```
/*****************************************************************************
*
*  Author:           Miklos Moreno
*  Title:            Overloaded Operators
*  Course:           2143
*  Semester:         Fall 2021
*
*  Description:
*        Use previously made MyVector class (this one is Griffins, it looks
prettier).
*        Implemented overloaded operators (+, -, =, *, /, [], ostream, fstream).
*
*
*
*  Usage:
*        Use it like a linked list now. More like a vector next program
*                                                (this is that program)
*
*  Files: output.txt
*****************************************************************************/
#include <fstream>
#include <iostream>
#include <string>

#define INF 1000000000 // infinity

using namespace std;

// Node
//             +--------+      +--------+      +--------+      +--------+
//   Head ---> |        | ---> |        | ---> |        | ---> |        | --->
NULL
//   NULL <--- |        | <--- |        | <--- |        | <--- |        |
//             +--------+      +--------+      +--------+      +--------+

// Node for our linked list
struct Node
{
    int data;

    Node *next;
    Node *prev;

    Node(int x)
    {
        data = x;
        prev = next = NULL;
    }

    /**
     * @brief Construct a new Node object and connect it to its neighbors
     *        directly in the constructor.
```

```
     *
     * @param int x - data value
     * @param Node*& p - previous node reference
     * @param Node*& n - next node reference
     */
    Node(int x, Node *&p, Node *&n)
    {
        data = x;

        // `p` was passed by address, so changes to `p` are
        // remembered! So, we point `p` to `this` ( `this` = the new node we are
in right now ).
        if (p)
        {
            p->next = this;
        }
        // Same for `n` as was for `p`.
        if (n)
        {
            n->prev = this;
        }

        // Now point `this` nodes previous and next to the nodes we passed in.

        prev = p;
        next = n;

        // Below is the same as above but we explicitly use the `this` keyword.
        // this->prev = p;
        // this->next = n;
    }
};

class MyVector
{
private:
    Node *head; // base pointer of list
    Node *tail;
    int size;
    static ofstream fout;
    string fileName;
    bool sorted;

    /**
     * @brief Private version of inOrder push.
     *
     * @param x
     */
    void _inorderPush(int x)
    {

        Node *current = head;

        while (current->data > x)
```

```
        { // loop to find proper location
            current = current->next;
        }

        Node *newNode = new Node(x, current->prev, current);

        // current->prev->next = newNode;
        // newNode->prev = current->prev;
        // newNode->next = current;
        // current->prev = newNode;

        size++; // add to size :)
    }

public:
    /**
     * @brief - Initialize the data members so we don't
     *      have duplicate lines in each constructor.
     *
     */
    void init()
    {
        head = tail = NULL;
        fileName = "";
        size = 0;
        sorted = 0;
    }

    /**
     * @brief Default constructor
     *
     */
    MyVector()
    {
        init();
    }

    /**
     * @brief Overloaded Constructor
     *
     * @param int   *A - pointer to array
     * @param int   aSize - size of array
     */
    MyVector(int A[], int aSize)
    {
        init();

        for (int i = 0; i < aSize; i++)
        {
            pushRear(A[i]);
        }
    }

    /**
```

```
     * @brief Overloaded Constructor
     *
     * @param string FileName - file to open and read
     *
     * Assumes infile will contain numbers only delimited by spaces or
     * new lines.
     */
    MyVector(string FileName)
    {
        init();

        ifstream fin;
        int x = 0;

        fin.open(FileName);
        while (!fin.eof())
        {
            fin >> x;
            pushRear(x);
        }
    }

    /**
     * @brief Copy Constructor
     *
     * @param MyVector &other
     */
    MyVector(const MyVector &other)
    {
        init();

        Node *temp = other.head;

        while (temp)
        {
            pushRear(temp->data);
            temp = temp->next;
        }
    }

    /**
     * @brief Public version of inOrder push.
     *
     * @param x
     */
    void inorderPush(int x)
    {
        if (!sorted)
        {
            sortList();
        }
        if (!head)
        {
            pushFront(x); // call push front for empty list (or pushRear would
```

```
work)
        }
        else if (x < head->data)
        {
            pushFront(x); // call push front if x is less than head
        }
        else if (x > tail->data)
        {
            pushRear(x); // call push rear if x > tail
        }
        else
        {
            _inorderPush(x); // call private version of push in order
        }
    }

    /**
     * @brief Sort the current values in the linked list. This doesn't require any
changing
     * because its a doubly linked list as we are swapping DATA not actual nodes.
     *
     * @returns None
     */
    void sortList()
    {
        Node *newFront = head;
        while (newFront->next)
        {
            Node *smallest = newFront;
            Node *current = newFront;
            int minimum = INF;
            while (current)
            {
                if (current->data < minimum)
                {
                    smallest = current;
                    minimum = current->data;
                }
                current = current->next;
            }
            smallest->data = newFront->data;
            newFront->data = minimum;
            newFront = newFront->next;
        }
        sorted = true;
    }

    /**
     * @brief Add value to front of list.
     *
     * @param x
     * @return bool - true = successful push
     */
    bool pushFront(int x)
```

```
{
    Node *newNode = new Node(x);

    // empty list make head and tail
    // point to new value
    if (!head)
    {
        head = tail = newNode;
        // otherwise adjust head pointer
    }
    else
    {
        newNode->next = head;
        head->prev = newNode;
        head = newNode;
    }
    size++;
    return true;
}

/**
 * @brief This method loads values from 'other' list in 'this' list.
 *          It loads an array first so we can process the values in
 *          reverse so they end up on 'this' list in the proper order.
 *          If we didn't use the array, we would reverse the values
 *          from the 'other' list.
 *
 * @depends - Uses `pushFront(int)`
 * @param MyVector& other
 * @return None
 */
void pushFront(const MyVector &other)
{
    Node *otherPtr = other.head;          // get copy of other lists head
    int *tempData = new int[other.size]; // allocate memory to hold values

    // load other list into array
    int i = 0;
    while (otherPtr)
    {
        tempData[i] = otherPtr->data;
        otherPtr = otherPtr->next;
        ++i;
    }

    // process list in reverse in order to keep them
    // in their original order.
    for (int i = other.size - 1; i >= 0; i--)
    {
        pushFront(tempData[i]);
    }
}

/**
```

```
 * @brief -  Add 'other' list's values to end of 'this' list.
 * @note - Uses `pushRear(int)`
 * @param MyVector& other
 * @return None
 */
void pushRear(const MyVector &other)
{
    Node *otherPtr = other.head; // get copy of other lists head

    while (otherPtr)
    { // traverse and add
        pushRear(otherPtr->data);
        otherPtr = otherPtr->next;
    }
}

/**
 * @brief - Add value to rear of list
 *
 * @param int x - value to be added
 * @return bool - successful push = 1
 */
bool pushRear(int x)
{
    Node *newNode = new Node(x);

    if (!head)
    {
        head = tail = newNode;
    }
    else
    {
        tail->next = newNode;
        newNode->prev = tail;
        tail = newNode;
    }
    size++; // add to size of list
    return true;
}

/**
 * @brief Push value onto list at soecified position, if it exists.
 *
 * @param int i - location index
 * @param inr x - value to add
 * @return bool - true add successful / false add failed
 */
bool pushAt(int i, int x)
{
    // IF index is at the end of the list
    // OR beyond the end, do a push rear,
    if (i >= size - 1)
    {
        return pushRear(x);
```

```
        }

        if (i == 0)
        {
            return pushFront(x);
        }

        // Index is not front or rear so ... find proper
        //

        Node *newNode = new Node(x); // allocate new node
        Node *current = head;

        while (i > 0)
        { // loop to find proper location
            current = current->next;
            i--;
        }

        // newNode is getting placed in front of current

        // update temp's previous pointer to point to node before current
        // update temp's next to point to current (making current come after
newNode)
        newNode->prev = current->prev;
        newNode->next = current;

        // update node before current to now point to newNode
        // and update current to point back to newNode
        current->prev->next = newNode;
        current->prev = newNode;

        // current->next is already pointing to proper node so we leave it.

        size++; // add to size :)
        return true;
    }

    /**
     * @brief Write LL values to console using "<<"
     *
     * @param ostream - need access to cout
     * @param MyVector - LL that will be printed to console
     * @return ostream - give cout back
     */
    friend ostream &operator<<(ostream &os, const MyVector &rhs)
    {
        Node *temp = rhs.head; // temp pointer copies head

        while (temp)
        { // this loops until temp is NULL
            // same as `while(temp != NULL)`

            os << temp->data; // print data from Node
```

```
            if (temp->next)
            {
                os << "->";
            }
            temp = temp->next; // move to next Node
        }
        os << endl;

        return os;
    }

    /**
     * @brief Write LL values to outfile using "<<"
     *
     * @param fstream - need access to fstream
     * @param MyVector - LL that will be printed to file
     * @return ostream - give fstream back
     */
    friend fstream &operator<<(fstream &os, const MyVector &rhs)
    {
        Node *temp = rhs.head; // temp pointer copies head

        while (temp)
        { // this loops until temp is NULL
            // same as `while(temp != NULL)`

            os << temp->data; // print data from Node
            if (temp->next)
            {
                os << "->";
            }
            temp = temp->next; // move to next Node
        }
        os << endl;

        return os;
    }

    /**
     * @brief traverses LL like an array
     *
     * @param index - num of times to loop LL
     */
    int &operator[](int index)
    {
        if (index < 0 || index >= size)
        {
            cout << "invalid index" << endl;
            exit(0);
        }
        else
        {
            Node *temp = head;
            for (int i = 0; i < index; i++)
```

```
            {
                temp = temp->next;
            }
            return temp->data;
        }
    }

    /**
     * @brief sets this equal to other
     *
     * @param MyVector - LL on right side of operator
     * @return MyVector - sets new LL to values of rhs
     */
    MyVector &operator=(const MyVector &rhs)
    {
        if (this == &rhs)
        {
            return *this;
        }

        this->head = rhs.head;
        this->tail = rhs.tail;

        return *this;
    }

    /**
     * @brief Checks if two LL are equal
     *
     * @param MyVector - compared to this for equality
     * @return bool - T or F
     */
    bool operator==(const MyVector &rhs)
    {
        bool returnthis;
        MyVector other = rhs;
        if (this->size != other.size)
        {
            returnthis = false;
        }
        else
        {
            for (int i = 0; i < size; i++)
            {
                returnthis = (*this)[i] == other[i];
            }
        }
        return returnthis;
    }

    /**
     * @brief destructive operator that adds rhs values to this
     *        difference in size means nodes that did not change
     *
```

```
     * @param MyVector - values being added
     * @return MyVector - returns LL after operator finishes
     */
    MyVector &operator+=(const MyVector &rhs)
    {
        MyVector other = rhs;

        if (this->size >= other.size)
        {
            for (int i = 0; i < other.size; i++)
            {
                (*this)[i] += other[i];
            }
        }
        else
        {
            for (int i = 0; i < this->size; i++)
            {
                (*this)[i] += other[i];
            }
            this->~MyVector();
            this->pushRear(other);
        }
        return *this;
    }

    /**
     * @brief destructive operator that subtracts rhs values to this
     *        difference in size means nodes that did not change
     *
     * @param MyVector - values being subtracted
     * @return MyVector - returns LL after operator finishes
     */
    MyVector &operator-=(const MyVector &rhs)
    {
        MyVector other = rhs;

        if (this->size >= other.size)
        {
            for (int i = 0; i < other.size; i++)
            {
                (*this)[i] -= other[i];
            }
        }
        else
        {
            for (int i = 0; i < this->size; i++)
            {
                (*this)[i] -= other[i];
            }
            this->~MyVector();
            this->pushRear(other);
        }
        return *this;
```

```
    }

    /**
     * @brief destructive operator that multiplies rhs values to this
     *        difference in size means nodes that did not change
     *
     * @param MyVector - values being multiplied
     * @return MyVector - returns LL after operator finishes
     */
    MyVector &operator*=(const MyVector &rhs)
    {
        MyVector other = rhs;

        if (this->size >= other.size)
        {
            for (int i = 0; i < other.size; i++)
            {
                (*this)[i] *= other[i];
            }
        }
        else
        {
            for (int i = 0; i < this->size; i++)
            {
                (*this)[i] *= other[i];
            }
            this->~MyVector();
            this->pushRear(other);
        }
        return *this;
    }

    /**
     * @brief destructive operator that divides rhs values to this
     *        difference in size means nodes that did not change
     *
     * @param MyVector - values divinding this
     * @return MyVector - returns LL after operator finishes
     */
    MyVector &operator/=(const MyVector &rhs)
    {
        MyVector other = rhs;

        if (this->size >= other.size)
        {
            for (int i = 0; i < other.size; i++)
            {
                if (other[i] == 0)
                {
                    cout << "Division by 0" << endl;
                }
                else
                {
                    (*this)[i] /= other[i];
```

```
                }
            }
        }
        else
        {
            for (int i = 0; i < this->size; i++)
            {
                if (other[i] == 0)
                {
                    cout << "Division by 0" << endl;
                }
                else
                {
                    (*this)[i] /= other[i];
                }
            }
            this->~MyVector();
            this->pushRear(other);
        }
        return *this;
    }
    MyVector operator/(const MyVector &rhs)
    {
        return MyVector(*this) /= rhs;
    }

    MyVector operator+(const MyVector &rhs)
    {
        return MyVector(*this) += rhs;
    }

    MyVector operator-(const MyVector &rhs)
    {
        return MyVector(*this) -= rhs;
    }

    MyVector operator*(const MyVector &rhs)
    {
        return MyVector(*this) *= rhs;
    }

    /**
     * @brief Destroy the My Vector object
     *
     */
    ~MyVector()
    {
        Node *current = head;
        Node *prev = head;

        while (current)
        {
            prev = current;
            current = current->next;
```

```
            //cout << "deleting: " << prev->data << endl;
            delete prev;
        }
    }
};

int main()
{
    int a1[] = {1, 2, 3, 4, 5};
    int a2[] = {10, 20, 30};

    MyVector v1(a1, 5);
    MyVector v2(a2, 3);

    ofstream fout;
    fout.open("output.txt");

    cout << v1[2] << endl;
    // writes out 3

    v1[4] = 9;
    // v1 now = [1,2,3,4,9]

    cout << v1 << endl;
    // writes out [1,2,3,4,9] to console.

    fout << v1 << endl;
    // writes out [1,2,3,4,9] to your output file.

    MyVector v3 = v1 + v2;
    cout << v3 << endl;
    // writes out [11,22,33,4,9] to console.

    v3 = v1 - v2;
    cout << v3 << endl;
    // writes out [-9,-18,-27,4,9] to console.

    v3 = v2 - v1;
    cout << v3 << endl;
    // writes out [9,18,27,4,9] to console.

    v3 = v2 * v1;
    cout << v3 << endl;
    // writes out [10,40,90,4,9] to console.

    v3 = v1 * v2;
    cout << v3 << endl;
    // writes out [10,40,90,4,9] to console.

    v3 = v1 / v2;
    cout << v3 << endl;
    // writes out [0,0,0,4,9] to console.

    v3 = v2 / v1;
```

```
    cout << v3 << endl;
    // writes out [10,10,10,4,9] to console.

    cout << (v2 == v1) << endl;
    // writes 0 to console (false) .

    MyVector v4 = v1;
    cout << (v4 == v1) << endl;
    // writes 1 to console (true) .
}
```