```cpp
#include <iostream>
#include <algorithm>
#include <functional>
#include <exception>
#include "grid.h"

using namespace std;

void divideRow(Grid<double>& matrix, int row, double by)
{
    transform(matrix.row_begin(row),
              matrix.row_end(row),
              matrix.row_begin(row),
              bind2nd(divides<double>(), by));
}

int findPivotRow(Grid<double>& matrix, int col, int numFound)
{
    for(int row = numFound; row < matrix.numRows(); ++row)
        if(matrix[row][col] != 0)
            return row;
    return -1;
}

void gaussJordanEliminate(Grid<double>& matrix, int numAugmentedCols)
{
    const int numMatrixColumns = matrix.numCols() - numAugmentedCols;
    const int maxPivots = min(matrix.numRows(), numMatrixColumns);

    int pivotsPlaced = 0;
    for(int col = 0; col < numMatrixColumns && pivotsPlaced < maxPivots; ++col)
    {
        const int row = findPivotRow(matrix, col, pivotsPlaced);
        if(row == -1) continue;

        divideRow(matrix, row, matrix[row][col]);

        for(int nextRow = 0; nextRow < matrix.numRows(); ++nextRow)
        {
            /* Skip the current row and rows that have zero in this column. */
            if(nextRow == row || matrix[nextRow][col] == 0.0) continue;

            divideRow(matrix, nextRow, matrix[nextRow][col]);

            transform(matrix.row_begin(nextRow), matrix.row_end(nextRow),
                      matrix.row_begin(row), // Subtract values from new pivot r
ow
                      matrix.row_begin(nextRow), // Overwrite existing values
                      minus<double>());

            Grid<double>::iterator nonzero =
                    find_if(matrix.row_begin(nextRow), matrix.row_begin(nextRow)
 + col,
                            bind2nd(not_equal_to<double>(), 0.0));

            if(nonzero != matrix.row_begin(nextRow) + col)
                divideRow(matrix, nextRow, *nonzero);
        }
        swap_ranges(matrix.row_begin(row), matrix.row_end(row),
                    matrix.row_begin(pivotsPlaced));
        ++pivotsPlaced;
    }
```



```cpp
}

void invertMatrix(Grid<double>& matrix)
{
    if(matrix.numRows() != matrix.numCols())
        throw invalid_argument("Cannot invert a non-square matrix.");

    Grid<double> scratchWork(matrix.numRows(), matrix.numCols() * 2);

    /* Copy existing data. */
    for(int row = 0; row < matrix.numRows(); ++row)
        copy(matrix.row_begin(row), matrix.row_end(row),
             scratchWork.row_begin(row));

    /* Write the identity matrix. */
    for(int row = 0; row < matrix.numRows(); ++row)
        scratchWork[row][row + matrix.numCols()] = 1.0;

    gaussJordanEliminate(scratchWork, matrix.numCols());

    for(int row = 0; row < matrix.numRows(); ++row)
        for(int col = 0; col < matrix.numCols(); ++col)
            if((row == col && scratchWork[row][col] != 1.0) ||
                (row != col && scratchWork[row][col] != 0.0))
                throw invalid_argument("Singular matrix.");

    for(int row = 0; row < matrix.numRows(); ++row)
        copy(scratchWork.row_begin(row) + matrix.numCols(),
             scratchWork.row_end(row),
             matrix.row_begin(row));
}

int main() {

    Grid<double> matrix(3, 5);
    matrix[0][0] = 0; matrix[0][1] = 5; matrix[0][2] = 10; matrix[0][3] = -1; ma
trix[0][4] = -7;
    matrix[1][0] = 2; matrix[1][1] = 6; matrix[1][2] = 12; matrix[1][3] = 10; ma
trix[1][4] = 20;
    matrix[2][0] = 2; matrix[2][1] = 3; matrix[2][2] = 6;  matrix[2][3] = 11; ma
trix[2][4] = 25;

    cout << matrix << endl;

    gaussJordanEliminate(matrix, 1);

    cout << matrix << endl;

    matrix.resize(3, 3);
    matrix[0][0] = 3; matrix[0][1] = 1; matrix[0][2] = 4;
    matrix[1][0] = 1; matrix[1][1] = 5; matrix[1][2] = 9;
    matrix[2][0] = 2; matrix[2][1] = 6; matrix[2][2] = 5;

    cout << matrix << endl;

    invertMatrix(matrix);

    cout << matrix << endl;

    return 0;
}
```