1. Introduction.

EGGDATA

Egg Database Access Tools

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

This program contains low-level utilities for manipulating "egg" data collected by the Global Consciousness Project. Egg sites around the world periodically report the random event generator results they collect each second to one or more "basket" sites, where the data are archived and eventually assembled into daily "eggsummary" files, in which data for each second in the day for each egg are tabulated in Comma-Separated Value (`.csv`) format.

This program implements a *generic_eggsummary* class which provides an in-memory representation of these files, extensible to handle data sets of arbitrary time span and resolution, with a user-defined data type for the individual samples. Known bad data can be excluded either based on a list of egg numbers and time intervals known to be bad, or based on limit values. Samples for individual eggs can be time-shifted to align them in ways other than simultaneous Universal Time (for example, by civil or local solar time), and an extract of a time interval may be prepared. Data can be exported in `.csv` format which can be subsequently reloaded.

A *generic_eggsummary_cache* class, in connection with a utility *eggdatabases* class manages daily egg summary information at a higher level. One can request data simply by specifying a time within the day, all mapping to file name, loading data into memory as required, and storage allocation occurring automatically. Users can access either "live" data from the egg network or the pseudorandom mirror data simply by changing a single parameter to the cache. The cache will prepare a summary file for an arbitrary time interval, merging and extracting data for individual days. The ability for this summary to block data into time intervals longer than the original one second resolution (to reduce memory requirements and computation time when studying extended intervals) is defined but not presently implemented.

Properties of individual egg sites are defined in a `.csv` format file which is accessed via the *egg_properties_database* class. Given the "egg number" for a site, one may obtain its latitude, longitude, altitude, random event generator type, and other information pertinent to analyses.

A utility *csv_parser* class is used to read the various `.csv` files used by the other classes, and is available for reading other files in that format.

This program uses the **systemtime** class, defined in the `timedate` program, to represent and operate on dates and times; please refer to that program for details of its implementation and use.

⟨ `eggdata_test.c` 1 ⟩ ≡
```
#define REVDATE "17th␣February␣2002"
```
See also section 74.

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 `eggdata.h` that programs which use this library may include.

```
⟨ eggdata.h 3 ⟩ ≡
#ifndef EGGDATA_HEADER_DEFINES
#define EGGDATA_HEADER_DEFINES
#include <math.h>     /* Make sure math.h is available */
#include <iostream>
#include <iomanip>
#include <fstream>
#include <cstdlib>
#include <exception>
#include <algorithm>
#include <stdexcept>
#include <string>
#include <vector>
#include <map>
#define HAVE_FDSTREAM_COMPATIBILITY
#ifdef HAVE_FDSTREAM_COMPATIBILITY
#include "fdstream.hpp"
#endif
  using namespace std;
#include <ctype.h>
#include <assert.h>
#include "timedate.h"
  ⟨ Class definitions 8 ⟩
#endif
```

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

```
⟨ Application include files 4 ⟩ ≡
#include "eggdata.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 ⟩ ≡
  ⟨ Egg data utilities 9 ⟩
```

This code is used in section 2.

6. Egg data utilities.

7. Comma-Separated-Value (CSV) Parsing.

The class *csv_parser* This function implements a somewhat extended flavour of `CSV`. In addition to the standard quoted fields, permitting embedded commas, with embedded quotes represented as `""`, backslash escaped character expressed as three octal digits are also permitted, with a double backslash representing an embedded backslash. This is necessary to permit fields which include end-of-line delimiters which would otherwise truncate the record when it is read.

8. A *csv_parser* operates on a line buffer supplied by its creator. It may be created with a Λ line buffer and a line buffer subsequently supplied with the *new_line* method. Changing the line buffer resets the scan pointer to the start of the line. You may query the scan pointer and reset it with the *scan_pointer* and *set_scan_pointer* methods.

```
⟨ Class definitions 8 ⟩ ≡
  class csv_parser {
  private:
    string line_buffer;
    int charpos;
    bool pending_field;
  public:
    csv_parser(string linebuf = "")
    {
      new_line(linebuf);
    }
    ~csv_parser( )
    { }
    void new_line(string linebuf)
    {
      line_buffer = linebuf;
      set_scan_pointer(0);
    }
    int scan_pointer(void)
    {
      return charpos;
    }
    void set_scan_pointer(int charnum = 0)
    {
      charpos = charnum;
      if (charnum ≡ 0) {
        pending_field = false;
      }
    }
    bool next_field(string &f);
  private:
    static bool isOctal(char c)
    {
      return (((c) ≥ '0') ∧ ((c) ≤ '7'));
    }
    bool chars_left(int n)
    {
      return ((charpos + (n − 1)) < line_buffer.length( ));
    }
  };
```

See also sections 11, 12, 13, 20, 24, 27, 28, 29, 33, 39, 43, 44, 48, 49, 50, 52, 53, 62, and 67.

This code is used in section 3.

9. The *next_field* method returns the parsed next field from the current CSV string to its argument f. The return value is a **bool** which is *true* is a field was found and *false* if the end of record was encountered.

The *pending_field* twiddling is a spot of bother due to our desire to properly handle CSV records with trailing commas. Such records should be parsed as containing a void final field, not the absence of such field. Thus, if we've parsed one or more fields from a CSV record and then find nothing after the last comma, we treat it as void final field, not as an end of record.

```
⟨ Egg data utilities 9 ⟩ ≡
  bool csv_parser::next_field(string &f)
  {
    string fld("");
    bool foundfield = false, quoted = false;
    if (chars_left(1)) {
      foundfield = true;
      while (chars_left(1) ∧ isspace(line_buffer[charpos])) {
        charpos++;
      }
      if (chars_left(1)) {
        if (line_buffer[charpos] ≡ '"') {
          ⟨ Parse quoted CSV field 10 ⟩;
        }
        while (chars_left(1) ∧ line_buffer[charpos] ≠ ',') {
          fld += line_buffer[charpos++];
        }
        if (chars_left(1)) {
          charpos++;
          pending_field = true;
        }
        else {
          pending_field = false;
        }
      }
    }
    if (foundfield ∧ ¬quoted) {
      while ((fld.length() > 0) ∧ isspace(fld[fld.length() − 1])) {
        fld.erase(fld.length() − 1);
      }
    }
    if (foundfield) {
      f = fld;
    }
    else if (pending_field) {
      f = fld;
      foundfield = true;
      pending_field = false;
    }
    return foundfield;
  }
```

See also sections 45, 63, 65, 66, 68, 69, 70, 71, 72, and 73.

This code is used in section 5.

10. Fields in CSV files may be quoted, permitting them to contain arbitrary characters such as commas or binary values. If the first character of a field is a `"`, the following code parses it. Such fields are terminated by a matching quote mark and may contain double quote marks denoting an embedded quote. Binary character codes may be included with the escape sequence "*ddd*" where *ddd* are three octal digits. To include a backslash in a quoted field, write two consecutive backslashes.

```
⟨ Parse quoted CSV field 10 ⟩ ≡
  quoted = true;
  charpos++;
  while (chars_left(1)) {
    if (line_buffer[charpos] ≡ '"') {
      if (chars_left(2) ∧ (line_buffer[charpos + 1] ≡ '"')) {
        fld += '"';
        charpos += 2;
      }
      else {
        charpos++;
        break;
      }
    }
    else if (line_buffer[charpos] ≡ '\\' ∧ chars_left(2)) {
      if (line_buffer[charpos + 1] ≡ '\\') {
        fld += '\\';
        charpos += 2;
      }
      else if (chars_left(4) ∧ isOctal(line_buffer[charpos + 1]) ∧ isOctal(line_buffer[charpos + 2]) ∧
          isOctal(line_buffer[charpos + 3])) {
        char v;
        v = ((line_buffer[charpos + 1] - '0') ≪ 6) | ((line_buffer[charpos + 2] - '0') ≪ 3) |
          (line_buffer[charpos + 3] - '0');
        fld += (char) v;
        charpos += 4;
      }
      else {
        fld += line_buffer[charpos + 1];
        charpos += 2;
      }
    }
    else {
      fld += line_buffer[charpos++];
    }
  }
```

This code is used in section 9.

11. The Egg Summary Class.

The *eggsummary* class is an in-memory representation of a day's data from a collection of random event generator "eggs". Daily archives in CSV format are prepared by the `basketran` program, which reads the raw binary basket data files and assembles them into CSV files with names of `basketdata-`*yyyy*`-`*mm*`-`*dd*.

We define a template class, *generic_eggsummary*, which is instantiated with the data type for the egg samples it contains. By default, this is **unsigned char**, which covers the 0–200 range in original egg data sets, as well as accommodating the special codes we use to represent missing samples, bad data, and the like. A **typedef** is supplied to instantiate *generic_eggsummary* as *eggsummary* with samples of **unsigned char**.

The reason for going to the trouble of defining a template with a parametric sample type is to make all of the general-purpose services of the class (loading databases, time shifting, extraction of time intervals, etc.) available for analyses which require non-integral sample types or those with larger capacity than the raw egg data files.

For example, suppose we wish to aggregate samples into blocks along the time axis: say, storing the mean (or some other metric such as a composite z score) for each minute's worth of samples. If we were forced to keep the samples in the default type of **unsigned char**, we'd lose precision in the aggregated values, introducing unacceptable bias due to round off. In such circumstances, we can simply declare the data table as "*generic_eggsummary* < **double** >" and *presto*...we have all the functionality of the class plus double precision floating point resolution for the samples.

```
⟨ Class definitions 8 ⟩ +≡
  template⟨class Sample⟩ class generic_eggsummary {
  public:      /* The following value identifies missing samples in the egg data table. We assume, therefore,
          that trial_size below is always less than MissingSample. */
    static const unsigned char MissingSample = 255;      /* The following value identifies samples
          marked as bad in the database loaded with exclude_bad_data. Again, trial_size must be less than
          this value. */
    static const unsigned char BadSample = 254;
      /* This value flags samples which failed the limit test applied by limit_to_range. */
    static const unsigned char OutsideLimitsSample = 253;      /* The following value replaces samples
          shifted outside the time spanned by the data set by time_shift. */
    static const unsigned char TimeShiftedOutSample = 252;
      /* The following gives the size in bytes of the Sample type we're instantiated with. */
    static const unsigned int sampleBytes = sizeof(Sample);      /* The following variables specify
          the protocol used by the eggs in collecting the data. These are required to be uniform across all
          eggs contributing to a day summary file. These are rarely relevant to analysis of the summary file,
          but do appear in it and are loaded with the file. */
    int samples_per_record;      /* Samples per data record */
    int seconds_per_record;      /* Seconds per data record */
    int records_per_packet;      /* Records per network data packet */      /* Each individual report from
          an egg consists of the number of one bits in a series of "trials" emitted by the random event
          generator. Like the network protocol the trial size must be uniform for all eggs in a day's data. The
          mean value of results is expected to be trial_size/2 and the standard deviation √(trial_size/4). */
    int trial_size;      /* Trials per report */
      /* The basketdata files contain all reports submitted by eggs in a given day. A column for an egg
          will be included in the report even if it submits the results of only one trial for that day. The
          following field indicates how many egg columns are present in the report. */
    int eggs_reporting;      /* Egg columns in this report */
      /* The start_time and end_time are systemtime objects which give the start and ending time of
          data contained in this report. For reports generated by the normal daily summary process, these
          are midnight on the day of the report and 23:59:59 on that day, respectively. */
    systemtime start_time;      /* Report start time */
```

```
  systemtime end_time;      /* Report end time */
    /* The seconds_of_data field indicates how many seconds of data this report contains, which should
       be equal to (end_time − start_time) + 1. */
  unsigned int seconds_of_data;      /* Seconds of data in report */      /* The seconds_per_row field
       indicates how many seconds separate successive rows in the egg data table. */
  unsigned int seconds_per_row;      /* Seconds of data per record */
    /* If the data source is pseudorandom, this variable so indicates */
  bool pseudorandom_data;      /* True if data source pseudorandom */
    /* Each egg is identified by an unsigned integer "egg number", with the hundreds digit denoting the
       type of random event generator used by the egg. The egg_number vector indicates which egg is
       responsible for data in a given column of the data table. */
  vector⟨unsigned int⟩ egg_number;      /* Egg number table */
public:      /* We'll frequently want to locate the column containing data for a given egg number, so to
       avoid repetitively searching the egg_number array, we create a map from the egg numbers to the
       column containing the data for the eggs. */
  map⟨unsigned int, unsigned int⟩ egg_index;
    /* Map egg number to index */      /* The egg_data array contains the individual samples from the
       eggs. Missing samples are denoted by the value MissingSample. To address the cell for egg n in a
       given row, use the subscript expression n + (row × eggs_reporting). */
  Sample *egg_data;      /* Address a cell in the egg data table */
  Sample *egg_cell(int row, int egg_index)
  {
    return egg_data + (egg_index + (row * eggs_reporting));
  }      /* Allocate an egg data table */
  void allocate_egg_data(void)
  {
    assert(egg_data ≡ Λ);
    egg_data = new Sample[eggs_reporting * (seconds_of_data/seconds_per_row)];
  }      /* Release the egg data table if allocated */
  void free_egg_data(void)
  {
    if (egg_data ≠ Λ) {
      delete egg_data;
    }
    egg_data = Λ;
  }
public:      /* Our constructor and destructor take care of releasing the egg_data table when required. */
  generic_eggsummary( )
  {
    egg_data = Λ;
  }
  ~generic_eggsummary( )
  {
    free_egg_data( );
  }      /* Retrieve egg data by row and egg index */
  Sample get_egg_index(int row, int egg_index)
  {
    return *egg_cell(row, egg_index);
  }      /* Set egg data by row and egg index */
  void set_egg_index(int row, int egg_index, Sample value)
```

```
    {
#if 0
      cout ≪ "Set␣egg␣index(" ≪ row ≪ ",␣" ≪ egg_index ≪ ")␣=␣" ≪ ((int) value) ≪ "\n";
#endif
      *egg_cell(row, egg_index) = value;
    }    /* Obtain egg index from egg number */
    int egg_number_to_index(int egg_number)
    {
      map⟨unsigned int, unsigned int⟩::iterator mpair;
      mpair = egg_index.find(egg_number);
      if (mpair ≡ egg_index.end()) {
        throw (invalid_argument("egg_number_to_index:␣Invalid␣egg␣number"));
      }
      return mpair→second;
    }    /* Retrieve egg data by row and egg number */
    Sample get_egg_number(int row, int egg_number)
    {
      return get_egg_index(row, egg_number_to_index(egg_number));
    }    /* Test whether a sample is valid based on its value */
    bool isSampleValid(Sample sample)
    {
      return sample ≤ trial_size;
    }    /* Test whether a sample for a given row and egg number is valid */
    bool isSampleValid(int row, int egg_number)
    {
      return isSampleValid(get_egg_number(row, egg_number));
    }    /* Load egg data from input stream of CSV file */
    void load_from_CSV(istream &i);
    void load_from_CSV(string filename);    /* Save data to CSV file */
    void save_to_CSV(ostream &o, bool interpret_missing = false);
    void save_to_CSV(string fileName, bool interpret_missing = false);
      /* Exclude known bad samples specified by CSV file */
    void exclude_bad_data(istream &i);
    void exclude_bad_data(string filename);
      /* Test samples within limits and mark bad those outside */
    void limit_to_range(Sample minLimit = 0, Sample maxLimit = 255, ostream *os = 0);
      /* Time-shift samples from a given egg by a specified number of seconds */
    void time_shift(int egg_index, int seconds);
      /* Extract data for a specified time range into another eggsummary */
    void extract_time_range(generic_eggsummary⟨Sample⟩ *es, systemtime begin, systemtime
        end, int sec_row = 1);    /* Extract data for a subset of eggs in the data set */
    void extract_eggs(generic_eggsummary⟨Sample⟩ *es, vector⟨unsigned int⟩ &eggs);
      /* Write description to output stream */
    void describe(ostream &os = cout);
  };
  typedef generic_eggsummary⟨unsigned char⟩ eggsummary;
```

12. Loading data from CSV files.

Egg summary files are complied on a daily basis as CSV (comma-separated-value) files. The *load_from_CSV* method reads one of these files from an input stream (in a strictly serial manner) into an in-memory **eggsummary** object.

```
⟨ Class definitions 8 ⟩ +≡
  template⟨class Sample⟩ void generic_eggsummary⟨Sample⟩::load_from_CSV(istream &i)
  {
    string s;
    int egg_row = 0;
    free_egg_data( );      /* Release egg data if already allocated */
    while (getline(i, s)) {
      csv_parser p(s);
      string f;
      int i, m_type, s_type, arg;      /* Direct CSV record to processing module by major type */
      if (¬p.next_field(f) ∨ f.length( ) < 1 ∨ ¬isdigit(f[0])) {
        throw (out_of_range("load_from_CSV: Invalid major type"));
      }
      m_type = atoi(f.c_str( ));
      switch (m_type) {
      case 10:
        ⟨ Process type 10 CSV record 16 ⟩;
        break;
      case 11:
        ⟨ Process type 11 CSV record 17 ⟩;
        break;
      case 12:
        ⟨ Process type 12 CSV record 18 ⟩;
        break;
      case 13:
        ⟨ Process type 13 CSV record 19 ⟩;
        break;
      }
    }
#ifdef CSV_LOAD_DEBUG
    cout ≪ "CSV load complete.\n";
    cout.flush( );
#endif
  }
```

13. We provide a variant of the *load_from_CSV* method which accepts a file name argument rather than an existing **istream**. This version identifies compressed files and sets up a pipe through which the uncompressed data are retrieved.

```
⟨ Class definitions 8 ⟩ +≡
  template⟨class Sample⟩ void generic_eggsummary⟨Sample⟩::load_from_CSV(string filename)
  {
#ifdef HAVE_FDSTREAM_COMPATIBILITY
    fdistream iscc;
#endif
    ifstream is;
    FILE *ip = Λ;
    ⟨ Configure compression suffix and command 14 ⟩;
#ifdef COMPRESSED_FILES
    if (filename.rfind(Compressed_file_type) ≠ string::npos) {
      string cmd(Uncompress_command);
      cmd += ' ' + filename;
      ip = popen(cmd.c_str(), "r");
#ifdef HAVE_FDSTREAM_COMPATIBILITY
      iscc.attach(fileno(ip));
#else
      is.attach(fileno(ip));
#endif
    }
    else {
#endif
      is.open(filename.c_str(), ios::in);
#ifdef COMPRESSED_FILES
    }
#endif
#ifdef HAVE_FDSTREAM_COMPATIBILITY
    if (ip ≡ Λ) {
      load_from_CSV(is);
    }
    else {
      load_from_CSV(iscc);
    }
#else
    load_from_CSV(is);
#endif
    if (ip ≠ Λ) {
      pclose(ip);
    }
  }
```

14. The type of compression and command required to expand compressed files may differ from system to system. The following code, conditional based on variables determined by the `autoconf` process, defines the file suffix denoting a compressed file and the corresponding command used to decode it. We only support one type of compression on a given system; if `gzip` is available, we use it in preference to `compress`.

```
⟨ Configure compression suffix and command 14 ⟩ ≡
#if (defined HAVE_GUNZIP) ∨ (defined HAVE_GZCAT) ∨ (defined HAVE_GZIP)
  # define COMPRESSED_FILES
      static const char Compressed_file_type[ ] = ".gz";
      static const char Uncompress_command[ ] =
      # if (defined HAVE_GUNZIP)
      "gunzip␣-c"
      # elif (defined HAVE_GZCAT)
      "gzcat"
      # elif (defined HAVE_GZIP)
      "gzip␣-cd"
      # endif
      ;
#elif (defined HAVE_ZCAT) ∨ (defined HAVE_UNCOMPRESS) ∨ (defined HAVE_COMPRESS)
      # define COMPRESSED_FILES
          static const char Compressed_file_type[ ] = ".Z";
          static const char Uncompress_command[ ] =
          # if (defined HAVE_ZCAT)
          "zcat"
          # elif (defined HAVE_HAVE_UNCOMPRESS)
          "uncompress␣-c"
          # elif (defined HAVE_COMPRESS)
          "compress␣-cd"
          # endif
          ;
#endif
```

This code is used in sections 13 and 27.

15. CSV header records types 10 and 11 have a sub-code as the second field which identifies the individual datum. The following code parses the subcode.

```
⟨ Parse CSV header subcode 15 ⟩ ≡
  if (¬p.next_field(f) ∨ f.length( ) < 1 ∨ ¬isdigit(f[0])) {
    throw (out_of_range("load_from_CSV:␣Invalid␣subtype"));
  }
  s_type = atoi(f.c_str( ));
  if (¬p.next_field(f) ∨ f.length( ) < 1 ∨ ¬isdigit(f[0])) {
    throw (out_of_range("load_from_CSV:␣Invalid␣argument"));
  }
  arg = atoi(f.c_str( ));
```

This code is used in sections 16 and 17.

16. Type 10 CSV records are main header information. We read them into the protocol definition and experiment parameters fields of the **eggsummary** object.

```
⟨ Process type 10 CSV record 16 ⟩ ≡
  ⟨ Parse CSV header subcode 15 ⟩;
  switch (s_type) {
  case 1:
    samples_per_record = arg;
    break;
  case 2:
    seconds_per_record = arg;
    break;
  case 3:
    records_per_packet = arg;
    break;
  case 4:
    trial_size = arg;
    assert(trial_size < MissingSample);
    break;
  }
```

This code is used in section 12.

17. Type 11 CSV records provide details of the data to follow: the start and ending times, and the number of eggs reporting data for this interval.

```
⟨ Process type 11 CSV record 17 ⟩ ≡
  ⟨ Parse CSV header subcode 15 ⟩;
  switch (s_type) {
  case 1:
    eggs_reporting = arg;
    if (¬p.next_field(f) ∨ (f.length( ) < 1)) {
      throw (out_of_range("load_from_CSV:␣Invalid␣Eggs␣reporting␣header"));
    }
    pseudorandom_data = f.find("pseudorandom") ≠ string::npos;
    break;
  case 2:
    start_time.set_time(arg);
    break;
  case 3:
    end_time.set_time(arg);
    break;
  case 4:
    seconds_of_data = arg;
    break;
  }
```

This code is used in section 12.

18. The single type 12 CSV record supplies the list of egg numbers which correspond to the columns in the type 13 records supplying the data table which follows.

```
⟨ Process type 12 CSV record 18 ⟩ ≡        /* The first two fields of the type 12 record are place-holders */
  for (i = 0; i < 2; i++) {
    if (¬p.next_field(f)) {
      throw (out_of_range("load_from_CSV: Invalid type 12 record"));
    }
  }     /* The subsequent fields give the egg numbers for the columns in the data table. */
  for (i = 0; i < eggs_reporting; i++) {
    unsigned int eggno;
    if (¬p.next_field(f) ∨ f.length( ) < 1 ∨ ¬isdigit(f[0])) {
      throw (out_of_range("load_from_CSV: Invalid type 12 egg table"));
    }
    eggno = atoi(f.c_str( ));       /* Add egg number to egg_number vector */
    egg_number.push_back(eggno);       /* Add egg number to egg_index map */
    egg_index.insert(make_pair(eggno, i));
  }
```

This code is used in section 12.

19. Type 13 records contain the actual egg data samples. The second field is the date and time of the record, followed by a void place-holder field and the actual egg sample data. Note that missing fields are void, not equal to the *MissingSample* convention we use here.

```
⟨ Process type 13 CSV record 19 ⟩ ≡
  time_t rtime, xrtime;      /* Allocate egg_data table before processing first type 13 record */
  if (egg_data ≡ Λ) {
    seconds_per_row = seconds_per_record / samples_per_record;
    egg_data = new Sample[eggs_reporting * (seconds_of_data / seconds_per_row)];
      /* Start time is expected time for first record */
    xrtime = start_time.get_time( );
    egg_row = 0;
  }
  if (¬p.next_field(f) ∨ f.length( ) < 1 ∨ ¬isdigit(f[0])) {
    throw (out_of_range("load_from_CSV:␣Invalid␣time␣in␣type␣13␣record"));
  }
  rtime = atoi(f.c_str( ));
  assert(rtime ≡ xrtime);      /* Update expected time for next record */
  xrtime += seconds_per_row;
  if (¬p.next_field(f)) {
    throw (out_of_range("load_from_CSV:␣Invalid␣place␣holder␣in␣type␣13␣record"));
  }      /* Now read the egg sample column data and store into the egg data table. Note that we must
            replace void fields with our MissingSample indicator. */
#ifdef CLEAN_BUT_SLOW
  for (i = 0; i < eggs_reporting; i++) {
    unsigned char eggdat = MissingSample;
    if (¬p.next_field(f)) {
      throw (out_of_range("load_from_CSV:␣Missing␣egg␣data␣in␣type␣13␣record"));
    }
    if (f.length( ) > 0) {
      if (¬isdigit(f[0])) {
        throw (out_of_range("load_from_CSV:␣Invalid␣egg␣data␣in␣type␣13␣record"));
      }
      eggdat = atoi(f.c_str( ));
    }
    set_egg_index(egg_row, i, eggdat);
  }
#else      /* Hey, I agree, this ain't pretty, but it's five times faster (when compiled with optimisation)
              than the squeaky clean CLEAN_BUT_SLOW implementation above. Cleanliness is fine, but you also
              need to get the answer out before the protons decay in your CPU chip, so we descend into low-level C
              string bashing to get the job done in the following inner loop. Note that this code assumes the CSV
              file is perfectly formatted and that the eggdat array is laid out with columns occupying consecutive
              memory addresses. */
  const char *cp = s.c_str( );
  Sample *ep = egg_cell(egg_row, 0);      /* Skip first three fields already scanned the slow way */
  cp = strchr(strchr(strchr(cp, ',') + 1, ',') + 1, ',') + 1;
  for (i = 0; i < eggs_reporting; i++) {
    unsigned char eggdat;
    if (¬isdigit(*cp)) {
      eggdat = MissingSample;
    }
    else {
```

```
      eggdat = (*cp++) - '0';
      while (isdigit(*cp)) {
        eggdat = (eggdat * 10) + ((*cp++) - '0');
      }
    }
    *ep++ = eggdat;
    cp++;
  }
#endif
  egg_row++;
```

This code is used in section 12.

20. Saving data to CSV files.

```
⟨ Class definitions 8 ⟩ +≡
  template⟨class Sample⟩ void generic_eggsummary⟨Sample⟩::save_to_CSV(ostream &o, bool
      interpret_missing)
  {
    int i;
    ⟨ Write CSV file header 21 ⟩;
    ⟨ Write CSV egg number table 22 ⟩;
    ⟨ Write CSV egg data table 23 ⟩;
  }
  template⟨class Sample⟩ void generic_eggsummary⟨Sample⟩::save_to_CSV(string fileName, bool
      interpret_missing)
  {
    ofstream of(fileName.c_str());
    save_to_CSV(of, interpret_missing);
  }
```

21. First we write the CSV file header, containing the global parameters for the data set.

```
⟨ Write CSV file header 21 ⟩ ≡
  o << "10,1," << samples_per_record << ",\"Samples␣per␣record\"" << endl;
  o << "10,2," << seconds_per_record << ",\"Seconds␣per␣record\"" << endl;
  o << "10,3," << records_per_packet << ",\"Records␣per␣packet\"" << endl;
  o << "10,4," << trial_size << ",\"Trial␣size\"" << endl;
  o << "11,1," << eggs_reporting << ",\"Eggs␣reporting\"" << endl;
  o << "11,2," << start_time.get_time() << ",\"Start␣time\"," << start_time.dateToString() << "␣" <<
      start_time.timeToString() << endl;
  o << "11,3," << end_time.get_time() << ",\"End␣time\"," << end_time.dateToString() << "␣" <<
      end_time.timeToString() << endl;
  o << "11,4," << seconds_of_data << ",\"Seconds␣of␣data\"" << endl;
```

This code is used in section 20.

22. Next, we emit the record giving the egg numbers for the columns in the data set which follows.

```
⟨ Write CSV egg number table 22 ⟩ ≡
  o ≪ "12,\"gmtime\",";
  for (i = 0; i < eggs_reporting; i++) {
    o ≪ "," ≪ egg_number[i];
  }
  o ≪ endl;
```

This code is used in section 20.

23. Finally, we write out the egg data for each time interval in the data set. If the *interpret_missing* argument is *true*, we include the codes which identify the reason for the absence of specific samples. Otherwise, they are simply output as empty fields.

```
⟨ Write CSV egg data table 23 ⟩ ≡
  int nrows = seconds_of_data/seconds_per_row;
  systemtime now = start_time;
  for (int n = 0; n < nrows; n++) {
    o ≪ "13," ≪ now.get_time( ) ≪ ",";
    for (int e = 0; e < eggs_reporting; e++) {
      Sample s = get_egg_index(n, e);
      o ≪ ",";
      if ((s ≤ trial_size) ∨ interpret_missing) {
        o ≪ ((double) get_egg_index(n, e));
      }
    }
    o ≪ endl;
    now.set_time(now.get_time( ) + seconds_per_row);
  }
```

This code is used in section 20.

24. Excluding data known to be bad.

To maintain the integrity of the primary Egg Summary database, we do not modify it, even to exclude known bad samples (due, in most cases, to a defective random event generator or misconfigured computer). Instead, we note the time range and egg number of data known to be bad in a separate CSV database. After loading the original data, bad samples may be excluded by passing the bad sample database to the *exclude_bad_data* method. Bad data are replaced with *BadSample*.

```
⟨ Class definitions 8 ⟩ +≡
  template⟨class Sample⟩ void generic_eggsummary⟨Sample⟩::exclude_bad_data(istream &i)
  {
    string s;
    systemtime startTime, endTime;
    int rottenEgg;
    while (getline(i, s)) {
      if ((s.length() > 0) ∧ (s[0] ≠ ';') ∧ (s[0] ≠ '#')) {
        csv_parser p(s);
        string f;
        int i, m_type, s_type, arg;
        if (¬p.next_field(f) ∨ f.length() < 1 ∨ ¬isdigit(f[0])) {
          throw (out_of_range("exclude_bad_data: Invalid record type"));
        }
        m_type = atoi(f.c_str());
        switch (m_type) {
        case 47: ⟨ Process record from bad sample database 25 ⟩;
          break;
        default: throw (out_of_range("exclude_bad_data: Invalid record type"));
          break;
        }
      }
    }
#ifdef EXCLUDE_DEBUG
    cout ≪ "Exclusion of bad data complete.\n";
    cout.flush();
#endif
  }
```

25. Records in the bad sample database have the following format:

`47,`*Start time*`,`*End time*`,`*Egg number*

where `47` is the record type, *Start time* and *End time* specify the start and end of the period during which the egg produced bad samples, and *Egg number* is the number of the rotten egg. The start and end time may be specified either as ISO 8601 date and time, for example, "`2001-08-19 03:35:18`" or as the equivalent UNIX date and time value, `998192118`.

Comments may be included in the bad sample database as lines which begin with "`#`" or "`;`".

```
⟨ Process record from bad sample database 25 ⟩ ≡
  if (¬p.next_field(f) ∨ f.length() < 1 ∨ ¬isdigit(f[0])) {
    throw (out_of_range("exclude_bad_data: Invalid start time in type 47 record"));
  }
  if (f.find('-') ≠ string::npos) {
    startTime.fromString(f);
  }
  else {
    startTime.set_time(atoi(f.c_str()));
  }
  if (¬p.next_field(f) ∨ f.length() < 1 ∨ ¬isdigit(f[0])) {
    throw (out_of_range("exclude_bad_data: Invalid end time in type 47 record"));
  }
  if (f.find('-') ≠ string::npos) {
    endTime.fromString(f);
  }
  else {
    endTime.set_time(atoi(f.c_str()));
  }
  if (endTime.get_time() < startTime.get_time()) {
    throw (out_of_range("exclude_bad_data: End time before start time in type 47 record"));
  }
  if (¬p.next_field(f) ∨ f.length() < 1 ∨ ¬isdigit(f[0])) {
    throw (out_of_range("exclude_bad_data: Invalid egg number in type 47 record"));
  }
  rottenEgg = atoi(f.c_str());
  assert(rottenEgg ≥ 1);
#ifdef EXCLUDE_DEBUG
  cout ≪ rottenEgg ≪ ": " ≪ startTime.dateToString() ≪ " " ≪ startTime.timeToString() ≪
      " --> " ≪ startTime.dateToString() ≪ " " ≪ endTime.timeToString() ≪ endl;
#endif
  ⟨ Exclude samples marked as bad 26 ⟩;
```

This code is cited in section 51.

This code is used in section 24.

26. For each record in the bad sample database, we first perform a quick reject to determine if the time range overlaps that represented in this **eggsummary**. If so, we walk through the table, row by row, and zap the sample for the specified bad egg for all samples within the bad data range. Samples are replaced with the special *BadSample* value (even if they were marked as *MissingSample*).

```
⟨ Exclude samples marked as bad 26 ⟩ ≡
  if (¬((endTime.get_time() < start_time.get_time()) ∨ (startTime.get_time() > end_time.get_time()))) {
    map⟨unsigned int, unsigned int⟩::iterator mpair;
    mpair = egg_index.find(rottenEgg);
    if (mpair ≠ egg_index.end()) {
      int egg_column = mpair→second;
      systemtime rowTime(start_time.get_time());
      time_t endt = end_time.get_time();
      if (endTime.get_time() < endt) {
        endt = endTime.get_time();
      }
      int row = 0;
      while (rowTime.get_time() ≤ endt) {
        if ((rowTime.get_time() ≥ startTime.get_time()) ∧ (rowTime.get_time() ≤ endTime.get_time()))
          {
          set_egg_index(row, egg_column, BadSample);
#ifdef EXCLUDE_DEBUG
          cout ≪ "Zapped␣egg␣" ≪ rottenEgg ≪ "␣at␣" ≪ rowTime.dateToString() ≪ "␣" ≪
              rowTime.timeToString() ≪ "␣index␣" ≪ row ≪ endl;
#endif
        }
        rowTime.set_time(rowTime.get_time() + seconds_per_row);
        row++;
      }
    }
#ifdef EXCLUDE_DEBUG
    else {
      cout ≪ "No␣such␣egg." ≪ endl;
    }
#endif
  }
#ifdef EXCLUDE_DEBUG
  else {
    cout ≪ "Quick␣reject." ≪ endl;
  }
#endif
```

This code is used in section 25.

27. We provide a variant of the *exclude_bad_data* method which accepts a file name argument rather than an existing **istream**. This version identifies compressed files and sets up a pipe through which the uncompressed data are retrieved.

```
⟨ Class definitions 8 ⟩ +≡
  template⟨class Sample⟩ void generic_eggsummary⟨Sample⟩::exclude_bad_data(string filename)
  {
    ifstream is;
#ifdef HAVE_FDSTREAM_COMPATIBILITY
    fdistream iscc;
#endif
    FILE *ip = Λ;
    ⟨ Configure compression suffix and command 14 ⟩;
#ifdef COMPRESSED_FILES
    if (filename.rfind(Compressed_file_type) ≠ string::npos) {
      string cmd(Uncompress_command);
      cmd += ' ' + filename;
      ip = popen(cmd.c_str(), "r");
#ifdef HAVE_FDSTREAM_COMPATIBILITY
      iscc.attach(fileno(ip));
#else
      is.attach(fileno(ip));
#endif
    }
    else {
#endif
      is.open(filename.c_str(), ios::in);
#ifdef COMPRESSED_FILES
    }
#endif
#ifdef HAVE_FDSTREAM_COMPATIBILITY
    if (ip ≡ Λ) {
      exclude_bad_data(is);
    }
    else {
      exclude_bad_data(iscc);
    }
#else
    exclude_bad_data(is);
#endif
    if (ip ≠ Λ) {
      pclose(ip);
    }
  }
```

28. Excluding samples which exceed "sanity check" limits.

Random event generators occasionally get "the vapours" or the computers they're connected to flip out to la-la land, resulting in absurd samples. These are normally excluded by the bad sample database applied by the *exclude_bad_data* method above, but absent specific exclusion thereby, it's often convenient to prune implausible outliers based solely on their value. The *limit_to_range* method allows you to replace all samples in an **eggsummary** not within the range $minLimit \le sample \le maxLimit$ with the value *OutsideLimitsSample* which causes it to be ignored in analysis. If *os* is non-Λ, a summary of all samples deleted as outside limits will be written to that output stream.

```
⟨ Class definitions 8 ⟩ +≡
  template⟨class Sample⟩ void generic_eggsummary⟨Sample⟩::limit_to_range(Sample
          minLimit, Sample maxLimit, ostream *os)
  {
    for (int i = 0; i < eggs_reporting; i++) {
      int row = 0;
      for (time_t t = start_time.get_time(); t ≤ end_time.get_time(); t += seconds_per_row) {
        if (isSampleValid(get_egg_index(row, i))) {
          Sample was = get_egg_index(row, i);
          if ((was < minLimit) ∨ (was > maxLimit)) {
            set_egg_index(row, i, OutsideLimitsSample);
            if (os ≠ 0) {
              *os ≪ "limit_to_range:␣zapping␣egg␣" ≪ egg_number[i] ≪ "␣at␣" ≪
                  systemtime(t).dateToString() ≪ "␣" ≪ systemtime(t).timeToString() ≪
                  "␣was␣" ≪ ((unsigned int) was) ≪ endl;
            }
          }
        }
        row++;
      }
    }
  }
```

29. Time shifting samples from individual eggs.

Data from individual eggs are arranged in the **eggsummary** so that each row contains samples made at the same moment (assuming the egg hosts' clocks are properly synchronised, of course). For some studies, we're interested not in instantaneous behaviour of the network but, for example, effects related to local solar or perhaps sidereal time at individual egg sites.

The *time_shift* method facilitates such explorations. It shifts the data for the specified *egg_index* (note: index, *not* egg number—you can obtain the index for an egg number with *egg_number_to_index*) by the specified number of *seconds*, which can be positive or negative. Data shifted out of the time spanned by the **eggsummary** are replaced by the invalid sample value *TimeShiftedOutSample*.

Suppose, for example, you wish to study the behaviour of eggs based on local solar time at the egg location over a period of one day. Begin by obtaining a **eggsummary** for three days centred on the day in question. Then, for each egg, use *time_shift* to shift the samples by a number of seconds corresponding to the egg site's latitude (which you can obtain from the *egg_properties_database*). Eggs to the East of the Greenwich meridian would have their data shifted by a negative number of seconds, eggs to the West, by a positive number. Starting with three days' data guarantees that after the shifting is complete, the middle day will contain no shifted out samples. We can then extract the aligned day from the middle of the table and perform the desired analyses.

The code is written out in a series of hand-optimised cases in the interest of efficiency; if you need this function, your code is probably going to spend a lot of time in here and it's worth a little additional complexity to speed things up.

```
⟨ Class definitions 8 ⟩ +≡
  template⟨class Sample⟩ void generic_eggsummary⟨Sample⟩::time_shift(int egg_index, int
          seconds)
  {
    assert((egg_index ≥ 0) ∧ (egg_index < eggs_reporting));      /* Egg index out of bounds ? */
    int rows_to_shift = seconds/seconds_per_row,      /* Number of rows to shift data */
    nrows = seconds_of_data/seconds_per_row;      /* Number of rows in table */
    if (rows_to_shift ≠ 0) {
      if (abs(seconds) ≥ seconds_of_data) {
        ⟨ Time shift data entirely out of the table 32 ⟩;
      }
      else {
        if (rows_to_shift < 0) {
          ⟨ Time shift data to earlier slots in table 30 ⟩;
        }
        else {
          ⟨ Time shift data to later slots in table 31 ⟩;
        }
      }
    }
  }
```

30. When the *seconds* argument is negative, we shift samples for the designated egg to earlier time slots in the **eggsummary** table, shifting in *TimeShiftedOutSample* into the slots at the end.

```
⟨ Time shift data to earlier slots in table 30 ⟩ ≡
  rows_to_shift = −rows_to_shift;
  int i, ncopy = nrows − rows_to_shift;      /* Rows to copy */
  Sample *dstp = egg_cell(0, egg_index), *srcp = egg_cell(rows_to_shift, egg_index);
  for (i = 0; i < ncopy; i++) {
    *dstp = *srcp;
    srcp += eggs_reporting;
    dstp += eggs_reporting;
  }     /* Fill data shifted out at end of table */
  dstp = egg_cell(ncopy, egg_index);
  for (i = 0; i < rows_to_shift; i++) {
    *dstp = TimeShiftedOutSample;
    dstp += eggs_reporting;
  }
```

This code is used in section 29.

31. Conversely, when *seconds* is positive, we move samples for the egg to later time slots in the table, filling vacated cells at the beginning of the table with *TimeShiftedOutSample*.

```
⟨ Time shift data to later slots in table 31 ⟩ ≡
  int i, ncopy = nrows − rows_to_shift;      /* Rows to copy */
  Sample *srcp = egg_cell((nrows − 1) − rows_to_shift, egg_index), *dstp = egg_cell((nrows − 1), egg_index);
  for (i = 0; i < ncopy; i++) {
    *dstp = *srcp;
    srcp −= eggs_reporting;
    dstp −= eggs_reporting;
  }     /* Fill data shifted out at start of table */
  dstp = egg_cell(0, egg_index);
  for (i = 0; i < rows_to_shift; i++) {
    *dstp = TimeShiftedOutSample;
    dstp += eggs_reporting;
  }
```

This code is used in section 29.

32. If the time shift specified is longer than the time interval present in the table, data for the egg are entirely wiped out and replaced by *TimeShiftedOutSample*. This is a pretty dopey thing to do, but one can imagine circumstances in which it makes sense.

```
⟨ Time shift data entirely out of the table 32 ⟩ ≡
  Sample *cellp = egg_cell(0, egg_index);
  for (int i = 0; i < nrows; i++) {
    *cellp = TimeShiftedOutSample;
    cellp += eggs_reporting;
  }
```

This code is used in section 29.

33. Extraction of data for a time interval within the data set.

When studying predictions with a predefined time span, you'll want to analyse a subset of the data covering the prediction. The *extract_time_range* method extracts the data for a given time period (which must be entirely present within the source **eggsummary**) and writes it into a destination **eggsummary** *es*. We do the somewhat eccentric pointer game so that this method can be used with classes derived from **eggsummary**, for example, those which add analytical facilities. Users of those classes can use *extract_time_range* to prepare the raw data table, then employ the methods of the derived class to perform the analyses.

```
⟨ Class definitions 8 ⟩ +≡
  template⟨class Sample⟩ void
          generic_eggsummary⟨Sample⟩::extract_time_range(generic_eggsummary⟨Sample⟩
          *es, systemtime begin, systemtime end, int sec_row)
  {
    assert(sec_row ≡ 1);      /* Aggregation of data into longer intervals isn't implemented yet! */
    assert(sec_row ≥ seconds_per_row);      /* Can't increase time resolution by extracting! */
    assert((begin.get_time( ) ≥ start_time.get_time( )) ∧ (end.get_time( ) ≤ end_time.get_time( )));
      /* Requested range outside that in data set */
    ⟨ Copy header fields into extracted data set 34 ⟩;
    ⟨ Set header fields which differ in extracted data set 35 ⟩;
    ⟨ Allocate and copy data table for extracted data set 36 ⟩;
  }
```

34. It's tedious, but there's nothing for it—we have to copy all of the header fields the extract inherits from the source **eggsummary**. There's never a `MOVE CORRESPONDING` when you need one.

```
⟨ Copy header fields into extracted data set 34 ⟩ ≡
#define Cf(x) es→x = x
  Cf(samples_per_record);
  Cf(seconds_per_record);
  Cf(records_per_packet);
  Cf(trial_size);
  Cf(eggs_reporting);
  Cf(pseudorandom_data);
  Cf(egg_number);
  Cf(egg_index);
#undef Cf
```

This code is used in section 33.

35. Naturally, some header fields differ in the extracted data set. Here we initialise them.

```
⟨ Set header fields which differ in extracted data set 35 ⟩ ≡
  es→start_time.set_time(begin.get_time( ));
  es→end_time.set_time(end.get_time( ));
  es→seconds_of_data = end.get_time( ) − begin.get_time( );
  es→seconds_per_row = sec_row;
```

This code is used in section 33.

36. Finally, we're ready to actually extract the data. Allocate the data table (releasing any previously allocated table) and copy the extract into it.

```
⟨ Allocate and copy data table for extracted data set 36 ⟩ ≡
  es→free_egg_data( );      /* Release existing egg data table, if any */
  es→egg_data = new Sample[eggs_reporting * (seconds_of_data/seconds_per_row)];
  Sample *dstp = es→egg_cell(0,
      0), *srcp = egg_cell(((es→start_time.get_time( ) − start_time.get_time( ))/seconds_per_row), 0);
  if (seconds_per_row ≡ es→seconds_per_row) {
    ⟨ Copy data table for extracted data set with identical time resolution 37 ⟩;
  }
  else {
    ⟨ Summarise data table for extracted data set with reduced time resolution 38 ⟩;
  }
```

This code is used in section 33.

37. When the time resolution of the extracted data set is identical to that of the source, we can simply blast-*memcpy* the bytes of the data table from the source to destination, offsetting the source pointer to begin at the row containing the start time of the requested extract.

```
⟨ Copy data table for extracted data set with identical time resolution 37 ⟩ ≡
  memcpy(dstp, srcp, sampleBytes * eggs_reporting * (es→seconds_of_data/es→seconds_per_row));
```

This code is used in section 36.

38.

```
⟨ Summarise data table for extracted data set with reduced time resolution 38 ⟩ ≡
  assert(false);      /* Not implemented yet */
```

This code is used in section 36.

39. Extraction of data for a subset of eggs.

```
⟨ Class definitions 8 ⟩ +≡
  template⟨class Sample⟩ void
          generic_eggsummary⟨Sample⟩::extract_eggs(generic_eggsummary⟨Sample⟩
          *es, vector⟨unsigned int⟩ &eggs)
  {
    ⟨ Copy header fields into extracted list of eggs data set 40 ⟩;
    ⟨ Create egg number table and map for extracted eggs 41 ⟩;
    ⟨ Copy data for extracted eggs 42 ⟩;
  }
```

40. Copy the header fields which are identical in the original data set and the extract.

```
⟨ Copy header fields into extracted list of eggs data set 40 ⟩ ≡
#define Cf(x) es→x = x
  Cf(samples_per_record);
  Cf(seconds_per_record);
  Cf(records_per_packet);
  Cf(trial_size);
  Cf(start_time);
  Cf(end_time);
  Cf(seconds_of_data);
  Cf(seconds_per_row);
  Cf(pseudorandom_data);
#undef Cf
```

This code is used in section 39.

41. To build the egg number table for the extracted data set, we iterate through the *eggs* argument vector and add each egg which appears in the source **eggsummary** to the extract's *egg_number* vector. We permit the specification of egg numbers in the argument which aren't present in the source data, as the user might want to request a subset of eggs for a series of days in which some of them may not figure. If the egg does not appear in the source data, it will not appear in the extract. We sort the egg number table before building the map from it to preserve the convention that eggs appear in ascending order by number.

```
⟨ Create egg number table and map for extracted eggs 41 ⟩ ≡
  es→egg_index.clear( );
  es→egg_number.clear( );
  vector⟨unsigned int⟩::iterator s;
  for (s = eggs.begin( ); s ≠ eggs.end( ); s++) {      /* cout ¡¡ *s ¡¡ endl; */
    if (egg_index.find(*s) ≠ egg_index.end( )) {
      es→egg_number.push_back(*s);
    }
  }
  es→eggs_reporting = es→egg_number.size( );
  int e;
  ⟨ Prepare egg table and map for extracted data set 59 ⟩;
```

This code is used in section 39.

42. All that remains is to allocate the egg data table for the extracted eggs and copy the data into it. Note that since the extract covers the same time span as the original data set and only eggs present in the source are included in the extract, there is no need to pre-fill the data table with *MissingSample*. We copy the data cowboy-style in the interest of not spending all day here.

```
⟨ Copy data for extracted eggs 42 ⟩ ≡
  es→free_egg_data( );
  es→allocate_egg_data( );
  for (e = 0; e < es→eggs_reporting; e++) {
    Sample *dstp = es→egg_cell(0, e), *srcp = egg_cell(0, egg_number_to_index(es→egg_number[e]));
    int nrows = seconds_of_data / seconds_per_row;
    for (int j = 0; j < nrows; j++) {
      *dstp = *srcp;
      dstp += es→eggs_reporting;
      srcp += eggs_reporting;
    }
  }
```

This code is used in section 39.

43. Describing the data set in human-readable form.

Our *describe* method writes global information for the data table to the designated output stream. Derived classes may call this parent class method to write the header for the more detailed information they describe.

```
⟨ Class definitions 8 ⟩ +≡
  template⟨class Sample⟩ void generic_eggsummary⟨Sample⟩::describe(ostream &os)
  {
    os ≪ "Egg␣summary:" ≪ endl ≪ "␣␣Start␣time:␣" ≪ start_time.dateToString( ) ≪ "␣" ≪
        start_time.timeToString( ) ≪ ".␣␣End␣time:␣" ≪ end_time.dateToString( ) ≪ "␣" ≪
        end_time.timeToString( );
    if (pseudorandom_data) {
      os ≪ "␣(Pseudorandom␣data)";
    }
    os ≪ endl;
    os ≪ "␣␣Seconds␣of␣data:␣" ≪ seconds_of_data ≪ "␣␣Seconds␣per␣row:␣" ≪ seconds_per_row ≪
        "␣␣Bytes␣per␣sample:␣" ≪ sampleBytes ≪ endl;
    os ≪ "␣␣Samples␣per␣record:␣" ≪ samples_per_record ≪ "␣␣Seconds␣per␣record:␣" ≪
        seconds_per_record ≪ "␣␣Records␣per␣packet:␣" ≪ records_per_packet ≪ "␣␣Trial␣size:␣" ≪
        trial_size ≪ endl;
    os ≪ "␣␣Eggs␣reporting:␣" ≪ eggs_reporting ≪ ":";
    for (int i = 0; i < eggs_reporting; i++) {
      if ((i % 10) ≡ 9) {
        os ≪ endl ≪ "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣";
      }
      os ≪ "␣" ≪ egg_number[i];
    }
    os ≪ endl;
  }
```

44. Egg databases class.

We have the ability to access any number of individual egg databases. There are usually two databases, one containing "live" data from the random event generators and a mirror containing pseudorandom data for control studies, but one can define additional databases with different selection criteria: for example, type of random event generator or general geographic location. The *eggdatabases* class maps a database category name into a path prefix where the `basketdata` files for that category are to be found.

```
⟨ Class definitions 8 ⟩ +≡
  class eggdatabases {
  private:
    map⟨string, string⟩ cat_to_path;
  public:
    eggdatabases( )
    { }
    void add_database(string dbname, string path)
    {
      cat_to_path.insert(make_pair(dbname, path));
    }
    string path(string database_name)
    {
      map⟨string, string⟩::const_iterator p = cat_to_path.find(database_name);
      if (p ≡ cat_to_path.end( )) {
        throw (out_of_range("eggdatabases::path: unknown database name"));
      }
      return p→second;
    }
    string database_file(systemtime t, string which = "gcp");
    ⟨ Define database locations for archive sites 46 ⟩;
  };
```

45. The *database_file* method returns the file name containing the data for a given **systemtime**.

```
⟨ Egg data utilities 9 ⟩ +≡
  string eggdatabases::database_file(systemtime t, string which)
  {
    return path(which) + "/basketdata-" + t.dateToString( ) + ".csv.gz";
  }
```

46. We provide functions to pre-initialise **eggdatabases** for the locations of the real and pseudo data egg summary files at the main Global Consciousness Project site and the backup mirror at Fourmilab.

```
⟨ Define database locations for archive sites 46 ⟩ ≡
  void set_Fourmilab_defaults(void)
  {
    add_database("gcp", "/files/Server/Backup/egg/eggsummary");
    add_database("pseudo", "/files/Server/Backup/egg/pseudoeggsummary");
  }
  void set_noosphere_defaults(void)
  {
    add_database("gcp", "/home/httpd/html/data/eggsummary");
    add_database("pseudo", "/home/httpd/html/data/pseudoeggsummary");
  }
```

See also section 47.

This code is used in section 44.

47. Finally, as a *really* dirty trick, we define a method which configures the proper defaults for the host we're running on based on the automatically sensed host name from the configurator. This permits writing code which adapts to any mirror of the database without any source code changes.

```
⟨ Define database locations for archive sites 46 ⟩ +≡
#ifdef HOSTNAME
  void set_local_defaults(void)
  {
    if (HOSTNAME ≡ "hayek.lan.fourmilab.ch") {
      set_Fourmilab_defaults( );
    }
    else if (HOSTNAME ≡ "noosphere.princeton.edu") {
      set_noosphere_defaults( );
    }
    else {
      string em = "eggdatabases::set_local_defaults: unknown host name ";
      em += HOSTNAME;
      throw (out_of_range(em.c_str( )));
    }
  }
#endif
```

48. Egg summary cache.

To provide for flexible and efficient access to archived `eggsummary` databases, we interpose the *eggsummary_cache* class between the archived data and in-memory representations. By going through this class, accessors need not worry about loading databases from disc or worrying about exhausting memory by keeping too many databases in memory simultaneously.

The cache is defined as a template class *generic_eggsummary_cache* which can be instantiated with any class derived from **eggsummary**; we provide a **typedef** of *eggsummary_cache* to create a cache of the basic **eggsummary** class. Using a template permits analysis code which extends the basic **eggsummary** to benefit from caching of its specialised database objects.

At the moment, the implementation of the cache simply accumulates **eggsummary** objects in memory without bound. In the future, it will be upgraded to limit memory consumption to a specified upper bound by flushing out cached data via an LRU algorithm.

```
⟨ Class definitions 8 ⟩ +≡
  template⟨class T⟩ class generic_eggsummary_cache { private:    /* The name_cache maps the
      path name used to load an egg summary file into the in-memory cached version of the file. */
  map ⟨string, T *> name_cache;
  eggdatabases *eDB;
  string eWhich;
  string rottenEggDatabase;
  int minSample, maxSample;
public:
  void setEggDatabases(eggdatabases *ed = 0, string which = "gcp")
  {
    eDB = ed;
    eWhich = which;
  }
  void setRottenEgg(string rottenEgg = "")
  {
    rottenEggDatabase = rottenEgg;
  }
  void setSampleLimits(int minSamp = −1, int maxSamp = 1024)
  {
    minSample = minSamp;
    maxSample = maxSamp;
  }
  generic_eggsummary_cache(eggdatabases *ed = 0, string rottenEgg = "", int minSamp = −1, int
      maxSamp = 1024, string which = "gcp")
  {
    setEggDatabases(ed, which);
    setRottenEgg(rottenEgg);
    setSampleLimits(minSamp, maxSamp);
  }
  T * get(string path_name);
  T * get_by_date(string date);
  T * get_by_date(systemtime &t);
  T * get_by_date(time_t tt);
  ~generic_eggsummary_cache( )
  {
    purge( );
  }
```

```
  void purge(void);      /* Extract data for a specified time range into a composite eggsummary */
  void extract_time_range(T * es, systemtime begin, systemtime end, int sec_row = 1); } ;
  typedef generic_eggsummary_cache⟨eggsummary⟩ eggsummary_cache;
```

49. The *get* method takes the path name used to load an eggsummary file and returns an in-memory **eggsummary** containing its contents, from the cache if already loaded, or after loading the data from the file and adding it to the cache.

```
⟨ Class definitions 8 ⟩ +≡
  template⟨class T⟩ T*generic_eggsummary_cache⟨T⟩::get(string path_name){ typename map
          ⟨string, T *> ::iterator item;
      item = name_cache.find(path_name);
      if (item ≡ name_cache.end()) {
        T * es = new T();
        es→load_from_CSV(path_name);
        es→pseudorandom_data = eWhich ≡ "pseudo";
        ⟨ Exclude bad samples from eggsummary loaded into cache 51 ⟩;
        name_cache.insert(make_pair(path_name, es));
#ifdef CACHE_DEBUG
        cout ≪ "Loaded␣" ≪ path_name ≪ "␣from␣file.\n";
        cout.flush();
#endif
        return es;
      }
#ifdef CACHE_DEBUG
      cout ≪ "Returned␣" ≪ path_name ≪ "␣from␣cache.\n";
      cout.flush();
#endif
      return item→second; }
```

50. If a source of Egg databases has been provided, we permit the user to request a database simply by supplying its date. Polymorphic functions are provided to permit specifying this date as a string in ISO-8601 format, a **systemtime**, or a UNIX **time_t** value.

```
⟨ Class definitions 8 ⟩ +≡
  template⟨class T⟩ T*generic_eggsummary_cache⟨T⟩::get_by_date(string date)
  {
    systemtime t;
    t.fromString(date);
    return get(eDB→database_file(t, eWhich));
  }
  template⟨class T⟩ T*generic_eggsummary_cache⟨T⟩::get_by_date(systemtime &t)
  {
    return get(eDB→database_file(t, eWhich));
  }
  template⟨class T⟩ T*generic_eggsummary_cache⟨T⟩::get_by_date(time_t tt)
  {
    systemtime t(tt);
    return get(eDB→database_file(t, eWhich));
  }
```

51. When we load an eggsummary file, if the user has specified a database of known bad data (see ⟨ Process record from bad sample database 25 ⟩ for details) or limits outside which a sample is considered invalid and discarded, we enforce those limits immediately the **eggsummary** for the day loaded into the cache. Limitation of data to a defined range is done silently; if you wish to log out of range samples, you'll have to call *limit_to_range* yourself after the cache returns the **eggsummary**.

```
⟨ Exclude bad samples from eggsummary loaded into cache 51 ⟩ ≡
  if (rottenEggDatabase ≠ "") {
    es→exclude_bad_data(rottenEggDatabase);
  }
  if (minSample ≥ 0) {
    es→limit_to_range(minSample, maxSample);
  }
```

This code is used in section 49.

52. The *purge* method empties the cache, releasing all of the in-memory **eggsummary** objects in it.

```
⟨ Class definitions 8 ⟩ +≡
  template⟨class T⟩ void generic_eggsummary_cache⟨T⟩::purge(void){ typename map ⟨string,
        T *> ::iterator item;      /* Delete all in-memory eggsummary objects */
    for (item = name_cache.begin(); item ≠ name_cache.end(); item++) {
      delete item→second;
    }
    name_cache.clear();
#ifdef CACHE_DEBUG
    cout ≪ "Purged␣map.\n";
    cout.flush();
#endif
    }
```

53. Extraction of data for a time interval.

The cache provides an *extract_time_range* method with the same arguments as the eponymous method of the **generic_eggsummary** class but, provided the cache has been properly initialised with the source of the required egg summary files, is able to produce an extract of any time period with a specified time resolution. We create the resulting **eggsummary** in an existing structure, dynamically allocating the *egg_data* table as required.

```
⟨ Class definitions 8 ⟩ +≡
#undef CETR_DEBUG
  template⟨class T⟩ void generic_eggsummary_cache⟨T⟩::extract_time_range(T * es, systemtime
        begin, systemtime end, int sec_row)
  {
    assert(sec_row ≡ 1);      /* Aggregation of data into longer intervals isn't implemented yet! */
    assert((begin.get_time() < end.get_time()));      /* Begin time ≥ end time */
    int e;
    ⟨ Initialise header for extracted data set 54 ⟩;
    ⟨ Perform first pass scan of days included in extracted data set 55 ⟩;
    ⟨ Prepare egg table and map for extracted data set 59 ⟩;
    ⟨ Perform second pass, extracting data from individual days 60 ⟩;
  }
```

54. We start by plugging in the few header fields we know directly from the arguments and clear out any existing data in the **eggsummary**.

```
⟨ Initialise header for extracted data set 54 ⟩ ≡
  es→free_egg_data( );
  es→seconds_of_data = (end.get_time( ) − begin.get_time( )) + 1;
  es→start_time = begin;
  es→end_time = end;
  es→seconds_per_row = sec_row;
  es→egg_index.clear( );
  es→egg_number.clear( );
  es→eggs_reporting = 0;
```

This code is used in section 53.

55. Before we can allocate the data table and load it with data for the requested period, we first need to determine how many unique eggs have contributed data for the period in question. Individual **eggsummary** objects for a day only contain a column for an egg if it contributed a sample that day, and hence when the requested intervals spans more than one day, the days contributing to the extract may differ in the number of eggs they report and the assignment of egg numbers to columns in the table. We make a preliminary pass over the days in the interval (bringing them into the cache), creating a list of all eggs in all days. Note that if an egg is present in a day in the interval but did not actually contribute any samples within the interval itself, the egg *will* be listed in the result, but all of its samples will be *MissingSample*.

```
⟨ Perform first pass scan of days included in extracted data set 55 ⟩ ≡
  systemtime t(begin.midnight( ));
  bool firstLaid = true;
  do {
#ifdef CETR_DEBUG
    cout ≪ "First␣pass␣loading␣" ≪ t.dateToString( ) ≪ endl;
#endif
    T ∗ ds = get_by_date(t);
    if (firstLaid) {
      ⟨ Set properties of extract from those of first day's data 56 ⟩;
      firstLaid = false;
    }
    else {
      ⟨ Verify consistency of parameters for all days in extract period 57 ⟩;
    }
    ⟨ Add new egg numbers to extract egg table 58 ⟩;
    t.nextDay( );
  } while (t.get_time( ) < end.get_time( ));
```

This code is used in section 53.

56. All the days contributing to the extract are assumed to have the same experimental parameters (we verify this below). When processing the first day's data, we set the parameters of the extract from its header.

```
⟨ Set properties of extract from those of first day's data 56 ⟩ ≡
  es→samples_per_record = ds→samples_per_record;
  es→seconds_per_record = ds→seconds_per_record;
  es→records_per_packet = ds→records_per_packet;
  es→trial_size = ds→trial_size;
  es→pseudorandom_data = ds→pseudorandom_data;
```

This code is used in section 55.

57. As noted above, we require that all days contributing data to the extract share the same experimental parameters. Having set the parameters of the extract from the first day's header, we confirm here that they haven't changed on subsequent days.

```
⟨ Verify consistency of parameters for all days in extract period 57 ⟩ ≡
  if ((es→samples_per_record ≠ ds→samples_per_record) ∨ (es→seconds_per_record ≠
      ds→seconds_per_record) ∨ (es→records_per_packet ≠ ds→records_per_packet) ∨ (es→trial_size ≠
      ds→trial_size) ∨ (es→pseudorandom_data ≠ ds→pseudorandom_data)) {
    throw (out_of_range("generic_eggsummary_cache::extract_time_range: Parameters di\
      ffer in days in extract"));
  }
```

This code is used in section 55.

58. For each day within the interval, we look up each egg in the day's egg table in the egg number to column map we're building for the extract and add it if it's not already present.

```
⟨ Add new egg numbers to extract egg table 58 ⟩ ≡
  for (e = 0; e < ds→eggs_reporting; e++) {
    if (es→egg_index.find(ds→egg_number[e]) ≡ es→egg_index.end()) {
      es→egg_number.push_back(ds→egg_number[e]);
      es→egg_index.insert(make_pair(ds→egg_number[e], es→eggs_reporting));
      es→eggs_reporting++;
    }
  }
```

This code is used in section 55.

59. After completing the first pass, we have a list of all eggs whose data appear in the interval, but this list isn't necessarily sorted in ascending order of egg number. This isn't strictly required, but since **eggsummary** files for individual days are, in fact, sorted, we sort the egg table and re-build the map using the sorted order here.

```
⟨ Prepare egg table and map for extracted data set 59 ⟩ ≡
  sort(es→egg_number.begin(), es→egg_number.end());
  es→egg_index.clear();
  for (e = 0; e < es→eggs_reporting; e++) {
    es→egg_index.insert(make_pair(es→egg_number[e], e));
  }
```

This code is used in sections 41 and 53.

60. Now that we have an egg table covering the entire extract, we need only walk back through the days in the extract and actually copy the requested data to the extract. We begin by allocating the egg data table and initialising it to *MissingSample*.

```
⟨ Perform second pass, extracting data from individual days 60 ⟩ ≡
  int egg_data_size = es→eggs_reporting * (es→seconds_of_data / es→seconds_per_row);
  es→allocate_egg_data( );
  for (e = 0; e < egg_data_size; e++) {
    es→egg_data[e] = es→MissingSample;
  }
  t.set_time(begin.midnight( ));
  systemtime now = begin;
  firstLaid = true;
  int nr = 0;
  do {
#ifdef CETR_DEBUG
    cout ≪ "Second␣pass␣loading␣" ≪ t.dateToString( ) ≪ endl;
#endif
    T * ds = get_by_date(t);
    if (firstLaid) {
      e = begin.get_time( ) − ds→start_time.get_time( );
      assert((e ≥ 0) ∧ (e < systemtime::SecondsPerDay));
      firstLaid = false;
    }
    else {
      e = 0;
    }
    ⟨ Copy day's data to extracted data set 61 ⟩;
    t.nextDay( );
  } while (t.get_time( ) < end.get_time( ));
```

This code is used in section 53.

61. Transcribe the egg data within the extract interval from this day's **eggsummary**. This is presently written as "clean but slow" code—it looks up the egg column in the destination map for every sample it stores. This could be optimised by computing a vector of column transformations once for each day's data, but in practice this wouldn't speed things up very much because the time spent in this code is very small compared to loading the CSV files into memory.

```
⟨ Copy day's data to extracted data set 61 ⟩ ≡
  while ((now.get_time( ) ≤ ds→end_time.get_time( )) ∧ (now.get_time( ) ≤ es→end_time.get_time( ))) {
    for (int n = 0; n < ds→eggs_reporting; n++) {
      es→set_egg_index(nr, es→egg_number_to_index(ds→egg_number[n]), ds→get_egg_index(e, n));
    }
    e++;
    nr++;
    now.set_time(now.get_time( ) + 1);
  }
```

This code is used in section 60.

62. Egg Properties Database.

The Egg Properties database describes the characteristics of each egg in the network. Each egg is identified by a unique "egg number", the range of which also identifies the type of random event generator employed by the egg.

We begin by introducing the class which expresses the properties of an individual egg.

```
⟨ Class definitions 8 ⟩ +≡
  class egg_properties {
  public:
    static const int unknownAltitude = −9999;      /* Code for unknown altitude */
    unsigned int eggNumber;
    double latitude, longitude, altitude;
    string location, REGtype, hostName;      /* Initialise fields from a CSV database record */
    void setFromCSV(string s);      /* Write description to output stream */
    void describe(ostream &os = cout);      /* Write tabular representation to output stream */
    void tabulate(ostream &os = cout);
    bool isAltitudeKnown(void)
    {
      return altitude ≠ unknownAltitude;
    }
  };
```

63. The egg properties database is normally loaded from a CSV file. Given a line from the CSV file, this method parses the fields and initialises the properties from them. Note that discarding comment lines must be done before this method is called.

```
⟨ Egg data utilities 9 ⟩ +≡
  void egg_properties::setFromCSV(string s)
  {
    csv_parser p(s);
    string f;
    int i, m_type, s_type, arg;
    if (¬p.next_field(f) ∨ f.length( ) < 1 ∨ ¬isdigit(f[0])) {
      throw (out_of_range("egg_properties::setFromCSV: Invalid record type"));
    }
    m_type = atoi(f.c_str( ));
    switch (m_type) {
    case 41: ⟨ Process record from egg property database 64 ⟩;
      break;
    default: throw (out_of_range("egg_properties::setFromCSV: Invalid record type"));
      break;
    }
  }
```

64. Records in the egg property database have the following form:

`41`, *Egg number*, *Latitude*, *Longitude*, *Altitude*, *Location*, *REG type*, *Host name*

The record begins with the record identifier `41`. The *Egg number* is a unique identifier assigned to the egg. The *Latitude* and *Longitude* are given as floating point numbers, with negative numbers denoting South latitude and East longitude. *Altitude* is in metres above (or, if negative, below) mean sea level; if the altitude is not known, this field is given as `"NA"` and the *altitude* will be set to *unknownAltitude*: -9999. The *Location* is a text field identifying the site at which the egg host is installed: if known, this should be its Unix time zone designation to permit conversion from UTC to local time. The *REG type* identifies the kind of random event generator employed by the egg: this is a text field. If the egg has a permanently assigned fully qualified domain name (for example, "`noosphere.princeton.edu`", it should be given in the *Host name* field, or `"NA"` for dial-up eggs or those with a dynamically assigned host name.

⟨ Process record from egg property database 64 ⟩ ≡

```
if (¬p.next_field(f) ∨ f.length( ) < 1 ∨ ¬isdigit(f[0])) {
  throw (out_of_range("egg_properties::setFromCSV: Invalid egg number in type 41 r\
      ecord"));
}
eggNumber = atoi(f.c_str( ));
assert(eggNumber > 0);
if (¬p.next_field(f) ∨ f.length( ) < 1) {
  throw (out_of_range("egg_properties::setFromCSV: Invalid latitude in type 41 record"));
}
latitude = atof(f.c_str( ));
assert(latitude ≥ −90 ∧ latitude ≤ 90);
if (¬p.next_field(f) ∨ f.length( ) < 1) {
  throw (out_of_range("egg_properties::setFromCSV: Invalid longitude in type 41 record"));
}
longitude = atof(f.c_str( ));
assert(longitude ≥ −180 ∧ longitude ≤ 180);
if (¬p.next_field(f) ∨ f.length( ) < 1) {
  throw (out_of_range("egg_properties::setFromCSV: Invalid altitude in type 41 record"));
}
if ((f ≡ "NA") ∨ (f ≡ "?")) {
  altitude = unknownAltitude;
}
else {
  altitude = atof(f.c_str( ));
  assert(longitude ≥ −180 ∧ longitude ≤ 180);
}
if (¬p.next_field(f) ∨ f.length( ) < 1) {
  throw (out_of_range("egg_properties::setFromCSV: Invalid location in type 41 record"));
}
location = f;
if (¬p.next_field(f) ∨ f.length( ) < 1) {
  throw (out_of_range("egg_properties::setFromCSV: Invalid REG type in type 41 record"));
}
REGtype = f;
if (¬p.next_field(f) ∨ f.length( ) < 1) {
  throw (out_of_range("egg_properties::setFromCSV: Invalid host name in type 41 record"));
}
if (f ≡ "NA") {
  f = "";
}
```

```
  hostName = f;
```

This code is used in section 63.

65. The *describe* method sends a primate-readable description of the egg properties to the designated output stream.

```
⟨ Egg data utilities 9 ⟩ +≡
  void egg_properties::describe(ostream &os)
  {
    os ≪ "Egg␣" ≪ eggNumber ≪ ":" ≪ endl;
    os ≪ "␣␣Latitude:␣" ≪ latitude ≪ "␣␣Longitude:␣" ≪ longitude ≪ "␣␣Altitude:␣" ≪ altitude ≪
        endl;
    os ≪ "␣␣Location:␣" ≪ location ≪ "␣␣REG␣type:␣" ≪ REGtype ≪ "␣␣Host␣name:␣" ≪
        hostName ≪ endl;
  }
```

66. The *tabulate* method is much like *describe* above, but outputs the information in a spiffy columnar format. Note the crazy requirement to break out a C string from our **string** objects; if you don't, looney **ostream** won't fill and justify the field!

```
⟨ Egg data utilities 9 ⟩ +≡
  void egg_properties::tabulate(ostream &os)
  {
    os.setf(ios::fixed, ios::floatfield);
    os.precision(3);
    os ≪ setw(4) ≪ eggNumber ≪ "␣␣" ≪ setw(10) ≪ latitude ≪ "␣" ≪ setw(10) ≪ longitude ≪ "␣";
    if (isAltitudeKnown()) {
      os ≪ setw(6) ≪ setprecision(0) ≪ altitude ≪ "␣␣";
    }
    else {
      os ≪ "␣␣N.A.␣␣";
    }
    os.setf(ios::left, ios::adjustfield);
    os ≪ setw(10) ≪ REGtype.c_str() ≪ "␣" ≪ setw(24) ≪ location.c_str() ≪ "␣" ≪
        hostName.c_str() ≪ endl;
    os.setf(ios::right, ios::adjustfield);
    os.precision(6);
  }
```

67. OK, with these preliminaries out of the way, we're now ready to implement the egg property database class. This is simply a list of **egg_properties** objects with tools to load and access them.

```
⟨ Class definitions 8 ⟩ +≡
  class egg_properties_database {
  private:
    map⟨unsigned int, egg_properties *⟩ eggs;
    vector⟨egg_properties *⟩ eggv;
  public:
    void loadFromCSV(istream &i);
    void loadFromCSV(string fileName);
    void describe(ostream &o = cout);
    void tabulate(ostream &o = cout);
    unsigned int size(void);
    egg_properties *find(unsigned int egg_number);
    egg_properties *operator[](int n);
  };
```

68. The *size* method simply returns the number of eggs in the database.

```
⟨ Egg data utilities 9 ⟩ +≡
  unsigned int egg_properties_database::size(void)
  {
    return eggs.size();
  }
```

69. The *find* method looks up an egg in the properties database and returns a pointer to its **egg_properties**. If the egg number is unknown, Λ is returned.

```
⟨ Egg data utilities 9 ⟩ +≡
  egg_properties *egg_properties_database::find(unsigned int egg_number)
  {
    map⟨unsigned int, egg_properties *⟩::iterator p = eggs.find(egg_number);
    return (p ≡ eggs.end()) ? Λ : p→second;
  }
```

70. We overload the [] operator to provide access to items in the *eggs* table by index.

```
⟨ Egg data utilities 9 ⟩ +≡
  egg_properties *egg_properties_database::operator[](int n)
  {
    return eggv[n];
  }
```

71. The *loadFromCSV* method adds eggs described by a records in a CSV file to the database.

```
⟨ Egg data utilities 9 ⟩ +≡
  void egg_properties_database::loadFromCSV(istream &i)
  {
    string s;
    while (getline(i, s)) {
      if ((s.length() > 0) ∧ (s[0] ≠ ';') ∧ (s[0] ≠ '#')) {
        egg_properties *ep = new egg_properties;
        ep→setFromCSV(s);
        eggs.insert(make_pair(ep→eggNumber, ep));
        eggv.push_back(ep);
      }
    }
  }
```

72. For convenience, we define a variant of *loadFromCSV* which accepts a file name instead of an already-open input stream.

```
⟨ Egg data utilities 9 ⟩ +≡
  void egg_properties_database::loadFromCSV(string fileName)
  {
    ifstream icsv(fileName.c_str());
    loadFromCSV(icsv);
  }
```

73. The *describe* method simply iterates through all of the eggs in the *eggs* map and asks each to describe itself. *tabulate* does the same thing, but calls the member's *tabulate* method instead.

```
⟨ Egg data utilities 9 ⟩ +≡
  void egg_properties_database::describe(ostream &o)
  {
    map⟨unsigned int, egg_properties *⟩::iterator e = eggs.begin();
    while (e ≠ eggs.end()) {
      (e→second)→describe(o);
      e++;
    }
  }
  void egg_properties_database::tabulate(ostream &o)
  {
    map⟨unsigned int, egg_properties *⟩::iterator e = eggs.begin();
    o ≪ "␣Egg␣␣␣␣Latitude␣␣Longitude␣␣␣␣Alt␣␣REG␣Type␣␣␣Location␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Host" ≪
        endl;
    while (e ≠ eggs.end()) {
      (e→second)→tabulate(o);
      e++;
    }
  }
```

74. Test program.

```
⟨ eggdata_test.c 1 ⟩ +≡
  ⟨ Test program include files 77 ⟩;
  ⟨ Show how to call test program 76 ⟩;
  int main(int argc, char *argv[])
  {
    extern char *optarg;      /* Imported from getopt */
    extern int optind;
    try {
      int opt;
      ⟨ Process command-line options 75 ⟩;
#if 0     /* Eggsummary file I/O tests */
      eggsummary es;
#if 0
      ifstream ci("../test.csv");
      es.load_from_CSV(ci);
      ci.close();
#endif
      es.load_from_CSV("../test.csv.gz");
      for (int j = 0; j < es.eggs_reporting; j++) {
        cout << es.egg_number[j] << ": " << (int) es.get_egg_index(86399, j) << "\n";
      }
#endif
#if 1     /* Egg properties database tests */
      {
        egg_properties_database ed;
        ed.loadFromCSV("eggs.csv");
        ed.tabulate();
      }
#endif
#if 0     /* Egg database tests */
      {
        eggdatabases ed;
        systemtime t;
        ed.set_Fourmilab_defaults();
        t.fromString("2001-04-21 18:21:19");
        cout << ed.database_file(t) << endl;
        t.nextDay();
        cout << ed.database_file(t, "pseudo") << endl;
      }
#endif
#if 0     /* Time shifting cache tests */
      {
        int j;
        eggsummary esr;
        esr.load_from_CSV("../test.csv.gz");
        int eggno = 7;
        for (j = 0; j < 10; j++) {
          cout << j << ": " << setw(3) << ((int) esr.get_egg_index(j, eggno)) << "  " << setw(3) << ((int)
              esr.get_egg_index((86400 - 10) + j, eggno)) << endl;
```

```
    }
    esr.time_shift(eggno, 0);
    cout ≪ endl ≪ "␣0" ≪ endl;
    for (j = 0; j < 10; j++) {
      cout ≪ j ≪ ":␣" ≪ setw(3) ≪ ((int) esr.get_egg_index(j, eggno)) ≪ "␣␣" ≪ setw(3) ≪ ((int)
          esr.get_egg_index((86400 − 10) + j, eggno)) ≪ endl;
    }
    esr.time_shift(eggno, −5);
    cout ≪ endl ≪ "␣-5" ≪ endl;
    for (j = 0; j < 10; j++) {
      cout ≪ j ≪ ":␣" ≪ setw(3) ≪ ((int) esr.get_egg_index(j, eggno)) ≪ "␣␣" ≪ setw(3) ≪ ((int)
          esr.get_egg_index((86400 − 10) + j, eggno)) ≪ endl;
    }
    esr.load_from_CSV("../test.csv.gz");
    esr.time_shift(eggno, 5);
    cout ≪ endl ≪ "␣+5" ≪ endl;
    for (j = 0; j < 10; j++) {
      cout ≪ j ≪ ":␣" ≪ setw(3) ≪ ((int) esr.get_egg_index(j, eggno)) ≪ "␣␣" ≪ setw(3) ≪ ((int)
          esr.get_egg_index((86400 − 10) + j, eggno)) ≪ endl;
    }
    esr.load_from_CSV("../test.csv.gz");
    esr.time_shift(eggno, −90000);
    cout ≪ endl ≪ "␣-90000" ≪ endl;
    for (j = 0; j < 10; j++) {
      cout ≪ j ≪ ":␣" ≪ setw(3) ≪ ((int) esr.get_egg_index(j, eggno)) ≪ "␣␣" ≪ setw(3) ≪ ((int)
          esr.get_egg_index((86400 − 10) + j, eggno)) ≪ endl;
    }
  }
#endif
#if 0  /* Cache range extract tests */
  {
    eggdatabases ed;
    ed.set_Fourmilab_defaults( );
    generic_eggsummary_cache⟨eggsummary⟩ ec(&ed, "rotten_egg.csv", 50, 150);
    eggsummary ext;
    systemtime exstart, exend;
    exstart.fromString("2001-04-20␣13:00:00");
    exend.fromString("2001-04-24␣22:30:00");
    ec.extract_time_range(&ext, exstart, exend);
    ext.describe( );
    ext.save_to_CSV("/tmp/extract.csv", true);
  }
#endif
#if 0  /* Eggsummary cache tests */
  {
    int j;
    eggsummary *es;
    eggsummary_cache ec;
    es = ec.get("../test.csv.gz");
    eggsummary &esr = *es;
    for (j = 0; j < esr.eggs_reporting; j++) {
```

```
      cout ≪ esr.egg_number[j] ≪ ": " ≪ (int) esr.get_egg_index(86399, j) ≪ "\n";
    }     /* Once more to test cache */
    es = ec.get("../test.csv.gz");
    esr = *es;
    for (j = 0; j < esr.eggs_reporting; j++) {
      cout ≪ esr.egg_number[j] ≪ ": " ≪ (int) esr.get_egg_index(86399, j) ≪ "\n";
    }
  }
#endif
#if 0    /* Interactively test CSV parsing */
  while ((cout ≪ "--> "), getline(cin, s)) {
    csv_parser p(s);
    string f;
    cout ≪ s;
    cout ≪ "\n";
    while (p.next_field(f)) {
      cout ≪ "{" + f + "}\n";
    }
  }
#endif
  }
  catch(exception &e)
  {
    cout ≪ "Blooie!!!  Exception popped: " ≪ e.what( ) ≪ endl;
#ifndef CORE_DUMP
#ifdef STACK_TRACE
    char s[160];
    sprintf(s,
        "/bin/echo 'where\nq' >/tmp/gdbcmd ; gdb  -batch --command ""/tmp/gdbcmd %s %d",
        argv[0], getpid( ));
    system(s);
    sleep(5);
#endif
    throw;    /* Re-throw exception to dump core */
#endif
  }
  return 0;
}
```

75. 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 75 ⟩ ≡
  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 74.

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

```
⟨ Show how to call test program 76 ⟩ ≡
  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 74.

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

⟨ Test program include files 77 ⟩ ≡

```
#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 "eggdata.h"      /* Class definitions for this package */
```

This code is used in section 74.

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

EGGDATA