

The Date Class, Part 3

by Jeffrey Copeland and Haemer

Last month, we delayed building our own version of `strftime()` because we had discovered bugs in everyone else's versions. We wanted to establish the right method before generating yet another wrong one. This month, we'll attempt to build a correct version of `strftime()`.

`strftime()` is the standard C function to format dates. You give `strftime()` a date and format specifier and it returns the formatted date as a string. For example,

```
time_t t = time(NULL);
strftime(s, strlen(s)+1, "%H:%M:%S",
        localtime(&t));
puts(s);
```

will produce the same result as

```
date '+%H:%M:%S'
```

For us, in particular, `strftime()` has some fascinating aspects:

1. It is standard—we've written a series on standards.
2. It is specifically designed for international applications—we've written a series on internationalization.
3. It is what's left of the corner we painted ourselves into last time.

In fact, it's so fascinating that we're going to take a whole column to show you how to write a routine that you'll almost never call (`date` is almost always good enough), that you don't need to write because it's supplied on your system, and that is so seldom used that most of the vendor-supplied versions have easily discovered bugs that no one has complained about yet.

You don't need it even if you have to format dates and don't have

Jeffrey Copeland (copeland@alumni.caltech.edu) is a member of the technical staff at QMS's languages group, in Boulder, CO. His recent adventures include internationalizing a large sales and manufacturing system and providing software services to the administrators of the 1993 and 1994 Hugo awards. His research interests include internationalization, typesetting, cats and children.

Jeffrey S. Haemer (jsh@canary.com) is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting and internationalization. Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.

`strftime()`. For example, here's another way to produce the same output from a C program:

```
system("date '+%H:%M:%S'");
```

Why go through all the work of creating a library function, especially when you can use the `date` command to do the same thing? Exercise for the reader: Why does this function exist? Or asked another way, for what specific application is this technique completely out of the question?

Hello. Doze off for a minute there, did you? We need to devise a way to motivate you to slog through this.

Here's what we're going to do. We'll show you some interesting tricks you can use to solve a relatively boring problem. Getting there will be more than half the fun.

If you read this and we succeed, please let us know.

Pounding Square Wheels into Round Ruts

We hate having to reinvent the wheel. Whenever we need a routine to do something unusual, we begin by searching our own database, using `man -k`, or its synonym, `apropos`. These commands perform keyword searches through `man` page synopses and often find routines that do just what we're looking for. Both work from a database that we rebuild frequently with `catman`. Our `crontab` entry is

```
15 2 * * 0,4 /bin/catman -w.
```

We can't always find what we want, so the next step is to search the Net. Sometimes, the routines we find aren't in the language that we require them to be. We'll use `strftime()` to show how we proceed in this situation.

The Comprehensive Perl Archive Network (CPAN) is a treasure trove of routines that is growing faster than you can invent things to look for. Of course, they're all in Perl. Check out <http://www.pascal.org/Family/Freeman/CPAN.html> for pointers to master and mirror sites.

A search at our nearest mirror site quickly turns up a package that implements `strftime()`, called `ptime.pl` by Paul Foley at Cambridge, MA-based Acsent Technology. The package includes two other routines—`asctime()` and `ctime()`, but we won't attempt to rewrite these.

We could extract the code for `sub strftime` and do a translation, reverting to the original when necessary to pick up missing pieces. But often when two things are related, it's a good idea to keep them together so they don't get out of sync. Time for a trick.

Consider the following code fragment:

```
#ifdef PERL
print "Hello world\n";
=comment
```

```
#endif /* PERL */
#include <stdio.h>
main(void) {
    printf ("Hello, world\n");
    exit(0);
}
#ifdef PERL
=cut
exit 0;
#endif /* PERL */
```

Here we have both legal C *and* legal Perl. Note: Throughout this column, we'll often use C to mean C or C++. Sometimes this is technically wrong, but we don't think it will be confusing and it saves us a lot of typing. Let's walk through it both ways.

First, C. The commands `#ifdef PERL ... #endif /* PERL */` hide all the Perl code from the C compiler. By the time `cpp` is done with this code, it reduces it to the following:

```
#include <stdio.h>
main(void) {
    printf ("Hello, world\n");
    exit(0);
}
```

Next, Perl. Perl author Larry Wall has put a piece of syntax into Perl 5 that lets us hide the C in an analogous way. Any statement that begins with `=word` (where `word` is any word at all) is treated by the Perl compiler as a signal to stop processing until it finishes another line that begins with `=cut`.

This makes it easy to intermix code and documentation. The Perl 5 distribution comes with the `man` page, `perlpod(1)`, which describes the semantics of a suite of formatting directives, such as `=head1` and `=item`. These can be used to embed `man` pages into Perl modules.

Note: *Pod* stands for "plain old documentation," an apparent tip-of-the-hat to the ISDN phone folks who refer to normal phone service as POTS—plain old telephone service. Many of the modules in Perl 5 have embedded `pod` directives, and the distribution comes with four translators—`pod2html`, `pod2latex`, `pod2man` and `pod2text`—which can interpret these directives to produce documentation in four popular formats.

We're not that ambitious. We just use `=comment` and `=cut` to bracket C code. In our case, the Perl compiler reads the following:

```
#ifdef PERL
print "Hello world\n";
exit 0;
#endif /* PERL */
```

And because Perl's comment character is #, the #ifdef PERL and #endif /* PERL */ pair is ignored.

Now we'll embed our C translation of Foley's module in the original. Depending on your viewpoint, either the C becomes a comment to help make the Perl more readable, or the Perl is pseudo-code documenting the design of the C implementation.

Designing strftime()

Foley's package is nicely laid out. He defines a few tables and a handful of utility routines and then lets the table lookups and Perl's rich regular expressions do most of the work. C doesn't have Perl's regular expressions, but we'll follow the same basic design.

First, we look at the tables. It's almost no work to translate these:

```
#ifdef PERL
=cut
@DoW = ('Sun', 'Mon', 'Tue', 'Wed', 'Thu',
        'Fri', 'Sat');
@DayOfWeek = ('Sunday', 'Monday', 'Tuesday',
              'Wednesday',
              'Thursday', 'Friday', 'Saturday');
@MoY = ('Jan', 'Feb', 'Mar', 'Apr', 'May',
```

```
'Jun', 'Jul', 'Aug', 'Sep', 'Oct',
        'Nov', 'Dec');
@MonthOfYear = ('January', 'February',
               'March', 'April', 'May', 'June',
               'July', 'August', 'September',
               'October', 'November', 'December');
=comment
#endif /* PERL */
static char *DoW[] = { "Sun", "Mon", "Tue",
                      "Wed", "Thu", "Fri", "Sat" };
static char *DayOfWeek[] = {
    "Sunday", "Monday",
    "Tuesday", "Wednesday",
    "Thursday", "Friday", "Saturday" };
static char *MoY[] = { "Jan", "Feb",
                      "Mar", "Apr", "May", "Jun",
                      "Jul", "Aug", "Sep", "Oct",
                      "Nov", "Dec" };
static char *MonthOfYear[] = {
    "January", "February", "March",
    "April", "May", "June",
    "July", "August", "September",
    "October", "November", "December" };
};
```

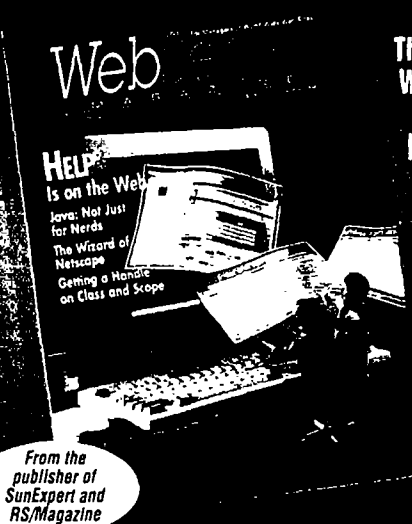
Let's skip over the utility routines for a second and go

Subscribe Today!

For Managers of World Wide Web Sites

If your job function includes one of the following:

- Manager of World Wide Web Site
- World Wide Web Site System Administrator
- World Wide Web Site Software Developer • Networking Specialist



Web

HELP
Is on the Web

Java: Not Just
for Nerds


The Wizard of
Netscape

Getting a Handle
on Class and Scope

The Corporate
Web Site and
Intranet
Publication

From the
publisher of
SunExpert and
RS/Magazine

You may qualify for a FREE SUBSCRIPTION to
WebServer Magazine by filling out the Web subscription form at:
<http://www.cpg.com/ws/subscribe.html>
To ADVERTISE in the FEBRUARY ISSUE call
(617) 739-7001 for your area sales representative



removes
the uncertainty from
open system security.


BUSINESS

With **Stalker**, you know who did **what, when and how...** and you have the evidence to prove it.

Stalker actively monitors your **entire** UNIX network, identifies external and internal misuse, and automatically reports directly to you via email or printed report.


For more information and a free white paper on UNIX system security, call us or visit our Web site today!

<http://www.haystack.com>



Haystack Labs, Inc.

10713 Hwy 620 North Austin, Texas 78726
512.918.3555 Fax 512.918.1265



**Active Security for
Open Systems**

Circle No. 13 on Inquiry Card

directly to `sub strftime`. Here's the prototype for the standard C function:

```
size_t strftime(char *s, size_t maxsize,
               const char *format,
               const struct tm *timeptr);
```

Note: In `ptime.pl` this function becomes `# strftime ($template, @time_struct)`. `$template` is the format string and `@time_struct` is an array whose elements correspond to the fields of C's `struct tm`. For example,

```
($sec, $min, $hour, $mday, $mon,
 $year, $yday, $yday, $isdst) = @time_struct;
```

instead of

```
struct tm {
    int tm_sec;
    int tm_min;
    int tm_hour;
    int tm_mday;
    int tm_mon;
    int tm_year;
    int tm_wday;
    int tm_yday;
    int tm_isdst;
    ...
}
```

Inside the routine, most of the code represents the assignments of values to variables with the same names as the field descriptors.

For example `strftime()` and `date` use `%A` to designate the day of the week, and `%I` for the hour on a 12-hour clock.

Here's what you see inside `sub strftime`:

```
# the full weekday name
local($A) = $DayOfWeek[$yday];
[...]
local($I) = sprintf("%2d", $hour % 12);
$I =~ y/ /0/; # hour of 12hr clock
$I = 12 if $I == 0;
```

To parallel this in C, we build an array of 256 elements, indexed by the format characters. Most of the elements will be `NULL` pointers, but legal format characters will have pointers to string representations of the corresponding information. Here's what the same code translates into:

```
// the full weekday name
dstr['A'] = dcpy(DayOfWeek[tp->tm_wday]);
int hr = tp->tm_hour % 12;
if (hr == 0) hr = 12;
// hour of 12hr clock
dstr['I'] = dprintf("%2d", hr);
b2z(dstr['I']);
```

We'd like to use this code to make a few points before going any further:

1. Perl makes copies of strings for you, allocating space as necessary. We have to do all this by hand and write a handful of utility routines to ease the workload.

One, `dcpy()`, creates a `malloc'd` copy of the string you pass it. Another, `dprintf()`, does a `sprintf()` into a string that it creates on the fly.

2. Perl scripts often rely heavily on Perl's typeless scalars. Programmers don't need to take special steps to convert numbers to strings, so no one worries about how a number is being represented. C doesn't have that feature, so we've chosen to keep track by converting everything to strings.

In the above code, we do all the arithmetic operations on `tp->tm_hour` first, then we convert the result to a string before storing it.

3. String substitutions have to be hand-crafted in C. Perl has built-in operators to handle this. Fortunately, nearly all of the string substitution in `strftime` is done to change one-digit numbers to two-digit numbers. For example,

```
$I =~ y/ /0/;
```

will turn numbers like "9" to "09." For our purposes, we wrote a utility routine, `b2z()`, that does nothing more than change a leading blank to zero.

4. Most of our `strftime()` function is straight-line code that we could have written in vanilla C. Our code isn't object-oriented, but we've still used a couple of C++ features. One feature was to use C++ comments to turn the following:

```
# the full weekday name
local($A) = $DayOfWeek[$yday];

into this:

// the full weekday name
dstr['A'] = dcpy(DayOfWeek[tp->tm_wday]);
```

Another was the freedom to declare variables where we first use them, which often makes things easier to read. For example,

```
int hr = tp->tm_hour % 12;
if (hr == 0) hr = 12;
// hour of 12hr clock
dstr['I'] = dprintf("%2d", hr);
b2z(dstr['I']);
```

Taking Out the Garbage

Another noteworthy issue that we have to deal with by hand in C is that of freeing up heap storage. When we say,

```
// the full weekday name
dstr['A'] = dcpy(DayOfWeek[tp->tm_wday]);
```

we've allocated several bytes on the heap to hold a copy of the weekday name. Analogous code in Perl, for example,

```
# the full weekday name
local($A) = $DayOfWeek[$swday];
```

puts data into the heap but gives it a reference count, so that when it's no longer in use, the storage is freed up. This works like files in UNIX. UNIX frees up storage used by a file as soon as there are no longer references to the file in the directory hierarchy, and once the last process using it has closed the file.

In C, we need to free up heap storage by hand. In `strftime()`, we do this by freeing up everything in `dstr[]` before we exit. For example,

```
// Now free everything
int i;
for (i=0; i<256; i++)
    if (cp = dstr[i]) {
        free(cp);
        dstr[i] = NULL;
    }
```

In a more complicated conversion, we might either free every `malloc'd` pointer or keep a separate array of pointers to allocated space, instead of making a single array do double duty.

Parsing the Format String

The remaining big task is to parse the format string and substitute in the right values. In Perl, it's a four-liner:

```
$template =~ s/\\\/\200/g; # hide '\\'
# replace each control sequence with
# value of corresponding variable name
# (we've done it in two lines for
# typesetting purposes --- it's really one)
$template =~ s/\\([aAbCdHIjMm])/"\${$1}"/eeg;
$template =~ s/\\([pSUwXxYyZ])/"\${$1}"/eeg;
# restore '\\' as single '\'
$template =~ s/\/200/\/g;
```

In C, there's more to do, but it's not too bad:

```
char *cp;
char *cq;
int space = maxsize;

memset(s, 0, space);
for (cp = (char *) format;
     *cp && (space > 0); cp++) {
    if (*cp == '%' &&
        (cq = dstr[(cp+1)]) != NULL) {
```

```
        if ((space -= strlen(cq)) < 0) return 0;
        strcat(s, cq);
        cp++;
        continue;
    }
    *(s+maxsize-space) = *cp;
    if (--space < 0) return 0;
}
```

Notice that we have to check at every step to make sure we have enough space. If we run out of space, we return zero, which is what the standard says we should do, and the resulting string may be a mess—the standard says its contents are undefined.

Testing the Final Code

We once worked on a project where the manager ended every team meeting with “Remember, don't ship it until it compiles.” When we finished converting the code, we decided to try it out.

First, we wrote a driver that exercises every required format option. Next, we surrounded our `strftime()` function with a `#ifdef LOCAL_STRFTIME / #endif` pair, and wrote a script that would build and run the program with either the standard `strftime()` or our homegrown version.

Last, we compared the outputs. We found several interesting differences:

1. Bugs in our code. Yes, bugs.
2. Optional behavior in the specification. The standard says that `%x`, `%X` and `%c` should all print out a locale's “appropriate representation.” For `%x`, our code prints Nov 11 1918, where the vendor's version prints 11/11/18.
3. Trivial bugs in `ptime.pl`. The specification requires that `%j`, day of the year, produce a number from 001 to 366. However, `ptime.pl`'s days run from 000 to 365, just like `tm_yday`.
4. Bigger bugs in `ptime.pl`. If you've read the previous two columns in this series, you'll remember we said that implementation errors in the `%U` and `%W` format specifiers are ubiquitous. Well, the implementation in `ptime.pl` was wrong too. All of these problems were easy to fix. To correct the last problem, we just stole code from our earlier columns.

So far, we've shown you an approach to implementing `strftime()` that has several interesting features:

- We've converted a freely available Perl package into a useful C program.
- We've left the Perl code in as documentation.
- We've shown you how to use one implementation to find bugs in another.

Next time, we'll show you our version of the full code. We encourage you to try your hand and compare it to ours. Until then, have a fine November. ▲