Calendar Library

revision: 1.31 of 2021/06/17 23:41:34
printed: 2021/06/17 16:43
printed from version being edited!!!

1. Introduction.

This program is a collection of calendar routines that we've written a number of times in a number of different forms. It makes sense to have definitive, common version of these.

My original exploration of calendar problems was an attempt to manually calculate the correspondence between the number of each issue of *The New York Times* and its date. A later version, as a C library at Interactive Systems, provided a linear date count from 14 September 1752, the date Great Britain and her colonies changed from the Julian to the Gregorian calendar. (This, unfortunately, was a computationally disastrous approach.)

The current version is (like several of my more recent implementations) based on Ed Reingold and Nachum Dershowitz's book *Calendrical Calculations* [Reingold].

[[[Update the Reingold/Dershowitz reference to the third edition. Need to check all refs to sections in that book.]]]

All three of these calendar approaches work by sequentially numbering dates from a set epoch. We then provide a number of routines for converting from dates in a particular calendar system (say, Gregorian or Hebrew) to and from these day numbers. Reingold calls these RD or *Rata die*, "fixed days." For [Reingold], the epoch date is Monday 1 January 1 CE.

This WEB file provides a library for all of these calculation routines. We provide an extensive test routines for each routine in the library. The main program generated by this WEB file is actually the test program for the library, as is explained in the body of the program.

[[[... should use these as the basis of the Perl `PM` *package, too.]]] [[[... and perhaps even a Python library]]]*

[[[... and note that we interchangably use rd *and* $fixed$ *in the code.]]]*

2. Header file, infrastructure, basic utilities.
Let's begin with some version identification information.

```
⟨ cal.h 2 ⟩ ≡
#if 0
  static char *warning =
      "THIS␣HEADER␣FILE␣IS␣AUTOMATICALLY␣GENERATED;␣"
      "refer␣to␣the␣CWEB␣source␣for␣details";
  static char *id =
      "$Id:␣cal.w,v␣1.31␣2021/06/17␣23:41:34␣jeff␣Exp␣jeff␣$";
#endif
```

See also sections 3, 5, 6, 14, 20, 25, 28, 34, 40, 46, 50, 54, 60, 66, 75, 84, 87, 100, 109, 110, and 116.

3. Even though the library is flat C, we are building the library with C++ as "Super-C" for better syntax checking. However, we'll still need to link to C programs, so we will need to add `extern "C"` to function declarations.

(Note that if we were building this as a pure C++ library, we could rely on polymorphism for the conversion routines. For example, instead of disjoint *CAL_gregorian_from_fixed*( ) and *CAL_fixed_from_gregorian*( ) routines, there would be two versions of *CAL_gregorian*( ), one taking *month*, *day*, *year* and one taking *fixed_date*.)

format *__externdecl* *int*

```
⟨ cal.h 2 ⟩ +≡
#ifdef __LANGUAGE_C_PLUS_PLUS__
#define __externdecl extern "C"
#else
#define __externdecl
#endif
```

4. It will be helpful to have an external version string which can be found by the RCS `ident` command. We'll build it as a separate source file. *[[[But this doesn't get linked into the final program, because there's no real reference to the contents of the data.]]]*

```
⟨ id.c 4 ⟩ ≡
  char *external_id =
      "$Id:␣cal.w,v␣1.31␣2021/06/17␣23:41:34␣jeff␣Exp␣jeff␣$";
```

5. However, to ensure that version this string is included in every executable, we'll add a definition to the header file.

```
⟨ cal.h 2 ⟩ +≡
  extern char *external_id;
```

6. We will also need some basic type definitions.

```
⟨ cal.h 2 ⟩ +≡
#ifndef __LANGUAGE_C_PLUS_PLUS__
  typedef int bool;
  enum bool {
    false = 0, true
  };
#endif
```

7. In parallel, we'll build the test program. Because we're primarily building a library, the test program is the main program in this WEB source.

```
#include <stdio.h>
#include <stdarg.h>
#include "cal.h"
  ⟨ test data 17 ⟩
  ⟨ test subroutines 9 ⟩
  main( )
  {
    ⟨ test execution 16 ⟩
    eprint("...testing␣done\n");
    return g_errors ? 1 : 0;
  }
```

8. Also, we'll add the descriptive text for the library's Unix manual page into this program text. The manual page text will use special TEX macros which will allow it to be extracted later and formatted by *troff*. This text will be in separate sections, and after this one, they will be labeled "User documentation," and be listed in the index under that title.

[[[Explain ordering rules used by the macros and extractor script. Or perhaps make them explicit in the macros themselves.]]]

NAME

libcal – standard calendar library routines

SYNOPSIS

```
#include "cal.h"
gcc ... -lcal [-lm]
```

DESCRIPTION

These library routines provide a standard way to manipulate dates and other calendar information. They are based, in part, on Reingold and Dershowitz's *Calendrical Calculations*. (See the CWEB source files for complete reference information.)

The basic data type within the library is the *fixed date*, which is a **long** representing the number of days since 1 January 1 CE. By beginning with a monotonically increasing day counter, date-to-date calculations are relatively easy.

The following sections describe the routines by class.

AUTHOR

Jeffrey Copeland

VERSION

version 1.31, dated 2021/06/17 23:41:34 – currently being edited

9. Just for convenience, we'll use an error status routine. This prints our placemarkers (lines which tell us what test we're starting that begin with `...`) flush left. It prints errors offset to the right by a couple of spaces so we can easily spot them.

⟨ test subroutines 9 ⟩ ≡

```
int g_errors = 0;
void eprint(const char *format, ...)
{
  va_list ap;
  if (*format ≠ '.') printf("␣␣");      /* initial spacer */
  va_start(ap, format);
  vprintf(format, ap);
  va_end(ap);
  if (*format ≠ '.') g_errors++;
}
```

See also sections 15, 22, 30, 36, 42, 56, 62, 68, 77, 89, 101, 112, and 118.

This code is used in section 7.

10. Since we are arranging our test data in arrays of **struct**s a macro to give us array length would be useful. (It would also be nice to have the Simula *using* construct, to make references to the test data from the structures easier.)

#define ARRAY_SIZE(x) (**sizeof** (x)/**sizeof** (x[0]))

11. Math utility functions.

All our work is based on some common math functions, such as *floor__* and *mod__*. The C library doesn't define these for **int**s, and the C standard doesn't provide a version of *mod__* that is compatible with mathematical practice for negative arguments. We'll define and test all these functions in the next few sections.

We are appending underscores to the names of these functions so they won't be confused with similarly named functions in the standard library.

We begin with *floor__* and *ceiling__* for a/b of a given numerator and denominator. For *floor__*, if one of the numerator or denominator is negative, and there is a residue in the division, the floor is actually one less than the truncated result of the division. For *ceiling__*, we use a similar technique: if the numerator and denominator have the same sign, and there is a residue in the division, the ceiling is one more than the truncated result of the division. In both cases, we return 0 if the denominator is 0.

We need to define **__externdecl** here because we are not including `cal.h`, where we have the main definition.

⟨ `mathfunc.cc` 11 ⟩ ≡

```
#define __externdecl extern "C"
  __externdecl long sgn_(long);
  __externdecl long floor__(long a, long b)
  {
    long r;
    if (b ≡ 0) return 0;
    r = a/b;
    if (sgn__(a) ≠ sgn__(b) ∧ (a % b) ≠ 0) r−−;
    return r;
  }
  __externdecl long ceiling_(long a, long b)
  {
    long r;
    if (b ≡ 0) return 0L;
    r = a/b;
    if (sgn__(a) ≡ sgn__(b) ∧ (a % b) ≠ 0) r++;
    return r;
  }
```

See also sections 12 and 13.

12. The *sgn__* function returns the sign of the argument: -1, 1, or 0. The most effective way to do this (though less efficient) is to return $a/|a|$.

```
⟨ mathfunc.cc 11 ⟩ +≡
  __externdecl long sgn__(long a)
  {
    if (a ≡ 0L) return 0;
    else if (a > 0L) return 1;
    else return −1;
  }
```

13. We continue with *mod__*, as defined in [Graham, 3.21 and 3.22] and Reingold's *amod__*. In the latter, if $a \bmod b = 0$, return b, otherwise return $a \bmod b$.

```
⟨ mathfunc.cc 11 ⟩ +≡
  __externdecl long mod__(long a, long b)
  {
    if (b ≡ 0) return a;
    return a − b ∗ floor__(a, b);
  }
  __externdecl long amod__(long a, long b)
  {
    long r = mod__(a, b);
    return r ? r : b;
  }
```

14. Prototypes for these functions:

```
⟨ cal.h 2 ⟩ +≡
  __externdecl long mod__(long, long);
  __externdecl long amod__(long, long);
  __externdecl long floor__(long, long);
  __externdecl long ceiling__(long, long);
  __externdecl long sgn__(long);
```

15. Tests for math utilities.

```
⟨ test subroutines 9 ⟩ +≡
  void math_function_tests(void)
  {
    int i, n;
    eprint("...basic␣math␣functions\n");
    n = ARRAY_SIZE(mod_data);
    for (i = 0; i < n; i++) {
      long m = mod__(mod_data[i].num, mod_data[i].denom);
      long a = amod__(mod_data[i].num, mod_data[i].denom);
      long f = floor_(mod_data[i].num, mod_data[i].denom);
      long c = ceiling__(mod_data[i].num, mod_data[i].denom);
      int s = sgn__(mod_data[i].num);
      if (m ≠ mod_data[i].mod)
        eprint("mod__(%ld,%ld)␣=␣%ld;␣function␣says␣%ld\n",
            mod_data[i].num, mod_data[i].denom, mod_data[i].mod, m);
      if (a ≠ mod_data[i].amod)
        eprint("amod__(%ld,%ld)␣=␣%ld;␣function␣says␣%ld\n",
            mod_data[i].num, mod_data[i].denom, mod_data[i].amod, m);
      if (f ≠ mod_data[i].floor)
        eprint("floor__(%ld,%ld)␣=␣%ld;␣function␣says␣%ld\n",
            mod_data[i].num, mod_data[i].denom, mod_data[i].floor, f);
      if (c ≠ mod_data[i].ceiling)
        eprint("ceiling__(%ld,%ld)␣=␣%ld;␣function␣says␣%ld\n",
            mod_data[i].num, mod_data[i].denom, mod_data[i].ceiling, c);
      if (s ≠ mod_data[i].sgn)
        eprint("sgn__(%ld)␣=␣%d;␣function␣says␣%d\n",
            mod_data[i].num, mod_data[i].sgn, s);
    }
  }
```

16. ⟨ test execution 16 ⟩ ≡

```
  math_function_tests( );
```

See also sections 23, 31, 37, 43, 57, 63, 70, 78, 90, 102, 113, and 119.

This code is used in section 7.

17. ⟨ test data 17 ⟩ ≡

```
struct {
  long num, denom, mod, amod, floor, ceiling, sgn;
} mod_data[] = {
    {12, 3, 0, 3, 4, 4, 1}, {14, 3, 2, 2, 4, 5, 1},
    {14, -3, -1, -1, -5, -4, 1}, {-14, -3, -2, -2, 4, 5, -1},
    {-14, 3, 1, 1, -5, -4, -1}, {-169, 400, 231, 231, -1, 0, -1},
    {169, 400, 169, 169, 0, 1, 1}, {-400, 400, 0, 400, -1, -1, -1},
    {0, 17, 0, 17, 0, 0, 0}, {17, 0, 17, 17, 0, 0, 1}
};
```

See also sections 24, 32, 38, 44, 58, 64, 71, 79, 103, 114, and 120.

This code is used in section 7.

18. Weekday routines.

The routines in the following sections compute dates such as "the Saturday following this date." This first function is from [Reingold, formula 1.41], where they are referred to as the "k-day functions."

(Note that we're using *CAL_weekday_from_fixed*, which we won't define until later.

```
⟨ kday.cc 18 ⟩ ≡
#include "cal.h"
  __externdecl long CAL_kday_on_or_before(long rd, int wkday)
  {
    return rd − CAL_weekday_from_fixed(rd − wkday);
  }
```

See also section 19.

19. These next few routines are simple variations on the basic routine in the previous section, and are developed in [Reingold, formulas 1.46 through 1.49].

```
⟨ kday.cc 18 ⟩ +≡
  __externdecl long CAL_kday_on_or_after(long rd, int wkday)
  {
    return CAL_kday_on_or_before(rd + 6, wkday);
  }
  __externdecl long CAL_kday_nearest(long rd, int wkday)
  {
    return CAL_kday_on_or_before(rd + 3, wkday);
  }
  __externdecl long CAL_kday_after(long rd, int wkday)
  {
    return CAL_kday_on_or_before(rd + 7, wkday);
  }
  __externdecl long CAL_kday_before(long rd, int wkday)
  {
    return CAL_kday_on_or_before(rd − 1, wkday);
  }
```

20.

```
⟨ cal.h 2 ⟩ +≡
  __externdecl long CAL_kday_on_or_before(long, int);
  __externdecl long CAL_kday_on_or_after(long, int);
  __externdecl long CAL_kday_nearest(long, int);
  __externdecl long CAL_kday_after(long, int);
  __externdecl long CAL_kday_before(long, int);
```

21. User documentation.

Weekday Routines.

These functions calculate dates such as “the Saturday after this date.”

long *CAL_kday_on_or_before*(**long** *fixed*, **int** *weekday*)
The *weekday* on or before the given fixed date.

long *CAL_kday_on_or_after*(**long** *fixed*, **int** *weekday*)
The *weekday* on or after the given fixed date.

long *CAL_kday_nearest*(**long** *fixed*, **int** *weekday*)
The *weekday* nearest the given fixed date.

long *CAL_kday_after*(**long** *fixed*, **int** *weekday*)
The next *weekday* after the fixed date.

long *CAL_kday_before*(**long** *fixed*, **int** *weekday*)
The first *weekday* before the fixed date.

22. Weekday routine tests.

These are the tests for the weekday routines we developed in the previous sections. It would be more complete to choose days and walk through complete weeks, but testing against single date/weekday pairs is both more transparent, and less susceptible to test code errors.

```
⟨ test subroutines 9 ⟩ +≡
  void kday_test(void)
  {
    int i, n = ARRAY_SIZE(kday_data);
    int wkday;
    long rd, rd_exp, rd_cal;

    eprint("...weekday␣routines\n");
    for (i = 0; i < n; i++) {
      rd = kday_data[i].rd;
      wkday = kday_data[i].wkday;

      rd_exp = kday_data[i].on_or_before;
      rd_cal = CAL_kday_on_or_before(rd, wkday);
      if (rd_exp ≠ rd_cal)
        eprint("calculated␣%s␣on_or_before␣%ld␣as␣%ld,␣should␣be␣%ld\n",
            CAL_weekday_name_full(CAL_type_gregorian, wkday),
            rd, rd_cal, rd_exp);

      rd_exp = kday_data[i].on_or_after;
      rd_cal = CAL_kday_on_or_after(rd, wkday);
      if (rd_exp ≠ rd_cal)
        eprint("calculated␣%s␣on_or_after␣%ld␣as␣%ld,␣should␣be␣%ld\n",
            CAL_weekday_name_full(CAL_type_gregorian, wkday),
            rd, rd_cal, rd_exp);

      rd_exp = kday_data[i].before;
      rd_cal = CAL_kday_before(rd, wkday);
      if (rd_exp ≠ rd_cal)
        eprint("calculated␣%s␣before␣%ld␣as␣%ld,␣should␣be␣%ld\n",
            CAL_weekday_name_full(CAL_type_gregorian, wkday),
            rd, rd_cal, rd_exp);

      rd_exp = kday_data[i].after;
      rd_cal = CAL_kday_after(rd, wkday);
      if (rd_exp ≠ rd_cal)
        eprint("calculated␣%s␣after␣%ld␣as␣%ld,␣should␣be␣%ld\n",
            CAL_weekday_name_full(CAL_type_gregorian, wkday),
            rd, rd_cal, rd_exp);

      rd_exp = kday_data[i].nearest;
      rd_cal = CAL_kday_nearest(rd, wkday);
```

```
    if (rd_exp ≠ rd_cal)
      eprint("calculated␣%s␣nearest␣to␣%ld␣as␣%ld,␣should␣be␣%ld\n",
          CAL_weekday_name_full(CAL_type_gregorian, wkday),
          rd, rd_cal, rd_exp);
  }
}
```

23. ⟨ test execution 16 ⟩ +≡

```
kday_test( );
```

24. Test data for the kday routines. These are all fixed date values in January 2000, from the table of CE 2000 dates on the [Reingold] CD, as follows: *[[[Does this calendar want to be in 8pt and 5pt, rather than 10 and 7?]]]*

2000 January						
Sun	Mon	Tue	Wed	Thu	Fri	Sat
						1 730120
2 730121	3 730122	4 730123	5 730124	6 730125	7 730126	8 730127
9 730128	10 730129	11 730130	12 730131	13 730132	14 730133	15 730134
16 730135	17 730136	18 730137	19 730138	20 730139	21 730140	22 730141
23 730142	24 730143	25 730144	26 730145	27 730146	28 730147	29 730148
30 730149	31 730150					

⟨ test data 17 ⟩ +≡

```
struct {
  long rd;
  int wkday;
  long on_or_before, on_or_after, before, after, nearest;
} kday_data[ ] = {
    {730134, saturday, 730134, 730134, 730127, 730141, 730134},
    {730134, wednesday, 730131, 730138, 730131, 730138, 730131},
    {730137, thursday, 730132, 730139, 730132, 730139, 730139},
    {730137, friday, 730133, 730140, 730133, 730140, 730140},
    {730137, monday, 730136, 730143, 730136, 730143, 730136}
};
```

25. Gregorian calendar.
We begin with the basic definitions and constants.

```
⟨ cal.h 2 ⟩ +≡  /* Gregorian calendar */
  static const int gregorian_epoch = 1;
  typedef enum _CAL_gregorian_months {
    january = 1, february, march, april, may, june, july, august, september, october,
        november, december
  } CAL_gregorian_months;
  typedef enum _CAL_gregorian_weekdays {
    sunday = 0, monday, tuesday, wednesday, thursday, friday, saturday
  } CAL_gregorian_weekdays;
```

26. User documentation.

Gregorian calendar conversions.

These routines perform conversions for Gregorian calendars, particularly to and from fixed dates.

Useful constants:

We define two **enum**s for convenience. The first, **CAL_gregorian_months**, has values of *january*, ..., *december*, and the second, **CAL_gregorian_weekdays**, has values of *sunday*, ..., *saturday*.

27. Gregorian leap year.
First, we implement Gregorian leap year rules.

```
⟨ gregorian_leap_year.cc 27 ⟩ ≡
#include "cal.h"
  __externdecl bool CAL_gregorian_leap_year(long year)
  {
    if ((year % 400) ≡ 0) return true;
    if ((year % 100) ≡ 0) return false;
    if ((year % 4) ≡ 0) return true;
    return false;
  }
```

28. ⟨ cal.h 2 ⟩ +≡

```
  __externdecl bool CAL_gregorian_leap_year(long);
```

29. User documentation.

bool *CAL_gregorian_leap_year*(**long** *year*);
Returns *true* if the given Gregorian year is a leap year.

30. Gregorian leap year tests. A routine to check our leap year calculations, and test data.

```
⟨ test subroutines 9 ⟩ +≡
  void gregorian_leap_year_test(void)
  {
    int i, n = ARRAY_SIZE(gregorian_leap_year_data);
    eprint("...leap␣year\n");
    for (i = 0; i < n; i++) {
      bool l = CAL_gregorian_leap_year(gregorian_leap_year_data[i].yr);
      bool which = gregorian_leap_year_data[i].leap;
      if ((l ∧ ¬which) ∨ (¬l ∧ which))
        eprint("%d␣should␣be␣%s\n", i, which ? "leap" : "non-leap");
    }
  }
```

31.
```
⟨ test execution 16 ⟩ +≡
  gregorian_leap_year_test( );
```

32.
```
⟨ test data 17 ⟩ +≡
  struct {
    long yr;
    bool leap;
  } gregorian_leap_year_data[ ] = {
      {1900, false}, {1904, true}, {1996, true}, {1997, false},
      {1963, false}, {2000, true}, {0, true}, {2004, true},
      {−168, true}, {−586, false}
  };
```

33. Gregorian month length. It is useful to have a function to return the last day of a given month.

```
⟨ gregorian_month.cc 33 ⟩ ≡
#include "cal.h"
  long days_norm[ ] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
  long days_leap[ ] = {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
  __externdecl long CAL_gregorian_month_length(long year, long month)
  {
    if (month < january ∨ month > december) return −1L;
    return CAL_gregorian_leap_year(year)
        ? days_leap[month] : days_norm[month];
  }
```

See also section 39.

34. ⟨ cal.h 2 ⟩ +≡
```
__externdecl long CAL_gregorian_month_length(long, long);
```

35. User documentation.

long *CAL_gregorian_month_length*(**long** *year*, **long** *month*);
Returns the Gregorian last day of the given month in the given year.

36. Gregorian month testing.

⟨ test subroutines 9 ⟩ +≡
```
void gregorian_month_length_tests(void)
{
  int i, n = ARRAY_SIZE(gregorian_month_length_data);
  for (i = 0; i < n; i++) {
    long y = gregorian_month_length_data[i].year;
    long m = gregorian_month_length_data[i].month;
    long d_exp = gregorian_month_length_data[i].days;
    long d_cal = CAL_gregorian_month_length(y, m);
    if (d_exp ≠ d_cal)
      eprint("calculated length of %s %ld is %ld, expected %ld\n",
          CAL_month_name_full(CAL_type_gregorian, m), y, d_cal, d_exp);
  }
}
```

37. ⟨ test execution 16 ⟩ +≡
```
gregorian_month_length_tests( );
```

38. ⟨ test data 17 ⟩ +≡
```
struct {
  long year, month, days;
} gregorian_month_length_data[ ] = {
    {1957, june, 30},
    {1960, june, 30},
    {1968, february, 29},
    {1967, february, 28}
};
```

39. Gregorian date sanity check. It's also useful to have a function to check that the parts of a Gregorian date are valid.

```
⟨ gregorian_month.cc 33 ⟩ +≡
  __externdecl bool CAL_gregorian_valid(long year, long month, long day)
  {
    if (month < january ∨ month > december) return false;
    if (day < 1 ∨ day > CAL_gregorian_month_length(year, month)) return false;
    return true;
  }
```

40.
```
⟨ cal.h 2 ⟩ +≡
  __externdecl bool CAL_gregorian_valid(long, long, long);
```

41. User documentation.

long *CAL_gregorian_valid*(**long** *year*, **long** *month*, **long** *day*);
Returns *true* if the Gregorian date is valid.

42. Gregorian date sanity check tests.

```
⟨ test subroutines 9 ⟩ +≡
  void gregorian_valid_tests(void)
  {
    int i, n = ARRAY_SIZE(gregorian_valid_data);
    for (i = 0; i < n; i++) {
      long y = gregorian_valid_data[i].year;
      long m = gregorian_valid_data[i].month;
      long d = gregorian_valid_data[i].days;
      bool v_exp = gregorian_valid_data[i].valid;
      bool v_cal = CAL_gregorian_valid(y, m, d);
      if (v_exp ≠ v_cal) eprint("expected␣%d/%d/%d␣to␣be␣%svalid,␣was␣%svalid\n",
          y, m, d, v_exp ? "" : "in", v_cal ? "" : "in");
    }
  }
```

43.
```
⟨ test execution 16 ⟩ +≡
  gregorian_valid_tests( );
```

44. ⟨ test data 17 ⟩ +≡

```
struct {
  long year, month, days;
  bool valid;
} gregorian_valid_data[] = {
    {1957, june, 30, true},
    {1960, 0, 30, false},
    {1967, february, 29, false},
    {1968, february, 0, false},
    {2000, december, 32, false},
    {2001, september, 11, true},
};
```

45. Gregorian to fixed. Next, we calculate the fixed date from the Gregorian year, month, day.

⟨ fixed_from_gregorian.cc 45 ⟩ ≡

```
#include "cal.h"
  __externdecl long CAL_fixed_from_gregorian(long year, long month, long day)
  {
    long r;
    r = gregorian_epoch − 1;
    r += 365 ∗ (year − 1);
    r += floor__((year − 1), 4);
    r −= floor__((year − 1), 100);
    r += floor__((year − 1), 400);
    r += floor__((367 ∗ month − 362), 12);
    if (month ≤ 2) r += 0;
    else if (month > 2 ∧ CAL_gregorian_leap_year(year)) r −= 1;
    else r −= 2;
    r += day;
    return r;
  }
```

46. ⟨ cal.h 2 ⟩ +≡

```
__externdecl long CAL_fixed_from_gregorian(long, long, long);
```

47. User documentation.

long *CAL_fixed_from_gregorian*(**long** *year*, **long** *month*, **long** *day*);
Returns the fixed day number for the given Gregorian year, month, and day.

48. Fixed to Gregorian. An inverse of *CAL_fixed_from_gregorian*, to calculate the Gregorian date from a fixed day. [Reingold, formula 2.20]

```
⟨ gregorian_from_fixed.cc 48 ⟩ ≡
#include "cal.h"
  __externdecl void CAL_gregorian_from_fixed(long rd, long *year, long *mon, long
        *day)
  {
    int prior_days;      /* day number in year */
    long jan1, mar1;
    int correction;
    *year = CAL_gregorian_year_from_fixed(rd);
    jan1 = CAL_fixed_from_gregorian(*year, january, 1);
    mar1 = CAL_fixed_from_gregorian(*year, march, 1);
    prior_days = rd - jan1;
    if (rd < mar1) correction = 0;
    else if (rd ≥ mar1 ∧ CAL_gregorian_leap_year(*year)) correction = 1;
    else correction = 2;
    *mon = floor__((12 * (prior_days + correction) + 373), 367);
    *day = rd - CAL_fixed_from_gregorian(*year, *mon, 1) + 1;
  }
```

See also section 49.

49. We also need the included calculation of the year from the fixed day. [Reingold, formulas 2.18]

```
⟨ gregorian_from_fixed.cc 48 ⟩ +≡
  __externdecl long CAL_gregorian_year_from_fixed(long rd)
  {
    long d0 = rd - gregorian_epoch;
    long n400 = floor_(d0, 146097);
    long d1 = mod__(d0, 146097);
    long n100 = floor_(d1, 36524);
    long d2 = mod__(d1, 36524);
    long n4 = floor__(d2, 1461);
    long d3 = mod__(d2, 1461);
    long n1 = floor__(d3, 365);
    long year = 400 * n400 + 100 * n100 + 4 * n4 + n1;
    if (n100 ≡ 4 ∨ n1 ≡ 4) return year;
    else return year + 1;
  }
```

50. ⟨ cal.h 2 ⟩ +≡
```
__externdecl void CAL_gregorian_from_fixed(long, long *, long *, long *);
__externdecl long CAL_gregorian_year_from_fixed(long);
```

51. User documentation.

void *CAL_gregorian_from_fixed*(**long** *fixed*, **long** ∗*year*, **long** ∗*month*, **long** ∗*day*);
Return the Gregorian year, month, and day corresponding to a particular fixed day number.

long *CAL_gregorian_year_from_fixed*(**long** *fixed*);
Return the Gregorian year in which the given fixed day number occurs. This function is mostly used by *CAL_gregorian_from_fixed*( ).

52. *[[[need to add ordinal day in gregorian year]]]*

53. Gregorian day of week. We'll also want a way to determine the day of the week from the fixed day. [Reingold, formula 1.39.]

⟨ weekday_from_fixed.cc 53 ⟩ ≡
```
#include "cal.h"
__externdecl int CAL_weekday_from_fixed(long date)
{
  return mod__(date, 7);
}
```

54. ⟨ cal.h 2 ⟩ +≡
```
__externdecl int CAL_weekday_from_fixed(long);
```

55. User documentation.

int *CAL_weekday_from_fixed*(**long** *fixed*);
Returns the day of the week for a given fixed day number. The constants defined for *weekday* will be helpful for converting the result.

56. Gregorian calendar tests. This test routine uses an array of data which will be defined shortly.

```
⟨ test subroutines 9 ⟩ +≡
  void fixed_gregorian_test(void)
  {
    int i, n = ARRAY_SIZE(fixed_gregorian_data);
    eprint("...gregorian␣to␣fixed\n");
    for (i = 0; i < n; i++) {
      long yr = fixed_gregorian_data[i].year;
      long mon = fixed_gregorian_data[i].mon;
      long day = fixed_gregorian_data[i].day;
      long rd = fixed_gregorian_data[i].fixed;
      long rdc = CAL_fixed_from_gregorian(yr, mon, day);
      int wkday = CAL_weekday_from_fixed(rdc);
      if (rd ≠ rdc)
        eprint("calculated␣RD␣%ld␣for␣YMD␣%ld/%ld/%ld,␣should␣be␣%ld\n",
            rdc, yr, mon, day, rd);
      if (wkday ≠ fixed_gregorian_data[i].weekday)
        eprint("weekday␣for␣gregorian␣YMD␣%ld/%ld/%ld␣is␣not␣%d\n",
            yr, mon, day, wkday);
    }
    eprint("...fixed␣to␣gregorian\n");
    for (i = 0; i < n; i++) {
      long yr = fixed_gregorian_data[i].year;
      long mon = fixed_gregorian_data[i].mon;
      long day = fixed_gregorian_data[i].day;
      long rd = fixed_gregorian_data[i].fixed;
      long yrc, monc, dayc;
      CAL_gregorian_from_fixed(rd, &yrc, &monc, &dayc);
      if (yrc ≠ yr ∨ monc ≠ mon ∨ dayc ≠ day)
        eprint("calculated␣gregorian␣YMD␣%ld/%ld/%ld␣from␣RD␣%ld,␣"
            "should␣be␣YMD␣%ld/%ld/%ld\n",
            yrc, monc, dayc, rd, yr, mon, day);
    }
  }
```

57. ⟨ test execution 16 ⟩ +≡

```
  fixed_gregorian_test( );
```

58. The following test data is extracted from `Dates/dates1.tsv` on the CD from [Reingold].

```
⟨ test data 17 ⟩ +≡
  struct {
    long fixed;
    int weekday;
    long year, mon, day;
  } fixed_gregorian_data[] = {
      {−214193, sunday, −586, 7, 24},
      {−61387, wednesday, −168, 12, 5},
      {25469, wednesday, 70, 9, 24},
      {49217, sunday, 135, 10, 2},
      {171307, wednesday, 470, 1, 8},
      {210155, monday, 576, 5, 20},
      {253427, saturday, 694, 11, 10},
      {369740, sunday, 1013, 4, 25},
      {400085, sunday, 1096, 5, 24},
      {434355, friday, 1190, 3, 23},
      {452605, saturday, 1240, 3, 10},
      {470160, friday, 1288, 4, 2},
      {473837, sunday, 1298, 4, 27},
      {507850, sunday, 1391, 6, 12},
      {524156, wednesday, 1436, 2, 3},
      {544676, saturday, 1492, 4, 9},
      {567118, saturday, 1553, 9, 19},
      {569477, saturday, 1560, 3, 5},
      {601716, wednesday, 1648, 6, 10},
      {613424, sunday, 1680, 6, 30},
      {626596, friday, 1716, 7, 24},
      {645554, sunday, 1768, 6, 19},
      {664224, monday, 1819, 8, 2},
      {671401, wednesday, 1839, 3, 27},
      {694799, sunday, 1903, 4, 19},
      {704424, sunday, 1929, 8, 25},
      {708842, monday, 1941, 9, 29},
      {709409, monday, 1943, 4, 19},
      {709580, thursday, 1943, 10, 7},
      {727274, tuesday, 1992, 3, 17},
      {728714, sunday, 1996, 2, 25},
      {744313, wednesday, 2038, 11, 10},
      {764652, sunday, 2094, 7, 18}
  };
```

59. Weekday within a month – Holidays. These routines are variations on the weekday routines developed in an earlier section. They operate on calendar dates rather than fixed days, and hence make calculating holidays easy: Election day in the United States is the first Tuesday after the first Monday in November; Memorial day is the last Monday in May; SFPA deadline is the last Thursday of odd numbered months.

These are developed in [Reingold, §2.4], specifically in formulas 2.28, 2.29, and 2.30. Days before the given date are negative. The last two are obviously just special cases of the first. These names may be marginally confusing: *last_kday* is actually "last *kday* before the given date," or "previous *kday*."

We add an extra hook here: if we specify a day of ≤ 0, we actually mean the last day of the month in question. (This last feature is an extension of the functionality in the code of [Reingold].)

```
⟨ holiday.cc 59 ⟩ ≡
#include "cal.h"
  __externdecl long CAL_nth_kday(int n, int wkday, long year, long month, long day)
  {
    long rd;
    if (month < january ∨ month > december
          ∨ day > CAL_gregorian_month_length(year, month)) return −1L;
    if (day ≤ 0) day = CAL_gregorian_month_length(year, month);
    rd = CAL_fixed_from_gregorian(year, month, day);
    if (n > 0)      /* from beginning of month */
      return 7 * n + CAL_kday_before(rd, wkday);
    else      /* from end of month */
      return 7 * n + CAL_kday_after(rd, wkday);
  }
  __externdecl long CAL_first_kday(int wkday, long year, long month, long day)
  {
    CAL_nth_kday(1, wkday, year, month, day);
  }
  __externdecl long CAL_last_kday(int wkday, long year, long month, long day)
  {
    CAL_nth_kday(−1, wkday, year, month, day);
  }
```

60.
```
⟨ cal.h 2 ⟩ +≡
  __externdecl long CAL_nth_kday(int, int, long, long, long);
  __externdecl long CAL_first_kday(int, long, long, long);
  __externdecl long CAL_last_kday(int, long, long, long);
```

61. User documentation.

Gregorian holidays.

These functions return the fixed day number of a weekday before or after a given year, month, and day.

long *CAL_nth_kday*(**int** n, **int** *wkday*, **long** *year*, **long** *month*, **long** *day*);
Returns the fixed day number of the n-th weekday on or after (or, if n is less than zero, on or before) the given Gregorian year, month, and day. As a special case, a day of less than or equal to zero is shorthand for the last day of the given month and year.

long *CAL_first_kday*(**int** *wkday*, **long** *year*, **long** *month*, **long** *day*);
A shorthand for the next weekday on or after the given Gregorian year, month, and day.

long *CAL_last_kday*(**int** *wkday*, **long** *year*, **long** *month*, **long** *day*);
A shorthand for the next weekday on or before the given Gregorian year, month, and day.

For example, Election Day is calculated with

CAL_nth_kday$(1, \mathit{tuesday}, \mathit{year}, \mathit{november}, 2)$, and Memorial Day with

CAL_nth_kday$(-1, \mathit{monday}, \mathit{year}, \mathit{may}, 0)$, or

CAL_last_kday$(\mathit{monday}, \mathit{year}, \mathit{may}, 31)$.

62. Holiday tests.

```
⟨ test subroutines 9 ⟩ +≡
#include <stdlib.h>
#define TH(n) ((n ≡ 1) ? "st" : ((n ≡ 2) ? "nd" : ((n ≡ 3) ? "rd" : "th")))
  void holiday_test(void)
  {
    int i, n = ARRAY_SIZE(holiday_data);
    eprint("...holiday␣routines\n");
    for (i = 0; i < n; i++) {
      int n = holiday_data[i].n;
      int wkday = holiday_data[i].wkday;
      long year = holiday_data[i].year;
      long month = holiday_data[i].month;
      long day = holiday_data[i].day;
      long rd_exp = holiday_data[i].rd_exp;
      long rd_cal = CAL_nth_kday(n, wkday, year, month, day);
      if (rd_exp ≠ rd_cal)
        eprint("calculated␣%d%s␣%s␣%s␣%ld/%ld/%ld␣as␣%ld,␣should␣be␣%ld\n",
            abs(n), TH(abs(n)), CAL_weekday_name_full(CAL_type_gregorian, wkday),
            (n < 0) ? "before" : "after", day, month, year, rd_cal, rd_exp);
    }
  }
```

63.
```
⟨ test execution 16 ⟩ +≡
  holiday_test( );
```

64. Test data for the kday routines. See calendar for January 2000 in the section containing *kday_data*[ ], for the dates involved.

```
⟨ test data 17 ⟩ +≡
  struct {
    int n, wkday;
    long year, month, day;
    long rd_exp;
  } holiday_data[ ] = {
      {−1, thursday, 2000, 1, 0, 730146},
      {4, thursday, 2000, 1, 1, 730146},
      {1, thursday, 2000, 1, 1, 730125},
      {−1, saturday, 2000, 1, 31, 730148},
      {−1, saturday, 2000, 1, −1, 730148},
      {5, sunday, 2000, 1, 1, 730149}
  };
```

65. Easter. This is the classic calculation developed by Clavius in the sixteenth century, for which he was later awarded a lunar crater. [Reingold, formula 8.3].

```
⟨ gregorian_easter.cc 65 ⟩ ≡
#include "cal.h"
  __externdecl long CAL_gregorian_easter(long year)
  {
    long century, shifted_epact, adjusted_epact, paschal_moon;
    century = floor__(year, 100) + 1;
    shifted_epact = 14 + 11 * mod__(year, 19)
        − floor__(3 * century, 4)
        + floor__(5 + 8 * century, 25);
    shifted_epact = mod__(shifted_epact, 30);
    adjusted_epact = shifted_epact;
    if (shifted_epact ≡ 0 ∨ (shifted_epact ≡ 1 ∧ 10 < mod__(year, 19))) adjusted_epact++;
    paschal_moon = CAL_fixed_from_gregorian(year, 4, 19) − adjusted_epact;
    return CAL_kday_after(paschal_moon, sunday);
  }
```

66. ⟨ cal.h 2 ⟩ +≡

```
  __externdecl long CAL_gregorian_easter(long);
```

67. User documentation.

long *CAL_gregorian_easter*(**long** *year*);

Return the fixed day of Easter on the Gregorian calendar for the given year.

68. Gregorian Easter tests. This test routine uses a small set of data.

```
⟨ test subroutines 9 ⟩ +≡
  void gregorian_easter_test(void)
  {
    int i, n = ARRAY_SIZE(gregorian_easter_data);
    long yr, mon, day, easter_rd, rd;
    eprint("...gregorian␣easter\n");
    for (i = 0; i < n; i++) {
      yr = gregorian_easter_data[i].year;
      mon = gregorian_easter_data[i].mon;
      day = gregorian_easter_data[i].day;
      easter_rd = CAL_gregorian_easter(yr);
      long yrc, monc, dayc;
      CAL_gregorian_from_fixed(easter_rd, &yrc, &monc, &dayc);
      if (yr ≠ yrc ∨ mon ≠ monc ∨ day ≠ dayc)
        eprint("calculated␣Easter␣for␣%ld␣is␣%ld/%ld/%ld,␣"
            "should␣be␣%ld/%ld/%ld\n",
            yr, monc, dayc, yrc, mon, day, yr);
    }
    ⟨ mardi gras, too 69 ⟩
  }
```

69. Just for insurance, we check the Mardi Gras date for an arbitrary year. Mardi Gras is the day before Ash Wednesday, which begins Lent. Lent is 40 days long, not including Sundays, or 46 days net. Thus, Mardi Gras is 47 days before Easter. (The actual date comes from the New Orleans visitors' bureau.) This also gives an example of how this kind of calculation can be performed.

```
⟨ mardi gras, too 69 ⟩ ≡
  rd = CAL_gregorian_easter(2009);
  rd −= 47;
  CAL_gregorian_from_fixed(rd, &yr, &mon, &day);
  if (yr ≠ 2009 ∨ mon ≠ 2 ∨ day ≠ 24)
    eprint("calculated␣Mardi␣Gras␣for␣2009␣is␣%ld/%ld/%ld,␣"
        "should␣be␣2/24/2009\n",
        mon, day, yr);
```

This code is used in section 68.

70.
```
⟨ test execution 16 ⟩ +≡
  gregorian_easter_test( );
```

71. This small set of test data is hand calculated. Note that 1609 and 1825 are interesting because Easter and Passover coincided in those years.

```
⟨ test data 17 ⟩ +≡
  struct {
    long year, mon, day;
  } gregorian_easter_data[ ] = {
      {1609, 4, 19},
      {1825, 4, 3},
      {1900, 4, 15},
      {1913, 3, 23},
      {2006, 4, 16}
  };
```

72. Location.

For some calendar functionality we need to know our location cuz, remember, no matter where you go, there you are. This data is useful for astronomical functions and any function that depends on time zone.

```
⟨ location.cc 72 ⟩ ≡
#include <stdlib.h>
#include "cal.h"
  ⟨ global location data 73 ⟩
  __externdecl void CAL_setlocation(float lon, float lat, float alt, char *tz, char
        *name)
  {
    if (CAL_location.allocated) {
      if (CAL_location.name) free(CAL_location.name);
      if (CAL_location.timezone) free(CAL_location.timezone);
    }
    CAL_location.longitude = lon;
    CAL_location.latitude = lat;
    CAL_location.altitude = alt;
    if (tz ≡ NULL ∨ *tz ≡ '\0') {
      CAL_location.timezone = strdup(getenv("TZ"));
    }
    else {
      CAL_location.timezone = strdup(tz);
      setenv("TZ", tz, 1);
    }
    tzset( );
    if (name ≡ NULL ∨ *name ≡ '\0') {
      CAL_location.name = strdup("␣");
    }
    else {
      CAL_location.name = strdup(name);
    }
    CAL_location.allocated = true;
  }
```

See also section 74.

73. We store the location data globally in this module. Our initial location is the Royal Observatory at Greenwich.

```
⟨ global location data 73 ⟩ ≡
  struct _CAL_location {
    float longitude, latitude, altitude;
    char *timezone, *name;
    bool allocated;
  } CAL_location = {
  0.0, 51.477, 44.8, "UTC", "Royal␣Greenwich␣Observatory", false};
```

This code is used in section 72.

74. As a convenience, we add a routine to convert degrees, minutes, seconds into degrees and fractions. Using arguments of float makes fractional seconds easier, such as the longitude of the Royal Observatory, $51° 28′ 37.48″ \equiv 51.477°$. Similarly, for converting feet to meters, such as (again) the Royal Observatory at 147ft ≡ 44.8m.

```
⟨ location.cc 72 ⟩ +≡
  __externdecl float CAL_deg_m_s(float deg, float min, float sec)
  {
    return deg + min/60. + sec/3600.;
  }
  __externdecl float CAL_feet_to_meters(float feet)
  {
    return feet * 12 * 2.54/100;
  }
```

75.

```
⟨ cal.h 2 ⟩ +≡
  __externdecl void CAL_setlocation(float, float, float, char *, char *);
  __externdecl float CAL_deg_m_s(float, float, float);
  __externdecl float CAL_feet_to_meters(float);
```

76. User documentation.

Location.

void *CAL_setlocation*(**float** *lon*, **float** *lat*, **float** *alt*, **char** **tz*, **char** **name*);
Set our current location to longitude *lon* degrees east, latitude *lat* degrees north, altitude *alt* meters above sea level, time zone identifier *tz*, and a name for the location *name*. If *tz* is blank or `NULL`, we default to the value of the environment variable `TZ`.

float *CAL_deg_m_s*(**float** *deg*, **float** *min*, **float** *sec*);
Convert degrees, minutes, seconds into degrees and fractions.

float *CAL_feet_to_meters*(**float** *feet*);
Convert feet to meters.

77. Location tests.
This is the test routine for the previous level 2 section.

```
⟨ test subroutines 9 ⟩ +≡
  void location_test(void)
  {
    int i, n = ARRAY_SIZE(location_data);
    eprint("location␣tests␣not␣implemented␣yet!!!\n");      /* ??? */
    for (i = 0; i < n; i++) {
      long dd = location_data[i].d;
      long ii = location_data[i].i;
    }
  }
```

78. ⟨ test execution 16 ⟩ +≡
```
  location_test( );
```

79. Test data for location.

```
⟨ test data 17 ⟩ +≡
  struct {
    long d, i;
  } location_data[ ] = {
      {0, 0},
      {0, 0}
  };
```

80. Formatting routines.
The next several sections provide lists of month and weekday names for the various calendars, arrays aggregating the names together and finally a routine to display them.

81. Gregorian month and weekday names. The Gregorian names are simple and familiar. (And, yes, we're redefining a macro that we use in the main test program. I don't want to make it public in *cal.h*, so I have defined it once for the test program, and once internally for the *calendar_names* module.)

```
⟨ calendar_names.cc 81 ⟩ ≡
#include "cal.h"
#define ARRAY_SIZE(x) (sizeof (x)/sizeof (*x))
  static char *error_name = "?????";
  static char *gregorian_weekday_abbr_names[] = {
      "Sun","Mon","Tue","Wed","Thu","Fri","Sat"
  };
  static char *gregorian_weekday_full_names[] = {
      "Sunday","Monday","Tuesday","Wednesday","Thursday","Friday","Saturday"
  };
  static char *gregorian_month_abbr_names[] = {
      "Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Oct","Nov","Dec"
  };
  static char *gregorian_month_full_names[] = {
      "January","February","March","April","May","June","July","August",
      "September","October","November","December"
  };
  static const int gregorian_weekday_count = ARRAY_SIZE(gregorian_weekday_full_names);
  static const int gregorian_month_count = ARRAY_SIZE(gregorian_month_full_names);
```

See also sections 82, 83, 85, and 86.

82. Islamic month and weekday names. We don't have abbreviations; we will use the full string when we call for an abbreviation.

```
⟨ calendar_names.cc 81 ⟩ +≡
  static char *islamic_weekday_full_names[] = {
      "yawm␣al-'ahad","yawm␣al-'ithnayn","yawm␣ath-thalatha'",
      "yawm␣al-'arba`a'","yawm␣al-khamis","yawm␣al-jum`a","yawm␣as-sabt"
  };
  static char *islamic_month_full_names[] = {
      "Muharram","Safar","Rabi`␣I","Rabi`␣II","Jumada␣I","Jumada␣II","Rajab",
      "Sha`ban","Ramadan","Shawwal","Dhu␣al-Qa`da","Dhu␣al-Hijja"
  };
  static const int islamic_weekday_count = ARRAY_SIZE(islamic_weekday_full_names);
  static const int islamic_month_count = ARRAY_SIZE(islamic_month_full_names);
```

83. Hebrew month and weekday names. This is the third time we've seen this pattern. Again, there are no abbreviations for month names. We will use English names for weekdays.

The month name routine may need to take account of leap years for Hebrew, since in a leap year the month of Adar I is inserted *before* the twelfth month, and Adar becomes Adar II. *[[[Hebrew leap year has effect on month name?]]]*

```
⟨ calendar_names.cc 81 ⟩ +≡
  static char *hebrew_month_full_names[] = {
      "Nisan", "Iyyar", "Sivan", "Tammuz", "Av", "Elul", "Tishri", "Heshvan",
      "Kislev", "Teveth", "Shevat", "Adar", "Adar␣II"
  };
  static const int hebrew_weekday_count = gregorian_weekday_count;
  static const int hebrew_month_count = ARRAY_SIZE(hebrew_month_full_names);
```

84. Formatting routines implementation. We begin by defining constants for the various calendars, which we will use both as array indexes, and as constants for the printing routines.

format *CAL_types* *int*

```
⟨ cal.h 2 ⟩ +≡
  typedef enum _CAL_types {
    CAL_type_undefined = -1,
    CAL_type_gregorian, CAL_type_julian, CAL_type_hebrew, CAL_type_islamic,
    CAL_type_end
  } CAL_types;
```

85. Next we provide an array of the arrays of data for each of full weekday name, abbreviated weekday name, full month name, abbreviated month name, weekday count, month count. They are indexed by the **enum** above.

```
⟨ calendar_names.cc 81 ⟩ +≡
  static char **weekday_abbr[] = {
      gregorian_weekday_abbr_names, gregorian_weekday_abbr_names,
      gregorian_weekday_abbr_names, islamic_weekday_full_names
  };
  static char **weekday_full[] = {
      gregorian_weekday_full_names, gregorian_weekday_full_names,
      gregorian_weekday_full_names, islamic_weekday_full_names
  };
  static char **month_abbr[] = {
      gregorian_month_abbr_names, gregorian_month_abbr_names,
      hebrew_month_full_names, islamic_month_full_names
  };
  static char **month_full[] = {
      gregorian_month_full_names, gregorian_month_full_names, hebrew_month_full_names,
      islamic_month_full_names
  };
  static int weekday_counts[] = {
      gregorian_weekday_count, gregorian_weekday_count, hebrew_weekday_count,
      islamic_weekday_count
  };
  static int month_counts[] = {
      gregorian_month_count, gregorian_month_count, hebrew_month_count,
      islamic_month_count
  };
```

86. Now we can proceed with the code. As a slight oddness, month names will have one-based numbers $1-n$, but weekdays will have zero-based numbers $0-6$.

```
⟨ calendar_names.cc 81 ⟩ +≡
  __externdecl char *CAL_month_name_full(CAL_types type, long month)
  {
    if (type > CAL_type_undefined ∧ type < CAL_type_end) {
      int count = month_counts[type];
      char **names = month_full[type];
      if (month ≥ 1 ∧ month ≤ count) return names[month − 1];
    }
    return error_name;
  }
  __externdecl char *CAL_month_name_abbr(CAL_types type, long month)
  {
    if (type > CAL_type_undefined ∧ type < CAL_type_end) {
      int count = month_counts[type];
      char **names = month_abbr[type];
      if (month ≥ 1 ∧ month ≤ count) return names[month − 1];
    }
    return error_name;
  }
  __externdecl char *CAL_weekday_name_full(CAL_types type, long weekday)
  {
    if (type > CAL_type_undefined ∧ type < CAL_type_end) {
      int count = weekday_counts[type];
      char **names = weekday_full[type];
      if (weekday ≥ 0 ∧ weekday < count) return names[weekday];
    }
    return error_name;
  }
  __externdecl char *CAL_weekday_name_abbr(CAL_types type, long weekday)
  {
    if (type > CAL_type_undefined ∧ type < CAL_type_end) {
      int count = weekday_counts[type];
      char **names = weekday_abbr[type];
      if (weekday ≥ 0 ∧ weekday < count) return names[weekday];
    }
    return error_name;
  }
```

87. ⟨ cal.h 2 ⟩ +≡

```
__externdecl char *CAL_month_name_full(CAL_types, long);
__externdecl char *CAL_month_name_abbr(CAL_types, long);
__externdecl char *CAL_weekday_name_full(CAL_types, long);
__externdecl char *CAL_weekday_name_abbr(CAL_types, long);
```

88. User documentation.

Calendar display.

char **CAL_month_name_full*(**CAL_types** *type*, **long** *month*);
char **CAL_month_name_abbr*(**CAL_types** *type*, **long** *month*);
char **CAL_weekday_name_full*(**CAL_types** *type*, **long** *weekday*);
char **CAL_weekday_name_abbr*(**CAL_types** *type*, **long** *weekday*);

Return the full or abbreviated month or weekday name for the calendar *type*. Note that the arguments for month are one-based, but the arguments for weekday are zero-based. The **CAL_types** enumerator can take values of: *CAL_type_gregorian*, *CAL_type_julian*, *CAL_type_hebrew*, *CAL_type_islamic*.

89. Formatting tests. These are not exhaustive, but are a basic sanity check.

⟨ test subroutines 9 ⟩ +≡

```
void name_tests(void)
{
  char *a;
  eprint("...names\n");
  a = CAL_month_name_full(CAL_type_gregorian, 9);
  if (strcmp("September", a) ≠ 0)
    eprint("CAL_month_name_full(CAL_type_gregorian, 9) returns %s\n", a);
  a = CAL_weekday_name_full(CAL_type_gregorian, 3);
  if (strcmp("Wednesday", a) ≠ 0)
    eprint("CAL_weekday_name_full(CAL_type_gregorian, 4) returns %s\n", a);
  a = CAL_month_name_abbr(CAL_type_julian, 75);
  if (strcmp("?????", a) ≠ 0)
    eprint("CAL_month_name_abbr(CAL_type_julian, 75) returns %s\n", a);
  a = CAL_month_name_abbr(CAL_type_gregorian, 0);
  if (strcmp("?????", a) ≠ 0)
    eprint("CAL_month_name_abbr(CAL_type_gregorian, 0) return %s\n", a);
  a = CAL_weekday_name_abbr(CAL_type_gregorian, sunday);
  if (strcmp("Sun", a) ≠ 0)
    eprint("CAL_weekday_name_abbr(CAL_type_gregorian, sunday) returns %s\n",
        a);
  ⟨ non-gregorian name testing 91 ⟩
}
```

90. ⟨ test execution 16 ⟩ +≡
name_tests( );

91. Islamic name tests.

⟨ non-gregorian name testing 91 ⟩ ≡
```
a = CAL_month_name_full(CAL_type_islamic, 9);
if (strcmp("Ramadan", a) ≠ 0)
  eprint("CAL_month_name_full(CAL_type_islamic, 9) returns %s\n", a);
a = CAL_weekday_name_full(CAL_type_islamic, 2);
if (strcmp("yawm ath-thalatha'", a) ≠ 0)
  eprint("CAL_weekday_name_full(CAL_type_islamic, 2) returns %s\n", a);
```
See also section 92.

This code is used in section 89.

92. Hebrew name tests.

⟨ non-gregorian name testing 91 ⟩ +≡
```
a = CAL_month_name_full(CAL_type_hebrew, 9);
if (strcmp("Kislev", a) ≠ 0)
  eprint("CAL_month_name_full(CAL_type_hebrew, 9) returns %s\n", a);
a = CAL_weekday_name_full(CAL_type_hebrew, 2);
if (strcmp("Tuesday", a) ≠ 0)
  eprint("CAL_weekday_name_full(CAL_type_hebrew, 2) returns %s\n", a);
```

93. General date formatting. It is helpful to provide a general date formatting routine, in the same style as the POSIX *strftime*. We'll introduce *CAL_format_date* with similar arguments to *strftime*. The function will take a buffer, a buffer size, a format specifier, and the *rdate* as arguments. It will fill the buffer with the date provided, formatted based on the format specifier. The buffer size is the maximum number of characters that can fit in the buffer, including the terminating `NUL`. The function returns the number of bytes written into the buffer (including the terminating `NUL`) if the number of characters written is less than the buffer size; otherwise it returns `0` and the contents of the buffer are undefined. We have tried to align the formatting specifiers to those used by *strftime* where possible.

(User documentation is in the section before the code rather than after it so we can make note of the formatting specifiers before implementing them.)

94. User documentation.

CAL_format_date(**char** ∗*buf*, **size_t** *bufsz*, **const char** ∗*fmt*, **long** *rd*);

Fill a buffer with a formatted date, similarly to *strftime*. The maximum length of the formatted string, including terminating NUL is *bufsize*. Return the length of the formatted string, including the terminating NUL. If the formatted string would be longer than *bufsize*, return 0, and the contents of *buf* are undefined.

The formatting specifiers follow. They are identical to those use by *strftime* where possible; those that are not are marked with an asterisk. Those recognized but not yet implemented are marked with an exclamation point.

%a
Abbreviated Gregorian weekday name.

%A
Full Gregorian weekday name.

%b
Abbreviated Gregorian month name.

%B
Full Gregorian month name.

%C
Century ($\lfloor year/100 \rfloor$).

%d
Day of month (01−31, a single digit is preceeded by a zero).

%e
Day of month (1−31, a single digit is preceeded by a space).

%E
Day of month (1−31, a single digit is preceeded by nothing). *

%j
Day of year. !

%m
Month number (01−12, a single digit is preceeded by a zero).

%n
Month number (1−12, a single digit is preceeded by a space). *

%N
Month number (1−12, a single digit is preceeded by nothing). *

%W
Week number. !

%y
2 digit year ($year \bmod 100$).

%Y
4 digit year.

%H*x*

Use Hebrew calendar particulars such as month name, where *x* is one of the above one-characater specifiers. !

%I*x*

Use Islamic calendar particulars, similarly to Hebrew above. !

Rather than allowing the %n specifier for newline used by *strftime*, we allow two C-style backslash escapes, \n for newline, and \t for tab.

[[[A useful generalization would be an extended specifier like %^ cx*, where* c *is a calendar type (e.g.,* F *for French Revolutionary calendar, or* J *for Julian calendar) and* x *is a single-character format specifier (e.g.,* B *for full month name).]]]*

95. Now we implement the *strftime*-equivalent routine as a separate C file. The following will use other external-facing routines such as *CAL_weekday_name_full*( ). The routine itself follows the general form of *strftime* or *printf*.

```
⟨ format.cc 95 ⟩ ≡
#include "cal.h"
#include <stdio.h>
  ⟨ formatting utilities for CAL_format_date 99 ⟩
  __externdecl size_t CAL_format_date(char *buf, size_t bufsz, const char *fmt, long
          rd)
  {
    char *bufend = buf + bufsz;
    char *b = buf;
    CAL_types current_calendar = CAL_type_undefined;
    long year, month, day;
    int weekday;
    memset(buf, 0, bufsz);
    for ( ; *fmt ∧ b < bufend; fmt++) {
      if (*fmt ≠ '%' ∧ *fmt ≠ '\\') {
        *b++ = *fmt;
        continue;
      }
      if (*fmt ≡ '%') {
        ⟨ process a format specifier 97 ⟩
      }
      else {      /* if we're here, we must have a backslash */
        ⟨ format a backslash escape 96 ⟩
      }
    }
    if (b ≥ bufend) return (size_t) 0;
    return (size_t)(b − buf);
  }
```

96. ⟨ format a backslash escape 96 ⟩ ≡

```
if (*fmt ≡ '\\') {
  switch (*++fmt) {
  case 'n': *b++ = '\n';
    break;
  case 't': *b++ = '\t';
    break;
  default: *b++ = *fmt;
    break;
  }
}
```

This code is used in section 95.

97. We must begin by assuming that we're formatting a date from the Gregorian calendar. If we have an escape into a different calendar, we'll convert to the correct month, day, year at that time, returning to the main switch statement to format the element. This means we'll be repeatedly readjusting the calendar type and alternately calling *CAL_gregorian_from_fixed* and *CAL_hebrew_from_fixed* if we're printing the date of Passover, for example.

The label allows us to skip back into the **switch** for a multi-level specifier, such as `%HM`.

[[[Do we need to revisit the calendar bouncing problem for Hebrew or Islamic dates? Perhaps by using the calendar escape mechanism we postulate earlier, and setting a "default calendar" when we get a global escape.]]]

```
⟨ process a format specifier 97 ⟩ ≡
  if (current_calendar ≠ CAL_type_gregorian) {
    CAL_gregorian_from_fixed(rd, &year, &month, &day);
    weekday = CAL_weekday_from_fixed(rd);
    current_calendar = CAL_type_gregorian;
  }
simple:
  switch (*++fmt) {
  case 'a':     /* abbr weekday */
    b = _add(
          CAL_weekday_name_abbr(current_calendar, weekday),
          b, bufend);
    break;
  case 'A':     /* full weekday */
    b = _add(
          CAL_weekday_name_full(current_calendar, weekday),
          b, bufend);
    break;
  case 'b':     /* abbr mon */
    b = _add(
          CAL_month_name_abbr(current_calendar, month),
          b, bufend);
    break;
  case 'B':     /* full mon */
    b = _add(
          CAL_month_name_full(current_calendar, month),
          b, bufend);
    break;
  case 'C':     /* century */
    b = _conv(year % 100, "%2d", b, bufend);
    break;
  case 'd':     /* day of month 01-31 */
    b = _conv(day, "%02d", b, bufend);
    break;
```

```
case 'e':      /* day of month b1-31 */
  b = _conv(day, "%2d", b, bufend);
  break;
case 'E':      /* day of month 1-31 */
  b = _conv(day, "%d", b, bufend);
  break;
case 'j':      /* day of year */
  b = _add("???", b, bufend);
  break;
case 'm':      /* mon number 01-12 */
  b = _conv(month, "%02d", b, bufend);
  break;
case 'n':      /* mon number b1-12 */
  b = _conv(month, "%2d", b, bufend);
  break;
case 'N':      /* mon number 1-12 */
  b = _conv(month, "%d", b, bufend);
  break;
case 'W':      /* week number */
  b = _add("???", b, bufend);
  break;
case 'y':      /* year % 100 */
  b = _conv(year % 100, "%02d", b, bufend);
  break;
case 'Y':      /* year */
  b = _conv(year, "%d", b, bufend);
  break;
  ⟨ escapes for non-Julian-like calendar names 98 ⟩
default: *b++ = *fmt;
  break;
}
```

This code is used in section 95.

98. We need to process the escapes for non-Julian month and day names. We've documented that we will support Hebrew and Islamic names. However, since *CAL_format_date* takes fixed day number, and we begin by converting to a Gregorian month, day, year, when we get one of these escapes, we need to reconvert to the appropriate calendar.

[[[These are currently dummied up: they need to do real conversions from fixed to the appropriate calendar.]]]

⟨ escapes for non-Julian-like calendar names 98 ⟩ ≡

```
case 'H':      /* Hebrew escape */
  b = _add("???", b, bufend);
  year = 5756;      /* gregorian 1996 */
  month = day = 4;
  weekday = saturday;      /* sabbath */
  current_calendar = CAL_type_hebrew;
  goto simple;
case 'I':      /* Islamic escape */
  b = _add("???", b, bufend);
  year = 1460;      /* gregorian 1996 */
  month = day = 5;
  weekday = friday;      /* sabbath */
  current_calendar = CAL_type_islamic;
  goto simple;
```

This code is used in section 97.

99. We can now back up and define the little formatting helpers we use in the guts of *CAL_format_date*. The first appends a string at the given point in the buffer, being careful not to overrun its end. The second formats a numeric value. Both return the pointer to the next write position into the buffer.

These are modeled after utilities used by the OpenBSD version of *strftime*, and, in particular, the macros are a fairly direct cut-and-paste. The macros are included (rather than just using a constant) because they are a neat hack. `INT_STRLEN_MAX` works on the observation that there are roughly 3 bits per decimal digit ($log_{10}2 \approx .302$). If the type is signed, remove 1 for the sign from the length calculation, but add it back to the final length to fit the printed sign. Also, round up by 1 because of truncation on the integer division.

```
⟨ formatting utilities for CAL_format_date 99 ⟩ ≡
  static char *_add(const char *str, char *p, char *pend)
  {
    while (p < pend ∧ (*p = *str++) ≠ '\0') p++;
    return p;
  }
#define CHAR_BIT 8
#define TYPE_BIT(type) (sizeof (type) * CHAR_BIT)
#define TYPE_SIGNED(type) ((((type) - 1) < 0) ? 1 : 0)
#define INT_STRLEN_MAX(type)
  (((TYPE_BIT(type) - TYPE_SIGNED(type)) * 302)/1000 + TYPE_SIGNED(type) + 1)
  static char *_conv(const int n, const char *fmt, char *p, char *pend)
  {
    char buf[INT_STRLEN_MAX(int) + 1];
    if (snprintf(buf, sizeof (buf), fmt, n) ≥ sizeof (buf)) return pend;
    return _add(buf, p, pend);
  }
```

This code is used in section 95.

100.

```
⟨ cal.h 2 ⟩ +≡
#include <string.h>
#include <stddef.h>
  __externdecl size_t CAL_format_date(char *, size_t, const char *, long);
```

101. General date formatting tests.

```
⟨ test subroutines 9 ⟩ +≡
#define TB_LEN 31
  void generic_date_formatting_tests( )
  {
    char b[TB_LEN];
    int i, n = ARRAY_SIZE(dateformat_data);
    size_t ret;
    eprint("...date␣formats\n");
    for (i = 0; i < n; i++) {
      char *fmt = dateformat_data[i].fmt;
      char *expected = dateformat_data[i].exp;
      long date = dateformat_data[i].rd;
      ret = CAL_format_date(b, TB_LEN, fmt, date);
      if (expected ≡ NULL ∧ ret ≠ 0)
        eprint("format␣%ld␣as␣<%s>␣got␣<%s>\n␣␣␣␣expected␣overflow\n", date,
            fmt, b);
      if (expected ≠ NULL ∧ strcmp(b, expected) ≠ 0)
        eprint("format␣%ld␣as␣<%s>␣got␣<%s>\n␣␣␣␣expected␣<%s>\n", date, fmt, b,
            expected);
    }
  }
```

102. ⟨ test execution 16 ⟩ +≡

```
  generic_date_formatting_tests( );
```

103. Test data for generic date formatting.

Note that we're testing escaping backslashes which requires redoubling the backslashes in the format specifier.

[[[The last entry here, for Hebrew, is dummied up to match the dummy version of the Hebrew formatting in the routine.]]]

```
⟨ test data 17 ⟩ +≡
  struct {
    char *fmt, *exp;
    long rd;
  } dateformat_data[] = {
      {"%B␣\\\\(en", "July␣\\(en", 733596},
      {"%A␣\\\"\\\\n", "Wednesday␣\"\\n", 733596},
      {"%A␣%m/%e/%Y", "Wednesday␣07/␣8/2009", 733596},
      {"%X␣%a␣%d/%n/%Y", "X␣Thu␣01/␣1/1970", 719163},
      {"%A,␣%B␣%d,␣%Y", "Wednesday,␣September␣18,␣2002", 731111},
      {"Date:␣%A,␣%B␣%d,␣%Y", NULL, 731111},
      {"%A␣%Y/%m/%d", "Friday␣1999/12/31", 730119},
      {"%HY/%Hm/%Hd", "???5756/???04/???04", 730119}
  };
```

104. General time formatting. Since we've done a general formatter for dates, we should also provide *format_time*, even though it will be used less frequently.

[[[Is the input argument to format_time an H/M/S triple, or a **float** *representing the fraction of the day?]]]*

[[[Do we output UTC, or local?]]]

105. User documentation.

Time display.

CAL_format_time(. . .);

General purpose print of a time value. Details to be defined.

The formatting specifiers follow. Those recognized but not yet implemented are marked with an exclamation point.

%T
: Time in 24-hour clock with seconds (`HH:MM:SS`). !

%t
: Time in 12-hour clock with seconds and AM/PM (`HH:MM:SSam`). !

%S
: Time in 24-hour clock without seconds (`HH:MM`). !

%s
: Time in 12-hour clock without seconds and AM/PM (`HH:MMam`). !

The mnemonics for this are that `%T` is the *strftime* format declaration, but `%S` is the shorter form without seconds. The `%t` and `%s` forms are lower-case because they represent a smaller number of hours.

106. The implementation of *format_time* follows the pattern of *format_date*.

[[[actually need to implement]]]

107. POSIX Time.

These next few sections allow conversions between our fixed day numbers and the POSIX standard time data structures, **time_t** and **struct** *tm*. We assume a time zone of UTC for these conversions. Note that in both cases we don't provide fractional time within days. As a consequence, we will have to rework these when we begin to do calculations that require times, such as sunrise.

*[[[We also need to throw an error and return something like (***time_t***) −1 if we're out of range.]]]*

⟨ posixtime.cc 107 ⟩ ≡

```
#include "cal.h"      /* 1 Jan 1970 == RD 719163 */
#define POSIX_EPOCH  719163
  __externdecl long CAL_fixed_from_time_t(time_t t)
  {
    long rd = (t)/(60L * 60L * 24L);
    rd += POSIX_EPOCH;
    return rd;
  }
  __externdecl time_t CAL_time_t_from_fixed(long rd)
  {
    rd -= POSIX_EPOCH;
    rd *= (60L * 60L * 24L);
    rd += (60L * 60L * 12L);
    return rd;
  }
```

See also section 108.

108. And now similarly for **struct** *tm*:

⟨ posixtime.cc 107 ⟩ +≡

```
  __externdecl long CAL_fixed_from_tm(const struct tm *t)
  {
    return CAL_fixed_from_gregorian(
        t→tm_year + 1900, t→tm_mon + 1, t→tm_mday);
  }
  __externdecl struct tm *CAL_tm_from_fixed(long rd)
  {
    static struct tm ts, *tp = &ts;
    time_t tt = CAL_time_t_from_fixed(rd);
    tp = gmtime(&tt);
    return tp;
  }
```

109. If we want to use **struct** *tm* and other POSIX time definitions, our library include file must include some basic time definitions from the standard library.

⟨ cal.h 2 ⟩ +≡
```
#include <time.h>
```

110. ⟨ cal.h 2 ⟩ +≡
```
__externdecl long CAL_fixed_from_time_t(time_t);
__externdecl time_t CAL_time_t_from_fixed(long);
__externdecl long CAL_fixed_from_tm(const struct tm *);
__externdecl struct tm *CAL_tm_from_fixed(long);
```

111. User documentation.

POSIX time conversions.

These routines all assume UTC as the time zone.

long *CAL_fixed_from_time_t*(**time_t** *t*);
Convert a **time_t** to a fixed day number.

time_t *CAL_time_t_from_fixed*(**long**);
Convert a fixed day number to the **time_t** for midnight of that day.

long *CAL_fixed_from_tm*(**const struct** *tm* **t*);
Convert a *struct tm* to a fixed day number.

struct *tm* **CAL_tm_from_fixed*(**long**);
Convert a fixed day number to a **struct** *tm*.

112. POSIX time tests.
This tests the POSIX time data routines in the previous two sections.

```
⟨ test subroutines 9 ⟩ +≡
  void posix_test(void)
  {
    int i, n = ARRAY_SIZE(posixtime_data);
    eprint("...posix␣time␣data\n");
    for (i = 0; i < n; i++) {
      long ptime = posixtime_data[i].fixed;
      long yr = posixtime_data[i].year;
      long mon = posixtime_data[i].mon;
      long day = posixtime_data[i].day;
      long rd = posixtime_data[i].fixed;
      long ptimec = CAL_fixed_from_time_t(posixtime_data[i].tt);
      struct tm t, *tp = &t;
      if (ptime ≠ ptimec)
        eprint("calculated␣POSIX␣%ld,␣should␣be␣%ld\n",
            ptimec, ptime);
      tp = CAL_tm_from_fixed(rd);
      if ((tp→tm_year + 1900) ≠ yr ∨
            (tp→tm_mon + 1) ≠ mon ∨
            tp→tm_mday ≠ day)
        eprint("calculated␣tm␣%ld/%ld/%ld␣should␣be␣%ld/%ld/%ld\n",
            tp→tm_year + 1900, tp→tm_mon + 1, tp→tm_mday,
            yr, mon, day);
    }
  }
```

113.
```
⟨ test execution 16 ⟩ +≡
  posix_test( );
```

114. Test data for POSIX time.

```
⟨ test data 17 ⟩ +≡
  struct {
    long fixed;
    time_t tt;
    long year, mon, day;
  } posixtime_data[] = {
      {719163, 0L, 1970, 1, 1},
      {719163, 10800, 1970, 1, 1},
      {719165, 172800, 1970, 1, 3},
      {724950, 500000000, 1985, 11, 5},
      {733597, 1247109526, 2009, 7, 9}
  };
```

115. Skeleton.

This is a skeleton for a full section of code with tests. We build additional sections using this.

```
⟨ skeleton.cc 115 ⟩ ≡
#include "cal.h"
  __externdecl long CAL_skeleton(void)      /* the routine itself */
  {
    return 0L;
  }
```

116.

```
⟨ cal.h 2 ⟩ +≡
  __externdecl long CAL_skeleton(void);      /* prototype */
```

117. User documentation.

CAL_skeleton()

POD

118. Skeleton tests.
This is the test routine for the previous level 2 section.

```
⟨ test subroutines 9 ⟩ +≡
  void skeleton_test(void)
  {
    int i, n = ARRAY_SIZE(skeleton_data);
    eprint("...skeleton\n");
    for (i = 0; i < n; i++) {
      long yr_mon_day = skeleton_data[i].d;
      long yr_mon_dayc = CAL_skeleton( );
      if (yr_mon_day ≠ yr_mon_dayc)
        eprint("calculated␣skeleton␣%ld,␣should␣be␣%ld\n",
            yr_mon_dayc, yr_mon_day);
    }
  }
```

119.
```
⟨ test execution 16 ⟩ +≡
  skeleton_test( );
```

120. Test data for skeleton.

```
⟨ test data 17 ⟩ +≡
  struct {
    long d, i;
  } skeleton_data[ ] = {
      {0, 0},
      {0, 0}
  };
```

121. End Skeleton.

122. Licenses.

[[[This is the original Reingold/Dershowitz license text. Add a note about it.]]]

(Also perhaps from *Reingold/Java/license.txt* include introductory material...)

1. LICENSE. The Authors grant you a license for personal use only. This means that for strictly personal use you may copy and use the code, and keep a backup or archival copy also. Any other uses, including without limitation, allowing the code or its output to be accessed, used, or available to others, is not permitted.
2. WARRANTY.
   (a) THE AUTHORS PROVIDE NO WARRANTIES OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITING THE GENERALITY OF THE FOREGOING, ANY IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.
   (b) THE AUTHORS SHALL NOT BE LIABLE TO YOU OR ANY THIRD PARTIES FOR DAMAGES OF ANY KIND, INCLUDING WITHOUT LIMITATION, ANY LOST PROFITS, LOST SAVINGS, OR ANY OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR RELATED TO THE USE, INABILITY TO USE, OR INACCURACY OF CALCULATIONS, OF THE CODE AND FUNCTIONS CONTAINED HEREIN, OR THE BREACH OF ANY EXPRESS OR IMPLIED WARRANTY, EVEN IF THE AUTHORS OR PUBLISHER HAVE BEEN ADVISED OF THE POSSIBILITY OF THOSE DAMAGES.
   (c) THE FOREGOING WARRANTY MAY GIVE YOU SPECIFIC LEGAL RIGHTS WHICH MAY VARY FROM STATE TO STATE IN THE U.S.A.
3. LIMITATION OF LICENSEE REMEDIES. You acknowledge and agree that your exclusive remedy (in law or in equity), and Authors' entire liability with respect to the material herein, for any breach of representation or for any inaccuracy shall be a refund of the license fee or service and handling charge which you paid the Authors, if any.

   SOME STATES IN THE U.S.A. DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE EXCLUSIONS OR LIMITATION MAY NOT APPLY TO YOU.
4. DISCLAIMER. Except as expressly set forth above, the Authors:
   (a) make no other warranties with respect to the material in the Program and expressly disclaim any others;
   (b) do not warrant that the material contained in the Program will meet your requirements or that their operation shall be uninterrupted or error-free;
   (c) license this material on an "as is" basis, and the entire risk as to the quality, accuracy, and performance of the Program is yours, should the code prove defective (except as expressly warranted herein). You alone assume the entire cost of all necessary corrections.

[[[Add either Artistic – or GPL if the Reingold license mandates.]]]

[[[Add a copyright note as a botofcontents macro.]]]

123. References and bibliography.

[Graham] Graham, Ronald L, Donald E Knuth, and Oren Patashnik: *Concrete Mathematics*, second edition, Addison-Wesley, 1994. (ISBN 0-201-55802-5.)

[Meeus] Meeus, Jean: *Astronomical Algorithms*, second edition, Willmann-Bell, 1998, with corrections to 15 June 2005. (ISBN 0-942296-61-1.)

[Reingold] Reingold, Edward M and Nachum Dershowitz: *Calendrical Calculations: Third Edition*, Cambridge University Press, 2008. (ISBN 978-0-521-70238-6.)

[Smart] Smart, William Marshall and Robin Michael Green: *Textbook on Spherical Astronomy*, sixth edition, Cambridge University Press, 1977. (ISBN 0-521-29180-1, 0-521-21516-1.)

124. To do list.

A number of sections, namely, 1, 4, 8, 24, 52, 83, 94, 97, 98, 103, 104, 106, 107, 122, still contain working notes, which can be recognized because they are set in *slanted sans serif type.*

In addition, we've got the following to do:

- Do the external API names really need to have the prefix `CAL_`?
- But note that the name printing routines (e.g., *gregorian_month_full*) don't have the `CAL_` prefix. (Now they do.) [[DONE]]
- We should also do parameter validation in routines that take (month, day, year) triples.
- Need a Unix-style man page. [[DONE]]
  - — But the man page is currently arranged in the order of exposition of the program, not the logical order for presenting the functions. Fix this.
- Need a license statement.
- Should we add a flag to make the test output contain errors only?

- ISO calendar
- Hebrew calendar
- Islamic calendar
- Chinese new year
- Calculate phase of the moon, sunrise, sunset, moonrise, moonset.
  - — Lunar longitude, Dershowitz 3/ed, §13.47
  - — Solar longitude, Dershowitz 3/ed, §13.30

We've got some things to do for setting up locale handling, and hence calculation of astronomical functions:

- Locales — two entry points, for known named place (e.g., "Boulder", "Bellevue") and for a specific location (i.e., longitude, latitude, elevation, time zone). Actually, this should be called "location", rather than overloading the I18N terminology. To set the time zone using the time zone functions and data from Arthur David Olson at the NIH.
- The existence of locale allows us to correctly handle the day boundaries in the POSIX time routines.
- The time printing function *CAL_format_time* will need to take not a fixed date as a **long**, but as a **double**, so that it can include fractional day. We can then do something in that function like $sec = (fixed - \mathrm{int}(fixed)) \cdot 60 \cdot 60 \cdot 24$, and then break *sec* into hours, minutes, seconds.

Then we've got the additions we want to make to the canonical [Reingold] functionality:

- Add jeff routines to draw the moon, both the existing `troff` code and the new naked PostScript routine we want to write.
  - — For the PostScript routine, we'd dump the PostScript commands to a buffer, which can then be included in the appropriate way in either TEX or `troff`. The PostScript prologue will have to be a static string.
  - — The geometry discussion that's in the existing man page included in `mphase.c` can be part of the `WEB` file.
  - — Revisit the geometry of the 3-D projection of the moon. (See Smart.)

  - For added amusement, point out that, in effect, the `troff` drawing is a TEX implementation of a `troff` pre-processor.
  - The routines for `troff` production should consist of alternate methods: output a generic macro to draw the moon picture; output calls to that macro; output just the parameter to that macro; output the full drawing command.
- By extension we'll add an equivalent of *strftime*, which takes a format string — with a format string as similar as possible to the existing POSIX *strftime* format. [[DONE]]
  - We'll want to either have additonal format specifiers for Hebrew and Islamic month and day names, or a format modifier – e.g., `%hB` for full Hebrew month name.
- And similarly, we'll need an equivalent of *strptime*.
- Can we even include the man page source in this? [[DONE]]
- `mach.usno.navy.mil` for astro software
- `http://astroinfo.sourceforge.net/` for Palm astro software
- `http://timeanddate.com` has a commercial time API

125. Index.