```
 1: #include <iostream>
 2: #include "range.h"
 3:
 4: class matrix;        // forward declaration to avoid circular includes
 5: class matrix_proxy;  // forward declaration to avoid circular includes
 6:
 7: /*********************************************************
 8:  * matrix_expression class
 9:  *
10:  * This is an "abstract" class in that it is impossible
11:  * to directly create instances of this class.  Its
12:  * purpose is to serve as a base class for concrete
13:  * implementations (e.g., matrix, matrix_proxy).
14:  *
15:  * As such, you cannot declare a standard value variable
16:  * of the form:
17:  *
18:  *   matrix_expression A;
19:  *
20:  * But can instead declare reference variables of this type, assigned
21:  * to expressions that evaluate to said type, as in
22:  *
23:  *   matrix_expression &A = B(range(2,5), range(3,8));
24:  *
25:  *********************************************************/
26: class matrix_expression {
27:
28:  public:
29:
30:   /*********************************************************
31:    *  The following two definitions are pure virtual functions
32:    *  that must be overridden by each child class.
33:    *********************************************************/
34:
35:   // Provides read-only access via indexing operator
36:   virtual double operator()(int r, int c) const = 0;
37:
38:   // Provides write-access through indexing operator
39:   virtual double& operator()(int r, int c)  = 0;
40:
41:   // Returns the number of rows
42:   virtual int numRows() const =0;
43:
44:   // Returns the number of columns
45:   virtual int numColumns() const =0;
46:
47:
48:   /*********************************************************
49:    *  The remaining definitions are functions that can be shared
50:    *  by all child classes (although they are free to override
51:    *  them if desired).
52:    *
53:    *  Generic implementations for these are provided in
54:    *  matrix_expression.cpp
55:    *********************************************************/
56:
57:   // assignment operator supports syntax:  A = B;
58:   // For generic expressions, this is only well defined if dimensions agree.
59:   virtual matrix_expression& operator=(const matrix_expression& other);
60:
61:   // Returns a 1x2 matrix describing the number of rows and columns respectively
62:   virtual matrix size() const;
63:
64:   // element-wise test of equality.
65:   virtual bool operator==(const matrix_expression &other) const;
```

```
66:
67:    // element-wise test of inequality
68:    virtual bool operator!=(const matrix_expression &other) const;
69:
70:    // provides read-only access to a submatrix via a proxy
71:    virtual const matrix_proxy operator()(range rows, range cols) const;
72:
73:    // provides write access to a submatrix via a proxy
74:    virtual matrix_proxy operator()(range rows, range cols);
75:
76:    //-----------------------------------------------
77:    // addition
78:    //-----------------------------------------------
79:
80:    // returns new matrix instance based on sum of two expressions
81:    virtual matrix operator+(const matrix_expression& other) const;
82:
83:    // in-place addition with another matrix expression
84:    virtual matrix_expression& operator+=(const matrix_expression& other);
85:
86:    // returns new matrix instance based on element-wise addition
87:    virtual matrix operator+(double scalar) const;
88:
89:    // in-place element-wise addition with a scalar
90:    virtual matrix_expression& operator+=(double scalar);
91:
92:    //-----------------------------------------------
93:    // multiplication
94:    //-----------------------------------------------
95:
96:    // returns a new matrix based on element-wise multiplication by a scalar
97:    // e.g.,   C = A*B;
98:    virtual matrix operator*(double scalar) const;
99:
100:   // in-place element-wise multiplication with a scalar
101:   // e.g.,  C = A*4;
102:   virtual matrix_expression& operator*=(double scalar);
103:
104:   // returns a new matrix based on product of expressions
105:   virtual matrix operator*(const matrix_expression& other) const;
106:
107:   //-----------------------------------------------
108:   // destructor (a formality in this case)
109:   //-----------------------------------------------
110:   virtual ~matrix_expression() { }
111: };
112:
113: // support for outputting a matrix expression
114: std::ostream& operator<<(std::ostream& out, const matrix_expression& m);
115:
116: // support for left-hand side scalar operations
117: matrix operator+(double scalar, const matrix_expression& m);
118: matrix operator*(double scalar, const matrix_expression& m);
```

matrix.h

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

matrix_proxy.h

```
 1: #include "range.h"
 2: #include "matrix_expression.h"
 3:
 4: /*************************************************
 5:  * matrix_proxy class
 6:  *************************************************/
 7: class matrix_proxy : public matrix_expression {
 8:  private:
 9:   matrix_expression& _M;  // reference to the underlying source (which may be ma
10:   const range _rows;      // copy of range describe extent of rows
11:   const range _cols;      // copy of range describing extent of columns
12:
13:  public:
14:
15:   // use inherited assignment operator (rather than default)
16:   using matrix_expression::operator=;
17:   matrix_proxy& operator=(const matrix_proxy& other);
18:
19:   // constructor
20:   matrix_proxy(matrix_expression& src, const range& rows, const range& ccols);
21:
22:   int numRows() const;
23:
24:   int numColumns() const;
25:
26:   // read-only version of indexing operator
27:   double operator()(int r, int c) const;
28:
29:   // write-access version of indexing operator
30:   double& operator()(int r, int c);
31:
32: };
```

matrix_expression.cpp

```
 1: #include "matrix_expression.h"
 2: #include "matrix.h"
 3: #include "matrix_proxy.h"
 4: using namespace std;
 5:
 6: /*************************************************
 7:  * support for matrix_expression class
 8:  *************************************************/
 9:
10: // assignment operator supports syntax:  A = B;
11: // For generic expressions, this is only well defined if dimensions agree.
12: matrix_expression& matrix_expression::operator=(const matrix_expression& other)
13:   if (numRows() != other.numRows() || numColumns() != other.numColumns())
14:     throw invalid_argument("Matrix dimensions must agree.");
15:
16:   for (int r=0; r < numRows(); r++)
17:     for (int c=0; c < numColumns(); c++)
18:       (*this)(r,c) = other(r,c);
19:
20:   return *this;
21: }
22:
23: // Returns a 1x2 matrix describing the number of rows and columns respectively
24: matrix matrix_expression::size() const {
25:   matrix result(1,2);
26:   result(0,0) = numRows();
27:   result(0,1) = numColumns();
28:   return result;
29: }
30:
31: // provides read-only access to a submatrix via a proxy
32: const matrix_proxy matrix_expression::operator()(range rows, range cols) const {
33:   return matrix_proxy(*const_cast<matrix_expression*>(this), rows, cols);
34: }
35:
36: // provides write access to a submatrix as a proxy
37: matrix_proxy matrix_expression::operator()(range rows, range cols) {
38:   return matrix_proxy(*this, rows, cols);
39: }
40:
41: //-----------------------------------------------
42: // addition
43: //-----------------------------------------------
44:
45: // returns new matrix instance based on sum of two expressions
46: matrix matrix_expression::operator+(const matrix_expression& other) const {
47:   if (numRows() != other.numRows() || numColumns() != other.numColumns())
48:     throw invalid_argument("Matrix dimensions must agree.");
49:
50:   matrix result(*this);        // a new matrix instance for result
51:   for (int r=0; r < numRows(); r++)
52:     for (int c=0; c < numColumns(); c++)
53:       result(r,c) += other(r,c);
54:
55:   return result;
56: }
57:
58: // in-place addition with another matrix expression
59: matrix_expression& matrix_expression::operator+=(const matrix_expression& other)
60:   if (numRows() != other.numRows() || numColumns() != other.numColumns())
61:     throw invalid_argument("Matrix dimensions must agree.");
62:
63:   for (int r=0; r < numRows(); r++)
64:     for (int c=0; c < numColumns(); c++)
65:       (*this)(r,c) += other(r,c);
```

```
66:
67:   return *this;
68: }
69:
70: // returns new matrix instance based on element-wise addition
71: matrix matrix_expression::operator+(double scalar) const {
72:   matrix result(*this);
73:   for (int r=0; r < numRows(); r++)
74:     for (int c=0; c < numColumns(); c++)
75:       result(r,c) += scalar;
76:   return result;
77: }
78:
79: // in-place element-wise addition with a scalar
80: matrix_expression& matrix_expression::operator+=(double scalar) {
81:
82:   for (int r=0; r < numRows(); r++)
83:     for (int c=0; c < numColumns(); c++)
84:       (*this)(r,c) += scalar;
85:
86:   return *this;
87: }
88:
89: //-----------------------------------------------
90: // rest omitted for brevity...
91: //-----------------------------------------------
92:
```

matrix.cpp

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

matrix_proxy.cpp

```
 1: #include "matrix_proxy.h"
 2:
 3: matrix_proxy::matrix_proxy(matrix_expression& src, const range& rows, const rang
 4:   :  _M(src), _rows(rows), _cols(cols) { }
 5:
 6: // cannot do traditional assignment on proxies, since we cannot rebind the sourc
 7: // We will only support value-based assignments
 8: matrix_proxy& matrix_proxy::operator=(const matrix_proxy& other) {
 9:   if (this != &other)
10:     matrix_expression::operator=(other);
11:   return *this;
12: }
13:
14: int matrix_proxy::numRows() const {
15:   return _rows.size();
16: }
17:
18: int matrix_proxy::numColumns() const {
19:   return _cols.size();
20: }
21:
22: // read-only version of indexing operator
23: double matrix_proxy::operator()(int r, int c) const {
24:   int actualRow = _rows.start() + r * _rows.stride();
25:   int actualCol = _cols.start() + c * _cols.stride();
26:   return _M(actualRow, actualCol);
27: }
28:
29: // write-access version of indexing operator
30: double& matrix_proxy::operator()(int r, int c) {
31:   int actualRow = _rows.start() + r * _rows.stride();
32:   int actualCol = _cols.start() + c * _cols.stride();
33:   return _M(actualRow, actualCol);
34: }
```