|  |
| --- |
| **Ch11-03:** *Matrix* |
| **Line#** | **Code** |
| 123456789101112131415161718192021222324252627282930313233343536373839404142434445464748495051525354555657585960616263646566676869707172737475767778798081828384858687888990919293949596979899100101102103104105106107108109110111112113114115116117118119120121122123124125126127128129130131132133134135136137138139140141142143144145146147148149150151152153154155156157158159160161162163164165166167168169170171172173174175176177178179180181182183184185186187188189190191192193194 | #include <iostream>#include <cstdarg>using namespace std;class Matrix { unsigned NRow, NCol; int\* Cells; Matrix(unsigned nRow, unsigned nCol, int\* cells) : NRow(nRow), NCol(nCol), Cells(cells) { //cout << "Calling Matrix(unsigned nRow, unsigned nCol, int\* cells)" << endl; }public: Matrix(unsigned nRow, unsigned nCol, ...) :NRow(nRow), NCol(nCol) { Cells = new int[nRow \* nCol]; va\_list args; va\_start(args, nCol); for (unsigned i = 0; i < nRow \* nCol; i++) Cells[i] = va\_arg(args, int); va\_end(args); //cout << "Calling Matrix(unsigned nRow, unsigned nCol, ...)" << endl; } Matrix(const Matrix& m) :NRow(m.NRow), NCol(m.NCol) {//Copy Constructor Cells = new int[NRow \* NCol]; memcpy(Cells, m.Cells, NRow \* NCol \* sizeof(int)); //cout << "Calling Matrix(const Matrix& m)" << endl; } ~Matrix() { //cout << "Calling ~Matrix()" << endl; if (Cells) delete[] Cells; } void Show() { for (unsigned r = 0; r < NRow; r++) { for (unsigned c = 0; c < NCol; c++) { cout << Cells[r \* NCol + c] << '\t'; } cout << endl; } } //Matrix Multiply(const Matrix& rhs) const{ Matrix operator\*(const Matrix& rhs) const { if (NCol != rhs.NRow) throw "Cannot Multiply-lah!"; unsigned nRow = NRow; unsigned nCol = rhs.NCol; int\* cells = new int[nRow \* nCol]; for (unsigned r = 0; r < nRow; r++) { for (unsigned c = 0; c < nCol; c++) { cells[r \* nCol + c] = 0; for (unsigned i = 0; i < NCol; i++) { cells[r \* nCol + c] += Cells[r \* NCol + i] \* rhs.Cells[i \* rhs.NCol + c]; } } } return Matrix(nRow, nCol, cells); } Matrix operator\*(int rhs) const { unsigned nRow = NRow; unsigned nCol = NCol; int\* cells = new int[nRow \* nCol]; for (unsigned r = 0; r < nRow; r++) { for (unsigned c = 0; c < nCol; c++) { cells[r \* nCol + c] = Cells[r \* NCol + c] \* rhs; } } return Matrix(nRow, nCol, cells); } //Matrix Add(const Matrix& rhs) const { Matrix operator+(const Matrix& rhs) const { if ((NRow != rhs.NRow) || (NCol != rhs.NCol)) throw "Cannot Add-lah!"; unsigned nRow = NRow; unsigned nCol = NCol; int\* cells = new int[nRow \* nCol]; for (unsigned r = 0; r < nRow; r++) { for (unsigned c = 0; c < nCol; c++) { cells[r \* nCol + c] = Cells[r \* NCol + c] + rhs.Cells[r \* rhs.NCol + c]; } } return Matrix(nRow, nCol, cells); } //Matrix Transpose() const { Matrix operator~() const { unsigned nRow = NCol; unsigned nCol = NRow; int\* cells = new int[nRow \* nCol]; for (unsigned r = 0; r < nRow; r++) { for (unsigned c = 0; c < nCol; c++) { cells[r \* nCol + c] = Cells[c \* NCol + r]; } } return Matrix(nRow, nCol, cells); } bool operator==(const Matrix& rhs) const { if (NRow != rhs.NRow) return false; if (NCol != rhs.NCol) return false; for (unsigned i = 0; i < (NRow \* NCol); i++) { if (Cells[i] != rhs.Cells[i]) return false; } return true; } bool operator!=(const Matrix& rhs) const { return !(\*this == rhs); } int\* operator[](unsigned row) const { if (row > NRow) throw "Row out of bound!"; return Cells + (row \* NCol); } Matrix& operator=(const Matrix& rhs) { if (Cells) delete[] Cells; NRow = rhs.NRow; NCol = rhs.NCol; Cells = new int[NRow \* NCol]; memcpy(Cells, rhs.Cells, NRow \* NCol \* sizeof(int)); return \*this; } friend Matrix operator\*(int lhs, const Matrix& rhs); friend ostream& operator<<(ostream& os, const Matrix& m);};Matrix operator\*(int lhs, const Matrix& rhs) { unsigned nRow = rhs.NRow; unsigned nCol = rhs.NCol; int\* cells = new int[nRow \* nCol]; for (unsigned r = 0; r < nRow; r++) { for (unsigned c = 0; c < nCol; c++) { cells[r \* nCol + c] = rhs.Cells[r \* rhs.NCol + c] \* lhs; } } return Matrix(nRow, nCol, cells);}//Matrix operator\*(int lhs, const Matrix& rhs) {// return rhs \* lhs;//}ostream& operator<<(ostream& os, const Matrix& m) { for (unsigned r = 0; r < m.NRow; r++) { for (unsigned c = 0; c < m.NCol; c++) { os << m.Cells[r \* m.NCol + c] << '\t'; } os << endl; } return os;}int main() { Matrix mA(3, 2, 1, 2, 3, 4, 5, 6); Matrix mB(3, 2, 2, 1, 5, 6, 8, 0); Matrix mC(3, 3, 1, 5, 0, 2, 3, 1, 4, 1, 2); //cout << ((mA \* ~mB) + mC) << endl; cout << mA << endl; //cout << (mA.operator\*(3)) << endl; cout << (3 \* mA) << endl; //if (mA!=mB) { // cout << "mA NOT the same as mB" << endl; //} //else { // cout << "mA same as mB" << endl; //} //int v = mA[1][0]; //cout << "v is " << v << endl; /\* Matrix mD = mA; //Declaration + Initialization mA = mC; // Assignment if ((mA = mC) == mB) { } cout << mA << endl; \*/ return 0;} |