# Date Stamp

## *Efficient date calculations for business applications*


by Julian Bucknall

I was checking up on my vacation entitlement the other day (my wife, Donna, had just passed the Bar examination for Colorado, and I needed to book as holiday the day of her swearing-in as a *real* lawyer) when I noticed something bizarre. I joined TurboPower Software on 1 May 1993 and my holiday year therefore runs from 1 May of one year to 30 April of the next. Seems easy enough, yet our internal company website seemed to think that my current holiday year ran out on 6 May 1999. Huh? It turns out that the page is generated automatically from an Excel spreadsheet and the formula was subtly wrong.

A couple of months before that someone was asking on one of our newsgroups about calculating the number of business days between two dates. *Not* the number of *days* between two dates, mind you (a common enough request on the Delphi newsgroups), but the number of *business* days. And then, just slightly before that, someone else wanted some other business date arithmetic.

Enough already. I can spot a trend if it bites me on the leg often enough. Especially if I can convert it into an *Algorithms Alfresco* article. Hence this month's column on efficient date algorithms and calculations.

What we shall do in this article is devise a compact date representation with which we can do *efficient* date arithmetic. The date representation we'll design will be optimized for business use. We'll then explore adding days and months to a date, and fiddling around with business date arithmetic.

### Poison Arrow

So, first question first. Why have another date type? Isn't `TDateTime` good enough? Frankly, no. Mind you, it *does* have one redeeming advantage: it's a standard for Delphi (although it has a different definition in Delphi 1 compared to the later Delphi compilers). But, oh, the other problems... It's a floating point type for a start, with the fractional part being the time (basically half the 8-byte storage size is unused for doing arithmetic with dates: you could make do with a `longint` quite easily). Because it is a floating point type and you want to do what amounts to integer arithmetic, you have to worry about whether you call `Trunc` or `Round` to convert to an integer. Also, although the conversions between a `TDateTime` variable and day-month-year representation work, they're hardly the model of efficiency. When you do a *lot* of date arithmetic (and in my previous job on the swaps trading desk of a large German bank in London, there was a *lot*) you need efficiency in converting to day-month-year and back again. One of the reasons for this inefficiency is the sheer range of possible dates that `TDateTime` covers, the vast majority of which we'll never encounter or use (unless our PCs survive us by thousands of years). And there's one mighty kicker for people who use both 16- and 32-bit Delphi: the Delphi 1 `TDateTime` is *not* the same as the `TDateTime` for later versions of Delphi (for Delphi 1, day 1 is 1 January 0001, but for the other Delphis, it's 31 December 1899). If you write out a `TDateTime` value to a file in Delphi 1 and read it in Delphi 2, you won't get the same date.

So, in this article we shall invent a better mousetrap. Of course, we'll provide a way to get from our mousetrap to Delphi's. And you can bet your bottom dollar we shall be making sure this mousetrap is Y2K-compliant.

Having decided that it would be a good thing to have a new date format, we need a little background exposition. We shall be dealing with Gregorian dates, that is, dates from the Western Gregorian calendar (in particular, I'm referring to the rule for leap years). We shall be building a set of routines for general present-day business use, in particular, over a range of just 400 years, from 1800 to 2200. That should be easily sufficient for modern business applications. We certainly won't be dealing with the problems caused by the crossover from the Julian calendar to the Gregorian calendar (for an excellent discussion of such date calculations, see *Delphi For Time Travellers* by Brandon Smith in Issue 34, June 1998). I also won't be talking about what a leap year is or why they're needed, I shall assume that you all know... Anyway, I couldn't do better than Brandon's article.

Just so that you can see where we're going with this article, we'll be defining the new type, converting from year-month-day to it, and *vice versa*, calculating the day of the week for a date, finding the number of days and months between two dates, adding days and months to a date, looking at how to calculate business dates, calculating Easter Day for a given year, and finally investigating the wacky world of bond dates. Whew! With all that we'd better get started.

### All Of My Heart

Right then, definition of a date. The dates we'll be talking about in this article are limited to the range from 1 January 1800 to 31 December 2199. 400 years (97 of them leap), or 146,097 days. This latter

```
function aaDaysInMonth(Y, M : integer) : integer;
begin
  if (Y < MinYear) or (Y > MaxYear) or
     (M < 1) or (M > 12) then
     raise Exception.Create(
       'aaDaysInMonth: invalid year and/or month');
  if (M = 2) and IsLeapYearPrim(Y) then
    Result := DaysInLeapFeb
  else
    Result := DaysInMonth[M];
end;
function aaIsLeapYear(Y : integer) : boolean;
begin
  if (Y < MinYear) or (Y > MaxYear) then
    raise Exception.Create(
      'aaIsLeapYear: invalid year, should be 1800-2199');
  Result := ((Y mod 4) = 0) and
    (Y <> 1800) and (Y <> 1900) and (Y <> 2100);
end;
function aaIsValidYMD(Y, M, D : integer) : boolean;
begin
  Result := false;
  {easy checks}
  if (Y < MinYear) or (Y > MaxYear) then
    Exit;
  if (M < 1) or (M > 12) then
    Exit;
  if (D < 1) then
    Exit;
  {full check on day}
  if (D > 28) then begin
    {if February..}
    if (M = 2) then begin
      {if leap year..}
      if ((Y mod 4) = 0) and (Y <> 1800) and (Y <> 1900) and
        (Y <> 2100) then begin
        if (D > DaysInLeapFeb) then
          Exit;
      end else if (D > DaysInFeb) then
        Exit;
    end else if (D > DaysInMonth[M]) then
      Exit;
  end;
  {otherwise it's OK}
  Result := true;
end;
```

➤ *Listing 1: Building-block date routines.*

value tells us immediately that representing a date as the number of days from 1 January 1800 we'll need at least a `longint` to store the date, since we'd require at least 18 bits. So, our date type, `TaaDate`, is a `longint`, with possible values from 0 (which is 1 January 1800) to 146,096 (which is 31 December 2199). Values outside this range are deemed invalid and, if encountered, will cause an exception to be raised. Note that since we're holding date values as the number of days from a base date, we haven't had any Y2K problems yet.

Well that was easy! What's next? We shall need a couple of utility routines before going too much further. The first one will return whether a year value is a leap year or not, where the year value is something between 1800 and 2199 (anything else causes an exception to be raised). Again, I'm avoiding any Y2K problem: no year values between 0 and 99 are allowed in this routine. The leap year rule is usually stated as 'a year is a leap year if it is divisible by 4, but not if it is divisible by 100, unless it is also divisible by 400'. Well, phooey, in our routine a leap year is one that is divisible by 4, unless it happens to be 1800, 1900 or 2100. Simple, eh? Listing 1 has the `aaIsLeapYear` routine.

The next handy routine is one that returns the number of days in a given month. We all know the rule here, indeed schoolchildren are taught a little mnemonic rhyme to help them remember. We'll pass in the year (1800 to 2199) and the month (1 to 12) and `aaDaysInMonth` will return the number of days in that particular month. Internally, we'll have to make use of the `aaIsLeapYear` function for February, otherwise we grab the value out of an internal `const` array. It's Listing 1 again for the details.

The final simple routine is one to validate a year-month-day triplet to be a real date. The `aaIsValidYMD` function uses both the above routines to do its stuff (essentially we check that the year is between 1800 and 2199, the month is between 1 and 12, and the day is between 1 and the number of days in that particular month). Again see Listing 1.

## Show Me

OK. Enough of the easy stuff. Time for some conversions. The first one we shall tackle is converting a year-month-day triplet to a date value, `aaYMDToDate`. In the majority of date libraries I've seen (I used to collect these for fun), the author launches into this majestic, bizarre calculation, and awards himself points if he can get the entire thing into one statement. `EncodeDate` in Delphi's `SysUtils` is not that bad, considering. Essentially, in layman's terms, the calculation goes like this: calculate the number of complete years from the base date to the date we are trying to convert. Count the number of leap years in that range. Multiply the number of years by 365 and add in the number of leap years. Now count the number of days in the complete months up to the date we're encoding (be careful about 29 February, if this is a leap year). Add that to our subtotal. Finally add the day value. Phew! The calculation is not that bad, but just wait until you try it the other way round.

Anyway, the only thing that can be said about this calculation is that it's fairly compact, but there isn't really a lot we can do to speed it up. So, put it to one side, and let's rethink it in our terms. Part one of the calculation is calculating the number of days from 1 January 1800 to 1 January of the year we're concerned with. Well, heck, for a given year, *that's* always fixed, so why don't we calculate all the possible values (400 of them) and store them in an array (the `FirstJanuarys` array), once and once only at start-up? Agreed, this is one chunky array (400 `longint` values, 1,600 bytes) but it's going to be a major speedup. The calculation then reduces to a simple array access, wham bam. It's a graphic example, in a way, of the tradeoff of speed versus size: to get better speed we shall use this big array (there's going to be another benefit to having this array, just wait and see).

Well, what about part two of the calculation, calculating the number of days in the complete months in this year? Let's play the same trick, pre-calculating the values. This time we need 24 values, one set of 12 for a leap year, and one set of 12 for a normal year. Again, we can do this at unit initialization time; again, the calculation reduces to an array access.

With these two arrays, the conversion of a year-month-day triplet becomes a mere addition. One statement, he said, awarding himself bonus points. Listing 2 shows the `aaYMDToDate` routine, and the pre-calculation of the two arrays. Note that the year value must be four digits: still no Y2K problems.

OK, what about the opposite conversion, converting a date value to a year-month-day triplet? This calculation is, frankly, dreadful. If you have the source code to Delphi's VCL, check out the implementation of `DecodeDate` in SYSUTILS.PAS. In Delphi 4, it's 45 lines of code, with no comments. Generally it proceeds a bit like this. First, estimate the year value, by dividing the day number by 365.25, the average number of days in a year (some implementations estimate the number of 4-year periods by dividing by 1461, the number of days in a 4-year period that includes a leap year, and then fine tune by calculating the number of years left over). Once you've calculated the year, fine tune it by calculating the day number of 1 January of that year (or the previous, or the next) and checking which year the date falls in. From that point, it's a fairly easy process to calculate the month number (move through the months of that year, subtracting the days in each month until the number of days left is less than the days in the succeeding month). The Delphi implementation has a couple of other wrinkles to help speed up this basic algorithm.

So what are we going to do in our implementation? Well, not that lot, for a kick-off. I've got other things to talk about, thanks very much. Step back again. Remember that we have an array containing all of the dates for all of the 1 Januarys in our accepted range. To calculate the year value for a given date using this array, we'd just find the first element in that array that is greater than the date we have; the year is then given by the index of the previous element. I love it when a plan comes together! And hang on just a moment longer! We have another handy array containing the number of days prior to each month in a normal and in a leap year. Subtract the date of 1 January for this year from the date, find the first element in the month cumulative days array that is greater than the number of days we have left, and the month is the index of the previous element. Subtracting that value from the remaining days gives us the day part. Yowsa!

## Think Again

Let's just have a quick aside here on searching algorithms. The algorithm I just outlined requires us to find the first element in a sorted array that is greater than a particular value. It turns out that there are two general algorithms to do this, and one specific algorithm.

The first algorithm is called sequential search, and it goes like this: start at the first element of the array and step though the array until we reach the first element that is greater than the value we are looking for. We could run off the end of the array, but to avoid this we can set another element at the end of the array, called the sentinel, that has a value greater than any we shall be looking for; in our specific case, the array initialization will already have done this for us. I'm sure that you have all coded this type of search before. Usually it's coded as a `for` loop, with a `Break` statement to exit the loop early if the required element was found. If you are not very careful, the quick-and-dirty code has an interesting loophole. Listing 3 shows a sequential search on a sorted array, coded with a `for` loop, that has the bug. Or does it? See if you can spot it before continuing. It looks innocuous enough: go through the loop trying to find the element we want; if we find it, break out of the loop, if we don't find it, fall out of the bottom of the loop. Well, believe it or not, this code *does not work in Delphi 1*. It does work in the current versions of 32-bit Delphi, but is not guaranteed to. The bug is that the value of

➤ *Listing 2: Converting year-month-day to date.*

```
type
  PFirstJanuarys = ^TFirstJanuarys;
  TFirstJanuarys = array [0..400] of TaaDate;
  PCumulativeDays = ^TCumulativeDays;
  TCumulativeDays = array [boolean, 0..12] of word;
var
  FirstJanuarys : PFirstJanuarys;
  CumulativeDays : PCumulativeDays;
function aaYMDToDate(Y, M, D : integer) : TaaDate;
var
  IsLeap : boolean;
begin
  if not aaIsValidYMD(Y, M, D) then
    raise Exception.Create(Format(
      'aaYMDToDate: invalid year %d, month %d, day %d',
      [Y, M, D]));
  IsLeap := ((Y mod 4) = 0) and (Y <> 1800) and
            (Y <> 1900) and (Y <> 2100);
  Result := FirstJanuarys^[Y-MinYear] +
    CumulativeDays^[IsLeap, pred(M)] + pred(D);
end;
procedure InitFirstJans;
var
  NextValue : longint;
  Year      : integer;
begin
  {allocate the memory}
  New(FirstJanuarys);
  {initialize the values}
  NextValue := 0;
  for Year := MinYear to MaxYear do begin
    FirstJanuarys^[Year-MinYear] := NextValue;
    if aaIsLeapYear(Year) then
      inc(NextValue, 366)
    else
      inc(NextValue, 365)
  end;
  FirstJanuarys^[400] := NextValue;
end;
procedure InitCumulativeDays;
var
  NextValue : longint;
  Month     : integer;
begin
  {allocate the memory}
  New(CumulativeDays);
  {initialize the non-leap year values}
  NextValue := 0;
  for Month := 1 to 12 do begin
    CumulativeDays^[false, pred(Month)] := NextValue;
    inc(NextValue, DaysInMonth[Month]);
  end;
  CumulativeDays^[false, 12] := NextValue;
  {initialize the non-leap year values}
  NextValue := 0;
  for Month := 1 to 12 do begin
    CumulativeDays^[true, pred(Month)] := NextValue;
    if (Month = 2) then
      inc(NextValue, DaysInLeapFeb)
    else
      inc(NextValue, DaysInMonth[Month]);
  end;
  CumulativeDays^[true, 12] := NextValue;
end;
```

the loop counter *is not defined* when you fall out of the bottom of a loop, it's only defined if you `Break` out of it (read it in your *Object Pascal Language Guide* if you don't believe me). The code is implicitly assuming that the value of `Inx`, after the loop exits, is one more than the final value in the `for` statement. Luckily this is so for Delphi 2 through 4, but it is not so for Delphi 1. With the latter compiler, the value of `Inx` after the loop completes normally is the final value in the `for` statement. And I warn you that this fortuitous state of affairs may change in the future, if Inprise decide to change the compiler optimizer. Anyway, enough moralizing, Listing 4 has the fixed code for a sequential search through the `FirstJanuarys` array.

The problem with the sequential search is that it requires, on average, $n/2$ comparisons to find a single element in an array with $n$ of them. Can we do better than this? Since the array is sorted we could perform a binary search instead, where we take advantage of the fact that the elements in the array are in order of increasing value.

Binary search is a divide and conquer algorithm that works like this. Look at the element in the middle of the array. Is the value we are trying to find less than, equal to, or greater than this middle element? If it's equal, we're finished. If our value is less than the middle element, what can we say? Well, since the array is sorted, we can say that the element we want, if anywhere, is going to be in the first half of the array. We can ignore the other half of the array, it's not going to be found there. We can make a similar argument if the value we want is greater than the middle element: ignore the first half of the array. In any case, with just one comparison, we have halved the number of elements we have to search through. We can now do the same test with the half-array we have left (find the middle element in the half-array and compare), resulting in a quarter-array to look through. We can continue this process until the sub-sub-sub-array we have left has but one element, which is either the one we want or not. This is much better: by halving the number of elements we have to check each time, we are doing $\log_2(n)$ comparisons for an array of $n$ elements (for example, if the array had 128 elements we'd perform 7 comparisons in the worst case, compare that with 64 comparisons on average, 128 in the worst case for the sequential search).

Listing 5 has the code for a binary search through the `FirstJanuarys` array.

```
for Inx := 0 to pred(ArrayCount) do
  if MyArray[Inx] >= GivenValue then
    Break;
if Inx = ArrayCount then
  ..all elements less than given value..
else
  ..found first element >= given value..
```

➤ *Listing 3: Buggy sequential search routine.*

➤ *Listing 4: Correct sequential search on FirstJanuarys array.*

```
FoundIt := true;
for Inx := 0 to 400 do
  if (aDate < FirstJanuarys^[Inx]) then begin
    FoundIt := true;
    Break;
  end;
if FoundIt then
  dec(Inx)
else
  Inx := 399;
```

➤ *Listing 5: Binary search through FirstJanuarys array.*

```
FoundIt := false;
L := 0;
R := 400;
while (L <= R) do begin
  Mid := (L + R) div 2;
  if (aDate < FirstJanuarys^[Mid]) then
    R := pred(Mid)
  else if (aDate > FirstJanuarys^[Mid]) then
    L := succ(Mid)
  else {equal} begin
    FoundIt := true;
    Break;
  end;
end;
if FoundIt then
  Inx := Mid
else
  Inx := L-1;
```

## Alphabet City

As it happens, it can be proven that, for a set of data values about which we know nothing except that they are sorted, binary search is the best we can do. However, if we *do* know something more about the data values, we could take advantage of this extra knowledge and perhaps do better. And it is true that we do know something else about the `FirstJanuarys` array.

Suppose you have a phone book in your hands, and have to look up my name, Bucknall. If you were a computer application and had been coded to do an efficient binary search, you'd start of in the middle of the book, determine that Bucknall is in the first half, find the middle of the first half, determine that Bucknall is before *that*, and continue until you'd found my entry. For a phone book with 100,000 entries you'd do about 17 comparisons. However, you, the person, certainly wouldn't do that. You'd estimate that the Bs start near the front of the book and open it there first. You'd then use your knowledge of the alphabet to zero in faster to Bucknall than the binary search's 17 comparisons. This kind of search is known as an interpolation search: you are making use of some extra knowledge of the *distribution* of the data values to improve your chances of striking lucky early on. Since B is one thirteenth of the way through the alphabet, the names beginning

with B would be roughly one thirteenth of the way into the book and you'd eliminate a much larger chunk of the book in one go.

So what do we know about the distribution of the values in the FirstJanuarys array? Well, how about this: each value is either 365 or 366 greater than its predecessor. Can we take advantage of this? Certainly we can. Divide the date by 365 and use the answer as an index into the array. The element we get will either be the one we want, or if it is greater than the given date, the element before is the one we want. One comparison.

To see why this is so, consider the first 6 elements of this array: 0, 365, 730, 1095, 1460, 1826, corresponding to the years 1800 to 1805. The difference between the last two is 366, whereas between the others, it's 365. This corresponds to the fact that 1800 was not a leap year but 1804 was. Suppose we were trying to find the year for date 364 (31 December 1800). Divide by 365 to give the index of 0 and hence a year of 1800. This is correct. For date 365 (1 January 1801), we divide by 365 to give an index of 1 and hence a year of 1801, again correct. The first interesting value is 1825 (31 December 1804). Divide by 365 to give an index of 5 (it's an exact division, no remainder). Element 5 is 1826 which is greater than the one we are looking for, so we decrement the index to 4 to give a year of 1804. Date 1826 (1 January 1805) divided by 365 gives an index of 5 or a year of 1805, which is correct. Date 146096 (31 December 2199) divided by 365 gives an index of 400. The element at index 400 has a value of 146097, greater than the date we are looking for and so we reduce the index by 1 to give 399, or a year of 2199. In fact, the algorithm would break down if we had a range of years that included 365 leap years, but we don't, so let's not worry about it.

For fun, I coded up the aaDateToYMD routine to use each of these different searches to see how each fared against the other (Listing 6 shows that for interpolation search). If the interpolation search took 1 unit of time then the binary search took 1.3 units and the sequential search 7.6 units.

All very fascinating, but how do the *Algorithms Alfresco* routines compare with Delphi's EncodeDate and DecodeDate? Well, in my tests, if aaYMDToDate took 1 unit of time, EncodeDate took 3.75 units, and if aaDateToYMD took 1 unit, DecodeDate took 3.1 units. As you see, much better: we can do at least three times as many calculations than the standard Delphi routines in the same amount of time.

## When Smokey Sings

Time to go back to more date stuff after our diversion into searching algorithms. Finding the number of days between two TaaDate dates is a matter of subtracting one from the other. Finding the date *n* days from a given date is a matter of adding *n* to the date.

More easy routines. The day of the week for a date is a good one. For some reason, Delphi doesn't declare an enumerated type for the days of the week. (Quick, without looking it up, what integer value for Wednesday does SysUtils.DayOfWeek return? I confess to having to look it up *every* time I use the routine afresh.) So we'll declare an enumerated type called TaaDOW that contains the values aaSunday to aaSaturday. Calculating the day of the week is simple in theory: find the modulus base 7 of the date. However, this doesn't give the correct value, it's three days off because date 0, 1 January 1800, is a Wednesday not a Sunday. So we'll add three to the date before taking the modulus.

Also, Delphi's SysUtils unit defines the ShortDayNames and LongDayNames arrays so that you can determine the visible representation of a day of the week value. Since our TaaDOW type is not an integer, we'll have to provide a couple of small routines to return the string day name for a TaaDOW value.

Talking of days of the week, many times you want to know if a particular date is a Friday, for example. Well, you could certainly just call aaDayOfWeek and compare the result with aaFriday, but it's done often enough to warrant a specific routine called aaIsDayOfWeek that will check against any day of the week.

Another handy routine is one to return the date of the next Friday from a particular date (or the next Sunday, or whatever). Enter the aaNextDayOfWeek function to do just that, and the aaPrevDayOfWeek routine to calculate the previous Friday, or any other day of the week.

Another good must-have routine: the date today. This actually requires different code for 32-bit than for 16-bit Windows. In 16-bit we have to make a call to DOS to get the information and then convert it into a TaaDate value; in 32-bit we make a call to GetLocalTime and then convert.

## Jealous Lover

All very simple, I'm sure you'll agree. I think I'd better get onto some more difficult stuff otherwise

➤ *Listing 6: Converting a date to year, month and day values.*

```
procedure aaDateToYMD(aDate : TaaDate; var Y, M, D : integer);
var
  Inx : integer;
  IsLeap : boolean;
begin
  if (aDate < 0) or (aDate > MaxDate) then
    raise Exception.Create('aaDateToYMD: invalid date');
  {use interpolation search to calculate 1 January, & hence the year}
  Inx := aDate div 365;
  if (aDate < FirstJanuarys^[Inx]) then
    dec(Inx);
  Y := MinYear + Inx;
  IsLeap := ((Inx mod 4) = 0) and (Inx <> 0) and (Inx <> 100) and (Inx <> 300);
  {use interpolation search to calculate the month}
  aDate := aDate - FirstJanuarys^[Inx];
  Inx := (aDate div 32) + 1;
  if (aDate < CumulativeDays^[IsLeap, Inx]) then
    dec(Inx);
  M := succ(Inx);
  {calculate the day}
  D := aDate - CumulativeDays^[IsLeap, Inx] + 1;
end;
```

Our Esteemed Editor will start cutting chunks out of the article. Let's see what we can do about adding a number of months to a date. Before we start talking about any code, it would be beneficial to think about this for a moment. What date results from adding 3 months to 10 April 1999? Well, in normal usage we'd work it out to be 10 July 1999. Seems pretty obvious. What about adding 3 months to 30 April 1999? This is where is gets a little more fuzzy. There is one school of thought that says, by applying the implicit rule we just used, the answer is 30 July, 1999. There's another school of thought that says, well, 30 April is at the end of the month, so adding 3 complete months to that date should result in 31 July 1999. Mmm, ponder on that whilst I dish up another example. What happens if you add 3 months to 29 November 1998? Me, I'd say the answer was 28 February 1999. Good old Object Professional from TurboPower had a routine that would return 1 March 1999 in that case (with the reasoning that the answer should be 29 February 1999, which doesn't exist, but which could be viewed as 1 day *after* the last day of February, ie 1 March), but to be honest I've never known *anyone* to use that definition. (For OPRO fans *[Count me in! Ed]*, the routine is `IncDate` in the `OpDate` unit, although it has been carried forward into `SysTools` in the `StDate` unit.)

So, I'm sure you can see that adding months to a date requires some defining before we put fingers to keyboard and code it. Way back when, before my TurboPower days, I had to make the same decision about the same problem. I came up with the concept of 'sticky' month ends. If

```
function aaDateAddMonths(aDate : TaaDate; aMonths : integer; aStickyMonthEnds :
  boolean) : TaaDate;
var
  Y, M, D : integer;
  DaysInM : integer;
  StickToMonthEnd : boolean;
begin
  aaDateToYMD(aDate, Y, M, D);
  StickToMonthEnd := aStickyMonthEnds and (D = DaysInMonthPrim(Y, M));
  {calculate the month number from January 1800}
  M := (Y - MinYear) * 12 + pred(M) + aMonths;
  {if its out of range say so}
  if (M < 0) or (M > MaxMonth) then
    raise Exception.Create('aaDateAddMonths: calculated date in out of range');
  {calculate the new year and month}
  Y := (M div 12) + MinYear;
  M := succ(M mod 12);
  {check to see that the date is in range for the month}
  DaysInM := DaysInMonthPrim(Y, M);
  if StickToMonthEnd or (D > DaysInM) then
    D := DaysInM;
  Result := aaYMDToDate(Y, M, D);
end;
```

➤ *Listing 7: Adding a number of months to a date.*

➤ *Listing 8: Calculating the number of months and days between two dates.*

```
function aaDateDiffInMonths(aDate1, aDate2 : TaaDate;
  aStickyMonthEnds : boolean; var aDays : integer) :
  integer;
var
  TempDate : TaaDate;
  Y1, M1, D1 : integer;
  Y2, M2, D2 : integer;
  Date1AtME  : boolean;
  Date2AtME  : boolean;
begin
  {make sure that aDate1 is less than aDate2}
  if (aDate1 > aDate2) then begin
    TempDate := aDate1;
    aDate1 := aDate2;
    aDate2 := TempDate;
  end;
  {convert dates to YMD}
  aaDateToYMD(aDate1, Y1, M1, D1);
  aaDateToYMD(aDate2, Y2, M2, D2);
  {make first approximation to answer}
  Result := ((Y2 - Y1) * 12) + (M2 - M1);
  {if both day numbers are less then 28, we don't have to
   worry about any month end calculations}
  if (D1 < 28) and (D2 < 28) then begin
    {if the first day is less than or equal to the second,
     then the day count is just the difference}
    if (D1 <= D2) then
      aDays := D2 - D1
    {otherwise, the month count is one too many, then we
     have to count the number of days from Y2/(M2-1)/D1 to
     Y2/M2/D2; the former date being Result whole months
     from aDate1}
    else begin
      dec(Result);
      dec(M2);
      if (M2 = 0) then begin
        M2 := 12;
        dec(Y2);
      end;
      if (D1 > DaysInMonthPrim(Y2, M2)) then
        D1 := DaysInMonthPrim(Y2, M2);
      aDays := aDate2 - aaYMDToDate(Y2, M2, D1);
    end;
    Exit;
  end;
  {if we reach this point, one or both of the dates might be
   at a month end, so *beware*}
  Date1AtME := D1 = DaysInMonthPrim(Y1, M1);
  Date2AtME := D2 = DaysInMonthPrim(Y2, M2);
  {the easiest case is both days are at month ends and we
   want sticky month ends: we're done after setting aDays
   to zero}
  if aStickyMonthEnds and Date1AtME and Date2AtME then begin
    aDays := 0;
    Exit;
  end;
  {the next easiest cases all use sticky month ends}
  if aStickyMonthEnds then begin
    {if the first date is at a month end (the second won't
     be) then the number of months is one too many, and the
     number of days is equal to the second day value}
    if Date1AtME then begin {note: Date2AtME = false}
      dec(Result);
      aDays := D2;
      Exit;
    end;
    {if the second date is at a month end (the first won't
     be) then the number of months is correct, and the
     number of days is equal to the second day value minus
     the first, or zero if this is negative}
    if Date2AtME then begin {note: Date1AtME = false}
      if D2 >= D1 then
        aDays := D2 - D1
      else
        aDays := 0;
      Exit;
    end;
  end;
  {if the second day number is greater or equal to the
   first, the number of days is the difference; the number
   of months is correct}
  if (D2 >= D1) then begin
    aDays := D2 - D1;
    Exit;
  end;
  {otherwise, the number of months is one too many, and the
   number of days is that from Y2/(M2-1)/D1 to Y2/M2/D2}
  dec(Result);
  dec(M2);
  if (M2 = 0) then begin
    M2 := 12;
    dec(Y2);
  end;
  if (D1 > DaysInMonthPrim(Y2, M2)) then
    D1 := DaysInMonthPrim(Y2, M2);
  aDays := aDate2 - aaYMDToDate(Y2, M2, D1);
end;
```

sticky month ends are in force, adding a number of months to a date which is at the end of the month always results in a date which is at a month end. Month ends stick. So 28 February 1999 plus one month is 31 March 1999, using this mode. If sticky month ends are *not* in force, then adding a month to a date that is a month end performs no special month end coercion. 28 February 1999 plus one month is 28 March 1999, using this mode. In *all* cases, if adding a number of months would result in a non-existent date, for example 30 January plus one month to 'give' 30 February, then the routine would force the result date back to the last day of the month concerned. Listing 7 has the details for the `aaDateAddMonths` routine. Notice that the routine accepts a `Boolean` parameter called `aStickyMonthEnds` that defines whether sticky month ends are to be used.

Cute. So how about calculating the number of months and odd days between two dates? This is a great one. If you thought adding a number of months was complicated enough, check this one out. The easy example first, to lull you into a false sense of security. The number of months between 15 April 1999 and 15 July 1999 is 3 months, I'm sure you'll agree. Piece of cake. Or is it? Well, it could be argued that there are two full months in between the two dates, May and June. Then, there are 15 remaining days in April and 15 in July, to make 30 days. So the answer should be 2 months and 30 days, no?

No. We take the usual rule that the whole months are counted from the lesser date. After all, it makes sense that if you calculate the number of months between two dates and then call `aaDateAddMonths` with this number of months and the lesser date, you get the greater date. In other words that the two routines we write should be complementary. So, again we shall have to use the sticky month end concept. If you read through the code in Listing 8, you'll get the idea about this algorithm.

## Ark-Angel

What's next on the agenda? Business date calculations. What we want to do here is to test whether a date is a business day, or a weekend or holiday day instead. Once we can determine whether a date is a business day, we shall then want to calculate how many business days there are between two dates, add a number of business days to a date, and indeed calculate the nearest business date to a given date.

The first routine, then, is a simple `Boolean` function that returns whether a date is a business day or not. To do this, we need to know, firstly, what days are non-business days every week (for example, Saturday and Sunday in Western countries) and, secondly, what days are non-working holidays during the year. This latter one, especially in the US, tends to be company-specific: for example, I have a different set of general holidays than does my wife.

What we'll do is define a class to encapsulate non-working time. The class will have methods to set the weekend days, and methods to add and remove holiday dates. Once a holiday object has been set up with weekend days and a list of holidays, a method can be called to determine if a date is a holiday or not. Of course, the holiday object must be persistent: we should be able to store the object to a stream and read it back.

Now having this object around, how do we calculate the number of business days between two dates? There's nothing for it but to be long-winded, I'm afraid. We have to step through every date from the lesser to the greater date, testing each date encountered to be a business date, and counting them as we go. Similarly, for adding a number of business days to a date, it's a step by step process.

Finally, how do you calculate the nearest business date for a given date? There are two schools of thought here, as far as I know. Rule one is, if the given date is not a business day, you calculate the next business date and use that. If the given date *is* a business day then there's nothing to do. For certain businesses, the nearest business date to a given date cannot be in a different month: if the given date is a holiday at the month end, then the nearest business date is the last business day of that month. In other words with rule 1 you always move forwards, with rule 2 you generally move forwards, except at month end when you move backwards.

I won't show the business date arithmetic here, compared with some of the things we've been looking at here, it's not too complicated. The code can be found on this month's disk.

## Many Happy Returns

Also in the unit on the disk, you'll find routines to convert to and from `TDateTime` values, and to convert to and from TurboPower's SysTools dates. I've added a conversion to and from standard ISO dates as well (ie, week numbers). I've included a routine to take a `TaaDate` value and return a string (the equivalent to `DateToStr`). It has a few more formats, including one that mimics the old Lotus 1-2-3 format, dd-Mmm-yyyy, of which I'm particularly fond.

And all through the code, there is nary a two-digit year to be found. No Y2K problem at all. I'm leaving all those 2-digit year problems to `SysUtils`'s `StrToDate` function.

I hope you've found this article and code to be of use. We've learned some lessons along the way: about searching algorithms; that reducing the scope of routines whilst keeping them useful is one way to squeeze more speed; that date routines can simultaneously be a pain in the neck and very interesting.

---

Julian Bucknall is a Saturday's child and works hard for a living. He also learnt his ABC at an early age. He can be reached at julianb@turbopower.com. The code that accompanies this article is freeware and can be used as-is in your own applications.