Programming Assignment 2, REPORT, Russell Lowke April 17th 2004.

Unfortunately my implementation of Strassen passes matrices "by value" instead of "by reference." This was not due to lack of trying to pass matrices by reference. Unfortunately, each such attempt implementation yielded bogus results, and/or segmentation faults, implying that array bounds were being exceeded.

Switching from a "by value" based metaphor to a "by reference" metaphor in theory should have been reasonably simple, but the compiler produced very numerous, cryptic, and untraceable errors that is suspected to be due to the use of STL vector class.

An attempt was made to remove the STL vectors [5 hours] and replace them with regular int arrays (which are implicitly passed by reference). This also generated very mysterious results. Data initialized in one Matrix was valid until a second Matrix was initialized, at which point the first one showed corrupted data. Unable to get this code working I have had to revert to the STL code, although I have included the bad array code to show an effort was made [as found in the strassenBAD folder included]. I believe that the right approach now would be to rewrite the code from the ground up with all Matrices built using regular int arrays.

Other issues that came up.

• In an attempt to get Strassen's to work when n is odd I initially implemented an overlap of the center row and column, thus a 3 x 3 matrix as such:

[1 , 2, 3]
[4, 5, 6]
[7, 8, 9] was broken into the following 4 smaller matrices

[1, 2] [2, 3] [4, 5] [5, 6]
[4, 5] [5, 6] [7, 8] [8, 9]

I had believed this to work after testing on paper, but actual implementation proved such a methodology incorrect. A new approach had to be taken, I've added a padding of 0s to the matrix instead, then strip the padding after computation. The padded matrix looks as such:

[1 , 2, 3, 0]
[4, 5, 6, 0]
[7, 8, 9, 0]
[0, 0, 0, 0] this methodology works well.

• To optimize my code I removed numerous instances of vector push_back() and replaced them with a vector<> v(n) initializations, which improved performance somewhat and paved the way [I had believed] for a "by reference" version. To get this to work I had to include a _Data(width) in my member initialization list, a step that wasn't immediately obvious.

Because my Stassen's STL implementation passes "by value" it copies all Matrices frequently, resulting in a slow Strassen's algorithm that is outperformed by the regular matrix multiplication. I also believe that the compiler optimizes the operator * Matrix function, further widening the gap between the running time of my Stassen's version and the regular version.

TASKS

1.

In analyzing Strassen's algorithm, at any given step there are 10 matrix add/sub operations in calculating p1-p7 for a term 10*(n/2)^2, 8 add/sub operations in putting them together to calculate the fourths that yields 8(n/2)^2, and 7 recursive calls on these quartered matrices which s 7*T(n/2) which, altogether, yields

T(n) = 7 T(n/2) + 18(n/2)^2

And, when it's put into Mathematica, gives

$$4^{-\frac{\text{Log}[x]}{\text{Log}[2]}} \left(28^{\frac{\text{Log}[x]}{\text{Log}[2]}} + 6\, 7^{\frac{\text{Log}[x]}{\text{Log}[2]}}\, x^2 - 6\, x^4\right)$$

, which while appearing horribly obfuscated, simplifies to $-6\, x^2 + 5\, x^{\frac{\text{Log}[7]}{\text{Log}[2]}}$. The normal algorithm, when analyzed, gives a running time of T(n) = $-n^2 + 2\, n^3$ and Mathematica would not solve this these two for n when set equal to one another. I was able to estimate a crossing over point of 102.66.

2.

Here are my timed results

| | | |
|---|---|---|
| d = 8 | regular:0 | strassen:9 |
| d = 16 | regular:1 | strassen:53 |
| d = 32 | regular:6 | strassen:386 |
| d = 64 | regular:48 | strassen:2752 |
| d = 128 | regular:398 | strassen:18845 |
| d = 256 | regular:3170 | strassen: [takes too long to computer ] |

All timings are in 1/100ths of a second.

As my Strassen's algorithm is so slow I can't experimentally find the crossover point. At least the implementation is robust.

Russell Lowke

```
//
//
//  Matrix.h
//
//      AUTHOR: Russell Lowke
//
//          Date: 4/15/2004
//

#ifndef MATRIX_H
#define MATRIX_H

#include <vector>
using namespace std;

class Matrix;

typedef vector<int>     IntList;
typedef vector<IntList> IntLList;
typedef vector<Matrix>  MatrixList;

class Matrix {

    public:

        Matrix operator+ ( const Matrix& other) const ;
        Matrix operator- ( const Matrix& other) const ;
        Matrix operator* ( const Matrix& other) const ;

        Matrix() {};  // constructor
        Matrix(int width, int height, int data[]);
        Matrix(IntLList data);
        Matrix(Matrix a, Matrix b, Matrix c, Matrix d, int dim);
       ~Matrix() {};  // destructor

        void print() const;
         void printDiag() const;

        MatrixList makeQuarters();
        void addPadding();
        void removePadding();


        //  accessors and mutators
        void setAt(int x, int y, int val);
        int  getWidth();
        int  getHeight();
        int  getAt(int x, int y);

    private:

        int           _Width;
        int           _Height;
```

```
        IntLList  _Data;
};


#endif /* DOCUMENT_H */


//
//
//  Matrix.cpp
//
//      AUTHOR: Russell Lowke
//
//         Date: 4/15/2004
//

#include "Matrix.h"
#include <iostream>

using namespace std;

//
//   constructor:  create a matrix of specified width and height from data[]
//
Matrix::Matrix (int width, int height, int data[]): _Data(width),
_Width(width), _Height(height) {

    for (int x = 0; x < width; ++x) {
        IntList col(height);
        for (int y = 0; y < height; ++y)
             col[y] = data[ width * y + x ];
        _Data[x] = col;
    }
}

//
//   constructor:  create matrix from IntLList
//
Matrix::Matrix (IntLList data) {
     _Width  = data.size();
    _Height = data[0].size();
    _Data   = data;
}

/*
//
//   constructor:  create matrix from four small matrices
//
Matrix::Matrix(Matrix a, Matrix b, Matrix c, Matrix d, int n): _Data(n) {

     _Width   = n;              // x
    _Height  = n;              // y
     IntLList _Data(_Width);
```

```
    int size = a.getWidth();    // size of small matrices
    int val;                    // misc. holder

    // add elements
    for (int x = 0; x < _Width; ++x) {
        IntList col(_Height);                    // make new column

        for (int y = 0; y < _Height; ++y ) {

            if (y <  size && x <  size)
                val = a._Data[x][y];
            else if (y <  size && x >= size)
                val = b._Data[x - size][y];
            else if (y >= size && x < size)
                val = c._Data[x][y - size];
            else if (y >= size && x >= size)
                val = d._Data[x - size][y - size];

            col[y] =val;
        }
        _Data[x] = col;         // add column to IntLList
    }
}
*/

//
//   constructor:  create matrix from four small matrices
//
Matrix::Matrix(Matrix a, Matrix b, Matrix c, Matrix d, int n): _Data(n) {

     _Width   = n;              // x
    _Height  = n;               // y

    // add elements
    Matrix* top;
    Matrix* bottom;
    for (int x = 0; x < n; ++x) {
        IntList col(n);                     // make new column

        if (x == 0){
            top = &a;
            bottom = &c;
        } else if (x == _Width / 2) {
            top = &b;
            bottom = &d;
        }

        for (int y = 0; y < _Height; ++y ) {
            if ( x < _Width / 2){
                if ( y < _Width / 2){
                    col[y] = top->_Data[x][y];
                } else {
```

```
                    col[y] = bottom->_Data[x][y - n / 2];
                }
            } else {
                if ( y < _Width / 2){
                    col[y] = top->_Data[x - n / 2][y];
                } else {
                    col[y] = bottom->_Data[x - n / 2][y - n / 2];
                }
            }
        }
        _Data[x] = col;                     // add column to IntLList
    }
     //cout<<"Done recombining"<<endl;
}


//
// add matrices
//
Matrix Matrix::operator+ ( const Matrix& other) const {

    IntLList mData(_Width);                 // Data for resultant Matrix

    // ensure matrices are compatable
    if (_Width == other._Width && _Height == other._Height) {

        // add elements
        for (int x = 0; x < _Width; ++x) {
            IntList col(_Height);       // make new column
            for (int y = 0; y < _Height; ++y )
                  col[y] = _Data[x][y] + other._Data[x][y];
             mData[x] = col;                // add column to IntLList
        }

    } else {
        // Matrices not compatable.
        cout << "Can't add these matrices" << endl;
    }

    return Matrix(mData);
}

//
// subtract matrices
//
Matrix Matrix::operator- ( const Matrix& other) const {

    IntLList mData(_Width);                     // Data for resultant Matrix

    // ensure matrices are compatable
    if (_Width == other._Width && _Height == other._Height) {

        // add elements
```

```
        for (int x = 0; x < _Width; ++x) {
            IntList col(_Height);           // make new column
            for (int y = 0; y < _Height; ++y )
                col[y] = _Data[x][y] - other._Data[x][y];
            mData[x] = col;             // add column to IntLList
        }

    } else {
        // Matrices not compatable.
        cout << "Can't subtract these matrices" << endl;
    }

    return Matrix(mData);
}


//
// multiply matrices
//
Matrix Matrix::operator* ( const Matrix& other) const {

    // Matrix will have m's height and n's width
    int height = _Height;
    int width  = other._Width;

     IntLList mData(width);                 // Data for resultant Matrix

    for (int x = 0; x < width; ++x) {
        IntList col(height);                // make new column
        for (int y = 0; y < height; ++y ) {
            int val = 0;
            for (int i = 0; i < height; ++i )
                val += (_Data[i][y] * other._Data[x][i] );
            col[y] = val;                   // add column to Matrix
        }
        mData[x] = col;
    }

    return Matrix(mData);
}


//
//   print Matrix
//
void Matrix::print() const {

    for (int y = 0; y < _Height; ++y) {
        cout << "[ ";

        for (int x = 0; x < _Width; ++x) {
            cout << _Data[x][y] << " ";
        }
```

```
        cout << "]" << endl;
    }

    cout << endl;
}


//
//   print Matrix
//
void Matrix::printDiag() const {

     cout << "[ ";

    for (int y = 0; y < _Height; ++y)
        for (int x = 0; x < _Width; ++x)
             if (x == y)
                  cout << _Data[x][y] << " ";

    cout << "]" << endl;

    cout << endl;
}


//
//  Partition Matrix into quarters
//           Note: Matrix assumed to be square and even [div. by 2] in size.
MatrixList Matrix::makeQuarters() {

    MatrixList mReturn(4);

    int n = _Width / 2;                     // small matrix size

    int data[4][n * n];                     // data for each new 1/4
    int counter[] = {-1, -1, -1, -1};
    int quarter;                          // quarter being used

    for (int y = 0; y < (_Width); ++y ) {
        for (int x = 0; x < (_Width); ++x) {

            if (x <= n-1 && y <= n-1)
                quarter = 0;              // top left
            else if (x >  n-1 && y <= n-1)
                quarter = 1;              // top right
            else if (x <= n-1 && y >  n-1)
                quarter = 2;              // bottom left
            else if (x >  n-1 && y >  n-1)
                quarter = 3;              // bottom right

            data[quarter][++counter[quarter]] = _Data[x][y];
        }
```

```
    }

    for (int i = 0; i < 4; ++i)
        mReturn[i] = Matrix(n, n, data[i]);

    return mReturn;
}


void Matrix::addPadding() {
    for (int i = 0; i < _Width; ++i) {
        _Data[i].push_back(0);          // add 0 to each column
    }
    ++_Height;

    IntList col;                        // create padded column
    for (int i = 0; i < _Height; ++i)
        col.push_back(0);

    _Data.push_back( col );                 // add new column
    ++_Width;
}


void Matrix::removePadding() {
    _Data.pop_back();                   // remove last colum of 0s
    --_Width;

    for (int i = 0; i < _Width; ++i) {
        _Data[i].pop_back();            // remove 0 from each column
    }
    --_Height;
}


//
//  accessors and mutators
//
void Matrix::setAt(int x, int y, int val)    { _Data[x][y] = val;  }
int      Matrix::getAt(int x, int y)                 { return _Data[x][y]; }
int      Matrix::getWidth()                          { return _Width;      }
int      Matrix::getHeight()                         { return _Height;     }



//
//
//  strassen.cpp
//
//      AUTHOR: Russell Lowke
//
//          Date: 4/14/2004
//
```

```
#include <iostream>
#include <cstring>
#include <string>
#include <fstream>

using std::string;

#include "Matrix.h"

void wait ( int seconds ) {
     clock_t endwait;
     endwait = clock() + seconds * CLK_TCK ;
     while (clock() < endwait) {}
}


Matrix mms(Matrix m, Matrix n, int d) {

    Matrix mReturn;

    if (d == 1)
        mReturn = m * n;                    // resultant 1 x 1 matrix
    else {

        bool paddingAdded = false;
        if (d % 2 != 0 ) {
            m.addPadding();
            n.addPadding();
            ++d;
            paddingAdded = true;
        }

        // separate each matrix into 4 smaller matrices
        //           [ A  B ]            [ E  F ]
        //           [ C  D ]            [ G  H ]

        //           [ mGrp[0]  mGrp[1] ][ nGrp[0]  nGrp[1] ]
        //           [ mGrp[2]  mGrp[3] ][ nGrp[2]  nGrp[3] ]
        MatrixList mGrp = m.makeQuarters();
        MatrixList nGrp = n.makeQuarters();
        int sd = mGrp[0].getWidth();

        // compute P1 thru 7 with recursive calls to mms

        // p1 = A(F - H)
        Matrix p1 = mms(mGrp[0], (nGrp[1] - nGrp[3]), sd);
        // p2 = (A + B)H
        Matrix p2 = mms((mGrp[0] + mGrp[1]), nGrp[3], sd);
        // p3 = (C + D)E
        Matrix p3 = mms((mGrp[2] + mGrp[3]), nGrp[0], sd);
        // p4 = D(G - E)
        Matrix p4 = mms(mGrp[3], (nGrp[2] - nGrp[0]), sd);
```

```
        // p5 = (A + D)(E + H)
        Matrix p5 = mms((mGrp[0] + mGrp[3]), (nGrp[0] + nGrp[3]), sd);
        // p6 = (B - D)(G + H)
        Matrix p6 = mms((mGrp[1] - mGrp[3]), (nGrp[2] + nGrp[3]), sd);
        // p7 = (A - C)(E + F)
        Matrix p7 = mms((mGrp[0] - mGrp[2]), (nGrp[0] + nGrp[1]), sd);

        /*
        Matrix p1 = mGrp[0] * (nGrp[1] - nGrp[3]);                  // p1 = A(F -
H)
        Matrix p2 = (mGrp[0] + mGrp[1]) * nGrp[3];                  // p2 = (A +
B)H
        Matrix p3 = (mGrp[2] + mGrp[3]) * nGrp[0];                  // p3 = (C +
D)E
        Matrix p4 = mGrp[3] * (nGrp[2] - nGrp[0]);                  // p4 = D(G -
E)
        Matrix p5 = (mGrp[0] + mGrp[3]) * (nGrp[0] + nGrp[3]);// p5 = (A + D)(E
+ H)
        Matrix p6 = (mGrp[1] - mGrp[3]) * (nGrp[2] + nGrp[3]);// p6 = (B - D)(G
+ H)
        Matrix p7 = (mGrp[0] - mGrp[2]) * (nGrp[0] + nGrp[1]);// p7 = (A - C)(E
+ F)
        */


        // compute [ AE + BG   AF + BH ]
        //         [ CE + DG   CF + DH ]

        // AE + BG  = p5 + p4 - p2 + p6
        // AF + BH  = p1 + p2
        // CE + DG  = p3 + p4
        // CF + DH  = p5 + p1 - p3 - p7

        // fill resultant matrix and return
        mReturn = Matrix( p5 + p4 - p2 + p6,
                          p1 + p2,
                          p3 + p4,
                          p5 + p1 - p3 - p7,
                          d );

        if (paddingAdded)
            mReturn.removePadding();
    }

    return mReturn;
}
//
// multiply matrices strassen   m x n
//
Matrix mms(Matrix m, Matrix n) {

    // get dimension
    int d = m.getWidth();
```

```
    if (d != m.getHeight() ||
        d != n.getWidth()  ||
        d != n.getHeight()) {

        cout << "Cannot Strassen multiply matrices of different dimensions." <<
endl;
        return Matrix();

    } else
        return mms(m, n, d);
}

MatrixList readFile (string filename, int d) {

    MatrixList mReturn;
    IntLList dataA, dataB;
    Matrix matrixA, matrixB;

    cout << "Creating document from " + filename << endl;

    fstream inputFile(filename.c_str(), ios::in);

    if (!inputFile)
        cerr << "ERR: File not found" << endl;
    else {
        int val, nItems = d * 2;

        for (int i = 0; i < nItems; ++i) {

            IntList col;                        // init column
            for(int j = 0; j < d; ++j) {
                inputFile>>val;
                col.push_back( val );
            }

            if (dataA.size() < d)
                dataA.push_back( col );         // add to first Matrix
            else
                dataB.push_back( col );         // add to second Matrix
        }

        matrixA = Matrix(dataA);
        matrixB = Matrix(dataB);
    }

    mReturn.push_back( matrixA );
    mReturn.push_back( matrixB );

    return mReturn;

}
```

```
int main(int argc, char* argv[]) {

    // strassen 0 dimension textfile

    int d;
    string filename;

    if (argc == 4) {
        d        = atoi(argv[2]);
        filename = argv[3];
    } else {
        cerr << "No arguments: using defaults" << endl;
        d = 3;
        filename = "Matrix.txt";
    }

    // read textfile.
    MatrixList data = readFile(filename, d);

    Matrix matrixA = data[0];
    Matrix matrixB = data[1];

    matrixA.printDiag();
    cout << "x" << endl;
    matrixB.printDiag();

     cout << "TIME Start:" << clock() << endl;
    cout << "=" << endl;
    Matrix matrixC = matrixA * matrixB;
    matrixC.printDiag();

     cout << "TIME prestrassen:" << clock() << endl;

    cout << "Strassen" << endl;
    Matrix matrixD = mms(matrixA, matrixB);
    matrixD.printDiag();

     cout << "TIME End:" << clock() << endl;

}
```