|  |  |
| --- | --- |
| **Ch11-03:** *Matrix* | |
| **Line#** | **Code** |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185  186  187  188  189  190  191  192  193  194 | #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;  } |