matrix_expression.h

```
#ifndef MATRIX_EXPRESSION_H
#define MATRIX_EXPRESSION_H

#include <iostream>

class matrix;        // forward declaration to avoid circular includes
class matrix_proxy;  // forward declaration to avoid circular includes

/*********************************************************
 * matrix_expression class
 *
 * This is an "abstract" class in that it is impossible
 * to directly create instances of this class.  Its
 * purpose is to serve as a base class for concrete
 * implementations (e.g., matrix, matrix_proxy).
 *
 * As such, you cannot declare a standard value variable
 * of the form:
 *
 *   matrix_expression A;
 *
 * But can instead declare reference variables of this type, assigned
 * to expressions that evaluate to said type, as in
 *
 *   matrix_expression &A = B(range(2,5), range(3,8));
 *
 *********************************************************/
class matrix_expression {

 public:

  /*********************************************************
   *  The following two definitions are pure virtual functions
   *  that must be overridden by each child class.
   *********************************************************/

  // Provides read-only access via indexing operator
  virtual double operator()(int r, int c) const = 0;

  // Provides write-access through indexing operator
  virtual double& operator()(int r, int c)  = 0;

  // Returns the number of rows
  virtual int numRows() const =0;

  // Returns the number of columns
  virtual int numColumns() const =0;


  /*********************************************************
   *  The remaining definitions are functions that can be shared
   *  by all child classes (although they are free to override
   *  them if desired).
   *
   *  Generic implementations for these are provided in
   *  matrix_expression.cpp
   *********************************************************/

  // assignment operator supports syntax:  A = B;
  // For generic expressions, this is only well defined if dimensions agree.
  virtual matrix_expression& operator=(const matrix_expression& other);

  // Returns a 1x2 matrix describing the number of rows and columns respectively
  virtual matrix size() const;

  // element-wise test of equality.
  virtual bool operator==(const matrix_expression &other) const;

  // element-wise test of inequality
  virtual bool operator!=(const matrix_expression &other) const;

  // provides read-only access to a matrix entry based on linearized order
```

```
 73:   double operator[](int k) const;
 74:
 75:   // provides write access to a matrix entry based on linearized order
 76:   double& operator[](int k);
 77:
 78:   // provides read-only access to a submatrix via a proxy
 79:   virtual const matrix_proxy operator()(const matrix_expression& rows,
 80:                                         const matrix_expression& cols) const;
 81:
 82:   virtual
 83:   const matrix_proxy operator()(int r, const matrix_expression& cols) const;
 84:
 85:   virtual
 86:   const matrix_proxy operator()(const matrix_expression& rows, int c) const;
 87:
 88:   // provides write access to a submatrix via a proxy
 89:   virtual matrix_proxy operator()(const matrix_expression& rows,
 90:                                   const matrix_expression& cols);
 91:   virtual matrix_proxy operator()(int r, const matrix_expression& cols);
 92:   virtual matrix_proxy operator()(const matrix_expression& rows, int c);
 93:
 94:   //-----------------------------------------------
 95:   // addition
 96:   //-----------------------------------------------
 97:
 98:   // returns new matrix instance based on sum of two expressions
 99:   virtual matrix operator+(const matrix_expression& other) const;
100:
101:   // in-place addition with another matrix expression
102:   virtual matrix_expression& operator+=(const matrix_expression& other);
103:
104:   // returns new matrix instance based on element-wise addition
105:   virtual matrix operator+(double scalar) const;
106:
107:   // in-place element-wise addition with a scalar
108:   virtual matrix_expression& operator+=(double scalar);
109:
110:   //-----------------------------------------------
111:   // multiplication
112:   //-----------------------------------------------
113:
114:   // returns a new matrix based on element-wise multiplication by a scalar
115:   // e.g.,   C = A*B;
116:   virtual matrix operator*(double scalar) const;
117:
118:   // in-place element-wise multiplication with a scalar
119:   // e.g.,  C = A*4;
120:   virtual matrix_expression& operator*=(double scalar);
121:
122:   // returns a new matrix based on product of expressions
123:   virtual matrix operator*(const matrix_expression& other) const;
124:
125:   //-----------------------------------------------
126:   // destructor (a formality in this case)
127:   //-----------------------------------------------
128:   virtual ~matrix_expression() { }
129: };
130:
131:
132: // support for outputting a matrix expression
133: std::ostream& operator<<(std::ostream& out, const matrix_expression& m);
134:
135: // support for left-hand side scalar operations
136: matrix operator+(double scalar, const matrix_expression& m);
137: matrix operator*(double scalar, const matrix_expression& m);
138:
139: #endif
```

matrix_expression.cpp

```
 1: #include <stdexcept>            // defines various exceptions we throw
 2: #include <sstream>              // need stringstream for operator<<
 3: #include <iomanip>              // needed for operator<<
 4: #include "matrix_expression.h"
 5: #include "matrix.h"
 6: #include "matrix_proxy.h"
 7: using namespace std;
 8:
 9: /*************************************************
10:  * support for matrix_expression class
11:  *************************************************/
12:
13: // assignment operator supports syntax:  A = B;
14: // For generic expressions, this is only well defined if dimensions agree.
15: matrix_expression& matrix_expression::operator=(const matrix_expression& other) {
16:     if (numRows() != other.numRows() || numColumns() != other.numColumns())
17:         throw invalid_argument("Matrix dimensions must agree.");
18:
19:     for (int r=0; r < numRows(); r++)
20:         for (int c=0; c < numColumns(); c++)
21:             (*this)(r,c) = other(r,c);
22:
23:     return *this;
24: }
25:
26:
27: // Returns a 1x2 matrix describing the number of rows and columns respectively
28: matrix matrix_expression::size() const {
29:     matrix result(1,2);
30:     result(0,0) = numRows();
31:     result(0,1) = numColumns();
32:     return result;
33: }
34:
35:
36:
37: bool matrix_expression::operator==(const matrix_expression &other) const {
38:     if (numRows() != other.numRows() || numColumns() != other.numColumns())
39:         return false;    // not equivalent
40:
41:     for (int r=0; r < numRows(); r++)
42:         for (int c=0; c < numColumns(); c++)
43:             if ((*this)(r,c) != other(r,c))
44:                 return false;   // not equivalent
45:
46:     return true;   // by process of elimination, must be equivalent
47: }
48:
49: bool matrix_expression::operator!=(const matrix_expression &other) const {
50:     return !(*this == other);    // piggy-back on existing definition of ==
51: }
52:
53:
54: // provides read-only access to a matrix entry based on linearized order
55: double matrix_expression::operator[](int k) const {
56:     int c,r;
57:     r = k % numRows();
58:     c = k / numRows();
59:     return (*this)(r, c);
60: }
61:
62: // provides write access to a matrix entry based on linearized order
63: double& matrix_expression::operator[](int k) {
64:     int c,r;
65:     r = k % numRows();
66:     c = k / numRows();
67:     return (*this)(r, c);
68: }
69:
70: // provides read-only access to a submatrix via a proxy
71: const matrix_proxy matrix_expression::operator()(const matrix_expression& rows,
72:                                                  const matrix_expression& cols) const {
```

```
    return matrix_proxy(*const_cast<matrix_expression*>(this), rows, cols);
}

const matrix_proxy matrix_expression::operator()(int r, const matrix_expression& cols) con
st {
    return operator()(matrix(1,1,r), cols);
}

const matrix_proxy matrix_expression::operator()(const matrix_expression& rows, int c) con
st {
    return operator()(rows, matrix(1,1,c));
}


// provides write access to a submatrix as a proxy
matrix_proxy matrix_expression::operator()(const matrix_expression& rows,
                                           const matrix_expression& cols) {
    return matrix_proxy(*this, rows, cols);
}

matrix_proxy matrix_expression::operator()(int r, const matrix_expression& cols) {
    return operator()(matrix(1,1,r), cols);
}

matrix_proxy matrix_expression::operator()(const matrix_expression& rows, int c) {
    return operator()(rows, matrix(1,1,c));
}


//-----------------------------------------------
// addition
//-----------------------------------------------

// returns new matrix instance based on sum of two expressions
matrix matrix_expression::operator+(const matrix_expression& other) const {
    if (numRows() != other.numRows() || numColumns() != other.numColumns())
        throw invalid_argument("Matrix dimensions must agree.");

    matrix result(*this);        // a new matrix instance for result
    for (int r=0; r < numRows(); r++)
        for (int c=0; c < numColumns(); c++)
            result(r,c) += other(r,c);

    return result;
}

// in-place addition with another matrix expression
matrix_expression& matrix_expression::operator+=(const matrix_expression& other) {
    if (numRows() != other.numRows() || numColumns() != other.numColumns())
        throw invalid_argument("Matrix dimensions must agree.");

    for (int r=0; r < numRows(); r++)
        for (int c=0; c < numColumns(); c++)
            (*this)(r,c) += other(r,c);

    return *this;
}

// returns new matrix instance based on element-wise addition
matrix matrix_expression::operator+(double scalar) const {
    matrix result(*this);
    for (int r=0; r < numRows(); r++)
        for (int c=0; c < numColumns(); c++)
            result(r,c) += scalar;
    return result;
}

// in-place element-wise addition with a scalar
matrix_expression& matrix_expression::operator+=(double scalar) {

    for (int r=0; r < numRows(); r++)
```

```
  143:         for (int c=0; c < numColumns(); c++)
  144:             (*this)(r,c) += scalar;
  145:
  146:     return *this;
  147: }
  148:
  149:
  150: //-----------------------------------------------
  151: // multiplication
  152: //-----------------------------------------------
  153:
  154: // returns a new matrix based on element-wise multiplication by a scalar
  155: // e.g.,   C = A*B;
  156: matrix matrix_expression::operator*(double scalar) const {   // multiply each element by s
calar
  157:     matrix result = matrix(*this);
  158:     for (int r=0; r < numRows(); r++)
  159:         for (int c=0; c < numColumns(); c++)
  160:             result(r,c) *= scalar;
  161:     return result;
  162: }
  163:
  164: // in-place element-wise multiplication with a scalar
  165: // e.g.,  C = A*4;
  166: matrix_expression& matrix_expression::operator*=(double scalar) {
  167:     for (int r=0; r < numRows(); r++)
  168:         for (int c=0; c < numColumns(); c++)
  169:             (*this)(r,c) *= scalar;
  170:
  171:     return *this;
  172: }
  173:
  174:
  175: matrix matrix_expression::operator*(const matrix_expression& other) const {
  176:     if (numColumns() != other.numRows())
  177:         throw invalid_argument("Inner matrix dimensions must agree.");
  178:
  179:     matrix result = matrix(numRows(), other.numColumns());  // all zeros initially
  180:     for (int r=0; r < numRows(); r++)
  181:         for (int c=0; c < other.numColumns(); c++)
  182:             for(int k=0; k < numColumns(); k++)          // compute appropriate dot-produc
t
  183:                 result(r,c) += (*this)(r,k) * other(k,c);
  184:
  185:     return result;
  186: }
  187:
  188:
  189: //-----------------------------------------------
  190: // support for outputting a matrix expression
  191: //-----------------------------------------------
  192:
  193: ostream& operator<<(ostream& out, const matrix_expression& m) {
  194:     string temp;
  195:
  196:     unsigned int maxfield = 0;
  197:     for (int r=0; r < m.numRows(); r++) {
  198:         for (int c=0; c < m.numColumns(); c++) {
  199:             stringstream s;
  200:             s << fixed << setprecision(3);
  201:             s << m(r,c);
  202:             s >> temp;
  203:             if (temp.size() > maxfield)
  204:                 maxfield = temp.size();
  205:         }
  206:     }
  207:
  208:     for (int r=0; r < m.numRows(); r++) {
  209:         for (int c=0; c < m.numColumns(); c++) {
  210:             stringstream s;
  211:             s << fixed << setprecision(3);
  212:             s << m(r,c);
```

```
213:             s >> temp;
214:             out << " " << setw(maxfield) << temp;
215:         }
216:         out << endl;
217:     }
218:
219:     return out;
220: }
221:
222: //-----------------------------------------------
223: // support for scalar arithmetic as left-hand operand
224: //-----------------------------------------------
225: matrix operator+(double scalar, const matrix_expression& m) {
226:     return m + scalar;    // reverse order of operands to invoke class method
227: }
228:
229: matrix operator*(double scalar, const matrix_expression& m) {
230:     return m * scalar;    // reverse order of operands to invoke class method
231: }
```

matrix.h

```
 1: #ifndef MATRIX_H
 2: #define MATRIX_H
 3:
 4: #include <iostream>
 5: #include <vector>
 6: #include "matrix_expression.h"
 7:
 8:
 9: /*****************************************************
10:  * matrix class
11:  *****************************************************/
12:
13: class matrix : public matrix_expression {
14: private:
15:   int _nr;                    /* number of rows */
16:   int _nc;                    /* number of columns */
17:   std::vector<double> _data;  /* underlying data storage */
18:
19: public:
20:
21:   matrix(int numRows=0, int numColumns=0, double value=0);
22:
23:   // copy constructor based on an existing matrix_expression
24:   matrix(const matrix_expression& m);
25:
26:   // Returns the number of rows
27:   int numRows() const;
28:
29:   // Returns the number of columns
30:   int numColumns() const;
31:
32:   // Change the apparent size of this matrix.
33:   // Overall number of elements must be preserved.
34:   void reshape(int r, int c);
35:
36:   // we need following declaration to properly inherit the version of
37:   // operator() for submatrices.  Then we can define our additional ones.
38:   using matrix_expression::operator();
39:
40:   // Provides read-only access via indexing operator
41:   double operator()(int r, int c) const;
42:
43:   // Provides write-access through indexing operator
44:   double& operator()(int r, int c);
45:
46:   // assignment operator based on existing matrix_expression.
47:   // Note: current matrix will be resized if necessary
48:   matrix& operator=(const matrix_expression& m);
49:   matrix& operator=(const matrix& other);
50:
51: };
52:
53: //-------------------------------------------------------
54: // define additional support for reading a matrix
55: //-------------------------------------------------------
56: std::istream& operator>>(std::istream& in, matrix& m);
57:
58: #endif
```

matrix.cpp

```
 1: #include <stdexcept>            // defines various exceptions we throw
 2: #include <sstream>              // need stringstream for operator<<
 3: #include "matrix.h"
 4: using namespace std;
 5:
 6:
 7: /*************************************************
 8:  * matrix class
 9:  *************************************************/
10:
11: matrix::matrix(int numRows, int numColumns, double value)
12:     : _nr(numRows), _nc(numColumns), _data(numRows*numColumns, value) {}
13:
14:
15: matrix::matrix(const matrix_expression& m) {
16:     *this = m;   // rely on operator= implementation
17: }
18:
19: matrix& matrix::operator=(const matrix& other) {
20:     if (this != &other) {
21:         _nr = other._nr;
22:         _nc = other._nc;
23:         _data = other._data;
24:     }
25:     return *this;
26: }
27:
28: matrix& matrix::operator=(const matrix_expression& m) {
29:     if (this != &m) {
30:         _nr = m.numRows();
31:         _nc = m.numColumns();
32:         _data.resize(_nr * _nc);
33:         matrix_expression::operator=(m);
34:     }
35:     return *this;
36: }
37:
38:
39: // Returns the number of rows
40: int matrix::numRows() const {
41:     return _nr;
42: }
43:
44: // Returns the number of columns
45: int matrix::numColumns() const {
46:     return _nc;
47: }
48:
49: // Change the apparent size of this matrix.
50: // Overall number of elements must be preserved.
51: void matrix::reshape(int r, int c) {
52:     if (r * c != _nr * _nc)
53:         throw invalid_argument("To reshape, the number of elements must not change.");
54:
55:     _nr = r;
56:     _nc = c;
57: }
58:
59:
60: // Provides read-only access via indexing operator
61: double matrix::operator()(int r, int c) const {
62:     if (r < 0 || r >= _nr || c < 0 || c >= _nc)
63:         throw out_of_range("Invalid indices for matrix");
64:
65:     return _data[r +  c * _nr];   // column-major
66: }
67:
68: // Provides write-access through indexing operator
69: double& matrix::operator()(int r, int c) {
70:     if (r < 0 || r >= _nr || c < 0 || c >= _nc)
71:         throw out_of_range("Invalid indices for matrix");
72:
```

```
 73:     return _data[r +  c * _nr];   // column-major
 74: }
 75:
 76:
 77: /*****************************************************
 78:  * IO operations
 79:  *****************************************************/
 80:
 81: istream& operator>>(istream& in, matrix& m) {
 82:     // presumes that there is a blank line to terminate the matrix
 83:     vector< vector<double> >  data;
 84:
 85:     unsigned int numColumns = 0;
 86:     bool done = false;
 87:     while (!done) {
 88:         vector<double> row;
 89:         string temp;
 90:         getline(in, temp);
 91:         stringstream s(temp);
 92:         double val;
 93:
 94:         while (s >> val) {
 95:             row.push_back(val);
 96:         }
 97:
 98:         if (row.size() == 0)
 99:             done = true;
100:         else if (numColumns > 0 and numColumns != row.size())
101:             done = true;
102:         else {
103:             numColumns = row.size();
104:             data.push_back(row);
105:         }
106:     }
107:
108:     m = matrix(data.size(), numColumns);
109:     for (int r=0; r < m.numRows(); r++)
110:         for (int c=0; c< m.numColumns(); c++)
111:             m(r,c) = data[r][c];
112:
113:     return in;
114: }
```

matrix_proxy.h

```
 1: #ifndef MATRIX_PROXY_H
 2: #define MATRIX_PROXY_H
 3:
 4: #include "matrix_expression.h"
 5: #include "matrix.h"
 6:
 7: /*************************************************
 8:  * matrix_proxy class
 9:  *************************************************/
10: class matrix_proxy : public matrix_expression {
11:  private:
12:   matrix_expression& _src;  // reference to the underlying source (which may be matrix or
13:   matrix _rows;             // copy of matrix expression describing extent of rows
14:   matrix _cols;             // copy of matrix expression describing extent of columns
15:
16:  public:
17:
18:   // use inherited assignment operator (rather than default)
19:   using matrix_expression::operator=;
20:   matrix_proxy& operator=(const matrix_proxy& other);
21:
22:   // we need following declaration to properly inherit the version of
23:   // operator() for submatrices.  Then we can define our additional ones.
24:   using matrix_expression::operator();
25:
26:   // constructor
27:   matrix_proxy(matrix_expression& src,
28:                const matrix_expression& rows,
29:                const matrix_expression& ccols);
30:
31:   int numRows() const;
32:
33:   int numColumns() const;
34:
35:   // read-only version of indexing operator
36:   double operator()(int r, int c) const;
37:
38:   // write-access version of indexing operator
39:   double& operator()(int r, int c);
40:
41: };
42:
43:
44: #endif
```

matrix_proxy.cpp

```
 1: #include <stdexcept>
 2: #include <algorithm>
 3: #include "matrix_proxy.h"
 4: using namespace std;
 5:
 6:
 7: matrix_proxy::matrix_proxy(matrix_expression& src,
 8:                            const matrix_expression& rows,
 9:                            const matrix_expression& cols)
10:     :  _src(src), _rows(rows), _cols(cols) {
11:     if (min(rows.numColumns(), rows.numRows()) != 1)
12:         throw invalid_argument("rows must be one-dimensional.");
13:     if (min(cols.numColumns(), cols.numRows()) != 1)
14:         throw invalid_argument("cols must be one-dimensional.");
15:
16:     for (int k=0; k < numRows(); k++)
17:         if (_rows[k] < 0 || _rows[k] >= src.numRows())
18:             throw out_of_range("invalid row specified");
19:
20:     for (int k=0; k < numColumns(); k++)
21:         if (_cols[k] < 0 || _cols[k] >= src.numColumns())
22:             throw out_of_range("invalid column specified");
23: }
24:
25: // cannot do traditional assignment on proxies, as we cannot rebind source matrix.
26: // We will only support value-based assignments
27: matrix_proxy& matrix_proxy::operator=(const matrix_proxy& other) {
28:     if (this != &other)
29:         matrix_expression::operator=(other);
30:     return *this;
31: }
32:
33: int matrix_proxy::numRows() const {
34:     return _rows.numRows()*_rows.numColumns();
35: }
36:
37: int matrix_proxy::numColumns() const {
38:     return _cols.numRows()*_cols.numColumns();
39: }
40:
41: // read-only version of indexing operator
42: double matrix_proxy::operator()(int r, int c) const {
43:     return _src(_rows[r], _cols[c]);
44: }
45:
46: // write-access version of indexing operator
47: double& matrix_proxy::operator()(int r, int c) {
48:     return _src(_rows[r], _cols[c]);
49: }
```