**Course: Programming Fundamentals – ESNF 480**

Lab 1

M. Moussavi

Alessandra Schiavi Paz and Muhammed Umar Khan

Lab Section: B02

Sept 13, 2024

# Exercise A

|  |  |
| --- | --- |
| **Program output and its order** | **Your explanation (why and where is the cause for this output)** |
| **constructor with int argument is called.** | It is called at line 12 in exAmain. The statement, Mystring c = 3 is interpreted by the compiler as a call to the constructor Mystring::Mystring (int n). |
| **default constructor is called.**  **default constructor is called.** | It is called at line 18 in exAmain. The statement Mystring x[2] is an array with two Mystring objects that are not assigned values so it is interpreted by the compiler as a call to the default constructor twice. |
| **constructor with char\* argument is called.** | It is called at line 22 in exAmain. The statement new Mystring(“4”), creates a new object of type Mystring on the heap. The argument “4” is a string literal and can be passed to constructor as a char pointer. |
| **copy constructor is called.**  **copy constructor is called.** | It is called at line 24 in exAmain. The function append in mystring.cpp returns the value of the “this” pointer at line 107 which is a Mystring object so the copy constructor is called. Since the append function is called twice, the copy constructor is called twice. |
| **destructor is called.**  **destructor is called.** | It is called at line 102 in mystring using the statement delete []charsM, where charsM is a global variable, for the statement x[0].append(\*z).append(x[1]) at line 24 in exAmain. |
| **copy constructor is called.** | It is called at line 26 in exAmain. The statement Mystring mars = x[0] is initializing Mystring mars variable with an already existing Mystring object, x[0], so, it is interpreted by the compiler as a call to the copy constructor. |
| **assignment operator called.** | It is called on line 28 in exAmain. The statement x[1] = x[0] is interpreted by the compiler as a call to the assignment operator because x[1] and x[0] already exist. |
| **constructor with char\* argument is called.**  **constructor with char\* argument is called.** | It is called at line 30 and line 32 in exAmain. In both statements Mystring Jupiter(“ White”) and new Mystring(“Yellow”), the arguments, “White” and “Yellow” are string literals and they are passed to the constructor as char pointers to their location in static memory. |
| **destructor is called.**  **destructor is called.**  **destructor is called.**  **destructor is called.**  **destructor is called.** | It is called at line 34 in exAmain because Mystring x[2], Mystring\* z = new Mystring(“4”), Mystring mars = x[0], and Mystring Jupiter(“White”) have gone out of scope. |
| **constructor with char\* argument is called.** | It is called at line 39 in exAmain. The statement, Mystring d = “Green” is interpreted by the compiler as a call to the constructor Mystring::Mystring(const char \*s): lengthM((int)strlen(s)) |
| **Program terminated successfully.** | At line 41 in exAmain this statement is printed out. |
| **destructor is called.**  **destructor is called** | It is called at line 43 in exAmain for the statements Mystring d = “Green” on line 39 and Mystring c = 3 on line 12 because they go out of scope. |

# Exercise B

## Part I

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
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
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
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
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
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


## Part II

Source Code:

// lookuptable.cpp

// Lab 1 - Exercise A

// Completed by: Muhammed Umar Khan

#include <assert.h>

#include <iostream>

#include <stdlib.h>

#include "dictionaryList.h"

#include "mystring\_B.h"

using namespace std;

Node::Node(const Key& keyA, const Datum& datumA, Node \*nextA)

: keyM(keyA), datumM(datumA), nextM(nextA)

{

}

DictionaryList::DictionaryList()

: sizeM(0), headM(0), cursorM(0)

{

}

DictionaryList::DictionaryList(const DictionaryList& source)

{

copy(source);

}

DictionaryList& DictionaryList::operator =(const DictionaryList& rhs)

{

if (this != &rhs) {

destroy();

copy(rhs);

}

return \*this;

}

DictionaryList::~DictionaryList()

{

destroy();

}

int DictionaryList::size() const

{

return sizeM;

}

int DictionaryList::cursor\_ok() const

{

return cursorM != 0;

}

const Key& DictionaryList::cursor\_key() const

{

assert(cursor\_ok());

return cursorM->keyM;

}

const Datum& DictionaryList::cursor\_datum() const

{

assert(cursor\_ok());

return cursorM->datumM;

}

void DictionaryList::insert(const int& keyA, const Mystring& datumA)

{

// Add new node at head?

if (headM == 0 || keyA < headM->keyM) {

headM = new Node(keyA, datumA, headM);

sizeM++;

}

// Overwrite datum at head?

else if (keyA == headM->keyM)

headM->datumM = datumA;

// Have to search ...

else {

//POINT ONE

// if key is found in list, just overwrite data;

for (Node \*p = headM; p !=0; p = p->nextM)

{

if(keyA == p->keyM)

{

p->datumM = datumA;

return;

}

}

//OK, find place to insert new node ...

Node \*p = headM ->nextM;

Node \*prev = headM;

while(p !=0 && keyA >p->keyM)

{

prev = p;

p = p->nextM;

}

prev->nextM = new Node(keyA, datumA, p);

sizeM++;

}

cursorM = NULL;

}

void DictionaryList::remove(const int& keyA)

{

if (headM == 0 || keyA < headM -> keyM)

return;

Node \*doomed\_node = 0;

if (keyA == headM-> keyM) {

doomed\_node = headM;

headM = headM->nextM;

// POINT TWO

}

else {

Node \*before = headM;

Node \*maybe\_doomed = headM->nextM;

while(maybe\_doomed != 0 && keyA > maybe\_doomed-> keyM) {

before = maybe\_doomed;

maybe\_doomed = maybe\_doomed->nextM;

}

if (maybe\_doomed != 0 && maybe\_doomed->keyM == keyA) {

doomed\_node = maybe\_doomed;

before->nextM = maybe\_doomed->nextM;

}

}

if(doomed\_node == cursorM)

cursorM = 0;

delete doomed\_node; // Does nothing if doomed\_node == 0.

sizeM--;

}

void DictionaryList::go\_to\_first()

{

cursorM = headM;

}

void DictionaryList::step\_fwd()

{

assert(cursor\_ok());

cursorM = cursorM->nextM;

}

void DictionaryList::make\_empty()

{

destroy();

sizeM = 0;

cursorM = 0;

}

// The following function are supposed to be completed by the stuents, as part

// of the exercise B part II. the given fucntion are in fact place-holders for

// find, destroy and copy, in order to allow successful linking when you're

// testing insert and remove. Replace them with the definitions that work.

void DictionaryList::find(const Key& keyA){

Node\* current = headM;

while (current != NULL) {

if (current->keyM == keyA) {

cursorM = current;

return;

}

current = current->nextM;

}

cursorM = NULL;

}

void DictionaryList::destroy() {

while (headM != nullptr) {

Node\* temp = headM;

headM = headM->nextM;

delete temp;

}

sizeM = 0;

headM = nullptr;

cursorM = nullptr;

}

void DictionaryList::copy(const DictionaryList& source) {

if (source.headM == nullptr) {

headM = nullptr;

cursorM = nullptr;

sizeM = 0;

return;

}

headM = new Node(source.headM->keyM, source.headM->datumM, nullptr);

Node\* thisPrev = headM;

Node\* sourceNode = source.headM->nextM;

sizeM = 1;

while (sourceNode != nullptr) {

Node\* newNode = new Node(sourceNode->keyM, sourceNode->datumM, nullptr);

thisPrev->nextM = newNode;

thisPrev = newNode;

sourceNode = sourceNode->nextM;

sizeM++;

}

if (source.cursorM == nullptr) {

cursorM = nullptr;

} else {

Node\* srcCursor = source.headM;

Node\* thisCursor = headM;

while (srcCursor != source.cursorM) {

srcCursor = srcCursor->nextM;

thisCursor = thisCursor->nextM;

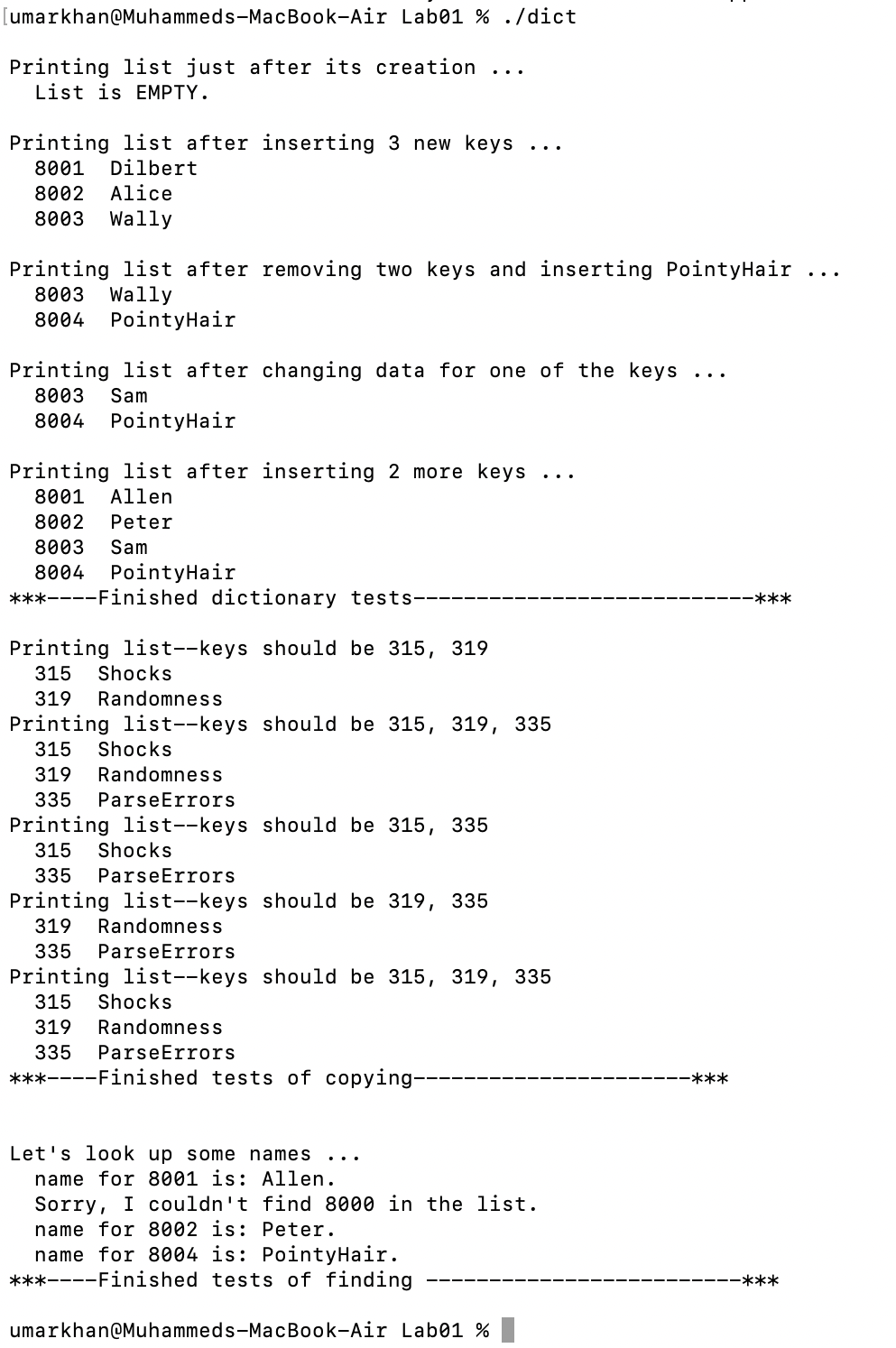
}

cursorM = thisCursor;

}

}

Output:



# Exercise C

Modified Code:

//compmay.cpp

#include <string>

#include <vector>

using namespace std;

class Company{

private:

Address companyAddress;

Name companyName;

vector <Employee> employeeList;

Date dateEstablished;

vector <Customer> customerList;

public:

//constructor & setters and getters

};

class Person {

protected:

Name name;

Address address;

public:

//setters and getters

};

class Employee : public Person {

private:

Date DOB;

string status;

public:

//setters and getters

};

class Customer : public Person {

private:

PhoneNumber phone;

public:

//setters and getters

};

class Name{

private:

string fName;

string lName = "";

public:

//setters and getters

};

class Address{

private:

string unit;

string street;

string country;

string postal;

public:

//setters and getters

};

class Date{

private:

int day;

int month;

int year;

public:

//setters and getters

};

class PhoneNumber{

private:

int areaCode;

int prefix;

int number;

};

# Exercise D

Point.h:

//Point.h

#ifndef POINT\_H

#define POINT\_H

class Point {

private:

double x; // x coordinate of a location on Cartesian Plain

double y; // y coordinate of a location on Cartesian Plain

public:

Point(double a = 0, double b = 0);

double get\_x() const;

double get\_y() const;

void set\_x(double a);

void set\_y(double a);

};

#endif

Point.cpp:

//Point.cpp

#include "Point.h"

Point::Point(double a, double b) : x(a), y(b) {}

double Point::get\_x() const { return x; }

double Point::get\_y() const { return y; }

void Point::set\_x(double a) { x = a; }

void Point::set\_y(double a) { y = a; }

Human.h:

//Human.h

#ifndef HUMAN\_H

#define HUMAN\_H

#include <string>

#include "Point.h"

class Human {

private:

Point location; // Location of an object of Human on a Cartesian Plain

std::string name; // Human's name

public:

Human(const std::string& nam = "", double x = 0, double y = 0);

// Copy constructor

Human(const Human& other);

// Assignment operator

Human& operator=(const Human& other);

// Destructor

~Human();

std::string get\_name() const;

void set\_name(const std::string& name);

Point get\_point() const;

void display() const;

};

#endif

Human.cpp:

//Human.cpp

#include "Human.h"

#include <iostream>

Human::Human(const std::string& nam, double x, double y)

: name(nam), location(x, y) {}

Human::Human(const Human& other)

: name(other.name), location(other.location) {}

Human& Human::operator=(const Human& other) {

if (this != &other) {

name = other.name;

location = other.location;

}

return \*this;

}

Human::~Human() {}

std::string Human::get\_name() const { return name; }

void Human::set\_name(const std::string& newName) { name = newName; }

Point Human::get\_point() const { return location; }

void Human::display() const {

std::cout << "Human Name: " << name << "\nHuman Location: "

<< location.get\_x() << " , "

<< location.get\_y() << ".\n" << std::endl;

}

main.cpp:

//main.cpp

#include "Human.h"

int main() {

double x = 2000, y = 3000;

Human h("Ken Lai", x, y);

h.display();

return 0;

}