NEWS · AUGUST 13, 2026

Two Countries, Twenty Holidays, One Data Structure

Part one of a two-part series on computing public holidays in Delphi — why a holiday calendar belongs in a data structure instead of a case statement, three small date primitives built on System.DateUtils, and the complete US federal holiday table in one readable array.

By Dr. Holger Flick

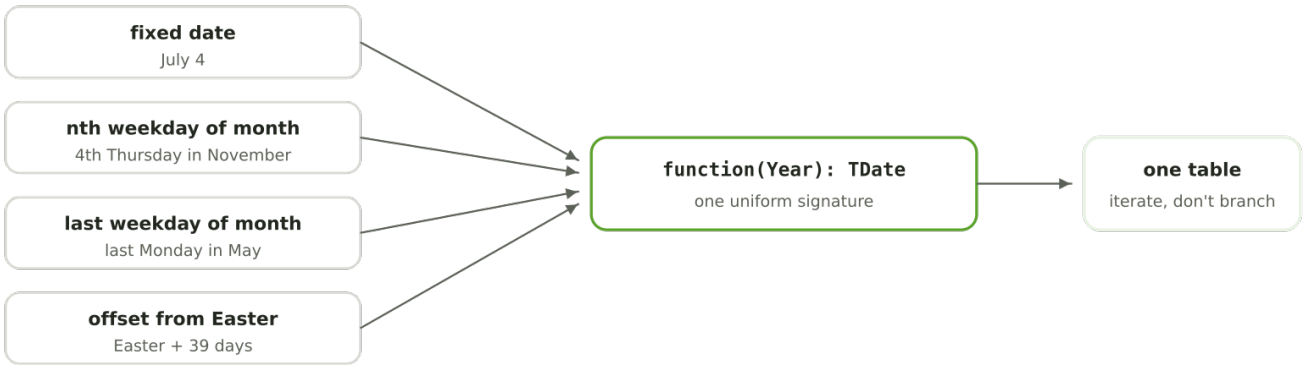
Every business application eventually needs to know whether a given day is a working day. Payroll needs it. Delivery-date estimates need it. Anything that computes an SLA in business days needs it. And the first version of that code, in almost every codebase I've seen, is a `case` statement that grows a new branch every December.

The problem isn't the branches. It's that a holiday isn't a date — it's a *rule*. "Christmas Day" is a fixed date and always will be. But "Thanksgiving" is the fourth Thursday in November, "Memorial Day" is the *last* Monday in May, and "Ostermontag" is whatever day follows a lunisolar calculation that predates the computer by about fifteen centuries. Hard-code the dates and you're back next year. Encode the rules and you're done forever.

This is a two-part series on doing exactly that in Delphi, built entirely on `System.DateUtils` — the RTL unit I toured in Part 1 of the DateUtils series. **This part** builds the data structure and the three date primitives every rule-based holiday is made of, then uses them to express all eleven US federal holidays in a single readable array. Part 2 takes on the interesting half — Easter, the four German holidays that hang off it, and merging two national calendars into one. By the end you'll have a holiday engine that's about a hundred lines and never needs updating.

A holiday is a function from a year to a date

The insight that makes all of this collapse is small: whatever else a holiday is culturally, computationally it is a function that takes a year and returns a date. `2026 ' 2026-11-26`. Nothing more. Once every holiday has that same shape, they all fit in one list, and the code that *uses* the list never needs to know which kind of rule it's looking at.



Every holiday, however it's defined, reduces to the same signature: year in, date out

Read the diagram left to right and you have the whole architecture. Four structurally different kinds of rule — and *because* they share a signature, the calling code has exactly one path through it. There is no `if Kind = hkFixed then … else if Kind = hkNthWeekday then …` anywhere in what follows. That branch is precisely what we're designing out.

Delphi expresses "a function you can store in a variable" with an anonymous method type, declared with `reference to`:

```
uses
  System.SysUtils, System.DateUtils, System.Generics.Collections;

type
  /// Computes the date this holiday falls on in a given year.
  TComputeHoliday = reference to function(const AYear: Word): TDate;

  THolidayRule = record
    UID: string;      // stable identifier — safe to store in a database
    Name: string;     // display name
    Compute: TComputeHoliday;
    constructor Create(const AUID, AName: string; const ACompute: TComputeHoliday);
  end;

constructor THolidayRule.Create(const AUID, AName: string;
  const ACompute: TComputeHoliday);
begin
  UID := AUID;
  Name := AName;
  Compute := ACompute;
end;
```

The `UID` deserves a word, because it's the field people leave out and regret. Display names change — they get translated, they get re-spelled, and in one memorable case a US holiday's name was amended by statute. A stable machine identifier means a row in your database that says "this employee took `memorial-day` off" still means something after any of that.

> **Why a record and not a class**
>
> A `record` with an anonymous-method field is managed by the compiler — no `Create`/`Free` pairs, no ownership question, and the array of them cleans itself up. The closure each `Compute` captures is reference-counted like an interface. A class would work identically; it would just hand you a memory-management chore in exchange for nothing.

The three primitives, and why each one is a one-liner

The eleven US federal holidays are defined by 5 U.S.C. § 6103, and reading the statute closely is genuinely useful here: it uses exactly three phrasings. Some holidays are a calendar date ("January 1"). Some are an ordinal weekday ("the third Monday in January"). One is a terminal weekday ("the last Monday in May"). That's the entire vocabulary, so that's exactly three helper functions.

The first one barely needs writing, but naming it pays off in how the rule table reads:

```
function Fixed(const AYear, AMonth, ADay: Word): TDate;
begin
  Result := EncodeDate(AYear, AMonth, ADay);
end;
```

The second is the interesting one. To find the *n*th Monday of a month, start at the 1st, ask which weekday that is, step forward to the first Monday, then add whole weeks. `DayOfTheWeek` from `System.DateUtils` is the right tool because it's ISO 8601-numbered — Monday is 1 through Sunday is 7 — which lets the arithmetic stay honest:

```
/// The ANth occurrence of ADayOfWeek in a month.
/// ADayOfWeek uses the DateUtils constants: DayMonday = 1 .. DaySunday = 7.
function NthWeekdayOfMonth(const AYear, AMonth, ADayOfWeek, ANth: Word): TDate;
var
  FirstOfMonth: TDate;
  Offset: Integer;
begin
  FirstOfMonth := EncodeDate(AYear, AMonth, 1);
  // How many days forward from the 1st to the first ADayOfWeek?
  Offset := (Integer(ADayOfWeek) - DayOfTheWeek(FirstOfMonth) + 7) mod 7;
  Result := IncDay(FirstOfMonth, Offset + 7 * (Integer(ANth) - 1));
end;
```

The `+ 7) mod 7` is the part worth pausing on: it turns a possibly-negative difference into a forward step of 0–6 days, so the function works whether the 1st falls before or after the target weekday. Drop it and the calendar is right about half the time — which is exactly the kind of bug that survives testing in January and surfaces in September.

The third primitive works backward from the end of the month, and this is where `EndOfTheMonth` from Part 1 earns its keep by knowing how long every month is, leap years included:

```
/// The last occurrence of ADayOfWeek in a month.
function LastWeekdayOfMonth(const AYear, AMonth, ADayOfWeek: Word): TDate;
var
  LastOfMonth: TDate;
  Offset: Integer;
begin
  LastOfMonth := DateOf(EndOfTheMonth(EncodeDate(AYear, AMonth, 1)));
  // How many days back from the last day to the last ADayOfWeek?
  Offset := (DayOfTheWeek(LastOfMonth) - Integer(ADayOfWeek) + 7) mod 7;
  Result := IncDay(LastOfMonth, -Offset);
end;
```

Note `DateOf` wrapped around `EndOfTheMonth`: `EndOfTheMonth` returns the last *millisecond* of the month,

`23:59:59.999`, and we want a clean midnight date. It's a one-word fix that prevents a `TDate` carrying a stray time fraction into every comparison you later make against it.

!!! warning "\"Last\" is not \"fifth,\" and it isn't \"fourth\" either" This is the single most common bug in hand-rolled holiday code. Memorial Day is the *last* Monday in May — and May has five Mondays in some years and four in

others, so neither `NthWeekdayOfMonth(…, 4)` nor `NthWeekdayOfMonth(…, 5)` is correct in general. In 2026 the last Monday of May is the 25th; in 2027 it's the 31st, which *is* the fifth. Ask for the fifth Monday of a May that only

has four and `NthWeekdayOfMonth` will happily hand you a date in June. That's why `LastWeekdayOfMonth` counts backward from the end instead of forward from the start — the question "which is last" has an answer every year, and the question "is there a fifth" does not.

> **The RTL has a version of this too**
>
> `System.DateUtils` ships `EncodeDayOfWeekInMonth` (and a `TryEncodeDayOfWeekInMonth` that returns `False` instead of raising `EConvertError`), which does the nth-weekday job directly. It's a perfectly good choice and one fewer function to own. I've written the arithmetic out here because it makes the boundary behavior visible — and because the "last weekday" case still needs the backward-counting version regardless. Use the RTL function for the nth case if you prefer; the rule table below doesn't care which implementation sits behind the name.

The US table: eleven holidays, eleven lines

With the primitives in place, the entire US federal holiday calendar becomes a list of rules that reads almost exactly like the statute it implements. This is the payoff for all the setup — and notice there is no logic here at all, just data:

```
function USRules: TArray<THolidayRule>;
begin
  Result := [
    THolidayRule.Create('new-years-day', 'New Year''s Day',
      function(const Y: Word): TDate begin Result := Fixed(Y, 1, 1); end),
    THolidayRule.Create('mlk-day', 'Martin Luther King Jr. Day',
      function(const Y: Word): TDate begin Result := NthWeekdayOfMonth(Y, 1, DayMonday, 3); end),
    THolidayRule.Create('presidents-day', 'Washington''s Birthday',
      function(const Y: Word): TDate begin Result := NthWeekdayOfMonth(Y, 2, DayMonday, 3); end),
    THolidayRule.Create('memorial-day', 'Memorial Day',
      function(const Y: Word): TDate begin Result := LastWeekdayOfMonth(Y, 5, DayMonday); end),
    THolidayRule.Create('juneteenth', 'Juneteenth National Independence Day',
      function(const Y: Word): TDate begin Result := Fixed(Y, 6, 19); end),
    THolidayRule.Create('independence-day', 'Independence Day',
      function(const Y: Word): TDate begin Result := Fixed(Y, 7, 4); end),
    THolidayRule.Create('labor-day', 'Labor Day',
      function(const Y: Word): TDate begin Result := NthWeekdayOfMonth(Y, 9, DayMonday, 1); end),
    THolidayRule.Create('columbus-day', 'Columbus Day',
      function(const Y: Word): TDate begin Result := NthWeekdayOfMonth(Y, 10, DayMonday, 2); end),
    THolidayRule.Create('veterans-day', 'Veterans Day',
      function(const Y: Word): TDate begin Result := Fixed(Y, 11, 11); end),
    THolidayRule.Create('thanksgiving', 'Thanksgiving Day',
      function(const Y: Word): TDate begin Result := NthWeekdayOfMonth(Y, 11, DayThursday, 4);
end),
    THolidayRule.Create('christmas-day', 'Christmas Day',
      function(const Y: Word): TDate begin Result := Fixed(Y, 12, 25); end)
  ];
end;
```

Two naming notes, since precision matters when the source is a statute. The holiday most Americans call "Presidents' Day" is named **Washington's Birthday** in federal law — the statute has never been amended to change it, though many states use the other name. And Juneteenth's full legal name is **Juneteenth National Independence Day**, added to the list by the Juneteenth National Independence Day Act in 2021, which is why any holiday table written before then is missing a row. I've kept the legal names in `Name` and the familiar ones in the `UID`; if you're displaying these to users, a display-name lookup is the place to disagree with Congress.

> **What this table deliberately doesn't do**
>
> These are the **federal** holidays — the days federal employees get off. The United States has no national holidays binding on private employers, and states add their own (Patriots' Day in Massachusetts, Cesar Chavez Day in California, and so on). This table is the common denominator every US calendar agrees on, which makes it the right default and the wrong final answer if you're doing payroll in a specific state. The same scoping applies to Germany, and I'll come back to it in Part 2.

Evaluating the table

Because every rule has the same signature, computing a year's calendar is one loop with no knowledge of rule types in it whatsoever:

```
type
  THoliday = record
    UID: string;
    Name: string;
    Date: TDate;
  end;

function ComputeYear(const ARules: TArray<THolidayRule>;
  const AYear: Word): TArray<THoliday>;
var
  I: Integer;
begin
  SetLength(Result, Length(ARules));
  for I := 0 to High(ARules) do
  begin
    Result[I].UID  := ARules[I].UID;
    Result[I].Name := ARules[I].Name;
    Result[I].Date := ARules[I].Compute(AYear);   // the only line that varies
  end;
end;
```

That single call `ARules[I].Compute(AYear)` is where the `case` statement went. Adding a twelfth holiday means adding a twelfth line to the table — this function never changes, and neither does anything downstream of it.

Printing the 2026 calendar is then a formality:

```
var
  H: THoliday;
begin
  for H in ComputeYear(USRules, 2026) do
    Writeln(Format('%-45s %s', [H.Name, FormatDateTime('ddd, dd mmm yyyy', H.Date)]));
end;
```

which produces the eleven days below. Worth spot-checking a couple against a calendar, because verified output is the only kind worth publishing:

```
New Year's Day                              Thu, 01 Jan 2026
Martin Luther King Jr. Day                  Mon, 19 Jan 2026
Washington's Birthday                       Mon, 16 Feb 2026
Memorial Day                                Mon, 25 May 2026
Juneteenth National Independence Day        Fri, 19 Jun 2026
Independence Day                            Sat, 04 Jul 2026
Labor Day                                   Mon, 07 Sep 2026
Columbus Day                                Mon, 12 Oct 2026
Veterans Day                                Wed, 11 Nov 2026
Thanksgiving Day                            Thu, 26 Nov 2026
Christmas Day                               Fri, 25 Dec 2026
```

Independence Day 2026 falls on a Saturday, which is a good reminder that "is a holiday" and "is a day off" are different questions — federal practice observes a Saturday holiday on the preceding Friday, per § 6103(b).

That's an *observation* rule layered on top of the *occurrence* rule, and keeping the two separate is what lets you answer both questions from one table. I'll wire that up in Part 2, because Germany deliberately does not do it.

Is a lookup function worth it?

Once the year is computed, the questions you actually ask are "is this date a holiday?" and "which one?" A linear scan over eleven items is genuinely fine, but if you're calling it inside a business-day loop over a date range, a dictionary keyed by the date is nicer to use and turns the answer into a hash lookup:

```
function HolidayIndex(const ARules: TArray<THolidayRule>;
  const AYear: Word): TDictionary<TDate, string>;
var
  H: THoliday;
begin
  Result := TDictionary<TDate, string>.Create;
  for H in ComputeYear(ARules, AYear) do
    Result.AddOrSetValue(H.Date, H.Name);   // AddOrSetValue: two holidays can share a date
end;
```

`AddOrSetValue` rather than `Add` is not defensive padding — it's load-bearing. Two holidays landing on the same date is a real scenario, and it shows up the moment you merge two countries' calendars. That merge is Part 2's job, and it's where this data structure starts genuinely paying for itself.

> **You don't have to build this yourself**
> If you want a holiday calendar rather than an understanding of one, mature open-source options exist and are worth knowing. In the Delphi world, TZDB handles the related time-zone problem with real IANA data. Outside it, python-holidays and Nager.Date (.NET) both cover well over a hundred countries including subdivisions, and date-holidays does the same for JavaScript. They carry the state-level and regional rules this series deliberately skips. The reason to write your own is that the rules for one or two countries are genuinely small — as the table above shows — and a dependency you fully understand in ninety lines is often the better trade. If you need forty countries, take the library.

Takeaways

Part 1 was really about one refactoring: turning behavior into data so the code that consumes it stops branching.

- **A holiday is a function from a year to a date.** Give every rule that signature and the calling code has exactly one path through it.
- **Three primitives cover the entire US statute**: a fixed date, the *n*th weekday of a month, and the *last* weekday of a month. `DayOfTheWeek` and `EndOfTheMonth` do the hard parts.
- **"Last" is not "fourth" or "fifth."** Count backward from the end of the month; forward-counting silently produces dates in the following month.
- **A stable `UID` alongside the display name** keeps stored data meaningful when names get translated, re-spelled, or amended by legislation.

> Encode the rule, not the date. A holiday table built from rules is written once and correct in every year you never tested.

The US calendar was the easy country: eleven holidays, all of them anchored to the Gregorian calendar we already compute in. Germany is the interesting one — four of its nine nationwide holidays are defined by their distance from Easter, and Easter is a lunisolar computation with no closed-form date at all. Part 2 implements it, adds the German table, and merges both countries into a single calendar with the same three-line loop. See you there.

> **The series: Computing Public Holidays in Delphi**

> Two parts: 1. Rules as data, and the US federal calendar **— you are here** 2. Easter, the German calendar, and merging two countries Both build on the Dates and Times in Delphi tutorial.

INDEPENDENT · AD-FREE · READER-SUPPORTED

Free to read. Not free to make.

Every tutorial, article, and line of example code on Holger's Code is published without a paywall and without ads. This page is about what keeps it that way.

Professional work, published free

The content here is held to the same standard as professional client work: every tutorial is built as a real project, tested against current versions, documented step by step, and revisited when the frameworks move on. A single in-depth tutorial routinely takes days — and that's before the bills behind it: servers, storage, build infrastructure, licenses, and the hardware everything is verified on.

If an article or tutorial here has saved you an afternoon of debugging, taught you a technique, or helped you ship, it created real value for you. Supporting it isn't charity — it's paying for professional work you found useful, at whatever price you choose. And if you can't, keep reading anyway. That's what the free part means.

Ways to support

KO-FI

Buy me a coffee

A one-time tip — no account, no subscription, any amount.

BOOKS & COURSES

Buy the paid work

The paid tier of everything published free here — and you get something lasting in return.

SPREAD THE WORD

Share an article

Send a tutorial to a colleague, cite an article, subscribe to the RSS feed.

Nothing here goes behind a paywall, whether you chip in or not. But if something on this site earned its keep today, this is where to say so.

Support on Ko-fi —