Q1

a. Not valid. arr_1 is a const address of the first element of the array, so it should not be changed.
b. Valid. arr_2 is a pointer that can be moved by arithmetic operators.
c. Not valid. &arr_1 is the pointer to the whole array, i.e., the data type of the value is an array, while arr_2 is a pointer to an int value. (In other words, &arr_1 is a pointer to a pointer to an int, while arr_2 is a pointer to an int – type mismatch.) https://stackoverflow.com/questions/8412694/address-of-an-array
d. Not valid. Even though both arr_1 and arr_2 are address and the difference between them can be calculated, the result of arr_2 - arr_1 is an integer and it cannot be directly assigned to a pointer.
e. Valid. Same as part d, the result of subtraction is an integer and can be stored as an element in the array.

Q2

```
class Node {
public:
   int val;
   Node *next;
   Node() : val(0), next(nullptr) {}
   Node(int x) : val(x), next(nullptr) {}
   Node(int x, Node *next) : val(x), next(next) {}
};

int getLength(Node* head) //this function give out the length of the linked list
{
   int length = 0;
   Node* current = head;
   while (current != nullptr) //if the pointer is not null
   {
      length++;//count it
      current = current->next;//go to the next node
   }
   return length;
}

void shuffle(Node* head)
{
   int length = getLength(head);
   if (length <= 2) //if it is less than length of 2, return
   {
      return;
   }
```

```
    Node* nd = head;//split the array in half
    Node* prev1 = nullptr;
    for(int i = 0; i < (length+1)/2; i++)//go to the middle of the linked list
    {
        prev1 = nd;
        nd = nd->next;
    }
    prev1->next = nullptr;//split the array in half


    //reverse part
    Node* current = nd;
    Node* next = nullptr;
    Node* prev = nullptr;

    while (current != nullptr)//reverse the second half of the linked list
    {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }


    Node* temp = head;
    Node* head1 = head->next;
    Node* tail = prev;
    while (head1 != nullptr && tail != nullptr)//combine two linked list together
    {
        temp->next = tail;
        tail = tail->next;
        temp = temp->next;
        temp->next = head1;
        head1 = head1->next;
        temp = temp->next;
    }

    if (tail != nullptr)//if there is still remain of the second linked list
    {
        temp->next = tail;
    }

}
```

Q3

```
class CircularQueue
{
public:
	CircularQueue(int n);
	CircularQueue(const CircularQueue& c);
	~CircularQueue();
	void enQueue(float value);
	float deQueue();
	bool isFull();
	bool isEmpty();
	float getLast();
	float getFront();
private:
	float* arr;
	int size;
	int head_ind;
	int tail_ind;
	int count;
};

bool CircularQueue::isFull()
{
	return (count == size);
}

bool CircularQueue::isEmpty()
{
	return (count == 0);
}

float CircularQueue::getFront()
{
	return arr[head_ind];
}

float CircularQueue::getLast()
{
	return arr[tail_ind];
}

// Constructor
```

```
CircularQueue::CircularQueue(int n)
{
	arr = new float[n];
	size = n;
	tail_ind = -1;
	head_ind = 0;
	count = 0;
}

// Copy constructor
CircularQueue::CircularQueue(const CircularQueue& c)
{
	arr = new float[c.size]; // Allocate new array
	size = c.size;
	tail_ind = c.tail_ind;
	head_ind = c.head_ind;
	count = c.count;

	// Copy all array element values
	for (int i = 0; i < c.size; i++)
	{
		arr[i] = c.arr[i];
	}
}

// Destructor
CircularQueue::~CircularQueue()
{
	delete[] arr;
}

void CircularQueue::enQueue(float value)
{
	if (isFull())
	{
		throw logic_error("Cannot add to full queue");
	}
	else
	{
		++tail_ind;
		tail_ind = tail_ind % size;
		arr[tail_ind] = value;
		count++;
	}
```

```
}

float CircularQueue::deQueue()
{
	if (isEmpty()) // Prevent reading from empty queue
	{
		throw logic_error("Cannot read from empty queue");
		return 0;
	}
	else // Return the first element added
	{
		float result = arr[head_ind];
		head_ind = (head_ind + 1) % size;
		count--;
		return result;
	}
}
```

Q4

```
//finds factor to zero out vector element
double findFactor(double pivot_row, double under_row) {
	return  (-1.0 * (under_row)) / pivot_row;
}

//multiples a scalar with a vector
vector<double> scalarMultiply(double factor, vector<double> vec){
	for (int i = 0; i < vec.size(); i++) { // loops through vecotr changing each element
		vec[i] *= factor;
	}
	return vec;
}
//swaps two vectors in memory
void swap(vector<double>& actual_pivot_row, vector<double>& row_with_pivot) {
	vector<double> temp = actual_pivot_row; //take the original pivot row and store it
	actual_pivot_row = row_with_pivot; //swap it with the row that actually can be pivot row
	row_with_pivot = temp;
}
//adds 2 vectors
vector<double> add(vector<double> actual_pivot_row, vector<double> under_rows) {
	for (int i = 0; i < actual_pivot_row.size(); i++) { // loops through vector adding each
element  to eachother
		under_rows[i] += actual_pivot_row[i];
```

```
	}
	return under_rows;
}

bool isInvertible(vector< vector<double> >& mat) {

	int columns = mat.size();

	if (columns == 0) { return false; } //checking the 0 case

	//checking to make sure its a square matrix
	for (int i = 0; i < columns; i++) {
		if (mat[i].size() != columns) {
			return false;
		}
	}
	int rows = mat.size();
	//Pseudo Gaussian elimination function
	for (int pivot = 0; pivot < columns; pivot++) {

		for (int under_row = pivot; under_row < columns; under_row++) {
			if (mat[under_row][pivot] != 0) {  //if we found we with a nonzero pivot, we
swap it to the beginning
				swap(mat[pivot], mat[under_row]);

				for (int k = pivot+1; k < rows; k++) { //go through each row and
make it 0]
					if (mat[k][pivot] == 0) { continue; } //if row already has a
zero, nothing to do
					mat[pivot] = scalarMultiply(findFactor(mat[pivot][pivot],
mat[k][pivot]), mat[pivot]); //reference function descriptions
					mat[k] = add(mat[pivot], mat[k]); //If row did not have a
zero, we follow through the gaussian algorithm 4and make a zero
					}
				break;
			}
			else if (under_row == rows - 1) {//means its gone through every row and
couldning find a nonzero one to switch
				return false;
			}
		}
	}
	return true; //finally returns true after the elimination and the diagonal numbers are all
nonzero
```

```
}
```