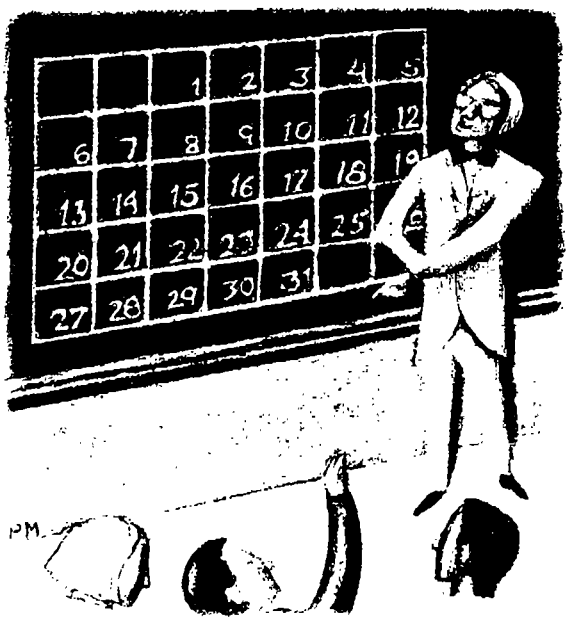


The Date Class, Part 4

by Jeffreys Copeland and Haemer

Last month, we described how we built our version of `strftime()` and promised to provide you with the code for it this month. We also promised we'd show you how that code interfaces to the `Date` class we've been building for months. And we'll, finally, abandon our flogging of the "date routines" dead horse.

But first...

Fan Mail from a Flounder

We had a long and thoughtful note from Stuart Gathman at Business Management Systems Inc. (`stuart@bmsi.com`) concerning our first column on the `Date` class (September 1996, Page 32). Stuart found an error in our code–our leap year routine ignores the 4,000-year glitch. That is, every 4,000th year is a non-leap year, even though every 400th is a leap year. He correctly points out that the existence of this rule is a tribute to the accuracy of 16th-century astronomy.

Our reply is that we can safely ignore the problem because our code will not be used 2,000 years from now, but, then again, that's what they said when they decided to use only two digits for years in all those COBOL programs written back in the 1960s. Folks in the insurance and banking industry estimate that the bill for fixing that problem is going to run into billions. Moral of the story: Get it right the first time.

Stuart also takes us to task over some style issues. In particular, he objects to our preference for successive approximation in converting

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.*

from absolute day number to a month/day/year triple. For example, he uses

```
dayofyear = (month*275)/9-32+day;
if (month < 3) dayofyear += 2;
```

to convert from month/day to day-of-year. We think it's a matter of style. We find our code easier to follow and revise, but that's our opinion. Meanwhile, take a look at the GNU version of `localtime.c`: Its underlying service routine, `offtime.c`, uses a successive approximation approach to get both the time and the date.

However, Stuart's comments also made us realize that we weren't entirely clear in our original article. The `Gregorian` derived class doesn't have a clearly defined starting date. This confused him, and probably the rest of you. It was intentional–no, not the confusion, the design.

The Gregorian calendar rules, like those for the Julian calendar, describe a method of calculating the progression of dates. We know that the calendars are defined so that October 5, 1582 on the Julian calendar is followed by October 15, 1582 on the Gregorian calendar. Our code doesn't worry about what the correct calendar is for the country or religious circumstance, it merely provides that progression backwards and forwards.

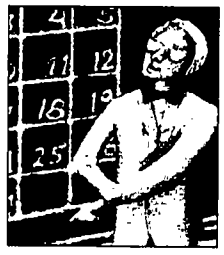
The code Stuart sent us, on the other hand, is wired for the actual Gregorian conversion. It uses the Julian rules up until 10/5/1582, skips to 10/15/1582, and uses the Gregorian rules from that time onwards.

In practical terms, no one is going to worry about using our Gregorian code for dates before 1582, and this means we don't need to tie our code to a particular country's conversion from Julian to Gregorian.

Last time, we promised to give you the whole listing of

The Whole Listing of strftime()

```
#include <sys/types.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

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"};

int wkyr(int wstart, int wday, int yday) {
  if (wstart == 0)
      return (yday - wday + 7)/7;
  else if (wstart == 1)
      return (yday - ((wday+6)%7) + 7)/7;
  else {
     fprintf(stderr, "%s\n",
        "first arg to wkyr must be \
0 (Sunday) or 1 (Monday)");
  }
}

static char *dstr[256];

char *dprintf(char *fmt, int n) {
    char *s;
    s = (char *) malloc(16);
    sprintf(s, fmt, n);
    return(s);
}

char *dcpy(char *s) {
    int n = strlen(s);
    char *t;
    t = (char *) malloc(n+1);
    strncpy(t, s, n+1);
    return(t);
}

char *b2z(char *s) {
    if (s[0] == ' ')
     s[0] = '0';
    return s;
}

#define LOCAL_STRFTIME
#define TEST
#ifdef LOCAL_STRFTIME
size_t strftime(char *s, size_t maxsize,
     const char *format,
     const struct tm *timeptr);

size_t strftime(char *s, size_t maxsize,
     const char *format,
     const struct tm *tp) {

        // the abbreviated weekday name
    dstr['a'] = dcpy(DoW[tp->tm_wday]);
```



Our code doesn't worry about what the correct calendar is for the country or religious circumstance, it merely provides that progression backwards and forwards.

our C++ translation of the Perl `strftime()` by Paul Foley at Ascent Technologies in Cambridge, MA. The Perl version is available at your favorite CPAN (Comprehensive Perl Archive Network) site–see `http://www.cpan.com` for mirror sites.

For space reasons, we're only providing the C++ version here; you'll need to refer back to last month's column, "The Date Class, Part 3," Page 31, for an explanation of what's going on. Send us email if you would like a complete interlineal translation.

So How Does It Fit into Date?

Last month, we promised to tell you how this code fits into the `Date` code. Basically, we generate a method for the `Gregorian` class that sets up an appropriate `struct tm` and feeds it to the `strftime()` routine we wrote last time.

For example,

```
void
Gregorian::_strftime(char *fmt, FILE *fp)
{
    char buf[BUFSIZ];
    Gregorian day1(1,1,year);
    struct tm tt;

    // the full weekday name
dstr['A'] = dcpy(DayOfWeek[tp->tm_wday]);
    // the abbreviated month name
dstr['b'] = dcpy(MoY[tp->tm_mon]);
    // the full month name
dstr['B'] = dcpy(MonthOfYear[tp->tm_mon]);
    // date
dstr['c'] = asctime(tp);
    // trim off newline
dstr['c'][strlen(dstr['c'])-1] = ' \0';
    // day of month
dstr['d'] = dprintf("%2d", tp->tm_mday);
  b2z(dstr['d']);
    // hour of 24hr clock
dstr['H'] = dprintf("%2d", tp->tm_hour);
  b2z(dstr['H']);
int hr = tp->tm_hour % 12;
if (hr == 0) hr = 12;
    // hour of 12hr clock
dstr['I'] = dprintf("%2d", hr);
  b2z(dstr['I']);
    // day of year
dstr['j'] = dprintf("%3d",
  tp->tm_yday + 1);
b2z(dstr['j']);
    // month number
dstr['m'] = dprintf("%2d", tp->tm_mon + 1);
  b2z(dstr['m']);
    // minutes after hour
dstr['M'] = dprintf("%2d", tp->tm_min);
  b2z(dstr['M']);
    // AM/PM indicator
dstr['p'] = dcpy(tp->tm_hour > 11 ?
  "PM" : "AM");
    // seconds after minute
dstr['S'] = dprintf("%2d",
  tp->tm_sec); b2z(dstr['S']);
    // Sunday of year
dstr['U'] = dprintf("%2d",
  wkyr(0, tp->tm_wday, tp->tm_yday));
    // day of week (Sun == 0)
dstr['w'] = dprintf("%2d", tp->tm_wday);
    // Monday of year
dstr['W'] = dprintf("%2d",
  wkyr(1, tp->tm_wday, tp->tm_yday));
int ysc = tp->tm_year % 100;
    // year since century
dstr['y'] = dprintf("%2d", ysc);
    // year
dstr['Y'] = dprintf("%4d",
  ysc + ((ysc < 70) ? 2000 : 1900));
dstr['x'] = (char *)malloc(12);
    // date
sprintf(dstr['x'], "%3s %2s %2s",
  dstr['b'], dstr['d'], dstr['Y']);
dstr['x'][11] = ' \0';
dstr['X'] = (char *)malloc(9);
    // time of day
sprintf(dstr['X'], "%2s:%2s:%2s",
  dstr['H'], dstr['M'], dstr['S']);
dstr['X'][8] = ' ';
    // timezone of machine
dstr['Z'] = dcpy(tp->tm_zone);
dstr['%'] = dcpy("%");

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

```
    // set time to something arbitrary
    tt.tm_hour = 12;
    tt.tm_min  = 0;
    tt.tm_sec  = 0;
    // extract the date information
    tt.tm_mday = this->day;
    tt.tm_mon  = this->month;
    tt.tm_year = this->year - 1970;
    tt.tm_wday = this->DayOfWeek();
    tt.tm_yday = this->date - day1.date + 1;
    Jstrftime(buf,BUFSIZ,fmt,&tt);
    fputs(buf,fp);
}
```

Exercise for the reader: Now provide an interface to `strftime()` for the `Julian` class.

In addition (thanks to Stuart Gathman), assuming we are willing to make a hard differentiation between Gregorian and Julian calendars–which we haven't done in our code–provide a `strftime()` that, in concert with the country information from the locale, knows when to make the switch from Julian to Gregorian calendars. Remember, that date will range from 1582 for Italy, to 1917 for Russia.

What have we left out? We never did a derived class for Hebrew or Arabic calendars. However, those of you who are interested can take a look at the Dershowitz and Reingold paper that started us on this calendar quest, in the September 1990 issue of *Software Practice and Experience*. Alternately, you can look up our excursions into Japanese and Arabic calendars in the midst of our POSIX series two years ago: see our columns in the August 1994 and September 1994 issues of *RS/Magazine*.

Last exercise for the reader: Generate `Hebrew` and `Arabic` derived classes from `Date`.

That's it until next time. Until then, happy holidays and a wonderful new year! ▲

```
    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;
    }

    // Now free everything
    int i;
    for (i=0; i<256; i++)
    if (cp = dstr[i]) {
      free(cp);
      dstr[i] = NULL;
    }
    return strlen(s);
}
#endif


#ifdef TEST
main() {
    struct tm *time_struct;
    char tstr[128];
    time_t t = time(NULL);
    time_struct = localtime(&t);

    strftime(tstr, sizeof(tstr),
      "a = %a, A = %A, b = %b, B = %B",
      time_struct);
    printf("%s\n", tstr);

    strftime(tstr, sizeof(tstr),
      "c = %c, d = %d, H = %H, I = %I",
      time_struct);
    printf("%s\n", tstr);

    strftime(tstr, sizeof(tstr),
      "j = %j, m = %m, M = %M, p = %p",
      time_struct);
    printf("%s\n", tstr);

    strftime(tstr, sizeof(tstr),
      "S = %S, U = %U, w = %w, W = %W",
      time_struct);
    printf("%s\n", tstr);

    strftime(tstr, sizeof(tstr),
      "x = %x, X = %X, y = %y, Y = %Y",
      time_struct);
    printf("%s\n", tstr);

    strftime(tstr, sizeof(tstr),
      "Z = %Z, % = %%, F = %F, & = %&",
      time_struct);
    printf("%s\n", tstr);

    exit(0);
}
#endif
```