```
// Prelim.cpp - DETERMINE WHETHER OR NOT A REPORT IS PRELIMINARY
//
// MODULE INDEX
// NAME                 CONTENTS
// ListSet          List a set
// DateSetUnion         Determine the union of two date sets
// DateSetIntersect     Determine the intersection of two date sets
// PrevDate         Return the previous date
// NextDate             Return the next date
// DateSetComplement    Determine the complement of a date set
// Evaluate         Evaluate the expression
// Prelim           Determine if a report is preliminary
// main                 Testing main line
//
// MAINTENANCE HISTORY
// DATE           PROGRAMMER AND DETAILS
// 06-09-11 JS    Original
// 09-09-11 HYC   Correct DateSet increment in DateSetUnion
// 15-09-11  JS   Corrected intersection function
//
//-----------------------------------------------------------------------

#include <cstring>        // C-style string manipulation functions
#include <cstdlib>        // C-style standard library
#include <string>         // C++ string declarations
#include <iostream>       // C++ input/output stream declarations
#include <sstream>        // C++ string stream declarations
#include <set>            // C++ set declarations
using namespace std;           // Expand the standard namespace
#include "dtime.h"        // Double time functions

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

// OPERATION ENUMERATOR

enum Operation_t {EQ, NE, GT, GE, LT, LE, AND, OR, NOT};

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

// BASIC NODE STRUCTURE

struct Node_t {
    Operation_t operation;   // Operation code
};

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

// COMPARE NODE

struct CompareNode_t : public Node_t {
    string       id;      // Selection key identifier
    string       value;       // Selection key value
};

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

// BINARY NODE

struct BinaryNode_t : public Node_t {
```

```
    Node_t      *leftOperand;    // Left operand
    Node_t      *rightOperand;   // Right operand
};

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

// UNARY NODE

struct UnaryNode_t : public Node_t {
    Node_t      *operand;    // Operand
};

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

// DATE RANGE

struct DateRange_t {
    string      fromDate;    // From-date
    string      toDate;      // To-date

    // Comparison operator

    bool operator < (const DateRange_t &r) const
        { return fromDate < r.fromDate; }
};

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

// DATE SET

typedef set<DateRange_t> DateSet_t;
                    // Date set type defintion
typedef DateSet_t::iterator DateIter_t;
                    // Date range iterator type definition
typedef DateSet_t::const_iterator DateCIter_t;
                    // Const date range iterator type def

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

// MINIMUM AND MAXIMUM DATES

static const string MIN_DATE = "0000-01-01";
                    // Minimum possible date
static const string MAX_DATE = "9999-12-31";
                    // Maximum possible date

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

// LIST A SET FOR DEBUGGING

string
ListSet (
    const DateSet_t &s)      // The date set
{
    ostringstream   oss;         // Output string stream
    DateCIter_t i;       // Date range iterator

    for (i = s.begin(); i != s.end(); i++) {
        if (i != s.begin()) oss << ", ";
```

```
        oss << i->fromDate << ".." << i->toDate;
    }
    return oss.str();
}

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

// DETERMINE THE UNION OF TWO DATE SETS

DateSet_t
DateSetUnion (
    const DateSet_t &s1,          // The first date set
    const DateSet_t &s2)          // The second date set
{
    DateCIter_t i1;      // Iterator in first set
    DateCIter_t i2;      // Iterator in second set
    DateSet_t   result;      // The result set
    DateRange_t r;       // An individual date range

    // Scan both sets until we get to the end of the sets

    i1 = s1.begin();
    i2 = s2.begin();
    while (i1 != s1.end() || i2 != s2.end()) {

        // Seed the next date range from the earlier of the
        // next date ranges in the two sets

        if (i1 != s1.end() && i2 != s2.end()) {
            if (i1->fromDate <= i2->fromDate) {
                r = *i1;
                i1 ++ ;
            } else {
                r = *i2;
                i2 ++ ;
            }
        } else if (i1 != s1.end()) {
            r = *i1;
            i1 ++ ;
        } else {
            r = *i2;
            i2 ++ ;
        }

        // Consolidate any overlapping or abutting date ranges

        while (
            (i1 != s1.end() && i1->fromDate <= r.toDate) ||
            (i2 != s2.end() && i2->fromDate <= r.toDate)
        ) {
            if (i1 != s1.end() && i1->fromDate <= r.toDate) {
                if (i1->toDate > r.toDate)
                    r.toDate = i1->toDate;
                i1 ++ ;
            } else {
                if (i2->toDate > r.toDate)
                    r.toDate = i2->toDate;
                i2 ++ ;
            }
```

```
        }

        // Add the date range to the result set

        result.insert (r);
    }
    return result;
}

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

// DETERMINE THE INTERSECTION OF TWO DATE SETS

DateSet_t
DateSetIntersect (
    const DateSet_t &s1,          // The first date set
    const DateSet_t &s2)          // The second date set
{
    DateCIter_t i1;      // Iterator in first set
    DateCIter_t i2;      // Iterator in second set
    DateSet_t   result;      // The result set
    DateRange_t r;       // An individual date range

    // Scan both sets until we get to the end of the sets

    i1 = s1.begin();
    i2 = s2.begin();
    while (i1 != s1.end() && i2 != s2.end()) {

        // If the first range intersects the second range
        // we can emit a result

        if (i1->toDate >= i2->fromDate && i1->fromDate <= i2->toDate) {
            if (i1->fromDate > i2->fromDate)
                r.fromDate = i1->fromDate;
            else
                r.fromDate = i2->fromDate;
            if (i1->toDate < i2->toDate)
                r.toDate = i1->toDate;
            else
                r.toDate = i2->toDate;
            result.insert (r);
        }

        // Skip the earlier range

        if (i1->toDate < i2->toDate)
            i1 ++ ;
        else
            i2 ++ ;
    }
    return result;
}

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

// RETURN THE PREVIOUS DATE

string
```

```
PrevDate (
    const string    &date)      // Date string
{
    string      result;     // Result date
    char        sqlDate[20];    // SQL date buffer
    Dtime_t     dtime;      // Double time value

    dtime = DtimeFromSql (date.c_str());
    if (dtime == NULL_DTIME) {
        cerr << "Error: invalid time\n";
        abort ();
    }
    dtime -= 24.0*60.0*60.0;
    return string(SqlDateFromDtime (sqlDate, dtime));
}

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

// RETURN THE NEXT DATE

string
NextDate (
    const string    &date)      // Date string
{
    string      result;     // Result date
    char        sqlDate[20];    // SQL date buffer
    Dtime_t     dtime;      // Double time value

    dtime = DtimeFromSql (date.c_str());
    if (dtime == NULL_DTIME) {
        cerr << "Error: invalid time\n";
        abort ();
    }
    dtime += 24.0*60.0*60.0;
    return string(SqlDateFromDtime (sqlDate, dtime));
}

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

// DETERMINE THE COMPLEMENT OF A DATE SET

DateSet_t
DateSetComplement (
    const DateSet_t &s)     // The date set
{
    DateCIter_t i1;     // Iterator in set
    DateCIter_t i2;     // Another iterator in set
    DateSet_t   result;     // The result set
    DateRange_t r;      // An individual date range

    // If the set is empty, the range is the full date range

    if (s.begin() == s.end()) {
        r.fromDate = MIN_DATE;
        r.toDate = MAX_DATE;
        result.insert (r);
    }

    // Non-empty sets
```

```
    else {
        i1 = s.begin();

        // Create a date range for the dates before the
        // first date in the set

        if (i1->fromDate > MIN_DATE) {
            r.fromDate = MIN_DATE;
            r.toDate = PrevDate (i1->fromDate);
            result.insert (r);
        }

        // Create date ranges from each gap

        i2 = i1;
        i1 ++ ;
        while (i1 != s.end()) {
            r.fromDate = NextDate(i2->toDate);
            r.toDate = PrevDate(i1->fromDate);
            result.insert (r);
            i2 = i1;
            i1 ++ ;
        }

        // Create a date range for the dates after the last date
        // in the set

        if (i2->toDate < MAX_DATE) {
            r.fromDate = NextDate(i2->toDate);
            r.toDate = MAX_DATE;
            result.insert (r);
        }
    }
    return result;
}

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

// EVALUATE THE EXPRESSION

DateSet_t
Evaluate (
    const Node_t    *node,       // Expression node
    bool        inverted)    // Result is to be inverted flag
{
    DateRange_t r;       // Date range element
    DateSet_t   result;      // Result
    const CompareNode_t *compareNode;
                    // Comparison node pointer
    const BinaryNode_t *binaryNode;
                    // Binary node pointer
    const UnaryNode_t *unaryNode;
                    // Unary node pointer

    switch (node->operation) {

    case EQ:
    case NE:
```

```
case GT:
case GE:
case LT:
case LE:
    // Non-date comparisons select all dates or no dates
    // depending on whether the result is to be inverted or not

    compareNode = static_cast<const CompareNode_t*>(node);
    if (compareNode->id != "date") {
        if ( !inverted) {
            r.fromDate = MIN_DATE;
            r.toDate = MAX_DATE;
            result.insert (r);
        } else {
            result.clear ();
        }
    }

    // Convert date expressions into the corresponding date set

    else {
        switch (node->operation) {
        case EQ:
            r.fromDate = compareNode->value;
            r.toDate = compareNode->value;
            result.insert (r);
            break;
        case NE:
            if (compareNode->value > MIN_DATE) {
                r.fromDate = MIN_DATE;
                r.toDate = PrevDate(compareNode->value);
                result.insert (r);
            }
            if (compareNode->value < MAX_DATE) {
                r.fromDate=NextDate(compareNode->value);
                r.toDate = MAX_DATE;
                result.insert (r);
            }
            break;
        case GT:
            if (compareNode->value < MAX_DATE) {
                r.fromDate=NextDate(compareNode->value);
                r.toDate = MAX_DATE;
                result.insert (r);
            }
            break;
        case GE:
            r.fromDate = compareNode->value;
            r.toDate = MAX_DATE;
            result.insert (r);
            break;
        case LT:
            if (compareNode->value > MIN_DATE) {
                r.fromDate = MIN_DATE;
                r.toDate = PrevDate(compareNode->value);
                result.insert (r);
            }
            break;
        case LE:
```

```
                r.fromDate = MIN_DATE;
                r.toDate = compareNode->value;
                result.insert (r);
                break;
            default:
                cerr << "Error: bad operation\n";
                abort ();
            }
        }
        break;

    // Logical AND operations result in the intersection of the date sets
    // of the left and right branches

    case AND:
        binaryNode = static_cast<const BinaryNode_t*>(node);
        result =  DateSetIntersect (
            Evaluate (binaryNode->leftOperand, inverted),
            Evaluate (binaryNode->rightOperand, inverted)
        );
        break;

    // Logical OR operations result in the union of the date sets
    // of the left and right branches

    case OR:
        binaryNode = static_cast<const BinaryNode_t*>(node);
        result =  DateSetUnion (
            Evaluate (binaryNode->leftOperand, inverted),
            Evaluate (binaryNode->rightOperand, inverted)
        );
        break;

    // Logical NOT operations result in the complement of the date set

    case NOT:
        unaryNode = static_cast<const UnaryNode_t*>(node);
        result =  DateSetComplement (
            Evaluate (unaryNode->operand, !inverted));
        break;

    // Other operations are unexpected

    default:
        cerr << "Error: unexpected operation\n";
        abort ();
    }
    return result;
}

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

// DETERMINE IF A REPORT IS PRELIMINARY

bool
Prelim (
    const Node_t    *node,      // Selection expression node
    const string    &commitDate)    // Commit date
{
```

```
    DateSet_t   selectedSet;     // Selected date set
    DateSet_t   prelimSet;   // Preliminary date set
    DateSet_t   interSet;    // Intersection set
    DateRange_t r;       // Date range

    // Determine the selected date set

    selectedSet = Evaluate (node, 0);

    // Load the preliminary date set

    r.fromDate = NextDate(commitDate);
    r.toDate = MAX_DATE;
    prelimSet.insert (r);

    // If the intersection of the selected and preliminary dates
    // is not empty, the report is preliminary

    interSet = DateSetIntersect (selectedSet, prelimSet);
    return interSet.begin() != interSet.end();
}

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

// TESTING MAIN LINE

int
main ()
{
    // Exercise the first example

    {
        CompareNode_t geNode;
        geNode.operation = GE;
        geNode.id = "date";
        geNode.value = "2011-10-18";

        CompareNode_t leNode;
        leNode.operation = LE;
        leNode.id = "date";
        leNode.value = "2011-10-20";

        BinaryNode_t andNode;
        andNode.operation = AND;
        andNode.leftOperand = &geNode;
        andNode.rightOperand = &leNode;

        CompareNode_t eqNode;
        eqNode.operation = EQ;
        eqNode.id = "colour";
        eqNode.value = "green";

        BinaryNode_t rootNode;
        rootNode.operation = AND;
        rootNode.leftOperand = &andNode;
        rootNode.rightOperand = &eqNode;

        if (Prelim (&rootNode, "2011-10-19"))
            cout << "Preliminary\n";
```

```
        else
            cout << "Final\n";
    }

    // Check complement operations

    {
        CompareNode_t geNode;
        geNode.operation = LT;
        geNode.id = "date";
        geNode.value = "2011-10-18";

        CompareNode_t leNode;
        leNode.operation = GT;
        leNode.id = "date";
        leNode.value = "2011-10-20";

        BinaryNode_t andNode;
        andNode.operation = OR;
        andNode.leftOperand = &geNode;
        andNode.rightOperand = &leNode;

        CompareNode_t eqNode;
        eqNode.operation = NE;
        eqNode.id = "colour";
        eqNode.value = "green";

        BinaryNode_t and2Node;
        and2Node.operation = OR;
        and2Node.leftOperand = &andNode;
        and2Node.rightOperand = &eqNode;

        UnaryNode_t rootNode;
        rootNode.operation = NOT;
        rootNode.operand = &and2Node;

        if (Prelim (&rootNode, "2011-10-19"))
            cout << "Preliminary\n";
        else
            cout << "Final\n";
    }

    return 0;
}
```