```
 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: };
```

```
 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 matrix
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:   // constructor
16:   matrix_proxy(matrix_expression& src, const range& rows, const range& cols);
17:
18:   int numRows() const;
19:
20:   int numColumns() const;
21:
22:   // read-only version of indexing operator
23:   double operator()(int r, int c) const;
24:
25:   // write-access version of indexing operator
26:   double& operator()(int r, int c);
27:
28: };
```

matrix_expression.cpp

```
 1: #include "matrix_exprression.h"
 2: #include "matrix.h"
 3:
 4:
 5:
 6:
 7: //------------    implementation of operator+    --------------/
 8:
 9: // returns new matrix instance based on sum of two expressions
10: matrix matrix_expression::operator+(const matrix_expression& other) const {
11:   if (numRows() != other.numRows() || numColumns() != other.numColumns())
12:     throw invalid_argument("Matrix dimensions must agree.");
13:
14:   matrix result(numRows(), numColumns());  // a new matrix instance for result
15:   for (int r=0; r < numRows(); r++)
16:     for (int c=0; c < numColumns(); c++)
17:       result(r,c) = (*this)(r,c) + other(r,c);
18:
19:   return result;
20: }
```