2026-08-09 – Counting the days, revisited

⇐ 2026-03-14 ☐ ⊨ ☆ ⇒

Many years ago I wrote about how to convert Gregorian dates to Julian Day numbers or similar counts such as *rata die* as used in Calendrical Calculations. This algorithm is the core of C's mktime() function that converts a broken-down date-time into linear `time_t`.

I recently learned from Ben Joffe that I was missing a few tricks, and my old code wasn't as good as it could have been. Here's a better version (using conventional not C numbering):

```
if m > 2 { m -= 2; } else { m += 10; y -= 1; }
y*365 + y/4 - y/100 + y/400 + m*979/32 + d - 336
```


the main idea

There's a helpful coincidence in the Gregorian calendar.

Although the month lengths aren't obviously regular, there's a repeating 5 month pattern that becomes easier to see when you start from March, as illustrated by the table below.

This pattern resets at the end of February, midway through its third repeat, *coincidentally* at the same point that leap days occur.

Thus the first line of the code above adjusts the month and year numbers so that January and February are counted at the end of the previous year, and the coincidental alignment occurs at the boundary between the adjusted year numbers.

I'll explain the details of the adjustment as I discuss the relevant parts of the second line

March 31 days	April 30 days	May 31 days	June 30 days	July 31 days
August 31 days	September 30 days	October 31 days	November 30 days	December 31 days
January 31 days	February 28 or 29			

Julian years

The first part of the main formula counts the number of days before the start of year `y`, in terms of normal years and leap days.

```
y * 365 + y / 4
```

The adjustment subtracts one from the year in January and February. The effect is that the leap day in year 4 is counted as a day before the start of the adjusted beginning of year 4, i.e. before March, i.e. exactly the right place.

I previously combined this part of the expression into a single term,

```
y * 1461 / 4
```

Ben Joffe pointed out that when it is written this way the function is only able to make use of 25% of the range of its output data type, because the multiplication overflows for very large year numbers. And on modern CPUs it isn't actually faster to eliminate the addition.

Gregorian correction

The next part corrects the number of leap years before the current year.

```
- y/100 + y/400
```

It works in basically the same way as the Julian leap year calculation, but whereas `y/4` trivially compiles to a simple shift operation, this needs a bit more cleverness.

As Hacker's Delight explains, a modern compiler will turn `y/100` into a multiply-and-shift:

```
y * (1<<N) * (1/25) >> (N+2)
```

That is, the compiler uses a fixed-point representation of the reciprocal of the divisor. Then it uses common subexpression elimination to suppress the second multiplication by 1/25.

So these two divisions are turned into a wide multiply and two shifts. Neat.

the month pattern

The next part counts the number of days in this (adjusted) year before the start of month `m`.

```
m * 979 / 32
```

I previously wrote it using the number of days in the repeating pattern of 5 months,

```
m * 153 / 5
```

But this is relatively difficult for compilers to optimize well (clang uses two multiplications instead of one), and they don't know the range of `m` is limited, so we can do better by turning it into a multiply-and-shift by hand.

979/32 == 30.59375 which is close enough to the exact value 153/5 == 30.6

Either of these expressions produce the right 5 month long/short pattern, but the pattern doesn't necessarily line up with the normal month numbering. (The two expressions above need different adjustments.)

We move March to number 1, just before the start of the pattern. When counting the days before April, we get 31 more than the count for March; when counting the days before May, we get 30 more than the count for April, etc.

January is adjusted to follow December to match the adjusted year numbering.

```
   m  *979/32  diff
--------------------
   1     30          March
   2     61     31   April
   3     91     30   May
   4    122     31
   5    152     30
   6    183     31
   7    214     31
   8    244     30
   9    275     31
  10    305     30   December
  11    336     31   January
  12    367     31
  13    397     30
```

the epoch

Because calendars count from 1, the Gregorian date 0001-01-01 gets numbered *rata die* 1.

The adjustments turn January into month 11, and so (as in the second column in the table above) we count 336 days in the adjusted year 0 before January. We need to subtract those extra days to compensate for the adjustment.

We can change the offset to choose a different epoch, e.g. the MJD epoch 1858-11-17 is *r.d.* 678576, and the Unix epoch 1970-01-01 is *r.d.* 719163.

domains and ranges

In my old C code I casually used `int`, which misleadingly implied that it worked with proleptic Gregorian calendar dates before year 1.

However signed division and modulus on common CPUs and low-level programming languages truncates towards zero, but this algorithm needs Euclidean or flooring division (which are equivalent for positive divisors).

So it's better to use `u32` for these calculations. (Compilers also do a better job when this code uses unsigned integers.) To support negative years, a multiple of 400 years can be added to move year 0 to the middle of the `u32` range, and subtracted from the return value to produce a signed count of days.

leap year test

Ben Joffe also examined fast leap year tests.

My new favourite one is I think the neatest if not the fastest:

```
if y % 25 == 0 { y % 16 == 0 }
else           { y %  4 == 0 }
```

Note that 25 * 16 == 400, so it's a leap year if it's divisible by both 25 and by 16, or if it's divisible by 4 but not by 25.

The classic version of this check first tests divisibility by 100. CPUs that rely on branch prediction will correctly predict it 99% of the time: much better than the 75% you get from testing divisibilty by 4 first!

But nowadays this `if` is compiled into a CMOV or CSEL (so branch prediction doesn't matter), and divisibility by 25 is easier to compile than divisibility by 100.

length of month

What prompted me to revisit this code was the idea that it's possible to work out a simpler multiply-and-shift optimization when the values have limited ranges (as in `m*979/32` above) and/or when we don't depend on the exact result of the multiplication.

I previously wrote this code for calculating the length of a month:

```
if m == 2 {
    28 + is_leap_year(y) as u32
} else {
    30 + (m * 275 % 9 > 3) as u32
}
```

The multiply-and-shift idea led me to this replacement for months other than February:

```
30 + (m * 7 % 16 < 9) as u32
```

I found it by writing a brute force program that tries successive bitmask widths and multipliers, until it finds a case where all the long months produce results greater than all the short months, or vice versa (as in the winner).

But there's a neater expression, apparently due to Dr Matthias Kretz:

```
30 | (m ^ (m >> 3))
```

This uses two tricks:

- The 1 bit of the month number matches the odd/even long/short pattern in the months before August (month 8), when the phase flips. The flip is done by using the 8 bit to toggle the 1 bit.
- Bitwise or with 30 sets bits 2, 4, 8, 16 so the higher bits of the month number don't matter.

It compiles to just two ARM instructions:

```
eor w0, w0, w0, lsr #3
orr w0, w0, #0x1e
```

It's so sweet I actually love how much better it is than my attempt!

⇐ 2026-03-14 ⇐ poached eggs ⇐ ⇒ ☆ ⇒ ☆ ⇒

Tony Finch <dot@dotat.at>