# Project Proposal

I am proposing to implement chrono::date which is almost completely based on the proposal by Howard Hinnant [1]. For saving space, only deviations in the interface are mentioned in the design below.

## Implementation Concept

This proposal uses a serial-based approach with 31 days a month and, thus, 372 days a year. The main data field in date is a 32-bit integer serial, so that the class looks somewhat like:

class date

{

private:

 int serial;

 bool weekday\_construct;

public:

// Other functions
};

Note that date uses a proleptic Gregorian Calendar. Serial value 0 corresponds to 1 Jan, year 0. All the serials generated are valid and maintain the ‘<’ property i.e. serial(d1) < serial(d2), if d1 < d2. But ‘==’ is not preserved i.e. serial(d1) can be not equal to serial(d2), even if d1 == d2. This happens in the case of the “invalid” dates in the “372 day year”. Precisely, this happens for 29 Feb (only in non-leap years), 30 Feb, 31 Feb, 31 April, 31 Jun, 31 Sep and 31 Nov. They all take the preceding valid date as their date value.

In a year, 372 values exist from 0 to 371. The months Jan to Dec are denoted by 0, …, 11. Days 1st to 31st are denoted by 0, …, 30.

serial(1st day of a month) = 31\*month.

And the next day in the same month takes just the next serial number.

This leaves gaps for invalid dates mentioned before. They represent the date of the first valid serial before them. For example, Apr 30 has 2 serial numbers:

 serial(Apr 30) = 31\*Apr + 30th = 31\*3 + 29 = 122

therefore, Apr 30 => serial(Apr 30) & serial(Apr 31) = 122 & 123

All examples right now were in year 0.

For a general date Y-M-D,

serial(Y-M-D) = Y\*372 + serial(M-D) as shown above.

# date Construction

## Design

Same as the proposal but there is also another corresponding constructor for each notation mentioned in proposal [1]. For the same of distinction, we call proposal’s notation as operational approach and corresponding notation given below as functional approach.

//operational

date d1\_1 = day(1) / month(2) / year(2011);

//functional

date d1\_2 = date(1, 2, 2011, DMY);

The fourth parameter here is the format of date and is defined as:

enum date\_format {YMD, DMY, MDY, ISO = YMD, ISO\_EXTENDED};

Similarly:

//operational

date d2\_1 = jan / day(1) / 2012;

//functional

date d2\_2 = date(jan, 1, 2012, MDY);

Or,

//operational

date d3\_1 = weekday(wd)[\_1st] / may / 2011;

//functional

date d3\_2 = date( wd[\_1st], may, 2011, DMY);

Another minor addition is the constants. This proposal recommends addition of constants \_1st, \_2nd up till \_31st. Also, as before, the special constant last is also there.

## Implementation

First, the functional approach:

date::date(int arg1, int arg2, int arg3, date\_format format)

{

 int d, m, y;

 switch(format)

 {

 case YMD:

 case ISO\_EXTENDED:

 y = arg1;

 m = arg2;

 d = arg3;

 break;

 case DMY:

 d = arg1;

 m = arg2;

 d = arg3;

 break;

 case MDY:

 m = arg1;

 d = arg2;

 y = arg3;

 }

 serial = serial\_year\_period \* y + serial\_month\_period \* (m-1) + (d-1);

}

Here, implementation is straightforward but unchecked.

If we use the operational approach, we have to construct three classes day, month and year and overload the ‘/’ operator for input purposes. It is again unchecked but offers a more user-friendly syntax.

A special note about last: it is equivalent to \_31st in our implementation. The reasons become apparent when we see the sections of year and month arithmetic.

Efficiency of either approach can only be tested in benchmark tests. If one approach offers significant gains over others, it can mean the chopping of the other from the final class. But that would happen only if it the gains are maintained over a wide variety of implementations, serial-based or field-based. For now, both are in.

# Year-oriented Arithmetic

## Design

Same as HH proposal.

## Implementation

Our implementation is straightforward and takes on all the edge-cases mentioned in the proposal:

typedef boost::chrono::duration<int, boost::ratio<32140800L>> years;

date date::operator+(years Y)

{

 date ret = \*this;

 ret.serial += serial\_year\_period \* Y.count();

 return ret;

}

# Month-oriented proposal

## Design

Same as HH proposal.

## Implementation

Our implementation is straightforward and follows the logic of year-based function:

typedef boost::chrono::duration<int, boost::ratio<2678400L>> months;

date date::operator+(months M)

{

 date ret = \*this;

 ret.serial += serial\_month\_period \* M.count();

 return ret;

}

Special Note: HH proposal mentions a twist in the logic when date is constructed using weekdays, e.g. in the case of following date:

date d\_wd = date(jan, sun[\_1st], 2010, mdy);

wherein addition of year gives date(jan, sun[\_1st], 2011, ymd). We use the never-explained-before field weekday\_ctrn. During date object construction, it is switched to true when such a syntax as above is employed using weekdays, otherwise false. Thus, aforementioned functions would be modified as:

date date::operator+(months M)

{

 if(!weekday\_ctrn)

 { //same as before

 date ret = \*this;

 ret.serial += serial\_month\_period \* M.count();

 return ret;

 }

 else

 {

 //specific weekday logic; not mentioned here

 }

}

If we increment the date in terms of days, weekday\_ctrn turns to false and never changes back again.

# Day-oriented Arithmetic

## Design

Same as HH proposal.

## Implementation

Due to our chosen representation, day-oriented arithmetic became complex for simpler year and month arithmetic. A function that increments a date by multiples of days may look like:

typedef boost::chrono::duration<int, boost::ratio<86400L>> days;

//private function

void date::increment\_date\_by\_1()

{

 int rem\_days = serial % serial\_year\_period;

 switch(rem\_days)

 {

 case FEB30:

 case APR30:

 case JUN30:

 case SEP30:

 case NOV30:

 serial += 2;

 break;

 case FEB29:

 serial += 3;

 break;

 case FEB28:

 if(is\_leap()) serial += 1;

 else serial += 4;

 break;

 default:

 serial += 1;

 }

}

date date::operator+(days D)

{

 date ret = \*this;

 weekday\_ctrn = false;

 for(int i = 0; i < D.count(); i++) // can optimize better than this

 {

 ret.increment\_date\_by\_1();

 }

 return ret;

}

The constants in switch case are actually defined as:

enum yday { // start at 0 for JAN1

 FEB28 = 58, FEB29, FEB30, FEB31,

 APR30 = 122, APR31,

 JUN30 = 184, JUN31,

 SEP30 = 277, SEP31,

 NOV30 = 339, NOV31

};

Also, difference of two dates is allowed. It returns a duration of type days as defined earlier. One approach to its implementation is to subtract the two serials and then further subtract number of invalid dates in the range of the two dates for the answer. The latter part, i.e. the number of invalid dates in a range, can be made into a private function because it can also be employed in week-based arithmetic.

# Finding the next or previous weekday

## Design

Same as in HH proposal.

## Implementation

We define an accessor function weekday() that returns weekday of given date. Then we can increment the returned date by the number of days by which the two weekdays – specified in function and today’s – are spaced apart.

# date I/O

## Design

Extending the date\_fmt(“string”) notation, we overload it to say day\_fmt(iso) or date\_fmt(ymd). They are the same constants that were used when inputting the date.

## Implementation

It is straightforward. We parse the string for the order of variable and then use various accessor functions to get data from the date object. And then we format it accordingly as return value.

The overloaded date\_fmt uses a dictionary: against each date format, we have a default saved string to be used. Thus, they work as fancy wrappers around date\_fmt(“string”) only.

# Further Suggestions

Design

* I have separate things to add to library to make it analogous to the rest of chrono library. We take date to be equivalent to time\_point<system\_clock, days> and system\_calendar to be equivalent to system\_clock.

A duration type can be added to or subtracted from a date if rep is in integral multiples of days like centuries (100 years) or weeks (7 days). Thus a generic template should be provided for the purpose.

Also, a date has a function days\_since\_epoch() equivalent to time\_point::time\_since\_epoch().

Other related functions in time\_point that have defined meaning in the domain of time can be employed too.

As for system\_calendar, the following definition adequately indicates its proposed structure:

class system\_calendar // modelled on system\_clock

{

public:

 typedef days::rep rep;

 typedef days period;

 static const bool is\_steady;

 static date now();

 static days days\_since\_epoch() { return now().days\_since\_epoch(); };

};

* Additional calendars can be provided. Some examples can actually be found in HH proposal. In that case, the current calendar and date be made into a template, e.g. date<proleptic\_gregorian>. Other calendar dates thus defined are either based on date<proleptic\_gregorian> or have their own implementations. Being templates, they all have a common set of functions. To convert from one to another, a date\_cast<calendar To, calendar From> is used.
* Any date<calendar> can be changed to and from time\_t structure. This is easily achieved in implementation using the date<calendar>::time\_since\_epoch().

# Final Notes

* The operations defined here are all unchecked. Another augmented implementation is also proposed with runtime checking. Its existence depends on the difference of performance of both.
* The implementation can (and should) be changed drastically if we find a better one on the ways. Therefore, a need for standardized benchmarks early in the process is noted keenly.

# References

[1] <https://svn.boost.org/svn/boost/sandbox/chrono_date/libs/date/doc/date.html>