```cpp
#include <iostream>
#include "row.h"
using namespace std;

// parameterized constructor
Row::Row(int length)
{
    if (length <= 0)
    {
        throw std::out_of_range("The length has to be greater than 0");
    }
    this->length = length; // this->length is making the length for the Row, while length is the length that is input
    row_data = new double[length];
    clear();
}

// copy constructor
Row::Row(const Row &from)
{
    length = from.length;
    row_data = new double[length];
    for (int i = 0; i < length; i++)
    {
        row_data[i] = from.row_data[i];
    }
}

// destructor
Row::~Row()
{
    delete[] row_data;
}

// access operator (const)
double Row::operator[](int column) const
{
    if (column < 0 || column >= length)
    {
        throw out_of_range("Column must be >= 0 and < length");
    }
    return row_data[column];
}

// access operator (non-const)
double &Row::operator[](int column)
{
    if (column < 0 || column >= length)
    {
        throw out_of_range("Column must be >= 0 and < length");
    }
    return row_data[column];
}

// assignment operator
Row &Row::operator=(const Row &rhs)
{
    if (this != &rhs)
    {
        length = rhs.length;
        delete[] row_data;
        row_data = new double[length];
        for (int i = 0; i < length; i++)
        {
            this->row_data[i] = rhs.row_data[i];
        }
    }
    return *this;
}

// clear row data
void Row::clear()
{
    for (int i = 0; i < length; i++)
    {
        row_data[i] = 0;
    }
}
```

```
#ifndef row_h
#define row_h
class Row{
    public:
        /* Parameterized constructor
         * Takes in length and creates a row matrix with values cleared
         * to zero
         * Should verify length > 0
         */
        Row(int length);

        /* Copy constructor
         * Create a new row matrix with the same size and values as the
         * from matrix
         */
        Row(const Row& from);

        /* Destructor
         * Correctly delete any heap memory
         */
        ~Row();

        /* Access operator (const version)
         * Allow access to row matrix data
         * Should return an exception if column is too large
        */
        double operator[](int column) const;

        /* Access operator (non const version)
         * Allow access to row matrix data
         * Should return an exception if column is too large
         */
        double& operator[] (int column);

        /* Assignment operator
         * 1. Check if two sides are the same object
         * 2. Delete the current row matrix
         * 3. Create a new row matrix with the same size and values as
         *    the rhs matrix
         */
        Row& operator= (const Row& rhs);

        /* Clear all data values to zero
         */
        void clear();
    private:
        // Row matrix data
        double * row_data;
        // Size of row matrix
        unsigned int length;
};
#endif
```

```cpp
#include <iostream>
#include "row.h"
using namespace std;
int main()
{
    // test constructor
    Row R1(3);
    cout << "R1: " << R1[0] << " " << R1[1] << " " << R1[2] << endl;
    // test setting row data
    R1[0] = 0;
    R1[1] = 1;
    R1[2] = 2;
    Row R3(4);
    R3[0] = 3;
    R3[1] = 4;
    R3[2] = 5;
    R3[3] = 6;
    cout << "R1: " << R1[0] << " " << R1[1] << " " << R1[2] << endl;
    // test copy constructor
    Row R2(R1);
    cout << "R2: " << R2[0] << " " << R2[1] << " " << R2[2] << endl;
    // test assignment operator when left side is larger

    cout << "R3: " << R3[0] << " " << R3[1] << " " << R3[2] << endl;
    R3 = R2;
    cout << "R3: " << R3[0] << " " << R3[1] << " " << R3[2] << endl;
    // test self assignment
    R3 = R3;
    cout << "R3: " << R3[0] << " " << R3[1] << " " << R3[2] << endl;
    // test assignment operator when right side is larger
    Row R4(4);
    R4[0] = 7;
    R4[1] = 8;
    R4[2] = 9;
    R4[3] = 10;
    cout << "R4: " << R4[0] << " " << R4[1] << " " << R4[2] << " " << R4[3] << endl;
    R3 = R4;
    cout << "R3: " << R3[0] << " " << R3[1] << " " << R3[2] << " " << R3[3] << endl;
    cout << "R4: " << R4[0] << " " << R4[1] << " " << R4[2] << " " << R4[3] << endl;
    // test const access operator
    const Row R5 = R1;
    cout << "R5: " << R5[0] << " " << R5[1] << " " << R5[2] << endl;
    // test clear function
    R1.clear();
    cout << "R1: " << R1[0] << " " << R1[1] << " " << R1[2] << endl;
    // test constructor is throwing exceptions correctly
    try
    {
        Row R6(0);
    }
    catch (...)
    {
        cout << "Row of length 0 failed" << endl;
    }
    try
    {
        Row R7(-4);
    }
    catch (...)
    {
        cout << "Row of length -4 failed" << endl;
    }
    // test access operator is throwing exceptions correctly
    try
    {
        R1[-1];
    }
    catch (...)
    {
        cout << "column = -1 failed" << endl;
    }
    try
    {
        R1[3];
    }
    catch (...)
    {
        cout << "column = length failed" << endl;
```

```
79          }
80          return 0;
81      }
```

Table of Contents