Building a Generic Date-Time Framework - An Experience Report using C++ Templates

Jeff Garland
President & CTO
CrystalClear Software, Inc
jeff@CrystalClearSoftware.com

Revised September 7, 2001

Abstract

This paper describes the experiences of the author using C++ templates to build the Generic Date Time Library (GDTL). While there are many date-time representations available for C++, the libraries are often unsuitable for domains that need high precision, long epochs, infinity, specialized calendars, or custom clock sources. The GDTL is an attempt to use generic and template programming techniques to provide a single library that meets all these demands, as well as more typical date-time programming. To build a single library, templates are used in several roles: allow user replacement of underlying date and time representations, to factor out calendar interfaces, building range and composite types, and providing interface constraint enforcement.

GDTL Background

I became interested in this several years ago while helping to implement date-time classes to support a satellite control system. The software performed extensive time calculations and had to manage details such as leap second tables. Some calculations, needed precision down to microseconds. Other parts of the system needed to store millions of date-times in a database (each day) with only second-level precision. So forcing microsecond resolution for all times unnecessarily bloated the size of the database.

In the end, the project had more than one time library. Some good components, but a host of project realities kept us from getting everything into a single library. In addition, traditional object-oriented design techniques did not directly support the required variations. Nevertheless, the question remained, could a single library support all the necessary variations avoiding the duplication of code? GDTL is, at least partially, an attempt to answer this question.

Overall Goals

The table below outlines the set of design goals for GDTL.

Category	Description	Functions
Interfaces	Provide concrete classes for manipulation of dates and times	• date, time, date_time, date_duration, time_duration, date_period, time_period, etc • support for infinity - positive infinity, negative infinity • iterators over time and date ranges
Calculation	Provide a basis for performing efficient time calculations	• days between dates • durations of times • durations of dates and times together
Representation Flexibility	Provide the maximum possible reusability and flexibility	• traits based customization of internal representations for size versus precision control • Allowing the use of different epochs and precision (eg: seconds versus microseconds, dates starting at the year 2000 versus dates starting in 1700) • Options for configuring unique calendar representations (Gregorian + others) • Allow for flexible adjustments including leap seconds
Clock Interfaces	Provide concrete classes for manipulation of time	• 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	• multi-lingual support • provide ISO8601 compliant time facet • use I/O facets for different local behavior

Table 1: Summary of GDTL Design Goals

Some Usage Examples

The following provides some simple examples of using GDTL date types. While GDTL supports time types, due to space constraints this paper will focus on date representations as examples.

```
//generate types based on the gregorian_calendar
using namespace gdtl;
typedef basic_date<gregorian_calendar> date;
typedef basic_date_duration<gregorian_calendar> date_duration;
typedef period<date,date_duration> date_period;
```

```
date d1(2000,1,1);
date_duration fourDays(4); //a number of days
date d2 = d1 + fourDays;
date_duration dd = d2 - d1;  //four days
date_period p1(d1,d2);  //a range of dates: Jan 1 - Jan 4
date_period p2(d1,date_duration(4)); //2001-Jan-01 - 2001-Jan-05
p2.contains(date(2001,1,1)); //true
p2.intersect(date_period(d1,d2)); //returns date_period
```

Library Status

GDTL is still an experimental library, but should be released as open source later this year.

GDTL Design Overview

GDTL uses 3 co-dependent types to enable time and date calculations: Points, durations, and periods (or intervals). Points (in time) represent a fixed position on the time continuum. For example, the date 2001-Jan-1 is a point with day level precision. The time 2001-Jan-1 15:14:00 is a point with second level precision. Durations represent an unanchored elapsed period of time (eg: 10 minutes, 22 seconds). Periods or intervals represent a fixed range of time starting at one point and ending at another.

Together, these elements provide the basis for complex calculation and reasoning about dates and times. This design is similar to those suggested in [1] and [2] for building time based systems.

Points In Time

To allow for efficient and accurate calculations, dates and times are represented as a single integer type. This representation is similar in concept to a Julian day number and allows for fast comparison and compact representation. To allow for direct calculations, the underlying representation must be monotonic. Adjustments such as leap years and leap seconds, do not enter directly into calculations, but are imposed as a view on the underlying representation. Adjustments are imposed on points in time when the component representation (eg: month, day, year) is extracted for printing or other purposes.

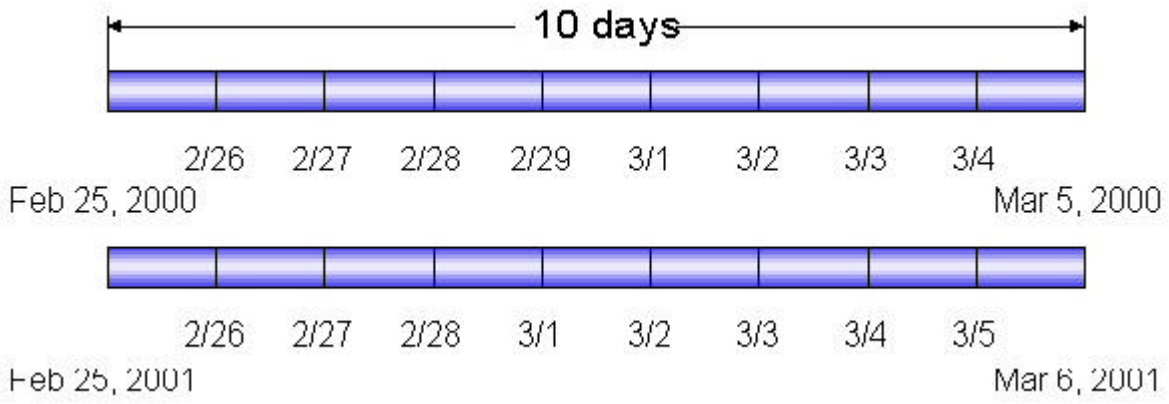

Figure 1: Date Adjustment Illustration

For example, in the illustration above a ten-day time duration starting from 2000-Feb-25 and 2001-Feb-25 end on 2000-Mar-5 and 2001-Mar-6 respectively. The placement of the extra day on the 29th day of February does not affect the internal calculations. In both cases, the elapsed time is 10 days.

For dates, a consistent feature across calendars is division of the timeline into years, months, and days. Different calendars, however, provide different handling of leap years and other adjustments. Thus, the goal of GDTL is to allow custom calendars to provide these adjustments and views by providing only a single component, the calendar (discussed later).

Time representations are like dates, but provide a higher resolution view of the time continuum. For times adjustments such as the time zone and leap seconds become relevant to providing the hour, minute, second view.

Durations

A duration is an elapsed time amount that is not anchored to a particular point on the time continuum. Durations can be added to points in time to get a new point or can be used with a point in time to create a time period. In addition, subtraction of two points in time produces a duration.

The following provides a synopsis of a `basic_date_duration`.

```
template<class calendar>
class basic_date_duration
  : boost::less_than_comparable<basic_date_duration<calendar>
  , boost::equality_comparable< basic_date_duration<calendar>
  > >
{
public:
  explicit basic_date_duration(unsigned long days);
  unsigned short days() const;
  unsigned long months() const;
  unsigned long years() const;
  bool operator==(const basic_date_duration& rhs) const
```

```
  bool operator<(const basic_date_duration& rhs) const;
  basic_date_duration operator-(const basic_date_duration& rhs);
  //...
private:
  typename calendar::duration_rep_type days_;
};
```

Periods/Intervals/Ranges

Many date-time calculations are greatly simplified by direct support for intervals and ranges. For example, questions such as:

- Is a date in a given year or month?
- What is the overlap of two time periods?

can be answered directly using a the period concept. A synopsis of period is shown below. Periods can be constructed from a point and a duration or from two points.

```
template<class point_rep, class duration_rep>
class period {
public:
  //! create a period from begin to last eg: [begin,last]
  period(point_rep begin, point_rep last);
  period(point_rep begin, duration_rep len);
  point_rep begin() const;
  point_rep end() const;
  point_rep last() const;
  //!various operators
  bool operator==(const period& rhs) const;
  bool operator<(const period& rhs) const;
  bool operator>(const period& rhs) const;
  //! True if the point is inside the period
  bool contains(const point_rep& point) const;
  //! True if this period fully contains (or equals) the other
period
  bool contains(const period& other) const;
  //! True if the periods overlap in any way
  bool intersects(const period& other) const;
  period intersection(const period& other) const;
private:
  point_rep begin_;
  point_rep last_;
};
```

Templates in the Design

Replacement of Representation

To meet several of the design goals basic classes such as date and time cannot be concrete. They must offer the ability to change basic assumptions such as the minimum date, or epoch, that a

date class can represent. For example, the `basic_date` class is a template that is customizable by providing a new calendar class. A simplified synopsis of `basic_date` is shown below.

```
template<class calendar>
class basic_date
  : boost::less_than_comparable<basic_date<calendar>
  , boost::equality_comparable< basic_date<calendar>
  > >
{
public:
  typedef calendar calendar_type;
  typedef basic_date_duration<calendar> duration_type;
  typedef typename calendar::ymd_type ymd_type;
  basic_date(typename calendar::year_type year,
          typename calendar::month_type month,
          typename calendar::day_type day);
  unsigned long year() const;
  unsigned short month() const;
  unsigned short day() const;
  bool operator<(const basic_date& rhs)  const;
  bool operator==(const basic_date& rhs) const;
  duration_type operator-(const basic_date& d) const;
  basic_date operator-(const duration_type& dd) const;
  basic_date operator+(const duration_type& dd);
private:
  basic_date(typename calendar::date_rep_type days) : days_(days)
{};
  typename calendar::date_rep_type days_;
};
```

The calendar template parameter provides the implementation type as well as customized calendar functions. The `basic_date` is responsible for implementation of calculation and comparison operators including interfacing with related types (period and duration). The calendar provides the mapping from external views (eg: yy/mm/dd) to the internal representation used in `basic_date`.

Calendar Classes

A major problem with most date-time representations is the presumption of the Gregorian calendar. GDTL breaks this assumption by providing the ability to customize `basic_date` with a different calendar class. The synopsis of the gregorian_calendar is shown below. Other calendars must provide similar interfaces to allow automatic generation of date and other related classes.

```
class gregorian_calendar {
public:
  typedef greg_month           month_type;
  typedef greg_day             day_type;
  typedef greg_year_month_day  ymd_type;
  typedef unsigned long        year_type;
  typedef unsigned long long   date_rep_type;
  typedef unsigned long long   duration_rep_type;
  static short day_of_week(const ymd_type& ymd);
```

```
    static unsigned long day_number(const ymd_type& ymd);
    static ymd_type from_day_number(unsigned long);
    static bool is_leap_year(year_type);
    static unsigned short end_of_month_day(year_type y, month_type
m);
    static unsigned short epoch_start_year();
  private:
  };
```

The calendar provides a basic set of conversion functions that allow `basic_date` to convert a point-based (integer) representation to a year, month, day and vice versa for formatting and other purposes. Overall, the calendar class can be thought of as a policy class [9] that describes the translation of the external date specification into the internal representation. In addition, the calendar class exports several types that enable the generation of compatible duration and period classes.

Precision

A fundamental issue for a time representation is the precision of representation. For example, an application calculating a spacecraft trajectory may require extremely high precision (microseconds) to accurately calculate orbit maneuvers. However, most applications don't require high resolution and may not wish to pay the storage and calculation costs associated with maintaining extremely high precisions. The resolution of a time representation is fundamental to its efficiency. Too large makes storage more expensive. However, too small does not allow sufficient resolution for applications.

With GDTL, an application could use the combination of a unlimited integer class to provide a large range and a high precision. Both `basic_date` and `basic_time` provide the ability to customize the internal representation. Typically an unsigned long makes a good representation. However, any type that supports addition, subtraction, and comparisons can be utilized.

Epochs

A point in time represents an absolute position in the infinite time dimension. Since no computer can represent an infinite dimension directly, a point must be anchored to a specific location in the time dimension. For dates this is called an epoch. Many C library functions provide the current date as the offset since 1/1/1900. In GDTL, the calendar provides the starting epoch. This is necessary because different calendar systems have different fixed starting points and apply different adjustments during different periods. In addition, this allows applications to adjust the minimum and maximum points to be represented.

Generation of Compatible Types

A key goal of GDTL is to provide systems of compatible date-time classes. For example, the basic_date class provides a core for the creation of specific date representations with different calendars and epochs. A specific date class also needs compatible date_iterators, date_periods, and date_durations. The generation of these compatible dependent types can be accomplished using templates.

The diagram below shows the instantiation of `basic_date`, `basic_date_duration`, and `period` to create a set of date classes based on the Gregorian calendar. The `gregorian_calendar` class provides interface types to `basic_date` for days, months, and years. In addition, the calendar provides compatible internal representations for both `basic_date` and `basic_date_duration`. The `basic_date_iterator` (described later) uses the arithmetic functions of the `basic_date` and `basic_date_duration` to provide various iteration capabilities over dates.

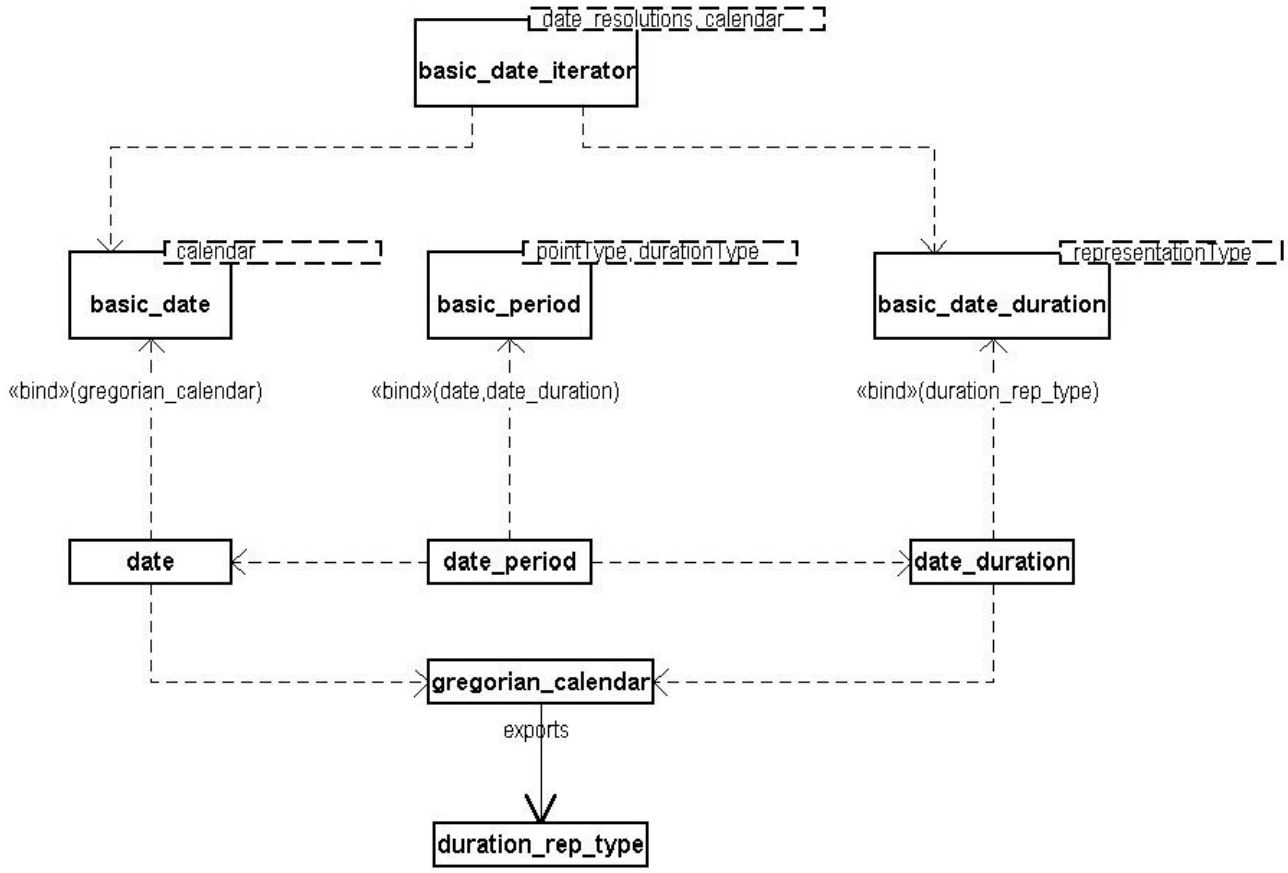

Figure 2: Generation of Related Types

Constraining the Interface - constrained_value

One fundamental problem of date-time classes is checking the validity of input data. Dates and times are frequently read from user interfaces, user edited files, and other error prone sources. Many, but not all, common mistakes can be detected and provide immediate feedback on the error. Consider the following interface:

```
class date {
  date(unsigned int year, unsigned int month, unsigned int day);
  //...
};
```

And imagine the following user code:

```
date d (12,31,2001); //oops year and day reversed
```

An unconstrained interface leaves open the potential of programming error propagating into the core of the date-time library. The GDTL design attempts to separate this error checking from the core of calculation, making error checking an independent aspect of the implementation. A helper template, constrained_value allows for rapid generation of classes to perform range checking. The synopsis of the constrained_value is as follows:

```
//!A constrained basic type similar to Range from Stroustrup [3]
template<typename rep_type, rep_type min_value,
        rep_type max_value, typename except_type>
class constrained_value {
public:
  typedef except_type exception_type;
  constrained_value(rep_type value) throw(exception_type);
  static rep_type max();
  static rep_type min();
  operator rep_type() const;
private:
  rep_type value_;
};
```

Constrained values check the min and max of the specified range. For example, the date interface can be re-written as follows:

```
typedef constrained_value<unsigned int, 1, 31, bad_day> day_rep;
typedef constrained_value<unsigned int, 1, 12, bad_month>
month_rep;
typedef constrained_value<unsigned int, 1, max_year, bad_year>
year_rep;
class checking_date {
  //Checking constructor
  checking_date(year_rep, month_rep, day_rep);
};
```

In this example, bad_day, bad_month, and bad_year are all exception types that are thrown if the user attempts to construct a value outside the specified range. This provides direct control over the interface and frees the internal calculation classes from checking these ranges. Of course, `constrained_value` cannot address all types of errors. A user could still switch the month and day, and as long as they are within the valid month range, the error will not be detected.

In Effective C++ Item 46 [7], Scott Meyers uses a date class as an example to discuss preference of compile-time versus runtime error checking. An enumeration is used to represent months in combination with static objects to ensure correct construction of months for dates written into source code. This technique can also be used in GDTL since the calendar author provides a month class implementation for `basic_date`.

While Meyers technique is targeted at managing compile time errors for dates written in the source code, the constrained_value approach is targeted at providing runtime error checking. That is, errors where the date attributes are not written into the source code, but are obtained from a user interface, file, or other runtime data source. In addition, constrained_value can go beyond Meyers technique since it can provide checking for the days of the month and the year as well as the month.

Of course, range checking is not the only type of checking needed to ensure correctness of a date. There are many calendar dependent checks that cannot be performed by a constrained_value. For example, 2001-Feb-29 is an invalid date even though the day specifier is in a valid range. This checking is performed by the calendar classes.

Customized Iterators

Another use of templates in the GDTL is in the construction of special iterators. The `date_iterator` class provides an iterator for stepping through a series of dates. The iterator is another example of a type dependent on characteristics of the date representation (such as the maximum allowed date) and hence is generated for each different date class.

In addition, `date_iterator` provides a mechanism to jump at a fixed offset. The following code, for example, will step through dates using a 7 day granularity.

```
 enum date_resolutions {day, week, months, year, decade, century,
NumDateResolutions};

//iterate by adding 7 days at a time
date_iterator<week, gregorian_calendar> ditr(date(2000,Jan,20));
for (ditr; ditr < date(2000,Feb,6); ++ditr) {
 std::cout << *ditr << " ";
 //...
}
```

The date_iterator is a template defined in terms of a `basic_date` and a `date_resolutions` enumeration. The enumeration is used to provide template specializations for the get_offset function that calculates the number of days to add to the current day.

```
template<date_resolutions resolution, class calendar>
 class date_iterator {
 public:
   typedef typename calendar::ymd_type ymd_type;
   typedef typename basic_date<calendar>::duration_type
duration_type;

   date_iterator(const basic_date<calendar>& start,
 unsigned int factor = 1);
   bool operator< (const basic_date<calendar>& d);
   bool operator<= (const basic_date<calendar>& d);
   date_iterator& operator++();
   date_iterator& operator++(int);
```

```
  basic_date<calendar> operator*();
private:
  unsigned int get_offset(const date_iterator& di) const;
  basic_date<calendar> current_;
  unsigned int factor_;
};
```

Use of boost::operators

Boost::operators provides a powerful template library to simplify the definition of operators. By using boost::operators the amount of code required to support the full complement of operators is reduced. As the documentation page [4] of the boost::operators describes, operators such as `operator>=`, can be defined in terms of `operator<`. Thus the library developer creates a few operators and then inherits from the set of desired operators. For example, the operators for the date class can add various comparisons as follows:

```
template<class calendar>
class basic_date
  : boost::less_than_comparable<basic_date<calendar>
  , boost::equality_comparable< basic_date<calendar>
  > >
{
```

Additional Opportunities for Templates

There are still several issues in GDTL design that might use templates in the solution. The following sections briefly discuss these.

Generic Interfaces

One interesting issue is the creation of a completely generic interface for the date and time classes. As the design stands, some date-time systems cannot be represented with the basic_date and basic_time. For example, in a Geologic date system the years, months, and days of the basic_date are not relevant. Rather, the unit of measurement might be a span of 100 years. Here relevant units of extraction would be centuries or millennia. Thus, currently, a template that replaces `basic_date` (eg: `basic_geologic_date`) would need to be written in addition to the `geologic_calendar` class.

One possible approach to this problem is for `basic_date` to provide additional interfaces (eg: century()) and use the "optional functionality through incomplete instantiation" technique described in [9]. Since `basic_date` uses the calendar to calculate the view, the calendar can provide an error message if a user attempts to use an interface that is not supported by that date system. So, for example, `basic_date::day()` would not be supported by a geologic calendar and produce an error if instantiated.

The drawback to this approach, however, is that it is difficult envision every view method a date system might need. In addition, it bloats the size of the `basic_date` making it harder to understand and maintain. Finally, each calendar must support a bigger interface.

Calendars As Templates

In the current versions of the library, the `gregorian_calendar` class is a concrete class. Unfortunately, to change the epoch, for example, this class must be re-written or changed. In a future version, this class will be templatized to allow for epoch selection within the Gregorian system without re-writing the calendar implementation.

Another policy aspect of the calendar that should be templatized is the error checking policy. The current implementation does not explicitly factor the checking. This is undesirable since some systems may desire to turn of all checking, preferring performance instead. Czarneecki and Eisenecker also describe this issue in [10].

Observations on Template Programming

Building a generic framework is extremely difficult. In many ways, it is similar to writing a design pattern that allows customization of the various tradeoffs by the user. Deep experience in the domain and plenty of experimentation with template programming are required.

Error messages are a major drawback of template-based approaches. Compilation errors involving a basic_date and other types create verbose error messages that typically don't contribute to the solution of user programming errors. For example, if a program uses dates in a standard collection, error messages can be difficult to decipher.

There are at least 3 approaches to this problem

- point users to Effective STL - Item 49 [8]
- instantiate types explicitly in the library
- enhance compilers to provide aliased error messages

One approach the library designer can use is to explicitly instantiate a type in the library instead of using a typedef. For example,

```
class date : public basic_date <gregorian_calendar>
{
     date(year_type y, month_type m, day_type d);
}
```

The main issue with this approach is that the constructors must be written for each class. Thus for a given date-time system this may entail writing at least a dozen constructors. However, this is likely the best solution for a common instantiation of GDTL since it simplifies error handling for library users.

A big improvement in compilers would be to provide aliased names in error messages. Fully expanded names could be provided as a detail section in the error. This would relieve both the library developer and users from the complexity of error messages in template-based solutions.

Documenting template-based frameworks is also difficult. The GDTL design covers a much broader design range than a conventional date time library. For the average user of a standard date class, documentation of the template customization points is not of interest. They are interested simply in how to use the provided classes. Developers that need to extend the library need a deeper understanding of how the library is designed. It is a struggle to design documentation which meets both needs. In addition, the amount of work is magnified since documentation of extension points effectively requires programming some extensions as examples.

Finally, testing and maintaining a template-based library is much more difficult than a typical library. The number of programmers with experience in template-based programming is relatively small. In addition, the range of application also makes changes more difficult. An extra dimension of 'compilation testing' may be required to ensure that extenders do not violate the fundamental library concepts.

Conclusions

Overall, templates have been instrumental in meeting most of the design goals for GDTL (see Table 1). They have enabled the factoring of many decisions that are normally fixed in date-time libraries. One major area of implementation remaining is support for the full range of input/output as described in I/O interfaces. This will expand the use of templates further due to the need to interface with C++ locales.

GDTL represents an example of template programming applied to the generation of concrete types. Other examples include template complex from the C++ standard library [3], a rational number template (boost rational) [5], and SIUnits [11]. This is an area of library development that is ripe for exploitation, because it facilitates more robust and flexible programs. In the case of GDTL, programs can be shielded from many complexities of the time-date domain. Most simple uses of dates and times can be accommodated with a standard template instantiation. However, template programming opens new avenues since projects that need type customizations are not required to rebuild an entire library, to work with high precisions or non Gregorian calendars.

Although template programming provides the tools to take software reuse to new levels, widespread application for project development will take time. The level of difficulty associated with building generic libraries means that only the largest projects will have justification to build such libraries. Even then, any given project will not have the time or inclination to build for future variability.

Acknowledgements

Thanks to Yannis Smaragdakis for suggesting the derivation-based approach for improving error handling. Thanks to Corwin Joy who has offered several suggestions and approaches for handling internal representations.

References

[1] Anderson, Francis, "A Collection of History Patterns", from Pattern Languages of Program Design, Addison-Wesley, 2000, pp 243-297.

[2] Carlson, Andy, "Temporal Patterns", from Pattern Languages of Program Design, Addison-Wesley, 2000, pp 241-262.

[3] Stroustrup, Bjarne, "C++ Programming Language 3rd Edition", Addison-Wesley, 1998, pp 236-242.

[4] Boost Operators - http://www.boost.org/libs/utility/operators.htm

[5] Boost Rational - http://www.boost.org/libs/rational/index.html

[6] Calendar FAQ - http://www.tondering.dk/claus/calendar.html

[7] Meyers, Scott, "Effective C++", Addison-Wesley, 1992, pp 175-178.

[8] Meyers, Scott, "Effective STL", Addison-Wesley, 2001, pp 210-217.

[9] Alexandescu, Andrei, "Modern C++ Design", Addison-Wesley, 2001, pp 13-26.

[10] Czarneecki, Krizysztof and Eisenecker, Ulrich, "Generative Programming: Methods, Tools, and Applications", Addison-Wesley, 2000, pp 217-218.

[11] Brown, Walter, "Introduction to the SI Library of Unit-Based Computation", http://fnalpubs.fnal.gov/archive/1998/conf/Conf-98-328.html, 2 September 1998.