```
// StockDisp.cpp - STOCK PRICE DISPLAY CONTROLLER
//
// USAGE
// StockDisp text-file
//
// text-file    is the name of a file containing the definition of the
//              message to be displayed.
//
// MAINTENANCE HISTORY
// DATE         PROGRAMMER AND DETAILS
// 07-10-04     JS      Original
//
//--------------------------------------------------------------------------

#include <sys/types.h>          // System type declarations
#include <ctype.h>              // Character type functions
#include <time.h>               // System time declarations
#include <string.h>             // String manipulation functions
#include <signal.h>             // Signal codes
#include <unistd.h>             // Operating system standard functions
#include <fcntl.h>              // File control options
#include <errno.h>              // Operating system error codes
#include <time.h>               // Time declarations
#include <sys/time.h>           // Interval timer declarations
#include <sys/socket.h>         // Socket declarations
#include <netinet/in.h>         // Internet type conversion functions
#include <arpa/inet.h>          // More Internet type conversion functions
#include <sys/ipc.h>            // IPC declarations
#include <sys/shm.h>            // Shared memory declarations
#include <sys/sem.h>            // Semaphore declarations
#include <iostream>             // C++ input/output streams
#include <iomanip>              // C++ input/output manipulators
#include <fstream>              // File stream declarations
#include <sstream>              // String stream declarations
#include "StockDef.h"           // Stock Price Display declarations
using namespace std;            // Expand standard namespace to global scope

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

// DEFINITIONS

const size_t            STOCK_CODE_LEN = 15;
                                // Stock code length
const time_t            MAX_STOCK_AGE = 5 * 60;
                                // Maximum stock age
const time_t            SERV_POLL_PERIOD = 60;
                                // Server polling period
const long              SCROLL_RATE = 25;
                                // Scroll rate in frames per second
const unsigned long     PRICE_SCALE = 100;
                                // Price scale
const unsigned short    STOCK_PRICE_REQ = 1486;
                                // Stock price request code
const unsigned short    STOCK_PRICE_RESP = 1487;
                                // Stock price response code

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

// STOCK DATA STRUCTURE

struct Stock_t {
    string              stockCode;              // Stock code
    Stock_t             *stockNext;             // Next stock on this server
    size_t              stockInd;               // Stock price index
};

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

// SERVER DATA STRUCTURE

struct Server_t {
    sockaddr_in         servAddr;               // Server address
    Server_t            *servNext;              // Next server in the list
    Stock_t             *servFirstStock;        // First stock served
    Stock_t             *servLastStock;         // Last stock served
    time_t              servUpdateTime;         // Time of last update
};

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

// STOCK PRICE STRUCTURE
```

```
struct Price_t {
    unsigned long       priceValue;             // Stock price
    time_t              priceTime;              // Time of the last update
};

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

// SEGMENT TYPE ENUMERATOR

enum SegType_t {
    SEG_TYPE_TEXT,      // A text segment
    SEG_TYPE_PRICE      // A stock price segment
};

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

// MESSAGE SEGMENT STRUCTURE

struct Segment_t {
    Segment_t           *segNext;               // Pointer to next segment
    SegType_t           segType;                // Segment type
    string              segText;                // Segment text string
    const Stock_t       *segStock;              // Pointer to the stock struct
};

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

// SERVER REQUEST STRUCTURE

struct ServReq_t {
    unsigned short      servReqCode;            // Request code
    unsigned short      servReqBodyLen;         // Request body length
    char                servReqStockCode[STOCK_CODE_LEN+1]; // Stock code
};

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

// SERVER RESPONSE STRUCTURE

struct ServResp_t {
    unsigned short      servRespCode;           // Response code
    unsigned short      servRespBodyLen;        // Response body length
    unsigned long       servRespPrice;          // Stock price in cents
};

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

// SERVER COMMUNICATIONS FAULT EXCEPTION

class ServFault_c {
public:
    int                 servFaultErrNo;         // Server fault error number
    const char          *servFaultMsg;          // Server fault message
    ServFault_c (int errNo, const char *msg)
        : servFaultErrNo (errno), servFaultMsg (msg) {}
                                                // Constructor
};

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

// GLOBAL DATA

ifstream::int_type      textChar;               // Text file character
ifstream                textStream;             // Text stream
Server_t                *firstServ;             // Pointer to first server
Server_t                *lastServ;              // Pointer to last server
Segment_t               *firstSeg;              // Pointer to first segment
Segment_t               *lastSeg;               // Pointer to last segment
size_t                  priceCnt;               // Stock price count
Price_t                 *priceArr;              // Stock price array (shared)
int                     shmId;                  // Shared memory identifier
int                     semId;                  // Semaphore identifier
pid_t                   seqGenPid = 0;          // Sequence generator proc id
const char              *dispChar;

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

// PROCESS APPLICATION-LEVEL ERROR

void
ErrApp (
    const char          *msg)           // Error message
```

```
{
    cerr << "Error: " << msg << '\n';
    kill (0, SIGTERM);
    exit (1);
}

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

// PROCESS SYSTEM-LEVEL ERROR

void
ErrSys (
    const char          *msg)           // Error message
{
    int                 savedErrNo;     // Saved error number

    savedErrNo = errno;
    cerr << "Error: " << msg << ": " << strerror(savedErrNo) << '\n';
    kill (0, SIGTERM);
    exit (1);
}

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

// COMPARE INTERNET SOCKET ADDRESSES FOR EQUALITY

inline bool
operator == (
    sockaddr_in         &a1,            // First address
    sockaddr_in         &a2)            // Second address
{
    return a1.sin_port == a2.sin_port &&
        a1.sin_addr.s_addr == a2.sin_addr.s_addr;
}

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

// COMPARE INTERNET SOCKET ADDRESSES FOR INEQUALITY

inline bool
operator != (
    sockaddr_in         &a1,            // First address
    sockaddr_in         &a2)            // Second address
{
    return ! (a1 == a2);
}

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

// SKIP WHITE SPACE IN THE TEXT STREAM

void
SkipSpace ()
{
    while (isspace(textChar))
        textChar = textStream.get();
}

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

// READ AN IDENTIFIER FROM THE TEXT STREAM

string
ReadId (
    bool                downShift)      // Down-shift the identifier
{
    string              id;             // Identifier

    while (isalpha(textChar)) {
        id += downShift ? tolower(textChar) : textChar;
        textChar = textStream.get();
    }
    return id;
}

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

// APPEND TEXT SEGMENT TO THE SEGMENT LIST

void
AppendText (
    string              *text)          // Text segment to be appended
```

```
{
    Segment_t           *seg;           // Segment structure pointer

    // Disregard null segments

    if (*text == "") return;

    // Create a new segment structure, load it and append it to the
    // segment list.

    seg = new Segment_t;
    seg->segNext = 0;
    seg->segType = SEG_TYPE_TEXT;
    seg->segText = *text;
    seg->segStock = 0;
    if (lastSeg == 0)
        firstSeg = seg;
    else
        lastSeg->segNext = seg;
    lastSeg = seg;

    // Reset the text accumulator

    *text = "";
}

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

// LOAD THE TEXT FILE

void
LoadText (
    const char          *fileName)      // Name of the message definition file
{
    string              text;           // Text string
    string              id;             // Identifier
    string              ipAddr;         // IP address
    unsigned short      port;           // Port number
    string              code;           // Stock code
    sockaddr_in         sockAddr;       // Socket address
    Server_t            *serv;          // Server structure pointer
    Stock_t             *stock;         // Stock structure pointer
    Segment_t           *seg;           // Segment structure pointer

    // Initialise the server and segment lists

    firstServ = 0;
    lastServ = 0;
    firstSeg = 0;
    lastSeg = 0;
    priceCnt = 0;

    // Open the text stream and load the look-ahead character

    textStream.open (fileName);
    if ( ! textStream) {
        cerr << "Error: cannot open " << fileName << '\n';
        exit (1);
    }
    textChar = textStream.get();

    // Process characters in the text stream until we reach end-of-file

    while (textChar != ifstream::traits_type::eof()) {

        // If the character is a new-line, convert it into a space

        if (textChar == '\n') {
            text += ' ';
            textChar = textStream.get();
        }

        // If the character is an ampersand, decode the special character
        // code and append it to the gathering text string.

        else if (textChar == '&') {
            textChar = textStream.get();
            id = ReadId (1);
            if (textChar != ';')
                ErrApp ("bad character code format");
            textChar = textStream.get();
            if (id == "amp")
```

```
            text += '&';
        else if (id == "lt")
            text += '<';
        else
            ErrApp ("unrecognised character code");
    }

    // Process stock price segments.

    else if (textChar == '<') {
        textChar = textStream.get();
        SkipSpace ();

        // Append any accumulated text to the segment list.

        AppendText (&text);

        // Validate the tag name

        if (ReadId (1) != "price" || (!isspace(textChar) && textChar!='>'))
            ErrApp ("unrecognised tag");
        SkipSpace ();

        // Process server and stock code options

        ipAddr = "";
        port = 1845;
        code = "";
        while (textChar!='>' && textChar!=ifstream::traits_type::eof()) {
            id = ReadId (1);
            SkipSpace ();
            if (textChar != '=')
                ErrApp ("invalid price parameter");
            textChar = textStream.get();
            SkipSpace ();
            if (id == "server") {
                if (ipAddr != "")
                    ErrApp ("duplicate server parameter");
                if ( ! isdigit(textChar))
                    ErrApp ("invalid IP address");
                while (isdigit(textChar) || textChar == '.') {
                    ipAddr += textChar;
                    textChar = textStream.get();
                }
                if (!isspace(textChar) && textChar!=':' && textChar!='>')
                    ErrApp ("invalid IP address");
                SkipSpace ();
                if (textChar == ':') {
                    textChar = textStream.get();
                    SkipSpace ();
                    if ( ! isdigit(textChar))
                        ErrApp ("invalid IP port");
                    port = 0;
                    while (isdigit(textChar)) {
                        port = port * 10 + textChar - '0';
                        textChar = textStream.get();
                    }
                    if ( ! isspace(textChar) && textChar != '>')
                        ErrApp ("invalid IP port");
                    SkipSpace ();
                }
            }

            else if (id == "code") {
                if (code != "")
                    ErrApp ("duplicate stock code parameter");
                code = ReadId (0);
                if (code == "" || (!isspace(textChar) && textChar != '>'))
                    ErrApp ("invalid stock code");
                if (code.size() > STOCK_CODE_LEN)
                    ErrApp ("stock code too long");
                SkipSpace ();
            }

            else
                ErrApp ("unrecognised parameter");
        }
        if (textChar != '>')
            ErrApp ("unexpected end-of-file");
        textChar = textStream.get();
        if (ipAddr == "")
            ErrApp ("no server address");
```

```
            if (code == "")
                ErrApp ("no stock code");

            // Load the internet address structure

            memset (&sockAddr, 0, sizeof(sockAddr));
            sockAddr.sin_family = AF_INET;
            sockAddr.sin_port = htons(port);
            if ( ! inet_aton (ipAddr.c_str(), &sockAddr.sin_addr))
                ErrApp ("invalid IP address");

            // Search for a server for this address

            serv = firstServ;
            while (serv != 0 && serv->servAddr != sockAddr)
                serv = serv->servNext;

            // If no server was found, create a new server structure.

            if (serv == 0) {
                serv = new Server_t;
                serv->servNext = 0;
                serv->servAddr = sockAddr;
                serv->servFirstStock = 0;
                serv->servLastStock = 0;
                serv->servUpdateTime = time(0) - SERV_POLL_PERIOD;
                if (lastServ == 0)
                    firstServ = serv;
                else
                    lastServ->servNext = serv;
                lastServ = serv;
            }

            // Create a stock structure and append it to the stock
            // list of the server.

            stock = new Stock_t;
            stock->stockCode = code;
            stock->stockNext = 0;
            stock->stockInd = priceCnt ++ ;
            if (serv->servLastStock == 0)
                serv->servFirstStock = stock;
            else
                serv->servLastStock->stockNext = stock;
            serv->servLastStock = stock;

            // Create a stock price segment and append it to the end
            // of the segment list.

            seg = new Segment_t;
            seg->segNext = 0;
            seg->segType = SEG_TYPE_PRICE;
            seg->segText = "";
            seg->segStock = stock;
            if (lastSeg == 0)
                firstSeg = seg;
            else
                lastSeg->segNext = seg;
            lastSeg = seg;
        }

        // If the character is printable and not special, append
        // the character to the current text string.

        else if (isprint(textChar)) {
            text += textChar;
            textChar = textStream.get();
        }
    }

    // Append a space to the last text segment to the segment list.

    text += ' ';
    AppendText (&text);

    // Close the text stream

    textStream.close ();
}

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

```
// CREATE SHARED MEMORY SEGMENT

void
CreateShm ()
{
    size_t              shmSize;        // Shared memory segment size
    time_t              outOfDate;      // Out-of-date update time
    size_t              i;              // General purpose index

    // Create the shared memory segment

    shmSize = priceCnt * sizeof(Price_t);
    if (shmSize != 0) {
        if ((shmId = shmget (IPC_PRIVATE, shmSize, 0666|IPC_CREAT)) == -1)
            ErrSys ("create shared memory segment");
        if ((priceArr = static_cast<Price_t*>(shmat (shmId, 0, 0))) == 0)
            ErrSys ("attach to shared memory segment");
        if (shmctl (shmId, IPC_RMID, 0) == -1)
            ErrSys ("remove shared memory id");
    }

    // Initialise the shared memory structure

    outOfDate = time(0) - MAX_STOCK_AGE - 1;
    for (i = 0; i < priceCnt; i++) {
        priceArr[i].priceValue = 0;
        priceArr[i].priceTime = outOfDate;
    }

    // Create the semaphore

    if ((semId = semget (IPC_PRIVATE, 1, 0666)) == -1)
        ErrSys ("create semaphore");
    if (semctl (semId, 0, SETVAL, 1) == -1)
        ErrSys ("initialise semaphore");
}

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

// ALARM SIGNAL HANDLER

void
AlarmHandler (
    int                 /*sig*/)       // Signal number
{
    // Empty
}

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

// WAIT FOR ALARM

void
WaitForAlarm ()
{
    sigset_t            emptySet;       // No signals

    // Suspend execution until an alarm is received

    sigemptyset (&emptySet);
    sigsuspend (&emptySet);
    if (errno != EINTR) ErrSys ("sigsuspend");
}

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

// DISPLAY TEXT

void
DispText (
    string              text)           // Text to display
{
    const char          *p;             // Character pointer
    const Bitmap_t      *bm;            // Bitmap pointer
    size_t              i;              // General purpose index

    // Display each bitmap in each character in the string
    // with the correct timing

    for (p = text.c_str(); *p != '\0'; p++) {
        bm = GetBitmap (*p);
        for (i = 0; i < bm->bmColCnt; i++) {
```

```
            WaitForAlarm ();
            ShiftAndLoad (bm->bmColArr[i]);
        }
    }
}

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

// GENERATE BITMAPS

void
GenerateBitmaps ()
{
    struct itimerval    iTimVal;        // Interval timer value
    struct sigaction    alarmSa;        // Alarm signal action
    sigset_t            alarmSet;       // Alarm signal set
    Segment_t           *seg;           // Segment pointer
    struct sembuf       sops;           // Semaphore operation
    Price_t             price;          // Price structure
    ostringstream       ost;            // Output string stream
    string              priceText;      // Stock price text

    // Block alarm signals

    sigemptyset (&alarmSet);
    sigaddset (&alarmSet, SIGALRM);
    if (sigprocmask (SIG_BLOCK, &alarmSet, 0) == -1)
        ErrSys ("block alarm signals");

    // Vector alarm signals to the Alarm Signal Handler

    memset (&alarmSa, 0, sizeof(alarmSa));
    alarmSa.sa_handler = AlarmHandler;
    sigemptyset (&alarmSa.sa_mask);
    sigaddset (&alarmSa.sa_mask, SIGALRM);
    if (sigaction (SIGALRM, &alarmSa, 0) == -1)
        ErrSys ("sigaction SIGALRM");

    // Configure the interval timer to generate an alarm at
    // the time every bitmap must be displayed

    memset (&iTimVal, 0, sizeof(iTimVal));
    iTimVal.it_interval.tv_sec = 0;
    iTimVal.it_interval.tv_usec = 1000000 / SCROLL_RATE;
    iTimVal.it_value = iTimVal.it_interval;
    if (setitimer (ITIMER_REAL, &iTimVal, 0) == -1)
        ErrSys ("setitimer");

    // Loop through sequences

    for (;;) {

        // Loop through segments

        for (seg = firstSeg; seg != 0; seg = seg->segNext) {

            // Process the segment according to its type

            switch (seg->segType) {

            // Text segments

            case SEG_TYPE_TEXT:

                // Display the text

                DispText (seg->segText);
                break;

            // Stock price segments

            case SEG_TYPE_PRICE:

                // Lock the shared memory segment, read the stock price
                // and then unlock the shared memory segment.

                sops.sem_num = 0;
                sops.sem_op = -1;
                sops.sem_flg = SEM_UNDO;
                if (semop (semId, &sops, 1) == -1)
                    ErrSys ("lock shared memory");
                price = priceArr[seg->segStock->stockInd];
```

```
                sops.sem_op = 1;
                if (semop (semId, &sops, 1) == -1)
                    ErrSys ("unlock shared memory");

                // Put the stock price into the output string stream

                ost.str("");
                if (price.priceTime <= time(0) - MAX_STOCK_AGE)
                    ost << "?.??";
                else
                    ost << price.priceValue / PRICE_SCALE
                        << '.' << setfill('0') << setw(2)
                        << price.priceValue % PRICE_SCALE;

                // Display the stock price

                DispText (ost.str());
                break;

            // Other segment types are unacceptable

            default:
                ErrApp ("bad segment type");
                // NOTREACHED
            }
        }
    }
}

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

// UPDATE STOCK PRICES

void
UpdatePrices ()
{
    Server_t            *serv;          // Server pointer
    Stock_t             *stock;         // Stock pointer
    int                 servSockFd;     // Server socket file descriptor
    sockaddr_in         servInAddr;     // Server's Internet address
    ServReq_t           servReq;        // Server request
    ServResp_t          servResp;       // Server response structure
    ssize_t             rdLen;          // Read length
    size_t              respLen;        // Response length
    long                waitTime;       // Wait time
    struct sembuf       sops;           // Semaphore operation

    // Select the first server.  If there are no servers, pause until
    // terminated.

    serv = firstServ;
    if (serv == 0) for (;;) pause ();

    // Loop processing consecutive connections to the server

    for (;;) {

        // Wait until the server update time has expired

        waitTime = serv->servUpdateTime + SERV_POLL_PERIOD - time(0);
        if (waitTime > 0)
            sleep (static_cast<unsigned>(waitTime));

        // Create an endpoint for communication

        servSockFd = socket (PF_INET, SOCK_STREAM, 0);
        if (servSockFd == -1)
            ErrSys ("open server socket");

        // Catch server communications faults

        try {

            // Attempt to connect to the server

            if (
                connect (
                    servSockFd,
                    reinterpret_cast<sockaddr*>(&serv->servAddr),
                    sizeof(servInAddr)
                ) == -1
            )
```

```
            throw ServFault_c (errno, "connect to server");

        // Process a request for each stock connected to the
        // server.

        for (
            stock = serv->servFirstStock;
            stock != 0;
            stock = stock->stockNext
        ) {

            // Load and send a request

            servReq.servReqCode = htons(STOCK_PRICE_REQ);
            servReq.servReqBodyLen = htons(16);
            strcpy (servReq.servReqStockCode, stock->stockCode.c_str());
            if (write (servSockFd, &servReq, sizeof(servReq))
              != sizeof(servReq))
                throw ServFault_c (errno, "write to server");

            // Receive the server's response

            respLen = 0;
            do {
                errno = 0;
                rdLen = read (
                    servSockFd,
                    reinterpret_cast<char*>(&servResp) + respLen,
                    sizeof(servResp) - respLen
                );
                if (
                    rdLen <= 0 ||
                    rdLen > static_cast<ssize_t>(sizeof(servResp)-respLen)
                )
                    throw ServFault_c (errno, "read failed");
                respLen += rdLen;
            } while (respLen < sizeof(servResp));

            // Validate the server response

            errno = 0;
            if (ntohs(servResp.servRespCode) != STOCK_PRICE_RESP)
                throw ServFault_c (errno, "wrong response code");
            if (ntohs(servResp.servRespBodyLen) != 4)
                throw ServFault_c (errno, "wrong response length");

            // Update the stock price in the shared memory segment

            sops.sem_num = 0;
            sops.sem_op = -1;
            sops.sem_flg = SEM_UNDO;
            if (semop (semId, &sops, 1) == -1)
                ErrSys ("lock shared memory");
            priceArr[stock->stockInd].priceValue = servResp.servRespPrice;
            priceArr[stock->stockInd].priceTime = time(0);
            sops.sem_op = 1;
            if (semop (semId, &sops, 1) == -1)
                ErrSys ("unlock shared memory");
        }
    }

    // Process server communications faults

    catch (ServFault_c servFault) {

        // Log the communications fault

        cout << "Warning: server fault: " << servFault.servFaultMsg;
        if (servFault.servFaultErrNo != 0)
            cout << ": " << strerror (servFault.servFaultErrNo);
        cout << '\n';
    }

    // Close the socket file descriptor

    close (servSockFd);

    // Save the last update time

    serv->servUpdateTime = time(0);

    // Point to the next server
```

```
        if (serv->servNext == 0)
            serv = firstServ;
        else
            serv = serv->servNext;
    }
}

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

// MAIN LINE

int
main (
    int                 argc,           // Argument count
    char                *argv[])        // Argument value pointers
{
    pid_t               btmGenPid;      // Sequence generator process id

    // Validate arguments

    if (argc != 2) {
        cerr << "Usage: StockDisp text-file\n";
        exit (1);
    }

    // Load the text file

    LoadText (argv[1]);

    // Create the shared memory segment

    CreateShm ();

    // Initiate the Bitmap Generator

    if ((btmGenPid = fork()) == -1)
        ErrSys ("create bitmap generator");
    if (btmGenPid == 0)
        GenerateBitmaps ();

    // Initiate stock price updating

    UpdatePrices ();
    return 0;
}
```