# Personal Details

|  |  |
| --- | --- |
| Name | Anurag Kalia |
| Institute | Hansraj College, Delhi University |
| Course | Computer Science |
| Degree | B. Sc. |
| Email | anurag.kalia@outlook.com |
| Availability | I am not involved in any internships/jobs to interfere with GSoC. Therefore, I can assure my availability as per the project demands. |

# Background Information

I am currently pursuing B.Sc. (H) Computer Science and have my exams scheduled for my final semester in the month of May.

I have keen interest in tools part of programming, like design of programming languages, protocols etc. I particularly follow the developments in the C++ language because I am long acquainted with it, being my first programming language. Much more importantly, I like programming in C++ because it is able to give me performance-critical code without sacrificing expressiveness and that it lets me play with the guts of the machine directly. I am excited even after C++11 because initializer lists and for\_each loops, auto keyword etc solve my major headaches with the language.

I learned about Boost through Q&A forums like Stack Overflow. I have always considered it to be a very reputed library, but had not seriously contemplated taking part in its development till now. I have never been part of a big community like Boost but GSoC gave me the impetus to actually look into its inner working. I am happily surprised that even after being a mature library, it is being actively developed. I intend to pursue it actively now, irrespective of GSoC. I have always bemoaned the absence of a type in C++ corresponding to ‘BigInteger’ of C#, for example. I now have a platform where I am able to put such and more ideas among a number of C++ experts.

As for the ‘date’ library that I intend to take up as my project, I am already familiar with ISO 8601 and various kinds of calendars in use as well as the existence of so many time standards. I am confident of this domain. Implementation of such a library is mostly straightforward. I expect most of the time to be taken up by the design of the APIs themselves. Further, if I work on my proposal of such library, I would be happy to maintain it for bugs and further extensions. For example, all standard calendars used in computing today are solar. I would like to see lunar and lunisolar calendars being offered too.

Ratings of my knowledge in domain (1-5):

|  |  |
| --- | --- |
| C++ | 3 |
| C++ Standard Library | 3 |
| Boost Library | 1 |
| Subversion | 1 (familiar with git though) |

IDEs used: Visual Studio on day-to-day basis, Notepad2 for lightweight purposes.

Compilers used: MSVC, g++, (infrequently) clang++.

Documentation tools used: None, till now.

# Project Proposal

I am proposing a Boost.chrono/date library that is based on and interoperates with existing Boost.chrono library. We define the length of duration “days” to be 24 “hours”, where “hours” is the pre-specified duration in Boost.chrono. Thus, a day is 86400 seconds. This fixed definition of a day follows TAI time standard rather than the conventional UTC time standard, which allows adding or subtracting of leap seconds in some specific days. This variation wouldn’t have any real-world consequence because:

* Since we involve only date-related arithmetic in this project, the difference between UTC time (at current frequency of leap seconds) and TAI time is no more than 2 hours over a period of 10000 years in the future, which is much less than a full day. Thus, UTC and TAI are not expected to diverge on date values for a very long time.
* Over a period of hundreds of years in future, the dates are so sketchy that a margin of 1 day wouldn’t really affect its accuracy in any way.
* Since there were no leap seconds before 1971, the length of a day is exactly 86400 seconds in the past

Therefore, we define basic types as follows:

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

typedef std::chrono::time\_point<std::chrono::system\_clock, days> day\_point;

class date : public day\_point

{

/\* extra fields and functions \*/

};

Moreover, we can define a day\_clock class that combines day and day\_point to give a clock that ticks once every day, analogous to clock in the Boost.chrono library. Thus, if we synchronize the clock at epoch at noon, we can be sure that between any two ticks of the clock the sunrises exactly one sunrise takes place i.e. each tick of clock corresponds to a new day. This is a modified form of Julian Day Number (± a constant duration due to changed epoch). All the functions like subtraction of two day\_points, comparison of two day\_points, addition /subtraction of days etc is supported automatically by the underlying Boost.chrono library.

But, Gregorian calendar being the standard calendar used, we have to wrap these basic JDN calendar API with conversion functions to and from Gregorian calendar. This is trivially possible as explained here [1].

**Public APIs** look like follows:

date today = boost::chrono::date::calendar.now();

//calendar is a wrapper of day\_clock above

std::cout << date::year() << " " << date::month() << " " << date::mday() << endl;

Since durations of months & years are not constant, we specify following durations:

typedef duration<int, std::ratio<2678400L>> month31;

typedef duration<int, std::ratio<2592000L>> month30;

typedef duration<int, std::ratio<2505600L>> month29;

typedef duration<int, std::ratio<2419200L>> month28;

typedef month31 jan, mar, may, jul, aug, oct, dec;

typedef month30 apr, jun, sep, nov;

typedef month28 feb\_normal;

typedef month29 feb\_leap;

typedef duration<int, std::ratio<31536000L>> year\_normal;

typedef duration<int, std::ratio<31622400L>> year\_leap;

On the basis of above constants, we define three functions change\_year(), change\_month() and change\_day(), for the purposes of incrementing and decrementing dates:

date today = calendar.now(); // calendar is based on day\_clock

date two\_months\_back = today.change\_month(-2);

date three\_years\_later = today.change\_year(3);

date yesterday = today.change\_day(-1);

// or

date yesterday\_2 = today – days(1);

On the latter point, we can also define two more approximate durations, months and year, defined as follows:

typedef duration<double, std::ratio<12622780800L>> eras;

typedef duration<double, std::ratio<2629746L>> months;

typedef duration<double, std::ratio<31556952L>> years;

Since these are approximate durations, we can opt not to show them to the public. But they can be very useful in quickly calculating the Gregorian calendar dates. Since the date class wraps day\_point, we can just provide + and – operator functions that wrap the underlying day\_point operation directly and handles the exceptional values too without showing it public. Outside it looks like:

date next\_month = today + months(1);

date century\_later = today + years(100);

Output of date is specified by to\_string():

std::cout << today; // YYYY-MM-DD, by default

std::cout << today.to\_string(); // YYYY-MM-DD, by default

std::cout << today.to\_string("%b %d, %Y"); //Mon DD, YYYY [modelled on strftime()]

std::cout << today.to\_string(iso); // YYYY-MM-DD

Here, one can replace iso with iso\_extended, iso\_week, iso\_year, ymd, mdy, dmy etc. (all enums). Since dates are preferred to be output in one consistent format, we store the output format in the structure itself as a private property, say output\_format. It defaults to iso but if to\_string() is accessed, this field changes to the second argument of to\_string().

Input of date is supplied via constructor:

date two\_apr\_2011 (2, 4, 2011, dmy);

date one\_nov\_2011 (2011, 11, 1, ymd);

date four\_feb\_2011 (2, 4, 2011, mdy);

The output\_format is stored according to the fourth parameter.

Week-based arithmetic would be based on following type:

typedef std::chrono::duration<float, std::ratio<604800L>> weeks;

We supply another function change\_weeks() to increment or decrement date in terms of weeks:

date last\_week = today.change\_weeks(-1);

A natural and elegant way to construct dates is specified in [2] and reproduced below. We wrap the three constructors of date above by a user-friendly syntax:

date d1 = date(2) / month(4) / year(2011);

date d2 = year(2011) / month(11) / day(1);

date d3 = month(2) / day(4) / year(2011);

These numbers because of explicit construction, get output\_format to be in the same order as order of units. The same is true for following similar constructions too.

The third unit is optional:

date d3\_2 = month(2) / day(4) / 2011;

We define enums of month type (Jan, Feb, …, Dec) and weekday type (Sun, Mon, …, Sat). Further we define five constants: \_1st, \_2nd, \_3rd, \_4th, \_5th, last.

Using them, we extend our previous notation to be more natural:

date d4 = day(4) / Feb / 2011;

//or

date d4\_2 = year(2011) / Feb / 4;

To specify the last day of the month (which varies month to month), we have following notation:

date d5 = last / Apr / 2011;

On the other hand, syntax like following (contrary to original proposal) shouldn’t be there:

date d\_err = \_2nd / Nov / 2011;

because it is just a substitute for day(2) / Nov / 2011, with no similar construct available for day(29) / Nov / 2011. This is inelegant as well as confusing because of asymmetric behaviour.

More interestingly, we have a construct like:

date d6 = Sun[\_4th] / Sep / 1993;

which refers to 4th Sunday of Sep, 1993. This syntax very smoothly defines a Mother’s day, for example, which is celebrated on second Sunday of May in most countries:

date mothers\_day\_this\_year = Sun[\_2nd] / May / 2013;

Similarly the fifth and last Monday of April falls on 29 April in 2013:

date 3\_days\_later = Mon[last] / April / 2013;

//or

date 3\_days\_after = Mon[\_5th] / April / 2013;

What if we try to construct Wed[\_5th] / April / 2013 which doesn’t actually exist? Any invalid date throws either a compile time error or, a runtime exception bad\_date.

Structures similar to day and day\_point are employed in Boost.date\_time. So, a separate header file can be provided that has functions that convert functions to and from the date structures in Boost.date\_time. It doesn’t have to be included by default. Similarly, conversion functions with time structures in <ctime> would also be included.

In separate header files, other calendars and date representations like ordinal date, week date (both specified in ISO 8601), Julian calendar etc can be provided (Note: these all calendars are solar).

# Proposed Milestones and Schedule

Here, weekly updates are provided regarding the development of the project. General idea is code for 40%, testing for 40% and documentation for 20% of given time.

|  |  |
| --- | --- |
| Start Date – Jun 17 | Get to know boost community and its coding & documentation guidelines, setup environment, interact with mentor and prepare rough guideline of the final date and calendar class. |
| Jun 17 – Jun 30 | An early prototype of date class and its basic input and output functions. |
| Jul 1 – Jul 14 | Make calendar class that generates date, analogous to chrono::system\_clock. |
| Jul 15 – Jul 21 | Implement date::change\_XX() functions and other functions. |
| Jul 22 – Jul 28 | Polish the library and docs to get a working prototype of date class. Submit it for mid-term evaluation. |
| Jul 29 – Aug 4 | Implement the JDN calendar using date class. |

|  |  |
| --- | --- |
| Aug 5 – Aug 11 | Implement ordinal date calendar using date class. |
| Aug 12 – Aug 18 | Implement week-year calendar using date class. |
| Aug 19 – Aug 25 | Implement Julian Calendar using date class. |
| Aug 26 – Sep 8 | Prepare for release, final bug fixing. Add examples to documentation. Performance testing. Final release to Boost community. |

# References

[1] http://quasar.as.utexas.edu/BillInfo/JulianDatesG.html

[2] http://home.roadrunner.com/~hinnant/bloomington/date.html