Rebuilding Boost Date Time for C++11/14

Jeff Garland
jeff@crystalclearsoftware.com

Alternate title:

C++11/14 Features for Building Valuetype Classes - an exploration

What Will This Talk Achieve?

- Examination of C++11 & 14 features for building ValueType classes
  - Using boost date-time as an example
  - Differences from c++98
- General discussion of considerations for ValueType classes
- Preview of date-time v2

Background

- boost date-time (bdt)
    - v1 put into boost in 1.29 in 2002
    - used in many, many projects
- Good
    - Simple to use and fairly powerful interface
    - Fanatical error checking
    - I/O
- Bad
    - Fanatical error checking (it's somewhat slow)
    - Relatively hard to extend
    - Many code base hacks for pre-2000 compilers (eg: g++2.9.8, vc6)
- Ugly
    - I/O - facet based strftime based interfaces is relatively slow

```
//bdt v1
using namespace boost::gregorian;
date weekstart(2002,Feb,1);         //obvious construction



//math
date weekend = weekstart + week(1);
date d2 = d1 + days(5);



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

date-time & chrono

- BDT v1 the basis is elements in <chrono>
- but…c++11 was NOT ready for a full date-time library
    - proposals were too late and immature (originally was targeting TR2)
    - date-time v1 is large enough that it's hard to standardize
    - committee was too busy
    - it was all we could do to get chrono…at the time
- Excuses :)

C++11/14 Features Covered

- Design considerations for the 'date' class (mostly)
- Specifically c++11/14 feature considerations (language and library)
    - final
    - noexcept (error handling)
    - move construction/assignment (R-values)
    - constexpr
    - member initialization
    - enum types
    - user defined literals
    - template aliases
    - std::to_string
    - delegating constructors
- Also, general class considerations
    - default construction
    - templated construction / conversion

Time to Start Over!

Reconsider everything…

What’s easiest to use type in C++?

- int!
- Want date to be as easy as an int
- BDT v1 date is close…but not quite
  - as easy as int
  - or as fast
  - much closer to ‘double’…which is a nightmare, really
- date can never be as good as int but we can do better than bdt v1

Properties of an Int

- Hard to break
  - Never throws exceptions
    - implicitly all functions are noexcept
  - used in virtually ever program — mostly error free
  - fast at almost everything
  - now in c++11 good conversions from string
- What can go wrong with an int?
  - fail to initialize (easy fix there)
  - overflow on arithmetic
  - signed versus unsigned compare (warnings)

Design Considerations

- Fast (er)
- Simple (r) consistent interface
- I/O, I/O, I/O
- Extensible, extensible, extensible
- Play nice with <chrono>
- Play nice with BDT v1 — if possible…
- Something that can be standardized…

C++11/14 Features 1 by 1

final

- c++11 supports keyword ‘final’
- In function context prevents derived classes from overloading
- In class context makes derivation an error

date as final class

```
class date final {.....};
```

```
class mydate : public boost::date_time2::date { }
```

```
> g++-4.9 -std=c++11 test.cpp
```

```
test.cpp:18:7: error: cannot derive from ‘final’ base
‘boost::date_time2::date’ in derived type ‘mydate’
 class mydate : public boost::date_time2::date
```

final - the final word

- Ultimately no classes in boost date_time2 are currently marked as final
- could prevent valid extension path — add stateless function / constructor
- no std library types use final

```
//bdt v1
using namespace boost::gregorian;
date weekstart(2002,Feb,1);        //obvious construction



//math
date weekend = weekstart + week(1);
date d2 = d1 + days(5);



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

```
//input streaming
std::stringstream ss("2004-Jan-1");
ss >> d3;


//date generator functions
date d4 = next_weekday(d3, Sunday);
```

```
//US labor day holiday is first Monday in Sept
nth_day_of_the_week_in_month labor_day(nth_dow::first,Monday, Sep);
date d6 = labor_day.get_date(2004);
```

Do you see the problem?

- Too many ways to represent a date!
- And bdt v1 was inconsistent in it’s approach

It’s all about construction

- default constructors
- move constructors (c++11)
- noexcept and errors (c++11)
- member initialization (c++11)
- templated constructor

```
date d3;
d3 += days(2); //value?
```

It’s all about construction

default constructor

- Should there be a default constructor?
  - initial answer was no in v1…
  - std::map requires key to be default constructible
  - some code clearly harder to write…
- What should it be?
  - current date…slow, slow, slow — breaks performance assumption
  - epoch - reasonable…answer for v2
  - ‘not a date time’…answer for v1

you’re so special - a diversion

- v1 had special values for date - nadt, neg_infinity, pos_infinity
- gone in v2
- advantages
  - handy for the domain on occasion
- disadvantages
  - requires addition of ~5 methods to date interface
  - date’s aren’t ‘always valid’ - can’t reason about functions
    - checking for these - aka special logic
  - i/o is harder…
  - makes ‘int’ into ‘double’

nadt == optional

- should be build optional into a low level value type?
- advantages
  - don’t need the wrapper
- disadvantages
  - burdens all applications with optional behavior
    - even if not needed
    - most don’t…
  - complexity…again

```
date d3;
d3 += days(2); //v2 == epoch() + 2 days
```

It’s all about construction

move construction?

- Should there be a move constructor?
  - value seems limited, however…
  - best practice to include anyway
  - can make it explicit in c++11

```
/// Trivial move constructor
date(date&&) noexcept = default;

/// Trivial copy constructor
date(const date&) noexcept = default;

/// Trivial assignment
date& operator=(const date&) noexcept = default;
```

It’s all about construction

Error Handling & NoExcept

- Problem: Sometimes you know your date it good…sometimes you don’t
- Solution: checking versus non-checking constructor
- Non checking variant is no-except for max speed — use with trusted source
- Checking variant insures correctness

noexcept

```
date(const year_month_day& ymd) noexcept;
date(const year_month_day& ymd, checking check);
```

- Performance boost for noexcept?
  - Unable to discern any at this point…
  - Lack of time & cleverness likely…

It’s all about construction

member initialization

- Allows class/struct data members to be explicitly initialized

```
 //user code…
 iso_week_number wn;
```

```
//library code…
struct iso_week_number {
   /** Construct to invalid state */
  iso_week_number() noexcept = default;

   …

   uint16_t year = 10001;
   uint8_t  week_no=54; ///<use iso week numbering
   uint8_t  day_in_week=8; ///<1==monday...7==sunday
```

Date - How do I represent thee?

- Strings — a gazillion variations
    - localization anyone?
    - iso
- month, day, year
- iso week number and day in week
- Calculated
    - third sunday in feb of 2014
    - last sunday in mar of 2014
    - sunday of week 5 in 2014

- modified julian day & julian day
- time_t and tm
- mayan calendar?
- klingon calendar?

It’s all about construction

templated construction

- Really, I don’t know how you’re going to represent a date…seriously, I don’t…

std::tm

```
//v1
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);
```

```
//v2
tm t;
t.tm_year = 113;//2013
t.tm_mon = 11; //Dec
t.tm_mday = 30;
date d(t);
```

std::tuple & chrono

```
//v2 only — can’t write in v1
std::tuple<int, int, int> t_ymd(1900, 1, 1);
date d(t_ymd);
```

```
using namespace std::chrono;
system_clock::time_point tp = system_clock::now();
date d(tp); //v2 only
```

```
//v2 api
using namespace boost::date_time2;
date d(2012, 1, 1);
year_month_day ymd(2012, 1, 1);
date d2(ymd);

date d1(year_month_day(“20140401T000000”), ISO);

date d2(year_month_day(“2014-04-01”));
date d1(year_month_day(“2004-10-01"));
date d2(year_month_day("2004/10/01"));


//calculated dates
day_of_week dow(First, Wed, Jan, 2013 );
date d(dow);

date d(day_of_year(2014, 1)); //jan 1, 2014

closest_day_of_week pdw(Sun, Before, 2013, May, 17);
date d(pdw);
```

```
iso_week_number wn1(“2014-W01-2");
date d1(wn1);

iso_week_number wn2("2014", "W1", “2");
date d2(wn2);

iso_week_number wn3(2014, 1, 5);
date d3(wn3); //2014-Jan-3
```

```
//bridge from bdt v1
boost::gregorian::date bd(1900, 1, 1);
date d(bd);

boost::posix_time::ptime pt(bd);
date d2(bd);
```

How is it done?

- Templated constructor
- Specializations for different types
- Allows users to add new representation — all construction is the same
- Alternative
    - Construct everything using make_date function
    - Users can provide their own
    - Feels odd for date…

Under the Hood

```
//declaration…
template<typename T>
explicit date(const T& t) noexcept;
```

```
/// Template specialization to construct a date from a chrono::system_clock::time_point
template<>
date::date<std::chrono::system_clock::time_point>(const std::chrono::system_clock::time_point& tp) noexcept
 {
   using namespace std::chrono;
   std::time_t tt = system_clock::to_time_t(tp);
   from_time_t(tt);
 }
```

User Defined Literals

- C+11 allows creation of user defined literal
- Convert a literal in code to a type
- C++14 has pre-defined values for chrono

user defined literal example

```
#include <chrono>
#include <iostream>
using namespace std::chrono;
using namespace std::literals::chrono_literals;
```

```
hours h(1);  //traditional
auto ns = 1h + 20us; //ns type == chrono::nanoseconds
std::cout << ns.count() << std::endl;
```

Nice, but…

- Standard limits user defined literals
    - must include underscore
    - _w for week?
- Types are limited…
- Easy to implement?

snippet of chrono ‘h’ operator

```
constexpr chrono::duration<long double, ratio<3600,1>>
operator""h(long double __hours)
{ return chrono::duration<long double, ratio<3600,1>>{__hours}; }

template <char... _Digits>
  constexpr typename
  __select_type::_Select_type<__select_int::_Select_int<_Digits...>::value,
                       chrono::hours>::type
  operator""h()
  {
   return __select_type::_Select_type<
                    __select_int::_Select_int<_Digits...>::value,
                    chrono::hours>::value;
  }
```

what’s wrong with this?

```
constexpr boost::date_time2::days operator""_d(short d)
{
  return boost::date_time2::days(d);
}
```

```
g++-4.9 -I ../bdt2 -std=c++14 test.cpp
test.cpp:22:45: error: ‘boost::gregorian::days boost::date_time2::literals::operator""_d(short int)’ has invalid argument list
 boost::date_time2::days operator""_d(short d)
                             ^
```

still no joy…

```
constexpr days operator”"_d(unsigned long long d)
{
  return days(d);
}
```

error…

boost::date_time2::days’ is not an aggregate, does not have a trivial default constructor, and has no constexpr constructor that is not a copy or move constructor

Remove the constexpr for now

```
~~constexpr~~ days operator”"_d(unsigned long long d)
{
  return days(d);
}
```

- question: is it worth it to be able to write?

  ```
  auto d = 1_d + 2_w; //15 days

  date d {2014, 1, 1};

  d+= 1_w;
  ```

- ???

constexpr

- Generalized constant expression
- Function evaluated at compile time
- Some C++11 limits
  - typically needs a single return value
  - can only call other constexpr functions
- C++14 generalizes constexpr
  - Allows control structures (if/switch)
  - Local variables

obvious constexpr

```
static constexpr uint8_t  day_of_month_min() { return 1; }
static constexpr uint8_t  day_of_month_max() { return 31; }

static void validate_ymd(uint16_t year, uint8_t month, uint8_t day)
{
range_check("year", year_min(), year_max(), year);
range_check("month", month_min(), month_max(), month);
range_check("day",  day_of_month_min(), day_of_month_max(), day);
}
```

constexpr limits

- consider ‘max_date’ function
- returns a date that represents max representable date

date::max_date

```
static constexpr date max_date()
{
	return date(year_max()-1,
				month_max(),
				day_of_month_max());
}
```

well, maybe not…

```
g++-4.8 -I ~/devtools/boost_1_55_0 -std=c++11 test.cpp
In file included from test.cpp:6:0:
date.hpp: In static member function ‘static constexpr boost::date_time2::date
boost::date_time2::date::max_date()’:
date.hpp:85:29: error: invalid return type ‘boost::date_time2::date’ of constexpr function ‘static constexpr
boost::date_time2::date boost::date_time2::date::max_date()’
    static constexpr date max_date()
                 ^
date.hpp:78:11: note: ‘boost::date_time2::date’ is not literal because:
   class date : public date_base<gregorian_calendar>
       ^
date.hpp:78:11: note:   ‘boost::date_time2::date’ is not an aggregate, does not have a trivial default
constructor, and has no constexpr constructor that is not a copy or move constructor
```

std::to_string

- new standard library functions to convert many integral types to std::string
- types covered include integer and floating point of various flavors
- corresponding string to type (eg: stoi) functions also there

using to_string - generic range check

```
template <typename T>
void range_check(std::string unit, T min, T max, T value)
{
  if (value > max || value < min) {
      throw std::out_of_range(unit + " is out of range "
              + std::to_string(min) + "..." + std::to_string(max)
              + ": " + std::to_string(value));

  }
}
```

delegating constructors

- Allows calling one constructor from another
- Avoids writing ‘init’ type functions

delegating constructor

```
class year_month_day {
public:
    year_month_day(const char* const ymd_string);

    template<typename T>
    explicit year_month_day(const T& ymd);
```

```
template<>
year_month_day::year_month_day(const std::string& ymd_string) :
  year_month_day(ymd_string.c_str())
{}
```

Building Valuetypes in C++11/14

- It’s a whole new world…
- Top features — Jeff’s view…
  - to_string
  - noexcept
  - explicit defaults for compiler generated constructors
  - member initialization
  - constexpr
  - delegating constructor
- questionable value
  - final
  - user defined literals

Conclusions: Valuetype Design Considerations

- How does your type integrate with others?
- Does it play well with things in standard library?
- Does if follow recognized / common patterns from standard?

Conclusion

- C++11 and 14 provide nice features for writing value types
- Implementation quality is much higher
- But don't forget about old features!
- Other resources
  - Sean Parent - C++Now 2012 - Value Semantics and Concepts-Based Polymorphism
  - Eric Niebler C++ Now 2014 - C++11 Library Design
  - Michael Caisse - The Canonical Class - Wed 9 am