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:

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

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:

```cpp
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 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++ function definition, the corresponding parameters are `double pointers` would be:

```cpp
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”
}
```

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

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,

```c
const int MAX = 50;
double scores[MAX] = {20, 30, 25, 40, 50, 10, 60}; // Initial first 7 elements
cost 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:

```c
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.