October, 2002

Advisor Answers

Computing Exact Age

VFP 7/6/5/3

Q: Has anyone developed an Age() function that also returns partial years as part of the numeric return value (e.g., 3.5479 is 3 years and 200 days between two dates [rounded])? I have seen many functions over the years but they only return the integer portion of the age.

In some of my programs, I need to sort by age where the sorting is done by both year and days. Converting the days to a decimal is good enough; it just has to be accurate. Having a single function also saves calling two different functions (i.e., one to return the year portion and the other to return the days portion).

I have developed a function myself because I could not find another. But it is fairly complex. It figures out the days between two dates, adjusts for the leap years between the two dates and then divides by 365 to get the result. It works and is accurate but for every year between the two dates, I have to test for a leap year and that slows it down. For example, June 1, 2001 to June 1, 2004 would normally be 3 years (1095 days) but there is a leap year so {06/01/2004} - {06/01/2001} (SET DATE TO AMERICAN) is 1096 but if I loop through and find that 2004 is a leap year, I subtract off 1 to get 1095 which divided by 365 gives 3.0. It is probably just one way to get the result but it is indeed accurate. If I could find something faster, I would switch!

–Albert Gostick (Guelph, Ontario, Canada)

A: Computing age is a problem that comes up pretty frequently in FoxPro applications. In fact, I first published a solution for it in this column in the February '95 issue. That answer showed how to compute the age in years or years and months, and was a demonstration of the power of FoxPro's date math capabilities.

However, your problem is much more challenging in some ways, including the concern about performance. Let me start with easy solutions. If your goal is simply to be able to sort by age, you can use the birth date itself. Indexing on dates is permitted-use a command like:

```
INDEX ON dBirthDate TAG dBirthDate
```

If you're doing some process where you don't want to carry the birth date along, you could also use just the number of days. So, in a query, you might do something like:

```
SELECT <the other data>, DATE()-dBirthDate AS nDays …
```

then order the results on nDays. (Remember that subtracting one date from another gives the number of days between them.)

However, if you really want a value that represents the age in a single number, you need to do some computations. The hardest problem is the difference between being born on February 28 and being born on February 29. In our system of computing age, there's no difference between those two, except in leap years. (In FoxPro terms, GOMONTH({^1960/2/29}, 12) = GOMONTH({^1960/2/28}, 12).) From your question, though, I think you want to distinguish those two cases. By doing so, you're really mixing two different numbering systems, age and calendar. Making the two coordinate is tricky.

The first approach I tried is the one you outlined in your question. Compute the number of days, then adjust for any leap days and divide by 365. However, there's no need to test every year for a leap year. We know that leap years come no more often than every four years. So, we can find the first and last leap year in the relevant period and compute the number of leap years. This algorithm is implemented in AgeByDays.PRG on this month's Professional Resource CD.

So what's wrong with this approach? It returns the same value for February 28 and February 29 of any leap year.

The next technique I tried was an extension of the code from the 1995 column. Compute the number of years, then figure out the number of days left over, and divide that by 365 (or 366, if the birth date is between the specified day in the year before a leap year and February 28 of a leap year). Here's the code (Age.PRG on this month's PRD):

```
* Age.PRG
* Compute the age of a person on a given date
LPARAMETERS tdBirthDate, tdSpecifiedDate

* Parameter checking omitted for space.

LOCAL nAdjustment, nBirthYear, nBirthMonth, nBirthDay
LOCAL nSpecYear, nSpecMonth, nSpecDay, nAge
LOCAL dAdjustedDate, nRemainingDays, nDivisor, lAdjust
```

```
* Break dates into components
nBirthYear = YEAR(tdBirthDate)
nBirthMonth = MONTH(tdBirthDate)
nBirthDay = DAY(tdBirthDate)

nSpecYear = YEAR(tdSpecifiedDate)
nSpecMonth = MONTH(tdSpecifiedDate)
nSpecDay = DAY(tdSpecifiedDate)

* Has birthday passed yet this year?
DO CASE
CASE nBirthMonth < nSpecMonth
   nAdjustment = 0
CASE nBirthMonth  > nSpecMonth
   nAdjustment = -1
CASE nBirthMonth = nSpecMonth
   IF nBirthDay <= nSpecDay
      nAdjustment = 0
   ELSE
      nAdjustment = -1
   ENDIF
ENDCASE

* Compute number of years
nYears = nSpecYear - nBirthYear + nAdjustment

* Move to last birthday
dAdjustedDate = GOMONTH(tdBirthDate, 12 * nYears)

* Compute remaining days
nRemainingDays = tdSpecifiedDate-dAdjustedDate

* Do we need to adjust for a leap day?
lAdjust = (IsLeap(nBirthYear) AND ;
           tdBirthDate <= DATE(nBirthYear, 2, 28)) OR ;
          (IsLeap(nBirthYear + 1) AND ;
           tdBirthDate > ;
             DATE(nBirthYear, nSpecMonth, nSpecDay))
IF lAdjust
   nRemainingDays = nRemainingDays + 1
   nDivisor = 366
ELSE
   nDivisor = 365
ENDIF

nAge = nYears + nRemainingDays/nDivisor

RETURN m.nAge
```

The good news is that this version works in all cases. However, in testing, this version turned out to be about a third slower than the first version. So, I continued looking for another solution.

Rather than trying to remove the leap days, how about dealing with them in the division? The next version (AgeByDivision.PRG on the PRD) computes the number of days and divides by 365.2425, the number of days per year in the Gregorian calendar. It keeps all the dates unique, so always provides an accurate sort. It's also significantly faster than the other two versions. (In my tests, it was more than twice as fast as the first version.)

However, it suffers from a different problem. If you check someone's age on their birthday, you don't get a round number back. That is, if I call the function as follows:

```
? AgeByDivision( {^1958/9/28}, {^2002/9/28})
```

the value returned is 44.0009, not 44. (In some cases, the result is less than the actual age, rather than more.) That may be good enough for your purposes.

Finally, I decided to try a different approach. Rather than making the decimal portion of the number a fraction of the year, how about just returning the number of days remaining as the fraction. That is, if the specified date is one day after the birthday, the decimal portion would .001. If it's the day before the birthday, return .364 (or, if there's a leap year involved, .365). This version (AgeWithDays.PRG on the PRD) looks much like Age.PRG, but differs in the final calculations. Here's the tail end of the code:

```
* Adjust for leap year?
IF (IsLeap(nBirthYear) AND ;
    tdBirthDate <= DATE(nBirthYear, 2, 28)) OR ;
   (IsLeap(nBirthYear + 1) AND ;
    tdBirthDate > ;
    DATE(nBirthYear, nSpecMonth, nSpecDay))
   nRemainingDays = nRemainingDays + 1
ENDIF

nAge = nYears + nRemainingDays/1000

RETURN m.nAge
```

My tests show this version to be slightly faster than Age.PRG, though still slower than the less accurate versions.

Since the two accurate versions rely on it, let's look at determining whether a year is a leap year. This being FoxPro, there's more than one way. I tested two techniques and found one to be about a third faster than the other. The faster version uses the MOD() function to check the rules for leap years. (A year is a leap year if it's divisible by

4, unless it's divisible by 100 and not by 400.) Here's the code for that version, included on this month's PRD as IsLeap.PRG:

```
* Check whether a specified year is a leap year
LPARAMETERS nYear

LOCAL lIsLeap

lIsLeap = MOD(nYear,4) = 0 and ;
         (MOD(nYear, 100) <> 0 or MOD(nYear, 400) = 0)

RETURN lIsLeap
```

The other version (IsLeap1.PRG on this month's PRD) uses a FoxPro trick; it tries to create a date variable of February 29 of the specified year. If it's successful, the year is a leap year; if the result is empty, it's not.

In addition to the functions mentioned here, this month's PRD includes the programs I used to test the speed of the different age computation functions (TestAge.PRG) and the speed of the leap year computations (TestLeap.PRG).

–Tamar