```cpp
#include <iostream>
using namespace std;

int getListOfCommon(int list1[], int p, int list2[], int q, int out[], int outcap)
{
    int i = 0; int j = 0; int k = 0;
    while (i < p && j < q)
    {
        if (k >= outcap) return k;
        if (list1[i] == list2[j])
        {
            if (k < outcap)
            {
                out[k] = list1[i];
                i++; j++; k++;
            }
            else return k;
        }
        if (list1[i] < list2[j]) i++;
        if (list2[j] < list1[i]) j++;
    }
    return k;
}

void printList(int list[], int len)
{
    for (int i = 0; i < len; i++)
    {
        cout << list[i] << endl;
    }
}

int main()
{
    int outcap = 4;
    int out[outcap];
    int list1[] = {1,5,7,9,12}; int p = 5;
    int list2[] = {1,3,4,5,6,7,8}; int q = 7;
    int size = getListOfCommon(list1, p, list2, q, out, outcap);
    printList(out, size);

    return 0;
}
```

```cpp
#include <iostream>
using namespace std;

const int n = 3;

int* all(int v[n][n]) // takes a square matrix of ints, returns a pointer to an int.
{ // Assumption: v contains non-negative integers
    int m = 0;
    int *ret = NULL; // creates a null pointer .

    // if nothing is done with the pointer, NULL is returned, which breaks the loop in line 23
    for (int i=0; i < n; i++) //iterates through every element
        for (int j=0; j < n; j++)
            if (v[i][j] > m) // if current element is greater than m, m is set to the current element
            {
                m = v[i][j];
                ret = &(v[i][j]); // the ret pointer is set to address of current element
            }
    return ret;
    // returns ret pointer, which is the location of the greatest value in the array
}

int main() {
    int t[n][n] = {{1, 23, 1}, {4, 0, 6}, {0, 12, 3}}; // Every element >= 0
    int *p = all(t); // initializes pointer to greatest element.
    while (p != NULL)
    {
        cout << *p << endl;
        *p = 0;
        p = all(t); // will print the elements in decreasing order.
    }
    return 0;
}
```

```cpp
#include <iostream>
using namespace std;

int findRep(int in[], int inLen, int reqLen)
{
    int first = -1;
    int count = 0;
    for (int i = 0; i < inLen -1; i++)
    { // iterates through elements
        if (in[i] == in[i+1]) // compares current value to next value
        {
            first = i; // sets first to current index
            while (in[i] == in[i+1] && i < inLen)
            { // iterates until the next number is no longer than same as current
                // i < inLen ensures we dont go out of array
                count++;
                i++;
            }
            count++; // this counts the last element
            if (count == reqLen) return first; // if the run is right
            first = -1; // if run is wrong, reset values;
            count = 0;
        }
    }
    return first;
}

int main()
{
    int list[] = {1,0,-3,4,6,6,6,8,8,-7,2,2,2,2};
    cout << findRep(list, 14, 4) << endl;
    return 0;
}
```

```
#include <iostream>
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++)
	{
		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;

	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);

   if (firstLen != secondLen) return false;
   for (int i = 0; i < firstLen; i++)
   {
	if (first[i] != second[i]) return false;
   }
   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;
}
```

```
#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;
}
```

```
#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;
}
```

```
	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);
	}

}
```

Prompting

When it comes to string comparisons,

a < z.

A < a.

```
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;
}
```

```
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;
}
```

STACKS

```
#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 <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);
```

```
#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;
    }
}
```

```
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;
}
```

```
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 <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()
{
}
```

```
#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 "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;
}
```

```
#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 <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"

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>
#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
```

```
#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;
}
```

```
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;
}
```

```
/*
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;
    }
}
```

```
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;
}
```