Course: Principles of Software Design - ENSF 480

Lab #: 1

Instructor Name: Mahmood Moussavi

Student Name: Samiul Haque, Elias Poitras-Whitecalf

Lab Section: B02

Date Submitted: Sept 13, 2024

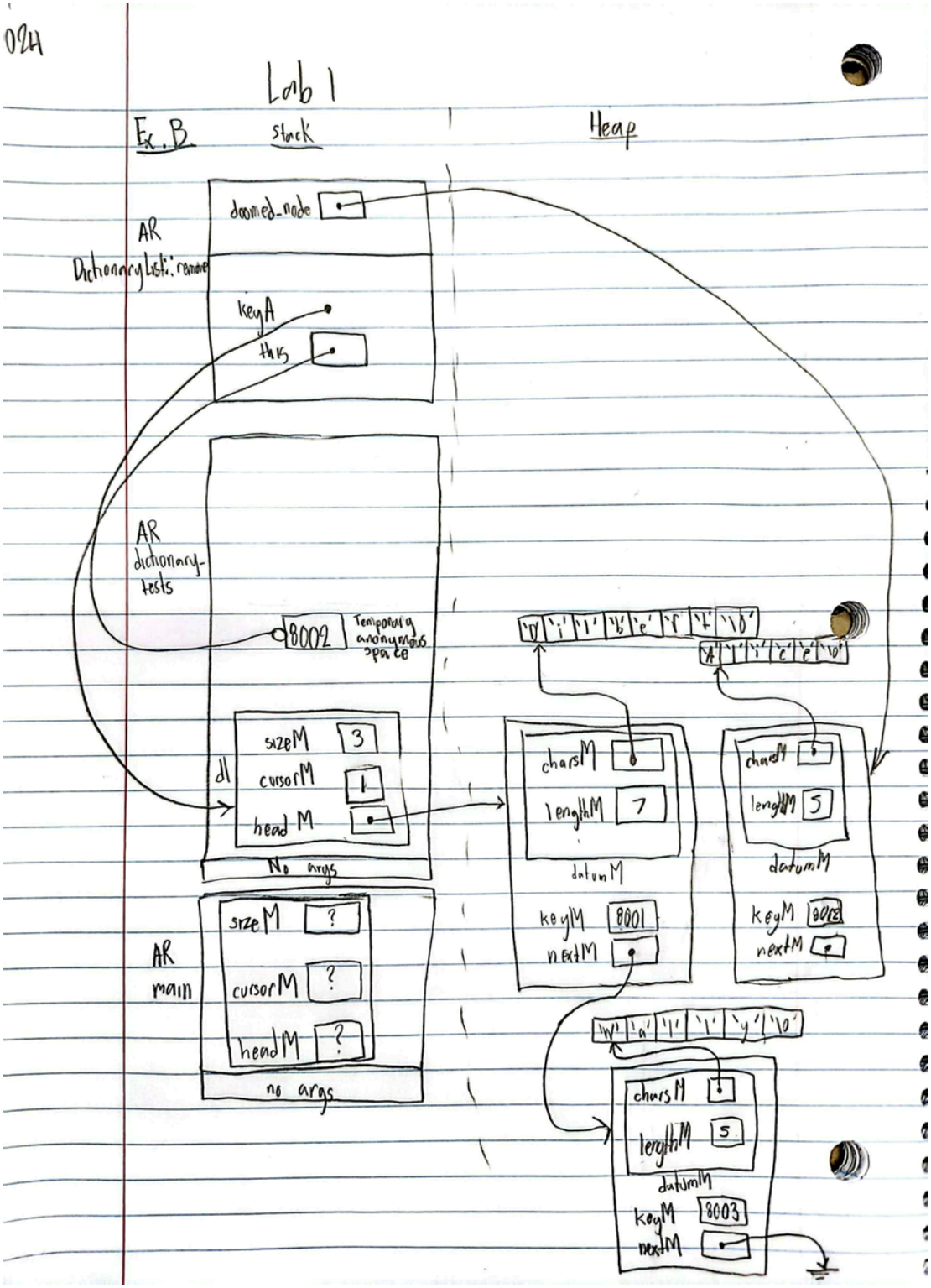
Exercise A:

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.**	Both are called at line 18 in exAmain. The statement, Mystring x[2] is interpreted by the compiler as two calls to the default constructor Mystring::Mystring() because two objects are being created.
constructor with char* argument is called.	It is called at line 22 in exAmain. The statement, Mystring* z is interpreted by the compiler as a call to the constructor Mystring::Mystring(const char *s).
copy constructor is called. **copy constructor is called.**	Both are called at line 24 in exAmain. The statement, x[0].append(*z).append(x[1]) is interpreted by the compiler as two calls to the copy constructor Mystring::Mystring(const Mystring& source) because in order to append z and x[1] respectively, a copy of them must be made.
destructor is called. **destructor is called.**	Both are called after line 24 in exAmain. This is because the *z and the x[1] copies are now out of the scope. The variables leaving the scope is interpreted as two called to the destructor.
copy constructor is called.	It is called at line 26 in exAmain. The statement, Mystring mars = x[0] is interpreted by the compiler as a call to the copy constructor Mystring::Mystring(const Mystring& source) because mars is a new object being assignend the value of a preexisting object.
assignment operator called.	It is called at line 28 in exAmain. The statement, x[1] = x[0] is interpreted by the compiler as a call to the assignment operator Mystring& Mystring::operator

	=(const Mystring& S) because both x[0] and x[1] are pre-existing.
constructor with char* argument is called. **constructor with char* argument is called.**	They are called at line 30 and 32 respectively. The statements Mystring.jupiter("White") and ar[0] = new Mystring("Yellow") are both interpreted by the compiler as calls to the constructor Mystring::Mystring(const char *s).
destructor is called. **destructor is called.** **destructor is called.** **destructor is called.** **destructor is called.**	The destructor, Mystring::~Mystring(), is called 5 times at **line 34**. Four of the five calls to the destructor occur in the cleanup process of the following Mystring objects that leave the scope: x[0], x[1], mars, Jupiter. The fifth call to the destructor is a result of **line 37** delete a[0] which is interpreted by the compiler as a call to the destructor.
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 with char* argument Mystring::Mystring(const char *s)
Program terminated successfully.	This is called on line 41 because of the cout statement.
destructor is called. **destructor is called**	The destructor, Mystring::~Mystring(), is called twice at line 43. They both occur in the cleanup process of the Mystring objects c and d, wherein they leave the scope.

Exercise B:

Part 1:



Part 2:

```
/*
* File Name: dictionaryList.cpp
* Assignment: Lab 1 Exercise B
* Lab Section: B02
* Completed by: Samiul Haque, Elias Poitras-Whitecalf
* Development Date: Sept 11, 2024
*/


#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)

{

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

      cursorM = 0;

   };



    Node *found_node = 0;



    if (keyA == headM-> keyM) {

        found_node = headM;

        headM = headM->nextM;
```

```
        cursorM = found_node;

    }

    else {

        Node *before = headM;

        Node *maybe_found = headM->nextM;

        while(maybe_found != 0 && keyA > maybe_found-> keyM) {

            before = maybe_found;

            maybe_found = maybe_found->nextM;

        }



        if (maybe_found != 0 && maybe_found->keyM == keyA) {

            cursorM = maybe_found;

            before->nextM = maybe_found->nextM;



        }



    }



}
```

```
void DictionaryList::destroy()

{

  while (headM != nullptr) {

    Node* current = headM;

    headM = headM->nextM;

    delete current;

  }

  cout << "\ndestructor is called. \n ";

}




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

  if (this == &source) {

        return;

    }

    sizeM = source.sizeM;

    cursorM = source.cursorM;



    if (source.headM != nullptr) {

        headM = new Node(*source.headM);

        Node* current = headM;

        Node* sourceCurrent = source.headM->nextM;

        while (sourceCurrent != nullptr) {
```

```
            current->nextM = new Node(*sourceCurrent);

            current = current->nextM;

            sourceCurrent = sourceCurrent->nextM;

        }

    } else {

        headM = nullptr;

    }



}
```

```
Printing list just after its creation ...
  List is EMPTY.

Printing list after inserting 3 new keys ...
  8001  Dilbert
  8002  Alice
  8003  Wally

Printing list after removing two keys and inserting PointyHair ...
  8003  Wally
  8004  PointyHair

Printing list after changing data for one of the keys ...
  8003  Sam
  8004  PointyHair

Printing list after inserting 2 more keys ...
  8001  Allen
  8002  Peter
  8003  Sam
  8004  PointyHair
***----Finished dictionary tests----------------------------***

Printing list--keys should be 315, 319
  315  Shocks
  319  Randomness
Printing list--keys should be 315, 319, 335
  315  Shocks
  319  Randomness
  335  ParseErrors

destructor is called.

destructor is called.
 Printing list--keys should be 315, 335
  315  Shocks
  335  ParseErrors
Printing list--keys should be 319, 335
  319  Randomness
  335  ParseErrors
Printing list--keys should be 315, 319, 335
  315  Shocks
  319  Randomness
  335  ParseErrors
***----Finished tests of copying----------------------***


destructor is called.

destructor is called.

destructor is called.

Let's look up some names ...
  name for 8001 is: Allen.
  Sorry, I couldn't find 8000 in the list.
  name for 8002 is: Peter.
  name for 8004 is: PointyHair.
***----Finished tests of finding --------------------------***
```

```
destructor is called.
```

Exercise C:

```
/*
* File Name: company.cpp
* Assignment: Lab 1 Exercise C
* Lab Section: B02
* Completed by: Samiul Haque, Elias Poitras-Whitecalf
* Development Date: Sept 12, 2024
*/
#include <string>
#include <vector>
using namespace std;

struct Company{
    private:
    string companyName;
    Address companyAddress;

    vector <Employee> employees;       //vector of information about
employee's information
                                      //(name, address, date of birth)
    Date dateEstablished;             //the data that company was
established

    vector <Customer> customers;      //vector of information about
customers
                                      //name, address, phone
};

class Date{
    private:
        int day;
        int month;
        int year;
};

class Name{
    private:
        string firstName;
        bool hasMidName;
```

```
        string midName;
        string lastName;
};

class Person{
    protected:
        Name name;
        Address address;
        Date dob;
};

class Customer : public Person{
    string phoneNumber;
};

class Employee : public Person{
    Status State;
};

class Status{
    public:
        enum State{active, suspended, retired, fired};

    private:
        State currentState;
};

class Address{
    private:
        int aptNum;
        string streetName;
        string postalCode;
        string city;
        string province;
        string country;
};
```

Exercise D:

human_program.cpp

```cpp
/*
* File Name: human_program.cpp
* Assignment: Lab 1 Exercise D
* Lab Section: B02
* Completed by: Samiul Haque, Elias Poitras-Whitecalf
* Development Date: Sept 12, 2024
*/
#include <cstring>
#include <iostream>
#include "human_program.h"
using namespace std;

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;};

Point::~Point(){}


Human::Human(const char* nam, double x, double y): name(new
char[strlen(nam)+1]){
    strcpy(this ->name, nam);
    location.set_x(x);
    location.set_y(y);
}

Human::Human() :name(nullptr) {
    location.set_x(0);
    location.set_y(0);
    name = new char[1];
    name[0] = '\0';
}

Human::~Human() {
```

```
    delete[] name;
}

char* Human::get_name() const {return name;}

void Human::set_name(const char* name) {
    delete[] this->name;
    this->name = new char[strlen(name)+1];
    strcpy(this ->name, name);
}

Point Human::get_point()const {return location;}

void Human::set_point(double x, double y) {
    location.set_x(x);
    location.set_y(y);
}


void Human::display() const {
    cout << "Human Name: " << name << "\nHuman Location: "
    << location.get_x() << " ,"
    << location.get_y() << ".\n" << endl;
}

#include <iostream>
#include "human_program.h"
using namespace std;

int main(int argc, char **argv)
{
    double x = 2000, y = 3000;
    Human h("Ken Lai", x , y);
    h.display();
    return 0;
}
//to run type:
//g++ -Wall -o myprog human_program.cpp
//./myprog
```

human_program.h

```
/*
* File Name: human_program.h
* Assignment: Lab 1 Exercise D
* Lab Section: B02
* Completed by: Samiul Haque, Elias Poitras-Whitecalf
* Development Date: Sept 12, 2024
*/
#ifndef POINT_HUMAN_H
#define POINT_HUMAN_H
#include <cstring>
#include <iostream>
using namespace std;


class Point{
    private:
        double x;     // x coordinate of a location on Cartisian Plain
        double y;     // y coordinate of a location on Cartisian Plain
    public:
        Point(double a = 0, double b = 0); //ctor
        void set_x(double a);    //setter
        void set_y(double a);    //setter
        double get_x() const;    //getter
        double get_y() const;    //getter
        ~Point();
};

class Human {
    protected:
        Point location;   // Location of an object of Human on a Cartisian
Plain
        char *name;       // Human's name
    public:
        Human();    //default ctor
        Human(const char* nam, double x, double y); //ctor with const
char* , double x, doule y args
        ~Human(); //destructor

        char* get_name() const;
```

```
        void set_name(const char* name);

        Point get_point() const;
        void set_point(double x, double y);

        void display() const;
};

#endif
```