GREGORIAN CALENDAR

T. Hürlimann

Working Paper

May 1994
updated July 95

INSTITUT D'INFORMATIQUE, UNIVERSITE DE FRIBOURG

INSTITUT FÜR INFORMATIK DER UNIVERSITÄT FREIBURG

Institute of Informatics, University of Fribourg, site Regina Mundi

rue de Faucigny 2

CH-1700 Fribourg / Switzerland

Bitnet: HURLIMANN@CFRUNI51

phone: (41) 37 21 95 60 fax: 37 21 96 70

Gregorian Calendar

Tony Hürlimann, Dr. rer. pol.

Key-words: Gregorian Calendar, date conversions

Abstract: This paper presents a package to manipulate dates using the Gregorian calendar valid since about 1582 in western countries. The routines are useful in database applications. They are not useful, however, for accurate historical research purposes. The algorithms are implemented in Pascal. A clear representation of the algorithms was the main purpose. This is in contract to several shareware implementation (mostly in C) available through the Internet. It is also very easy to translate the package to C.

Stichworte: Der Gregorianische Kalender, Datumskonversionen

Zusammenfassung: Dieses Paper stellt ein Software-Packet zur Manipulation von Daten des Gregorianischen Kalenders vor, der seit etwa 1582 in der westlichen Welt gilt. Das Packet ist nützlich für Datenbankapplikationen, jedoch ungeeignet für historische Zwecke. Die Algorithmen sind in Pascal implementiert. Ein klare, transparente Darstellung und Implementation der Algorithmen war das Hauptziel dieser Arbeit. Diese stehen ganz im Gegensatz zu einigen über das Internet verfügbaren Shareware-Implementationen, die meist in C implementiert sind. Es ist zudem sehr einfach, das Packet auf C zu übersetzen.

INTRODUCTION

"This must be Thursday. I never could get the hang of Thursdays"

The mean tropical year (used for fixing the seasons) has 365.242199 days.

The **Julian calendar** was introduced in 45 BC (709 AUC[1]). It has two simple rule to calculate the number of days per year:

1 Each year normally has 365 days.

2 Beginning with 45 BC each forth year has 366 days (the leap years). (So 1 BC was a leap year and 4 AD too, since there was no year zero).

Thus the Julian calendar assumes a average year of exactly 365.25. That means that the Julian year is about 11 minutes 14 seconds too long, an error which grows to about a day in 133 years, which is about 3 days in 400 years.

The Julian calendar was in place all over Europe until the sixteenth century when Pope Gregory XIII authorized a revision of the calendar. At that date the Julian calendar had accumulated an error of about 10 days.

The **Gregorian calendar** reform contained the following modifications:

1 Centennial years should in future no longer be leap years unless they were divisible by 400 (i.e. 1600, 2000, 2400 are leap years, 1700, 1800, 1900 are not).

2 Thursday 4 October 1582 (Julian calendar) would followed by Friday 15 October 1582 (Gregorian calendar). The gap of 10 days corrected the accumulated error in the Julian calendar.

Thus the Gregorian calendar assumes a average year of exactly 365.2425. That is still 26 seconds too long. The error will grow to a day in about 3300 years. We can safely leave future calendar corrections to a far future generation!

Historically, calendar reforms were not that simple. In early Rome, March was the first month in the year until 153 BC. At that date, when the consuls started entering office on 1 January, it became the first day of the official year. But

[1] AUC means "ad orbe condita". The founding of the city of Rome was 753 BC.

such a change was only gradually adopted in the different parts of Europe. The situation in Italy in the 18th century was especially confusing:

– The Venetian new year started on the following 1 March

– The Pisan new year started on the preceding 25 March

– The Florentine new year started on the following 25 March

– Rome used various new years for different purposes

In England the new year changed gradually from the 25 December to the 25 March during the 14th century. Only beginning with 1753 the new year started with 1 January.

The Gregorian calendar also was not adopted throughout Europe. Various regional ad hoc modifications have taken place. For example: a Pisan year 1 June 1588 became 1 June 1587. By and large, the catholic countries adopted the Gregorian calendar reform, Protestant countries didn't. The most curious case was Sweden: They had the marvelous idea to gradually adapt to the Gregorian calendar by dropping all leap days, starting with 1700. This would have taken 40 years(!) to overbridge this 10 day gap. However, it did not work that way: 1704 and 1708 was a leap year. In 1712 it was decided to make an end of all this and to return to the Julian calendar – a splendid isolation decision! (Therefore Sweden got two leap days in 1712, they had a 30 February 1712!).

Many curious stories could be compiled in this context. The reason why this makes part of this paper is, that we must be very careful not to extrapolate our calendar to more they – say – 100 years. It would be meaningless, for example, to ask whether 12 February 1231 was a Saturday or a Sunday. We must keep this in the mind when calculating with our Gregorian calendar.

Besides of this, the Gregorian reform was adopted mainly – not to bring dates more in line with seasons throughout the year – but to get a simpler calculation for Easter (to bring Easter more closer to the 21nd of March)! Therefore, the Protestant and the orthodox were especially reluctant to adopt this reform.

HOW TO CALCULATE USING THE GREGORIAN CALENDAR?

To indicate a date we work with days, months, and years. This makes it

difficult to calculate with them: It is not so easy, for example, to get the number of days between two dates or to get the weekday of a date. But this is exactly what we use in various database applications.

An year has 365 days that is partitioned into 12 months. They have 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, and 31 days. A leap year has 366 days: the second month has 29 days instead. To be able to do calculations easy, we map each date into a positive integer – called the *Gregorian Integer* in the subsequent part of the paper – in the following way:

```
01/01/0001    1
02/01/0001    2
. . . .
01/02/0001    32
. . . .
01/01/1900    693596
. . . .
25/05/1994    728073
26/05/1994    728074
27/05/1994    728075
```

Hence, the Gregorian Integer of "25/05/1994" is 728073.
The algorithm to do this mapping is easy: suppose the date is given as DD/MM/YYYY (DD for days, MM for month and YYYY for year), then the mapping can be done by the following steps:

– multiply the number of years (YYYY) minus one with 365, giving y
– count all leap years from 0 to the preceding year of YYYY, giving L
– add all days up to the preceding month of MM, giving m
– add 1 to DD if YYYY is a leap year and if MM>2, giving d
– the solution is: y+L+m+d (this is the Gregorian Integer).

Example: 25/05/1994

– 1993*365 = 727445
– there are 483 leap years up to 1993 (in the Gregorian calendar)
– there are 120 days within the first four months
– since 1994 is not a leap year the number of days is just 25
– the Gregorian Integer is: 727445 + 483 + 120 + 25 = 728073

Once the Gregorian Integer has been calculated, it is easy to find the weekday of a date or to find the number of days between two date: just work with the Gregorian Integer using addition and subtraction and modulo 7 arithmetics! For the weekday, for example, we must just calibrate the number modulo seven. In this way we get a number from 0 to 6, which are interpreted as Sunday to Saturday.

```
Sunday    0
Monday    1
Tuesday   2
Wednesday 3
Thursday  4
Friday    5
Saturday  6
```

Since the 25/05/1994 is a Wednesday, we calculate 728073 modulo 7. The answer is 3. The calibration is already fine, since 3 is exactly the number we assigned for Wednesday. The 01/01/1900 was a Monday, since 693596 modulo 7 = 1.
It is also easy to calculate the difference between two dates: there were 34477 days between 01/01/1900 and 25/05/1994, since 728073–693596=34477.

But how do we find the inverse: given a Gregorian Integer what is its date? Suppose you want to know the date of 25/05/1994 plus 1000 days. How do we get this? The algorithm is a little bit harder, since the function is not continuous. The following is the algorithm given G as a Gregorian Integer:

- Make a first guess y for the year by dividing G by 365 (y=G div 365). Certainly, y is at least as big as the true year, since the leap years have not been taken into account. So, make a test whether G is smaller than 365*y plus the number of leap years up to y. If this is the case, y is replaced by y–1. The test is repeated and y is decrease as long as the test is true. At the end (when d is bigger than 365*y plus the number of leap years up to y), we know that y is the precedent of the true year.
- Subtract 365*y minus the number of leap years up to y from G and keep the result in G. G is now to number of days left in year y.
- Make a first guess m for the month by dividing G by 29 (m=G div 29). m is either the true month or one too small. (For G=1 to 28, i.e., the result is 0, for G=29 to 31 it is 1). It is one too small if G is bigger than the accumulated number of days of the preceding months. But we must be careful since the year might be a leap year in which case we must increase the accumulated days of the preceding months if the second month (February) was included completely. Suppose m is now the right month.
- The days within the month are now easy to find: subtract the accumulated number of days from the preceding months from G.

Example: Given the Gregorian Integer 728073.

- the first guess of the year is: 728073 div 365 = 1994
  calculate 365*1994+483 (the no of leap years till 1994) = 728293
  since 728073 <= 728293, the next guess is 1993.
  calculate again 365*1993+483 = 727928
  since now 728073 > 727928, we have reached the preceding of the true year so the true year is 1994.
- calculate 728073–365*1993-483=145 (that's the day left in 1994)
  the first guess of the month is 145 div 29 =5. Since 145 is smaller than 151 (the days in the first five month). 5 is the right month.
- subtract 120 (the days in the first four months) from 145. And we are left with the days (25).

IMPLEMENTATION

The kernel of the date procedures package is the translation between the Gregorian date written as DD/MM/YYYY and the Gregorian Integer and vice versa. The algorithms have been just described. The rest follows easy from this two procedures.

Three data structures are defined for the date:

- date as strings ('DD.MM.YYYY' or 'DD.MM.YY')
- date as structure containing integers for the day, the month, and the year
- date as Gregorian Integer

This is easily done in PASCAL as

```
type
  DateN = longint;              { Gregorian Integer }
  DateS = string[10];           { date as string: DD.MM.YYYY }
  DateR = record                { date as day/month/year structure }
    day, month, year: integer;
   end;
```

To store the accumulated days of a normal (not leap) year, a structured constant is used as following:

```
const MO_DAYS : TDAYS =
   ( 0 , 31 , 59 , 90 , 120 , 151 , 181 , 212 , 243 , 273 , 304 , 334 ,366);
```

This keeps all information of how the days are distributed between the months.

Two 'low level' functions are used to find out whether the year is leap or not and how many leap years have been accumulated up to a given year.
The first function is *IsLeap(y)*. It returns true, if y is a leap year otherwise it returns false. It is implemented as following:

```
function IsLeap (y: integer): boolean;
begin
 IsLeap := ((y mod 4 = 0) and (y mod 100 <> 0)) or (y mod 400 = 0);
end;
```

Every forth year is a leap year (y mod 4 = 0). But not if it is divisible by 100 (y mod 100 = 0) except all forth century (y mod 400=0).

The secomnd function is *LeapYears(y)*. It returns the number of leap years between the (hypothetical, means Gregorian) year 0001 and y (y included). This is almost the same as the preceding function. Every forth year is a leap year (gives y div 4), but not every 100th year (minus y div 100) but again every 400th (plus y div 400). Its implementation is straightforward:

```
function LeapYears (y: integer): integer;
begin
 LeapYears := y div 4 - y div 100 + y div 400;
end;
```

Two other (rather simple) functions simplify several tasks: the translation between the string date and the day/month/year data structure. They are called *DateS2R* and *DateR2S*.
The procedure *DateS2R* accepts a date string in the form 'DD.MM.YYYY' or 'DD.MM.YY' and returns a DateR-structure with the day, month, and year as numbers. The implementation is as following:

```
procedure DateS2R (s: DateS; var r: DateR);
 { string ('01.12.91')  to  (day/month/year)  translation }
 var  i: integer;
begin
 if length(s) = 8 then begin  {add century 1900 }
   s[10] := s[8];
   s[9] := s[7];
   s[7] := '1';
   s[8] := '9';
 end;
 r.day := 10 * (ord(s[1]) - 48) + ord(s[2]) - 48;
 r.month := 10 * (ord(s[4]) - 48) + ord(s[5]) - 48;
 r.year := 1000 * (ord(s[7]) - 48) + 100 * (ord(s[8]) - 48)
           + 10 * (ord(s[9]) - 48) + ord(s[10]) - 48;
end;
```

The inverse function *DateR2S* accepts a day/month/year data structure and returns a string date. Its implementation is also very easy:

```
function DateR2S (r: DateR): DateS;
```

```
 var  s: DateS;
begin
 s[0] := chr(10);
 s[2] := chr(r.day mod 10 + 48);
 s[1] := chr(r.day div 10 + 48);
 s[3] := '.';
 s[5] := chr(r.month mod 10 + 48);
 s[4] := chr(r.month div 10 + 48);
 s[6] := '.';
 s[10] := chr(r.year mod 10 + 48);
 s[9] := chr((r.year mod 100) div 10 + 48);
 s[7] := chr((r.year div 1000) + 48);
 s[8] := chr((r.year div 100) mod 10 + 48);
 DateR2S := s;
end;
```

Now it is time to present the 'high level' functions. The following functions are available in the *GregCal*-Package:

```
function Today: DateS;
function DateString (day, month, year: integer): DateS;
function RemoveCentury (s: DateS): DateS;
function WeekDay (s: DateS): integer;
function DateIsOk (s: DateS): boolean;
function DateS2N (s: DateS): DateN;
function DateN2S (d: DateN): DateS;
procedure WriteCalendarOfYear (year: integer; FileName: string);
```

Today returns the actual date as a date string. It is the only function that depends on the underlying operating system. Since every operating system has a corresponding function, it is easy to adapt it.

DateString accepts three parameters: day, month, and year and returns the date as a string. It is similar to the low level function *DateR2S*, which it calls. The implementation is

```
function DateString (day, month, year: integer): DateS;
 var  r: DateR;
begin
 if year < 100 then  year := year + 1900;
 r.day:=day;  r.month:=month;  r.year:=year;
 DateString := DateR2S(r);
end;
```

RemoveCentury just removes the century part of the year within a string-date: From 'DD.MM.YYYY' we get 'DD.MM.YY'.

```
function RemoveCentury (s: DateS): DateS;
 begin
  if length(s) = 10 then begin
    s[7] := s[9];
    s[8] := s[10];
    s[0] := chr(8);
   end;
  RemoveCentury := s;
```

```
end;
```

WeekDay accepts a string-date and returns a integer between 0 and 6 for the corresponding weekdays (Sunday till Saturday).

```
function WeekDay (s: DateS): integer;
 var  n: DateN;
begin
 n := DateS2N(s);
 WeekDay := n mod 7;
end;
```

DateIsOk returns true if the parameters is a legal date (entered as string) otherwise it returns false. Date with a year smaller than 1600 are considered as illegal.

```
function DateIsOk (s: DateS): boolean;
  var  c: boolean;  r: DateR;
begin
  DateS2R(s, r);
  c := (r.day >= 1) and (r.month >= 1) and (r.year >= 1600);
  if c then begin
    case r.month of
      1, 3, 5, 7, 8, 10, 12:   c := r.day <= 31;
      4, 6, 9, 11:   c := r.day <= 30;
      2:  c := (r.day <= 28) or (IsLeap(r.year) and (r.day <= 29));
    end;
  end;
  DateIsOk := c;
 end;
```

DateS2N translates a date as a string to a Gregorian Integer. Its algorithm was explained above.

```
function DateS2N (s: DateS): DateN;
  var  r: DateR;  c: integer;
 begin
  DateS2R(s, r);
  if IsLeap(r.year) and (r.month > 2) then  r.day := r.day + 1;
  DateS2N := 365*longint(r.year-1) + LeapYears(r.year-1) + r.day +
MO_DAYS[r.month-1];
 end;
```

DateN2S is the inverse translation. It translates a Gregorian Integer to a string-date. The algorithm was also explained in the last section. Its concrete implementation, however, gives rise to some subtle details which were generously ignored in the informal algorithm.

```
function DateN2S (d: DateN): DateS;
  var  r: DateR;  c: integer;
 begin
  r.year := d div 365; {better guess is: trunc(d/365.24);}
  while d <= 365*longint(r.year) + LeapYears(r.year) do  r.year := r.year-1;
  r.day := d - 365*r.year - LeapYears(r.year);
```

```
  r.year := r.year + 1;

  if IsLeap(r.year) and (r.day >= 60) then c := 1  else  c := 0;  { for 29.02 }

  r.month := r.day div 29;
  if r.day <= (MO_DAYS[r.month] + c) then  r.month := r.month - 1;
  if r.day = 60 then  c := 0;
  r.day := r.day - (MO_DAYS[r.month] + c);
  r.month := r.month + 1;
  DateN2S := DateR2S(r);
end;
```

The last procedure *WriteCalendarOfYear* combines all the tools in the package and produces a complete calendar for a given year. The calendar is written to a file. The procedure is easy to understand.

```
procedure WriteCalendarOfYear (year: integer; FileName: string);
  var  r: DateR;   s: DateS;
   n, m, i, j: DateN;
   month: integer;
   fil: text;
begin
  open(fil, FileName);
  rewrite(fil);

  r.day := 1;
  r.month := 1;
  r.year := year;
  s := DateR2S(r);
  n := DateS2N(s);
  if IsLeap(year) then  m := n+365  else   m := n + 364;
  month := 0;
  for i := n to m do begin
    s := DateN2S(i);
    DateS2R(s, r);
    if r.month <> month then begin
      month := month + 1;
      writeln(fil);
      writeln(fil);
      writeln(fil, month : 2, '/', year : 4);
      writeln(fil, '  mon  tue  wed  thu  fri  sat  sun');
      for j := 1 to 5 * (((i - 1) mod 7)) do  write(fil, ' ');
    end;
    write(fil, r.day : 5);
    if (i mod 7) = 0 then    writeln(fil); { WeekDay=0 (=sunday) }
  end;
  close(fil);
end;
```

In the Appendix the output of the package is shown: the calendar of the years 1994 and 2000.

A NEW TYPE “DATE” ?

The preceding procedure can be used to implement a data type DATE. Several operations would be useful:

<date> + <int> = <date> (add <int> days to first date)

<date> - <date> = <int> (difference of two dates in days)
<date> - <int> = <date> (subtract <int> days from a date)

<date> has the format dd.mm.yyyy, where dd are two digits for the day, mm are two digits for the month, and yyyy are four digits for the year (can be abreviated as yy for the 20th century).
Example: 24.06.95

The integer type could be extended as following, if {x} is sequence of digits then

{x}w : x number of weeks (example: 3w)
{x}m : x number of months (example: 15m)
{x}y : x number of years (example: 4y)

This integer extension makes only sense together with the date type. One could use the integer in the operations as above:

<date> + <int>m (add <int> months to <date>)
<date> - <int>w (subtract <int> weeks from <date>)
<date> + <int>y (add <int> years to <date>)

While a number extended by w (weeks) is just a number multiplied by 7 - therefore not very useful, the two other extensions are produces not operations having several drawbacks:
- They are not well defined:
Example: what is the result of 31.05.95 + 1m ? There is no 31.06.95! So the result should be 30.06.95. Fine! Now what is the result of 30.06.95 + 1m. Of course, one would expect 30.07.95. But then we get the result of 31.05.95 + 1m + 1m = 30.07.95, whereas we might expect the result 31.07.95 (as in the addition: 31.05.95 + 2m)! The same is true for: 29.02.96 + 1y.
To excape these difficulties, one could decide that all operations with days beyond 28 have undefined results. And one should not use them (like in floating point operations, certain results give inaccurate results or different results depending on how they are calculated). All operations with months and years can be done savely with days below 29. Do get the last days of each month in 1995, one could just subtract one from the first day in each month as following:

```
for i:=1 to 12 do
  EndOfMonth[i] := (01.01.95 + i*1m) - 1;
```

But there is another difficulty with such operations: How should 1m-1 be evaluated? Is it 30 or 29? At this point, one can see that the whole idea of extending the integer type (with y and m) breaks down, and the w extension is not needed. It is wiser to add functions to do the job:

```
AddMonths(d:date; NumberOfMonth:date):date;
AddYears(d:date; NumberOfYeras:int):date;
```

Other date functions are:

```
WeekDay(d:date):[0..6] (or: ['Sunday'..'Saturday']) ;
Date2Str(d:date):string;
Str2Date(s:string):date;
IsLeap(year:int):boolean;
Int2Date(day,month,year:int):date;
Date2Int(d:date; var day,month,year:int);  (procedure)
```

(Of course, 34.13.95 is an illegal date and should be refused by the parser, this means that the Str2Date and the Int2Date functions must also check for a legal date string.)

The approach using functions can be extended to the first three operation alltogether:

```
AddDays(d:date; NumberOfDays); (NumberOfDays can be negative)
DiffDate(d1,d2:date):int;
```

(AddWeeks can be easily be done by: AddDays(d,NumberOfDays*7)).

The conclusion to the section is the following: it is unwise to overload the + and the - operators to calculate with dates. A better idea is just to offer a set of functions doing the same job better.
This releaves the parser to recognize a new type. The date type can be implemented as string type. But now we need still another function:

```
IsDate(s:string):boolean;  (checks whether s is a legal date)
```

APPENDIX

1/1994

mon	tue	wed	thu	fri	sat	sun
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

2/1994

mon	tue	wed	thu	fri	sat	sun
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28						

3/1994

mon	tue	wed	thu	fri	sat	sun
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

4/1994

mon	tue	wed	thu	fri	sat	sun
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	

5/1994

mon	tue	wed	thu	fri	sat	sun
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

6/1994

mon	tue	wed	thu	fri	sat	sun
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30			

7/1994

mon	tue	wed	thu	fri	sat	sun
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

8/1994

mon	tue	wed	thu	fri	sat	sun
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

9/1994

mon	tue	wed	thu	fri	sat	sun
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30		

10/1994

mon	tue	wed	thu	fri	sat	sun
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

11/1994

mon	tue	wed	thu	fri	sat	sun
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30				

12/1994

mon	tue	wed	thu	fri	sat	sun
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

1/2000

mon	tue	wed	thu	fri	sat	sun
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

2/2000

mon	tue	wed	thu	fri	sat	sun
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29					

3/2000

mon	tue	wed	thu	fri	sat	sun
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

4/2000

mon	tue	wed	thu	fri	sat	sun
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

5/2000

mon	tue	wed	thu	fri	sat	sun
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

6/2000

mon	tue	wed	thu	fri	sat	sun
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30		

7/2000

mon	tue	wed	thu	fri	sat	sun
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

8/2000

mon	tue	wed	thu	fri	sat	sun
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

9/2000

mon	tue	wed	thu	fri	sat	sun
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	

10/2000

mon	tue	wed	thu	fri	sat	sun
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

11/2000

mon	tue	wed	thu	fri	sat	sun
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30			

12/2000

mon	tue	wed	thu	fri	sat	sun
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

REFERENCES

Nachum Dershowitz and Edward M. Reingold. Calendrical Calculations, Software-Practice and Experience, Vol. 20, number 9 (September 1990), pp 899-928.

Van Zandt Jim, A date package, from the Internet.

Emmons Robert, [1992], Julian Date Library, Shareware from the Internet (not free).

All cited historical facts and much more can be found in *comp.lang.c* through the Internet.