```cpp
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <vector>
#include <stack>
#include <map>
using namespace std;

#include "variable.h"
#include "constant.h"
#include "operator.h"

/**
 * This program parses an input string that represents an arithmetic expression,
 * converting it into an expression tree. This tree can then be evaluated to
 * compute the result.
 *
 * @author Robert C. Duvall
 */

// local constants
const string TITLE = "Arithmetica";
const string VERSION = "2.0";
const string PROMPT = "-";
const string COMMENT = ";";
const string EXIT = ".";
const string PLUS = "+";
const string MINUS = "-";
const string TIMES = "*";
const string DIVIDE = "/";
const string MOD = "%";
const string POWER = "^";
const string NEGATION = "~";
const string ASSIGNMENT = "=">
const int OPERAND = -1;

// local global data structures :
vector<string> thePrecedenceTable;
map<string, string> thePossibleOperators;
map<string, Expression*> theMemory;

/**
 * Initialize possible operators that can be recognized in expressions.
 * BUGBUG: probably should read this from a file now
 */
void setOperators ()
{
    // operators
    thePossibleOperators[PLUS] = "+";
    thePossibleOperators[MINUS] = "-";
    thePossibleOperators[TIMES] = "*";
    thePossibleOperators[DIVIDE] = "/";
    thePossibleOperators[MOD] = "%";
    thePossibleOperators[POWER] = "^";
    thePossibleOperators[NEGATION] = "~";
    thePossibleOperators[ASSIGNMENT] = "=";
    thePossibleOperators[GROUP] = "()";

    thePrecedenceTable.push_back(thePossibleOperators[GROUP]);
    thePrecedenceTable.push_back(" " + thePossibleOperators[ASSIGNMENT] + " ");
}
```
thePrecedenceTable.push_back(" " + thePossibleOperators[PLUS] + " " + thePossibleOperators[MINUS] + " ");
thePrecedenceTable.push_back(" " + thePossibleOperators[TIMES] + " " + thePossibleOperators[DIVIDE] + " " + thePossibleOperators[MOD] + " ");
thePrecedenceTable.push_back(" " + thePossibleOperators[POWER] + " ");
thePrecedenceTable.push_back(" " + thePossibleOperators[NEGATION] + " ");

/*
// alternate syntax
thePossibleOperators[PLUS] = "plus";
thePossibleOperators[MINUS] = "minus";
thePossibleOperators[TIMES] = "x";
thePossibleOperators[DIVIDE] = "divide";
thePossibleOperators[MOD] = "mod";
thePossibleOperators[POWER] = "pow";
thePossibleOperators[NEGATION] = "!";
thePossibleOperators[ASSIGNMENT] = ":=";
thePossibleOperators[GROUP] = " BEGIN END ";

// alternate precedences
thePrecedenceTable.push_back(thePossibleOperators[GROUP]);
thePrecedenceTable.push_back(" " + thePossibleOperators[ASSIGNMENT] + " ");
thePrecedenceTable.push_back(" " + thePossibleOperators[TIMES] + " " + thePossibleOperators[DIVIDE] + " " + thePossibleOperators[MOD] + " ");
thePrecedenceTable.push_back(" " + thePossibleOperators[PLUS] + " " + thePossibleOperators[MINUS] + " ");
thePrecedenceTable.push_back(" " + thePossibleOperators[POWER] + " ");
thePrecedenceTable.push_back(" " + thePossibleOperators[NEGATION] + " ");*/

/**
 * Print an opening remark to user.
 * BUGBUG: this should probably be read from a file.
 */
void printGreeting ()
{
    cout << "Welcome to " << TITLE << " " << VERSION << "!
    << endl << endl;
}

/**
 * Print a closing remark to user.
 * BUGBUG: this should probably be read from a file.
 */
void printClosing ()
{
    cout << endl << "Goodbye." << endl;
}

/**
 * @return true iff str contains a comment or nothing but whitespace
 */
bool isEmpty (const string& str)
{
    return (str.find(COMMENT) == 0) ||
            (count_if(str.begin(), str.end(), ::isspace) == int(str.length()));}
/**
 * @return true iff str contains only exit command
 */
bool isExit (const string & str)
{
    return (str == EXIT);
}

/**
 * @param op operator to test as beginning of group
 * @return true iff op signifies the beginning of a grouped expression
 */
bool isOpenGroup (const string& op)
{
    return (op == thePossibleOperators[GROUP].substr(1, op.length()));
}

/**
 * @param op operator to test for grouper
 * @return true iff op signifies the beginning or end of a grouped expression
 */
bool isGrouping (const string& op)
{
    return (thePossibleOperators[GROUP].find(" "+op+" ") != string::npos);
}

/**
 * @return relative importance of given op
 * Note, higher precedence indicates that op should be performed earlier
 */
int orderOfOperation (const string& op)
{
    for (unsigned int k = 0; k < thePrecedenceTable.size(); k++)
    {
        if (thePrecedenceTable[k].find(" "+op+" ") != string::npos)
        {
            return k;
        }
    }
    return OPERAND;
}

/**
 * Converts token into an expression by getting its arguments from operands.
 * @param token string representing the name of the operator to create
 * @param operands stack of sub-expressions available to be operands to new expression
 * @return expression representing the given token
 */
Expression* makeExpression (const string& token, stack<Expression*>& operands)
{
    // deal with unary operator
    Expression * rhs = operands.top();  operands.pop();
    if (token == thePossibleOperators[NEGATION])    return new Negate(rhs);

    // now deal with binary operators
    Expression * lhs = operands.top();   operands.pop();
    if (token == thePossibleOperators[PLUS])    return new Plus(lhs, rhs);
    else if (token == thePossibleOperators[MINUS])    return new Minus(lhs, rhs);
    else if (token == thePossibleOperators[TIMES])    return new Times(lhs, rhs);
    else if (token == thePossibleOperators[DIVIDE])    return new Divide(lhs, rhs);
else if (token == thePossibleOperators[MOD]) return new Mod(lhs, rhs);
else if (token == thePossibleOperators[POWER]) return new Power(lhs, rhs);
else if (token == thePossibleOperators[ASSIGNMENT])
{
    // link variable name to expression
    theMemory[lhs->toString()] = rhs;
    return lhs;
}
else
{
    cerr << "Parse Error: unrecognized operator " << token << endl;
    return 0;
}

/**
 * This implements Dijkstra’s algorithm for converting infix to postfix.
 * This algorithm iterates through the infix expression: as operands are
 * found, they are converted into expressions and put onto a stack of
 * expressions; as operators are found, all operators that have higher
 * precedence (i.e., should be evaluated first) are popped off the stack
 * and converted into expressions by getting their left and right from the
 * operand stack. They are then pushed onto the operand stack themselves
 * to become the left or right side of a larger expression. When complete,
 * the stack should contain one expression that represents all the combined
 * sub-expressions.
 * By giving grouping operators a lower precedence than anything else and
 * creating a special case that prevents group opening operators from popping
 * previous operators off the stack, it gives the effect of boosting the
 * precedence of those operators included in the group.
 * @param infix a space separated expression given in infix order
 * @return tree is created that represents the given expression
 */
Expression* parse (const string& infix)
{
    stack<string> operators;
    stack<Expression*> operands;

    istringstream in(infix);
    string token;
    while (in >> token)
    {
        int order = orderOfOperation(token);
        // convert operand directly to expression
        if (order == OPERAND)
        {
            int value;
            istringstream extractor(token);
            if (extractor >> value)
            {
                operands.push( new Constant(value) );
            }
            else
            {
                // anything not a number is treated as a variable
                operands.push( new Variable(token, theMemory) );
            }
        }
        // convert operator by popping all previous operators of higher order,
else
{
    // special case: do not pop any operators when beginning parentheses
    if (!isOpenGroup(token))
    {
        while (operators.size() > 0 &&
            orderOfOperation(operators.top()) >= order)
        {
            if (!isGrouping(operators.top()))
            {
                operands.push(makeExpression(operators.top(), operands))
            }
            operators.pop();
        }
        operators.push(token);
    }
}

// pop remaining operators
while (operators.size() > 0)
{
    if (!isGrouping(operators.top()))
    {
        operands.push(makeExpression(operators.top(), operands));
    }
    operators.pop();
}
return operands.top();

/**
 * Repeatedly prompts for expressions to evaluate from the given input stream.
 * @param input either the console or a file of expressions to interpret
 */
void interpret (istream& input)
{
    int numExpressions = 1;
    cout << numExpressions << PROMPT;
    string line;
    while (input.good())
    {
        if (getline(input, line) && ! isEmpty(line))
        {
            if (isExit(line))
            {
                break;
            }
            else
            {
                Expression* expression = parse(line);
                cout << *expression << "\"" << expression->evaluate() << endl;
            }
            numExpressions++;
            cout << numExpressions << PROMPT;
        }
    }
}
int main (int argc, char* argv[]) {
    // by default, read from console;
    // otherwise read from given file
    istream* input = &cin;
    if (argc > 1) {
        input = new ifstream(argv[1]);
        if (input->fail()) {
            cerr << "ERROR: unable to open input file " << argv[1] << endl;
            return 0;
        }
    }
    setOperators();
    printGreeting();
    interpret(*input);
    printClosing();
    return 0;
}
#ifndef CONSTANT_INCLUDED
#define CONSTANT_INCLUDED

#include <string>
#include <sstream>
using namespace std;
#include "expression.h"

/**
 * An Expression that represents a single value.
 * @author Robert C. Duvall
 */
class Constant : public Expression
{
  public:
  /**
   * Make a default constant value
   */
  Constant ()
  {}

  /**
   * Make a constant from a value
   */
  Constant (int value)
    : myValue(value)
  {}

  /**
   * Clean up an expression
   */
  virtual ~Constant ()
  {}

  /**
   * Allow a value to be assigned to an expression
   */
  Constant& operator= (int value)
  {
    myValue = value;
    return *this;
  }

  /**
   * Evaluate the expression
   */
  virtual int evaluate () const
  {
    return myValue;
  }

  /**
   * Print the expression as an infix expression
   */
  virtual string toString () const
  {
    ostringstream out;
    return out.str();
  }
};
out << myValue;
    return out.str();
  }

private:

  int myValue;
};

#ifndef CONSTANT_INCLUDED
// CONSTANT_INCLUDED

#endif
#ifndef DEBUG_INCLUDED
#define DEBUG_INCLUDED

  // #define DEBUG

#define DEBUGOUT(s) cerr << s << endl;
#else
#define DEBUGOUT(s)
#endif
#endif // DEBUG_INCLUDED
#ifndef EXPRESSION_INCLUDED
#define EXPRESSION_INCLUDED
#include <string>
#include <iostream>
using namespace std;

/**
 * An Expression represents a mathmatical expression as a tree.
 * In this design, the internal nodes represent arithmetic functions
 * and the leaves represent constant values.
 *
 * @author Robert C. Duvall
 */
class Expression
{
    public:
        /**
         * Clean up an expression
         */
        virtual ~Expression ()
        {

        }

        /**
         * Evaluate the expression
         *
         * @return integer value of expression
         */
        virtual int  evaluate () const = 0;

        /**
         * Compare two expressions by the value they evaluate to
         *
         * @return 0 if evaluates to equal, negative if less, positive if greater
         */
        virtual int  compare (const Expression& other) const
        {
            return evaluate() - other.evaluate();
        }

        /**
         * Print the expression as an infix expression
         */
        virtual string toString () const = 0;
};

// output operator
inline ostream& operator << (ostream & out, const Expression& e)
{
    out << e.toString();
    return out;
}

#endif // EXPRESSION_INCLUDED
```cpp
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <vector>
#include <stack>
#include <map>
using namespace std;
#include "variable.h"
#include "constant.h"
#include "operator.h"

/**
 * This program parses an input string that represents an arithmetic expression,
 * converting it into an expression tree. This tree can then be evaluated to
 * compute the result.
 *
 * @author Robert C. Duvall
 */

// local constants
const string TITLE = "Arithmetica";
const string VERSION = "2.0";
const string PROMPT = "−>";
const string COMMENT = ";#";
const string EXIT = ";.";
const string PLUS = "+";
const string MINUS = "−";
const string TIMES = "*";
const string DIVIDE = "/";
const string MOD = "%";
const string POWER = "^";
const string NEGATION = "~";
const string ASSIGNMENT = "=";
const string GROUP = "()";
const int OPERAND = −1;

// local global data structures :
vector<string> thePrecedenceTable;
map<string, string> thePossibleOperators;
map<string, Expression*> theMemory;

/**
 * Initialize possible operators that can be recognized in expressions.
 * BUGBUG: probably should read this from a file now
 */
void setOperators ()
{
    // operators
    thePossibleOperators[PLUS] = "+";
    thePossibleOperators[MINUS] = "−";
    thePossibleOperators[TIMES] = "*";
    thePossibleOperators[DIVIDE] = "/";
    thePossibleOperators[MOD] = "%";
    thePossibleOperators[POWER] = "^";
    thePossibleOperators[NEGATION] = "~";
    thePossibleOperators[ASSIGNMENT] = "=";
    thePossibleOperators[GROUP] = "()";

    thePrecedenceTable.push_back(thePossibleOperators[GROUP]);
    thePrecedenceTable.push_back(" + thePossibleOperators[ASSIGNMENT] + ");
}
```
thePrecedenceTable.push_back(" " + thePossibleOperators[PLUS] + " " + thePossibleOperators[MINUS] + " ");
thePrecedenceTable.push_back(" " + thePossibleOperators[TIMES] + " " + thePossibleOperators[DIVIDE] + " " + thePossibleOperators[MOD] + " ");
thePrecedenceTable.push_back(" " + thePossibleOperators[POWER] + " ");
thePrecedenceTable.push_back(" " + thePossibleOperators[NEGATION] + " ");

/*
   // alternate syntax
   thePossibleOperators[PLUS] = "plus";
   thePossibleOperators[MINUS] = "minus";
   thePossibleOperators[TIMES] = "x";
   thePossibleOperators[DIVIDE] = "divide";
   thePossibleOperators[MOD] = "mod";
   thePossibleOperators[POWER] = "pow";
   thePossibleOperators[NEGATION] = "!";
   thePossibleOperators[ASSIGNMENT] = ":=";
   thePossibleOperators[GROUP] = " BEGIN END ";

   // alternate precedences
   thePrecedenceTable.push_back(thePossibleOperators[GROUP]);
   thePrecedenceTable.push_back(" " + thePossibleOperators[ASSIGNMENT] + " ");
   thePrecedenceTable.push_back(" " + thePossibleOperators[TIMES] + " " + thePossibleOperators[DIVIDE] + " " + thePossibleOperators[MOD] + " ");
   thePrecedenceTable.push_back(" " + thePossibleOperators[PLUS] + " " + thePossibleOperators[MINUS] + " ");
   thePrecedenceTable.push_back(" " + thePossibleOperators[POWER] + " ");
   thePrecedenceTable.push_back(" " + thePossibleOperators[NEGATION] + " ");
   */
}

/**
 * Print an opening remark to user.
 * BUGBUG: this should probably be read from a file.
 */
void printGreeting ()
{
    cout << "Welcome to " << TITLE << " " << VERSION << "!" << endl << endl;
}

/**
 * Print a closing remark to user.
 * BUGBUG: this should probably be read from a file.
 */
void printClosing ()
{
    cout << endl << "Goodbye." << endl;
}

/**
 * @return true iff str contains a comment or nothing but whitespace
 */
bool isEmpty (const string& str)
{
    return (str.find(COMMENT) == 0) || (count_if(str.begin(), str.end(), ::isspace) == int(str.length()));
}
/**
 * @return true iff str contains only exit command
 */
bool isExit (const string & str)
{
    return (str == EXIT);
}

/**
 * @param op operator to test as beginning of group
 * @return true iff op signifies the beginning of a grouped expression
 */
bool isOpenGroup (const string& op)
{
    return (op == thePossibleOperators[GROUP].substr(1, op.length()));
}

/**
 * @param op operator to test for grouper
 * @return true iff op signifies the beginning or end of a grouped expression
 */
bool isGrouping (const string& op)
{
    return (thePossibleOperators[GROUP].find(" " + op + " ") != string::npos);
}

/**
 * @return relative importance of given op
 * Note, higher precedence indicates that op should be performed earlier
 */
int orderOfOperation (const string& op)
{
    for (unsigned int k = 0; k < thePrecedenceTable.size(); k++)
    {
        if (thePrecedenceTable[k].find(" " + op + " ") != string::npos)
        {
            return k;
        }
    }
    return OPERAND;
}

/**
 * Converts token into an expression by getting its arguments from operands.
 * @param token string representing the name of the operator to create
 * @param operands stack of sub-expressions available to be operands to new expression
 * @return expression representing the given token
 */
Expression* makeExpression (const string& token, stack<Expression*>& operands)
{
    // deal with unary operator
    Expression * rhs = operands.top();  operands.pop();
    if (token == thePossibleOperators[NEGATION]) return new Negate(rhs);

    // now deal with binary operators
    Expression * lhs = operands.top();   operands.pop();
    if (token == thePossibleOperators[PLUS]) return new Plus(lhs, rhs);
    else if (token == thePossibleOperators[MINUS]) return new Minus(lhs, rhs);
    else if (token == thePossibleOperators[TIMES]) return new Times(lhs, rhs);
    else if (token == thePossibleOperators[DIVIDE]) return new Divide(lhs, rhs);
else if (token == thePossible Operators[MOD]) return new Mod(lhs, rhs);
else if (token == thePossible Operators[POWER]) return new Power(lhs, rhs);
else if (token == thePossible Operators[ASSIGNMENT])
{
    // link variable name to expression
    theMemory[lhs->toString()] = rhs;
    return lhs;
}
else {
    cerr << "Parse Error: unrecognized operator " << token << endl;
    return 0;
}

/**
* This implements Dijkstra’s algorithm for converting infix to postfix.
* This algorithm iterates through the infix expression: as operands are
* found, they are converted into expressions and put onto a stack of
* expressions; as operators are found, all operators that have higher
* precedence (i.e., should be evaluated first) are popped off the stack
* and converted into expressions by getting their left and right from the
* operand stack. They are then pushed onto the operand stack themselves
* to become the left or right side of a larger expression. When complete,
* the stack should contain one expression that represents all the combined
* sub-expressions.
* By giving grouping operators a lower precedence than anything else and
* creating a special case that prevents group opening operators from popping
* previous operators off the stack, it gives the effect of boosting the
* precedence of those operators included in the group.
* @param infix a space separated expression given in infix order
* @return tree is created that represents the given expression
*/
Expression* parse (const string &infix)
{
    stack<string> operators;
    stack<Expression*> operands;
    istringstream in(infix);
    string token;
    while (in >> token)
    {
        int order = orderOfOperation(token);
        // convert operand directly to expression
        if (order == OPERAND)
        {
            int value;
            istringstream extractor(token);
            if (extractor >> value)
            {
                operands.push(new Constant(value));
            }
            else
            {
                // anything not a number is treated as a variable
                operands.push(new Variable(token, theMemory));
            }
        }
        // convert operator by popping all previous operators of higher order,
else
{
    // special case: do not pop any operators when beginning parentheses
    if (!isOpenGroup(token))
    {
        while (operators.size() > 0 &&
               orderOfOperation(operators.top()) >= order)
        {
            if (!isGrouping(operators.top()))
            {
                operands.push(makeExpression(operators.top(), operands));
            }
            operators.pop();
        }
        operators.push(token);
    }
}

// pop remaining operators
while (operators.size() > 0)
{
    if (!isGrouping(operators.top()))
    {
        operands.push(makeExpression(operators.top(), operands));
    }
    operators.pop();
}

return operands.top();

/**
 * Repeatedly prompts for expressions to evaluate from the given input stream.
 * @param input either the console or a file of expressions to interpret
 */
void interpret (istream& input)
{
    int numExpressions = 1;
    cout << numExpressions << PROMPT;
    string line;
    while (input.good())
    {
        if (getline(input, line) && !isEmpty(line))
        {
            if (isExit(line))
            {
                break;
            }
            else
            {
                Expression* expression = parse(line);
                cout << *expression << " = " << expression->evaluate() << endl;
                numExpressions++;
                cout << numExpressions << PROMPT;
            }
        }
    }
}
int main (int argc, char* argv[]) {
    // by default, read from console;
    // otherwise read from given file
    istream* input = &cin;
    if (argc > 1) {
        input = new ifstream(argv[1]);
        if (input->fail()) {
            cerr << "ERROR: unable to open input file " << argv[1] << endl;
            return 0;
        }
    }
    setOperators();
    printGreeting();
    interpret(*input);
    printClosing();
    return 0;
}
#ifndef OPERATOR_INCLUDED
#define OPERATOR_INCLUDED
#include <string>
#include <cmath>
using namespace std;
#include "expression.h"

/**
 * An Expression that represents a mathematical operation on a set of arguments.
 * @author Robert C. Duvall
 */
class Operator : public Expression
{
    public:
    /**
     * Make an expression as a combination of subexpressions
     */
    Operator (const string& description, Expression* lhs, Expression* rhs)
        : myDescription(description),
          myLHS(lhs),
          myRHS(rhs)
    {}

    /**
     * Clean up an expression
     */
    virtual ~Operator ()
    {
        delete myLHS; myLHS = 0;
        delete myRHS; myRHS = 0;
    }

    /**
     * Print the expression as an infix expression
     */
    virtual string toString () const
    {
        return "( + myLHS->toString() + " + myDescription + " + myRHS->toString() + ")";
    }

    protected:
    string myDescription;
    Expression* myLHS;
    Expression* myRHS;
};

/**
 * An Expression that adds its arguments.
 * @author Robert C. Duvall
 */
class Plus : public Operator
{
    public:
    /**
     * Make an expression as a combination of subexpressions
     */
Plus (Expression* lhs, Expression* rhs)
  : Operator("+", lhs, rhs)
{}

/**
 * Evaluate the expression
 */
virtual int evaluate () const
{
    return myLHS->evaluate() + myRHS->evaluate();
}
};

/**
 * An Expression that subtracts its arguments.
 * @author Robert C. Duvall
 */
class Minus : public Operator
{
public:
/**
 * Make an expression as a combination of subexpressions
 */
Minus (Expression* lhs, Expression* rhs)
  : Operator("-", lhs, rhs)
{}

/**
 * Evaluate the expression
 */
virtual int evaluate () const
{
    return myLHS->evaluate() - myRHS->evaluate();
}
};

/**
 * An Expression that multiplies its arguments.
 * @author Robert C. Duvall
 */
class Times : public Operator
{
public:
/**
 * Make an expression as a combination of subexpressions
 */
Times (Expression* lhs, Expression* rhs)
  : Operator("*", lhs, rhs)
{}

/**
 * Evaluate the expression
 */
virtual int evaluate () const
{
    return myLHS->evaluate() * myRHS->evaluate();
}
};
class Divide : public Operator
{
    public:
        /**
         * Make an expression as a combination of subexpressions
         */
        Divide (Expression* lhs, Expression* rhs)
            : Operator("/", lhs, rhs)
        {}

        /**
         * Evaluate the expression
         */
        virtual int evaluate () const
        {
            return myLHS->evaluate() / myRHS->evaluate();
        }
};

class Mod : public Operator
{
    public:
        /**
         * Make an expression as a combination of subexpressions
         */
        Mod (Expression* lhs, Expression* rhs)
            : Operator("%", lhs, rhs)
        {}

        /**
         * Evaluate the expression
         */
        virtual int evaluate () const
        {
            return myLHS->evaluate() % myRHS->evaluate();
        }
};

class Power : public Operator
{
    public:
        /**
         * Make an expression as a combination of subexpressions
         */
        Power (Expression* lhs, Expression* rhs)
            : Operator("^", lhs, rhs)
        {
virtual int  evaluate () const
{
    return int(pow(double(myLHS->evaluate()), myRHS->evaluate()));
}

class Negate : public Expression
{
public:
    Negate (Expression* expr)
    : myDescription("~"),
        myExpr(expr)
    {}

    virtual int  evaluate () const
    {
        return -myExpr->evaluate();
    }

    virtual string toString () const
    {
        return myDescription + " " + myExpr->toString();
    }

protected:
    string myDescription;
    Expression* myExpr;
};

#endif // OPERATOR_INCLUDED
```cpp
#ifndef VARIABLE_INCLUDED
#define VARIABLE_INCLUDED
#include <string>
#include <map>
using namespace std;
#include "expression.h"

/**
 * An Expression that represents a variable.
 *
 * @author Robert C. Duvall
 */
class Variable : public Expression
{
public:
    /**
     * Make a variable from a string
     *
     * @param name
     * @param memory
     */
    Variable (const string& name, const map<string, Expression*>& memory)
        : myName(name),
          myMemory(memory)
    {}

    /**
     * Clean up an expression
     */
    virtual ~Variable ()
    {}

    /**
     * Evaluate the expression
     *
     * @return integer value of expression
     */
    virtual int evaluate () const
    {
        map<string, Expression*>::const_iterator place = myMemory.find(myName);
        if (place != myMemory.end())
            return place->second->evaluate();
        else
        {
            cout << "Parse Error: variable " << myName << " undefined." << endl;
            return 0;
        }
    }

    /**
     * Print the expression as an infix expression
     */
    virtual string toString () const
    {
        return myName;
    }

private:
    string myName;
    const map<string, Expression*>& myMemory;
};
```
};

#error // VARIABLE_INCLUDED