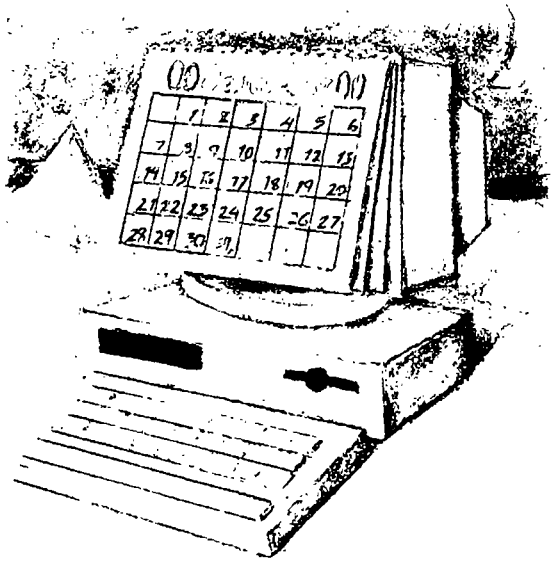


The Date Class

by Jeffreys Copeland and Haemer

It was late one Friday afternoon, the coffee pot was empty, and the sun was over the yardarm, so we had opened the first beer of the evening. We were standing in front of the blackboard arguing over a date class, when our colleague, Henry, came in and said that if taking our date class would improve his social life, he was certainly interested.

It's a little more complicated than that. We have been taking Carol Meier's introductory C++ course at the university. After years of doing object-oriented things, we thought it would be useful to have some formal background in C++.

In her first class, Carol gave a "by-the-way" exercise: Write a C++ class to handle dates. Because this is one of our favorite kind of problems, we spent much of the next month debating the appropriate data structures and interfaces to accomplish this. As a result, although we've written code in Perl for the last 14 or so columns, we're going to change direction and write the code for this month's column in C++.

Tricks of the Trade

There are three tricks that we use when we talk about object-oriented programming:

• **Data hiding**–Careful programmers have been using this trick for years, by writing dedicated routines to access data structures. C++ has language features to enforce this. Our rule of thumb has been to define the class method interface carefully enough so that if the underlying data structure changed entirely, the interface would remain constant.

Jeffrey Copeland (copeland@alumni.caltech.edu) *is a member of the technical staff at QMS's languages group, in Boulder, CO. His recent adventures include internationalizing a large sales and manufacturing system and providing software services to the administrators of the 1993 and 1994 Hugo awards. His research interests include internationalization, typesetting, cats and children.*

Jeffrey S. Haemer (jsh@canary.com) *is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting and internationalization. Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

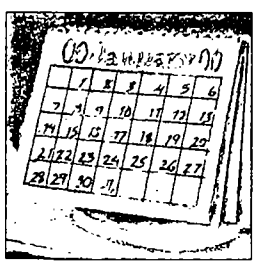
• **Polymorphism**–We support the overloading of functions so that, for example, with appropriate definitions, the plus operator can also handle concatenation of strings. This beats having the old FORTRAN standby of a different version of each function for each data type. For example, separate square root functions for floats, doubles and integers.

• **Inheritance**–Data structures can be defined to be derived from already existing data structures. The classic example is a data structure to handle mammals: It absorbs the data and methods of a simpler data structure to handle general animals.

We'll use all three tricks in this month's column.

Some Calendar Problems

Our history with calendar problems goes back several years, as longtime readers are well aware. We've used some methodology and algorithms from Nachum Dershowitz and Edward M. Reingold's paper "Calendrical Calculations," in *Software Practice & Experience*, Vol. 20, pp 899-928, September 1990.



When we consider dates, we typically need to calculate things such as the number of days between two dates, or the date some time before or after a given date. Dershowitz and Reingold make the observation–though they are not the first–that if we number the days from some arbitrary starting point and have convenient conversion routines to and from the day number, we can perform those calculations easily.

Dershowitz and Reingold use Monday, Jan. 1, in the year 1 A.D. as their starting point. Contrast this with a similar set of code we once wrote that used September 14, 1752 as day number one–computationally, a really bad choice. (Exercise for the reader: Why that date? Hint: Try `cal 9 1752`.) Alternately, astronomers do their calculations in Julian dates, beginning with Jan. 1, 4713 B.C. on the Julian calendar–offset 1,721,425 days from Dershowitz and Reingold's starting point.

Our debates centered around a basic problem. While there is a common Gregorian calendar that we use in civil discourse in the Western world, it's not the only one of interest. If we want to know when Passover occurs, we need to refer to the Hebrew calendar. Similarly, the Moslem holy month of Ramadan doesn't appear on Gregorian calendars. It is also useful to be able to do calculations in the older Julian calendar for historical purposes.

We can begin our task by defining a base `Date` class:

```
class Date {
public:
     void SetDate( long d ) { date = d; }
     friend long operator-(const Date& a,
          const Date& b);
     friend Date operator+(const Date& a,
          const long& b);
     void WeekdayOnOrBefore( int WeekDay ) {
     date -= (date - WeekDay) % 7;
     }
private:
     long date;
};
```

We've declared a single data item–the day number–and a handful of useful routines. Normally, we'd have done the work of `SetDate()` in the `Date` constructor, but we were being lazy about constructors in inherited classes. More important, we redefine the plus and minus operators to be useful in the date context–we can now calculate the difference between two dates, or a date some time after a given date without knowing the underlying data structure.

As an aside, we include a useful utility routine to tell us the weekday preceding a certain date. For example, this allows us to find the Monday on or before New Year's Day, that is, the beginning of the first work week of a new year.

The aforementioned operator redefinitions are pretty straightforward:

```
long operator-(const Date& a, const Date& b)
{
     return a.date - b.date;
}

Date operator+(const Date& a, const long& b)
{
     Date c;
     c.date = a.date + b;
     return c;
}
```

The Gregorian Calendar

Historically, it would make sense to declare the interface for the Julian calendar next. But it's the Gregorian calendar that we use most often, and with which we're

most familiar, so do that next instead.

The class header is quite a bit larger than the one for the base Date class:

```
class Gregorian : public Date {
public:
   Gregorian() {
   date = month = day = year = 0;
   }
   Gregorian(int month, int day, int year);
   Gregorian( Date d );
   void SetDate(int month, int day, int year);
   void SetDate(long d);
   int DayOfWeek() { return date % 7; }
      // ... because day 1 is Monday
   void print();
   void Easter(int year);
protected:
   int month, day, year;
   bool LeapYear( int year ) {
    if( (year%400) == 0 ) return true;
    if( (year%100) == 0 ) return false;
    if( (year%4)   == 0 ) return true;
    return false;
   }
   int DaysInMonth( int month, int year ) {
      int m[] = { 31,28,31,30,31,30,
            31,31,30,31,30,31 };
      if( month == 2 && LeapYear(year) )
            return 29;
      return m[month-1];
   }
};
```

Notice that we begin by declaring Gregorian to be a subclass of Date. In addition, we provide several alternate constructor definitions and some public methods that will be useful–including several alternate overloaded versions of SetDate(), to set the value of a Gregorian date.

To go with the DayOfWeek() method–and the WeekdayOnOrBefore() method of the base class–it's useful to have the following extra definitions:

```
const int Sun = 0;
const int Mon = Sun+1;
const int Tue = Mon+1;
const int Wed = Tue+1;
const int Thu = Wed+1;
const int Fri = Thu+1;
const int Sat = Fri+1;
```

Our hidden data and methods are also simple. We provide the obvious data of month, day and year–which we could skip, if we were willing to pay the computational overhead of constantly converting from the absolute day number in Date. We also provide private methods–with definitions in line–to tell us if it is a leap year and return the number of days in a given month.

Remember the correction that took us from Julius Caesar's calendar to Pope Gregory's? It's more accurate to make each century year a non-leap year, except when it's evenly divisible by 400. For example, 1992 and 2000 are leap years, but 1800 wasn't. We'll return to this correction later.

Let's start looking at the public methods for class Gregorian. We begin with the pair of overloaded constructors, which are brief. For example,

```
Gregorian::Gregorian( Date d )
{
     SetDate( d.date );
}

Gregorian::Gregorian(
     int month, int day, int year )
{

     SetDate( month, day, year);
}
```

Notice that in the first line, we're being a bit tricky. We're taking a Date as an argument, but passing long on to SetDate. More important, that first version of the constructor allows us to have fragments in our code like the following:

```
Gregorian today, tomorrow;
tomorrow = today + 1;
```

How can we do this? Our overloading of the + operator only applies to the Date class, so we fix this by providing a constructor to take a Date and convert it to a Gregorian.

Of course, we must follow these with the methods for SetDate(). Both versions of SetDate take the date in one form and generate it in the other–that is, given the absolute day number, we calculate month, day and year, and vice versa. The simpler version is the latter:

```
void
Gregorian::SetDate(
        int month_, int day_, int year_ )
{
     month   = month_;
     day     = day_;
     year    = year_;
     --year_;

     date = 0;
```

```
    // now calculate days before this year:
    // basic years
    date += year_ * 365;
    // leap year days
    date += year_ / 4;
    // century non-leap years
    date -= year_ / 100;
    // 400 years
    date += year_ / 400;
    // add the days before this month
    for( int i = 1; i < month_; i++ )
        date += DaysInMonth( i, year );
    // days in this month
    date += day_;
}
```

This method stores the given month, day and year, and calculates the days since our epoch of Jan. 1, 1. It is much more complicated to do that calculation in reverse. Dershowitz and Reingold provide, in a footnote, an exact calculation to do it, but the resulting code would be obscure.

It turns out that everyone–Henry Spencer's date routines, the Free Software Foundation versions of 'date' and 'strftime()', D. J. Delorie's port of the GNU C Compiler and its libraries to DOS–uses the same method as Dershowitz and Reingold. That is, they make a guess and then zero in on the correct date after the epoch.

Dershowitz and Reingold also provide an algorithm that makes a guess, and then approaches the calendar date from below. We thought that there had to be a better way, so we examined all the versions of the source code for the UNIX `date` command and `localtime()` and `strftime()` routines we had at hand. It turns out that everyone–Henry Spencer's date routines, the Free Software Foundation versions of `date` and `strftime()`, D.J. Delorie's port of the GNU C Compiler and its libraries to DOS–uses the same method as Dershowitz and Reingold. That is, they make a guess and then zero in on the correct date after the epoch.

So, we use that algorithm for the second overloaded version of `SetDate()`:

```
void
Gregorian::SetDate( long dd )
{
  date = dd;
  // we approximate m/d/y from below
  year = dd / 366;
  Gregorian guess(1,1,year);
  while( guess.date <= dd )
  {
  guess.SetDate(1,1,++year);
  }
  guess.SetDate(1,1,--year);

  // now approach the month
  month = 1;
  while( guess.date <= dd )
  {
        guess.SetDate(++month,1,year);
  }
  guess.SetDate(--month,1,year);

  // now get the difference for day of month
  day = (int) dd - guess.date + 1;
}
```

Notice that we use the month, day, year form of the constructor to do the internal calculation to ensure we haven't gone beyond the correct date.

We also need a routine to display dates. Ideally, we would include a routine that provided a parameterized format specifier, such as `strftime()`. Indeed, a version of `strftime()` for the `Date` base class would simply need to be overloaded with new month names to handle all four of the Gregorian, Julian, Hebrew and Moslem calendars.

By the way, we're ignoring the issue of overloading for different languages, so we can print July 4, 1776, or juillet 14, 1789, depending on the value of our LANG environment variable. But, we're trying to prove the concept at the moment; our `print()` code follows. Notice that we also print the absolute day number for debugging purposes.

```
void
Gregorian::print( )
{
     cout << month << "/" << day << "/" << year;
     cout << " (" << date << ")" << endl;
}
```

Exercise for the reader: Write a `strftime()` to replace the above code.

Finishing with an Example

We've now got enough base to write an example or two to show that our methods are marginally correct. Let's begin by convincing ourselves that the basic conversion code works. For example,

```
main()
```

```
{
    Gregorian zero(1,1,1);
    zero.print();
```

Next, we do a trick because we know what day number actually corresponds to a particular date. The two dates should be the same.

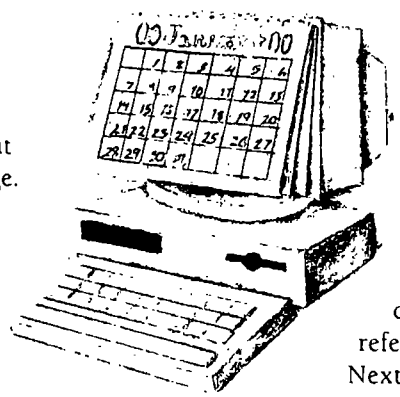
```
    Gregorian a;
    a.SetDate(719163);
    Gregorian b(1,1,1970);
    a.print(); b.print();
```

Finally, we amuse ourselves by figuring out when our son, JJ will be half his father's age. This gives us a chance to test the operator overloading for + and -.

```
    Gregorian jlc(6,5,1957);
    Gregorian jj(4,26,1990);
    long diff;
    diff = jj - jlc;
    jj = jj + diff;
    jj.print();
}
```

This provides us with the output we expected:

```
1/1/1 (1)
1/1/1970 (719163)
1/1/1970 (719163)
3/17/2023 (738596)
```



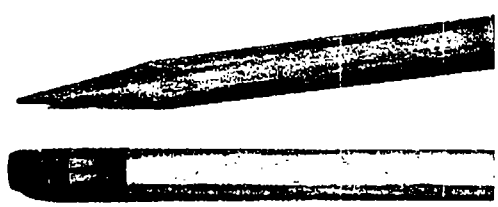
By way of further reading, we recently discovered that Dershowitz and Reingold followed up their 1990 paper with a survey of some other calendars. See Reingold, Dershowitz & Clamen, "Calendric Calculations, II: Three Historical Calendars," in *Software Practice & Experience*, Vol. 23, pp 383-404, April 1993.

Also, there is an interesting calendar calculation in Volume 1 of Donald Knuth's *The Art of Computer Programming*, Addison-Wesley, (2nd edition) 1973, ISBN 0-201-03809-9, see Exercise 14 of Section 1.3.2. We'll revisit that reference next time.

Next month, we'll present our version of `strftime()`. Then we'll finish off the `Gregorian` class with the `Easter` function, which will allow us to calculate the date of Mardi Gras. Also, we'll add the class for Julian dates as one derived from `Gregorian`. Space permitting, we'll look at a lunar calendar for contrast. Until then, happy dating. ▲

Reader Feedback

To help *RS/Magazine* serve you better, please take a few minutes to close the feedback loop by circling the appropriate numbers on the Reader Service Card located elsewhere in this magazine. Rate the following column and feature topics in this issue.



	INTEREST LEVEL		
Features:	High	Medium	Low
AIX Shines in the Front Office	170	171	172
SNA Takes the Express Way	173	174	175
CenterLine's QualityCenter Earns Mixed Marks	176	177	178
Columns:			
Q&AIX–Trade REXX for a Python	179	180	181
Systems Wrangler–Wonders of the WWW	182	183	184
Datagrams–The TLD Fiasco	185	186	187
AIXtensions–Web-Based Collaboration	188	189	190
Work–The Date Class	191	192	193