matrix.h

```
#ifndef MATRIX_H
#define MATRIX_H



#include <iostream>
#include <stdexcept>
#include <vector>

class matrix_proxy;  // forward reference as place holder

using namespace std;


/****************************************************
 * range class
 ****************************************************/

class range {
private:
  int _start;
  int _stop;
  int _stride;

public:

  // supports construction such as range(3) for the singleton set {3}
  range(int start);

  // supports construction such as range(3,6), which includes values {3, 4, 5}
  range(int start, int stop);

  // supports construction such as range(3,2,8), which includes values {3, 5, 7}
  range(int start, int stride, int stop);

  // Returns starting index
  int start() const;

  // Returns stopping index
  int stop() const;

  // Returns stopping index
  int stride() const;

  // Returns the number of values included within the range
  int size() const;

};
```

matrix.h

```
/*****************************************************
 * matrix class
 ****************************************************/

class matrix {
private:
  int _nr;                    /* number of rows */
  int _nc;                    /* number of columns */
  vector<double> _data;       /* underlying data storage */

public:
  matrix();

  matrix(int numRows, int numColumns, double value=0);

  int numRows() const;

  int numColumns() const;

  matrix size() const;

  void reshape(int r, int c);

  bool operator==(const matrix &other) const;

  bool operator!=(const matrix &other) const;

  // provides read-only access to a matrix entry
  double operator()(int r, int c) const;

  // provides write access to a matrix entry (albeit, without expansion)
  double& operator()(int r, int c);


  // provides write access to a submatrix as a proxy
  matrix_proxy operator()(range rows, range cols);


  //-----------------------------------------------
  // addition
  //-----------------------------------------------
  matrix operator+(const matrix& other) const;

  matrix operator+(double scalar) const;


  //-----------------------------------------------
  // multiplication
  //-----------------------------------------------
  matrix operator*(double scalar) const;

  matrix operator*(const matrix& other) const;
};

//------------------------------------------------------
// define additional support for reading/writing matrices
//------------------------------------------------------
ostream& operator<<(ostream& out, const matrix& m);
istream& operator>>(istream& in, matrix& m);


#include "matrix_proxy.h"    // time to get the full class definition

#endif
```

matrix.cpp (excerpt)

```
#include <iostream>
#include <iomanip>
#include <sstream>
#include <vector>
#include "matrix.h"
using namespace std;

/*****************************************************
 * range class
 *****************************************************/

range::range(int start) : _start(start), _stop(start+1), _stride(1) { }

// supports construction such as range(3,6), which includes values {3, 4, 5}
range::range(int start, int stop) : _start(start), _stop(stop), _stride(1) { }

// supports construction such as range(3,2,8), which includes values {3, 5, 7}
range::range(int start, int stride, int stop)
  : _start(start), _stop(stop), _stride(stride) {
  if (stride < 1)
    throw invalid_argument("stride must be positive.");
}

// Returns starting index
int range::start() const {
  return _start;
}

// Returns stopping index
int range::stop() const {
  return _stop;
}

// Returns stopping index
int range::stride() const {
  return _stride;
}

// Returns the number of values included within the range
int range::size() const {
  // partials strides should count as one.  e.g. range(1,2,4).size() should be 2
  return max(0, (_stop - _start + _stride - 1) / _stride);  // trucates properly
}
```

matrix.cpp (excerpt)

```
/****************************************************
 * matrix class
 ***************************************************/

matrix::matrix() : _nr(0), _nc(0), _data() {};

matrix::matrix(int numRows, int numColumns, double value)
  : _nr(numRows), _nc(numColumns), _data(numRows*numColumns, value) {}

int matrix::numRows() const {
  return _nr;
}

int matrix::numColumns() const {
  return _nc;
}

matrix matrix::size() const {
  matrix result(1,2);
  result(0,0) = numRows();
  result(0,1) = numColumns();
  return result;
}

void matrix::reshape(int r, int c) {
  if (r * c != _nr * _nc)
    throw invalid_argument("To reshape, the number of elements must not change.");

  _nr = r;
  _nc = c;
}

bool matrix::operator==(const matrix &other) const {
  return (_nr == other._nr && _nc == other._nc && _data == other._data);
}

bool matrix::operator!=(const matrix &other) const {
  return !(*this == other);
}

// provides read-only access to a matrix entry
double matrix::operator()(int r, int c) const {
  if (r < 0 || r >= _nr || c < 0 || c >= _nc)
    throw out_of_range("Invalid indices for matrix");

  return _data[r +  c * _nr];   // column-major
}

// provides write access to a matrix entry (albeit, without expansion)
double& matrix::operator()(int r, int c) {
  if (r < 0 || r >= _nr || c < 0 || c >= _nc)
    throw out_of_range("Invalid indices for matrix");

  return _data[r +  c * _nr];   // column-major
}


// provides write access to a submatrix as a proxy
matrix_proxy matrix::operator()(range rows, range cols) {
  return matrix_proxy(*this, rows, cols);
}
```

matrix_proxy.h

```
#ifndef MATRIX_PROXY_H
#define MATRIX_PROXY_H

#include "matrix.h"

/****************************************************
 * matrix_proxy class
 ****************************************************/
class matrix_proxy {
 private:
  matrix& _M;            // reference to the underlying source matrix
  const range _rows;     // copy of range describe extent of rows
  const range _cols;     // copy of range describing extent of columns

 public:
  matrix_proxy(matrix& src, const range& r, const range& c)
    :  _M(src), _rows(r), _cols(c) { }

  int numRows() const {
    return _rows.size();
  }

  int numColumns() const {
    return _cols.size();
  }

  // allows assignment from another matrix
  matrix_proxy&  operator=(const matrix& 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;
  }

  // read-only version of indexing operator
  double operator()(int r, int c) const {
    int actualRow = _rows.start() + r * _rows.stride();
    int actualCol = _cols.start() + c * _cols.stride();
    return _M(actualRow, actualCol);
  }

  // write-access version of indexing operator
  double& operator()(int r, int c) {
    int actualRow = _rows.start() + r * _rows.stride();
    int actualCol = _cols.start() + c * _cols.stride();
    return _M(actualRow, actualCol);
  }

};

//-----------------------------------------------------
// define additional support for outputting matrix proxies
//-----------------------------------------------------
ostream& operator<<(ostream& out, const matrix_proxy& m);


#endif
```