::string());
};
```

Description

ambiguous_result public construct/copy/destruct

1. `ambiguous_result(std::string const & msg = std::string());`

Struct time_label_invalid

boost::local_time::time_label_invalid — simple exception for when time label given cannot exist

Synopsis

```
// In header: <boost/date_time/local_time/local_date_time.hpp>

struct time_label_invalid {
    // construct/copy/destroy
    time_label_invalid(std::string const & = std::string());
};
```

Description

time_label_invalid public construct/copy/destroy

1. `time_label_invalid(std::string const & msg = std::string());`

Struct dst_not_valid

boost::local_time::dst_not_valid

Synopsis

```
// In header: <boost/date_time/local_time/local_date_time.hpp>

struct dst_not_valid {
    // construct/copy/destroy
    dst_not_valid(std::string const & = std::string());
};
```

Description

dst_not_valid public construct/copy/destroy

1. `dst_not_valid(std::string const & msg = std::string());`

Class template local_date_time_base

boost::local_time::local_date_time_base — Representation of "wall-clock" time in a particular time zone.

Synopsis

```
// In header: <boost/date_time/local_time/local_date_time.hpp>

template<typename utc_time_ = posix_time::ptime,
        typename tz_type = date_time::time_zone_base<utc_time_, char> >
class local_date_time_base {
public:
    // types
    typedef utc_time_                utc_time_type;
    typedef utc_time_type::time_duration_type time_duration_type;
    typedef utc_time_type::date_type      date_type;
    typedef date_type::duration_type      date_duration_type;
    typedef utc_time_type::time_system_type time_system_type;

    enum DST_CALC_OPTIONS { EXCEPTION_ON_ERROR, NOT_DATE_TIME_ON_ERROR };

    // construct/copy/destruct
    local_date_time_base(utc_time_type, boost::shared_ptr< tz_type >);
    local_date_time_base(date_type, time_duration_type,
        boost::shared_ptr< tz_type >, bool);
    local_date_time_base(date_type, time_duration_type,
        boost::shared_ptr< tz_type >, DST_CALC_OPTIONS);
    local_date_time_base(const local_date_time_base &);
    explicit local_date_time_base(const boost::date_time::special_values,
        boost::shared_ptr< tz_type > = boost::shared_ptr< tz_type >());
    ~local_date_time_base();

    // public member functions
    boost::shared_ptr< tz_type > zone() const;
    bool is_dst() const;
    utc_time_type utc_time() const;
    utc_time_type local_time() const;
    std::string to_string() const;
    local_date_time_base
    local_time_in(boost::shared_ptr< tz_type >,
        time_duration_type = time_duration_type(0, 0, 0)) const;
    std::string zone_name(bool = false) const;
    std::string zone_abbrev(bool = false) const;
    std::string zone_as_posix_string() const;
    bool operator==(const local_date_time_base &) const;
    bool operator!=(const local_date_time_base &) const;
    bool operator<(const local_date_time_base &) const;
    bool operator<=(const local_date_time_base &) const;
    bool operator>(const local_date_time_base &) const;
    bool operator>=(const local_date_time_base &) const;
    local_date_time_base operator+(const date_duration_type &) const;
    local_date_time_base operator+=(const date_duration_type &);
    local_date_time_base operator-(const date_duration_type &) const;
    local_date_time_base operator-=(const date_duration_type &);
    local_date_time_base operator+(const time_duration_type &) const;
    local_date_time_base operator+=(const time_duration_type &);
    local_date_time_base operator-(const time_duration_type &) const;
    local_date_time_base operator-=(const time_duration_type &);
    time_duration_type operator-(const local_date_time_base &) const;

    // public static functions
    static time_is_dst_result
    check_dst(date_type, time_duration_type, boost::shared_ptr< tz_type >);
```

```
// private member functions
utc_time_type
construction_adjustment(utc_time_type, boost::shared_ptr< tz_type >, bool);
std::string zone_as_offset(const time_duration_type &, const std::string &) const;
};
```

Description

Representation of "wall-clock" time in a particular time zone `Local_date_time_base` holds a time value (date and time offset from 00:00) along with a time zone. The time value is stored as UTC and conversions to wall clock time are made as needed. This approach allows for operations between wall-clock times in different time zones, and daylight savings time considerations, to be made. Time zones are required to be in the form of a `boost::shared_ptr<time_zone_base>`.

`local_date_time_base` public construct/copy/destruct

1.

```
local_date_time_base(utc_time_type t, boost::shared_ptr< tz_type > tz);
```

This constructor interprets the passed time as a UTC time. So, for example, if the passed timezone is UTC-5 then the time will be adjusted back 5 hours. The time zone allows for automatic calculation of whether the particular time is adjusted for daylight savings, etc. If the time zone shared pointer is null then time stays unadjusted.

Parameters: t A UTC time
 tz Timezone for to adjust the UTC time to.

2.

```
local_date_time_base(date_type d, time_duration_type td,
                    boost::shared_ptr< tz_type > tz, bool dst_flag);
```

This constructs a local time -- the passed time information understood to be in the passed tz. The DST flag must be passed to indicate whether the time is in daylight savings or not.

Throws: -- [time_label_invalid](#) if the time passed does not exist in the given locale. The non-existent case occurs typically during the shift-back from daylight savings time. When the clock is shifted forward a range of times (2 am to 3 am in the US) is skipped and hence is invalid. [dst_not_valid](#) if the DST flag is passed for a period where DST is not active.

3.

```
local_date_time_base(date_type d, time_duration_type td,
                    boost::shared_ptr< tz_type > tz,
                    DST_CALC_OPTIONS calc_option);
```

This constructs a local time -- the passed time information understood to be in the passed tz. The DST flag is calculated according to the specified rule.

4.

```
local_date_time_base(const local_date_time_base & rhs);
```

Copy constructor.

5.

```
explicit local_date_time_base(const boost::date_time::special_values sv,
                             boost::shared_ptr< tz_type > tz = boost::shared_ptr< tz_type >());
```

Special values constructor.

6.

```
~local_date_time_base();
```

Simple destructor, releases time zone if last referrer.

local_date_time_base public member functions

1. `boost::shared_ptr< tz_type > zone() const;`

returns time zone associated with calling instance

2. `bool is_dst() const;`

returns false if time_zone is NULL and if time value is a special_value

3. `utc_time_type utc_time() const;`

Returns object's time value as a utc representation.

4. `utc_time_type local_time() const;`

Returns object's time value as a local representation.

5. `std::string to_string() const;`

Returns string in the form "2003-Aug-20 05:00:00 EDT".

Returns string in the form "2003-Aug-20 05:00:00 EDT". If time_zone is NULL the time zone abbreviation will be "UTC". The time zone abbrev will not be included if calling object is a special_value

6. `local_date_time_base
local_time_in(boost::shared_ptr< tz_type > new_tz,
 time_duration_type td = time_duration_type(0, 0, 0)) const;`

returns a `local_date_time_base` in the given time zone with the optional time_duration added.

7. `std::string zone_name(bool as_offset = false) const;`

Returns name of associated time zone or "Coordinated Universal Time".

Optional bool parameter will return time zone as an offset (ie "+07:00" extended iso format). Empty string is returned for classes that do not use a time_zone

8. `std::string zone_abbrev(bool as_offset = false) const;`

Returns abbreviation of associated time zone or "UTC".

Optional bool parameter will return time zone as an offset (ie "+0700" iso format). Empty string is returned for classes that do not use a time_zone

9. `std::string zone_as_posix_string() const;`

returns a posix_time_zone string for the associated time_zone. If no time_zone, "UTC+00" is returned.

10. `bool operator==(const local_date_time_base & rhs) const;`

Equality comparison operator.

Equality comparison operator

11. `bool operator!=(const local_date_time_base & rhs) const;`

Non-Equality comparison operator.

12. `bool operator<(const local_date_time_base & rhs) const;`

Less than comparison operator.

13. `bool operator<=(const local_date_time_base & rhs) const;`

Less than or equal to comparison operator.

14. `bool operator>(const local_date_time_base & rhs) const;`

Greater than comparison operator.

15. `bool operator>=(const local_date_time_base & rhs) const;`

Greater than or equal to comparison operator.

16. `local_date_time_base operator+(const date_duration_type & dd) const;`

Local_date_time + date_duration.

17. `local_date_time_base operator+=(const date_duration_type & dd);`

Local_date_time += date_duration.

18. `local_date_time_base operator-(const date_duration_type & dd) const;`

Local_date_time - date_duration.

19. `local_date_time_base operator-=(const date_duration_type & dd);`

Local_date_time -= date_duration.

20. `local_date_time_base operator+(const time_duration_type & td) const;`

Local_date_time + time_duration.

21. `local_date_time_base operator+=(const time_duration_type & td);`

Local_date_time += time_duration.

22. `local_date_time_base operator-(const time_duration_type & td) const;`

Local_date_time - time_duration.

23. `local_date_time_base operator-=(const time_duration_type & td);`

Local_date_time -= time_duration.

24. `time_duration_type operator-(const local_date_time_base & rhs) const;`

local_date_time -= local_date_time --> time_duration_type

local_date_time_base public static functions

1. `static time_is_dst_result
check_dst(date_type d, time_duration_type td, boost::shared_ptr< tz_type > tz);`

Determines if given time label is in daylight savings for given zone.

Determines if given time label is in daylight savings for given zone. Takes a date and time_duration representing a local time, along with time zone, and returns a time_is_dst_result object as result.

local_date_time_base private member functions

1. `utc_time_type
construction_adjustment(utc_time_type t, boost::shared_ptr< tz_type > z,
bool dst_flag);`

Adjust the passed in time to UTC?

2. `std::string zone_as_offset(const time_duration_type & td,
const std::string & separator) const;`

Simple formatting code -- todo remove this?

Header **<boost/date_time/local_time/local_time_io.hpp>**

```

namespace boost {
    namespace local_time {
        typedef boost::date_time::time_facet< local_date_time, wchar_t > wlocal_time_facet;
        typedef boost::date_time::time_facet< local_date_time, char > local_time_facet;
        typedef boost::date_time::time_input_facet< local_date_time::utc_time_type, wchar_t > wlocal_time_input_facet;
        typedef boost::date_time::time_input_facet< local_date_time::utc_time_type, char > local_time_input_facet;

        // operator<< for local_date_time - see local_time docs for formatting details
        template<typename CharT, typename TraitsT>
        std::basic_ostream< CharT, TraitsT > &
        operator<<(std::basic_ostream< CharT, TraitsT > & os,
            const local_date_time & ldt);

        // input operator for local_date_time
        template<typename CharT, typename Traits>
        std::basic_istream< CharT, Traits > &
        operator>>(std::basic_istream< CharT, Traits > & is,
            local_date_time & ldt);

        // output operator for local_time_period
        template<typename CharT, typename TraitsT>
        std::basic_ostream< CharT, TraitsT > &
        operator<<(std::basic_ostream< CharT, TraitsT > & os,
            const boost::local_time::local_time_period & p);

        // input operator for local_time_period
        template<typename CharT, typename Traits>
        std::basic_istream< CharT, Traits > &
        operator>>(std::basic_istream< CharT, Traits > & is,
            boost::local_time::local_time_period & tp);
    }
}

```

Header **<boost/date_time/local_time/local_time_types.hpp>**

```

namespace boost {
    namespace local_time {
        typedef boost::date_time::period< local_date_time, boost::posix_time::time_duration > local_time_period;
        typedef date_time::time_itr< local_date_time > local_time_iterator;
        typedef date_time::second_clock< local_date_time > local_sec_clock;
        typedef date_time::microsec_clock< local_date_time > local_microsec_clock;
        typedef date_time::time_zone_base< posix_time::ptime, char > time_zone;
        typedef date_time::time_zone_base< posix_time::ptime, wchar_t > wtime_zone;
        typedef boost::shared_ptr< time_zone > time_zone_ptr; // Shared Pointer for custom_time_zone and posix_time_zone objects.
        typedef boost::shared_ptr< wtime_zone > wtime_zone_ptr;
        typedef date_time::time_zone_names_base< char > time_zone_names;
        typedef date_time::time_zone_names_base< wchar_t > wtime_zone_names;
    }
}

```

Header <boost/date_time/local_time/posix_time_zone.hpp>

```
namespace boost {
    namespace local_time {
        struct bad_offset;
        struct bad_adjustment;

        template<typename CharT> class posix_time_zone_base;

        typedef posix_time_zone_base< char > posix_time_zone;
    }
}
```

Struct bad_offset

boost::local_time::bad_offset — simple exception for UTC and Daylight savings start/end offsets

Synopsis

```
// In header: <boost/date_time/local_time/posix_time_zone.hpp>

struct bad_offset {
    // construct/copy/destruct
    bad_offset(std::string const & = std::string());
};
```

Description

bad_offset public construct/copy/destruct

1. `bad_offset(std::string const & msg = std::string());`

Struct bad_adjustment

boost::local_time::bad_adjustment — simple exception for UTC daylight savings adjustment

Synopsis

```
// In header: <boost/date_time/local_time/posix_time_zone.hpp>

struct bad_adjustment {
    // construct/copy/destruct
    bad_adjustment(std::string const & = std::string());
};
```

Description

bad_adjustment public construct/copy/destruct

1. `bad_adjustment(std::string const & msg = std::string());`

Class template `posix_time_zone_base`

`boost::local_time::posix_time_zone_base` — A time zone class constructed from a POSIX time zone string.

Synopsis

```
// In header: <boost/date_time/local_time/posix_time_zone.hpp>

template<typename CharT>
class posix_time_zone_base {
public:
    // types
    typedef boost::posix_time::time_duration           time_duration_type;
    typedef date_time::time_zone_names_base< CharT >   time_zone_names;
    typedef date_time::time_zone_base< posix_time::ptime, CharT > base_type;
    typedef base_type::string_type                     string_type;
    typedef CharT                                       char_type;
    typedef base_type::stringstream_type               stringstream_type;
    typedef boost::char_separator< char_type, std::char_traits< char_type > > char_separator_type;
    typedef boost::tokenizer< char_separator_type, typename string_type::const_iterator, string_type > tokenizer_type;
    typedef tokenizer_type::iterator                   tokenizer_iterator_type;

    // construct/copy/destruct
    posix_time_zone_base(const string_type &);
    ~posix_time_zone_base();

    // public member functions
    string_type std_zone_abbrev() const;
    string_type dst_zone_abbrev() const;
    string_type std_zone_name() const;
    string_type dst_zone_name() const;
    bool has_dst() const;
    posix_time::ptime dst_local_start_time(gregorian::greg_year) const;
    posix_time::ptime dst_local_end_time(gregorian::greg_year) const;
    time_duration_type base_utc_offset() const;
    time_duration_type dst_offset() const;
    string_type to_posix_string() const;

    // private member functions
    void calc_zone(const string_type &);
    void calc_rules(const string_type &, const string_type &);
    void M_func(const string_type &, const string_type &);
    void julian_no_leap(const string_type &, const string_type &);
    void julian_day(const string_type &, const string_type &);

    // private static functions
    static std::string td_as_string(const time_duration_type &);
};
```

Description

A POSIX time zone string takes the form of:

"std offset dst [offset],start[/time],end[/time]" (w/no spaces) 'std' specifies the abbrev of the time zone.

'offset' is the offset from UTC.

'dst' specifies the abbrev of the time zone during daylight savings time.

The second offset is how many hours changed during DST. Default=1

'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. Time defaults=02:00:00

'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}

Example "PST-5PDT01:00:00,M4.1.0/02:00:00,M10.1.0/02:00:00"

Exceptions will be thrown under these conditions:

An invalid date spec (see date class)

A `boost::local_time::bad_offset` exception will be thrown for:

A DST start or end offset that is negative or more than 24 hours

A UTC zone that is greater than +14 or less than -12 hours

A `boost::local_time::bad_adjustment` exception will be thrown for:

A DST adjustment that is 24 hours or more (positive or negative)

Note that UTC zone offsets can be greater than +12: <http://www.worldtimezone.com/utc/utc+1200.html>

posix_time_zone_base public construct/copy/destruct

```
1.  posix_time_zone_base(const string_type & s);
```

Construct from a POSIX time zone string.

```
2.  ~posix_time_zone_base();
```

posix_time_zone_base public member functions

```
1.  string_type std_zone_abbrev() const;
```

String for the zone when not in daylight savings (eg: EST)

```
2.  string_type dst_zone_abbrev() const;
```

String for the timezone when in daylight savings (eg: EDT)

For those time zones that have no DST, an empty string is used

```
3.  string_type std_zone_name() const;
```

String for the zone when not in daylight savings (eg: Eastern Standard Time)

The full STD name is not extracted from the posix time zone string. Therefore, the STD abbreviation is used in it's place

```
4.  string_type dst_zone_name() const;
```

String for the timezone when in daylight savings (eg: Eastern Daylight Time)

The full DST name is not extracted from the posix time zone string. Therefore, the STD abbreviation is used in it's place. For time zones that have no DST, an empty string is used

5. `bool has_dst() const;`

True if zone uses daylight savings adjustments otherwise false.

6. `posix_time::ptime dst_local_start_time(gregorian::greg_year y) const;`

Local time that DST starts -- NADT if has_dst is false.

7. `posix_time::ptime dst_local_end_time(gregorian::greg_year y) const;`

Local time that DST ends -- NADT if has_dst is false.

8. `time_duration_type base_utc_offset() const;`

Base offset from UTC for zone (eg: -07:30:00)

9. `time_duration_type dst_offset() const;`

Adjustment forward or back made while DST is in effect.

10. `string_type to_posix_string() const;`

Returns a POSIX time_zone string for this object.

posix_time_zone_base private member functions

1. `void calc_zone(const string_type & obj);`

Extract time zone abbreviations for STD & DST as well as the offsets for the time shift that occurs and how much of a shift. At this time full time zone names are NOT extracted so the abbreviations are used in their place

2. `void calc_rules(const string_type & start, const string_type & end);`

3. `void M_func(const string_type & s, const string_type & e);`

4. `void julian_no_leap(const string_type & s, const string_type & e);`

Julian day. Feb29 is never counted, even in leap years.

5. `void julian_day(const string_type & s, const string_type & e);`

Julian day. Feb29 is always counted, but exception thrown in non-leap years.

posix_time_zone_base private static functions

1. `static std::string td_as_string(const time_duration_type & td);`

helper function used when throwing exceptions

Header <boost/date_time/local_time/tz_database.hpp>

```
namespace boost {
  namespace local_time {
    typedef date_time::tz_db_base< custom_time_zone, nth_kday_dst_rule > tz_database;
  }
}
```

Type definition tz_database

tz_database — Object populated with boost::shared_ptr<time_zone_base> objects.

Synopsis

```
// In header: <boost/date_time/local_time/tz_database.hpp>

typedef date_time::tz_db_base< custom_time_zone, nth_kday_dst_rule > tz_database;
```

Description

Object populated with boost::shared_ptr<time_zone_base> objects Database is populated from specs stored in external csv file. See date_time::tz_db_base for greater detail