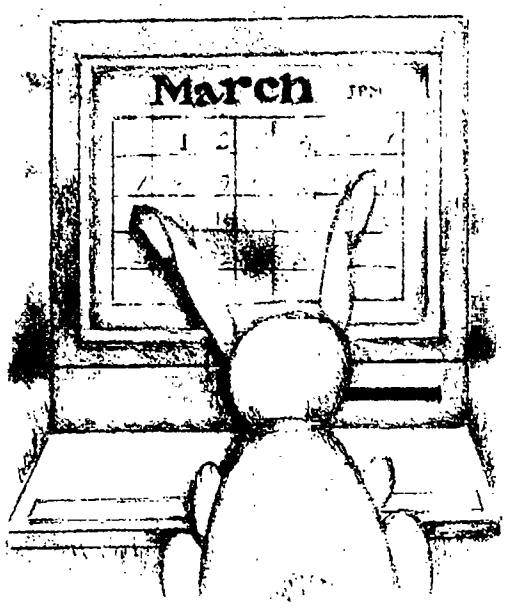

The Date Class, Part 2

by Jeffreys Copeland and Haemer

Last month, we started building a C++ class to handle dates. We explored the basics of building a class to handle a count of days from some arbitrary "dawn of time," and overlaying a derived class to deal with the Gregorian calendar.

We left you with the exercise of writing a `strftime()` function to print the dates out in a useful form, promising to begin this month's column with our solution to that problem. However, in the meantime, we discovered some interesting things about some common implementations of `strftime()` which we want to discuss in detail.

So we'll begin by looking at Easter on the Gregorian calendar. Then we'll look at the Julian calendar and finish with a discussion of the problems in getting a correct `strftime()`.

For reference, let's quickly review the classes we set up last time:

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

class Gregorian : public Date {
public:
    Gregorian(int month, int day, int year);
    Gregorian() { month = day = year = 0; }
    Gregorian( Date d );
    void SetDate(int month, int day, int year);
```

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.*

```
    void SetDate(long d);
    int DayOfWeek();
    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];
       }
};
```

Easter on the Gregorian Calendar

Easter, of course, is the important movable feast in the Christian calendar. Historically, it needed to occur in spring to correspond with older Pagan planting festivals, which it does by being celebrated on the Sunday following the first full moon after the spring equinox, which is assumed to always fall on March 21. If you've ever wondered why Easter appears to be celebrated near a full moon, now you know why. Passover also falls near a full moon–often the same one as Easter–but for different reasons.

Unfortunately, because of the error in the leap year rule for the Julian calendar, by the time of Pope Gregory XIII in the 1500s, the date of the spring equinox had drifted back 10 days. Remember that the Julian calendar has a leap year every four years, which makes its years a touch too long, while the Gregorian correction leaves three of every four century years as non-leap years, which pretty much smoothes out the difference.

It would have been sufficient simply to change the leap year rule beginning in 1582 and ignore those 10 days, but Pope Gregory XIII, in an effort to fight the Reformation, declared that in 1582, Thursday, October 4 would be followed by Friday, October 15.

The Catholic countries of Europe adopted the change immediately, but the Protestant countries dragged their feet. Great Britain–and its colony on the western fringe of the Atlantic–finally adopted the new calendar in 1752.

Russia waited until after the revolution, and, in fact, the Russian Orthodox church still operates on the Julian calendar. There's a good article about the history of the Gregorian calendar in the May 1982 issue of *Scientific American*.

While we're discussing the history of calendars, it is interesting to note that the Romans invented the fiscal year because the celebration of Saturnalia came close enough to the end of the calendar year to interfere with year-end reports. The workaround was to declare the year beginning in March.

Figure 1

```
void
Gregorian::Easter( int yr )
{
    int GoldenNr = yr % 19 + 1;
    int Century = yr / 100 + 1;
    int x = 3 * Century / 4 - 12;
    int z = (8 * Century + 5) / 25 - 5;
    int d = 5 * yr / 4 - x - 10;       // Sunday

       // Epact is a magic number to approximate
       // the full moon by roughly calculating
       // lunar cycles, from the 19 year (rough)
       // "golden cycle"
    int Epact = (11*GoldenNr+20+z-x) % 30;
    if (Epact == 25 && GoldenNr > 11) Epact++;
    if (Epact == 24) Epact++;

       // calculated full moon
    int N = 44 - Epact;
    if (N < 21) N += 30;

       // following Sunday
    N = N + 7 - (d+N) % 7;

       // Easter is N-1 days after March 1
    if (N > 31)
       this.SetDate(4, N-31, yr);
    else
       this.SetDate(3, N, yr);
}
```

With the historical context in mind, we can provide the Easter method for our `Gregorian` class shown last month (see Figure 1).

Remember that we're using Nachum Dershowitz and Edward M. Reingold's papers on calendric calculations–*Software Practice and Experience*, Vol. 20 (1990), pp 899-928, and Vol. 23 (1993), pp 383-404–as our primary reference.

For Easter, we also like to refer to Donald E. Knuth's *The Art of Computer Programming*, Vol. 1, Section 1.3.2. We'll shortly refer to his paper on Easter under the Julian calendar from *Communications of the ACM*, Vol. 5 (1962), pp 209-210. (Notice that the longer of the two implementations is in COBOL.)

When we know Easter's date, it's easy to calculate Mardi Gras, the all-important movable feast for those of you in New Orleans:

```
Gregorian z;
cout << "Easter 1997: ";
```

```
z.Easter(1997); z.print();
cout << "Mardi Gras: ";
z = z + (-47);  z.print();
```

Notice that we need to say `z=z+(-47);` rather than the more obvious `z=z-47;` because of the way we overloaded our plus and minus operators last time. In any case, the following code gives us the expected output:

```
Easter 1997: 3/30/1997 (729113)
Mardi Gras:  2/11/1997 (729066)
```

At this point, we've got enough grounding to explore the historical companion to the Gregorian calendar, the Julian calendar.

The Basics of the Julian Calendar

We've dealt with most of the basics of the Julian calendar already–the notion of leap year, the idea that each abstract date has a corresponding month, day and year, and methods to convert from the civil calendar to our abstract one. The difference between the Julian and Gregorian calendars is in their leap year rules. Therefore, it makes sense to make `Julian` a subclass of `Gregorian`, so we can inherit most of the methods we've already written. We declare `Julian` thus:

```
class Julian : public Gregorian {
public:
  Julian(int month, int day, int year);
  Julian(Date d);
  Julian() { month = day = year = 0; }
  void SetDate(int month_,
          int day_, int year_);
  void SetDate(long date);
  void Easter(int year);
private:
  bool LeapYear( int year ) {
    if( (year%4) == 0 ) return true;
    return false;
    }
};
```

We play similar games with the constructors for Julian dates as we did with those for Gregorian dates. Our constructors use service routines that can also be used as stand-alone routines.

```
Julian::Julian(int month, int day, int year)
{
  SetDate(month, day, year);
}

Julian::Julian( Date d )
{
  SetDate( d.date );
}
void
Julian::SetDate( int month_,
      int day_, int year_ )
{
  month = month_;
  day  = day_;
  year = year_;
  date = 0;
       --year_;
       // basic years
  date += year_ * 365;
       // leap years
  date += year_ / 4;
       // days until this month
  for( int i = 1; i < month_; i++ )
  date += DaysInMonth( i, year_ );
       // days in this month
  date += day_;
       // correction for base difference
  date -= 2;
}

void
Julian::SetDate( long dd )
{
  date = dd;
      // we approximate this from below
  year = dd / 366;
  Julian 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 the last line of the first `SetDate` method is a correction to align absolute day one on both the Gregorian and Julian calendars.

We can confirm the operation of this code with a simple test that checks the date of Pope Gregory's change:

```
Julian qj(10,5,1582);
Gregorian qg(10,15,1582);
cout << "Julian vs Gregorian" << endl;
qj.print(); qg.print();
```

If our code works, we should get the following output:

```
Julian vs Gregorian
10/5/1582 (577736)
10/15/1582 (577736)
```

This leaves us with the issue of Easter under the old calendar. Easter from 325 A.D. through the adoption of the Gregorian calendar was based on rules invented by the first Council of Nicaea. The calculation is a relatively pure implementation of the rule we stated earlier–the Sunday after the first full moon after March 21.

```
void
Julian::Easter( int year ) {
  Julian paschal_moon(4,19,year); // presumed
  int epact = ((11 * year%19) + 14) % 30;
      // corrected for calculated moon
  paschal_moon = paschal_moon + (-epact);
      // get the following Sunday
      // (note that this gives us the absolute
      // date, but not the Julian calendar day
      // so we've got a little work to do)
  Date following_sunday;
  following_sunday.date = paschal_moon.date+7;
  following_sunday.date =
      following_sunday.date -
          (following_sunday.date % 7);

  Julian easter(following_sunday);
  this->date = easter.date;
  this->month = easter.month;
  this->day = easter.day;
  this->year = easter.year;
}
```

We'll leave it to you to write the test fragment for this method. Suffice it to say that Easter in 325 fell on April 3.

The next step is to develop a version of `strftime()` so that we don't have to keep looking at this ugly debugging output. Our first thought was to adapt some existing version of the routine. Unfortunately, a funny thing happened on the way to doing that.

Problem of the Week

It turns out that this section heading is literally true, as we will explain. In the XPG and POSIX standards, we added the `%U` and `%W` specifiers to the `date` command, and to `strftime()`. In the case of `%U`, we should get the week number in the year, with the first day of week one being Sunday. Any day before the first Sunday is in week zero. `%W` is the same, but the week begins on Monday. In effect, this gives us the number of Sundays or Mondays from the beginning of the year.

"Yeah, so what?" you might ask. Well, `strftime()` and `date` now recognize the same set of data specifiers, so in general `date` is now implemented by passing all its arguments to `strftime()`. This means that a bug in `strftime()` will also make itself known in `date`.

We were going to use the `Date` class we're developing to print labels for our daily backup tapes. Those tapes are in two sets, one each for even and odd weeks. But the driving shell script for the backup kept resulting in different notions of which week it was and, hence, whether it was an even or odd week. We were very confused. We both got different results. Haemer's results were the same as Henry's, but Copeland's results were the same as Robin's.

Finally, we realized that the SunOS `strftime()` was giving different results for `%W` and `%U` from the version in the GNU shell utilities. Further investigation revealed that the Sun version was correct, and queries indicate that the version of the shell utilities about to be released by the Free Software Foundation fixes this problem (`http://www.gnu.ai.mit.edu/fsf/`).

REPRINTS

Keep up with the latest in RS technology with the best minds in the industry. Use reprints to *promote, inform,* and *sell.*

Reprint Management Services™ provides you the opportunity to obtain reprints of reviews, articles, and features in ***RS/Magazine.***

High-quality editorial reprints will help your company in many ways:

- Increased EXPOSURE for your product or service
- Credible, believable information that consumersTRUST
- Excellent SALES tool for trade shows, mailings and media kits
- Powerful EDUCATIONAL RESOURCE for consumers and employees

Reprints are completely customized to your needs. Call today for additional information!

JODY LISTER
Reprint Operations Specialist
REPRINT MANAGEMENT SERVICES™
147 West Airport Road
P.O. Box 5363
Lancaster, PA 17606
Phone: (717) 560-2001 Fax: (717) 560-2063

So, we thought, let's try to find a quick fix to the existing GNU version of `strftime()`, rebuild `date` and then everyone will be back in sync. Easier said than done, it seems. We began by looking at the version of `strftime()` in P.J. Plaugher's *Standard C Library*, Prentice-Hall, 1992, ISBN 0-13-131509-9. But that appears to have week numbers off by one in most cases. Some versions of the Berkeley BSD UNIX code were found to be wrong. In fact, overall we found more implementations that were wrong on this minor point than were correct. So, we did what comes naturally, we wrote a replacement.

From the observation above–that we were in effect counting the number of Sundays from the beginning of the year–and given a `tm` structure as defined in the standard `time.h` header file, we can provide a correct code fragment. We find the preceding Sunday, round up to the next week and return the number of days since January 1 divided by seven.

```
static int
sun_week (tm)
     struct tm *tm;
{
  int lastsun = tm->tm_yday -
     tm->tm_wday;
  return (lastsun+7)/7;
}
```

This is predicated on the assumption that `tm_wday` is zero for Sunday. The Monday case is similar but requires a bit of hand-waving because `tm_wday` for Monday is one.

```
static int
mon_week (tm)
     struct tm *tm;
{
  int lastmon = tm->tm_yday -
     ((tm->tm_wday+6)%7);
  return (lastmon+7)/7;
}
```

That doesn't solve the problem for our `Date` class, of course, because we don't have a `tm` structure. However, creating one from the data available shouldn't be very difficult. We'll leave that as an exercise for the reader, and take up next month with a `Date` method equivalent to `strftime()`.

Meanwhile, have a good October. ▲

In The Works

NOVEMBER DECEMBER 1997 JANUARY

SOFTWARE DEVELOPMENT
- Rapid Application Development Tools
- Multiprocessing and Multithreading

AIX AND THE INTERNET
Bonus Distribution:
DCI CLIENT/SERVER
Chicago, IL

NEW STORAGE TECHNOLOGIES FOR THE RS/6000
- UltraSCSI
- Serial Storage Architecture
- Cross Platform Solutions

Survey: RAID

To advertise call your area
RS/Magazine
Sales Representative listed elsewhere in this issue.

Visit Our Web Site
http://www.cpg.com