1)

```
#include <iostream>
using namespace std;
int main()  {
    double *dp[10]; //pointer to an array of 10
    double pointToMe = 0.0;
    for (int i =0; i<10 ; i++){
        dp[i] = &pointToMe; //allow dp in each index to point to memory
address 0.0;
    }
    //value display
    for (int i =0; i<10; i++){
        cout << *dp[i] << endl;
    }
    //address display
    for (int i =0; i<10; i++){
        cout << dp[i] << endl;
    }
    return 0;
}
```

2)

```
void myTest(int **dataStructure){
    int init =10;
    int rows =10;
    int columns = 10;
    //skipped step of pointing double pointer to a new pointer[] (rows)
    //then pointing to init (values)
    dataStructure = new int* [rows];

    for(int i =0; i < rows; i++){
        dataStructure[i] = new int[columns];
    }

    for (int i =0; i < rows; i++){
        for (int j =0; j<columns;j++){
            dataStructure[i][j] = init;
            init = init +10;
        }
    }
    for (int i = 0; i < 10; i++){
        cout <<endl;
        for (int j =0; j<columns;j++)  {
            cout << dataStructure[i][j] << " ";
        }
    }

}
```

3)
void f(int x){std::cout << ++x;}
Suppose you have
int j =5;
passing f(j) will create a copy for the function f to use without changing the value of j

void g(int &x){std::cout << ++x;}
This is using the address of variable j, which will allow the change of the variable j.
j will equal 6 with the same address

4.a - Why does the following code have memory leaks? (10 points)
The code attempts to delete a matrix that is not initialized. Array is in main which is passed to buildMatrix, buildMatrix creates a new matrix without connecting it to array in main. So when deleteMatrix is called, there is nothing to be deleted. Typically we use a copy constructor and override = in a class to avoid issues with pointer deletion. In this case, the memory leaks occurred because a new matrix separate from array was created and was not deleted.

4.b - How can we fix the problem? Find the bug that is causing the problem. (10 points)

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

int ** buildMatrix(int **matrix, int m, int n){
    int init = 10;
    matrix = new int*[m];
    for(int i = 0; i < m; i++)
        matrix[i] = new int[n]; //double pointer

    for (int i =0; i < m; i++){
        for (int j =0; j<n;j++){
            matrix[i][j] = init;
            init = init +10;
        }
    }
    return matrix;
}

void deleteMatrix(int **matrix, int m){

    for (int i=0;i<m;i++) {
            delete[] matrix[i];
            matrix[i] = nullptr;
        }
    delete [] matrix;

}

int main(){
    int **array;
    array = buildMatrix(array, 2, 2);
    deleteMatrix(array, 2);
    return 0;
}
```

I changed buildMatrix to return an int double pointer to be stored in array. Then when deleteMatrix(array,2) is called, there exists an array to be deleted.

5)
Append():

```
void LinkedList::append(const string& toInsert){
    //insert at tail of list
    //to test this function:
    //case: list is empty
    //case: list not empty, only one node/multiple nodes

    if (m_head != nullptr){

        Node* toAdd = m_head; //create a variable to point at node to find
our point of insertion
        //when toAdd points at NULL
        while(toAdd->m_next != NULL){ //find the null at end
            toAdd = toAdd->m_next; //update toAdd value with the next node
        }

        if (toAdd->m_next == NULL){
            Node* newNode = new Node; //create new Node with memory of type
node
            newNode->m_elem = toInsert; //update information into element
            toAdd->m_next = newNode; //toAdd points to newNode
            newNode->m_next = nullptr; //new Node points to null
        }
    }else {
        cout << "Inserting ..." <<endl; //otherwise call addFront to add a
node to empty list
        addFront(toInsert);
    }
}
```

RemoveTail():

```
void LinkedList::removeTail(){
    //to test this function:
    //case: list is empty> if(head == NULL)
    //case: only one node
    //case: list not empty, multiple nodes
    if (m_head != nullptr){ //list is not empty
        Node* temp; //store value of head in a temp variable
        Node* toDelete = m_head; //create a variable to point at node to
delete

        //toDelete next pointing cant be null
        while(toDelete->m_next != NULL){ //find the null at end
            temp = toDelete;
            toDelete = toDelete->m_next; //update temp value with the next
node
        }
        if (toDelete->m_next == NULL && temp != NULL){
            temp->m_next = nullptr;
            delete toDelete; // new node value in temp next
        }else {
            m_head = nullptr; //link head to null
            delete temp; //delete the list entirely
        }

    }else
        throw runtime_error("Error: Attempt removing from empty list!");
}
```

6) append(): test

   1- when there are multiple nodes Note: when there are only one node append functions the same

   3- when list is empty

7) removeTail(): test

1- When there are multiple nodes
2- When there is only one node
3- When list is empty

8)

Append test:
In main:

```
testAppend1();
testAppend2();
```

```
bool testAppend1(){
    bool returnVal = false;
    LinkedList aPath;
    string airports[] = {"BOS","ATL","MSP","LAX"};
    for (int i=0;i<4;i++)
        aPath.addFront(airports[i]);
    aPath.printList();
    aPath.append("JAC");
    aPath.printList();
    string airports1[] = {"JAC","BOS","ATL","MSP","LAX"};
    //start at the end bc its LAX first =
    for (int j=4;j>0;j--){
        if (aPath.front() == airports1[j]){
            returnVal = true;
            aPath.removeFront();
            cout << "testAppend1() PASSED!" << endl;
        }
        else {
            cout << "testAppend1() FAILED!" << endl;
            returnVal = false;
        }

    }
    return returnVal;

}

bool testAppend2(){
    bool returnVal = false;
    LinkedList aPath;
    string airports[] = {};

    aPath.printList();
    aPath.append("JAC");
    if (!aPath.empty()) {
        cout << "testAppend2() PASSED!" << endl;
        return true;
    }
    else
        cout << "testAppend2() FAILED!" << endl;
        return false;


}
```

removeTail tests:

In main:
```
testRemoveTail1();
testRemoveTail2();
try{
    testRemoveTail3();
}
catch(exception &e){
    cout << "testRemoveTail1() PASSED empty list exception thrown!" << endl;
    cout << e.what() << endl;
}
```

```
bool testRemoveTail1(){
    bool returnVal = false;
    LinkedList aPath;
    string airports[] = {"BOS","ATL","MSP","LAX"};
    for (int i=0;i<4;i++)
        aPath.addFront(airports[i]);
    //removing a tail node that's linked to other nodes
    for (int i = 0; i<3; i++){
        aPath.removeTail(); //removing a tail node that's linked to other
nodes
    }
    cout << "Removed all nodes except for one"<<endl;
    aPath.printList();
    if ("LAX" == aPath.front()){
        cout << "testRemoveTail1() PASSED!" << endl;
        return true;
    }
    else{
        cout << "testRemoveTail1() FAILED!" << endl;
        return false;
    }

}

bool testRemoveTail2(){
    bool returnVal = false;
    LinkedList aPath;
    string airports[] = {"BOS"};
    aPath.addFront(airports[0]);
    aPath.printList();
    aPath.removeTail(); //removing a tail node linked to null
    //aPath.printList();
    if (aPath.empty()){
        cout << "testRemoveTail2() PASSED!" << endl;
        return true;
    }
    else{
        cout << "testRemoveTail2() FAILED!" << endl;
        return false;
    }
}

bool testRemoveTail3(){
    bool returnVal = false;
    LinkedList aPath;
    string airports[] = {"BOS"};
    aPath.addFront(airports[0]);
    aPath.printList();
    aPath.removeTail(); //removing a tail node linked to null
    aPath.removeTail(); //list empty
}
```