```
// AppRes.cpp - APPROVAL RESOLUTION PROGRAM
//
// MODULE INDEX
// NAME                 CONTENTS
// Crash                Process a fatal error
// ReadExecutives       Read the executives file
// GetToken             Read a token
// ReadApprovals        Read the approvals file
// GetStatus            Get the status of an approval level
// main                 Main line
//
// MAINTENANCE HISTORY
// DATE         PROGRAMMER AND DETAILS
// 04-10-15     JS      Original
//
//-----------------------------------------------------------------------------

#include <cstring>              // C-style string manipulation functions
#include <cctype>               // C-style character typing functions
#include <string>               // C++ string declarations
#include <iostream>             // C++ I/O stream declarations
#include <fstream>              // C++ file stream declarations
#include <vector>               // C++ vector declarations
#include <map>                  // C++ map declarations
#include <set>                  // C++ set declarations
using namespace std;            // Expand the standard namespace

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

// TOKEN TYPE ENUMERATOR

enum TokenType_t {PLUS, TIMES, SEMI, INTEGER, EVP, SVP, VP, AVP,
        BUY, SELL, END};

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

// APPROVAL STATUS ENUMERATOR

enum Status_t {NONE, PART, FULL};

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

// EXECUTIVES MAP DECLARATION

typedef map<string,TokenType_t> ExecMap_t;
                                        // Map type definition
typedef ExecMap_t::iterator ExecIter_t;
                                        // Iterator type definition

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

// LEVEL VECTOR DECLARATION

typedef vector<TokenType_t> LevelVec_t;
                                        // Vector declaration

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

// APPROVAL VECTOR DECLARATION

struct AppRec_t {
        LevelVec_t      appLevelVec;    // Level vector
        bool            appBuyAuth;     // Purchase approval authorised
        long            appBuyLimit;    // Purchase limit
```

```
        bool            appSellAuth;    // Sale approval authorised
        long            appSellLimit;   // Sale limit
};
typedef vector<AppRec_t> AppVec_t;

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

// GLOBAL DATA

ifstream                tokenStream;    // Token input stream
TokenType_t             tokenType;      // Token type code
int                     tokenVal;       // Token value
istream::int_type       nextCh;         // Next character
ExecMap_t               execMap;        // Executives map
AppVec_t                appVec;         // Approvals vector

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

// PROCESS A FATAL ERROR

void
Crash (
        const char      *message)       // Error message
{
        cerr << "Error: " << message << '\n';
        exit (1);
}

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

// READ THE EXECUTIVES FILE

void
ReadExecutives ()
{
        ifstream        inStream;       // Input stream
        istream::int_type inCh;         // Input character
        string          execName;       // Executive name
        string          execLevel;      // Executive level

        inStream.open ("EXECUTIVES.TXT");
        if ( ! inStream) Crash ("cannot open EXECUTIVES.TXT");
        inCh = inStream.get();
        while (inCh != istream::traits_type::eof()) {
                execName = "";
                while (inCh != '\t' && inCh != istream::traits_type::eof()) {
                        if (islower(inCh)) inCh += 'A' - 'a';
                        execName += static_cast<char>(inCh);
                        inCh = inStream.get();
                }
                while (inCh == '\t') inCh = inStream.get();
                execLevel = "";
                while (inCh != '\n' && inCh != istream::traits_type::eof()) {
                        if (islower(inCh)) inCh += 'A' - 'a';
                        execLevel += static_cast<char>(inCh);
                        inCh = inStream.get();
                }
                while (inCh == '\n') inCh = inStream.get();
                if (execLevel == "EVP")
                        execMap[execName] = EVP;
                else if (execLevel == "SVP")
                        execMap[execName] = SVP;
                else if (execLevel == "VP")
                        execMap[execName] = VP;
                else if (execLevel == "AVP")
```

```
                        execMap[execName] = AVP;
                else
                        Crash ("invalid level in executives file");
        }
}

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

// READ A TOKEN

void
GetToken ()
{
        string          word;           // Keyword string
        long            newVal;         // New token value

        while (isspace(nextCh)) nextCh = tokenStream.get();
        if (isalpha(nextCh)) {
                word = "";
                do {
                        word += static_cast<char>(nextCh);
                        nextCh = tokenStream.get();
                } while (isalpha(nextCh));
                if (word == "EVP")
                        tokenType = EVP;
                else if (word == "SVP")
                        tokenType = SVP;
                else if (word == "VP")
                        tokenType = VP;
                else if (word == "AVP")
                        tokenType = AVP;
                else if (word == "BUY")
                        tokenType = BUY;
                else if (word == "SELL")
                        tokenType = SELL;
                else
                        Crash ("bad keyword in APPROVALS.TXT");
        }
        else if (isdigit (nextCh)) {
                tokenVal = 0;
                while (isdigit (nextCh)) {
                        newVal = tokenVal * 10;
                        if (newVal/10 != tokenVal) Crash ("number too big");
                        tokenVal = newVal + nextCh - '0';
                        if (tokenVal < 0) Crash ("number too big");
                        nextCh = tokenStream.get();
                }
                tokenType = INTEGER;
        }
        else {
                if (nextCh == '*') tokenType = TIMES;
                else if (nextCh == '+') tokenType = PLUS;
                else if (nextCh == ';') tokenType = SEMI;
                else if (nextCh == istream::traits_type::eof()) tokenType = END;
                else Crash ("invalid character in APPROVALS.TXT");
                nextCh = tokenStream.get();
        }
}

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

// READ THE APPROVALS FILE

void
ReadApprovals ()
```

```
{
        AppRec_t        appRec;         // Approval record
        long            quantity;       // Number of approvals required
        long            i;              // General purpose index

        // Open the input token stream

        tokenStream.open ("APPROVALS.TXT");
        if ( ! tokenStream) Crash ("cannot open APPROVALS.TXT");
        nextCh = tokenStream.get ();
        GetToken ();

        // Process approvals until end-of-file

        while (tokenType != END) {

                // Read the approval combination

                appRec.appLevelVec.clear ();
                for (;;) {
                        if (tokenType == INTEGER) {
                                quantity = tokenVal;
                                GetToken ();
                                if (tokenType != TIMES)
                                        Crash ("expected '*'");
                                GetToken ();
                        } else {
                                quantity = 1;
                        }
                        if (
                                tokenType != EVP && tokenType != SVP &&
                                tokenType != VP && tokenType != AVP
                        ) Crash ("invalid level code");
                        for (i = 0; i < quantity; i++)
                                appRec.appLevelVec.push_back (tokenType);
                        GetToken ();
                        if (tokenType != PLUS) break;
                        GetToken ();
                }

                // Read the approval limits

                if (tokenType == BUY) {
                        appRec.appBuyAuth = 1;
                        GetToken ();
                        if (tokenType != INTEGER)
                                Crash ("buy limit is not integer");
                        appRec.appBuyLimit = tokenVal;
                        GetToken ();
                } else {
                        appRec.appBuyAuth = 0;
                }
                if (tokenType == SELL) {
                        appRec.appSellAuth = 1;
                        GetToken ();
                        if (tokenType != INTEGER)
                                Crash ("sell limit is not integer");
                        appRec.appSellLimit = tokenVal;
                        GetToken ();
                } else {
                        appRec.appSellAuth = 0;
                }
                if (tokenType != SEMI)
                        Crash ("missing semicolon");
                GetToken ();
```

```
                // Append the approval to the approval vector

                appVec.push_back (appRec);
        }
        tokenStream.close ();
}

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

// GET THE STATUS OF AN APPROVAL LEVEL

Status_t
GetStatus (
        LevelVec_t      availLevelVec, // Available levels
        const LevelVec_t *rqrdLevelVec) // Required levels
{
        size_t          i, j;           // General purpose indices
        size_t          matchCnt;       // Required levels that match

        matchCnt = 0;
        for (i = 0; i < rqrdLevelVec->size(); i++) {
                j = 0;
                while (
                        j < availLevelVec.size() &&
                        availLevelVec[j] != (*rqrdLevelVec)[i]
                ) j ++ ;
                if (j < availLevelVec.size()) {
                        matchCnt ++ ;
                        availLevelVec.erase (availLevelVec.begin() + j);
                }
        }
        return matchCnt==0 ? NONE : matchCnt<rqrdLevelVec->size() ? PART : FULL;
}

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

// MAIN LINE

int
main ()
{
        set<string>     execNameSet;    // Executive name set
        LevelVec_t      availLevelVec;  // Available levels vector
        istream::int_type inCh;         // Input character
        string          execName;       // Executive name
        ExecIter_t      execIter;       // Executive map iterator
        size_t          i;              // General purpose index
        TokenType_t     txType;         // Transaction type
        long            txValue;        // Transaction value
        long            newValue;       // New transaction value
        Status_t        bestStatus;     // Best status value
        Status_t        thisStatus;     // This approval level status

        // Read the executives table

        ReadExecutives ();

        // Read the approvals table

        ReadApprovals ();

        // Read the vector of executives who have approved the transaction

        cout << "Enter the names of executives who have approved\n";
```

```
cout << "the transaction.  Enter a blank line to end the list.\n";
while ((inCh = cin.get ()) != '\n') {
        execName = "";
        while (inCh != '\n') {
                if (islower(inCh)) inCh += 'A' - 'a';
                execName += static_cast<char>(inCh);
                inCh = cin.get();
        }
        execIter = execMap.find (execName);
        if (execIter == execMap.end()) {
                cout << "Error: invalid executive name\n";
                cout << "Continue entering names\n";
                continue;
        }
        if (execNameSet.count(execName) != 0) {
                cout << "Error: duplicate executive name\n";
                cout << "Continue entering names\n";
                continue;
        }
        availLevelVec.push_back (execIter->second);
        execNameSet.insert (execName);
}

// Read the transaction type

cout << "Enter the transaction type ('B' for buy or 'S' for sell):\n";
txType = END;
do {
        inCh = cin.get();
        if (inCh == 'b' || inCh == 'B')
                txType = BUY;
        else if (inCh == 's' || inCh == 'S')
                txType = SELL;
        else
                cout << "Error: invalid selection, please try again\n";
        while (inCh != '\n') inCh = cin.get();
} while (txType == END);

// Read the transaction value

cout << "Enter the transaction value:\n";
for (;;) {
        txValue = 0;
        inCh = cin.get();
        while (isdigit (inCh)) {
                newValue = txValue * 10;
                if (newValue / 10 != txValue) break;
                txValue = newValue + inCh - '0';
                if (txValue < 0) break;
                inCh = cin.get();
        }
        if (inCh == '\n') break;
        while (inCh != '\n') inCh = cin.get();
        cout << "Error: invalid transaction value, please try again\n";
}

// Test for full approval

bestStatus = NONE;
for ( i = 0; i < appVec.size(); i++) {
        if (
                (
                        (txType == BUY && appVec[i].appBuyAuth &&
                                appVec[i].appBuyLimit >= txValue) ||
                        (txType == SELL && appVec[i].appSellAuth &&
```

```
                                    appVec[i].appSellLimit >= txValue)
                    ) &&
                    (thisStatus = GetStatus (availLevelVec,
                            &appVec[i].appLevelVec)) > bestStatus
            ) bestStatus = thisStatus;
        }
        switch (bestStatus) {
        case NONE:      cout << "Unapproved\n";         break;
        case PART:      cout << "Partly approved\n";    break;
        case FULL:      cout << "Fully approved\n";     break;
        default:        cout << "?\n";                  break;
        }
        return 0;
}
```