```
// TicketRep.cpp - GENERATE THE TICKET FINANCIAL SUMMARY REPORT
//
// USAGE
// TicketRep -D from-date to-date
// from-date       is the reporting period from-date
// to-date         is the reporting period to-date
//
// MODULE INDEX
// NAME                CONTENTS
// FormatNum           Format a number
// main                Main line
//
// MAINTENANCE HISTORY
// DATE          PROGRAMMER AND DETAILS
// 29-08-10 JS   Original
//
//-----------------------------------------------------------------------------

#include <cstdlib>       // C-style standard library
#include <cstring>       // C-style string manipulation functions
#include <cctype>        // C-style character typing functions
#include <iostream>      // C++ input/output streams
#include <iomanip>       // C++ input/output manipulators
#include <sstream>       // C++ string stream declarations
#include <set>           // C++ set declarations
#include <map>           // C++ map declarations
using namespace std;     // Expand the standard namespace
#include <unistd.h>      // Unix standard functions
#include "dtime.h"       // Double time declarations
exec sql include sqlca; // Include SQL communications area

//-----------------------------------------------------------------------------

// TICKET STATUS CODES

exec sql begin declare section;
    static const short TICKET_STATUS_ISSUED = 1;
                // The ticket has been issued by a ride,
                // but not yet redeemed
    static const short TICKET_STATUS_ACCEPTED = 2;
                // The ticket has been accepted by a ride
    static const short TICKET_STATUS_CASHED = 3;
                // The ticket has been cashed at an ATM
    static const short TICKET_STATUS_CANCELLED = 4;
                // The ticket has been cancelled
    static const short TICKET_STATUS_EXPIRED = 5;
                // The ticket has expired
exec sql end declare section;

//-----------------------------------------------------------------------------

// TRANSACTION TYPE CODES

exec sql begin declare section;
    static const short TX_TYPE_ISSUE = 1;
                // Issue by a ride
    static const short TX_TYPE_ACCEPT = 2;
```

```
                    // Acceptance by a ride
    static const short TX_TYPE_CASH = 3;
                    // Encashment at an ATM
    static const short TX_TYPE_CANCEL = 4;
                    // Cancellation
    static const short TX_TYPE_EXPIRE = 5;
                    // Expiry
exec sql end declare section;

//------------------------------------------------------------------------

// GLOBAL DATA

Dtime_t     fromDate;       // From date
Dtime_t     toDate;         // To date

//------------------------------------------------------------------------

// FORMAT A NUMBER

string
FormatNum (
    double      n,      // Number to format
    size_t      dec)    // Decimal places
{
    size_t      i;      // General purpose index
    double      prevN;  // Previous value of n
    size_t      grpCnt; // Group count
    char        buf[20], *p;    // Formatting variables
    bool        negative;   // Number is negative flag

    if (n > 1.0e10) {
        cerr << "Error: numeric overflow\n";
        exit (0);
    }

    p = buf + sizeof(buf);
    *--p = '\0';
    negative = 0;
    if (n < 0) {
        negative = 1;
        n = -n;
    }
    for (i = 0; i < dec; i++) {
        prevN = n;
        modf (n/10, &n);
        *--p = static_cast <char> (prevN - 10*n + '0');
    }
    if (dec != 0) *--p = '.';
    grpCnt = 0;
    do {
        if (grpCnt >= 3) {
            *--p = ',';
            grpCnt = 0;
        }
        prevN = n;
        modf (n/10, &n);
```

```
        if (p == buf) {
            cerr << "Error: number too big\n";
            exit (1);
        }
        *--p = static_cast <char> (prevN - 10*n + '0');
        grpCnt ++ ;
    } while (n != 0);
    if (negative) *--p = '-';
    return string(p);
}

//---------------------------------------------------------------------

// MAIN LINE

int
main (
    int     argc,       // Argument count
    char        *argv[])    // Argument value pointers
{
    int     c;          // Argument character
    bool        periodDefined;  // Reporting period defined flag
    char        dateBuf[10];    // Date formatting buffer
    Dtime_t     tradingDate;    // Current trading date
    long        closeIssuedCnt; // Closing issued ticket count
    double      closeIssuedVal; // Closing issued ticket value
    long        openIssuedCnt;  // Opening issued ticket count
    double      openIssuedVal;  // Opening issued ticket value
    exec sql begin declare section;
        char        sqlFromDate[10+1]; // From SQL date
        char        sqlToDate[10+1]; // To SQL date
        char        sqlTradingDate[10+1]; // Current trading date
        short       dayOpen;    // Trading day open flag

        // Current values

        long        curIssuedCnt;   // Issued count
        double      curIssuedVal;   // Issued value
        short       curIssuedValInd; // Indicator for curIssuedVal

        // Rollback values

        long        rbIssueCnt; // Issue count
        double      rbIssueVal; // Issue value
        short       rbIssueValInd; // Indicator for rbIssueVal
        long        rbRedeemCnt;    // Redeemed count
        double      rbRedeemVal;    // Redeemed value
        short       rbRedeemValInd; // Indicator for rbRedeemVal

        // Reporting period values

        long        prdIssueCnt;    // Issue count
        double      prdIssueVal;    // Issue value
        short       prdIssueValInd; // Indicator for prdIssueVal
        long        prdAcceptCnt;   // Accepted count
        double      prdAcceptVal;   // Accepted value
        short       prdAcceptValInd; // Indicator for prdAcceptVal
```

```
        long        prdCashedCnt;    // Cashed count
        double      prdCashedVal;    // Cashed value
        short       prdCashedValInd; // Indicator for prdCashedVal
        long        prdCannedCnt;    // Cancelled count
        double      prdCannedVal;    // Cancelled value
        short       prdCannedValInd; // Indicator for prdCannedVal
        long        prdExpiredCnt;   // Expired count
        double      prdExpiredVal;   // Expired value
        short       prdExpiredValInd; // Indicator for prdExpiredVal
    exec sql end declare section;

    // Connect to the database

    exec sql connect to ticketdb;

    // Jump to DbError whenever an SQL error occurs

    exec sql whenever sqlerror goto DbError;

    // Decode arguments

    periodDefined = 0;
    while ((c = getopt (argc, argv, "D:")) != -1) {
        switch (c) {
        case 'D':
            fromDate = DtimeFromUserDate (optarg);
            if (fromDate == NULL_DTIME) {
                cerr << "Error: bad from-date\n";
                exit (1);
            }
            SqlDateFromDtime (sqlFromDate, fromDate);
            if (optind >= argc) {
                cerr << "Error: missing to-date\n";
                exit (1);
            }
            toDate = DtimeFromUserDate (argv[optind]);
            if (toDate == NULL_DTIME) {
                cerr << "Error: bad to-date\n";
                exit (1);
            }
            if (fromDate > toDate) {
                cerr << "Error: from-date greater than"
                    << " to-date\n";
                exit (1);
            }
            SqlDateFromDtime (sqlToDate, toDate);
            optind ++ ;
            periodDefined = 1;
            break;
        default:
            cerr << "Error: invalid option\n";
            exit (1);
            // NOTREACHED
        }
    }
    if (optind < argc) {
        cerr << "Error: superfluous arguments\n";
```

```
    exit (1);
}

// If no reporting period was defined, it's a problem

if ( ! periodDefined) {
    cerr << "Error: no reporting period\n";
    exit (1);
}

// Look up the current trading day date and open flag

exec sql select tradingDate, dayOpen
    into    :sqlTradingDate, :dayOpen
    from    dayStatus;
if (SQLCODE != 0) goto DbError;
tradingDate = DtimeFromSql (sqlTradingDate);

// Check that the reporting period does not extend past the
// current trading day

if (toDate > tradingDate) {
    cerr << "Error: to-date is after current trading day\n";
    exit (1);
}

// Load the number and value of currently issued tickets

exec sql select count(*), sum(ticketValue)
    into    :curIssuedCnt, :curIssuedVal indicator :curIssuedValInd
    from    tickets
    where   ticketStatus = :TICKET_STATUS_ISSUED;
if (curIssuedValInd < 0) curIssuedVal = 0;

// Calculate the number and value of unredeemed tickets as at
// the end of the reporting period

exec sql select count(*), sum(ticketValue)
    into    :rbIssueCnt, :rbIssueVal indicator :rbIssueValInd
    from    transactions, tickets
    where   tickets.ticketNo = transactions.ticketNo and
        transactions.txType = :TX_TYPE_ISSUE and
        transactions.txDate > :sqlToDate;
if (rbIssueValInd < 0) rbIssueVal = 0;
exec sql select count(*), sum(ticketValue)
    into    :rbRedeemCnt, :rbRedeemVal indicator :rbRedeemValInd
    from    transactions, tickets
    where   tickets.ticketNo = transactions.ticketNo and
        transactions.txType <> :TX_TYPE_ISSUE and
        transactions.txDate > :sqlToDate;
if (rbRedeemValInd < 0) rbRedeemVal = 0;
closeIssuedCnt = curIssuedCnt - rbIssueCnt + rbRedeemCnt;
closeIssuedVal = curIssuedVal - rbIssueVal + rbRedeemVal;

// Calculate the number and value of the various transactions
// during the reporting period
```

```
    exec sql select count(*), sum(ticketValue)
        into    :prdIssueCnt, :prdIssueVal indicator :prdIssueValInd
        from    transactions, tickets
        where   tickets.ticketNo = transactions.ticketNo and
            txType = :TX_TYPE_ISSUE and
            txDate >= :sqlFromDate and
            txDate <= :sqlToDate;
    if (prdIssueValInd < 0) prdIssueVal = 0;
    exec sql select count(*), sum(ticketValue)
        into    :prdAcceptCnt, :prdAcceptVal indicator :prdAcceptValInd
        from    transactions, tickets
        where   tickets.ticketNo = transactions.ticketNo and
            txType = :TX_TYPE_ACCEPT and
            txDate >= :sqlFromDate and
            txDate <= :sqlToDate;
    if (prdAcceptValInd < 0) prdAcceptVal = 0;
    exec sql select count(*), sum(ticketValue)
        into    :prdCashedCnt, :prdCashedVal indicator :prdCashedValInd
        from    transactions, tickets
        where   tickets.ticketNo = transactions.ticketNo and
            txType = :TX_TYPE_CASH and
            txDate >= :sqlFromDate and
            txDate <= :sqlToDate;
    if (prdCashedValInd < 0) prdCashedVal = 0;
    exec sql select count(*), sum(ticketValue)
        into    :prdCannedCnt, :prdCannedVal indicator :prdCannedValInd
        from    transactions, tickets
        where   tickets.ticketNo = transactions.ticketNo and
            txType = :TX_TYPE_CANCEL and
            txDate >= :sqlFromDate and
            txDate <= :sqlToDate;
    if (prdCannedValInd < 0) prdCannedVal = 0;
    exec sql select count(*),
            sum(ticketValue)
        into    :prdExpiredCnt,
            :prdExpiredVal indicator :prdExpiredValInd
        from    transactions, tickets
        where   tickets.ticketNo = transactions.ticketNo and
            txType = :TX_TYPE_EXPIRE and
            txDate >= :sqlFromDate and
            txDate <= :sqlToDate;
    if (prdExpiredValInd < 0) prdExpiredVal = 0;

    // Calculate the opening balance

    openIssuedCnt = closeIssuedCnt - prdIssueCnt + prdAcceptCnt
        + prdCashedCnt + prdCannedCnt + prdExpiredCnt;
    openIssuedVal = closeIssuedVal - prdIssueVal + prdAcceptVal
        + prdCashedVal + prdCannedVal + prdExpiredVal;

    // Show the report title

    if (toDate >= tradingDate && dayOpen)
        cout << "PRELIMINARY";
    else
        cout << "FINAL";
    cout << " TICKET FINANCIAL SUMMARY\n";
```

```
    cout << "FOR " << UserDateFromDtime (dateBuf, fromDate);
    cout << " TO " << UserDateFromDtime (dateBuf, toDate) << "\n";
    cout << setw(37) << " " << "Number           Value\n\n";

    // Show the opening balance

    cout << left << setw(31) << "Opening unredeemed tickets"
        << right << setw(12) << FormatNum (openIssuedCnt,0)
        << right << setw(16) << FormatNum (openIssuedVal,2) << '\n';

    // Show the tickets issued by rides

    cout << left << setw(31) << "Tickets issued by rides"
        << right << setw(12) << FormatNum (prdIssueCnt,0)
        << right << setw(16) << FormatNum (prdIssueVal,2) << '\n';

    // Show the tickets accepted by rides

    cout << left << setw(31) << "Tickets accepted by rides"
        << right << setw(12) << FormatNum (prdAcceptCnt,0)
        << right << setw(16) << FormatNum (prdAcceptVal,2) << '\n';

    // Show the tickets cashed at ATMs

    cout << left << setw(31) << "Tickets cashed at ATMs"
        << right << setw(12) << FormatNum (prdCashedCnt,0)
        << right << setw(16) << FormatNum (prdCashedVal,2) << '\n';

    // Show cancelled tickets

    cout << left << setw(31) << "Cancelled tickets"
        << right << setw(12) << FormatNum (prdCannedCnt,0)
        << right << setw(16) << FormatNum (prdCannedVal,2) << '\n';

    // Show expired tickets

    cout << left << setw(31) << "Expired tickets"
        << right << setw(12) << FormatNum (prdExpiredCnt,0)
        << right << setw(16) << FormatNum (prdExpiredVal,2) << '\n';

    // Show the underscore

    cout << left << setw(31) << " "
        << right << setw(12) << "-------"
        << right << setw(16) << "-----------" << '\n';

    // Show closing balance

    cout << left << setw(31) << "Closing unredeemed tickets"
        << right << setw(12) << FormatNum (closeIssuedCnt,0)
        << right << setw(16) << FormatNum (closeIssuedVal,2) << '\n';

    return 0;

    // Process database errors

DbError:
```

```
    cerr << "Error: SQLCODE=" << SQLCODE << '\n';
    return 1;
}
```