13.4 Case Study: vector<T>-Based Matrices

A two-dimensional numeric array having m rows and n columns is called an $\boldsymbol{m \times n}$ **matrix.** There are many important applications of matrices because there are many problems that can be solved most easily using matrices and matrix operations. Thus a `Matrix` class would be very useful and the task at hand is to build such a class.

To begin, we must know what matrix operations to include. Here we will confine our attention to addition, subtraction, multiplication, and transpose. The **sum** of two matrices that have the same number of rows and the same number of columns is defined as follows: If A_{ij} and B_{ij} are the entries in the ith row and jth column of $m \times n$ matrices A and B, respectively, then $A_{ij} + B_{ij}$ is the entry in the ith row and jth column of the sum, which will also be an $m \times n$ matrix. For example,

$$\begin{bmatrix} 1 & 0 & 2 \\ -1 & 3 & 5 \end{bmatrix} + \begin{bmatrix} 4 & 2 & 1 \\ 7 & 0 & 3 \end{bmatrix} = \begin{bmatrix} 5 & 2 & 3 \\ 6 & 3 & 8 \end{bmatrix}$$

The **difference** of two such matrices is obtained simply by replacing + by –; for example,

$$\begin{bmatrix} 1 & 0 & 2 \\ -1 & 3 & 5 \end{bmatrix} - \begin{bmatrix} 4 & 2 & 1 \\ 7 & 0 & 3 \end{bmatrix} = \begin{bmatrix} -3 & -2 & 1 \\ -8 & 3 & 2 \end{bmatrix}$$

The **product** of two matrices is more difficult to calculate. For $A * B$ to be defined, the number of columns in A must match the number the number of rows in B. So suppose that A is an $m \times n$ matrix and B is an $n \times p$ matrix. The product C of A and B is an $m \times p$ matrix with the entry C_{ij}, which appears in the ith row and the jth column, given by

$$\begin{aligned} C_{ij} &= \text{The sum of the products of the entries in row } i \text{ of } A \\ &\quad\ \text{with the entries in column } j \text{ of } B \\ &= A_{i1} \times B_{1j} + A_{i2} \times B_{2j} + \cdots + A_{in} \times B_{nj} \end{aligned}$$

To illustrate, suppose that A is the 2×3 matrix

$$\begin{bmatrix} 1 & 0 & 2 \\ 3 & 0 & 4 \end{bmatrix}$$

and that B is the 3×4 matrix

$$\begin{bmatrix} 4 & 2 & 5 & 3 \\ 6 & 4 & 1 & 8 \\ 9 & 0 & 0 & 2 \end{bmatrix}$$

Because the number of columns (3) in A equals the number of rows in B, the product matrix is defined. The entry in the first row and first column is obtained by multiplying the first row of mat1 with the first column of mat2, element by element, and adding these products:

$$\begin{bmatrix} \boxed{1 \quad 0 \quad 2} \\ 3 \quad 0 \quad 4 \end{bmatrix} \begin{bmatrix} 4 & 2 & 5 & 3 \\ 6 & 4 & 1 & 8 \\ 9 & 0 & 0 & 2 \end{bmatrix}$$

$$1 \;\; 4 + 0 \;\; 6 + 2 \;\; 9 = 22$$

Similarly, the entry in the first row and second column is

$$\begin{bmatrix} \boxed{1 \quad 0 \quad 2} \\ 3 \quad 0 \quad 4 \end{bmatrix} \begin{bmatrix} 4 & 2 & 5 & 3 \\ 6 & 4 & 1 & 8 \\ 9 & 0 & 0 & 2 \end{bmatrix}$$

$$1 \;\; 2 + 0 \;\; 4 + 2 \;\; 0 = 2$$

The complete product matrix is the 2×4 matrix given by

$$\begin{bmatrix} 22 & 2 & 5 & 7 \\ 48 & 6 & 15 & 17 \end{bmatrix}$$

BUILDING A `Matrix` CLASS: THE EXTERNAL APPROACH

If we imitate the `Table` class in the preceding section, then building a `Matrix` class is quite easy, because a matrix can be thought of as a vector of vectors of numbers. We can simply use a `typedef` statement to declare the name `Matrix` as an alias for `vector< vector<double> >`. To make this declaration reusable, we would place it in a `Matrix` library header *Matrix.h*:

```
/* Matrix.h provides the type Matrix and its
 * operation prototypes.
 *********************************************/

#include <vector>
using namespace std;
#include "Table.h"                  // Table prototypes

typedef vector<double> MatrixRow;
typedef vector< MatrixRow > Matrix;

// ... Matrix operation prototypes go here
```

A program that includes this header file can now define an empty `Matrix` object as follows:

```
Matrix aMatrix;
```

A non-empty `Matrix` can be defined using the same approach as in Section 13.3:

```
const int ROWS = 3,
          COLS = 4;
Matrix theMatrix(ROWS, MatrixRow(COLS, 0.0));
```

This definition builds `theMatrix` as a 3×4 matrix, and sets each of its elements to zero.

theMatrix:	[0]	[1]	[2]	[3]
[0]	0.0	0.0	0.0	0.0
[1]	0.0	0.0	0.0	0.0
[2]	0.0	0.0	0.0	0.0

`Matrix` OPERATIONS

Because the identifier `Matrix` is a synonym for `vector< vector<double> >`, any operation defined for `vector< vector<double> >` can be applied to a `Matrix` object. For example, the double-subscript operation can be used to access a particular element of a `Matrix`; that is, `theMatrix[r][c]` is the entry of `theMatrix` in row `r` and column `c`. Similarly, the `size()` function can be used to determine the number of rows in a `Matrix`. The statements

```
for (int r = 0; r < theMatrix.size(); r++)
   for (int c = 0; c < theMatrix[r].size(); c++)
      theMatrix[r][c] = r + c + 1;
```

will modify `theMatrix` as follows:

theMatrix:	[0]	[1]	[2]	[3]
[0]	1.0	2.0	3.0	4.0
[1]	2.0	3.0	4.0	5.0
[2]	3.0	4.0	5.0	6.0

In addition, because `Matrix` is a synonym for `vector< vector<double> >` and `Table` is also a synonym for `vector< vector<double> >`, the operations defined for `Table` (e.g., `fill()` from Figure 13-4 of the text) can also be applied to `Matrix` objects. Including the directive `#include "Table.h"` in *Matrix.h* adds the prototypes of these operations.

Operations that are specific to matrices such as addition, subtraction, and multiplication must be defined as functions. For example, the following function definitiom shows the implementation of the matrix multiplication operation by overloading `operator*`. Because this is a reasonably complicated operation, we define it in a separately compiled implementaton file *Matrix.cpp* and place its prototype in *Matrix.h*.

Case Study 13.4-1 `Matrix` Multiplication.

```
/* Matrix.cpp defines the Matrix operations.
 * ...
 ************************************************************/

#include "Matrix.h"                       // type Matrix
#include <cassert>                        // assert()
using namespace std;

Matrix operator*(const Matrix& mat1, const Matrix& mat2)
{
   const int ROWS1 = mat1.size(),
             ROWS2 = mat2.size();
   assert(ROWS1 > 0 && ROWS2 > 0);      // verify nonzero

   const int COLS1 = mat1[0].size(),
             COLS2 = mat2[0].size();
   assert(COLS1 == ROWS2);              // check precondition

   Matrix mat3(ROWS1,                   // define result Matrix
               MatrixRow(COLS2, 0.0));

   for (int i = 0; i < ROWS1; i++)      // for each row in mat1:
     for (int j = 0; j < COLS2; j++)    //   for each col in mat2:
     {
       double sum = 0;
       for (int k = 0; k < COLS1; k++)  //    for each col in mat1:
         sum += mat1[i][k] * mat2[k][j];//      sum the products
       mat3[i][j] = sum;                //    put sum in result Matrix
     }

   return mat3;                         // return the result Matrix
}
```

Once `Matrix` operations have been defined, a program can make use of these operations in the same manner as those of any other class, as in the following program:

Case Study 13.4-2 Program To Demonstrate `Matrix` Multiplication.

```
/* matMult.cpp tests the matrix multiplication method.
 *
 *  Input (keyboard): names of files containing matrices
 *  Input (files):    two matrices
 *  Precondition:     the first line of each file == rows & columns
 *  Output (screen):  the matrices together with their product
 *****************************************************************/

#include <iostream>                           // cin, cout, <<, >>
#include <string>                             // string type
using namespace std;
#include "Matrix.h"                           // Matrix, operator *

int main()
{
  cout << "\nThis program demonstrates matrix multiplication,\n"
          "by multiplying two matrices stored in separate files.\n"
          "\nA file must list the # of rows and columns of its "
          "matrix.\n";
                                              // get file names
  cout << "\nPlease enter the name of the first file: ";
  string file1;
  cin >> file1;

  cout << "and the name of the second file: ";
  string file2;
  cin >> file2;

  Matrix matrix1,
         matrix2;

  fill(file1, matrix1);                  // load matrix1 from file1
  fill(file2, matrix2);                  // load matrix2 from file2
                                         // Display the matrices
  cout << "\n- Matrix1 --------------------------------------\n";
  print(cout, matrix1);
  cout << "\n- Matrix2 --------------------------------------\n";
  print(cout, matrix2);

  Matrix matrix3 = matrix1 * matrix2;    // perform multiplication
                                         // display matrix3
  cout << "\n- Matrix3 --------------------------------------\n";
  print(cout, matrix3);
}
```

Listing of Input File *mat2x3.dat*:

```
2 3
1 0 2
3 0 4
```

Listing of Input File *mat3x4.dat*:

```
3 4
4 2 5 3
6 4 1 8
9 0 0 2
```

Sample run:

```
This program demonstrates matrix multiplication,
by multiplying two matrices stored in separate files.

A file must list the # of rows and columns of its matrix.

Please enter the name of the first file: mat2x3.dat
and the name of the second file: mat3x4.dat

- Matrix1 -------------------------------------
1       0       2
3       0       4

- Matrix2 -------------------------------------
4       2       5       3
6       4       1       8
9       0       0       2

- Matrix3 -------------------------------------
22      2       5       7
48      6       15      17
```

It is important to understand that the statement

```
Matrix matrix1,
       matrix2;
```

builds `matrix1` and `matrix2` as empty vectors of vectors of numbers. The statements

```
fill(file1, matrix1);
fill(file2, matrix2);
```

use function `fill()` from the `Table` library defined in Section 13.3 of the text, and the statements

```
print(cout, matrix1);
print(cout, matrix2);
```

apply the function `print()` from the same `Table` library. The declaration of the result matrix

```
Matrix matrix3 = matrix1 * matrix2;
```

constructs `matrix3` as a `Matrix`, and initializes it with the `Matrix` returned by `operator*`, rather than using the default assignment mechanism. The definition of `print()` from the `Table` library is then used a final time to display `matrix3`.

Implementing the addition and subtraction operations is much easier than for multiplication and do we leave them as exercises.

APPLICATION: SOLVING LINEAR SYSTEMS

A linear system is a set of linear equations, each of which involves several unknowns; for example,

$$\begin{aligned} 5x_1 - x_2 - 2x_3 &= 11 \\ -x_1 + 5x_2 - 2x_3 &= 0 \\ -2x_1 - 2x_2 + 7x_3 &= 0 \end{aligned}$$

is a linear system of three equations involving the three unknowns x_1, x_2, and x_3. A solution of such a system is a collection of values for these unknowns that satisfies all of the equations simultaneously.

One method for solving a linear system is called **Gaussian elimination.** In this method, we first eliminate x_1 from the second equation by adding 1/5 times the first equation to the second equation and, from the third equation, by adding times the first equation to the third equation. This yields the linear system

$$\begin{aligned} 5x_1 - x_2 - 2x_3 &= 11 \\ 4.8x_2 - 2.4x_3 &= 2.2 \\ -2.4x_2 + 6.2x_3 &= 4.4 \end{aligned}$$

which is equivalent to the first system because it has the same solution as the original system. We next eliminate x_2 from the third equation by adding 2.4 / 4.8 = times the second equation to the third, giving the new equivalent linear system:

$$\begin{aligned} 5x_1 - x_2 - 2x_3 &= 11 \\ 4.8x_2 - 2.4x_3 &= 2.2 \\ 5x_3 &= 5.5 \end{aligned}$$

Once the original system has been reduced to such a *triangular* form, it is easy to find the solution. It is clear from the last equation that the value of x_3 is

$$x_3 = \frac{5.5}{5} = 1.100$$

Substituting this value for x_3 in the second equation and solving for x_2 gives

$$x_2 = \frac{2.2 + 2.4(1.1)}{4.8} = 1.008$$

and substituting these values for x_2 and x_3 in the first equation gives

$$x_1 = \frac{11 + 1.008 + 2(1.100)}{5} = 2.842$$

The original linear system can also be written as a single matrix equation

$$Ax = b$$

where A is the 3 × 3 **coefficient matrix,** ***b*** is the 3 × 1 **constant vector,** and ***x*** is the 3 × 1 **vector of unknowns:**

$$A = \begin{bmatrix} 5 & -1 & -2 \\ -1 & 5 & -2 \\ -2 & -2 & 7 \end{bmatrix}, \; x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, \; b = \begin{bmatrix} 11 \\ 0 \\ 0 \end{bmatrix}$$

The operations used to reduce the original linear system to triangular form use only the coefficient matrix *A* and the constant vector *b*. Thus, if we combine these into a single matrix by adjoining *b* to *A* as a last column,

$$Aug = \begin{bmatrix} 5 & -1 & -2 & 11 \\ -1 & 5 & -2 & 0 \\ -2 & -2 & 7 & 0 \end{bmatrix}$$

we can carry out the required operations on this new matrix, called the **augmented matrix,** without writing down the unknowns at each step. Thus we add – *Aug*[1][0] / *Aug*[0][0]5 = 1/5 times the first row of *Aug* to the second row, and – *Aug*[2][0] / *Aug*[0][0]5 = 2/5 times the first row of *Aug* to the third row, to obtain the new matrix:

$$Aug = \begin{bmatrix} 5 & -1 & -2 & 11 \\ 0 & 4.8 & -2.4 & 2.2 \\ 0 & -2.4 & 6.2 & 4.4 \end{bmatrix}$$

Then adding – *Aug*[2][1] / *Aug*[1][1]5 = 1/2 times the second row to the third row gives the following triangular matrix, which corresponds to the final triangular system of equations:

$$Aug = \begin{bmatrix} 5 & -1 & -2 & 11 \\ 0 & 4.8 & -2.4 & 2.2 \\ 0 & 0 & 5 & 5.5 \end{bmatrix}$$

From this example, we see that the basic row operation performed at the *i*th step of the reduction process is:

for $k = i + 1, i + 2, \ldots, n$

$$\text{Replace row}_k \text{ by row}_k - \frac{Aug[k][i]}{Aug[i][i]} \times \text{row}_i$$

Clearly, for this to be possible, the element *Aug*[*i*][*j*], called a **pivot** element, must be nonzero. If it is not, we must interchange the ith row with a later row to produce a nonzero pivot.

The following program solves linear systems using Gaussian elimination. To minimize the effect of roundoff error in the computations, it selects as a pivot at each stage in the reduction the candidate that is largest in absolute value.

Case Study 13.4-3 Linear Equation Solver that Uses Gaussian Elimination

```
/* gaussElim.cpp solves systems of linear equation using
 * Gaussian Elimination.
 *
 * Input:  a series of linear equation coefficients
 * Output: the solution of the linear equation or a "singular
 *           system" message
 * Joel Adams of Calvin College.
 **************************************************************/

#include <iostream>       // cout, cin, ...
#include <cstdlib>        // exit()
using namespace std;
#include "Matrix.h"       // Matrix class

void readEquations(Matrix & augMat);
int reduce(Matrix & augMat);
Matrix solve(Matrix & augMat);
Matrix GaussElim();

int main()
{
  cout << "This program solves a linear system "
          "using Gaussian Elimination. \n";

  Matrix solutionVector = GaussElim();

  cout << "\nThe solution vector (x1, x2, ...)  for this system is:\n\n"
       << solutionVector << endl;
}
```

```
/* GaussElim() performs the Gaussian Elimination algorithm
 *
 * Input:  the coefficients of a linear system and its constant vector
 * Return: the solution to the linear system
 * Joel Adams of Calvin College.

***********************************************************************
*/

Matrix GaussElim()
{
   Matrix augmentedMatrix;

   readEquations(augmentedMatrix);

   bool isSingular = reduce(augmentedMatrix);

   if (isSingular)
   {
      cerr << "\n*** GaussElim: Coefficient Matrix is (nearly) singu-
lar!\n";
      exit (0);
   }

   Matrix solutionVector = solve(augmentedMatrix);

   return solutionVector;
}


void readEquations(Matrix & augMat)
{
  int numEquations;

  for (;;)
    {
      cout << "\nPlease enter the number of equations in the system: ";
      cin >> numEquations;
      if (numEquations > 1) break;
      cerr << "\n*** At least two equations are needed ...\n";
    }

  augMat = Matrix(numEquations,       // numEquations rows
                  numEquations+1);    // numEquations+1 columns

  cout << "\nPlease enter the coefficient matrix row-by-row...\n";
  for (int r = 0; r < numEquations; r++)
    for (int c = 0; c < numEquations; c++)
      cin >> augMat[r][c];

  cout << "\nPlease enter the constant vector...\n";
  for (int r = 0; r < numEquations; r++)
    cin >> augMat[r][numEquations];
}


inline double abs(double val)
{
  return (val < 0) ? -(val) : val;
}


inline void swap(double & a, double & b)
{
  double t = a; a = b; b = t;
}


int reduce(Matrix & augMat)
{
  const double EPSILON = 1.0E-6;
  bool isSingular = false;
  int i = 0,
      j,
      k,
      numRows = augMat.Rows(),
      pivotRow;
  double quotient,
         absolutePivot;

  while ((!isSingular) && (i < numRows))
  {
    absolutePivot = abs(augMat[i][i]);
    pivotRow = i;
    for (k = i+1; k < numRows; k++)
      if (abs(augMat[k][i]) > absolutePivot)
      {
         absolutePivot = abs(augMat[k][i]);
         pivotRow = k;
      }
    isSingular = absolutePivot < EPSILON;
    if (!isSingular)
    {
       if (i != pivotRow)
         for (j = 0; j <= numRows; j++)
           swap(augMat[i][j], augMat[pivotRow][j]);

       for (j = i+1; j < numRows; j++)
       {
          quotient = -augMat[j][i] / augMat[i][i];
          for (k = i; k <= numRows; k++)
            augMat[j][k] = augMat[j][k] + quotient * augMat[i][k];
       }
    }
    i++;
  }
  return isSingular;
}


Matrix solve(Matrix & augMat)
{
  Matrix solutionVector(1, augMat.Rows());
  int n = augMat.Rows()-1;

  solutionVector[0][n] = augMat[n][n+1] / augMat[n][n];

  for (int i = n-1; i >= 0; i--)
  {
    solutionVector[0][i] = augMat[i][n+1];

    for (int j = i+1; j <= n; j++)
      solutionVector[0][i] -= augMat[i][j] * solutionVector[0][j];

    solutionVector[0][i] /= augMat[i][i];
  }

  return solutionVector;
}
```

EXERCISES 13.4

1. Add the addition operation to the `Matrix` class.
2. Add the subtraction operation to the `Matrix` class.
3. Add a method to class `Matrix` to find the transpose of a matrix which is defined in Exercise 6 of Section 13.5 of the text.
4. Proceed as in Programming Problem 27 at the end of Chapter 13 of the text, but use the `Matrix` class of this section to solve the product-cost problem.
5. Proceed as in Programming Problem 28 at the end of Chapter 13 of the text, but use the `Matrix` class of this section to solve the coordinate-transformation problem.
6. Proceed as in Programming Problem 29 at the end of Chapter 13 of the text, but use the `Matrix` class of this section to solve the Markov chain problem.
7. Proceed as in Programming Problem 30 at the end of Chapter 13 of the text, but use the `Matrix` class of this section to solve the directed-graph problem.
8. The *inverse* of an $n \times n$ matrix A is a matrix A^{-1} for which both the products $A * A^{-1}$ and $A^{-1} * A$ are equal to the identity matrix having 1s on the diagonal from the upper left to the lower right and 0s

elsewhere. The inverse of matrix A can be calculated by solving the linear systems $Ax = b$ for each of the following constant vectors b:

$$\begin{bmatrix} 1 \\ 0 \\ 0 \\ \cdot \\ \cdot \\ \cdot \\ 0 \end{bmatrix} \quad \begin{bmatrix} 0 \\ 1 \\ 0 \\ \cdot \\ \cdot \\ \cdot \\ 0 \end{bmatrix} \quad \begin{bmatrix} 0 \\ 0 \\ 1 \\ \cdot \\ \cdot \\ \cdot \\ 0 \end{bmatrix} \qquad \begin{bmatrix} 0 \\ 0 \\ 0 \\ \cdot \\ \cdot \\ \cdot \\ 1 \end{bmatrix}$$

These solutions give the first, second, third, . . . , *n*th column of A^{-1}. Write a program that uses Gaussian elimination to solve these linear systems and thus calculates the approximate inverse of a matrix.