1. Recall that in C++:

- A function allows us to return one piece of information.
- Pass-by-Reference parameter passing is one way to get multiple pieces of information from a function. In pass-by-reference parameter passing, the formal parameter in the function definition refers to the memory location of the actual parameter in the call.
- Pass-by-Value parameter passing is the other way to passing information into a function. In pass-by-value parameter passing, the formal parameter in the function definition is assigned a copy of the actual parameter's value.

For example, consider a program that allows the user to calculate the area (length * width) and perimeter (2*(length + width)) of a rectangle. If we have a `calculateAreaAndPerimeter` function with inputs of length and width and outputs of area and perimeter. The function call from main:

```
calculateAreaAndPerimeter(l, w, area, perimeter);
```

A C++ function definition would be:

```
void calculateAreaAndPerimeter(double length, double width, double & area,
                                                            double & perimeter) {
      area = length * width;
      perimeter = 2*(length + width);

} // end calculateAreaAndPerimeter
```

The older programming language C (early 1970's) only had pass-by-value, so the programmer had to explicitly pass the value of the memory address of a variable (called a *pointer)* to simulate pass-by-reference. The '&' symbol preceeding a variable is the "address-of-operator" and returns the memory address of the variable. To access what a pointer points at, the pointer is *dereferenced* by preceeding the pointer variable by an asterisk '*', called the *indirection operator*. To make it more confusing, the asterisk '*' is also used to declare a pointer variable. Consider a simple example of pointers:

```
int myInt = 25;
int * ptr;                // ptr is an integer pointer

ptr = &myInt;             // assigns ptr the address of myInt
cout << *ptr << endl;     // prints 25 since ptr is followed/dereferenced
*ptr = *ptr + 5;
cout << "myInt = " << myInt << endl;
```

a) What would be printed by the last "cout" statement?

2. Let's consider how the above `calculateAreaAndPerimeter` function could be implemented in C using pointers and pass-by-value. The function call from `main` needs to pass the address of area and the address of perimeter so they can be changed by the function:

```
      double l = 5, w = 10, area, perimeter;

      calculateAreaAndPerimeter(l, w, &area, &perimeter);
```

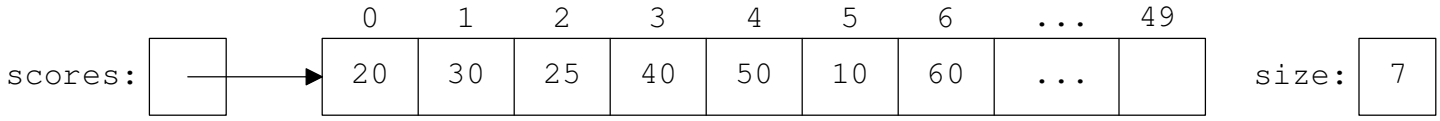
In the C-style function definition, the corresponding parameters are *double pointers* as in:

```
void calculateAreaAndPerimeter(double length, double width, double* ptrToArea,
                                                            double* ptrToPerimeter) {
      *ptrToArea = length * width;              // notice the indirection operator '*'
      *ptrToPerimeter = 2*(length + width);     // to allow us to "dereference the pointer"

} // end calculateAreaAndPerimeter
```

a) Write a `swap` function and a sample call using C pointers to exchange the value in two variables of type `int`.

3. Recall that an *array* is implemented as a contiguous block of memory. The array name is really a *constant pointer* to the first element of the array. (This means that the pointer cannot be changed.) For example,

```
const int MAX = 50;
double scores[MAX] = {20, 30, 25, 40, 50, 10, 60}; // Initial first 7 elements
const double payRates[] = {8.75, 15.75, 20.00, 30.00};  // array cannot change
int sizeOfScores = 7, sizeOfPayRates = 4;
```



So `scores` is of type "double * const scores" meaning that it is a pointer to a double which is constant, but `payRates` is of type "const double * const payRate" meaning that it is a pointer to a const double and the pointer cannot be changed.

If we wanted to write a `displayValues` function to print either of the above arrays, then the definition could be written as either:

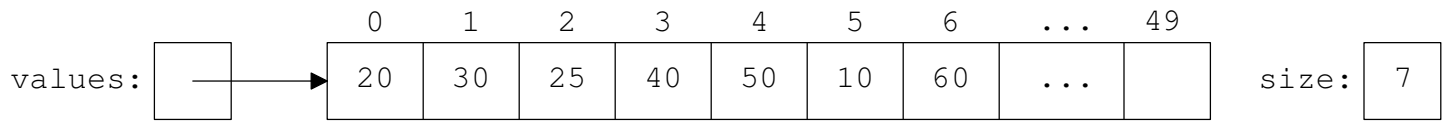
```
void displayValues (const double values[], int size) or
void displayValues (const double *values, int size)
```

NOTES:

- the "`const`" is needed for the first formal parameter to allow `payRates` to be passed to it
- the "`const`" tells the compiler and programmers that `displayValues` should NOT be changing the array so IT IS GOOD PROGRAMMING practice to use "`const`" where appropriate

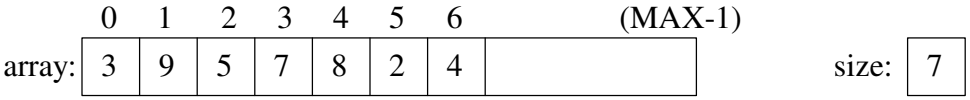
a) Write the displayValues code using the second approach.

4. Recall the "walking pointers" implementation of the `displayValues` function to print an array of constant doubles.




```
void displayValues (const double *values, int size) {
     const double * ptrToNext;  // pointer to walk down the array

     for (ptrToNext = values; ptrToNext <= &(values[size-1]); ptrToNext++) {
          cout << *ptrToNext << " ";
     } // end for
     cout << endl;
} // end displayValues
```

Modify the following bubble sort code to use pointers instead of indexing.



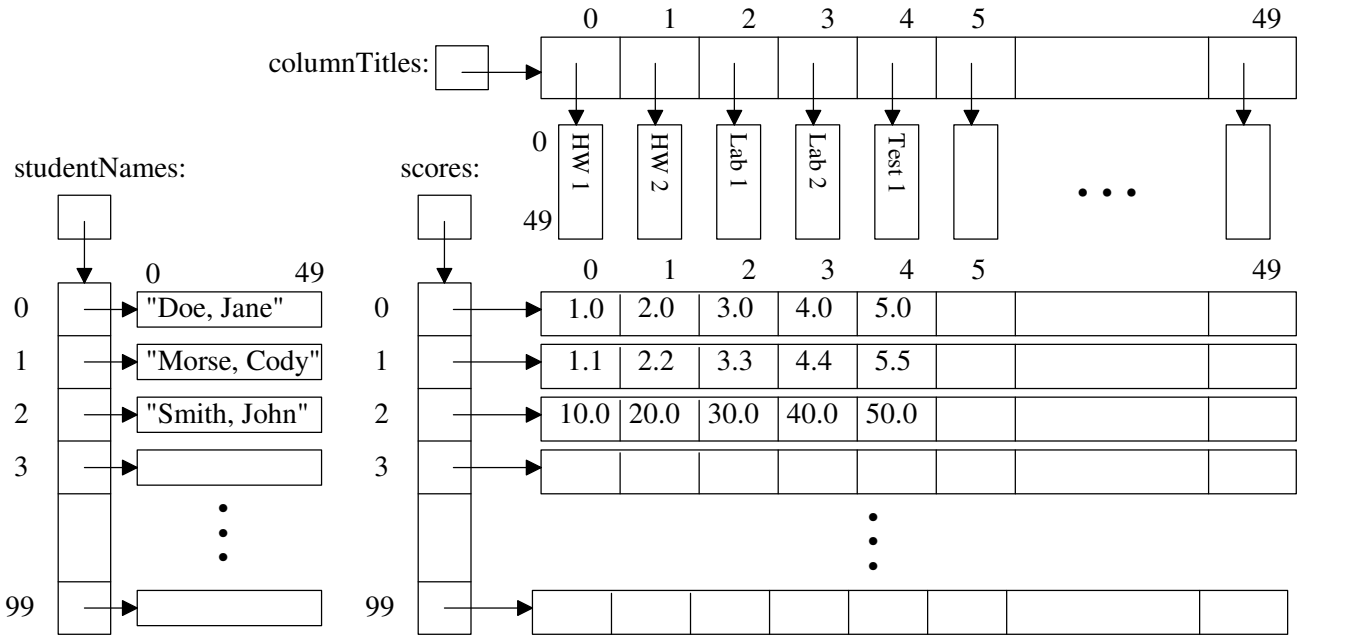

```
void bubbleSort(int array[], int size) {
     bool swap;

     int temp;

     int lastUnsorted;

     lastUnsorted = size-1;

     do   {
          swap = false;

          for (int count = 0; count < lastUnsorted; count++) {

               if (array[count] > array[count + 1]) {

                    temp = array[count];

                    array[count] = array[count + 1];

                    array[count + 1] = temp;

                    swap = true;

               } // end if

          } // end for

          lastUnsorted--;

     } while (swap);

} // end bubbleSort
```

5. When writing a program using an array, we might not know the array size at compile-time, so we CANNOT declare its size with a constant. Therefore, at run-time we want to *dynamically allocate memory* for the array from the "*heap*" using the "new" operator. The new operator returns a pointer to the first element in the dynamically allocated array. For example,

```
int * ptrToNewArray;          // does not point anywhere initially
int size, index;

cout << "Enter the array size:  ";
cin >> size;
ptrToNewArray = new int [size];     // dynamically allocate array from heap

for (index = 0; index < size; index++) {
     ptrToNewArray[index] = index;        // access array as normal
} // end for

for (index = 0; index < size; index++) {
     cout << ptrToNewArray[index] << " ";
} // end for
cout << endl;

delete [] ptrToNewArray;  // frees the storage for the array
```

In the grade book program, the declaration of arrays are:

```
// Global Constant
const int CLASS_SIZE = 100, MAX_SCORES = 50, NAME_SIZE = 50;

int main() {
      double scores[CLASS_SIZE][MAX_SCORES];
      char  columnTitles[MAX_SCORES][NAME_SIZE], studentNames[CLASS_SIZE][NAME_SIZE];
      int numberOfScores, numberOfStudents;
```



a) How might we save on storage?

b) When inserting a new student alphabetically, how might we save on time?