ENSF 694 – Summer 2024

**Principles of Software Development II**

**University of Calgary**

**Lab Assignment 4**

Student Name: Yael Gonzalez

Instructor: M. Moussavi, PhD, Peng

Submission Date: July 26, 2024

# Exercise A

## Part 1

### Source code

#### lookupTable.h

/\*\*

 \*  File Name: lookupTable.h

 \*  Assignment: ENSF 694 Summer 2024 - Lab 4 Exercise A Part 1

 \*  Created by: Mahmood Moussavi

 \*  Completed by: Yael Gonzalez

 \*  Submission Date: July 26, 2024

 \*/

#ifndef LOOKUPTABLE\_H

#define LOOKUPTABLE\_H

#include <iostream>

using namespace std;

// class LookupTable: GENERAL CONCEPTS

//

//    key/datum pairs are ordered.  The first pair is the pair with

//    the lowest key, the second pair is the pair with the second

//    lowest key, and so on.  This implies that you must be able to

//    compare two keys with the < operator.

//

//    Each LookupTable has an embedded iterator class that allows users

//    of the class to traverse trhough the list and  have acess to each

//    node.

//    In this version of the LookupTable a new struct type called Pair

//    is introduced which represents a key/data pair.

typedef *string* *Type*;

struct *Pair*

{

    int key;

*Type* datum;

    // This ctor is writtent for convenience in creating objects of Pair and copy

    Pair(int *keyA*, *Type* *datumA*) : key(*keyA*), datum(*datumA*) {}

};

struct *LT\_Node*

{

*Pair* pairM;

*LT\_Node* \*nextM;

    // This ctor should be convenient in insert and copy operations.

    LT\_Node(const *Pair* &*pairA*, *LT\_Node* \**nextA*) : pairM(*pairA*), nextM(*nextA*) {}

    // PROMISES: initializes the data members pairM and nextM, with pairA and nextA

    // respectively

};

class *LookupTable*

{

public:

    // Nested class

    LookupTable() : sizeM(0), headM(nullptr), cursorM(nullptr) {}

    // PROMISES: An empty LookupTable object with all data members. Setting size, cursor and

    // head to zero or nullptr

    // copy ctor

    LookupTable(const *LookupTable* &*source*);

    // assignment operator

*LookupTable* &operator=(const *LookupTable* &*rhs*);

    // dtor

    ~LookupTable();

*LookupTable* &begin();

    // PROMISES: Moves cursorM to the beginning of the list

    int size() const;

    // PROMISES: Returns number of keys in the table.

    int cursor\_ok() const;

    // PROMISES:

    //   Returns 1 if the cursor is attached to a key/datum pair,

    //   and 0 if the cursor is in the off-list state.

    const int &cursor\_key() const;

    // REQUIRES: cursor\_ok()

    // PROMISES: Returns key of key/datum pair to which cursor is attached.

    const *Type* &cursor\_datum() const;

    // REQUIRES: cursor\_ok()

    // PROMISES: Returns datum of key/datum pair to which cursor is attached.

    void insert(const *Pair* &*pairA*);

    // PROMISES:

    //   If keyA matches a key in the table, the datum for that

    //   key is set equal to datumA.

    //   If keyA does not match an existing key, keyA and datumM are

    //   used to create a new key/datum pair in the table.

    //   In either case, the cursor goes to the off-list state.

    int remove(const int &*keyA*);

    // PROMISES:

    //   If keyA matches a key in the table, the corresponding

    //   key/datum pair is removed from the table.

    //   If keyA does not match an existing key, the table is unchanged.

    //   In either case, the cursor goes to the off-list state.

    void find(const int &*keyA*);

    // PROMISES:

    //   If keyA matches a key in the table, the cursor is attached

    //   to the corresponding key/datum pair.

    //   If keyA does not match an existing key, the cursor is put in

    //   the off-list state.

    void go\_to\_first();

    // PROMISES: If size() > 0, cursor is moved to the first key/datum pair

    //   in the table.

    void step\_fwd();

    // REQUIRES: cursor\_ok()

    // PROMISES:

    //   If cursor is at the last key/datum pair in the list, cursor

    //   goes to the off-list state.

    //   Otherwise the cursor moves forward from one pair to the next.

    void make\_empty();

    // PROMISES: size() == 0.

    void display() const;

    // PROMISES: displays the values o

    bool isEmpty() const;

    // PROMISES: returns true is list is empty

    int \*retrieve\_at(int *i*);

    // PROMISES: returns the adress of the key at the position index. Reminder: the index

    // number for the first node in the list is 0, 2nd node is 1, 3rd node is 2 and so on.

private:

    int sizeM;        // size of list (number of availble nodes)

*LT\_Node* \*headM;   // pointer to the first node in the list

*LT\_Node* \*cursorM; // pointer that can travers through the list

    void destroy();

    // Deallocate all nodes, set headM to zero.

    void copy(const *LookupTable* &*source*);

    // Establishes \*this as a copy of source.  Cursor of \*this will

    // point to the twin of whatever the source's cursor points to.

};

#endif

#### lookupTable.cpp

/\*\*

 \*  File Name: lookupTable.cpp

 \*  Assignment: ENSF 694 Summer 2024 - Lab 4 Exercise A Part 1

 \*  Created by: Mahmood Moussavi

 \*  Completed by: Yael Gonzalez

 \*  Submission Date: July 26, 2024

 \*/

#include "lookupTable.h"

*LookupTable*::LookupTable(const *LookupTable* &*source*)

{

    copy(*source*);

}

*LookupTable* &*LookupTable*::operator=(const *LookupTable* &*rhs*)

{

    if (*this* != &*rhs*)

    {

        destroy();

        copy(*rhs*);

    }

    return \**this*;

}

*LookupTable*::~LookupTable()

{

    destroy();

}

*LookupTable* &*LookupTable*::begin()

{

    cursorM = headM;

    return \**this*;

}

int *LookupTable*::size() const

{

    return sizeM;

}

int *LookupTable*::cursor\_ok() const

{

    return cursorM != nullptr;

}

const int &*LookupTable*::cursor\_key() const

{

    if (cursorM == nullptr)

    {

        cerr << "Cursor is pointing to null.";

    }

    return cursorM->pairM.key;

}

const *Type* &*LookupTable*::cursor\_datum() const

{

    if (cursorM == nullptr)

    {

        cerr << "Cursor is pointing to null.";

    }

    return cursorM->pairM.datum;

}

void *LookupTable*::insert(const *Pair* &*pairA*)

{

*LT\_Node* \*new\_node = **new** *LT\_Node*(*pairA*, nullptr);

    // Case 1: List is empty

    if (headM == nullptr)

    {

        headM = new\_node;

        sizeM++;

    }

    // Case 2: First node's key is greater than new insert

    else if (*pairA*.key < headM->pairM.key)

    {

        new\_node->nextM = headM;

        headM = new\_node;

        sizeM++;

    }

    // Case 3: First node's key is equal to new insert

    else if (*pairA*.key == headM->pairM.key)

    {

        headM->pairM.datum = *pairA*.datum; // update datum only

**delete** new\_node;                  // delete unnecessary node

    }

    // Case 4: New insert to be added across the list (after first node)

    else

    {

*LT\_Node* \*curr = headM;

        while (curr->nextM != nullptr && curr->nextM->pairM.key < *pairA*.key)

        {

            curr = curr->nextM;

        }

        if (curr->nextM != nullptr && curr->nextM->pairM.key == *pairA*.key)

        {

            curr->nextM->pairM.datum = *pairA*.datum; // update datum only

**delete** new\_node;                        // delete unnecessary node

        }

        else

        {

            new\_node->nextM = curr->nextM;

            curr->nextM = new\_node;

            sizeM++;

        }

    }

    cursorM = nullptr;

}

int *LookupTable*::remove(const int &*keyA*)

{

    // Case 1: List is empty

    if (headM == nullptr)

    {

        cerr << "List is empty." << endl;

        return 0;

    }

*LT\_Node* \*curr = headM;

*LT\_Node* \*prev = nullptr;

    int removed\_key;

    // Case 2: Node to be removed is the first in the list

    if (headM->pairM.key == *keyA*)

    {

        headM = headM->nextM;

        removed\_key = curr->pairM.key;

**delete** curr;

        sizeM--;

        cursorM = nullptr;

        return removed\_key;

    }

    // Case 3: Node to be removed is elsewhere in the list

    while (curr != nullptr && curr->pairM.key != *keyA*)

    {

        prev = curr;

        curr = curr->nextM;

    }

    if (curr == nullptr)

    {

        cerr << "Key not found." << endl;

        return 0;

    }

    prev->nextM = curr->nextM;

    removed\_key = curr->pairM.key;

**delete** curr;

    sizeM--;

    cursorM = nullptr;

    return removed\_key;

}

void *LookupTable*::find(const int &*keyA*)

{

    for (*LT\_Node* \*curr = headM; curr != nullptr; curr = curr->nextM)

    {

        if (curr->pairM.key == *keyA*)

        {

            cursorM = curr;

            return;

        }

    }

    cursorM = nullptr;

}

void *LookupTable*::go\_to\_first()

{

    if (sizeM > 0)

    {

        cursorM = headM;

    }

}

void *LookupTable*::step\_fwd()

{

    if (cursor\_ok())

    {

        cursorM = cursorM->nextM;

    }

}

void *LookupTable*::make\_empty()

{

    destroy();

    headM = nullptr;

    cursorM = nullptr;

    sizeM = 0;

}

void *LookupTable*::display() const

{

    if (headM == nullptr)

    {

        cerr << "List is empty.";

    }

    else

    {

        cout << " " << cursorM->pairM.key << " " << cursorM->pairM.datum << endl;

    }

}

bool *LookupTable*::isEmpty() const

{

    return headM == nullptr;

}

int \**LookupTable*::retrieve\_at(int *i*)

{

    if (*i* < 0 || *i* >= sizeM)

    {

        cerr << "Index should be positive and less than " << sizeM << endl;

        return nullptr;

    }

*LT\_Node* \*curr = headM;

    for (int j = 0; j < *i*; ++j)

    {

        curr = curr->nextM;

    }

    return &(curr->pairM.key);

}

void *LookupTable*::destroy()

{

    while (headM != nullptr)

    {

*LT\_Node* \*temp = headM;

        headM = headM->nextM;

**delete** temp;

    }

    cursorM = nullptr;

}

void *LookupTable*::copy(const *LookupTable* &*source*)

{

    if (*source*.headM == nullptr)

    {

        headM = nullptr;

        sizeM = 0;

        cursorM = nullptr;

        return;

    }

    headM = **new** *LT\_Node*(*source*.headM->pairM, nullptr);

*LT\_Node* \*srcNode = *source*.headM->nextM;

*LT\_Node* \*thisNode = headM;

    while (srcNode != nullptr)

    {

        thisNode->nextM = **new** *LT\_Node*(srcNode->pairM, nullptr);

        thisNode = thisNode->nextM;

        srcNode = srcNode->nextM;

    }

    sizeM = *source*.sizeM;

    if (*source*.cursorM != nullptr)

    {

*LT\_Node* \*srcCursor = *source*.headM;

*LT\_Node* \*newCursor = headM;

        while (srcCursor != *source*.cursorM)

        {

            srcCursor = srcCursor->nextM;

            newCursor = newCursor->nextM;

        }

        cursorM = newCursor;

    }

    else

    {

        cursorM = nullptr;

    }

}

### Program output

A screenshot of a computer program

Description automatically generated

## Part 2

### Source code

#### Point.h

/\*\*

 \*  File Name: Point.h

 \*  Assignment: ENSF 694 Summer 2024 - Lab 4 Exercise A Part 2

 \*  Created by: Mahmood Moussavi on 2024-05-10.

 \*  Completed by: Yael Gonzalez

 \*  Submission Date: July 26, 2024

 \*/

#ifndef Point\_h

#define Point\_h

#include <string.h>

class *Point*

{

public:

    Point(int *x*, int *y*, const char \**label*);

    ~Point();

    Point(const *Point* &*src*);

*Point* &operator=(const *Point* &*rhs*);

    int getx() const;

    int gety() const;

    char \*get\_label() const;

private:

    int x, y;    // x and y coordinates of a point on Cartesian plain

    char \*label; // pointer to an array allocated on the heap to store the label for a point

};

#endif /\* Point\_h \*/

#### Point.cpp

/\*\*

 \*  File Name: Point.cpp

 \*  Assignment: ENSF 694 Summer 2024 - Lab 4 Exercise A Part 2

 \*  Created by: Yael Gonzalez

 \*  Submission Date: July 26, 2024

 \*/

#include "Point.h"

*Point*::Point(int *x*, int *y*, const char \**label*) : x(*x*), y(*y*), label(**new** char[strlen(*label* + 1)])

{

    strcpy(*this*->label, *label*);

}

*Point*::~Point()

{

    delete[] label;

}

*Point*::Point(const *Point* &*src*) : x(*src*.x), y(*src*.y), label(**new** char[strlen(*src*.label + 1)])

{

    strcpy(label, *src*.label);

}

*Point* &*Point*::operator=(const *Point* &*rhs*)

{

    if (*this* != &*rhs*) // avoid self copy

    {

        x = *rhs*.x;

        y = *rhs*.y;

        delete[] label;

        label = **new** char[strlen(*rhs*.label + 1)];

        strcpy(label, *rhs*.label);

    }

    return \**this*;

}

int *Point*::getx() const

{

    return x;

}

int *Point*::gety() const

{

    return y;

}

char \**Point*::get\_label() const

{

    return label;

}

#### lookupTable.h

/\*\*

 \*  File Name: lookupTable.h

 \*  Assignment: ENSF 694 Summer 2024 - Lab 4 Exercise A Part 2

 \*  Created by: Mahmood Moussavi

 \*  Completed by: Yael Gonzalez

 \*  Submission Date: July 26, 2024

 \*/

#ifndef LOOKUPTABLE\_H

#define LOOKUPTABLE\_H

#include "Point.h"

#include <iostream>

using namespace std;

// class LookupTable: GENERAL CONCEPTS

//

//    key/datum pairs are ordered.  The first pair is the pair with

//    the lowest key, the second pair is the pair with the second

//    lowest key, and so on.  This implies that you must be able to

//    compare two keys with the < operator.

//

//    Each LookupTable has an embedded iterator class that allows users

//    of the class to traverse trhough the list and  have acess to each

//    node.

//    In this version of the LookupTable a new struct type called Pair

//    is introduced which represents a key/data pair.

typedef *Point* *Type*;

struct *Pair*

{

    int key;

*Type* datum;

    // This ctor is writtent for convenience in creating objects of Pair and copy

    Pair(int *keyA*, *Type* *datumA*) : key(*keyA*), datum(*datumA*) {}

};

struct *LT\_Node*

{

*Pair* pairM;

*LT\_Node* \*nextM;

    // This ctor should be convenient in insert and copy operations.

    LT\_Node(const *Pair* &*pairA*, *LT\_Node* \**nextA*) : pairM(*pairA*), nextM(*nextA*) {}

    // PROMISES: initializes the data members pairM and nextM, with pairA and nextA

    // respectively

};

class *LookupTable*

{

public:

    // Nested class

    LookupTable() : sizeM(0), headM(nullptr), cursorM(nullptr) {}

    // PROMISES: An empty LookupTable object with all data members. Setting size, cursor and

    // head to zero or nullptr

    // copy ctor

    LookupTable(const *LookupTable* &*source*);

    // assignment operator

*LookupTable* &operator=(const *LookupTable* &*rhs*);

    // dtor

    ~LookupTable();

*LookupTable* &begin();

    // PROMISES: Moves cursorM to the beginning of the list

    int size() const;

    // PROMISES: Returns number of keys in the table.

    int cursor\_ok() const;

    // PROMISES:

    //   Returns 1 if the cursor is attached to a key/datum pair,

    //   and 0 if the cursor is in the off-list state.

    const int &cursor\_key() const;

    // REQUIRES: cursor\_ok()

    // PROMISES: Returns key of key/datum pair to which cursor is attached.

    const *Type* &cursor\_datum() const;

    // REQUIRES: cursor\_ok()

    // PROMISES: Returns datum of key/datum pair to which cursor is attached.

    void insert(const *Pair* &*pairA*);

    // PROMISES:

    //   If keyA matches a key in the table, the datum for that

    //   key is set equal to datumA.

    //   If keyA does not match an existing key, keyA and datumM are

    //   used to create a new key/datum pair in the table.

    //   In either case, the cursor goes to the off-list state.

    int remove(const int &*keyA*);

    // PROMISES:

    //   If keyA matches a key in the table, the corresponding

    //   key/datum pair is removed from the table.

    //   If keyA does not match an existing key, the table is unchanged.

    //   In either case, the cursor goes to the off-list state.

    void find(const int &*keyA*);

    // PROMISES:

    //   If keyA matches a key in the table, the cursor is attached

    //   to the corresponding key/datum pair.

    //   If keyA does not match an existing key, the cursor is put in

    //   the off-list state.

    void go\_to\_first();

    // PROMISES: If size() > 0, cursor is moved to the first key/datum pair

    //   in the table.

    void step\_fwd();

    // REQUIRES: cursor\_ok()

    // PROMISES:

    //   If cursor is at the last key/datum pair in the list, cursor

    //   goes to the off-list state.

    //   Otherwise the cursor moves forward from one pair to the next.

    void make\_empty();

    // PROMISES: size() == 0.

    void display() const;

    // PROMISES: displays the values o

    bool isEmpty() const;

    // PROMISES: returns true is list is empty

    int \*retrieve\_at(int *i*);

    // PROMISES: returns the adress of the key at the position index. Reminder: the index

    // number for the first node in the list is 0, 2nd node is 1, 3rd node is 2 and so on.

private:

    int sizeM;        // size of list (number of availble nodes)

*LT\_Node* \*headM;   // pointer to the first node in the list

*LT\_Node* \*cursorM; // pointer that can travers through the list

    void destroy();

    // Deallocate all nodes, set headM to zero.

    void copy(const *LookupTable* &*source*);

    // Establishes \*this as a copy of source.  Cursor of \*this will

    // point to the twin of whatever the source's cursor points to.

};

#endif

#### lookupTable.cpp

/\*\*

 \*  File Name: lookupTable.cpp

 \*  Assignment: ENSF 694 Summer 2024 - Lab 4 Exercise A Part 2

 \*  Created by: Mahmood Moussavi

 \*  Completed by: Yael Gonzalez

 \*  Submission Date: July 26, 2024

 \*/

#include "lookupTable.h"

*LookupTable*::LookupTable(const *LookupTable* &*source*)

{

    copy(*source*);

}

*LookupTable* &*LookupTable*::operator=(const *LookupTable* &*rhs*)

{

    if (*this* != &*rhs*)

    {

        destroy();

        copy(*rhs*);

    }

    return \**this*;

}

*LookupTable*::~LookupTable()

{

    destroy();

}

*LookupTable* &*LookupTable*::begin()

{

    cursorM = headM;

    return \**this*;

}

int *LookupTable*::size() const

{

    return sizeM;

}

int *LookupTable*::cursor\_ok() const

{

    return cursorM != nullptr;

}

const int &*LookupTable*::cursor\_key() const

{

    if (cursorM == nullptr)

    {

        cerr << "Cursor is pointing to null.";

    }

    return cursorM->pairM.key;

}

const *Type* &*LookupTable*::cursor\_datum() const

{

    if (cursorM == nullptr)

    {

        cerr << "Cursor is pointing to null.";

    }

    return cursorM->pairM.datum;

}

void *LookupTable*::insert(const *Pair* &*pairA*)

{

*LT\_Node* \*new\_node = **new** *LT\_Node*(*pairA*, nullptr);

    // Case 1: List is empty

    if (headM == nullptr)

    {

        headM = new\_node;

        sizeM++;

    }

    // Case 2: First node's key is greater than new insert

    else if (*pairA*.key < headM->pairM.key)

    {

        new\_node->nextM = headM;

        headM = new\_node;

        sizeM++;

    }

    // Case 3: First node's key is equal to new insert

    else if (*pairA*.key == headM->pairM.key)

    {

        headM->pairM.datum = *pairA*.datum; // update datum only

**delete** new\_node;                  // delete unnecessary node

    }

    // Case 4: New insert to be added across the list (after first node)

    else

    {

*LT\_Node* \*curr = headM;

        while (curr->nextM != nullptr && curr->nextM->pairM.key < *pairA*.key)

        {

            curr = curr->nextM;

        }

        if (curr->nextM != nullptr && curr->nextM->pairM.key == *pairA*.key)

        {

            curr->nextM->pairM.datum = *pairA*.datum; // update datum only

**delete** new\_node;                        // delete unnecessary node

        }

        else

        {

            new\_node->nextM = curr->nextM;

            curr->nextM = new\_node;

            sizeM++;

        }

    }

    cursorM = nullptr;

}

int *LookupTable*::remove(const int &*keyA*)

{

    // Case 1: List is empty

    if (headM == nullptr)

    {

        cerr << "List is empty." << endl;

        return 0;

    }

*LT\_Node* \*curr = headM;

*LT\_Node* \*prev = nullptr;

    int removed\_key;

    // Case 2: Node to be removed is the first in the list

    if (headM->pairM.key == *keyA*)

    {

        headM = headM->nextM;

        removed\_key = curr->pairM.key;

**delete** curr;

        sizeM--;

        cursorM = nullptr;

        return removed\_key;

    }

    // Case 3: Node to be removed is elsewhere in the list

    while (curr != nullptr && curr->pairM.key != *keyA*)

    {

        prev = curr;

        curr = curr->nextM;

    }

    if (curr == nullptr)

    {

        cerr << "Key not found." << endl;

        return 0;

    }

    prev->nextM = curr->nextM;

    removed\_key = curr->pairM.key;

**delete** curr;

    sizeM--;

    cursorM = nullptr;

    return removed\_key;

}

void *LookupTable*::find(const int &*keyA*)

{

    for (*LT\_Node* \*curr = headM; curr != nullptr; curr = curr->nextM)

    {

        if (curr->pairM.key == *keyA*)

        {

            cursorM = curr;

            return;

        }

    }

    cursorM = nullptr;

}

void *LookupTable*::go\_to\_first()

{

    if (sizeM > 0)

    {

        cursorM = headM;

    }

}

void *LookupTable*::step\_fwd()

{

    if (cursor\_ok())

    {

        cursorM = cursorM->nextM;

    }

}

void *LookupTable*::make\_empty()

{

    destroy();

    headM = nullptr;

    cursorM = nullptr;

    sizeM = 0;

}

void *LookupTable*::display() const

{

    if (headM == nullptr)

    {

        cerr << "List is empty.";

    }

    else

    {

        cout << " " << cursorM->pairM.key << " " << cursorM->pairM.datum.getx() << ", " << cursorM->pairM.datum.gety() << ", " << cursorM->pairM.datum.get\_label() << endl;

    }

}

bool *LookupTable*::isEmpty() const

{

    return headM == nullptr;

}

int \**LookupTable*::retrieve\_at(int *i*)

{

    if (*i* < 0 || *i* >= sizeM)

    {

        cerr << "Index should be positive and less than " << sizeM << endl;

        return nullptr;

    }

*LT\_Node* \*curr = headM;

    for (int j = 0; j < *i*; ++j)

    {

        curr = curr->nextM;

    }

    return &(curr->pairM.key);

}

void *LookupTable*::destroy()

{

    while (headM != nullptr)

    {

*LT\_Node* \*temp = headM;

        headM = headM->nextM;

**delete** temp;

    }

    cursorM = nullptr;

}

void *LookupTable*::copy(const *LookupTable* &*source*)

{

    if (*source*.headM == nullptr)

    {

        headM = nullptr;

        sizeM = 0;

        cursorM = nullptr;

        return;

    }

    headM = **new** *LT\_Node*(*source*.headM->pairM, nullptr);

*LT\_Node* \*srcNode = *source*.headM->nextM;

*LT\_Node* \*thisNode = headM;

    while (srcNode != nullptr)

    {

        thisNode->nextM = **new** *LT\_Node*(srcNode->pairM, nullptr);

        thisNode = thisNode->nextM;

        srcNode = srcNode->nextM;

    }

    sizeM = *source*.sizeM;

    if (*source*.cursorM != nullptr)

    {

*LT\_Node* \*srcCursor = *source*.headM;

*LT\_Node* \*newCursor = headM;

        while (srcCursor != *source*.cursorM)

        {

            srcCursor = srcCursor->nextM;

            newCursor = newCursor->nextM;

        }

        cursorM = newCursor;

    }

    else

    {

        cursorM = nullptr;

    }

}

### Program output

A screenshot of a computer

Description automatically generated

# Exercise B

## Source code

/\*\*

 \*  File Name: lab4exe\_B.cpp

 \*  Assignment: ENSF 694 Summer 2024 - Lab 4 Exercise B

 \*  Created by: Mahmood Moussavi

 \*  Completed by: Yael Gonzalez

 \*  Submission Date: July 26, 2024

 \*/

void print\_from\_binary(char \**filename*)

{

*ifstream* is(*filename*, *ios*::binary);

    if (is.fail())

    {

        cerr << "failed to open file: " << *filename* << endl;

        exit(1);

    }

*City* cities[size];

    for (int i = 0; i < size; i++)

        is.read((char \*)(&cities[i]), sizeof(*City*));

    for (int i = 0; i < size; i++)

        cout << "Name: " << cities[i].name << ", x coordinate: " << cities[i].x << ", y coordinate: " << cities[i].y << endl;

    is.close();

}

## Program output

A screen shot of a computer

Description automatically generated

# Exercise C

## AR Diagram

Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings


## Source code

/\*\*

 \*  File Name: lab4exe\_C.cpp

 \*  Assignment: ENSF 694 Summer 2024 - Lab 4 Exercise C

 \*  Created by: Mahmood Moussavi

 \*  Completed by: Yael Gonzalez

 \*  Submission Date: July 26, 2024

 \*/

#include <iostream>

#include <string.h>

using namespace std;

void insertion\_sort(int \**int\_array*, int *n*);

/\* REQUIRES

 \*    n > 0.

 \*    Array elements int\_array[0] ... int\_array[n - 1] exist.

 \* PROMISES

 \*    Element values are rearranged in non-decreasing order.

 \*/

void insertion\_sort(const char \*\**str\_array*, int *n*);

/\* REQUIRES

 \*   n > 0.

 \*   Array elements str\_array[0] ... str\_array[n - 1] exist.

 \* PROMISES

 \*   pointers in str\_array are rearranged so that strings:

 \*   str\_array[0] points to a string with the smallest string (lexicographicall) ,

 \*   str\_array[1] points to the second smallest string, ..., str\_array[n-2]

 \*   points to the second largest, and str\_array[n-1] points to the largest string

 \*/

int main(void)

{

    const char \*s[] = {"AB", "XY", "EZ"};

    const char \*\*z = s;

    z += 1;

    cout << "The value of \*\*z is: " << \*\*z << endl;

    cout << "The value of \*z is: " << \*z << endl;

    cout << "The value of \*\*(z-1) is: " << \*\*(z - 1) << endl;

    cout << "The value of \*(z-1) is: " << \*(z - 1) << endl;

    cout << "The value of z[1][1] is: " << z[1][1] << endl;

    cout << "The value of \*(\*(z+1)+1) is: " << \*(\*(z + 1) + 1) << endl;

    // point 1

    int a[] = {413, 282, 660, 171, 308, 537};

    int i;

    int n\_elements = sizeof(a) / sizeof(int);

    cout << "Here is your array of integers before sorting: \n";

    for (i = 0; i < n\_elements; i++)

        cout << a[i] << endl;

    cout << endl;

    insertion\_sort(a, n\_elements);

    cout << "Here is your array of ints after sorting:  \n";

    for (i = 0; i < n\_elements; i++)

        cout << a[i] << endl;

#if 1

    const char \*strings[] = {"Red", "Blue", "pink", "apple", "almond", "white",

                             "nut", "Law", "cup"};

    n\_elements = sizeof(strings) / sizeof(char \*);

    cout << "\nHere is your array of strings before sorting: \n";

    for (i = 0; i < n\_elements; i++)

        cout << strings[i] << endl;

    cout << endl;

    insertion\_sort(strings, 9);

    cout << "Here is your array of strings after sorting:  \n";

    for (i = 0; i < n\_elements; i++)

        cout << strings[i] << endl;

    cout << endl;

#endif

    return 0;

}

void insertion\_sort(int \**a*, int *n*)

{

    int i;

    int j;

    int value\_to\_insert;

    for (i = 1; i < *n*; i++)

    {

        value\_to\_insert = *a*[i];

        /\* Shift values greater than value\_to\_insert. \*/

        j = i;

        while (j > 0 && *a*[j - 1] > value\_to\_insert)

        {

*a*[j] = *a*[j - 1];

            j--;

        }

*a*[j] = value\_to\_insert;

    }

}

void insertion\_sort(const char \*\**str\_array*, int *n*)

{

    int i;

    int j;

    const char \*str\_to\_insert;

    for (i = 1; i < *n*; i++)

    {

        str\_to\_insert = \*(*str\_array* + i);

        /\* Shift strings lexicographically greater than str\_to\_insert. \*/

        j = i;

        while (j > 0 && strcmp(\*(*str\_array* + j - 1), str\_to\_insert) > 0)

        {

            \*(*str\_array* + j) = *str\_array*[j - 1];

            j--;

        }

        \*(*str\_array* + j) = str\_to\_insert;

    }

}

## Program output

A screenshot of a computer program

Description automatically generated

# Exercise D

## Source code

/\*\*

 \*  File Name: matrix.cpp

 \*  Assignment: ENSF 694 Summer 2024 - Lab 4 Exercise D

 \*  Created by: Mahmood Moussavi

 \*  Completed by: Yael Gonzalez

 \*  Submission Date: July 26, 2024

 \*/

#include "matrix.h"

*Matrix*::Matrix(int *r*, int *c*) : rowsM(*r*), colsM(*c*)

{

    matrixM = **new** double \*[rowsM];

    assert(matrixM != NULL);

    for (int i = 0; i < rowsM; i++)

    {

        matrixM[i] = **new** double[colsM];

        assert(matrixM[i] != NULL);

    }

    sum\_rowsM = **new** double[rowsM];

    assert(sum\_rowsM != NULL);

    sum\_colsM = **new** double[colsM];

    assert(sum\_colsM != NULL);

}

*Matrix*::~Matrix()

{

    destroy();

}

*Matrix*::Matrix(const *Matrix* &*source*)

{

    copy(*source*);

}

*Matrix* &*Matrix*::operator=(const *Matrix* &*rhs*)

{

    if (&*rhs* != *this*)

    {

        destroy();

        copy(*rhs*);

    }

    return \**this*;

}

double *Matrix*::get\_sum\_col(int *i*) const

{

    assert(*i* >= 0 && *i* < colsM);

    return sum\_colsM[*i*];

}

double *Matrix*::get\_sum\_row(int *i*) const

{

    assert(*i* >= 0 && *i* < rowsM);

    return sum\_rowsM[*i*];

}

void *Matrix*::sum\_of\_rows() const

{

    double sum;

    for (int i = 0; i < rowsM; i++)

    {

        sum = 0.0;

        for (int j = 0; j < colsM; j++)

        {

            sum += matrixM[i][j];

        }

        sum\_rowsM[i] = sum;

    }

}

void *Matrix*::sum\_of\_cols() const

{

    double sum;

    for (int j = 0; j < colsM; j++)

    {

        sum = 0.0;

        for (int i = 0; i < rowsM; i++)

        {

            sum += matrixM[i][j];

        }

        sum\_colsM[j] = sum;

    }

}

void *Matrix*::copy(const *Matrix* &*source*)

{

    if (*source*.matrixM == NULL)

    {

        matrixM = NULL;

        sum\_rowsM = NULL;

        sum\_colsM = NULL;

        rowsM = 0;

        colsM = 0;

        return;

    }

    rowsM = *source*.rowsM;

    colsM = *source*.colsM;

    sum\_rowsM = **new** double[rowsM];

    assert(sum\_rowsM != NULL);

    for (int i = 0; i < rowsM; i++)

        sum\_rowsM[i] = *source*.sum\_rowsM[i];

    sum\_colsM = **new** double[colsM];

    assert(sum\_colsM != NULL);

    for (int i = 0; i < colsM; i++)

        sum\_colsM[i] = *source*.sum\_colsM[i];

    matrixM = **new** double \*[rowsM];

    assert(matrixM != NULL);

    for (int i = 0; i < rowsM; i++)

    {

        matrixM[i] = **new** double[colsM];

        assert(matrixM[i] != NULL);

        for (int j = 0; j < colsM; j++)

        {

            matrixM[i][j] = *source*.matrixM[i][j];

        }

    }

}

void *Matrix*::destroy()

{

    if (matrixM != NULL)

    {

        for (int i = 0; i < rowsM; i++)

        {

**delete[]** matrixM[i];

        }

**delete[]** matrixM;

    }

**delete[]** sum\_rowsM;

**delete[]** sum\_colsM;

}

## Program output

A screenshot of a computer

Description automatically generated