1. Introduction.

TIMEDATE

Time and Date Utilities

by John Walker
http://www.fourmilab.ch/

This program implements the **systemtime** class, which provides a variety of services for date and time quantities which use UNIX **time_t** values as their underlying data type. One can set these values from strings, edit them to strings, convert to and from Julian dates, determine the sidereal time at Greenwich for a given moment, and compute properties of the Moon and Sun including geocentric celestial co-ordinates, distance, and phases of the Moon.

A utility *angle* class provides facilities used by the positional astronomy services in **systemtime**, but may prove useful in its own right.

⟨ timedate_test.c 1 ⟩ ≡
```
#define REVDATE "2nd␣February␣2002"
```
See also section 32.

2. Program global context.

```
#include "config.h"     /* System-dependent configuration */
  ⟨ Preprocessor definitions ⟩
  ⟨ Application include files 4 ⟩
  ⟨ Class implementations 5 ⟩
```

3. We export the class definitions for this package in the external file `timedate.h` that programs which use this library may include.

```
⟨ timedate.h 3 ⟩ ≡
#ifndef TIMEDATE_HEADER_DEFINES
#define TIMEDATE_HEADER_DEFINES
#include <stdio.h>
#include <math.h>     /* Make sure math.h is available */
#include <time.h>
#include <assert.h>
#include <string>
#include <stdexcept>
  using namespace std; ⟨ Class definitions 6 ⟩
#endif
```

4. The following include files provide access to external components of the program not defined herein.

```
⟨ Application include files 4 ⟩ ≡
#include "timedate.h"     /* Class definitions for this package */
```

This code is used in section 2.

5. The following classes are defined and their implementations provided.

```
⟨ Class implementations 5 ⟩ ≡
  ⟨ Angle utilities 7 ⟩
  ⟨ Time and date utilities 9 ⟩
```

This code is used in section 2.

6. Angle utilities.

In positional astronomy code we're forever working with angles in different forms: radians, degrees, arc-seconds, etc. This utility class provides a collection of transformations among these representations.

```
⟨ Class definitions 6 ⟩ ≡
  class angle {
#define PI  3.14159265358979323846      /* π */
  public:      /* Return π */
    static double Pi(void)
    {
      return PI;
    }      /* Convert arc-seconds to degrees */
    static double Asec(double x)
    {
      return x/3600.0;
    }      /* Degrees to radians */
    static double dtr(double d)
    {
      return d * (PI/180);
    }      /* Radians to degrees */
    static double rtd(double r)
    {
      return r/(PI/180);
    }      /* Range reduce an angle in degrees */
    static double fixangle(double d)
    {
      return d − 360.0 * (floor(d/360.0));
    }      /* Range reduce an angle in radians */
    static double fixangr(double r)
    {
      return r − (PI * 2) * (floor(r/(PI * 2)));
    }      /* Convert degrees (or hours), minutes, and seconds to decimal fraction */
    static double d_m_s_to_decimal(int d, int m, double s)
    {
      return d + (m/60.0) + (s/(60.0 * 60.0));
    }      /* Edit an angle in degrees to degrees, minutes, and seconds */
    static string degrees_to_d_m_s(double a);
#undef PI
  };
```

See also section 8.

This code is used in section 3.

7. When debugging positional astronomy code, it's handy to be able to print angles as degrees, minutes, and seconds. This **static** function so formats its **double** argument and returns a **string** containing the edited text.

```
⟨ Angle utilities 7 ⟩ ≡
  string angle::degrees_to_d_m_s(double a)
  {
    int dd, mm;
    double ss;
    char result[16];
    char *sign = "";
    if (a < 0) {
      sign = "-";
      a = -a;
    }
    dd = (int) a;
    mm = (int)((a - dd) * 60);
    ss = (a - (dd + (mm/60.0))) * 3600;
    sprintf(result, "%s%d%d'%.3f\"", sign, dd, mm, ss);
    return string(result);
  }
```

This code is used in section 5.

8. Time and date utilities.

Our databases represent date and time as Unix **time_t** values, which specify the number of seconds elapsed since 00:00:00 UTC on January 1st, 1970 (the "Epoch"), ignoring leap seconds subsequently inserted. The **systemtime** class provides facilities for manipulating time and date quantities in this form.

The **systemtime** class contains a time and date in system format and defines methods for setting and retrieving time in various formats.

We start with some definitions of numerical quantities we can't store in **static const** quantities because they're non-integral.

```
#define JulianCentury 36525.0 /* Days in Julian century */
#define JulianMillennium (JulianCentury * 10) /* Days in Julian millennium */
#define J2000 2451545.0 /* Julian day of J2000.0 epoch */
#define SynMonth 29.53058868 /* Synodic month (mean time from new Moon to new Moon) */
```

⟨ Class definitions 6 ⟩ +≡

```
class systemtime : protected angle {
private:
  time_t t;
public:
  static const int SecondsPerDay = 24 * 60 * 60, SecondsPerHour = 60 * 60, SecondsPerMinute = 60;
  systemtime(time_t it = 0)
  {
    set_time(it);
  }
  systemtime(string dateTime)
  {
    fromString(dateTime);
  }     /* Return time as a time_t */
  time_t get_time(void)
  {
    return t;
  }     /* Set time to a time_t */
  void set_time(const time_t nt)
  {
    t = nt;
  }     /* Obtain current time and date */
  void now(void)
  {
    t = time(Λ);
  }     /* Obtain midnight of this day */
  time_t midnight(void)
  {
    return t − (t % SecondsPerDay);
  }     /* Increment or decrement by a specified number of days. The time of the result is always
       midnight on the requested day, but the input need not be set to midnight. */
  void dayStep(int days)
  {
    t = ((t/SecondsPerDay) + days) * SecondsPerDay;
  }     /* Obtain midnight of next day, given a time in a given day */
  void nextDay(void)
  {
    dayStep(1);
```

```
  }    /* Obtain midnight of previous day, given a time in a given day */
  void previousDay(void)
  {
    dayStep(−1);
  }    /* Extract civil UTC date and time */
  void toUTC(int *year = Λ, int *month = Λ, int *mday = Λ, int *hour = Λ, int *min = Λ, int
      *sec = Λ);    /* Set a date from a civil UTC date and time */
  void fromUTC(int year = 1970, int month = 1, int mday = 1, int hour = 0, int min = 0, int
      sec = 0);    /* Return a string containing the date in ISO YYYY-MM-DD format */
  string dateToString(void);    /* Return a string containing the time as HH:MM:SS */
  string timeToString(void);
    /* Set a time from a string in the form YYYY-MM-DD HH:MM:SS. The time may be omitted. */
  void fromString(string s);    /* Return the Julian day and fraction for a system time */
  double toJulian(void);    /* Set a system time from a Julian day and fraction */
  void fromJulian(double jd);    /* Determine the day of the week. Sunday = 0,... Saturday = 6 */
  int weekday(void)
  {
    return jwday(toJulian( ));
  }    /* Return Sidereal Time at Greenwich for system time */
  double siderealTime(void)
  {
    return gmst(toJulian( ));
  }    /* Convert UTC date and time to Julian day and fraction */
  static double utctoj(long year, int mon, int mday, int hour = 0, int min = 0, int sec = 0);
    /* Convert Julian day to year, month, and day */
  static void jyear(double td, int *yy = Λ, int *mm = Λ, int *dd = Λ);
    /* Convert Julian day fraction to hours, minutes, and seconds */
  static void jhms(double j, int *h = Λ, int *m = Λ, int *s = Λ);
    /* Obtain weekday from a Julian day. Sunday = 0,... Saturday = 6 */
  static int jwday(double j)
  {
    return ((int)(j + 1.5)) % 7;
  }    /* Sidereal time at Greenwich */
  static double gmst(double jd);
    /* Mean obliquity of the ecliptic in radians for a given Julian day */
  static double meanObliquityOfEcliptic(double jd);
    /* Nutation in longitude and obliquity of the ecliptic for Julian day jd */
  static void nutation(double jd, double *deltaPsi, double *deltaEpsilon);
    /* Equation of time for a given Julian day jd */
  static double equationTime(double jd);    /* Calculate the geocentric position of the Sun */
  static void sunpos(double jd, bool apparent, double *ra, double *dec, double *rv, double
      *slong);    /* Calculate Moon and Sun information */
  static void moon_and_sun(double pdate, double *phaseang, double *pphase, double *mage, double
      *dist, double *angdia, double *sudist, double *suangdia);
    /* Solve the equation of Kepler */
  static double kepler(double m, double ecc);
  static void moonphases(double sdate, double phases[5]);    /* Find Moon phases */
protected:    /* Store an int value if the result pointer is non-Λ */
  static void store_if(int *p, const int v)
  {
```

```
    if (p ≠ Λ) {
      *p = v;
    }
  }    /* Store a double value if the result pointer is non-Λ */
  static void dstore_if(double *p, const double v)
  {
    if (p ≠ Λ) {
      *p = v;
    }
  }
  static double meanphase(double sdate, double k);    /* Find time of mean new Moon */
  static double truephase(double k, double phase);    /* Find time of Moon phase (k) */
};
```

9. The *toUTC* method returns the Coordinated Universal Time (UTC) corresponding to the current value of a **systemtime**. Any of the result pointers may be Λ (their default value if omitted) in which case no result is stored. The date is returned in "civil" form, with the year in A.D. and the month ranging from 1 to 12.

⟨ Time and date utilities 9 ⟩ ≡

```
void systemtime::toUTC(int *year, int *month, int *mday, int *hour, int *min, int *sec)
{
  struct tm *gt = gmtime(&t);
  store_if(year, gt→tm_year + 1900);
  store_if(month, gt→tm_mon + 1);
  store_if(mday, gt→tm_mday);
  store_if(hour, gt→tm_hour);
  store_if(min, gt→tm_min);
  store_if(sec, gt→tm_sec);
}
```

See also sections 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, and 27.

This code is used in section 5.

10. The *fromUTC* method sets the time and date from a "civil" UTC date and time, as described above for *toUTC*. The time arguments default to 0 and may be omitted if the time at midnight for a given date is required.

⟨ Time and date utilities 9 ⟩ +≡

```
void systemtime::fromUTC(int year, int month, int mday, int hour, int min, int sec)
{
  double jd;
  const double J1970 = 2440587.5;
  jd = utctoj(year, month, mday, hour, min, sec);
  t = (time_t)(((jd - J1970) * (60 * 60 * 24)) + 0.5);
}
```

11. The *dateToString* method returns a string containing the date in ISO-8601 YYYY-MM-DD format.

```
⟨ Time and date utilities 9 ⟩ +≡
  string systemtime::dateToString(void)
  {
    int year, month, day;
    char edate[16];
    toUTC(&year, &month, &day);
    sprintf(edate, "%04d-%02d-%02d", year, month, day);
    return string(edate);
  }
```

12. *timeToString* returns a string containing the time as `HH:MM:SS`.

```
⟨ Time and date utilities 9 ⟩ +≡
  string systemtime::timeToString(void)
  {
    int hour, min, sec;
    char etime[16];
    toUTC(Λ, Λ, Λ, &hour, &min, &sec);
    sprintf(etime, "%02d:%02d:%02d", hour, min, sec);
    return string(etime);
  }
```

13. You can set the date and time to the value given by a string of the form "`YYYY-MM-DD HH:MM:SS`" with the *fromString* method. You may omit the time if you wish the time at midnight of the given day.

```
⟨ Time and date utilities 9 ⟩ +≡
  void systemtime::fromString(string s)
  {
    int year = 1970, month = 1, mday = 1, hour = 0, min = 0, sec = 0, n;
    char d1, d2, d3, d4, d5;
    n = sscanf(s.c_str(), "%d%c%d%c%d%c%d%c%d", &year, &d1, &month, &d2, &mday, &d3, &hour,
        &d4, &min, &d5, &sec);
    if (n < 5) {
      throw (invalid_argument("Invalid date format"));
    }
    fromUTC(year, month, mday, hour, min, sec);
  }
```

14. The most convenient way to do arithmetic with date and time quantities without worrying about time zones, restricted range of computer representation, etc. is to convert them to Julian day number and fraction representation. The static *utctoj* method takes a date and time and returns a **double** containing the corresponding Julian day and fraction. The canonical 1582 date for adoption of the Gregorian calendar is assumed; dates prior to that are assumed to be in the Julian system.

```
⟨ Time and date utilities 9 ⟩ +≡
  double systemtime::utctoj(long year, int mon, int mday, int hour, int min, int sec)
  {     /* Algorithm as given in Meeus, Astronomical Algorithms, Chapter 7, page 61 */
    int a, b, m;
    long y;
    mon--;
    assert(mon ≥ 0 ∧ mon < 12);
    assert(mday > 0 ∧ mday < 32);
    assert(hour ≥ 0 ∧ hour < 24);
    assert(min ≥ 0 ∧ min < 60);
    assert(sec ≥ 0 ∧ sec < 60);
    m = mon + 1;
    y = year;
    if (m ≤ 2) {
      y--;
      m += 12;
    }     /* Determine whether date is in Julian or Gregorian calendar based on canonical date of calendar
          reform. */
    if ((year < 1582) ∨ ((year ≡ 1582) ∧ ((mon < 9) ∨ (mon ≡ 9 ∧ mday < 5)))) {
      b = 0;
    }
    else {
      a = ((int)(y/100));
      b = 2 − a + (a/4);
    }
    return (((long)(365.25 * (y + 4716))) + ((int)(30.6001 * (m + 1))) + mday + b − 1524.5) + ((sec +
        60L * (min + 60L * hour))/86400.0);
  }
```

15. The static *jyear* method converts a Julian day number to the year, month, and day. Omitted arguments are not stored. Julian day numbers representing dates of October 15, 1582 and later are returned in the Gregorian calendar; earlier dates are returned in the Julian calendar.

```
⟨ Time and date utilities 9 ⟩ +≡
  void systemtime::jyear(double td, int *yy, int *mm, int *dd)
  {
    double z, f, a, alpha, b, c, d, e;
    td += 0.5;
    z = floor(td);
    f = td - z;
    if (z < 2299161.0) {
      a = z;
    }
    else {
      alpha = floor((z - 1867216.25)/36524.25);
      a = z + 1 + alpha - floor(alpha/4);
    }
    b = a + 1524;
    c = floor((b - 122.1)/365.25);
    d = floor(365.25 * c);
    e = floor((b - d)/30.6001);
    store_if(dd, (int)(b - d - floor(30.6001 * e) + f));
    store_if(mm, (int)((e < 14) ? (e - 1) : (e - 13)));
    store_if(yy, (int)((*mm > 2) ? (c - 4716) : (c - 4715)));
  }
```

16. The static *jhms* method converts the Julian day fraction to UTC hours, minutes, and seconds. Note that Julian day numbers begin at *noon* on the prime meridian.

```
⟨ Time and date utilities 9 ⟩ +≡
  void systemtime::jhms(double j, int *h, int *m, int *s)
  {
    long ij;
    j += 0.5;       /* Astronomical to civil */
    ij = (long)((j - floor(j)) * 86400.0);
    store_if(h, (int)(ij/3600L));
    store_if(m, (int)((ij/60L) % 60L));
    store_if(s, (int)(ij % 60L));
  }
```

17. The *toJulian* and *fromJulian* methods allow retrieving the Julian day and fraction corresponding to a system date and setting a system date from a Julian day number.

```
⟨ Time and date utilities 9 ⟩ +≡
  double systemtime::toJulian(void)
  {
    int year, month, mday, hour, min, sec;
    toUTC(&year, &month, &mday, &hour, &min, &sec);
    return utctoj(year, month, mday, hour, min, sec);
  }
  void systemtime::fromJulian(double jd)
  {
    int year, month, mday, hour, min, sec;
    jyear(jd, &year, &month, &mday);
    jhms(jd, &hour, &min, &sec);
    fromUTC(year, month, mday, hour, min, sec);
  }
```

18. When exploring interested in periodicities related to fixed objects in the sky, you need the sidereal time (at which objects outside the Solar System are at the same right ascension on any given day) corresponding to a civil date. The *gmst* method returns the sidereal time at the prime meridian (Greenwich) for the given Julian day *jd*. The time is returned as a **double** in $0 \leq hours < 24$.

```
⟨ Time and date utilities 9 ⟩ +≡
  double systemtime::gmst(double jd)
  {
    double t, theta0;
    t = ((floor(jd + 0.5) − 0.5) − J2000)/JulianCentury;
    theta0 = d_m_s_to_decimal(6, 41, 50.54841) + (d_m_s_to_decimal(0, 0,
        8640184.812866) ∗ t) + (d_m_s_to_decimal(0, 0, 0.093104) ∗ t ∗ t) + (d_m_s_to_decimal(0, 0,
        0.0000062) ∗ t ∗ t ∗ t);
    t = (jd + 0.5) − (floor(jd + 0.5));
    theta0 += (t ∗ 24) ∗ 1.00273790935;
    theta0 = (theta0 − 24.0 ∗ (floor(theta0/24.0)));
    return theta0;
  }
```

19. Computer clocks keep "civil time", which is based on the *mean solar day* of 24 hours. Because the Earth's orbit is elliptical and is perturbed by the gravity of the Moon and the other planets, the position of the Sun varies with respect to the mean solar day: for part of the year the Sun will "run fast" and cross the meridian before local civil noon, and in the balance of the year it will "run slow" and cross the meridian after civil noon. The relationship between civil time and the apparent position of the Sun is called the *Equation of time*, and varies slowly over the centuries.

We use the approximation of the equation of time given by W.M. Smart, *Text-Book on Spherical Astronomy*, Cambridge University Press, 1956, p. 149:

$$E = y \sin 2L_0 - 2e \sin M + 4ey \sin M \cos 2L_0 - \frac{1}{2}y^2 \sin 4L_0 - \frac{5}{4}e^2 \sin 2M$$

where E is in radians. To obtain the equation of time as a fraction of an hour (the absolute value of the equation of time is always less than 20 minutes), convert to degrees and divide by 15 (360/24). Quantities in the equation are as follows:

y	$\tan^2 \frac{\varepsilon}{2}$, ε = obliquity of the ecliptic
L_0	Mean longitude of the Sun
e	Eccentricity of Earth's orbit
M	Mean anomaly of the Sun

⟨ Time and date utilities 9 ⟩ +≡

```
double systemtime::equationTime(double jd)
{
  double T, tau, eps, deltaEps, y, L0, e, M, E;
  T = (jd - J2000)/JulianCentury;      /* Julian centuries from J2000 */
  tau = T/10;      /* Julian millennia from J2000 */
  eps = meanObliquityOfEcliptic(jd);
  nutation(jd, Λ, &deltaEps);
  y = eps + deltaEps;      /* y = tan^2 ε/2 */
  y = tan(y/2);
  y *= y;
  L0 = 280.4664567 + (360007.6982779 * tau) + (0.03032028 * tau * tau) + ((tau * tau * tau)/49931) -
       ((tau * tau * tau * tau)/15300) - ((tau * tau * tau * tau * tau)/2000000);
  L0 = fixangle(L0);
  e = 0.016708634 - (0.000042037 * T) - (0.0000001267 * T * T);
  M = 357.52911 + (35999.05029 * T) - (0.0001537 * T * T);
  E = y * sin(dtr(2 * L0)) - (2 * e * sin(dtr(M))) + (4 * e * y * sin(dtr(M)) * cos(dtr(2 * L0))) - (((y * y) *
       sin(dtr(4 * L0)))/2) - ((5 * (e * e) * sin(dtr(2 * M)))/4);
#if 0
  cout << "E␣=␣" << E << "␣␣degrees␣=␣" << rtd(E) << "␣␣min␣=␣" << ((rtd(E)/15) * 60) << "\n";
#endif      /* Return equation of time in seconds */
  return rtd(E) * (3600/15);
}
```

20. Calculate the mean obliquity of the ecliptic in *radians* for a given Julian date. This uses Laskar's tenth-degree polynomial fit[1] which is accurate to within 0.01 arc second between A.D. 1000 and A.D. 3000, and within a few seconds of arc for ±10000 years around A.D. 2000. If we're outside the range in which this fit is valid (deep time) we simply return the J2000 value of the obliquity, which happens to be almost precisely the mean.

```
⟨ Time and date utilities 9 ⟩ +≡
  double systemtime::meanObliquityOfEcliptic(double jd)
  {
    ⟨ Coefficients for the obliquity of the ecliptic 29 ⟩;
    double eps = 23 + (26/60.0) + (21.448/3600.0), u, v;
    int i;
    v = u = (jd − J2000)/(JulianCentury ∗ 100);
    if (fabs(u) < 1.0) {
      for (i = 0; i < 10; i++) {
        eps += oterms[i] ∗ v;
        v ∗= u;
      }
    }
    return dtr(eps);
  }
```

[1] Laskar, J., *Astronomy and Astrophysics*, **157**: 68 (1986).

21. Calculate the nutation in longitude ($\Delta\psi$) and obliquity of the ecliptic ($\Delta\varepsilon$) for a given Julian date *jd*. Results are returned through the pointers *deltaPsi* and *deltaEpsilon*, both in radians. If you only require one of these values, you may supply a Λ pointer for the other.

The *ta* array contains the parameters of the periodic terms as follows:

ta[0]	D	Mean elongation of the Moon from the Sun
ta[1]	M	Mean anomaly of the Sun (Earth)
ta[2]	M'	Mean anomaly of the Moon
ta[3]	F	Moon's argument of latitude
ta[4]	Ω	Longitude of the ascending node of the Moon's mean orbit measured from the mean equinox of the date

```
#define NUTERMS 63 /* Number of periodic terms for nutation */
⟨ Time and date utilities 9 ⟩ +≡
  void systemtime::nutation(double jd, double *deltaPsi, double *deltaEpsilon)
  {
    int i, j;
    double t = (jd − J2000)/JulianCentury, t2, t3, to10;
    double ta[5];
    double dp = 0, de = 0, ang;
    ⟨ Nutation argument multiple table 30 ⟩;
    ⟨ Nutation coefficient table 31 ⟩;
    t3 = t * (t2 = t * t);
    ta[0] = dtr(297.85036 + 445267.11148 * t − 0.0019142 * t2 + t3/189474.0);
    ta[1] = dtr(357.52772 + 35999.05034 * t − 0.0001603 * t2 − t3/300000.0);
    ta[2] = dtr(134.96298 + 477198.867398 * t + 0.0086972 * t2 + t3/56250.0);
    ta[3] = dtr(93.27191 + 483202.017538 * t − 0.0036825 * t2 + t3/327270.0);
    ta[4] = dtr(125.04452 − 1934.136261 * t + 0.0020708 * t2 + t3/450000.0);
        /* Range reduce the angles in case the sine and cosine functions don't do it as accurately or quickly. */
    for (i = 0; i < 5; i++) {
      ta[i] = fixangr(ta[i]);
    }
    to10 = t/10.0;
    for (i = 0; i < NUTERMS; i++) {
      ang = 0;
      for (j = 0; j < 5; j++) {
        if (nutArgMult[i][j] ≠ 0) {
          ang += nutArgMult[i][j] * ta[j];
        }
      }
      dp += (nutArgCoeff[i][0] + nutArgCoeff[i][1] * to10) * sin(ang);
      de += (nutArgCoeff[i][2] + nutArgCoeff[i][3] * to10) * cos(ang);
    }     /* Return the result, converting from ten thousandths of arc seconds to radians in the process. */
    dstore_if(deltaPsi, dtr(dp/(3600.0 * 10000.0)));
    dstore_if(deltaEpsilon, dtr(de/(3600.0 * 10000.0)));
  }
```

22. Calculate position of the Sun. *jd* is the Julian date of the instant for which the position is desired and *apparent* should be *true* if the apparent position (corrected for nutation and aberration) is desired. The Sun's geocentric co-ordinates are returned in *ra* and *dec*, both specified in *radians* (convert *ra* to degrees and divide by 15 to obtain hours). The radius vector to the Sun in astronomical units is returned in *rv* and the Sun's longitude (true or apparent, as desired) is returned as *radians* in *slong*. You may pass Λ pointers for any results you aren't interested in.

```
⟨ Time and date utilities 9 ⟩ +≡
  void systemtime::sunpos(double jd, bool apparent, double *ra, double *dec, double *rv, double
        *slong)
  {
    double t, t2, C, l, m, e, v, theta, omega, eps;
      /* Time, in Julian centuries of 36525 ephemeris days, measured from the epoch J2000.0. */
    t = (jd - J2000)/JulianCentury;
    t2 = t * t;      /* Geometric mean longitude of the Sun, referred to the mean equinox of the date. */
    l = fixangle(280.46646 + (36000.76983 * t) + (0.0003032 * t2));      /* Sun's mean anomaly. */
    m = fixangle(357.52911 + (35999.05029 * t) - (0.0001537 * t2));
      /* Eccentricity of the Earth's orbit. */
    e = 0.016708634 - (0.000042037 * t) - (0.0000001267 * t2);      /* Sun's equation of the centre. */
    C = ((1.914602 - (0.004817 * t) - (0.000014 * t2)) * sin(dtr(m))) + ((0.019993 - (0.000101 * t)) * sin(2 *
        dtr(m))) + (0.000289 * sin(3 * dtr(m)));      /* Sun's true longitude. */
    theta = l + C;      /* True anomaly */
    v = m + C;      /* Obliquity of the ecliptic. */
    eps = rtd(meanObliquityOfEcliptic(jd));      /* Corrections for Sun's apparent longitude, if desired. */
    if (apparent) {
      omega = fixangle(125.04 - 1934.136 * t);
      theta = theta - 0.00569 - 0.00478 * sin(dtr(omega));
      eps += 0.00256 * cos(dtr(omega));
    }      /* Return Sun's longitude and radius vector */
    dstore_if(slong, dtr(theta));
    dstore_if(rv, (1.000001018 * (1 - (e * e)))/(1 + e * cos(dtr(v))));      /* Determine solar co-ordinates. */
    dstore_if(ra, fixangr(atan2(cos(dtr(eps)) * sin(dtr(theta)), cos(dtr(theta)))));
    dstore_if(dec, asin(sin(dtr(eps)) * sin(dtr(theta))));
  }
```

23. Compute properties of the Sun and Moon at Julian date *jd*. Results are returned through pointer arguments to **double** quantities—you may pass Λ pointers for any items you don't wish returned. Values returned are as follows:

`phaseang`	Moon terminator phase angle (0–1)
`pphase`	Illuminated fraction
`mage`	Age of moon in days and fraction
`dist`	Geocentric distance of Moon in kilometres
`angdia`	Angular diameter of Moon in radians
`sudist`	Geocentric distance to Sun in kilometres
`suangdia`	Sun's angular diameter in radians

```
⟨ Time and date utilities 9 ⟩ +≡
  void systemtime::moon_and_sun(double jd, double *phaseang, double *pphase, double
          *mage, double *dist, double *angdia, double *sudist, double *suangdia)
  {
    double Day, N, M, Ec, Lambdasun, ml, MM, MN, Ev, Ae, A3, MmP, mEc, A4, lP, Varia,
        lPP, NP, y, x, Lambdamoon, MoonAge, MoonPhase, MoonDist, MoonDFrac, MoonAng, F,
        SunDist, SunAng;
    static const double AstronomicalUnit = 149597870.0,      /* Astronomical unit in kilometres */
    SunSMAX = (AstronomicalUnit * 1.000001018),      /* Semi-major axis of Earth's orbit */
    epoch = 2444238.5,      /* 1980 January 0.0 */      /* Constants defining the Sun's apparent orbit */
    elonge = 278.833540,      /* Ecliptic longitude of the Su at epoch 1980.0 */
    elongp = 282.596403,      /* Ecliptic longitude of the Sun at perigee */
    eccent = 0.016718,      /* Eccentricity of Earth's orbit */
    sunangsiz = 0.533128,      /* Sun's angular size, degrees, at semi-major axis distance */
      /* Elements of the Moon's orbit, epoch 1980.0 */
    mmlong = 64.975464,      /* Moon's mean longitude at the epoch */
    mmlongp = 349.383063,      /* Mean longitude of the perigee at the epoch */
    mlnode = 151.950429,      /* Mean longitude of the node at the epoch */
    minc = 5.145396,      /* Inclination of the Moon's orbit */
    mecc = 0.054900,      /* Eccentricity of the Moon's orbit */
    mangsiz = 0.5181,      /* Moon's angular size at distance a from Earth */
    msmax = 384401.0;      /* Semi-major axis of Moon's orbit in km */      /* mparallax = 0.9507; /*
        Parallax at distance a from Earth */ */      /* Calculation of the Sun's position */
    Day = jd - epoch;      /* Date within epoch */
    N = fixangle((360/365.2422) * Day);      /* Mean anomaly of the Sun */
    M = fixangle(N + elonge - elongp);      /* Convert from perigee co-ordinates to epoch 1980.0 */
    Ec = kepler(M, eccent);      /* Solve equation of Kepler */
    Ec = sqrt((1 + eccent)/(1 - eccent)) * tan(Ec/2);
    Ec = 2 * rtd(atan(Ec));      /* True anomaly */
    Lambdasun = fixangle(Ec + elongp);      /* Sun's geocentric ecliptic longitude */
      /* Orbital distance factor */
    F = ((1 + eccent * cos(dtr(Ec)))/(1 - eccent * eccent));
    SunDist = SunSMAX/F;      /* Distance to Sun in km */
    SunAng = F * sunangsiz;      /* Sun's angular size in degrees */
      /* Calculation of the Moon's position */      /* Moon's mean longitude */
    ml = fixangle(13.1763966 * Day + mmlong);      /* Moon's mean anomaly */
    MM = fixangle(ml - 0.1114041 * Day - mmlongp);      /* Moon's ascending node mean longitude */
    MN = fixangle(mlnode - 0.0529539 * Day);      /* Evection */
    Ev = 1.2739 * sin(dtr(2 * (ml - Lambdasun) - MM));      /* Annual equation */
    Ae = 0.1858 * sin(dtr(M));      /* Correction term */
    A3 = 0.37 * sin(dtr(M));      /* Corrected anomaly */
```

```
  MmP = MM + Ev - Ae - A3;      /* Correction for the equation of the centre */
  mEc = 6.2886 * sin(dtr(MmP));      /* Another correction term */
  A4 = 0.214 * sin(dtr(2 * MmP));      /* Corrected longitude */
  lP = ml + Ev + mEc - Ae + A4;      /* Variation */
  Varia = 0.6583 * sin(dtr(2 * (lP - Lambdasun)));      /* True longitude */
  lPP = lP + Varia;      /* Corrected longitude of the node */
  NP = MN - 0.16 * sin(dtr(M));      /* Y inclination coordinate */
  y = sin(dtr(lPP - NP)) * cos(dtr(minc));      /* X inclination coordinate */
  x = cos(dtr(lPP - NP));      /* Ecliptic longitude */
  Lambdamoon = rtd(atan2(y, x));
  Lambdamoon += NP;      /* Ecliptic latitude */
    /* BetaM = rtd(asin(sin(dtr(lPP - NP)) * sin(dtr(minc)))); */
    /* Calculation of the phase of the Moon */      /* Age of the Moon in degrees */
  MoonAge = lPP - Lambdasun;      /* Phase of the Moon */
  MoonPhase = (1 - cos(dtr(MoonAge)))/2;
    /* Calculate distance of moon from the centre of the Earth */
  MoonDist = (msmax * (1 - mecc * mecc))/(1 + mecc * cos(dtr(MmP + mEc)));
    /* Calculate Moon's angular diameter */
  MoonDFrac = MoonDist/msmax;
  MoonAng = mangsiz/MoonDFrac;      /* Calculate Moon's parallax */
    /* MoonPar = mparallax / MoonDFrac; */
  dstore_if(pphase, MoonPhase);
  dstore_if(mage, SynMonth * (fixangle(MoonAge)/360.0));
  dstore_if(dist, MoonDist);
  dstore_if(angdia, dtr(MoonAng));
  dstore_if(sudist, SunDist);
  dstore_if(suangdia, dtr(SunAng));
  dstore_if(phaseang, fixangle(MoonAge)/360.0);
}
```

24. To find the position of a body in an elliptical orbit from its orbital elements, we must solve the *equation of Kepler*

$$E = M + e \sin E$$

given the *mean anomaly* M of the object and the eccentricity e of its orbit. There is no closed form solution for this equation, so we solve it iteratively to within the accuracy `EPSILON` specified in the code. The solution for E is returned in *radians*.

```
⟨ Time and date utilities 9 ⟩ +≡
  double systemtime::kepler(double m, double ecc)
  {
    double E, delta;
    static const double EPSILON = 1 · 10^-6;
    E = m = dtr(m);
    do {
      delta = E - ecc * sin(E) - m;
      E -= delta/(1 - ecc * cos(E));
    } while (fabs(delta) > EPSILON);
    return E;
  }
```

25. Calculate time of the mean new Moon for a given base date *sdate*. The argument k is the precomputed synodic month index, given by:

$$k = (year - 1900) \times 12.3685$$

where *year* is expressed as a year and fraction.

```
⟨ Time and date utilities 9 ⟩ +≡
  double systemtime::meanphase(double sdate, double k)
  {
    double t, t2, t3, nt1;      /* Time in Julian centuries from 1900 January 0.5 */
    t = (sdate - 2415020.0)/JulianCentury;
    t2 = t * t;      /* Square for frequent use */
    t3 = t2 * t;      /* Cube for frequent use */
    nt1 = 2415020.75933 + SynMonth * k + 0.0001178 * t2 - 0.000000155 * t3 + 0.00033 * sin(dtr(166.56 +
        132.87 * t - 0.009173 * t2));
    return nt1;
  }
```

26. Given a k value used to determine the mean phase of the new moon, and a phase selector (0.0, 0.25, 0.5, 0.75), obtain the Julian day number of the true, corrected phase time.

```
⟨ Time and date utilities 9 ⟩ +≡
  double systemtime::truephase(double k, double phase)
  {
    double t, t2, t3, pt, m, mprime, f;
    k += phase;      /* Add phase to new moon time */
    t = k/1236.85;      /* Time in Julian centuries from 1900 January 0.5 */
    t2 = t * t;      /* Square for frequent use */
    t3 = t2 * t;      /* Cube for frequent use */
    pt = 2415020.75933      /* Mean time of phase */
    + SynMonth * k + 0.0001178 * t2 - 0.000000155 * t3 + 0.00033 * sin(dtr(166.56 + 132.87 * t - 0.009173 * t2));
    m = 359.2242      /* Sun's mean anomaly */
    + 29.10535608 * k - 0.0000333 * t2 - 0.00000347 * t3;
    mprime = 306.0253      /* Moon's mean anomaly */
    + 385.81691806 * k + 0.0107306 * t2 + 0.00001236 * t3;
    f = 21.2964      /* Moon's argument of latitude */
    + 390.67050646 * k - 0.0016528 * t2 - 0.00000239 * t3;
    if ((phase < 0.01) ∨ (fabs(phase - 0.5) < 0.01)) {      /* Corrections for New and Full Moon */
      pt += (0.1734 - 0.000393 * t) * sin(dtr(m)) + 0.0021 * sin(dtr(2 * m)) - 0.4068 * sin(dtr(mprime)) + 0.0161 *
          sin(dtr(2 * mprime)) - 0.0004 * sin(dtr(3 * mprime)) + 0.0104 * sin(dtr(2 * f)) - 0.0051 * sin(dtr(m +
          mprime)) - 0.0074 * sin(dtr(m - mprime)) + 0.0004 * sin(dtr(2 * f + m)) - 0.0004 * sin(dtr(2 * f - m)) -
          0.0006 * sin(dtr(2 * f + mprime)) + 0.0010 * sin(dtr(2 * f - mprime)) + 0.0005 * sin(dtr(m + 2 * mprime));
    }
    else if ((fabs(phase - 0.25) < 0.01 ∨ (fabs(phase - 0.75) < 0.01))) {
      pt += (0.1721 - 0.0004 * t) * sin(dtr(m)) + 0.0021 * sin(dtr(2 * m)) - 0.6280 * sin(dtr(mprime)) +
          0.0089 * sin(dtr(2 * mprime)) - 0.0004 * sin(dtr(3 * mprime)) + 0.0079 * sin(dtr(2 * f)) - 0.0119 *
          sin(dtr(m + mprime)) - 0.0047 * sin(dtr(m - mprime)) + 0.0003 * sin(dtr(2 * f + m)) - 0.0004 *
          sin(dtr(2 * f - m)) - 0.0006 * sin(dtr(2 * f + mprime)) + 0.0021 * sin(dtr(2 * f - mprime)) + 0.0003 *
          sin(dtr(m + 2 * mprime)) + 0.0004 * sin(dtr(m - 2 * mprime)) - 0.0003 * sin(dtr(2 * m + mprime));
      if (phase < 0.5)      /* First quarter correction */
        pt += 0.0028 - 0.0004 * cos(dtr(m)) + 0.0003 * cos(dtr(mprime));
      else      /* Last quarter correction */
        pt += -0.0028 + 0.0004 * cos(dtr(m)) - 0.0003 * cos(dtr(mprime));
    }
    return pt;
  }
```

27. Find time of phases of the moon which surround the given Julian day *sdate*. Five phases are found, starting and ending with the new moons which bound the current lunation. The Julian day and fraction for are stored in the *phases* array as follows:

`phases[0]`	Previous new Moon
`phases[1]`	First quarter
`phases[2]`	Full Moon
`phases[3]`	Last quarter
`phases[4]`	Next new Moon

```
⟨ Time and date utilities 9 ⟩ +≡
  void systemtime::moonphases(double sdate, double phases[5])
  {
    double adate, k1, k2;
    int yy, mm, dd;
    adate = sdate − 45;
    jyear(adate, &yy, &mm, &dd);
    k1 = floor((yy + ((mm − 1) ∗ (1.0/12.0)) − 1900) ∗ 12.3685);
    adate = meanphase(adate, k1);
    while (true) {
      adate += SynMonth;
      k2 = k1 + 1;
      if (truephase(k1, 0.0) ≤ sdate ∧ truephase(k2, 0.0) > sdate) {
        break;
      }
      k1 = k2;
    }
    phases[0] = truephase(k1, 0.0);
    phases[1] = truephase(k1, 0.25);
    phases[2] = truephase(k1, 0.5);
    phases[3] = truephase(k1, 0.75);
    phases[4] = truephase(k2, 0.0);
  }
```

28. Coefficients for positional astronomy calculations.

Calculation of quantities for positional astronomy often involves evaluating a large number of periodic terms of a perturbation series. The following sections provide these "magic numbers" for the calculation functions above. We banish them to the end since, notwithstanding their importance in obtaining accurate results and the large intellectual investment they represent, they are less than gripping reading.

29. The following are the coefficients, in arc-seconds, for Lasker's 10th degree polynomial expression for the obliquity of the ecliptic as computed in *meanObliquityOfEcliptic*.

```
⟨ Coefficients for the obliquity of the ecliptic 29 ⟩ ≡
  static double oterms[10] = {
  Asec(−4680.93),
  Asec(−1.55),
  Asec(1999.25),
  Asec(−51.38),
  Asec(−249.67),
  Asec(−39.05),
  Asec(7.12),
  Asec(27.87),
  Asec(5.79),
  Asec(2.45)
  };
```

This code is used in section 20.

30. The following table supplies the multiples of quantities used to evaluate the nutation in longitude ($\Delta\psi$) and obliquity ($\Delta\varepsilon$). The integers in each row represent the multiples of D, M, M', F, and Ω to be multiplied by the coefficients in the *nutArgCoeff* table below. These quantities are described in the documentation of the *nutation* method above.

⟨ Nutation argument multiple table 30 ⟩ ≡

```
static signed char nutArgMult[NUTERMS][5] = {
{0, 0, 0, 0, 1},
{−2, 0, 0, 2, 2},
{0, 0, 0, 2, 2},
{0, 0, 0, 0, 2},
{0, 1, 0, 0, 0},
{0, 0, 1, 0, 0},
{−2, 1, 0, 2, 2},
{0, 0, 0, 2, 1},
{0, 0, 1, 2, 2},
{−2, −1, 0, 2, 2},
{−2, 0, 1, 0, 0},
{−2, 0, 0, 2, 1},
{0, 0, −1, 2, 2},
{2, 0, 0, 0, 0},
{0, 0, 1, 0, 1},
{2, 0, −1, 2, 2},
{0, 0, −1, 0, 1},
{0, 0, 1, 2, 1},
{−2, 0, 2, 0, 0},
{0, 0, −2, 2, 1},
{2, 0, 0, 2, 2},
{0, 0, 2, 2, 2},
{0, 0, 2, 0, 0},
{−2, 0, 1, 2, 2},
{0, 0, 0, 2, 0},
{−2, 0, 0, 2, 0},
{0, 0, −1, 2, 1},
{0, 2, 0, 0, 0},
{2, 0, −1, 0, 1},
{−2, 2, 0, 2, 2},
{0, 1, 0, 0, 1},
{−2, 0, 1, 0, 1},
{0, −1, 0, 0, 1},
{0, 0, 2, −2, 0},
{2, 0, −1, 2, 1},
{2, 0, 1, 2, 2},
{0, 1, 0, 2, 2},
{−2, 1, 1, 0, 0},
{0, −1, 0, 2, 2},
{2, 0, 0, 2, 1},
{2, 0, 1, 0, 0},
{−2, 0, 2, 2, 2},
{−2, 0, 1, 2, 1},
{2, 0, −2, 0, 1},
{2, 0, 0, 0, 1},
{0, −1, 1, 0, 0},
```

```
  {−2, −1, 0, 2, 1},
  {−2, 0, 0, 0, 1},
  {0, 0, 2, 2, 1},
  {−2, 0, 2, 0, 1},
  {−2, 1, 0, 2, 1},
  {0, 0, 1, −2, 0},
  {−1, 0, 1, 0, 0},
  {−2, 1, 0, 0, 0},
  {1, 0, 0, 0, 0},
  {0, 0, 1, 2, 0},
  {−1, −1, 1, 0, 0},
  {0, 1, 1, 0, 0},
  {0, −1, 1, 2, 2},
  {2, −1, −1, 2, 2},
  {0, 0, −2, 2, 2},
  {0, 0, 3, 2, 2},
  {2, −1, 0, 2, 2}
  };
```

This code is used in section 21.

31. This table, used in conjunction with the one above, supplies the coefficients used to evaluate ($\Delta\psi$) and ($\Delta\varepsilon$). The first two terms in each row are constant and time-dependent coefficients for ($\Delta\psi$). the second two the constant and time-dependent coefficients for ($\Delta\varepsilon$). The time-dependent coefficients are multiplied by T, the time in Julian centuries from the epoch J2000.0. Constant terms are in units of 0.0001" and time-dependent terms in units of 0.00001".

⟨ Nutation coefficient table 31 ⟩ ≡ /* Periodic terms for nutation in longitude ($\Delta\psi$) and obliquity ($\Delta\varepsilon$) as given in table 22.A of Meeus, *Astronomical Algorithms*, second edition. The comments give the multiples of D, M, M', F, and Ω multiplied by each term as given in the *nutArgMult* table above. */

```
static long nutArgCoeff[NUTERMS][4] = {
{−171996, −1742, 92095, 89},      /* 0, 0, 0, 0, 1 */
{−13187, −16, 5736, −31},      /* -2, 0, 0, 2, 2 */
{−2274, −2, 977, −5},      /* 0, 0, 0, 2, 2 */
{2062, 2, −895, 5},      /* 0, 0, 0, 0, 2 */
{1426, −34, 54, −1},      /* 0, 1, 0, 0, 0 */
{712, 1, −7, 0},      /* 0, 0, 1, 0, 0 */
{−517, 12, 224, −6},      /* -2, 1, 0, 2, 2 */
{−386, −4, 200, 0},      /* 0, 0, 0, 2, 1 */
{−301, 0, 129, −1},      /* 0, 0, 1, 2, 2 */
{217, −5, −95, 3},      /* -2, -1, 0, 2, 2 */
{−158, 0, 0, 0},      /* -2, 0, 1, 0, 0 */
{129, 1, −70, 0},      /* -2, 0, 0, 2, 1 */
{123, 0, −53, 0},      /* 0, 0, -1, 2, 2 */
{63, 0, 0, 0},      /* 2, 0, 0, 0, 0 */
{63, 1, −33, 0},      /* 0, 0, 1, 0, 1 */
{−59, 0, 26, 0},      /* 2, 0, -1, 2, 2 */
{−58, −1, 32, 0},      /* 0, 0, -1, 0, 1 */
{−51, 0, 27, 0},      /* 0, 0, 1, 2, 1 */
{48, 0, 0, 0},      /* -2, 0, 2, 0, 0 */
{46, 0, −24, 0},      /* 0, 0, -2, 2, 1 */
{−38, 0, 16, 0},      /* 2, 0, 0, 2, 2 */
{−31, 0, 13, 0},      /* 0, 0, 2, 2, 2 */
{29, 0, 0, 0},      /* 0, 0, 2, 0, 0 */
{29, 0, −12, 0},      /* -2, 0, 1, 2, 2 */
{26, 0, 0, 0},      /* 0, 0, 0, 2, 0 */
{−22, 0, 0, 0},      /* -2, 0, 0, 2, 0 */
{21, 0, −10, 0},      /* 0, 0, -1, 2, 1 */
{17, −1, 0, 0},      /* 0, 2, 0, 0, 0 */
{16, 0, −8, 0},      /* 2, 0, -1, 0, 1 */
{−16, 1, 7, 0},      /* -2, 2, 0, 2, 2 */
{−15, 0, 9, 0},      /* 0, 1, 0, 0, 1 */
{−13, 0, 7, 0},      /* -2, 0, 1, 0, 1 */
{−12, 0, 6, 0},      /* 0, -1, 0, 0, 1 */
{11, 0, 0, 0},      /* 0, 0, 2, -2, 0 */
{−10, 0, 5, 0},      /* 2, 0, -1, 2, 1 */
{−8, 0, 3, 0},      /* 2, 0, 1, 2, 2 */
{7, 0, −3, 0},      /* 0, 1, 0, 2, 2 */
{−7, 0, 0, 0},      /* -2, 1, 1, 0, 0 */
{−7, 0, 3, 0},      /* 0, -1, 0, 2, 2 */
{−7, 0, 3, 0},      /* 2, 0, 0, 2, 1 */
{6, 0, 0, 0},      /* 2, 0, 1, 0, 0 */
{6, 0, −3, 0},      /* -2, 0, 2, 2, 2 */
{6, 0, −3, 0},      /* -2, 0, 1, 2, 1 */
```

```
{−6, 0, 3, 0},      /* 2, 0, -2, 0, 1 */
{−6, 0, 3, 0},      /* 2, 0, 0, 0, 1 */
{5, 0, 0, 0},      /* 0, -1, 1, 0, 0 */
{−5, 0, 3, 0},      /* -2, -1, 0, 2, 1 */
{−5, 0, 3, 0},      /* -2, 0, 0, 0, 1 */
{−5, 0, 3, 0},      /* 0, 0, 2, 2, 1 */
{4, 0, 0, 0},      /* -2, 0, 2, 0, 1 */
{4, 0, 0, 0},      /* -2, 1, 0, 2, 1 */
{4, 0, 0, 0},      /* 0, 0, 1, -2, 0 */
{−4, 0, 0, 0},      /* -1, 0, 1, 0, 0 */
{−4, 0, 0, 0},      /* -2, 1, 0, 0, 0 */
{−4, 0, 0, 0},      /* 1, 0, 0, 0, 0 */
{3, 0, 0, 0},      /* 0, 0, 1, 2, 0 */
{−3, 0, 0, 0},      /* -1, -1, 1, 0, 0 */
{−3, 0, 0, 0},      /* 0, 1, 1, 0, 0 */
{−3, 0, 0, 0},      /* 0, -1, 1, 2, 2 */
{−3, 0, 0, 0},      /* 2, -1, -1, 2, 2 */
{−3, 0, 0, 0},      /* 0, 0, -2, 2, 2 */
{−3, 0, 0, 0},      /* 0, 0, 3, 2, 2 */
{−3, 0, 0, 0}      /* 2, -1, 0, 2, 2 */
};
```

This code is used in section 21.

32. Test program.

```
⟨ timedate_test.c 1 ⟩ +≡
  ⟨ Test program include files 35 ⟩;
  ⟨ Show how to call test program 34 ⟩;
  int main(int argc, char *argv[])
  {
    extern char *optarg;      /* Imported from getopt */
    extern int optind;
    int opt;
    ⟨ Process command-line options 33 ⟩;      /* Test time and date utilities */
    string s;
    systemtime td, td1;
    td.now();
    cout << td.dateToString() << " " << td.timeToString() << "\n";
    td1 = td;
    td1.nextDay();
    cout << "Tomorrow: " << td1.dateToString() << " " << td1.timeToString() << "\n";
    td1 = td;
    td1.previousDay();
    cout << "Yesterday: " << td1.dateToString() << " " << td1.timeToString() << "\n";
    td1 = td;
    td1.dayStep(7);
    cout << "Next week: " << td1.dateToString() << " " << td1.timeToString() << "\n";
    td.fromString("1981-04-12 13:51:18");
    cout << td.dateToString() << " " << td.timeToString() << "\n";
    td.fromString("2021-04-12");
    cout << td.dateToString() << " " << td.timeToString() << "\n";
    cout << "Weekday = " << systemtime::jwday(systemtime::utctoj(2001, 5, 12, 23, 59, 59)) << "\n";
    td.fromJulian(2436116.31);
    cout << td.dateToString() << " " << td.timeToString() << "\n";
    cout << "To Julian = " << td.toJulian() << " Weekday = " << td.weekday() << "\n";
    td.fromString("1987-04-10 19:21:00");
    cout << "Sidereal time = " << angle::degrees_to_d_m_s(td.siderealTime()) << "\n";
    cout.flush();
    int t1, t2, t3;
    systemtime::jyear(2436116.31, &t1, &t2, &t3);
    printf("%04d-%02d-%02d\n", t1, t2, t3);
    systemtime::jhms(systemtime::utctoj(1989, 11, 14, 16, 23, 18), &t1, &t2, &t3);
    printf("%02d:%02d:%02d\n", t1, t2, t3);
    {
      double j = 2446895.5;
      double o = angle::rtd(systemtime::meanObliquityOfEcliptic(j));
      double delpsi, deleps;
      cout << "meanObliquityOfEcliptic(" << ((long) j) << ") = " << angle::degrees_to_d_m_s(o) <<
          "\n";
      systemtime::nutation(j, &delpsi, &deleps);
      cout << "  delpsi = " << angle::degrees_to_d_m_s(angle::rtd(delpsi)) << " deleps = " <<
          angle::degrees_to_d_m_s(angle::rtd(deleps)) << "\n";
      double eOt = systemtime::equationTime(2448908.5);
```

```
    cout ≪ "Equation␣of␣time␣=␣" ≪ ((int)(eOt/60)) ≪ "␣minutes␣" ≪
        (eOt − (60 ∗ ((int)(eOt/60)))) ≪ "␣seconds\n";
    j = 2448908.5;
    double sura, sudec, surv, suslong;
    systemtime::sunpos(j, true, &sura, &sudec, &surv, &suslong);
    cout ≪ "Sun␣position:␣␣RA␣=␣" ≪ angle::rtd(sura) ≪ "␣␣Dec␣=␣" ≪ angle::rtd(sudec) ≪
        "␣␣Rv␣=␣" ≪ surv ≪ "␣␣Long␣=␣" ≪ angle::rtd(suslong) ≪ "\n";
  }
  {
    double phaseang, pphase, mage, dist, angdia, sudist, suangdia;
    systemtime::moon_and_sun(2452060.13874, &phaseang, &pphase, &mage, &dist, &angdia, &sudist,
        &suangdia);
    cout ≪ "Moon:␣␣Phaseang␣=␣" ≪ (phaseang ∗ 100) ≪ "␣␣Pphase␣=␣" ≪
        ((int)(pphase ∗ 100)) ≪ "%\n␣␣␣␣␣␣␣␣" ≪ "␣␣Moon␣age␣=␣" ≪ ((int)
        mage) ≪ "␣days,␣" ≪ angle::degrees_to_d_m_s((mage − ((int) mage)) ∗ 24) ≪ "␣␣Dist␣=␣" ≪
        dist ≪ "␣km\n" ≪ "Sun:␣␣␣␣Angdia␣=␣" ≪ angle::rtd(angdia) ≪ "␣␣Sudist␣=␣" ≪ ((int)
        sudist) ≪ "␣km" ≪ "␣␣Suangdia␣=␣" ≪ angle::rtd(suangdia) ≪ "\n";
  }
  {
    double phases[5];
    systemtime pt[5];
    int i;
    systemtime::moonphases(2452060.13874, phases);
    for (i = 0; i < 5; i++) {
      pt[i].fromJulian(phases[i]);
      cout ≪ "Phase␣" ≪ i ≪ ":␣" ≪ pt[i].dateToString( ) ≪ "␣" ≪ pt[i].timeToString( ) ≪ "\n";
    }
  }
  return 0;
}
```

33. We use *getopt* to process command line options. This permits aggregation of options without arguments and both `-d`*arg* and `-d` *arg* syntax.

```
⟨ Process command-line options 33 ⟩ ≡
  while ((opt = getopt(argc, argv, "nu-:")) ≠ −1) {
    switch (opt) {
    case 'u':     /* -u Print how-to-call information */
      case '?': usage( );
      return 0;
    case '-':     /* -- Extended options */
      switch (optarg[0]) {
      case 'c':     /* --copyright */
        cout ≪ "This␣program␣is␣in␣the␣public␣domain.\n";
        return 0;
      case 'h':     /* --help */
        usage( );
        return 0;
      case 'v':     /* --version */
        cout ≪ PRODUCT ≪ "␣" ≪ VERSION ≪ "\n";
        cout ≪ "Last␣revised:␣" ≪ REVDATE ≪ "\n";
        cout ≪ "The␣latest␣version␣is␣always␣available\n";
        cout ≪ "at␣http://www.fourmilab.ch/eggtools/eggshell\n";
        return 0;
      }
    }
  }
```

This code is used in section 32.

34. Procedure *usage* prints how-to-call information.

```
⟨ Show how to call test program 34 ⟩ ≡
  static void usage(void)
  {
    cout ≪ PRODUCT ≪ "␣␣--␣␣Analyse␣eggsummary␣files.␣␣Call:\n";
    cout ≪ "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣" ≪ PRODUCT ≪ "␣[options]␣[infile]␣[outfile]\n";
    cout ≪ "\n";
    cout ≪ "Options:\n";
    cout ≪ "␣␣␣␣␣␣␣␣␣␣␣␣--copyright␣␣␣␣␣␣␣␣Print␣copyright␣information\n";
    cout ≪ "␣␣␣␣␣␣␣␣␣␣␣␣-u,␣--help␣␣␣␣␣␣␣␣Print␣this␣message\n";
    cout ≪ "␣␣␣␣␣␣␣␣␣␣␣␣--version␣␣␣␣␣␣␣␣␣Print␣version␣number\n";
    cout ≪ "\n";
    cout ≪ "by␣John␣Walker\n";
    cout ≪ "http://www.fourmilab.ch/\n";
  }
```

This code is used in section 32.

35. We need the following definitions to compile the test program.

⟨ Test program include files 35 ⟩ ≡

```
#include "config.h"      /* Our configuration */      /* C++ include files */
#include <iostream>
#include <exception>
#include <stdexcept>
#include <string>
  using namespace std;
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#ifdef HAVE_GETOPT
#ifdef HAVE_UNISTD_H
#include <unistd.h>
#endif
#else
#include "getopt.h"      /* No system getopt—use our own */
#endif
#include "timedate.h"      /* Class definitions for this package */
```

This code is used in section 32.

36. Index. The following is a cross-reference table for `timedate`. Single-character identifiers are not indexed, nor are reserved words. Underlined entries indicate where an identifier was declared.

⟨ Angle utilities 7 ⟩ Used in section 5.
⟨ Application include files 4 ⟩ Used in section 2.
⟨ Class definitions 6, 8 ⟩ Used in section 3.
⟨ Class implementations 5 ⟩ Used in section 2.
⟨ Coefficients for the obliquity of the ecliptic 29 ⟩ Used in section 20.
⟨ Nutation argument multiple table 30 ⟩ Used in section 21.
⟨ Nutation coefficient table 31 ⟩ Used in section 21.
⟨ Process command-line options 33 ⟩ Used in section 32.
⟨ Show how to call test program 34 ⟩ Used in section 32.
⟨ Test program include files 35 ⟩ Used in section 32.
⟨ Time and date utilities 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27 ⟩ Used in section 5.
⟨ `timedate.h` 3 ⟩
⟨ `timedate_test.c` 1, 32 ⟩

TIMEDATE