Boost Date-Time

Authors Corner for BoostCon 2008

Jeff Garland

...Philosophy Moment…

Every generalization is dangerous, especially this one.

-- Mark Twain

History and Stats

- Motivation: 1 project – too many time libraries
  - Differing needs – space efficiency versus resolution
  - 98% overlap
  - Convinced ‘OO’ design couldn’t work
- Stumbled across Boost around that time
- Reviewed and accepted into 1.29
  - Lacking a few semi-important features
- Later releases brought
  - format-based i/o facets
  - Time zones and local_time
- Library size
  - ~ 5000 statements (counted on semi-colons)
  - Test/Example code: ~6000 statements

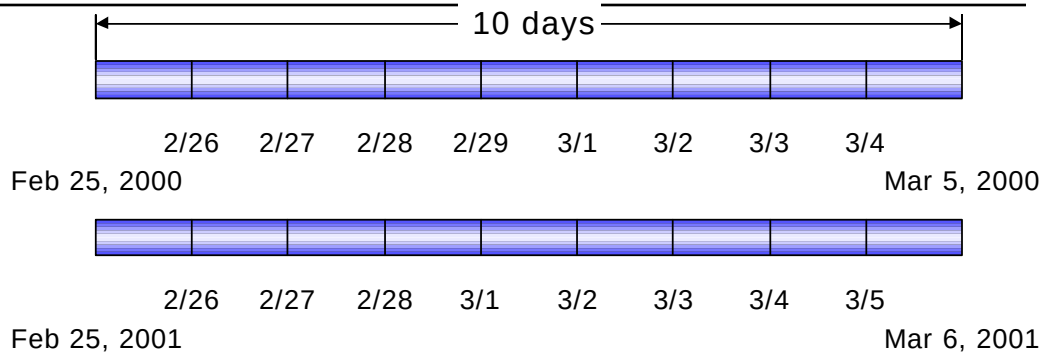
Where is Date-Time Used Now?

- Many, many users worldwide
  - Who's Using Boost only scratches the surface
- Boost.Thread 1.35 uses duration types
- Boost.Asio uses duration types for timing
- Proposals to C++ Committee
  - TR2 for full date-time interfaces
  - C++0x threading interfaces

Library Design Requirements

- Basic Interfaces
  - Date, Time, Durations, Periods
  - Represented as valuetype – should behave like ‘integer’
- Calculation
  - Efficient and type safe
  - No hidden calculation error -- round off effects
- Representation
  - Different calendars
  - Different date epoch’s an time precisions
- Clock Interfaces
- Various I/O Interfaces

Some Primary Concepts in Date-Time



- Point in Time
  - 2008-May-04 00:00:00.0
- Duration
  - 10 minutes
  - 2 days
- Period
  - [2008-May-04 00:00:00.0/2008-May-05 00:00:00.0)
- Clock
  - Hardware device to get the current time from the computer
- Resolution
  - Smallest representable time value
- Time Zones
  - Assist in representing local times
- Calendar
  - Maps time to ‘label’
  - Core calculation engine for representing years

Realization of Concepts

- Duration Types:
  - hours, minutes, seconds
  - microseconds, milliseconds, nanoseconds
  - **years, months**, weeks, days,
- Time Point Types:
  - date, ptime, local_time
- Period Types
  - date_period, time_period, local_time_period
- Iterator Type
  - date_iterator, time_iterator

From Concepts to Types

- Durations as types
  - Want to be able to represent durations in function interfaces and as data members
  - 1 set of durations types provide core library
- Time point types
- Durations and Time Points define the mathematical rules

Doing the Math - Examples

```
using namespace boost::gregorian;
date weekstart(2002,Feb,1);
date weekend = weekstart + week(1);
date d2 = d1 + days(5);

date today = day_clock::local_day();
if (d2 >= today) {} //date comparison operators
```

Doing the Math – Time Examples

```
using namespace boost::posix_time;
date d(2002,Feb,1); //an arbitrary date
ptime t1( d, hours(5) + nanoseconds(100) );
//date + time of day offset
ptime t2 = t1 - minutes(4) + seconds(2);
ptime now = second_clock::local_time();
now += nanoseconds(100);
now -=  days(1);
```

Doing the Math: Time Points

Defined:

- Timepoint + Duration --> Timepoint
- Timepoint - Duration --> Timepoint
- Timepoint - Timepoint --> Duration

Undefined:

- Duration + Timepoint --> Undefined
- Duration - Timepoint --> Undefined
- Timepoint + Timepoint --> Undefined

Doing the Math: Special Values

- Special Values – ‘not a date time’, infinities
- What happens when the user does arithmetic with these?

Timepoint(NADT) + Duration --> Timepoint(NADT)

Timepoint(∞) + Duration --> Timepoint(∞)

Timepoint + Duration(∞) --> Timepoint(∞)

Timepoint - Duration(∞) --> Timepoint(-∞)

Doing the Math – Non Reversibility

- Unfortunately some time durations types aren't fixed size
- Leads to an issue with math reversibility
- Initially left these types out – added due to user request

```
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
```

Format Based I/O

```
date d(2004, Feb, 29);
time_duration td(12,34,56,789);
stringstream ss;
ss << d << ' ' << td;  //"2004-Feb-29 12:34:56.000789"

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"
```

Facet Types and Format Specifiers

- Facet Type breakout
  - Wide and Narrow stream versions
  - Input and Output facets for each time point type
- Facets Provide “complete control” over input and output
- Date Facets
  - date_facet / wdate_facet
  - date_input_facet / wdate_input_facet
- Time Facets
  - time_facet / wtime_facet
  - time_input_facet / wtime_input_facet
- Local Time Facets
  - local_time_facet* / wlocal_time_facet*
  - local_time_input_facet* / wlocal_time_input_facet*

Using Format Strings - Localization

- Like strftime
  - Extensions for items not supported by strftime
- %x %X date/time format from the imbued locale.

```
ptime pt(date d(2005,Oct,31), 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"
```

Using Format Strings – ISO format

%Y%m%dT%H%M%S%F%q

// Oct 15, 2005 13:12:11 MST

ISO: "20051015T131211-0700“

- Simple way – call methods to set iso format

date_facet f = …

f->set_iso_format(); // "%Y%m%d

f->set_iso_extended_format(); // "%Y-%m-%d"

Changing the Format

```
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"
```

Date-Time: Flight Challenge

- Challenge: Program to calculate local arrival times for a cross country flight (Phoenix to New York) that flies through the daylight savings transition
- Flight is 4 hours and 30 minutes
- DST changed on April 2, 2005

User Code for Flight Challenge

```
typedef boost::shared_ptr<time_zone_base> zone_ptr;

 //setup some timezones for creating and adjusting times
 tz_database tz_db;
 tz_db.load_from_file("date_time_zonespec.csv");
 zone_ptr nyc_tz = tz_db.time_zone_from_region("America/New_York");
 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);
```

Hidden Gem: constrained_value

- Problem: Need to range check myriad of values on various interfaces.
  - month: 1..12
  - day: 1..31
- Solution: constrained_value template

- Stolen from Pascal

  var x: 1..10;

- A bit more capable
  - Allows for customization of
- Not really so hidden
  - Dr. Dobbs article

    http://www.ddj.com/cpp/184401886
  - Boost library proposal floating around

constrained_value template in detail

```
template<class value_policies>
class constrained_value {
public:
  typedef typename value_policies::value_type value_type;
  constrained_value(value_type value)
 {
     //does range checking – calls value_policies:on_error if
     //value is out of range
      assign(value);
  }
  constrained_value& operator=(value_type v) ; //calls assign
  static value_type max() {return (value_policies::max)();};
  static value_type min () {return (value_policies::min)();};
  operator value_type() const {return value_;};
```

Realization

- 3 typedefs define the rules for ‘months’

greg_month.hpp

```
// Exception thrown if a greg_month is constructed with a value out of range
struct bad_month : public std::out_of_range
{
  bad_month() : std::out_of_range(std::string("Month number is out of range 1..12")) {}
};

// Build a policy class for the greg_month_rep
typedef CV::simple_exception_policy<unsigned short, 1, 12, bad_month> greg_month_policies;

// A constrained range that implements the gregorian_month rules
typedef CV::constrained_value<greg_month_policies> greg_month_rep;
```

Hidden Gems: Templates and Text

- Problem: Need a string constant as part of a library.
- Conventional wisdom
  - Definition in .cpp file
  - Build strings into a library to link
- Templates can allow header only implementation without ‘multiply defined’ linking problem

Conventional Approach

```
//date_facet.hpp
//mythical class for this example
class date_facet {
private:
    static const char default_period_separator[4];
//….
};
//…

//date_facet.cpp
date_facet::default_period_separator[4] = “ * “;
```

Template Solution

```
template <class date_type,
          class CharT,
          class OutItrT = std::ostreambuf_iterator<CharT, std::char_traits<CharT> > >
  class date_facet : public std::locale::facet {
  public:
    static const char_type default_period_separator[4];
//….
};
//…

  template <class date_type, class CharT, class OutItrT>
  const typename date_facet<date_type, CharT, OutItrT>::char_type
  date_facet<date_type, CharT, OutItrT>::default_period_separator[4] = { ' ', '/', ' '};
```

Things I've Learned

- Valuetype programming is hard and under appreciated
  - Valuetypes often present large complex interfaces
  - Must be extremely efficient
- Writing good C++ I/O is hard
  - Extremely hard to understand all factors to write a truly standard operator<<
  - No one does it 'correctly'
  - Implementing facets and manipulators is painful
- Naming is hard
  - The ptime mistake
    - Needed to avoid 'time' because of conflicts on some older platforms
    - Should be date_time, but used that in the namespace
    - Too obscure
- Documentation best practice
  - Examples for every function
  - Date-time uses table
    - Function signature, description, example

Future Directions

- Duration Types
  - Will be refactored to match new standardization proposals
  - Replaces inheritance
- Standard
  - C++ 0x - ?
  - TR2
- 2008 Boost Summer of Code Project
  - New calendar types – historical and astronomical
- Boost.Timer integration – one of these centuries