```
// TimeServer.cpp - TIME SERVER
//
// MODULE INDEX
// NAME             CONTENTS
// EncHttpDate      Encode http date
// getch            Get a character from the input stream disregarding
//                  carriage return
// main             Main line
//
// MAINTENANCE HISTORY
// DATE         PROGRAMMER AND DETAILS
// 26-09-12 JS  Original
//
//-----------------------------------------------------------------------

#include <cstdlib>       // C-style standard library
#include <cstring>       // C-style string manipulation functions
#include <cerrno>        // C-style operating system error codes
#include <cstdio>        // C-style standard input/output functions
#include <iostream>      // C++ I/O stream declarations
#include <iomanip>       // C++ I/O manipulator declarations
#include <sstream>       // C++ string stream declarations
#include <fstream>       // C++ file stream declarations
using namespace std;     // Expand the standard namespace
#include <sys/types.h>   // Operating system type declarations
#include <sys/socket.h>  // Socket declarations
#include <netinet/in.h>  // Internet socket declarations
#include <unistd.h>      // Unix standard functions

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

// ENCODE HTTP DATE

string
EncHttpDate (
    struct tm   *brokenTime,     // Broken down time
    const char  *timeCode)   // Time code
{
    ostringstream   dateStream; // Date stream

    char        *dowTxt[] = {
                "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"
            };
    char        *moTxt[] = {
                "Jan", "Feb", "Mar", "Apr", "May", "Jun",
                "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
            };

    // Encode the date

    dateStream << dowTxt[brokenTime->tm_wday];
    dateStream << ", ";
    dateStream << setw(2) << setfill('0') << brokenTime->tm_mday;
    dateStream << ' ';
    dateStream << moTxt[brokenTime->tm_mon];
    dateStream << ' ';
    dateStream << setw(4) << setfill('0') << (brokenTime->tm_year+1900);
```

```
    dateStream << ' ';
    dateStream << setw(2) << setfill('0') << brokenTime->tm_hour;
    dateStream << ':';
    dateStream << setw(2) << setfill('0') << brokenTime->tm_min;
    dateStream << ':';
    dateStream << setw(2) << setfill('0') << brokenTime->tm_sec;
    dateStream << ' ';
    dateStream << timeCode;
    return dateStream.str();
}

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

// GET A CHARACTER FROM THE INPUT STREAM DISREGARDING CARRIAGE RETURN

int
getch (
    FILE        *acceptFile)     // Accepted file stream
{
    int     ch;     // Received character

    while ((ch = getc (acceptFile)) == '\r');
    return ch;
}

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

// MAIN LINE

int
main ()
{
    struct sockaddr_in listenAddr;   // Address to listen on
    int     listenSock;     // Listening socket file descriptor
    int     opt;            // Socket option
    int     acceptSock;     // Accepted socket file descriptor
    struct sockaddr_in acceptAddr;   // Accepted client address
    socklen_t   acceptLen;  // Accepted client address length
    FILE        *acceptFile;     // Accepted file pointer
    string      rxBuf;      // Receive buffer
    int     rxLen;          // Received length
    int     i;              // General purpose pointer
    ostringstream   bodyStream; // Response body stream
    string      bodyString; // Response body string
    ostringstream   respStream; // Response stream
    string      respString; // Response string
    time_t      serverTime; // Current server time
    string      cmd;        // Command
    string      path;       // Path name
    ifstream    ifs;        // Input file stream
    int     ch;             // A character
    int     sleepTime;      // Sleep time

    // Put the port number and host address into the socket structure

    memset (&listenAddr, 0, sizeof(listenAddr));
    listenAddr.sin_family = AF_INET;
```

```
    listenAddr.sin_port = htons (2212);
    listenAddr.sin_addr.s_addr = htonl(INADDR_ANY);

    // Allocate a socket for incomming connections

    if ((listenSock = socket (AF_INET, SOCK_STREAM, 0)) == -1) {
        cerr << "Error: cannot open listen socket: "
            << strerror(errno) << '\n';
        abort ();
    }

    // Enable socket address reuse - anything else will drive
    // everybody crazy with rejected binds.  Old addresses appear
    // to remain allocated for around one minute after the close
    // call returns.

    opt = 1;
    if (setsockopt (listenSock, SOL_SOCKET, SO_REUSEADDR,
        reinterpret_cast<char*>(&opt), sizeof(opt)) == -1) {
        cerr << "Error: setsockopt: " << strerror(errno) << '\n';
        abort ();
    }

    // Bind the socket to the service port so we hear incoming
    // requests

    if (bind (listenSock, reinterpret_cast<sockaddr*>(&listenAddr),
        sizeof(listenAddr)) == -1) {
        cerr << "Error: bind: " << strerror(errno) << '\n';
        abort ();
    }

    // Initiate listening on the socket

    if (listen (listenSock, 5) == -1) {
        cerr << "Error: listen: " << strerror(errno) << '\n';
        abort ();
    }

    // Listen for connections until shut down by a signal

    for (;;) {

        // Accept a connection

        memset (&acceptAddr, 0, sizeof(acceptAddr));
        acceptLen = sizeof(acceptAddr);
        acceptSock = accept (listenSock,
            reinterpret_cast<sockaddr*>(&acceptAddr),
            &acceptLen);
        if (acceptSock == -1) {
            cerr << "Error: accept: " << strerror(errno) << '\n';
            abort ();
        }

        // Open the accepted file stream
```

```
    if ((acceptFile = fdopen (acceptSock, "r")) == 0) {
        cerr << "Error: fdopen: " << strerror(errno) << '\n';
        abort ();
    }

    // Read requests

    while ((ch = getch (acceptFile)) != -1) {

        // Read the request header

        rxBuf = "";
        do {
            while (ch != '\n' && ch != -1) {
                rxBuf += ch;
                ch = getch (acceptFile);
            }
            rxBuf += ch;
            ch = getch (acceptFile);
        } while (ch != '\n');

        // Log the request

        cout << rxBuf;

        // Decode the command and path

        i = 0;
        cmd = "";
        while (i < rxLen && rxBuf[i] != ' ')
            cmd += rxBuf[i++];
        while (i < rxLen && rxBuf[i] == ' ')
            i ++ ;
        path = "";
        while (i < rxLen && rxBuf[i] != ' ')
            path += rxBuf[i++];

        // Initialise the response body

        bodyStream.str("");
        bodyStream.clear();

        // Process requests for the time

        if (path == "/TIME") {

            // Sleep for a pseudo-random time to
            // simulate network delay

            sleepTime = lrand48() % 10 + 1;
            sleep (sleepTime);
            cout << "sleepTime = " << sleepTime << '\n';

            // Load the server time

            serverTime = time(0);
```

```
        // Create the response body

        bodyStream << "<html>\r\n";
        bodyStream << "<body>\r\n";
        bodyStream << EncHttpDate
            (localtime(&serverTime), "MYT")
                << "\r\n";
        bodyStream << "</body>\r\n";
        bodyStream << "</html>\r\n";
    }

    // Process requests for files

    else {
        ifs.clear();
        ifs.open (path.c_str()+1);
        if ( ! ifs) {
            cerr << "Cannot open " << path << '\n';
        } else {
            while ((ch = ifs.get()) != -1)
                bodyStream.put (ch);
            ifs.close ();
        }
    }


    // Load the response body string

    bodyString = bodyStream.str();

    // Create the response message

    respStream.str("");
    respStream.clear();
    respStream << "HTTP/1.1 200 OK\r\n";
    respStream << "Server: TimeServer\r\n";
    respStream << "Date: " <<
        EncHttpDate(gmtime(&serverTime), "GMT")
            << "\r\n";
    respStream << "Accept-ranges: none\r\n";
    respStream << "Content-type: text/plain\r\n";
    respStream << "Connection: close\r\n";
    respStream << "Content-length: " <<
        bodyString.length() << "\r\n";
    respStream << "\r\n";
    respStream << bodyString;
    respStream << "\r\n";
    respString = respStream.str();

    // Log the response

    // cout << respString;

    // Send the response message

    if (write (acceptSock, respString.c_str(),
        respString.length()) != respString.length()) {
```

```
                cerr << "Error: write: "
                    << strerror(errno) << '\n';
                abort ();
            }
        }

        // Close the request stream

        fclose (acceptFile);
    }

    return 0;
}
```