

Date Manipulation

By Nathan Torkington

Introduction

As the new millennium approaches, many people are learning (some for the first time) that there are two "l"s and two "n"s in millennium. Others are learning that not all languages have date functions with obvious return values. In this column, you'll learn how Perl handles dates, how to avoid writing a Y2K bug, and how to determine exactly when the Y2.038K bug will strike.

What's in a Date

There are many different ways to represent dates as strings:

```
"18 Jan 1973"  
"18/01/1973"  
"01/18/1973"  
"Jan 18 1973"  
"18-01-73"  
"18-01-1973"  
"01/73"
```

Some formats are particular to email messages, some to HTTP headers, some to checkbooks, some to utility bills; some are used only on credit cards, while some formats are simply ambiguous (for example, is "01-06-1973" the 1st day of the 6th month, or the 6th day of the 1st month?). Regardless of their uses all formats are hard to manipulate.

If you want to find the difference between "18 Jan 1973" and "6 Sep 1950", you'll need to convert to a numeric representation. Unix uses such a representation internally: epoch seconds. A date and time together are represented as seconds since midnight January 1, 1970 GMT. "18 Jan 1973" (let's assume midnight) comes out as 96163200. Midnight starts a day under this system.

You can test this yourself with the `gmtime` function built into Perl. Give it an integer representing seconds since the epoch, and in scalar context it will return a string representing the date at that point in time:

```
perl -le 'print scalar gmtime 96163200'  
Thu Jan 18 00:00:00 1973
```

If you call `gmtime()` in list context, you'll get back a list of distinct values for hour, minute, seconds, day, month, year, etc:

```
perl -le 'print join(", ", gmtime 96163200)'  
0,0,0,18,0,73,4,17,0
```

The first three zeroes represent seconds, minutes, and hour values respectively. Hours are 0-23, so P.M. is hour 12 and later. The next value is the day in the month (the 18th in this case).

Date Manipulation

By Nathan Torkington

The fifth value is the month number, which starts at zero (representing January). It starts at zero because it's meant to be a subscript into an array of month names:

```
@months = qw(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec); $month = (gmtime 96163200)[4]; # "Jan"
```

The year (73 in the example above) is the source of much confusion. It is not the last two digits of the year. It is the year with 1900 subtracted from it. Why is this the case? That's the way C did it, and Perl tries to make its versions of the library and system calls as close to the operating system's as possible.

So if you wanted to print out a 4-digit year, you would say:

```
$year = (gmtime 96163200)[5] + 1900;
```

Failure to understand how to do this is one cause of those Y2K bugs you've probably heard about:

```
$year = "19" . (gmtime 96163200)[5];  
# BAD. 2000 gives 19100
```

You're not done with the return value of `gmtime()`, though: there's still 4, 17, and 0. Those are the day of the week (Sunday is day 0), the day of the year (0 is the first day of the year. Go figure), and whether or not daylight savings time is in effect (0 indicating it wasn't, positive if it was, negative if the information was unavailable).

The `time()` function, built into Perl, returns you the current date and time as an epoch seconds value. If you want to turn this into a string, use the `gmtime()` and `localtime()` functions:

```
$now = localtime(time());  
($sec, $min, $hour, $day, $mon, $year, $wday, $yday, $isdst) = localtime(time());
```

If you call `localtime()` or `gmtime()` with no argument, it will call `time()` for you:

```
$now = localtime();  
($sec, $min, $hour, $day, $mon, $year, $wday, $yday, $isdst) = localtime();
```

If you want to find the difference between two points in time, convert them to epoch seconds values and then subtract one from the other:

```
$difference_in_seconds = $later_datetime —  
$earlier_datetime;
```

Date Manipulation

By Nathan Torkington

To convert seconds to minutes, hours, or days, you simply divide by 60, 3600, and 86400 respectively:

```
$difference_in_minutes = $difference_in_seconds / 60;  
$difference_in_hours = $difference_in_seconds / 3600;  
$difference_in_day = $difference_in_seconds / 86400;
```

You can use this division in reverse to answer the question "what will the date be four days from now?"

```
$then = time() + 86400 * 4;  
print scalar localtime $then;
```

This gives an answer accurate to the second. For instance, if the epoch seconds value four days in the future is 932836935, then when you print the date out as a string you see:

```
Sat Jul 24 11:23:17 1999
```

If you want to drop back to the midnight that started the date (e.g. "Sat Jul 24 00:00:00 1999") use modulus:

```
$then = $then — $then % 86400;  
# truncate to the day
```

Similarly you can round to the nearest midnight with:

```
$then += 43200; # add on half a day  
$then = $then — $then % 86400;  
# truncate to the day
```

This works if your time zone is an even number of hours away from GMT. Not all time zones are so obliging. What you really want is an epoch seconds value where the seconds are measured in your own time zone, not in GMT.

Perl comes with a module called `Time::Local` which provides you with two functions, `timelocal()` and `timegm()`. These take the same list of values that `localtime()` and `gmtime()` return, and give you back an epoch seconds value:

```
use Time::Local;  
$then = time() + 4*86400;  
$then = timegm localtime $then;  
# local epoch seconds  
$then -= $then % 86400;  
# truncate to the day
```

Date Manipulation

By Nathan Torkington

```
$then = timelocal gmtime $then;  
# back to gmt epoch seconds  
print scalar localtime $then, "\n";
```

So far you've been manipulating distinct hour, day, year, etc. values, as well as epoch seconds values. But the world presents you dates and times as strings. How do you go from a string to an epoch seconds value? One way is to write a small custom parser. This has the advantage of being flexible and fast:

```
use Time::Local;  
@months{qw(Jan Feb Mar Apr May Jun  
Jul Aug Sep Oct Nov Dec)} = (0..11);  
$_ = "19 Dec 1997 15:30:02";  
/(\d\d)\s+(\w+)\s+(\d+)\s+(\d+):(\d+):(\d+)/  
or die "Not a date";  
$mday = $1;  
$mon = exists($months{$2}) ? $months{$2} : die  
"Bad month";  
$year = $3 — 1900;  
($h, $m, $s) = ($4, $5, $6);  
$epoch_seconds = timelocal($s,$m,$h,$mday,$mon,$year);
```

A more general solution, however, is to install the Date::Manip module from CPAN.

```
use Date::Manip;  
$epoch_seconds = UnixDate("19 Dec 1997 15:30:02", "s");
```

Be warned, however, that Date::Manip is a large module and will increase the starting time of your program. One reason to accept the startup penalty is because Date::Manip parses dates in all sorts of different and interesting formats:

```
"today"  
"now"  
"first sunday in april 2000"  
"3:15, today"  
"3:15pm, first sunday in april 2000"  
"2000/01/18 09:15"
```

Common Date/Time Operations 2036, 2037, 2038, ..., 1901?!

Most C programs store an epoch seconds value in a signed integer, which means you can have positive and negative dates. But computer memory cannot represent an integer of

Date Manipulation

By Nathan Torkington

infinite size; hence we only have so many bits in which to represent seconds. This means that there are limits on the date calculations we can do with epoch seconds values.

The precise limitations depend on how many bits your machine will give to such an integer. Perl tries hard to store integer values in 32 bits. Crudely speaking, one bit will be used to indicate positive or negative, so we're down to 31 bits for the number. The biggest number you can store in 8 bits is 255, which is 2 to the power of 8, less one. So here's how Perl can tell us the biggest number we can store in a signed 32-bit integer:

```
print 2**31-1, "\n";  
2147483647
```

What date does this correspond to?

```
print scalar(gmtime 2**31-1), "\n";  
Tue Jan 19 03:14:07 2038
```

What happens one second after then?

```
print scalar(gmtime 2**31), "\n";  
Fri Dec 13 20:45:52 1901
```

Whoa! What happened? 2^{31} is too big for a signed 32-bit integer. It "wraps around," setting the negative bit, and becoming the largest negative number that's representable. This corresponds to the most number of seconds before the 1970 epoch. What is the end result of this? You can't store dates before `gmtime(2**31)` or after `gmtime(2**31-1)` as epoch seconds.

Don't reach for the cyanide pill just yet, though — it's not a major problem. If you need to work on dates before or after the boundaries of signed 32-bit epoch seconds, simply change your representation. There are plenty of modules on CPAN that work with dates: `Date::Calc` and `Date::Manip` are probably the two most powerful.

Both modules use their own date representations, and so avoid the Y1901-Y2038 restrictions. `Date::Manip` uses the Gregorian calendar, extrapolated back to the year 0000 AD and forward to the year 9999AD. `Date::Calc` also uses the Gregorian calendar and can work with years from 1 to at least 32767.

Making Sense of the World

Various countries adopted the Gregorian calendar at different times (you've probably heard the story of the missing fortnight in England in 1752 caused by the change from Julian to Gregorian calendars), and you should be aware of which calendar your date assumes. Taking a date derived from the Gregorian calendar and working with it as though it were a Julian date is not going to give accurate results. However, this rarely turns out to be

Date Manipulation

By Nathan Torkington

something that people other than astronomers and astrologers care about (Shakespeare was born on a Thursday? Or was it a Saturday? Was Pluto ascending Uranus?).

The 32 bit limit does have wider ramifications than those associated with Perl programs. Inside every Unix machine is an integer being incremented once each second, counting seconds since the epoch. If this value, as it is on many Unices, is kept in a 32-bit signed integer, the system will experience severe time confusion in 2038 when the rollover occurs. Most operating systems are expected to fix this well before 2038.

For dates and times in the range 1902-2037, storing them as the number of seconds before or after the epoch (midnight January 1, 1970 GMT) lets you use integer arithmetic, the built-in `gmtime()` and `localtime()` functions, and the standard `Time::Local` module. To calculate dates outside that range, or to parse dates in odd formats, try the `Date::Manip` and `Date::Calc` modules from CPAN.