```
// OfferTest.cpp - OFFER SERVER TESTING MAIN LINE
//
// MODULE INDEX
// NAME                          CONTENTS
// OfferServer_c::Initiate       Initiate the offer server
// OfferServer_c::Tick           Process a clock tick
// OfferServer_c::ReplyReceived  Process a reply received from an atm
// OfferServer_c::AnswerReceived Process an answer received from a bcs
//
// MAINTENANCE HISTORY
// DATE       PROGRAMMER AND DETAILS
// 23-09-12 JS   Original
//
//-----------------------------------------------------------------------

#include <cstring>       // C-style string manipulation functions
#include <cctype>        // C-style character typing functions
#include <cstdlib>       // C-style standard library
#include <cmath>         // C-style Mathematical functions
#include <iostream>      // C++ input/output streams
#include <sstream>       // C++ string streams
#include <map>           // C++ map declarations
#include <list>          // C++ list declarations
#include <set>           // C++ set declarations
#include <vector>        // C++ vector declarations
#include <string>        // C++ string declarations
using namespace std;          // Expand the standard namespace
#include "OfferComms.h"      // Offer communications declarations
#include "OfferServer.h"     // Offer server declarations

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

// DEFINITIONS

#define ATM_CNT      200 // Number of ATMs
#define BCS_CNT      10  // Number of BCSs
#define CUSTOMER_CNT     1000      // Number of customers
#define OFFER_CODE_CNT   20   // Number of distinct offer codes
#define OFFER_PER_CUST   4    // Number of offers per customer
#define OFFER_CNT    (CUSTOMER_CNT*OFFER_PER_CUST)
                  // Total number of offers

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

// MESSAGE TYPE CODE

enum MessageType_t {
    BASIC_REPLY,              // Basic reply from an ATM
    RELAYED_REPLY,            // Relayed reply from the offer server
    BASIC_ANSWER,             // Basic answer from a BCS
    RELAYED_ANSWER            // Relayed answer from the offer server
};

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

// MESSAGE STRUCTURE
```

```
struct Message_t {
    MessageType_t   messageType;     // Message type
    long        bcsId;       // BCS identifier
    long        atmId;       // ATM identifier
    long        offerCode;   // Offer code
    long        customerId;  // Customer identifier
    bool        offerAccepted;   // Offer accepted flag
    bool        replyAccepted;   // Reply accepted flag
    string      reason;      // Reason for rejection
};

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

// MESSAGE QUEUE

typedef list<Message_t> MessageQueue_t;
                    // Message queue type definition
typedef MessageQueue_t::iterator MessageIter_t;
                    // Message iterator type definition

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

// WAKEUP TYPE

enum WakeupType_t {
    WAKEUP_ATM,          // Wake up an ATM
    WAKEUP_BCS,          // Wake up a BCS
    WAKEUP_SERVER        // Wake up the server
};

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

// WAKEUP REQUEST STRUCTURE

struct WakeupReq_t {
    time_t      wakeupTime; // Wakeup time
    WakeupType_t    wakeupType; // Wakeup type
    long        atmId;       // ATM identifier
    long        bcsId;       // BCS identifier

    // Default Constructor

    WakeupReq_t () {}

    // Comparator

    bool
    operator < (const WakeupReq_t &wakeupReq)
    const
    {
        if (wakeupTime < wakeupReq.wakeupTime) return 1;
        if (wakeupTime > wakeupReq.wakeupTime) return 0;
        if (wakeupType < wakeupReq.wakeupType) return 1;
        if (wakeupType > wakeupReq.wakeupType) return 0;
        if (wakeupType == WAKEUP_ATM) {
            if (atmId < wakeupReq.atmId) return 1;
            if (atmId > wakeupReq.atmId) return 0;
```

```
        } else if (wakeupType == WAKEUP_BCS) {
            if (bcsId < wakeupReq.bcsId) return 1;
            if (bcsId > wakeupReq.bcsId) return 0;
        }
        return 0;
    }
};

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

// WAKEUP QUEUE TYPE DEFINITIONS

typedef set<WakeupReq_t> WakeupQueue_t;
                        // Wakeup queue type definition
typedef WakeupQueue_t::iterator WakeupIter_t;
                        // Wakeup iterator type definition

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

// REMAINING OFFERS LIST TYPE DEFINITIONS

typedef list<Offer_t> RemainList_t;
                        // Remaining offers list type definition
typedef RemainList_t::iterator RemainIter_t;
                        // Remaining offers iterator type def

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

// ATM EMULATOR

class AtmEmulator_c {

    // ATM Emulator State

    enum AtmState_t {
        ATM_STATE_SLEEPING,      // Sleeping between replies
        ATM_STATE_RECEIVING,     // Receiving an answer
        ATM_STATE_FINISHED       // ATM is finished
    };

    // Offer Key Structure

    struct OfferKey_t {
        long         offerCode;  // Offer code
        long         customerId; // Customer identifier

        // Default Constructor

        OfferKey_t ()
        {
            // Empty
        }

        // Construct from Values

        OfferKey_t (long offerCodeParm, long customerIdParm)
        {
```

```
            offerCode = offerCodeParm;
            customerId = customerIdParm;
        }

        // Comparator

        bool
        operator < (const OfferKey_t &offerKey)
        const
        {
            return offerCode < offerKey.offerCode ||
                (offerCode == offerKey.offerCode &&
                    customerId < offerKey.customerId);
        }
    };

    // Offer Data Structure

    struct OfferData_t {
        bool        offerAccepted;   // Offer accepted flag
        bool        replyAccepted;   // Reply accepted flag
        string      reason;      // Reason for rejection
    };

    // Offer Map

    typedef map<OfferKey_t,OfferData_t> OfferMap_t;
                        // Offer map type definition
    typedef OfferMap_t::iterator OfferIter_t;
                        // Offer iterator type def

    // Private Variables

    AtmState_t  atmState;    // Current ATM state
    long        atmId;       // This ATM's ATM identifier
    WakeupReq_t wakeupReq;   // Wakeup request
    OfferMap_t  answeredOffers; // Answered offers
    OfferKey_t  currentKey; // Key to current offer
    bool        offerAccepted;   // Offer accepted flag

    // Private Methods

    void SelectOffer ();
                    // Select a free offer

    // Public Methods
public:
    void Initiate (long atmId);
                    // Initiate the ATM
    void Wakeup ();
                    // Wake up the ATM
    void AnswerReceived (Message_t *message);
                    // Process a received answer
};

//---------------------------------------------------------------------------
```

```
// BCS EMULATOR

class BcsEmulator_c {

    // BCS Emulator State

    enum BcsState_t {
        BCS_STATE_RECEIVING,     // Receiving a reply
        BCS_STATE_SLEEPING       // Sleeping before answering
    };

    // Offer Key Structure

    struct OfferKey_t {
        long         offerCode;  // Offer code
        long         customerId; // Customer identifier

        // Default Constructor

        OfferKey_t ()
        {
            // Empty
        }

        // Construct from Values

        OfferKey_t (long offerCodeParm, long customerIdParm)
        {
            offerCode = offerCodeParm;
            customerId = customerIdParm;
        }

        // Comparator

        bool
        operator < (const OfferKey_t &offerKey)
        const
        {
            return offerCode < offerKey.offerCode ||
                (offerCode == offerKey.offerCode &&
                    customerId < offerKey.customerId);
        }
    };

    // Offer Data Structure

    struct OfferData_t {
        bool         offerAccepted;   // Offer accepted flag
        bool         replyAccepted;   // Reply accepted flag
        string       reason;          // Reason for rejection
    };

    // Offer Map

    typedef map<OfferKey_t,OfferData_t> OfferMap_t;
                            // Offer map type definition
    typedef OfferMap_t::iterator OfferIter_t;
```

```
                            // Offer iterator type def

    // Private Variables

    BcsState_t  bcsState;       // Current BCS state
    long        bcsId;          // This BCS's BCS identifier
    WakeupReq_t wakeupReq;      // Wakeup request
    OfferMap_t  answeredOffers; // Answered offers
    OfferKey_t  currentKey;     // Key to current offer
    bool        offerAccepted;  // Offer accepted flag

    // Public Methods
public:
    void Initiate (long bcsId);
                        // Initiate the BCS
    void Wakeup ();
                        // Wake up the BCS
    void ReplyReceived (Message_t *message);
                        // Process a received reply
};

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

// GLOBAL DATA

MessageQueue_t  messageQueue;   // Message queue
WakeupQueue_t   wakeupQueue;    // Wakeup queue
RemainList_t    remainList;     // Remaining offers list
size_t      activeAtmCnt;       // Number of active ATMs
time_t      currentTime;        // Current time

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

// OVERLOAD THE SYSTEM TIME FUNCTION

time_t
time (
    time_t      */*target*/)
{
    return currentTime;
}

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

// GENERATE THE OFFER ARRAY

void
GenerateOffers (
    Offer_t     *offerArr)  // Offer array
{
    size_t      offerCnt;   // Offer count
    size_t      i, j;       // General purpose indices
    set<long>   offerSet;   // Offer set
    long        offerCode;  // Offer code

    // Generate the offer array
```

```
    offerCnt = 0;
    for (i = 0; i < CUSTOMER_CNT; i++) {
        offerSet.clear ();
        for (j = 0; j < OFFER_PER_CUST; j++) {
            do {
                offerCode = lrand48() % OFFER_CODE_CNT;
            } while (offerSet.find(offerCode) != offerSet.end());
            offerSet.insert (offerCode);
            offerArr[offerCnt].offerCode = offerCode;
            offerArr[offerCnt].customerId = i;
            offerArr[offerCnt].bcsId = lrand48() % BCS_CNT;
            offerCnt ++ ;
        }
    }
    if (offerCnt != OFFER_CNT) {
        cerr << "Error: offerCnt != OFFER_CNT\n";
        abort ();
    }
}

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

// SELECT A FREE OFFER

void
AtmEmulator_c::SelectOffer ()
{
    size_t      i, j;        // General purpose indices
    RemainIter_t    remainIter; // Remaining offers iterator
    time_t      sleepTime;   // Amount of sleep time

    // If there are no more offers to process, return in the finished state

    if (remainList.size() == 0) {
        atmState = ATM_STATE_FINISHED;
        activeAtmCnt -- ;
    }

    // Process offers available

    else {

        // Randomly select an offer

        i = lrand48() % remainList.size();
        remainIter = remainList.begin();
        for (j = 0; j < i; j++) remainIter ++ ;
        currentKey.offerCode = remainIter->offerCode;
        currentKey.customerId = remainIter->customerId;
        offerAccepted = lrand48() % 2;
        remainList.erase (remainIter);

        // Sleep for 60 to 600s

        sleepTime = lrand48() % (600 - 60 + 1) + 60;
        wakeupReq.wakeupTime = currentTime + sleepTime;
        wakeupReq.wakeupType = WAKEUP_ATM;
```

```
        wakeupReq.atmId = atmId;
        wakeupQueue.insert (wakeupReq);
        atmState = ATM_STATE_SLEEPING;
    }
}

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

// INITIATE AN ATM EMULATOR

void
AtmEmulator_c::Initiate (
    long        atmIdParm)   // ATM identifier
{
    // Initialise the class instance variables

    atmId = atmIdParm;
    answeredOffers.clear ();

    // Select the next offer

    SelectOffer ();
}

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

// PROCESS AN ATM EMULATOR WAKEUP

void
AtmEmulator_c::Wakeup ()
{
    Message_t   message;     // Message structure
    time_t      sleepTime;   // Amount of sleep time

    // Process the wakeup stimulus depending on the current state

    switch (atmState) {

    // Sleeping between offers and receiving a reply

    case ATM_STATE_SLEEPING:
    case ATM_STATE_RECEIVING:

        // Append the reply message to the message queue

        message.messageType = BASIC_REPLY;
        message.atmId = atmId;
        message.offerCode = currentKey.offerCode;
        message.customerId = currentKey.customerId;
        message.offerAccepted = offerAccepted;
        messageQueue.push_back (message);

        // Schedule a wakeup at the retransmission time

        sleepTime = lrand48() % (45 - 15 + 1) + 15;
        wakeupReq.wakeupTime = currentTime + sleepTime;
        wakeupReq.wakeupType = WAKEUP_ATM;
```

```
        wakeupReq.atmId = atmId;
        wakeupQueue.insert (wakeupReq);
        atmState = ATM_STATE_RECEIVING;
        break;

    // Other states are weird

    default:
        cerr << __FILE__ << '(' << __LINE__
            << ") bad state " << atmState << '\n';
        abort ();
        // NOTREACHED
    }
}

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

// PROCESS RECEIPT OF AN ANSWER BY AN ATM

void
AtmEmulator_c::AnswerReceived (
    Message_t   *message)    // Received message
{
    OfferKey_t  offerKey;    // Offer key
    OfferData_t offerData;   // Offer data
    OfferIter_t offerIter;   // Offer iterator

    // Load the offer key

    offerKey.offerCode = message->offerCode;
    offerKey.customerId = message->customerId;

    // Process messages received in sequence

    if (
        atmState == ATM_STATE_RECEIVING &&
        message->offerCode == currentKey.offerCode &&
        message->customerId == currentKey.customerId
    ) {
        // Save the answer in the answered offers map

        offerData.offerAccepted = offerAccepted;
        offerData.replyAccepted = message->replyAccepted;
        offerData.reason = message->reason;
        answeredOffers[offerKey] = offerData;

        // Cancel the wakeup request

        wakeupQueue.erase (wakeupReq);

        // Select another offer

        SelectOffer ();
    }

    // Process messages received out-of-sequence
```

```
    else {
        // Check that the new answer is the same as the old

        offerIter = answeredOffers.find (offerKey);
        if (offerIter == answeredOffers.end()) {
            cerr << __FILE__ << '(' << __LINE__
                << ") unrecognised offer\n";
            abort ();
        }
        if (offerIter->second.replyAccepted != message->replyAccepted) {
            cerr << __FILE__ << '(' << __LINE__
                << ") answer changed\n";
            abort ();
        }
        if (
            offerIter->second.replyAccepted &&
            offerIter->second.reason != message->reason
        ) {
            cerr << __FILE__ << '(' << __LINE__
                << ") reason changed\n";
            abort ();
        }
    }
}

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

// INITIATE A BCS EMULATOR

void
BcsEmulator_c::Initiate (
    long        bcsIdParm)  // BCS identifier
{
    // Initialise the class instance variables

    bcsId = bcsIdParm;
    answeredOffers.clear ();

    // Wait for a reply

    bcsState = BCS_STATE_RECEIVING;
}

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

// WAKEUP A BCS EMULATOR

void
BcsEmulator_c::Wakeup ()
{
    Message_t   message;    // Message structure

    // Validate the state

    if (bcsState != BCS_STATE_SLEEPING) {
        cerr << __FILE__ << '(' << __LINE__
            << ") bad state " << bcsState << '\n';
```

```
        abort ();
    }

    // Send the answer

    message.messageType = BASIC_ANSWER;
    message.bcsId = bcsId;
    message.offerCode = currentKey.offerCode;
    message.customerId = currentKey.customerId;
    message.offerAccepted = answeredOffers[currentKey].offerAccepted;
    messageQueue.push_back (message);

    // Update the state

    bcsState = BCS_STATE_RECEIVING;
}

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

// PROCESS RECEIPT OF A REPLY BY A BCS

void
BcsEmulator_c::ReplyReceived (
    Message_t   *message)    // Received message
{
    OfferIter_t offerIter;   // Offer iterator
    OfferData_t offerData;   // Offer data
    ostringstream   oss;     // Output string stream
    time_t      sleepTime;   // Amount of sleep time

    // Disregard replies received while processing other replies

    if (bcsState == BCS_STATE_SLEEPING) return;

    // Load the current offer key

    currentKey.offerCode = message->offerCode;
    currentKey.customerId = message->customerId;

    // If the reply has been received before, check that the reply
    // is the same

    offerIter = answeredOffers.find (currentKey);
    if (offerIter != answeredOffers.end()) {
        if (offerIter->second.offerAccepted != message->offerAccepted) {
            cerr << __FILE__ << '(' << __LINE__
                << ") reply changed"
                << " old=" << offerIter->second.offerAccepted
                << " new=" << message->offerAccepted
                << '\n';
            abort ();
        }
    }

    // If the reply has not been received before, generate a new answer

    else {
```

```
        offerData.offerAccepted = message->offerAccepted;
        offerData.replyAccepted = lrand48() % 2;
        if ( ! offerData.replyAccepted) {
            oss.str("");
            oss << "Rejection code " << lrand48();
            offerData.reason = oss.str();
        } else {
            offerData.reason = "";
        }
        answeredOffers[currentKey] = offerData;
    }

    // Sleep to emulate the processing time

    sleepTime = lrand48() % (50 - 5 + 1) + 5;
    wakeupReq.wakeupTime = currentTime + sleepTime;
    wakeupReq.wakeupType = WAKEUP_BCS;
    wakeupReq.bcsId = bcsId;
    wakeupQueue.insert (wakeupReq);
    bcsState = BCS_STATE_SLEEPING;
}

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

// SEND A REPLY TO A BUSINESS CLIENT'S SERVER

void
OfferComms_c::SendToBCS (
    long        bcsId,       // BCS identifier
    long        atmId,       // ATM identifier
    long        offerCode,   // Offer code
    long        customerId, // Customer identifier
    bool        offerAccepted)   // Offer accepted flag
{
    Message_t   message;     // Message structure

    // Append the message to the message queue

    message.messageType = RELAYED_REPLY;
    message.bcsId = bcsId;
    message.atmId = atmId;
    message.offerCode = offerCode;
    message.customerId = customerId;
    message.offerAccepted = offerAccepted;
    messageQueue.push_back (message);
}

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

// SEND AN ANSWER TO AN ATM

void
OfferComms_c::SendToATM (
    long        atmId,       // ATM identifier
    long        offerCode,   // Offer code,
    long        customerId, // Customer identifier
    bool        replyAccepted,   // Reply accepted flag
```

```
    const char  *reason)     // Reason reply was rejected
{
    Message_t   message;     // Message structure

    // Append the message to the message queue

    message.messageType = RELAYED_ANSWER;
    message.atmId = atmId;
    message.offerCode = offerCode;
    message.customerId = customerId;
    message.replyAccepted = replyAccepted;
    if ( ! replyAccepted) message.reason = reason;
    messageQueue.push_back (message);
}

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

// MAIN LINE

int
main ()
{
    Offer_t     offerArr[OFFER_CNT];     // Offer array
    OfferComms_c    offerComms;      // Offer communications
    OfferServer_c   offerServer;     // Offer server instance
    size_t      i, j;            // General purpose index
    AtmEmulator_c   atmArr[ATM_CNT];     // ATM emulators
    BcsEmulator_c   bcsArr[BCS_CNT];     // BCS emulators
    MessageIter_t   messageIter;         // Message iterator
    Message_t   message;         // Message structure
    WakeupIter_t    wakeupIter;      // Wakeup iterator
    WakeupReq_t wakeupReq;       // Wakeup request
    vector<WakeupReq_t> wakeupVec;   // Wakeup vector

    // Initialise the pseudo random number generator

    srand48 (2031960);

    // Generate the offer array

    GenerateOffers (offerArr);

    // Load the remaining offers

    for (i = 0; i < OFFER_CNT; i++)
        remainList.push_back (offerArr[i]);

    // Initialise the time

    currentTime = lrand48() % 12345678;

    // Load a request to wake up the server

    wakeupReq.wakeupType = WAKEUP_SERVER;
    wakeupReq.wakeupTime = currentTime + 1;
    wakeupQueue.insert (wakeupReq);
```

```
    // Initiate the offer server

    offerServer.Initiate (&offerComms, offerArr, OFFER_CNT);

    // Initiate the ATMs and BCSs

    activeAtmCnt = ATM_CNT;
    for (i = 0; i < ATM_CNT; i++) atmArr[i].Initiate (i);
    for (i = 0; i < BCS_CNT; i++) bcsArr[i].Initiate (i);

    // Process timeouts and messages until all the ATMs have finished

    for (;;) {

        // Relay messages
        // With a 1 in 100 probability, lose the message

        while (messageQueue.size() != 0) {
            i = lrand48() % messageQueue.size();
            messageIter = messageQueue.begin();
            for (j = 0; j < i; j++) messageIter ++ ;
            message = *messageIter;
            messageQueue.erase (messageIter);
            if (lrand48() % 100 == 20) continue;
            switch (message.messageType) {
            case BASIC_REPLY:
                cerr << "Send basic reply from ATM "
                    << message.atmId
                    << " offerCode=" << message.offerCode
                    << " customerId=" << message.customerId
                    << " reply=" << message.offerAccepted
                    << '\n';
                offerServer.ReplyReceived (message.atmId,
                    message.offerCode, message.customerId,
                    message.offerAccepted);
                break;
            case RELAYED_REPLY:
                if (message.bcsId < 0 ||
                    message.bcsId >= BCS_CNT) {
                    cerr << __FILE__ << '(' << __LINE__
                        << ") bad bcsId "
                        << message.bcsId << '\n';
                    abort ();
                }
                cerr << "Send relayed reply to BCS "
                    << message.bcsId
                    << " offerCode=" << message.offerCode
                    << " customerId=" << message.customerId
                    << " reply=" << message.offerAccepted
                    << '\n';
                bcsArr[message.bcsId].ReplyReceived (&message);
                break;
            case BASIC_ANSWER:
                cerr << "Send basic answer from BCS "
                    << message.bcsId
                    << " offerCode=" << message.offerCode
                    << " customerId=" << message.customerId
```

```
                << '\n';
            offerServer.AnswerReceived (message.bcsId,
                message.offerCode, message.customerId,
                message.replyAccepted,
                message.reason.c_str());
            break;
        case RELAYED_ANSWER:
            if (message.atmId < 0 ||
                message.atmId >= ATM_CNT) {
                cerr << __FILE__ << '(' << __LINE__
                    << ") bad atmId "
                    << message.atmId << '\n';
                abort ();
            }
            cerr << "Send relayed answer to ATM "
                << message.atmId
                << " offerCode=" << message.offerCode
                << " customerId=" << message.customerId
                << '\n';
            atmArr[message.atmId].AnswerReceived (&message);
            break;
        default:
            cerr << __FILE__ << '(' << __LINE__
                << ") bad message type "
                << message.messageType << '\n';
            abort ();
            // NOTREACHED
        }
    }

    // If that resolves everything, the break out of the loop

    if (activeAtmCnt == 0) break;

    // Advance time to next thing to do

    wakeupIter = wakeupQueue.begin();
    if (wakeupIter == wakeupQueue.end()) {
        cerr << __FILE__ << '(' << __LINE__
            << ") nothing waiting\n";
        abort ();
    }
    currentTime = wakeupIter->wakeupTime;

    // Select all the wakeups with the current time

    wakeupVec.clear ();
    while (
        wakeupIter != wakeupQueue.end() &&
        wakeupIter->wakeupTime == currentTime
    ) {
        wakeupVec.push_back (*wakeupIter);
        wakeupIter ++ ;
    }

    // Randomly select one of the wakeups and execute it
```

```
        i = lrand48() % wakeupVec.size();
        wakeupQueue.erase (wakeupVec[i]);
        switch (wakeupVec[i].wakeupType) {
        case WAKEUP_ATM:
            if (wakeupVec[i].atmId < 0 ||
                wakeupVec[i].atmId >= ATM_CNT) {
                cerr << __FILE__ << '(' << __LINE__
                    << ") bad atmId "
                    << wakeupVec[i].atmId << '\n';
                abort ();
            }
            cerr << "Wake up ATM " << wakeupVec[i].atmId << '\n';
            atmArr[wakeupVec[i].atmId].Wakeup ();
            break;
        case WAKEUP_BCS:
            if (wakeupVec[i].bcsId < 0 ||
                wakeupVec[i].bcsId >= BCS_CNT) {
                cerr << __FILE__ << '(' << __LINE__
                    << ") bad bcsId "
                    << wakeupVec[i].bcsId << '\n';
                abort ();
            }
            cerr << "Wake up BCS " << wakeupVec[i].bcsId << '\n';
            bcsArr[wakeupVec[i].bcsId].Wakeup ();
            break;
        case WAKEUP_SERVER:
            // cerr << "Wake up offerServer\n";
            offerServer.Tick ();
            wakeupReq.wakeupType = WAKEUP_SERVER;
            wakeupReq.wakeupTime = currentTime + 1;
            wakeupQueue.insert (wakeupReq);
            break;
        default:
            cerr << __FILE__ << '(' << __LINE__
                << ") bad wakeup type "
                << wakeupVec[i].wakeupType << '\n';
            abort ();
            // NOTREACHED
        }
    }

    // Exit gracefully

    cerr << "Test satisfactorily completed\n";
    return 0;
}
```