ADAM TUROFF

practical Perl

DATE AND TIME FORMATTING IN PERL

Adam is a consultant who specializes in using Perl to manage big data. He is a long-time Perl Monger, a technical editor for *The Perl Review*, and a frequent presenter at Perl conferences.

ziggy@panix.com

DEALING WITH DATES AND TIMES IS A common source of needless errors. The brute-force methods of dealing with dates tend to ignore the many little details that are easy to forget. Thankfully, there are better alternatives. Using modules like POSIX or DateTime not only makes date-handling code easier to manage, but it also makes programs much more featureful and robust.

Date handling is one of those topics that is easily overlooked in many programs. The vast majority of programs I have written over the years do not need to deal with dates and times. The most common use of dates and times is simply informative, like putting a timestamp on entries in a log file:

```
#!/usr/bin/perl -w

use strict;

## Method 1: peppering print statements about
print STDERR "[" . localtime() . "] - process ".
    "starting\n";
## ... do stuff ...
print STDERR "[" . localtime() . "] - process ".
    "complete\n";

## Method 2: use a logging function
sub logmsg ($) {
    my $msg = shift;
    my $time = localtime();
    print STDERR "[$time] $msg\n";
}
```

Using localtime() to grab the current time is common because it's easy to use and its behavior is so simple. In order to work properly, Perl assumes a lot of context so that it can do the right thing. First, when the localtime() built-in function is called with no parameters, it assumes that you want to get the time right now and operates on the value that would be supplied by time(), a value representing the number of seconds since the beginning of the UNIX epoch.

The second piece of context here is how localtime() is used. Depending on how it is called, this function will produce either a single scalar value (a timestamp string) or a list of date-time components (seconds, minutes, hours, etc.). In the instances above, the output of localtime() is concatenated into a string, so it is used in a scalar context and would produce output like this:

```
[Fri Mar 25 12:35:28 2005] - process starting
[Fri Mar 25 12:48:02 2005] - process complete
```

Some common uses of date information are a little more involved. For example, I might want to express

the current date in YYYY-MM-DD format for archiving log files. In a shell script this is fairly trivial to do, with the date(1) utility:

```
#!/bin/sh
cd $APPHOME/logs
mv app.log app.log.'date +%Y-%m-%d'
```

In Perl, this kind of date formatting is possible, but a little more involved. To start, localtime() needs to be called in list context to convert UNIX epoch time into values such as year, month, and day:

```
#!/usr/bin/perl -w

use strict;

my ($sec, $min, $hour, $day, $month, $year, $wday, $yday, $dst)
   = localtime();
```

When trying to retrieve just date information, we can ignore the unnecessary values and focus on the year, month, and day values by using an array slice:

```
#!/usr/bin/perl -w

use strict;

my ($day, $month, $year) = (localtime())[3..5];
```

While localtime() does provide values for month and year, it mimics the format returned by the standard C library functions. Month values fall in the range 0..11, and years are the actual year minus 1900. In order to produce sensible values from localtime(), these values must be adjusted after each and every call:

```
my ($day, $month, $year) = (localtime())[3..5];
$month++;
$year+=1900;

print "$year-$month-$day"; ## format as YYYY-MM-DD
```

However, even this isn't quite correct. In order to produce a two-digit month, these values must be formatted using a function such as sprintf or printf:

```
my ($day, $month, $year) = (localtime())[3..5];

## format YYYY-MM-DD properly
printf ("%04d-%02d-%02d", $year+1900, $month+1, $day);
```

Clearly, this is a lot of work in order to do something that should be easy.

Formatting with the POSIX Module

These issues are typical of the kinds of small details that pervade handling dates and times. Thankfully, correct date and time formatting is a solved problem. C programmers may remember the strftime(3) function for handling this problem. A version of this function is available by default in Perl and is provided in the POSIX module. (This same behavior is exposed in the shell through the date(1) utility.)

Perl's POSIX::strftime() function takes a date format string as its first argument and a series of time components (seconds, minutes, . . . year, etc.) to produce a formatted date-time value. Fortunately, the order of the time values that strftime() expects is precisely the order of values that localtime() produces. Therefore, producing a date formatted as YYYY-MM-DD is as simple as:

```
#!/usr/bin/perl -w

use strict;
use POSIX qw(strftime);

print strftime("%Y-%m-%d", localtime()), "\n";
```

(The meaning of the formatting specifiers used in the first argument is described in the strftime(3) man page.)

Another common requirement for producing date values is to use names for months and days of the week. Frequently, programs that need to do this contain an array with the relevant names:

```
my @months;
$months[0] = "January";
$months[1] = "February";
##....
$months[11] = "December";

## Or, more succinctly:
my @months = qw(January February ... December);
```

Sadly, this is an antipattern common among programmers who do not deal with dates and times on a regular basis—that is to say, most programmers. I know I've done this more times than I care to count, and every time I feel guilty. The problem here isn't that defining an array of month or day names is necessarily wrong or bad, but it is needlessly repetitive.

Instead of redefining these lookup tables in each and every script that needs them (or, better, redefining them once in a module), why not just use the lookup tables that are already predefined in the standard C library? Here are some common formats, available through POSIX::strftime():

```
#!/usr/bin/perl -w

use strict;
use POSIX qw(strftime);

## Friday, March 25, 2005
print strftime("%A, %B %m, %Y", localtime()), "\n";

## Fri, Mar 25, 2005
print strftime("%a, %b %m, %Y", localtime()), "\n";
```

Creating Dates and Times POSIX-Style

Formatting times can be a tricky business, but not as tricky as performing arithmetic on dates. All UNIX date handling is ultimately done in terms of seconds since January 1, 1970, and the time() built-in function returns the current number of seconds since the start of the UNIX epoch. Figuring out the count at midnight this morning,or midnight tomorrow morning should be a simple process of adding and subtracting seconds from the current time. (The output of localtime() in list context can tell us how many hours, minutes, and seconds to fill in the missing pieces.)

For example, determining the time a few days in the past or future is just a matter of adding or subtracting multiples of the value 86,400 (that is, 24∗60∗60). While this usually works, this brute-force solution isn't quite accurate. In most time zones, there is one day a year that has 23 hours, and another that has 25 hours, marking the switch to and from Daylight Savings Time. Periodically, 86,400 seconds ago could still be "today," or it could be "two days ago." A milder version of this bug occurs when "three days after 9 a.m. Friday morning" becomes Monday morning at 8 a.m., 9 a.m., or 10 a.m., depending on the week.

There are other complications to this method. How do you obtain the time value for the beginning of next month? How do you add three months to a specific date? How do you determine "three weeks ago"?

The simple solution is to use the mktime() function, also found in the POSIX module. This function takes the same series of time components returned by localtime() and expected by strftime() and returns the corresponding epoch time. That is, the same caveats about month values being in the range 0..11 and year values being year – 1900 still apply to the inputs to mktime().

```
#!/usr/bin/perl -w

use strict;
use POSIX qw(mktime strftime);

## Print a timestamp for the start of 1999

print scalar(localtime(mktime(0,0,0,1,0,99))), "\n";
```

Fortunately, the values processed by mktime are not strictly limited in range. That is, mktime expects days to start at 1, seconds, minutes, hours, and months to start at 0, and so on. To ask for the time at one second before midnight midway through 2010, simply adjust the inputs accordingly:

```
##         sec min hr day mon year
print scalar(localtime(mktime(-1, 0, 0,183, 0, 110))), "\n";
```

Similarly, if I want to know what the epoch time was three weeks ago or will be three weeks hence, I can add or subtract 21 days to the current day value returned from localtime():

```
my @now = localtime();  ## get the current [sec, min, ...] values

my @past = @now;
$past[3] -= 21;  ## same time, 3 weeks ago

my @future = @now;
$future[3] += 21; ## same time, 3 weeks from now

## Print out all three dates, in chronological order
print scalar(localtime(mktime(@past))), "\n";
print scalar(localtime(mktime(@now))), "\n";
print scalar(localtime(mktime(@future))), "\n";

## Output:
Fri Mar 4 12:52:23 2005
Fri Mar 25 12:52:23 2005
Fri Apr 15 13:52:23 2005
```

(Note the switch from standard time to daylight savings time between March 25 and April 15.)

Date Handling with the DateTime Modules

For casual uses, time(), localtime(), POSIX::mktime(), and POSIX::strftime() can be used in conjunction to solve simple problems of creating and formatting time values. But there are still other problems that frequently arise when dealing with dates. One limitation of localtime() and strftime() is that they only work in the current time zone, whatever that may be. If you need to format the current time for a user in another time zone, things start to get tricky.

Thankfully, these issues are easily solved with the DateTime family of modules. As an added bonus, DateTime does away with the silliness of years being represented as "year – 1900" and months falling in the range 0..11. Here is an example of how to construct a new DateTime object that represents a single point in time:

```
#!/usr/bin/perl -w

use strict;
use DateTime;

## Construct an object at a fixed point in time
my $date = new DateTime (
    year => 2005,
    month => 1,  ## 1..12
    day => 1,
    hour => 12,  ## 0..23
    minute => 30,
    time_zone => "America/New_York"
);
```

```
## Construct an object for the current time
my $now = DateTime->now->set_time_zone("America/New_York");
```

The DateTime module handles a lot of details with dates and times, but it does not assume what the current time zone might be. For best results, a time zone should be specified whenever constructing a DateTime object. The time zone names that DateTime recognizes are the same ones that are found in the Olsen database, a public database of all time-zone information. (This is also the source data that is used to build the files in /usr/share/zoneinfo.) A DateTime object that is constructed without a time zone is constructed in the GMT time zone; specifying a time zone adjusts the component values accordingly.

DateTime objects can be formatted using the strftime() method, which accepts the same format strings as the POSIX::strftime() function. Because a DateTime object represents a fixed point in time, adjusting the time zone adjusts the formatted representation as expected:

```
my $now = DateTime->now->set_time_zone("America/New_York");
print $now->strftime("%c");  ## prints 'Mar 25, 2005 12:52:23 PM'

## Same time, different time zones
$now->set_time_zone("America/Los_Angeles");
print $now->strftime("%c");  ## prints 'Mar 25, 2005 9:52:23 AM'

$now->set_time_zone("Europe/London");
print $now->strftime("%c");  ## prints 'Mar 25, 2005 5:52:23 PM'
```

DateTime also handles many localization issues. For example, in French, not only are the names of the days and months different, but the standard date formats are different. Taking the same time value, we can display that time in Paris for an American viewer, a British viewer, and a French viewer. To switch the localization of a date, simply update the locale on that DateTime object:

```
## Convert to Paris time
$now->set_time_zone("Europe/Paris");

## Display, using the default (US English) localization
print $now->strftime("%c");  ## 'Mar 25, 2005 6:52:23 PM'

## Convert to a British localization
$now->set_locale("en_GB");
print $now->strftime("%c");  ## '25 Mar 2005 18:52:23'

## Convert to a French localization
$now->set_locale("fr");
print $now->strftime("%c");  ## '25 mars 05 18:52:23'
```

The vagaries of time-zone arithmetic are handled through the DateTime::TimeZone family of modules. Each of these modules define the offset from GMT and the rules for switching to and from Daylight Savings Time. The DateTime::Locale modules define the localization interfaces, and include data such as the native formats for dates and the names of the days and months. Both of these modules are installed with DateTime.

The Perl DateTime project has also built many other extensions to the core DateTime modules. Some of these modules provide calendar handling, formatting and parsing of dates, date calculations, date spans, and many other features. Sadly, the features provided by these modules are beyond the scope of this article; for more information, please visit http://datetime.perl.org.

Conclusion

Date and time handling is an area that does not get a lot of attention in many Perl programs. Using the simple and obvious brute-force techniques is actually quite complicated and very error-prone. Using a standardized library to handle dates makes the process easy and robust, whether you areusing the standard POSIX module or the DateTime modules from CPAN.