To Do or Not to Do

by Jeffreys Copeland and Haemer

When Kenneth Branagh's movie version of *Henry V* was released, the distributor had phone calls from folks wanting to know where they could get the first four movies in the series on videotape. In that vein, welcome to "Diaries: The Sequel."

Last month, we covered the daily diary—that is, a record of events that have already happened. This month, we'll be discussing daily to-do lists—events that need to happen—in a little more detail. We'll cover how our to-do lists can become diary entries.

Once again, we are presenting software to manage data in a flat ASCII file because we believe that a configurable command-line interface to our information is more useful than a strictly graphical interface. For an interesting discussion about the trade-offs between fixed interfaces and programmable ones, in particular for library and patent searches, see *Risks Digest*, Volume 17, Number 5 and following issues, beginning with Jerry Leichter's submission "Re: Errors in patent databases." (Back issues of *Risks* are available by anonymous ftp at `ftp.unix.sri.com` in directory `risks`.)

Creating a To-Do List

As we discussed at the end of last month's column, we begin creating our to-do list with a directory containing our lists of daily, monthly,

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.*

weekly and fixed events. For example, we can have a file daily containing things we need to do each day:

```
process email
take out trash
```

and a file weekly containing tasks to do each week:

```
Monday
       status report
```

and a file monthly containing tasks to do at a fixed time each month:

```
01
       send out monthly billing
```

and an events file containing one-time events:

```
10/09/95
       Canadian Thanksgiving
```

Notice that we've said 10/09/95, not 10/9/95—this is significant, and we'll explain why later.

We want our todo script to run once a day, draw from each of the files of regular and one-time events, collect the uncompleted tasks from yesterday's list and provide us with a list of today's events and tasks in a new file.

We begin by reusing a trick from last month: building a routine named getdateinfo to get information from the date command. As you know, date takes arguments to return different aspects of the date and time: For example, date +%D prints 10/3/95.

We invoke date from Perl, with all upper case and lowercase arguments, separated by pound signs: date +#%A#%B#%C...#%Z#%a...#%z. We split the results and put them into an associative array, strftime. Why strftime? Because it matches the name of the POSIX.1 routine that generates formatted time data.

```
# put all date information into %strftime
# $strftime{"X"} has the value returned
# by "date +%X"
sub getdateinfo {
    local($lower, $upper,
        $l, $alphabet);
    local(@tmp);

    for ($l = "a"; 1; $l++)
    {
        $lower .= "#%$l";
        last if $l eq "z";
    }
    $upper = $lower;
    $upper =~ tr/a-z+/A-Z/d;
    $alphabet = $lower . $upper;
    @alphind = split(/#%/, $alphabet);

    @dateinfo = split('#',
        `date "+$alphabet"`);
    @tmp = @dateinfo;
    foreach (@alphind) {
        $strftime{$_} = shift(@tmp);
    }
}
```

Note that in Perl 5, this same functionality is available in a different form by using the POSIX.pm module and calling its strftime() function. We still find this trick cleaner—strftime() takes seven arguments—and we suspect it's also faster. (Exercise for the reader: Write a program using our timing tool from a few columns back [see "Who Lives Near Here?" September 1995, Page 30] to test the question.)

Next, using the data in strftime, we capture today's and yesterday's dates:

```
# September 3, 1995 returned
# as "1995/246" (Julian date)
sub getdate {
    local($spec) = @_;
    local($jd) = $strftime{"j"};
    local($year) = $strftime{"Y"};

    if ($spec eq "yesterday") {
        $jd = $jd - 1;
    } elsif ($spec eq "tomorrow") {
        $jd = $jd + 1;
    }

    if ($jd > 365) {
        $jd = &leap_year($year) ?
                $jd - 366 : $jd - 365;
        $year++;
    } elsif ($jd < 1) {
        $year--;
        $jd = &leap_year($year) ?
                366 - $jd: 365 - $jd;
    }

    return ("$year/$jd");
}
```

It would be helpful to define the leap_year subroutine at the same time:

```
# do the leap year calculation.
sub leap_year {
        local($year) = @_;
        return 1 if (($year%4 == 0)
              && ($year%100 != 0));
        return 1 if ($year%400 == 0);
        return 0;
}
```

Interesting fact: The Julian calendar has leap years that are divisible by four, except that years divisible by 100 are not leap years, except that years divisible by 400 are leap years. Thus, 1900 is not a leap year, but 2000 is. This keeps our seasons in sync with our calendar. Even so, we still need a leap second every now and again.

Now, as with the earlier "diary" program, where we get our files is pretty arbitrary, with a broad choice of how much flexibility to provide the user and where to allow defaults. In this case, though, we choose to file by Julian date—so events for 3 September 1995 would be filed in `$CALDIR/1995/246`. We create a function, getfilenames, to create the names. If, instead, you want to organize calendar files in the hierarchy like "Y/M/D," or go back to filing by week, change this function.

```
# get the names of all files
sub getfilenames {
    local($yesterday, $today) =
       (&getdate("yesterday"),
       &getdate("today"));
&debug("today is $today, yesterday was
        $yesterday");

    local($caldir) = $ENV{"CALDIR"} ?
       $ENV{"CALDIR"} :
       $ENV{"HOME"} . "/Calendar";
    $yeardir = "$caldir/" . $strftime{"Y"};
    unless (-d $yeardir) {
       system "mkdir -p $yeardir" ||
             die "can't mkdir $yeardir: $!";
    }

    @filenames =
       ($yesterday, $today,
       "daily", "weekly",
       "monthly", "events", "now");

    @filenames = grep($_ =
       "$caldir/$_", @filenames);
}
```

Next, we need a routine, getevents, to extract events from our files.

```
# read all relevant events from named file
sub getevents {
        local($filename, $which) = @_;
        local(@events, @conditions);
        local($ts, $te, $i);

        open(I, $filename) || return;
        $old_RS = $/;

        # get by paragraphs
        $/ = "";
        @events = <I>;
        $/ = $old_RS;

        # handle special case of $which eq "todo"
        if ($which eq "todo") {
            for ($i = 0; $i < @events; $i++) {
                $ts = $i if $events[$i] =~ /.tS/;
                $te = $i if $events[$i] =~ /.tE/;
            }
            if ($ts < $te) {
                @events = @events[$ts+1..$te-1];
            }
        }

        # now look to see which ones are relevant
        if (($which eq "all") ||
            ($which eq "todo")) {
            undef @conditions;
        } else {
            @conditions = @dateinfo;
        }
        @events = grep(&relevant($_, @conditions),
            @events);

        # filter individual events
        @events = grep($_ =
            &strip($_), @events);

        push(@events, "\n) if (@events);

        return @events;
}
```

Notice that getevents does a grep for strings in the strftime array. This is why we tag fixed events with 10/09 instead of 10/9—date returns a two-digit, zero-padded date.

We have two special cases in getevents: a second argument of all corresponds to no conditions at all; this is because every event is relevant. In this oversimplified version, without a second argument, we just look for any event whose first line matches some return from strftime.

A fancier version might understand arguments like "weekly" and construct a more appropriate @conditions array. The other special case is a second argument of todo, which extracts a to-do list, bounded by the appropriate macros, and then goes on to act as though it had been called as all.

Notice that this lets us add more file names to search later by returning a larger array, because Perl handles variable-sized array returns. Currently, getevents() just takes a filename and either gets relevant events or all events, based on the second argument.

We need a routine to define relevant events. Therefore, we construct an array of conditions and then screen for those conditions with relevant:

```
sub relevant {
   local($event, @conditions) = @_;
          # no conditions, it's relevant
   return 1 unless (@conditions);
   local(@event) = split(/\n/, $event);
   local($date) = shift(@event);
   @conditions = grep($date eq $_, @conditions);
          # some condition matched
   return 1 if (@conditions);
   return 0;
}
```

We also need a routine to strip out events that are already completed:

```
# strip events that are done
sub strip {
    local($event) = @_;
    return $event unless &done($event);
    local(@event) = split(/\n/, $event);
    local($head);
    while ($head = shift(@event)) {
          push(@nevent, $head)
                 unless &done($head);
    }
    $event = join("\n, @nevent) . "\n\n";
    return $event;
}
```

Why do we go to the trouble to strip out completed events? First, we assume that we're marking completed events. Then we scan yesterday's event file for non-marked events and add them to today's file. Thus, we pass unfinished events from day to day until we can mark them as complete. (Exercise: How do we mark items as complete? How should we handle partially complete items?)

We'll also need a routine to write out the events we've selected:

```
# spit all events into the named file
sub putevents {
    local($filename, @events) = @_;
    $old_AS = $";
    $" = "\n";
    open(O, ">>$filename") ||
        die "can't append to $filename: $!";
      # todo start
    print O "\n.tS\n\n";
    print O @events;
          # todo end
    print O ".tE\n\n";
    close O;
    $" = $old_AS;
}
```

Bringing It All Together

We're now set up, and we can prepare our last and all-encompassing utility routine:

```
sub todo {
  &getdateinfo();

  ($yesterday, $today, $daily,
  $weekly, $monthly, $events, $now) =
       &getfilenames();

&debug("($yesterday, $today, $daily, \
  $weekly, $monthly, $events, $now)");

  # look for undone events in:
       # specific events file
  push(@events, &getevents($events));
       # recurring day file
  push(@events, &getevents($daily, "all"));

       # recurring week file
  push(@events, &getevents($weekly));
       # recurring month file
  push(@events, &getevents($monthly));
       # yesterday's file
  push(@events,
       &getevents($yesterday, "todo"));

  # write out today's file
  &putevents($today, @events);

  # link to "now" file -- file used by other programs
&debug($today);
  unlink $now;
  link($today, $now)
}
```

Here, we cull the relevant events with getevents and then put them with putevents. We finish by unlinking the last events file, $now, and linking it to today's events file.

This makes the main program very, very simple:

```
&todo();
```

OK, we cheated a little: We're still missing two utility routines.

```
# is there something done here?
sub done {
    local($string) = @_;
    return ($string =~ /^\.DN/) ? 1 : 0;
}

# for debugging output, invoke as "command -debug"
sub debug {
    warn @_ if $debug;
}
```

The first routine, done, checks each event to see if it's not completed, as we discussed earlier.

The second of these two utilities warrants a little discussion. debug prints output conditionally if $debug is set. We set $debug by invoking todo with the -debug flag. If we invoke Perl with the -s flag, we automatically set $debug.

Another point to notice: We've put our &debug() calls at the left margin. This is personal style—it allows us to avoid interrupting the control flow and makes them easy to find. Similarly, in C code, we drop /*???*/ at the left margin on our debugging lines.

Now that we have a file, now, which is our list of prescheduled tasks for today, we can manually add the one-shot items we need to do today:

```
finish RS article
prepare Hugo nomination software for San Antonio
send pushd function to Chip Jarred
```

To Do for the To Dos

This leaves us with some loose ends. We need to write some vi macros to mark entries as complete or partially complete. We also need some vi assistance to add one-shots to now. We further need some troff macros to print out our to-do list.

Next time, we'll consider those macros. Until then, keep those cards and letters coming, folks. ▲