#include <iostream>

#include <fstream>

using namespace std;

#define MAXLINE 100

int getLenChars(char str[])

{ // returns number of characters in a string

int i = 0;

int len = 0;

while (str[i] != '\0')

{

len++;

i++;

}

return len;

}

void toUpper(char input[])

{

int length = getLenChars(input);

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

{

if ((input[i] >= 'a') && (input[i] <= 'z'))

{// if the character is lowercase

input[i] -= 32; // maker uppercase

}

}

}

bool isLetter(char c)

{

return((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z'));

}

void countChars(char inputLine[], int len, int counts[], int &totalLetters)

{

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

{

if (isLetter(inputLine[i]) )

{

totalLetters++;

counts[int(inputLine[i]) - 65]++;

}

}

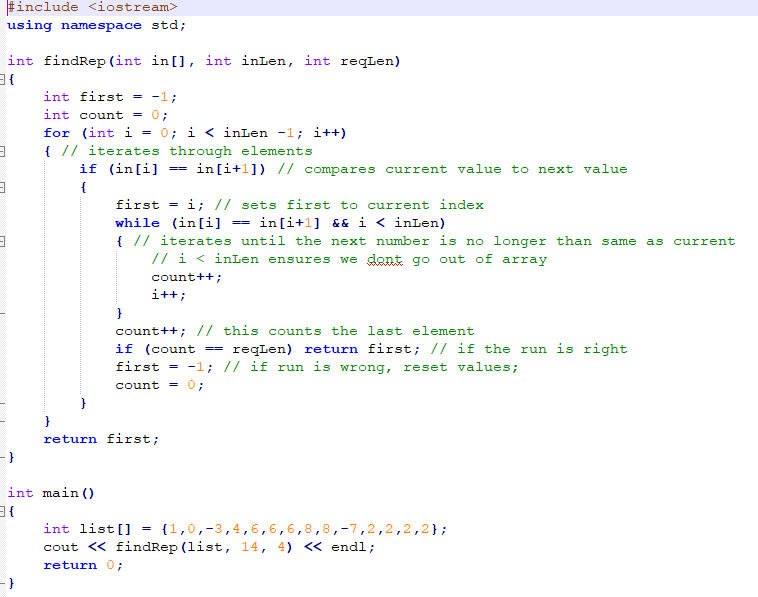
}

void printCounts(int counts[])

{

for (int i = 0; i < 26; i++)

{

 cout << char(i+65) << " : " << counts[i] << endl;

}

}

void findFrequencies(int counts[], int totalLetters, float frequencies[])

{

for (int i = 0; i < 26; i++)

{

if (counts[i] > 0)

{

frequencies[i] = (0.0+counts[i])/(0.0+totalLetters);

}

}

}

void printFrequencies(float frequencies[])

{

for (int i = 0; i < 26; i++)

{

cout << char(i+65) << " : " << frequencies[i] << endl;

}

}

void printAllData(int counts[],float frequencies[],int totalLetters)

{

cout << "Total letter count: " << totalLetters << endl;

cout << endl;

for (int i = 0; i < 26; i++)

{

cout << char(i+65) << ": count: " << counts[i] << " frequencies: " << frequencies[i] << endl;

}

}

int main()

{

ifstream file;

file.open("input.txt", ios\_base::in);

int counts[26] = {0}; // counts[0] is count of A; [1] for Z

int totalLetters = 0;

float frequencies[26] = {0.0};

char inputLine[MAXLINE];

while (file.getline(inputLine, MAXLINE))

{

toUpper(inputLine);

int len = getLenChars(inputLine);

countChars(inputLine, len, counts, totalLetters);

}

findFrequencies(counts, totalLetters, frequencies);

printAllData(counts, frequencies, totalLetters);

file.close();

return 0;

}

#include <iostream>

cout << "--------------------------" << endl;

cout << " Ordering System " << endl;

cout << "--------------------------" << endl;

cout << "Options: " << endl;

cout << "\t'N' - Create new order" << endl;

cout << "\t'S' - Show order queue" << endl;

cout << "\t'F' - Complete first order" << endl;

cout << "\t'P' - Show phone orders" << endl;

cout << "\t'D' - Show delivery queue" << endl;

cout << "\t'U' - Complete first delivery" << endl;

cout << "\t'C' - Clear order queue" << endl;

cout << "\t'Q' - Quit" << endl;

bool charInArray(char c, const char s[])

{ // assumes string is null terminated

int i = 0;

while (s[i] != '\0')

{

if (c == s[i++]) return true;

}

return false;

}

char prompt(const char msg[], const char validChoices[])

{

char input;

if (validChoices[0] == '\0') return '\0';

cout << msg << " ";

while (true)

{

cout << "[";

int i = 0;

while (validChoices[i] != '\0')

{

cout << upcase(validChoices[i]);

if (validChoices[i+1] != '\0') cout << "/";

i++;

}

cout << "] ";

cin >> input;

if (charInArray(upcase(input), validChoices)) return upcase(input);

}

}

#include <iostream>

#include <fstream>

using namespace std;

char endPunctuation[] = ".?!";

bool validChar(char c)

{

if ( (c >= 'A') && (c <= 'Z')) return true;

if ( (c >= 'a') && (c <= 'z')) return true;

if (c == '&' || c == '-' || c == '\'') return true;

return false;

}

void revSingleWord(char s[], int f, int l)

{

for (int i = 0; i < (l-f+1)/2; i++)

{

char hold = s[f+i];

s[f+i] = s[l-i];

s[l-i] = hold;

}

}

void revSingleWordRecursive(char s[], int f, int l)

{

if (f >= l) return;

char hold = s[f];

s[f] = s[l];

s[l] = hold;

revSingleWordRecursive(s, f+1, l-1);

}

void revWords(char s[])

{

int i = 0;

while (s[i] != '\0')

{

int j = i;

while (!validChar(s[j]) && s[j] != '\0')

{

j++;

}

// j is now the beginning of a word

i = j;

while (validChar(s[j]))

{

j++;

}

revSingleWord(s, i, j-1);

i = j;

}

}

int main()

{

char s[] = "this is a string of char's and now it should! be reversed";

revWords(s);

cout << s << endl;

return 0;

}

using namespace std;

#define MAXSUIT 9

struct card\_t{

char suit[MAXSUIT];

int value; // 1 is ace, 11 jack, 12 queen, 13 king

};

bool isFace(card\_t card)

{

return (card.value > 10 || card.value ==1);

}

void sort(int list[], int n)

{

int tmp;

for (int i = 0; i < n-1; i++)

**Prompting**

{

if (list[i] > list[i+1])

{

tmp = list[i];

list[i+1] = list[i];

list[i] = tmp;

}

}

}

bool isSequence(card\_t cards[], int n)

{

int values[n];

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

{

values[i] = cards[i].value;

}

sort(values, n);

for (int i = 0; i < n-1; i++)

{

if (values[i] == 1) // if king

{

if ( values[i+1] != 13 ) return false;

}

else if (values[i+1] != values[i]+1) return false;

}

return true;

}

int getLenChars(char str[])

{ // returns number of characters in a string

int i = 0;

int len = 0;

When it comes to string comparisons,

a < z.

A < a.

while (str[i] != '\0')

{

len++;

i++;

}

return len;

}

bool sameString(char first[], char second[])

{ // return true iff two strings are identical, character for character

int firstLen = getLenChars(first);

int secondLen = getLenChars(second);

int main()

{

ifstream file;

file.open("input.txt", ios\_base::in);

int counts[26] = {0}; // counts[0] is count of A; [1] for Z

int totalLetters = 0;

float frequencies[26] = {0.0};

char inputLine[MAXLINE];

while (file.getline(inputLine, MAXLINE))

{

toUpper(inputLine);

int len = getLenChars(inputLine);

countChars(inputLine, len, counts, totalLetters);

}

findFrequencies(counts, totalLetters, frequencies);

printAllData(counts, frequencies, totalLetters);

file.close();

return 0;

}

if (firstLen != secondLen) return false;

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

{

if (first[i] != second[i]) return false;

#include <iostream>

using namespace std;

void mystery(int l[], int n)

{

int tmp = l[0];

for (int i = 0; i < n -1; i++)

{

l[i] = l[i+1];

}

l[n-1] = tmp;

}

int main()

{

int l[] = {100, 200, 300, 400, 500};

int n = 5;

mystery(l, n);

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

{

cout << l[i] << endl;

}

return 0;

}

}

return true;

}

bool sameSuit(card\_t cards[], int n)

{

char testSuit[MAXSUIT];

int i = 0;

int lenTestSuit = getLenChars(cards[0].suit);

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

{

testSuit[i] = cards[0].suit[i];

}

testSuit[lenTestSuit] = '\0';

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

{

if (!sameString(cards[i].suit, testSuit))

return false;

}

return true;

}

bool isFlush(card\_t cards[], int n)

{

return (sameSuit(cards, n) && !isSequence(cards, n));

}

int main()

{

card\_t cards[3] = { {"Club", 1}, {"Club", 13}, {"Heart", 5}};

cout << isFlush(cards, 3) << endl;

return 0;

}

**STACKS**

bool string\_eq(char first[], char second[])

{ // return true iff two strings are identical, character for character

int firstLen = getLenChars(first);

int secondLen = getLenChars(second);

if (firstLen != secondLen) return false;

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

{

if (first[i] != second[i]) return false;

}

return true;

}

void toUpper(char input[])

{

int length = getLenChars(input);

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

{

if ((input[i] >= 'a') && (input[i] <= 'z'))

{// if the character is lowercase

input[i] -= 32; // maker uppercase

}

}

}

bool string\_eq\_nocase(char first[], char second[])

{ // a case insensitive check for string equality

toUpper(first);

toUpper(second);

if ( string\_eq(first, second) ) return true;

return false;

}

void strip\_dup\_spaces(char str[])

{ // removes all consecutive spaces from str

int len = usedLen(str);

for (int i = 0; i < len-1; i++)

{

while((str[i] == ' ') && (str[i+1] == ' ') )

{

for (int j = i; j < len-1; j++)

{

str[j] = str[j+1];

}

}

}

}

bool contains\_sub\_str(char haystack[], char needle[])

{ // return true iff needle appears within haystack

int len2 = getLenChars(needle);

int len1 = usedLen(haystack);

char test[len2];

test[len2] = '\0';

for (int i = 0; i < len1-len2; i++)

{

if (haystack[i] == needle[0])

{

for(int j = 0; j < len2; j++)

{

test[j] = haystack[i+j];

}

if (string\_eq(test, needle)) return true;

}

}

return false;

}

int index\_sub\_str(char haystack[], char needle[])

{ // return true iff needle appears within haystack

int len2 = getLenChars(needle);

int len1 = usedLen(haystack);

char test[len2];

test[len2] = '\0';

for (int i = 0; i < len1-len2; i++)

{

if (haystack[i] == needle[0])

{

for(int j = 0; j < len2; j++)

{

test[j] = haystack[i+j];

}

if (string\_eq(test, needle)) return i;

}

}

return -1;

}

bool del\_first\_occur(char input[], char cut[])

{ // if cut appears in input, remove it and return true;

// otherwise leave input as is and return false

if (!contains\_sub\_str(input, cut)) return false;

int lenIn = usedLen(input);

int lenCut = getLenChars(cut);

int indexCut = index\_sub\_str(input, cut);

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

{

for (int j = indexCut; j < lenIn; j++)

{

input[j] = input[j+1];

}

}

return true;

}

#include "stack.h" // We implement this functionality

#include <string> /\* Using the C++ standard string type \*/

#include <assert.h>

using namespace std;

struct stack\_item\_t {

string data;

stack\_item\_t \*next;

};

struct stack\_t {

stack\_item\_t \*head;

};

stack\_t \*create\_stack()

{

stack\_t \*stk = new stack\_t; // Make a new data structure to store the stack

assert(stk != NULL); // If this failed, we're in trouble

stk->head = NULL;

return stk;

}

void destroy\_stack(stack\_t \*stk)

{

assert(stk != NULL);

stack\_item\_t \*p = NULL;

while (!is\_empty\_stack(stk)) // Remove and free the items

pop\_off\_stack(stk);

delete stk; // Delete the stack itself

}

void push\_on\_stack(stack\_t \*stk, string s)

{

assert(stk != NULL);

stack\_item\_t \*p = new stack\_item\_t; // Create a stack item

assert(p != NULL); // If this failed, we're in trouble

p->data = s; // String will copy all the necessary bits

p->next = stk->head;

stk->head = p;

}

bool is\_empty\_stack(stack\_t \*stk)

{

assert(stk != NULL);

return (stk->head == NULL);

}

string pop\_off\_stack(stack\_t \*stk)

{

assert(stk != NULL);

assert(stk->head != NULL); // Stack must not be empty

stack\_item\_t \*p = stk->head;

stk->head = stk->head->next;

string ret = p->data;

delete p;

return ret;

}

string peek\_top\_of\_stack(stack\_t \*stk)

{

assert(stk != NULL);

assert(stk->head != NULL); // Stack must not be empty

return stk->head->data;

}

#include <iostream>

#include <string>

#include <assert.h>

#include "sets.h"

using namespace std;

struct str\_t {

string word;

str\_t \* next;

};

struct set\_of\_str\_t {

str\_t \* head;

};

struct iterator\_t {

set\_of\_str\_t \* set;

str\_t \* current;

};

iterator\_t \* create\_iterator(set\_of\_str\_t \* setToIterate)

{

iterator\_t \* iterator = new iterator\_t;

iterator->set = setToIterate;

iterator->current = setToIterate->head;

return iterator;

}

str\_t \* getFirst(iterator\_t \* it)

{

return (it->set->head);

}

str\_t \* getNext(iterator\_t \* it)

{

it->current = it->current->next;

return (it->current);

}

bool hasMore(iterator\_t \* it)

{

return (it->current != NULL);

}

string getCurrWord(iterator\_t \* it)

{

return it->current->word;

}

set\_of\_str\_t \* create\_set\_of\_str()

{

set\_of\_str\_t \* set = new set\_of\_str\_t;

assert(set!=NULL); // something went wrong

set->head = NULL;

return set;

}

void print\_list(set\_of\_str\_t \* list)

{

str\_t \* walker = list->head;

while (walker!=NULL)

{

string toPrint = walker->word;

cout << toPrint << endl;

walker = walker->next;

}

}

bool contains\_item(set\_of\_str\_t \* list, string element)

{

if (list == NULL) return false;

str\_t \* walker = list->head;

while (walker != NULL)

{

if (walker->word == element) return true;

walker = walker->next;

}

return false;

}

void add\_item(set\_of\_str\_t \* &list, string toAdd)

{

assert(list!=NULL);

if (!contains\_item(list, toAdd))

{

str\_t \* newElement = new str\_t;

newElement->word = toAdd;

newElement->next = list->head;

list->head = newElement;

}

}

#include <string> // sets.h

using namespace std;

struct str\_t;

struct set\_of\_str\_t;

struct iterator\_t;

set\_of\_str\_t \* create\_set\_of\_str();

set\_of\_str\_t \* union\_(set\_of\_str\_t\*, set\_of\_str\_t\*);

set\_of\_str\_t \* intersection(set\_of\_str\_t \*, set\_of\_str\_t \*);

void print\_list(set\_of\_str\_t\*);

bool is\_empty\_(set\_of\_str\_t\*);

bool contains\_item(set\_of\_str\_t \*, string);

void add\_item(set\_of\_str\_t \* &name, string);

void del\_item(set\_of\_str\_t \* &name, string);

void destroy\_set\_of\_str(set\_of\_str\_t\* &name);

iterator\_t \* create\_iterator(set\_of\_str\_t \* setToIterate);

string getCurrWord(iterator\_t \* it);

str\_t \* getFirst(iterator\_t \* it);

str\_t \* getNext(iterator\_t \* it);

bool hasMore(iterator\_t \* it);

bool string\_eq(char first[], char second[])

{ // return true iff two strings are identical, character for character

int firstLen = getLenChars(first);

int secondLen = getLenChars(second);

if (firstLen != secondLen) return false;

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

{

if (first[i] != second[i]) return false;

}

return true;

}

void toUpper(char input[])

{

int length = getLenChars(input);

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

{

if ((input[i] >= 'a') && (input[i] <= 'z'))

{// if the character is lowercase

input[i] -= 32; // maker uppercase

}

}

}

bool string\_eq\_nocase(char first[], char second[])

{ // a case insensitive check for string equality

toUpper(first);

toUpper(second);

if ( string\_eq(first, second) ) return true;

return false;

}

void strip\_dup\_spaces(char str[])

{ // removes all consecutive spaces from str

int len = usedLen(str);

for (int i = 0; i < len-1; i++)

{

while((str[i] == ' ') && (str[i+1] == ' ') )

{

for (int j = i; j < len-1; j++)

{

str[j] = str[j+1];

}

}

}

}

bool contains\_sub\_str(char haystack[], char needle[])

{ // return true iff needle appears within haystack

int len2 = getLenChars(needle);

int len1 = usedLen(haystack);

char test[len2];

test[len2] = '\0';

for (int i = 0; i < len1-len2; i++)

{

if (haystack[i] == needle[0])

{

for(int j = 0; j < len2; j++)

{

test[j] = haystack[i+j];

}

if (string\_eq(test, needle)) return true;

}

}

return false;

}

int index\_sub\_str(char haystack[], char needle[])

{ // return true iff needle appears within haystack

int len2 = getLenChars(needle);

int len1 = usedLen(haystack);

char test[len2];

test[len2] = '\0';

for (int i = 0; i < len1-len2; i++)

{

if (haystack[i] == needle[0])

{

for(int j = 0; j < len2; j++)

{

test[j] = haystack[i+j];

}

if (string\_eq(test, needle)) return i;

}

}

return -1;

}

bool del\_first\_occur(char input[], char cut[])

{ // if cut appears in input, remove it and return true;

// otherwise leave input as is and return false

if (!contains\_sub\_str(input, cut)) return false;

int lenIn = usedLen(input);

int lenCut = getLenChars(cut);

int indexCut = index\_sub\_str(input, cut);

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

{

for (int j = indexCut; j < lenIn; j++)

{

input[j] = input[j+1];

}

}

return true;

}

void del\_item(set\_of\_str\_t \* &list, string toDelete)

{

if (list == NULL) return;

if (!contains\_item(list, toDelete)) return;

str\_t \* walker = list->head;

str\_t \* prev = walker;

while(walker!= NULL)

{

if (walker->word == toDelete)

{ // if current element is the one to delete

if (walker == list->head)

{ // to delete first element

list->head = walker->next;

delete walker;

walker = list->head;

}

else if (walker->next == NULL)

{ // to delete last element

prev->next = NULL;

delete walker;

}

else

{ // delete any elements in between first and last

prev->next = walker->next;

delete walker;

walker = prev->next;

}

}

else

{ // if nothing is removed

prev = walker;

walker = walker-> next;

}

}

}

set\_of\_str\_t \* union\_(set\_of\_str\_t\* listA, set\_of\_str\_t\* listB)

{

set\_of\_str\_t \* both = create\_set\_of\_str();

str\_t \* walkerA = listA->head;

str\_t \* walkerB = listB->head;

if (is\_empty\_(listA) && is\_empty\_(listB)) return both;

while (walkerA!=NULL)

{

add\_item(both, walkerA->word);

walkerA = walkerA->next;

}

while (walkerB!=NULL)

{

add\_item(both, walkerB->word);

walkerA = walkerB->next;

}

return both;

}

set\_of\_str\_t \* intersection(set\_of\_str\_t \* listA, set\_of\_str\_t \*listB)

{

set\_of\_str\_t \* cross = create\_set\_of\_str();

str\_t \* walkerA = listA->head;

if (is\_empty\_(listA) || is\_empty\_(listB)) return cross;

while (walkerA != NULL)

{

if (contains\_item(listB, walkerA->word) && !contains\_item(cross, walkerA->word))

{

add\_item(cross, walkerA->word);

}

walkerA = walkerA->next;

}

return cross;

}

bool is\_empty\_(set\_of\_str\_t\* list)

{

return (list == NULL);

}

void destroy\_set\_of\_str(set\_of\_str\_t \* &list)

{

str\_t \* walker = list->head;

while (walker!=NULL)

{

str\_t \* hold = walker->next;

delete walker;

walker = hold;

}

list->head = NULL;

delete list;

}

#include <iostream> // vehicles.h

#include <string>

using namespace std;

#ifndef \_\_VEHICLE\_\_

#define \_\_VEHICLE\_\_

class Vehicle {

public:

string getReg();

float getFee();

void setReg(string reg);

string getType();

protected:

string licensePlate;

float regFee;

};

#endif

#include "vehicle.h" // vehicles.cpp

#include <iostream>

using namespace std;

string Vehicle::getReg()

{

return licensePlate;

}

float Vehicle::getFee()

{

return regFee;

}

void Vehicle::setReg(string reg)

{

licensePlate = reg;

}

string Vehicle::getType()

{

if (regFee == 50) return "trailer";

if (regFee == 200) return "car";

if (regFee == 100) return "electric car";

}

#include "vehicle.h" // trailer.h

#ifndef \_\_TRAILER\_\_

#define \_\_TRAILER\_\_

class Trailer: public Vehicle

{

public:

Trailer();

Trailer(string reg);

virtual ~Trailer();

private:

/\*

string licensePlate;

float regFee;

\*/

};

#endif

#include "trailer.h" // trailer.cpp

#include <iostream>

using namespace std;

Trailer::Trailer()

{

regFee = 50;

licensePlate = "";

}

Trailer::Trailer(string reg)

{

regFee = 50;

licensePlate = reg;

}

Trailer::~Trailer()

{

}

#include "vehicle.h" // car.h

#ifndef \_\_CAR\_\_

#define \_\_CAR\_\_

class Car : public Vehicle

{

public:

Car();

Car(string reg);

virtual ~Car();

private:

};

#endif

#include "car.h" // car.cpp

#include <iostream>

using namespace std;

Car::Car()

{

regFee = 200;

licensePlate = "";

}

Car::Car(string reg)

{

regFee = 200;

licensePlate = reg;

}

Car::~Car()

{

}

#ifndef \_\_TREE\_NODE\_\_

#define \_\_TREE\_NODE\_\_

class TreeNode;

class TreeNode {

public:

TreeNode(TreeNode \*l\_child, TreeNode\* r\_child, double v);

double get\_value();

TreeNode \* get\_left();

TreeNode \* get\_right();

protected:

double val; // value stored in this node

TreeNode \*left; // pointer to left child, NULL if none

TreeNode \*right; // pointer to right child, NULL if none

};

#endif

#include "car.h" // electric car. cpp

#include "electriccar.h"

#include <iostream>

using namespace std;

ElectricCar::ElectricCar()

{

licensePlate = "";

regFee = 100;

}

ElectricCar::ElectricCar(string reg)

{

regFee = 100;

licensePlate = reg;

}

ElectricCar::~ElectricCar()

{

//cout << "insert electric car decontructor" << endl;

}

#include "car.h" //electriccar.h

#ifndef \_\_ELECTRIC\_CAR\_\_

#define \_\_ELECTRIC\_CAR\_\_

class ElectricCar : public Car

{

public:

ElectricCar();

ElectricCar(string reg);

virtual ~ElectricCar();

private:

};

#endif

#include <iostream>

#include "tree.h"

TreeNode::TreeNode(TreeNode \*l\_child, TreeNode \* r\_child, double v)

{

left = l\_child;

right = r\_child;

val = v;

}

double TreeNode::get\_value()

{

return val;

}

TreeNode\* TreeNode::get\_left()

{

return left;

}

TreeNode \* TreeNode::get\_right()

{

return right;

}

#include <iostream>

using namespace std;

#include "vehicle.h"

#include "electriccar.h"

#include "car.h"

#include "trailer.h"

void printTaxDetails(Vehicle\* ride)

{

string type\_ = ride->getType();

string license = ride->getReg();

float fee = ride->getFee();

string toPrint = license+" Tax for "+type\_+"s is $";

cout <<toPrint << fee <<endl;

}

int main(int argc, char \*\*argv)

{

Trailer \*oneHorseSlant = new Trailer();

Car \*lincolnCont = new Car("TX567");

ElectricCar \*tesla = new ElectricCar("TX945");

oneHorseSlant->setReg("TX642");

printTaxDetails(oneHorseSlant);

printTaxDetails(lincolnCont);

printTaxDetails(tesla);

delete oneHorseSlant;

delete lincolnCont;

delete tesla;

}

#include <iostream>

#include "tree.h"

using namespace std;

// Naming convention is to describe the path of the tree

// H - head

// L - left

// R - right

double max(double a, double b, double c)

{

double m = 0;

if (a > m) m = a;

if (b > m) m = b;

if (c > m) m = c;

return m;

}#include <iostream>

#include "tree.h"

using namespace std;

// Naming convention is to describe the path of the tree

// H – head // L – left // R - right

double max(double a, double b, double c)

{

double m = 0;

if (a > m) m = a;

if (b > m) m = b;

if (c > m) m = c;

return m;

}

double tree\_max(TreeNode \* tree)

{

if (tree==NULL) return 0.0;

return max(tree->get\_value(), tree\_max(tree->get\_left()), tree\_max(tree->get\_right()));

}

int main()

{

TreeNode \* HLLR = new TreeNode(NULL,NULL, 5.92);

TreeNode \* HLR = new TreeNode(NULL,NULL, 4.5);

TreeNode \* HRRR = new TreeNode(NULL,NULL,0.5);

TreeNode \* HRRL = new TreeNode(NULL,NULL,13.5);

TreeNode \* HRR = new TreeNode(HRRL,HRRR, 7.6);

TreeNode \* HR = new TreeNode(NULL, HRR,89.0);

TreeNode \* HLL = new TreeNode(NULL,HLLR, 45.8);

TreeNode \* HL = new TreeNode(HLL,HLR, 23.1);

TreeNode \* H = new TreeNode(HL,HR,56.8);

TreeNode \* root = H;

cout << tree\_max(root) << endl;

}

#include <iostream>

using namespace std;

#ifndef \_\_POLY\_\_

#define \_\_POLY\_\_

struct term;

class Poly {

public:

Poly(); // constructor

virtual ~Poly(); // deconstructor

void add\_term(double coeff, int degree);

double eval(double x);

private:

term \* head;

};

#endif

double Poly::eval(double x)

{

double sum = 0;

term \* walker = head;

while (walker != NULL)

{

sum += ((walker->coeff)\*pow(x,walker->degree));

walker = walker->next;

}

return sum;

}

#include <iostream>

using namespace std;

#include "poly.h"

double pow(double base, int exponent)

{

double output = 1;

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

{

output \*= base;

}

return output;

}

struct term {

double coeff;

int degree;

term \* next;

};

Poly::Poly()

{

head = NULL;

}

Poly::~Poly()

{

term \* walker = head;

while (walker != NULL)

{

term \* hold = walker->next;

delete walker;

walker = walker->next;

}

}

void Poly::add\_term(double c, int deg)

{

term \* toAdd = new term;

toAdd->coeff = c;

toAdd->degree = deg;

toAdd->next = NULL;

term \* walker = head;

if (head == NULL)

{

head = toAdd;

return;

}

while (walker->next != NULL)

{

walker = walker->next;

}

// at this point, walker is the last term

walker->next = toAdd;

}

void deleteLocations(location\_t \* head)

{

location\_t \* walker = head;

while (walker!=NULL)

{

location\_t \* hold = walker->next;

delete walker;

walker = hold;

}

}

bool hasLoop(item\_t \* head)

{

item\_t \* walker = head;

location\_t \* locations = new location\_t;

locations->item = NULL;

locations->next = NULL;

while (walker != NULL)

{

if (inList(locations, walker))

{

deleteLocations(locations);

return true;

}

addToList(walker, locations);

walker = walker->next;

}

deleteLocations(locations);

return false;

}

int main()

{

item\_t \* A = new item\_t;

item\_t \* B = new item\_t;

item\_t \* C = new item\_t;

item\_t \* D = new item\_t;

A->str = "A";

B->str = "B";

C->str = "C";

D->str = "D";

A->next = B; B->next = C; C->next = D; D->next = C;

cout << hasLoop(A) << endl;

delete A;

delete B;

delete C;

delete D;

return 0;

}

/\*

cannot create an array of visited values because we dont know size.

cannot go through list to find size because there might be a loop.

\*/

#include <iostream>

#include <string>

#include <assert.h>

using namespace std;

struct item\_t;

struct item\_t {

string str;

item\_t \* next;

};

struct location\_t;

struct location\_t {

item\_t \* item;

location\_t \* next;

};

bool inList(location\_t \* head, item\_t \* check)

{

location\_t \* walker = head;

while(walker != NULL)

{

if (walker->item == check) return true;

walker = walker->next;

}

return false;

}

void addToList(item\_t \* toAdd, location\_t \* &head)

{

if (head == NULL)

{

head = new location\_t;

head->item = toAdd;

head->next = NULL;

}

else

{

location\_t \* hold = new location\_t;

hold->item = toAdd;

hold->next = NULL;

location\_t \* walker = head;

while (walker->next != NULL)

{

walker=walker->next;

}

//at this point, walker is last element

walker->next = hold;

}

}