

VendRep.cpp

```

// VendGen.cpp - VENDING MACHINE DATABASE GENERATOR
//
// MODULE INDEX
// NAME                                CONTENTS
// Proc_c::SleepUntil                  Sleep until a particular time
// Proc_c::WaitUntilFree                Wait until a vending machine is free
// Cust_c::NextCustTxTime               Determine the next customer transaction time
// Cust_c::EmulateCardIn               Emulate a card-in operation
// Cust_c::EmulateBillIn               Emulate a bill-in operation
// Cust_c::EmulateSale                  Emulate a sale operation
// Cust_c::EmulateCommsFault           Emulate a communications fault
// Cust_c::EmulateCollectCard          Emulate a collect card operation
// Cust_c::CustInit                    Initialise the customer emulator
// Cust_c::Wakeup                       Process a customer wakeup stimulus
// Cust_c::VendIsFree                  Process vending machine is free event
// Attend_c::NextAttendTxTime          Determine the next attendant transaction time
// Attend_c::EmulateClearMach          Emulate clearing the machine
// Attend_c::AttendInit                Initialise the attendant emulator
// Attend_c::Wakeup                     Process an attendant wakeup stimulus
// Attend_c::VendIsFree                Process vending machine is free event
// main                                 Main line
//
// MAINTENANCE HISTORY
// DATE          PROGRAMMER AND DETAILS
// 28-09-10      JS          Original
//
//-----

#include <cstring>           // C-style string manipulation functions
#include <iostream>          // C++ input/output streams
#include <iomanip>            // C++ input/output manipulators
#include <list>              // C++ list declarations
#include <set>                // C++ set declarations
using namespace std;        // Expand the standard namespace
#include "dttime.h"          // Double time declarations
exec sql include sqlca;     // Include SQL communications area

//-----

// DEFINITIONS

static const size_t    CUST_CNT = 100;
                        // Number of customers
static const size_t    VEND_CNT = 20;
                        // Number of vending machines
static const char      FROM_DATE[] = "2010-01-01";
                        // From date
static const char      TO_DATE[] = "2010-06-30";
                        // Last date

//-----

// TRANSACTION TYPE CODES

exec sql begin declare section;
    static const short TX_TYPE_CARD_IN = 1;
                        // Transfers from a customer's account to a
                        // machine interface
    static const short TX_TYPE_COLLECT_CARD = 2;
                        // Transfers from a machine interface to a
                        // customer's account
    static const short TX_TYPE_BILL_IN = 3;
                        // Insertion of a banknote into the vending

```

VendRep.cpp

```

        // machine
        static const short TX_TYPE_SALE = 4;
        // Sale of a product
        static const short TX_TYPE_REVERSAL = 5;
        // Reversal because of communications failure
exec sql end declare section;

//-----

// METER IDENTIFIERS

exec sql begin declare section;
    static const short METER_ID_MONEY_IN = 0;
        // Money transferred in
    static const short METER_ID_MONEY_OUT = 1;
        // Money transferred out
    static const short METER_ID_BILLS_IN = 2;
        // Bills inserted
    static const short METER_ID_SALES = 3;
        // Sales
exec sql end declare section;

//-----

// PROCESS SUPER-CLASS

class Proc_c {
    // Class Instance Variables

    Dtime_t      wakeupTime;    // Next wakeup time

    // Public Methods
public:
    void SleepUntil (Dtime_t wakeupTime);
        // Sleep until a specified time
    Dtime_t GetWakeupTime () const { return wakeupTime; }
        // Get this process's wakeup time
    virtual void Wakeup () = 0;
        // Wakeup after a sleep
    void WaitUntilFree (size_t vendNo);
        // Wait until a vending machine is free
    virtual void VendIsFree () = 0;
        // Vending machine is free
};

//-----

// COMPARE PROCESS POINTERS

class CompProc_c {
public:
    bool
    operator () (
        const Proc_c *p1,          // First process
        const Proc_c *p2)          // Second process
    {
        return p1->GetWakeupTime() < p2->GetWakeupTime() ||
            (p1->GetWakeupTime() == p2->GetWakeupTime() &&
             p1 < p2);
    }
};

```

```
//-----  
  
// SCHEDULER QUEUE  
  
typedef set<Proc_c*,CompProc_c> SchQue_t;  
                                // Scheduler queue  
typedef SchQue_t::iterator SchQueIter_t;  
                                // Scheduler queue iterator  
  
//-----  
  
// WAIT LIST  
  
typedef list<Proc_c*> WaitList_t;  
                                // Waiting process list  
typedef WaitList_t::iterator WaitIter_t;  
                                // Waiting process iterator  
  
//-----  
  
// CUSTOMER EMULATOR  
  
class Cust_c : public Proc_c {  
  
    // Customer States  
  
    enum CustState_t {  
        CUST_UNINITIATED,    // Uninitiated  
        CUST_BETWEEN_TX,    // Between transactions  
        CUST_QUEUED,        // Queued for vending machine  
        CUST_SPACE_PURCHASE, // Space clock-in to purchase  
        CUST_SPACE_BILL_IN, // Space bill-in to purchase  
        CUST_SPACE_CLOCK_OUT, // Space purchase to clock-out  
        CUST_SPACE_RELEASE, // Space clock-out to release  
        CUST_FINISHED       // Emulation finished  
    };  
  
    // Private Variables  
  
    long        custId;        // Customer identifier  
    CustState_t custState;    // Customer state  
    double      custBal;      // Current account balance  
    double      purchaseVal;  // Purchase value  
    size_t      vendNo;       // Vending machine number  
  
    // Private Methods  
  
    Dtime_t NextCustTxTime (); // Determine the next transaction time  
    void EmulateCardIn ();    // Emulate a card-in operation  
    void EmulateBillIn ();    // Emulate a bill insertion operation  
    void EmulateSale ();     // Emulate a sale  
    void EmulateCommsFault (); // Emulate a communications fault  
    void EmulateCollectCard (); // Emulate a collect card operation  
  
    // Public Methods  
public:  
    Cust_c () { custState = CUST_UNINITIATED; }
```

VendRep.cpp

```

        // Constructor
void CustInit (long custId);
        // Initiate the customer emulator
void Wakeup ();
        // Wakeup after a sleep
void VendIsFree ();
        // Vending machine is free
};

//-----

// ATTENDANT EMULATOR

class Attend_c : public Proc_c {

    // Attendant States

    enum AttendState_t {
        ATTEND_UNINITIATED,    // Uninitiated
        ATTEND_BETWEEN_TX,    // Between transactions
        ATTEND_QUEUED,        // Queued for vending machine
        ATTEND_SPACE_RELEASE, // Space clear to release
        ATTEND_FINISHED       // Emulation finished
    };

    // Class Instance Variables

    size_t      vendNo;        // Vending machine number
    AttendState_t  attendState; // Attendant state

    // Private Methods

    Dtime_t NextAttendTxTime ();
        // Determine the next transaction time
    void EmulateClearMach ();
        // Clear the vending machine

    // Public Methods
public:
    Attend_c () { attendState = ATTEND_UNINITIATED; }
        // Constructor
    void AttendInit (size_t vendNo);
        // Initiate the attendant emulator
    void Wakeup ();
        // Wakeup after a sleep
    void VendIsFree ();
        // Vending machine is free
};

//-----

// VENDING MACHINE DATA STRUCTURE

struct Vend_t {
    long      machId;        // Machine identifier
    Dtime_t   machDate;     // Machine date
    double    moneyInMeter;  // Money transferred in meter
    double    moneyOutMeter; // Money transferred out meter
    double    billsInMeter; // Bills in meter
    double    salesMeter;   // Sales meter
    bool      isInUse;      // Vending machine is in use flag
    WaitList_t  waitList;   // Waiting process list
};

```

```
//-----  
  
// GLOBAL DATA STRUCTURES  
  
Dtime_t      virtTime;           // Current virtual time  
Dtime_t      endTime;           // End of simulation time  
long         nextTxNo;           // Next transaction number  
SchQue_t     schQue;             // Scheduler queue  
Cust_c       custArr[CUST_CNT];  // Customer emulators  
Attend_c     attendArr[VEND_CNT]; // Attendant emulators  
Vend_t       vendArr[VEND_CNT];  // Vending machine data  
  
//-----  
  
// SLEEP UNTIL A PARTICULAR TIME  
  
void  
Proc_c::SleepUntil (   
    Dtime_t      xWakeupTime)    // Wakeup time  
{  
    wakeupTime = xWakeupTime;  
    schQue.insert (this);  
}  
  
//-----  
  
// WAIT UNTIL A VENDING MACHINE IS FREE  
  
void  
Proc_c::WaitUntilFree (   
    size_t      vendNo)          // Vending machine number  
{  
    vendArr[vendNo].waitList.push_back (this);  
}  
  
//-----  
  
// DETERMINE THE NEXT CUSTOMER TRANSACTION TIME  
  
Dtime_t  
Cust_c::NextCustTxTime ()  
{  
    double      r;                // Random double  
    Dtime_t     t;                // Time  
    long        secRem;           // Seconds remaining  
    long        skipSecs;         // Skip seconds  
    int         yr, mo, dy;       // Date components  
    int         hr, mi, se;       // Time components  
  
    // Select a transaction time between 8am and 8pm  
  
    r = rand() / (static_cast<double>(RAND_MAX) + 1.0);  
    if (r >= 1.0/6.0) {  
        r = rand() / (static_cast<double>(RAND_MAX) + 1.0);  
        secRem = static_cast<long>(r * 24*60*60);  
        t = virtTime;  
        UnpackDtime (t, &yr, &mo, &dy, &hr, &mi, &se);  
        if (hr < 8)  
            skipSecs = 8*60*60 - ((hr*60 + mi)*60 + se);  
        else if (hr >= 20)  
            skipSecs = (24+8)*60*60 - ((hr*60 + mi)*60 + se);  
        else
```

VendRep.cpp

```

        skipSecs = 0;
    t += skipSecs;
    while (secRem != 0) {
        UnpackDtime (t, &yr, &mo, &dy, &hr, &mi, &se);
        if (secRem < 20*60*60 - ((hr*60 + mi)*60 + se)) {
            t += secRem;
            secRem = 0;
        } else {
            t += (20+12)*60*60 - ((hr*60 + mi)*60 + se);
            secRem -= 20*60*60 - ((hr*60 + mi)*60 + se);
        }
    }
}

// Select a transaction time between 8am and 8pm

else {
    r = rand() / (static_cast<double>(RAND_MAX) + 1.0);
    secRem = static_cast<long>(r * 24*60*60);
    t = virtTime;
    UnpackDtime (t, &yr, &mo, &dy, &hr, &mi, &se);
    if (hr >= 8 && hr <= 20)
        skipSecs = 20*60*60 - ((hr*60 + mi)*60 + se);
    else
        skipSecs = 0;
    t += skipSecs;
    while (secRem != 0) {
        UnpackDtime (t, &yr, &mo, &dy, &hr, &mi, &se);
        if (hr < 8) {
            if (secRem < 8*60*60 - ((hr*60+mi)*60+se)) {
                t += secRem;
                secRem = 0;
            } else {
                t += (8+12)*60*60 - ((hr*60+mi)*60+se);
                secRem -= 8*60*60 - ((hr*60+mi)*60+se);
            }
        } else if (hr >= 20) {
            if (secRem < 24*60*60 - ((hr*60+mi)*60+se)) {
                t += secRem;
                secRem = 0;
            } else {
                t += 24*60*60 - ((hr*60+mi)*60+se);
                secRem -= 24*60*60 -
((hr*60+mi)*60+se);
            }
        } else {
            cerr << "Error: bad time\n";
            exit (1);
        }
    }
}
return t;
}

//-----

// EMULATE A CARD-IN OPERATION

void
Cust_c::EmulateCardIn ()
{
    exec sql begin declare section;
        long                cardInTxNo;        // Transaction number

```

VendRep.cpp

```

        long          cardInCustId;    // Customer identifier
        char          cardInCustDate[10+1]; // Customer date
        long          cardInMachId;    // Vending machine id
        char          cardInMachDate[10+1]; // Machine date
        char          cardInTxTime[26+1]; // Transaction time
        double        cardInTxValue;   // Transaction value
exec sql end declare section;

// Jump to DbError whenever an SQL error occurs

exec sql whenever sqlerror goto DbError;

// Load SQL variables

cardInTxNo = nextTxNo ++ ;
cardInCustId = custId;
SqlDateFromDtime (cardInCustDate, virtTime);
cardInMachId = vendArr[vendNo].machId;
SqlDateFromDtime (cardInMachDate, vendArr[vendNo].machDate);
SqlTimestampFromDtime (cardInTxTime, virtTime);
cardInTxValue = custBal;

// Insert the customer transaction row

exec sql insert into custTx (
        custId, custDate, txNo
) values (
        :cardInCustId, :cardInCustDate, :cardInTxNo
);

// Insert the vending machine transaction row

exec sql insert into vendTx (
        machId, machDate, txNo
) values (
        :cardInMachId, :cardInMachDate, :cardInTxNo
);

// Insert the transaction data row

exec sql insert into txData (
        txNo, txTime, txType, txValue
) values (
        :cardInTxNo, :cardInTxTime, :TX_TYPE_CARD_IN, :cardInTxValue
);

// Increment the meters on the vending machine

vendArr[vendNo].moneyInMeter += cardInTxValue;
return;

// Process a database error
DbError:
    cerr << "Error(" << __LINE__ << "): SQLCODE=" << SQLCODE << '\n';
    exit (1);
}

//-----
// EMULATE A BILL-IN OPERATION

void
Cust_c::EmulateBillIn ()

```

VendRep.cpp

```
{
    exec sql begin declare section;
        long          billInTxNo;      // Transaction number
        long          billInCustId;    // Customer identifier
        char          billInCustDate[10+1]; // Customer date
        long          billInMachId;    // Vending machine id
        char          billInMachDate[10+1]; // Machine date
        char          billInTxTime[26+1]; // Transaction time
        double        billInTxValue;   // Transaction value
    exec sql end declare section;

    // Jump to DbError whenever an SQL error occurs

    exec sql whenever sqlerror goto DbError;

    // Load SQL variables

    billInTxNo = nextTxNo ++ ;
    billInCustId = custId;
    SqlDateFromDtime (billInCustDate, virtTime);
    billInMachId = vendArr[vendNo].machId;
    SqlDateFromDtime (billInMachDate, vendArr[vendNo].machDate);
    SqlTimestampFromDtime (billInTxTime, virtTime);

    // Calculate the transaction value

    billInTxValue = purchaseVal - custBal
        + rand() % (10001 - static_cast<int>(purchaseVal - custBal));

    // Insert the customer transaction row

    exec sql insert into custTx (
        custId, custDate, txNo
    ) values (
        :billInCustId, :billInCustDate, :billInTxNo
    );

    // Insert the vending machine transaction row

    exec sql insert into vendTx (
        machId, machDate, txNo
    ) values (
        :billInMachId, :billInMachDate, :billInTxNo
    );

    // Insert the transaction data row

    exec sql insert into txData (
        txNo, txTime, txType, txValue
    ) values (
        :billInTxNo, :billInTxTime, :TX_TYPE_BILL_IN, :billInTxValue
    );

    // Increment the meters on the vending machine

    vendArr[vendNo].billsInMeter += billInTxValue;

    // Increase the customer balance

    custBal += billInTxValue;
    return;

    // Process a database error
```

```
DbError:
    cerr << "Error(" << __LINE__ << "): SQLCODE=" << SQLCODE << '\n';
    exit (1);
}

//-----

// EMULATE A SALE OPERATION

void
Cust_c::EmulateSale ()
{
    exec sql begin declare section;
        long          saleTxNo;          // Transaction number
        long          saleCustId;        // Customer identifier
        char          saleCustDate[10+1]; // Customer date
        long          saleMachId;        // Vending machine id
        char          saleMachDate[10+1]; // Machine date
        char          saleTxTime[26+1];  // Transaction time
        double        saleTxValue;      // Transaction value
    exec sql end declare section;

    // Jump to DbError whenever an SQL error occurs

    exec sql whenever sqlerror goto DbError;

    // Load SQL variables

    saleTxNo = nextTxNo ++ ;
    saleCustId = custId;
    SqlDateFromDtime (saleCustDate, virtTime);
    saleMachId = vendArr[vendNo].machId;
    SqlDateFromDtime (saleMachDate, vendArr[vendNo].machDate);
    SqlTimestampFromDtime (saleTxTime, virtTime);
    saleTxValue = purchaseVal;

    // Insert the customer transaction row

    exec sql insert into custTx (
        custId, custDate, txNo
    ) values (
        :saleCustId, :saleCustDate, :saleTxNo
    );

    // Insert the vending machine transaction row

    exec sql insert into vendTx (
        machId, machDate, txNo
    ) values (
        :saleMachId, :saleMachDate, :saleTxNo
    );

    // Insert the transaction data row

    exec sql insert into txData (
        txNo, txTime, txType, txValue
    ) values (
        :saleTxNo, :saleTxTime, :TX_TYPE_SALE, :saleTxValue
    );

    // Increment the meters on the vending machine

    vendArr[vendNo].salesMeter += saleTxValue;
}
```

```
        // Increase the customer balance

        custBal -= saleTxValue;
        return;

        // Process a database error
DbError:
    cerr << "Error(" << __LINE__ << "): SQLCODE=" << SQLCODE << '\n';
    exit (1);
}

//-----

// EMULATE A COMMUNICATIONS FAULT

void
Cust_c::EmulateCommsFault ()
{
    exec sql begin declare section;
        long          comFltTxNo;      // Transaction number
        long          comFltCustId;    // Customer identifier
        char          comFltCustDate[10+1]; // Customer date
        long          comFltMachId;    // Vending machine id
        char          comFltMachDate[10+1]; // Machine date
        char          comFltTxTime[26+1]; // Transaction time
        double        comFltTxValue;  // Transaction value
    exec sql end declare section;

    // Jump to DbError whenever an SQL error occurs

    exec sql whenever sqlerror goto DbError;

    // Load SQL variables

    comFltTxNo = nextTxNo ++ ;
    comFltCustId = custId;
    SqlDateFromDtime (comFltCustDate, virtTime);
    comFltMachId = vendArr[vendNo].machId;
    SqlDateFromDtime (comFltMachDate, vendArr[vendNo].machDate);
    SqlTimestampFromDtime (comFltTxTime, virtTime);
    comFltTxValue = custBal;

    // Insert the vending machine transaction row

    exec sql insert into vendTx (
        machId, machDate, txNo
    ) values (
        :comFltMachId, :comFltMachDate, :comFltTxNo
    );

    // Insert the transaction data row

    exec sql insert into txData (
        txNo, txTime, txType, txValue
    ) values (
        :comFltTxNo, :comFltTxTime, :TX_TYPE_REVERSAL, :comFltTxValue
    );

    // Increment the meters on the vending machine

    vendArr[vendNo].moneyOutMeter += comFltTxValue;
    vendArr[vendNo].moneyInMeter += comFltTxValue;
```

```
        return;

        // Process a database error
DbError:
    cerr << "Error(" << __LINE__ << "): SQLCODE=" << SQLCODE << '\n';
    exit (1);
}

//-----

// EMULATE A COLLECT CARD OPERATION

void
Cust_c::EmulateCollectCard ()
{
    exec sql begin declare section;
        long          collectTxNo;    // Transaction number
        long          collectCustId;  // Customer identifier
        char          collectCustDate[10+1]; // Customer date
        long          collectMachId;  // Vending machine id
        char          collectMachDate[10+1]; // Machine date
        char          collectTxTime[26+1]; // Transaction time
        double        collectTxValue; // Transaction value
    exec sql end declare section;

    // Jump to DbError whenever an SQL error occurs

    exec sql whenever sqlerror goto DbError;

    // Load SQL variables

    collectTxNo = nextTxNo ++ ;
    collectCustId = custId;
    SqlDateFromDtime (collectCustDate, virtTime);
    collectMachId = vendArr[vendNo].machId;
    SqlDateFromDtime (collectMachDate, vendArr[vendNo].machDate);
    SqlTimestampFromDtime (collectTxTime, virtTime);
    collectTxValue = custBal;

    // Insert the customer transaction row

    exec sql insert into custTx (
        custId, custDate, txNo
    ) values (
        :collectCustId, :collectCustDate, :collectTxNo
    );

    // Insert the vending machine transaction row

    exec sql insert into vendTx (
        machId, machDate, txNo
    ) values (
        :collectMachId, :collectMachDate, :collectTxNo
    );

    // Insert the transaction data row

    exec sql insert into txData (
        txNo, txTime, txType,
        txValue
    ) values (
        :collectTxNo, :collectTxTime, :TX_TYPE_COLLECT_CARD,
        :collectTxValue
    );
}
```

```
);

// Increment the meters on the vending machine
vendArr[vendNo].moneyOutMeter += collectTxValue;
return;

// Process a database error
DbError:
cerr << "Error(" << __LINE__ << "): SQLCODE=" << SQLCODE << '\n';
exit (1);
}

//-----

// INITIALISE THE CUSTOMER EMULATOR

void
Cust_c::CustInit (
    long         xCustId)          // Customer identifier
{
    custId = xCustId;
    custBal = 1000.0 + rand() % (10000 - 1000 + 1);
    custState = CUST_BETWEEN_TX;
    SleepUntil (NextCustTxTime());
}

//-----

// PROCESS A CUSTOMER WAKEUP STIMULUS

void
Cust_c::Wakeup ()
{
    Dtime_t      txTime;          // Transaction time

    // Process the stimulus according to the current state
    switch (custState) {

    // Waiting for the next transaction

    case CUST_BETWEEN_TX:

        // Randomly select a vending machine from among the
        // alternatives

        vendNo = rand() % VEND_CNT;

        // If the vending machine is in use, wait until the
        // vending machine is free

        if (vendArr[vendNo].isInUse) {
            custState = CUST_QUEUED;
            WaitUntilFree (vendNo);
        }

        // If the vending machine is immediately available
        // emulate a card insertion

    else {
        vendArr[vendNo].isInUse = 1;
        EmulateCardIn ();
    }
}
}
```

```
        custState = CUST_SPACE_PURCHASE;
        SleepUntil (virtTime + 5 + rand() % 10);
    }
    break;

// Process wakeup after spacing clock-in to purchase
case CUST_SPACE_PURCHASE:

    // Choose a purchase value

    purchaseVal = 200 + rand() % (2000 - 200 + 1);

    // Put more money in the vending machine if needed

    if (custBal < purchaseVal) {
        EmulateBillIn ();
        custState = CUST_SPACE_BILL_IN;
        SleepUntil (virtTime + 5 + rand() % 10);
    }

    // Emulate the sale

    else {
        EmulateSale ();
        custState = CUST_SPACE_CLOCK_OUT;
        SleepUntil (virtTime + 5 + rand() % 10);
    }
    break;

// Space bill-in to purchase
case CUST_SPACE_BILL_IN:

    // Emulate the sale

    EmulateSale ();
    custState = CUST_SPACE_CLOCK_OUT;
    SleepUntil (virtTime + 5 + rand() % 10);
    break;

// Space purchase to clock-out
case CUST_SPACE_CLOCK_OUT:

    // Emulate a communications fault

    if (rand() % 200 < 1) {
        EmulateCommsFault ();
        custState = CUST_SPACE_CLOCK_OUT;
        SleepUntil (virtTime + 5 + rand() % 10);
    }

    // Emulate a successful collect card operation

    else {
        EmulateCollectCard ();
        custState = CUST_SPACE_RELEASE;
        SleepUntil (virtTime + 5 + rand() % 10);
    }
    break;

// Space clock-out to release
```

```
    case CUST_SPACE_RELEASE:
        // Release the vending machine
        vendArr[vendNo].isInUse = 0;

        // Select the next transaction time

        txTime = NextCustTxTime();
        if (txTime < endTime) {
            custState = CUST_BETWEEN_TX;
            SleepUntil (txTime);
        } else {
            custState = CUST_FINISHED;
        }
        break;

    // Other states are weird

default:
    cerr << "Error(" << __LINE__ << "): bad state "
         << custState << '\n';
    exit (1);
}

//-----

// PROCESS VENDING MACHINE IS FREE EVENT

void
Cust_c::VendIsFree ()
{
    // Process the stimulus according to the current state

    switch (custState) {

    // Waiting for the vending machine to become free

    case CUST_QUEUED:

        // Emulate a card insertion

        vendArr[vendNo].isInUse = 1;
        EmulateCardIn ();
        custState = CUST_SPACE_PURCHASE;
        SleepUntil (virtTime + 5 + rand() % 10);
        break;

    default:
        cerr << "Error(" << __LINE__ << "): bad state "
             << custState << '\n';
        exit (1);
    }
}

//-----

// DETERMINE THE NEXT ATTENDANT TRANSACTION TIME

Dtime_t
Attend_c::NextAttendTxTime ()
```

VendRep.cpp

```

{
    double          r;                // A random value
    Dtime_t         t;                // Next transaction time
    int             yr, mo, dy;       // Date components
    int             hr, mi, se;       // Time components

    UnpackDtime (virtTime+24*60*60, &yr, &mo, &dy, &hr, &mi, &se);
    r = rand() / (static_cast<double>(RAND_MAX) + 1.0);
    t = PackDtime (yr, mo, dy, 5, 0, 0) +
        static_cast<long>(r*2*60*60);
    return t;
}

//-----

// EMULATE CLEARING THE MACHINE

void
Attend_c::EmulateClearMach ()
{
    exec sql begin declare section;
        long          clearMachId;    // Machine identifier
        char          clearMachDate[10+1]; // Machine date
        double        meteredValue;   // Metered value
    exec sql end declare section;

    // Jump to DbError whenever an SQL error occurs

    exec sql whenever sqlerror goto DbError;

    // Load SQL variables

    clearMachId = vendArr[vendNo].machId;
    SqlDateFromDtime (clearMachDate, vendArr[vendNo].machDate);

    // Insert the meter reading rows

    meteredValue = vendArr[vendNo].moneyInMeter;
    exec sql insert into vendMeters (
        machId, machDate, meterId, meteredValue
    ) values (
        :clearMachId, :clearMachDate, :METER_ID_MONEY_IN, :meteredValue
    );
    meteredValue = vendArr[vendNo].moneyOutMeter;
    exec sql insert into vendMeters (
        machId, machDate, meterId, meteredValue
    ) values (
        :clearMachId, :clearMachDate, :METER_ID_MONEY_OUT,
:meteredValue
    );
    meteredValue = vendArr[vendNo].billsInMeter;
    exec sql insert into vendMeters (
        machId, machDate, meterId, meteredValue
    ) values (
        :clearMachId, :clearMachDate, :METER_ID_BILLS_IN, :meteredValue
    );
    meteredValue = vendArr[vendNo].salesMeter;
    exec sql insert into vendMeters (
        machId, machDate, meterId, meteredValue
    ) values (
        :clearMachId, :clearMachDate, :METER_ID_SALES, :meteredValue
    );
};

```

VendRep.cpp

```

// Execute a checkpoint on a daily basis

if (vendNo == 0)
    exec sql commit work;

// Reset the meters

vendArr[vendNo].moneyInMeter = 0;
vendArr[vendNo].moneyOutMeter = 0;
vendArr[vendNo].billsInMeter = 0;
vendArr[vendNo].salesMeter = 0;

// Update the machine date

vendArr[vendNo].machDate = virtTime;
return;

// Process a database error
DbError:
cerr << "Error(" << __LINE__ << "): SQLCODE=" << SQLCODE << '\n';
exit (1);
}

//-----

// INITIALISE THE ATTENDANT EMULATOR

void
Attend_c::AttendInit (
    size_t          xVendNo)          // Vending machine number
{
    vendNo = xVendNo;
    attendState = ATTEND_BETWEEN_TX;
    SleepUntil (NextAttendTxTime());
}

//-----

// PROCESS AN ATTENDANT WAKEUP STIMULUS

void
Attend_c::Wakeup ()
{
    Dtime_t          txTime;          // Transaction time

    // Process the stimulus according to the current state

    switch (attendState) {

        // Waiting for the next transaction

        case ATTEND_BETWEEN_TX:

            // If the vending machine is in use, wait until the
            // vending machine is free

            if (vendArr[vendNo].isInUse) {
                attendState = ATTEND_QUEUED;
                WaitUntilFree (vendNo);
            }

            // If the vending machine is immediately available
            // emulate clearing of the machine

```

```
        else {
            vendArr[vendNo].isInUse = 1;
            EmulateClearMach ();
            attendState = ATTEND_SPACE_RELEASE;
            SleepUntil (virtTime + 5 + rand() % 10);
        }
        break;

// Space cleared to release

case ATTEND_SPACE_RELEASE:

    // Release the vending machine

    vendArr[vendNo].isInUse = 0;

    // Select the next transaction time

    txTime = NextAttendTxTime();
    if (txTime < endTime) {
        attendState = ATTEND_BETWEEN_TX;
        SleepUntil (txTime);
    } else {
        attendState = ATTEND_FINISHED;
    }
    break;

// Other states are weird

default:
    cerr << "Error(" << __LINE__ << "): bad state "
        << attendState << '\n';
    exit (1);
}

}

//-----

// PROCESS VENDING MACHINE IS FREE EVENT

void
Attend_c::VendIsFree ()
{
    // Process the stimulus according to the current state

    switch (attendState) {

// Waiting for the vending machine to become free

case ATTEND_QUEUED:

        // Emulate clearing the machine

        vendArr[vendNo].isInUse = 1;
        EmulateClearMach ();
        attendState = ATTEND_SPACE_RELEASE;
        SleepUntil (virtTime + 5 + rand() % 10);
        break;

default:
        cerr << "Error(" << __LINE__ << "): bad state "
            << attendState << '\n';
    }
}
```

```
        exit (1);
    }
}

//-----

// MAIN LINE

int
main ()
{
    size_t          i;          // General purpose index
    Proc_c          *proc;      // Process pointer

    exec sql begin declare section;
    exec sql end declare section;

    // Connect to the database

    exec sql connect to venddb;

    // Drop tables

    exec sql whenever sqlerror continue;
    exec sql drop table custTx;
    exec sql drop table vendTx;
    exec sql drop table txData;
    exec sql drop table vendMeters;
    exec sql commit work;

    // Jump to DbError whenever an SQL error occurs

    exec sql whenever sqlerror goto DbError;

    // Create the database tables

    exec sql create table custTx (
        custId          integer not null,
        custDate        date not null,
        txNo            integer not null
    );
    exec sql create table vendTx (
        machId          integer not null,
        machDate        date not null,
        txNo            integer not null
    );
    exec sql create table txData (
        txNo            integer not null,
        txTime          timestamp not null,
        txType          smallint not null,
        txValue         double precision not null
    );
    exec sql create table vendMeters (
        machId          integer not null,
        machDate        date not null,
        meterId         smallint not null,
        meteredValue    double precision not null
    );

    // Initialise the random number generator

    srand (20360L);
}
```

```
// Initialise the current virtual time

virtTime = DtimeFromSql (FROM_DATE);
endTime = DtimeFromSql (TO_DATE) + 24*60*60;
nextTxNo = 0;

// Initialise the vending machines

for (i = 0; i < VEND_CNT; i++) {
    vendArr[i].machId = i + 2000;
    vendArr[i].machDate = DtimeFromSql (FROM_DATE);
    vendArr[i].moneyInMeter = 0;
    vendArr[i].moneyOutMeter = 0;
    vendArr[i].billsInMeter = 0;
    vendArr[i].salesMeter = 0;
    vendArr[i].isInUse = 0;
}

// Initialise the customers

for (i = 0; i < CUST_CNT; i++)
    custArr[i].CustInit (i + 1000);

// Initialise the attendants

for (i = 0; i < VEND_CNT; i++)
    attendArr[i].AttendInit (i);

// Continue processing until there is nothing left to do

for (;;) {

    // Check whether any of the vending machines are free
    // to process the next item in their wait lists

    do {
        i = 0;
        while (
            i < VEND_CNT && (
                vendArr[i].isInUse ||
                vendArr[i].waitList.begin() ==
                vendArr[i].waitList.end()
            )
        ) i ++ ;
        if (i < VEND_CNT) {
            vendArr[i].waitList.front()->VendIsFree ();
            vendArr[i].waitList.pop_front();
        }
    } while (i < VEND_CNT);

    // If the scheduler queue is empty, that's all

    if (schQue.begin() == schQue.end()) break;

    // Drop the scheduler queue entry

    proc = *schQue.begin();
    schQue.erase (schQue.begin());

    // Advance the virtual time to the wakeup time of the process
    // and wake up the process

    virtTime = proc->GetWakeupTime ();
}
```

```
        proc->Wakeup ();
    }

    // Commit the database changes
    exec sql commit work;

    // And that's all
    return 0;

    // Process database errors
DbError:
    cerr << "Error(" << __LINE__ << "): SQLCODE=" << SQLCODE << '\n';
    return 1;
}
```