THE DIGITAL REPRESENTATION OF DATES

E. Henry Beitz
St. Paul, Minnesota 1971

Time is a critical element in most information systems. The real concern is not with time per se, but rather with the intervals between times. Sometimes these time intervals may be very small (say nanoseconds) but more often the concern is with intervals that span days, or even years. Days will be the focus in this article, because it is with these that the 'lumpiness' of the calendar creates problems. Smaller intervals are handled easily as decimal fractions of a day (or of a minute or second).

Once the cyclical nature of the seasons was noticed it was inevitable that calendars would evolve. Calendars have become a very important part of our lives. However, there are some contexts in which it would be preferable to think of days as a simple continuum. A system is needed which will permit one to refer to the calendar and at the same time represent time as a simple linear monotonic function. Mapping days onto an integer-set is an effective solution (each day being a successive integer). Numerous systems purport to do this, but few successfully achieve this goal.

What follows is a brief history of the Gregorian Calendar, a perspective on this calendar (that has some bearing on the routines that follow), and algorithms for encoding and decoding dates. The algorithms will be coded in a Pascal-like language that takes a few liberties.

THE GREGORIAN CALENDAR

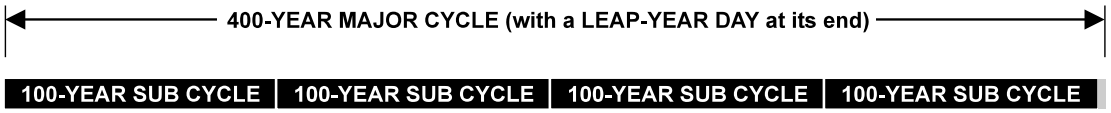
Consider how the Gregorian Calendar came to be. In 1545 Pope Paul III was authorized by the Council of Trent to amend the existing calendar. Pope Paul III died before he was able to solve the problem, and his successor Pope Gregory XIII, from whom the calendar was to take its name, inherited the task. Father Christopher Clavius, a Jesuit astronomer, performed the astronomical and mathematical calculations and Luigi Lilio, the Vatican librarian, who was also a physician and astronomer, provided a viable algorithm.

Devising a calendar that could account for the years lasting 365 days, 5 hours and nearly 48 minutes (or 365.242199 days) presented a challenge. Lilio's method was to have a 365-day year and to slip an extra day into 97 of the years in every 400-year period. Years containing this extra day would be known as leap years. The extra day would be inserted after the last day of February, and every fourth year would be a leap year unless it was also a century year (one that was a multiple of 100). One century year in every 400 years, the century year that was a multiple of 400, would also be a leap year. The additional 97 days would thereby be incorporated into each 400-year period. Though largely effective, this algorithm will make the calendar one day 'fast' in less than 4000 years.

Interesting extensions and alternatives have been suggested over the years, some of which would make every year identical.

THE GREGORIAN CALENDAR SEEN AS NESTED CYCLES

The Gregorian Calendar may be thought of as having a 400-year major cycle. Within this cycle assume a sub-cycle of 100 years, which is further divided into 4-year minor cycles. If one thinks of the major cycle as beginning on March 1 of each year that is a multiple of 400, then the major cycle may be visualized as four identical 100-year sub-cycles followed by a leap year day. Each of the 100-year sub-cycles is identical and consists of 25 minor cycles separated from one another by leap year days. Major cycles, sub cycles, and minor cycles begin on March 1 of their starting year and end on February 28, four hundred years, one hundred years, and four years later, respectively!



CHECKING A DATE

Because all numbers will represent valid dates, it is vital that the validity of a date be checked before encoding it. Two simple and useful routines are involved:

1) A Boolean function to determine if a given year is a leap year:

```
FUNCTION LeapYear (year: INTEGER) : BOOLEAN;
BEGIN IF (year MOD 400 = 0)
        OR (year MOD 100 <> 0 AND year MOD 4 = 0)
      THEN LeapYear := TRUE
      ELSE LeapYear := FALSE
END;
```

2) A routine for checking the validity of a date:

```
FUNCTION CheckDate (year, month, day:INTEGER):ErrorSet;
     (*The type errorset is an enumerated set of errors.*)
CONST month_type = (1, 3, 1, 2, 1, 2, 1, 1, 2, 1, 2, 1);
BEGIN CheckDate := [];                          (*no errors*)
      IF month < 1 OR month > 12
        THEN CheckDate := CheckDate + [month_error]
      ELSE IF day < 1
        THEN CheckDate := CheckDate + [day_error]
      ELSE
        CASE month_type[month] OF
        1:IF day > 31                        (*31 day months*)
          THEN CheckDate := CheckDate + [day_error];
        2:IF day > 30                        (*30 day months*)
          THEN CheckDate := CheckDate + [day_error];
        3:IF (day > 29) OR
           (day = 29 AND NOT LeapYear(year))    (*February*)
          THEN CheckDate := CheckDate + [day_error];
        END
END (*CheckDate*);
```

ENCODING A DATE

Certain adjustments need to be made prior to encoding a valid date. Consider extending the Gregorian Calendar back in time to the year 0, with March 1 of that year as day 0. The months will need to be renumbered, with March the first month, April the second, . . . , December the tenth, January the eleventh, and February the twelfth. Because of this renumbering we must also subtract one from the year for all dates in both January and February. The algorithm for both making these adjustments and encoding the date is:

```
PROCEDURE EncodeDate
     (year,
     month,
     day: INTEGER;
     VAR daycode: INTEGER);
     CONST month_start = (-1, 30, 60, 91, 121, 152,
              183, 213, 244, 274, 305, 336, 999);
BEGIN IF month > 2
        THEN month := month - 3
     ELSE BEGIN
       month := month + 9;
       year := year - 1
     END;
     daycode := year * 365
           (*days for normal years*)
     + (year DIV 100) * 24
           (*leap year days for centuries*)
     + (year DIV 400)
           (*leap year days for every 400 years*)
     + ((year MOD 100) DIV 4)
           (*leap year days for current century*)
     + month_start[month]
           (*days to beginning of current month*)
     + day;
END (*EncodeDate*);
```

DECODING THE DAYCODE

Decoding the daycode into a date is also relatively straight-forward. However, one must be able to recognize when a leap year day is involved. The algorithm that follows takes care of this:

```
PROCEDURE DecodeDate
     (VAR year,
          month,
          day: INTEGER;
          daycode: INTEGER);
     CONST month_start = (-1, 30, 60, 91, 121, 152,
               183, 213, 244, 274, 305, 336, 999);
VAR  rem:  INTEGER;
     indx: INTEGER;
BEGIN rem := daycode;
     year := (rem DIV 146097) * 400;
     rem  := rem MOD 146097;
     IF rem = 146096
     THEN BEGIN
       rem := 365;
       year := year + 399
     END
     ELSE BEGIN
       year := year + (rem DIV 36524) * 100;
       rem := rem MOD 36524;
       year := year + (rem DIV 1461) * 4;
       rem := rem MOD 1461;
       IF rem = 1460
       THEN BEGIN
         rem := 365;
         year := year + 3
       END
       ELSE BEGIN
         year := year + rem DIV 365;
         rem := rem MOD 365
       END
     END;
     indx := rem DIV 32;
            (*quick access into month_start*)
     WHILE rem > month_start[indx] DO
       indx := indx + 1;
       month := (indx + 14) MOD 12;
       CASE month OF
         0: month := 12;
         1..2: year := year + 1;
         OTHERWISE
     END;
     day := rem - month_start[indx - 1]
END (*DecodeDate*);
```

THE DAY OF THE WEEK

Fortuitously, the major cycle is a multiple of 7 days (146097 days to be precise). Because the date is encoded as an index-set, the residue modulo 7 of each daycode will always identify the day of the week. The March 1 at the beginning of each 400-year cycle always falls on a Wednesday! This means that any day can be made the first day of the week and a suitable table can be stored to provide ready access to the names of the days. The residue modulo 7 of the daycode will be called the day ordinal, and it will be assumed that the first day of the week (with day ordinal 0) is a Sunday.

THE STORED REPRESENTATION OF THE DAYCODE

The daycode should be stored as an integer. The size and precision of an integer in the system being used will determine the range of time covered by the daycode. (At the resolution of a day: two-byte integers will cover a period of nearly one hundred and eighty years, three-byte integers will cover more than forty thousand years and four-byte integers will suffice for about twelve million years.)

Bringing the desired period into the range of a specific integer representation will require some modifications: the encoding algorithm will have to remove a bias from the daycode as calculated, and the decoding algorithm will have to restore the bias prior to evaluating the date. The value of this bias will depend on the number of bytes available for an integer. The integers may be either signed or unsigned. If signed then the bias must map the start of the wanted period to the smallest negative value, and if unsigned then it must be mapped to zero.

The arithmetic will always have to be done with sufficient precision to avoid errors. To calculate the bias for a signed, two-byte representation: calculate the daycode for March 1 of the first year of the desired range (using the EncodeDate routine), determine the dayordinal for this daycode, and calculate the bias (bias = daycode + 32673 - dayordinal).

The last year covered by the resulting two-byte integers will be the starting year + 178. It will be necessary to check that the year being encoded lies within this range. The dayordinal may be calculated directly from the daycode. The residue modulo 7 should be calculated by subtracting the floor of the daycode divided by 7, from the daycode. This calculation is essential for signed integers and will work equally well with positive integers.

POSTSCRIPT

The number of days in any interval is the difference in the daycodes of the starting and ending dates. There is no need to decode the dates. The day of the week for any date is also easy to determine.

Astronomers have used a similar system for the past 400 years. Their Julian Period is 7980 years long. The last Julian Period began on January 1, 4713 B.C.E. (It was named for Julius Caesar Scaliger, whose son, Joseph Justus Scaliger, devised it, and shouldn't be confused with the Julian Calendar.)

A period much shorter than a Julian Period will generally suffice. A two-byte integer representation (covering about 178 years) is enough for most data processing applications. Where the need for a longer period arises a three or four-byte integer will surely do!