Project 4, Program Design

1. (50 points) Pointers as function arguments

   Suppose you are given an array of integers and an integer value key. Write a program to calculate the indices of the two numbers in the array such that they add up to key. You can assume that there is exactly one solution for each input array and key.

   Example input/output #1:

```
Enter the length of the array: 7
Enter the elements of the array: 3 5 1 4 9 8 11
Enter the key value: 19
Output: 5 6
```

   Example input/output #2:

```
Enter the length of the array: 4
Enter the elements of the array: 2 5 8 4
Enter the key value: 10
Output: 0 2
```

   The program should include the following function. **Do not modify the function prototype.**

```
void find_indices(int a[], int n, int key, int * index1,
int *index2);
```

   `a` represents the input array with length `n`, and `key` represents the key value. The `index1 and index2` parameters point to variables in which the function will store the indices of the array elements that add up to key.

   1) Name your program `indices.c`

   2) In the main function, ask the user to enter the length of the input array, declare the input array. Then ask the user to enter the elements of the array.

   3) In the main function, call the find_indices function to find the indices of the two array elements that add up to key.

   4) The `main` function displays the two indices.

2. (50 points) Suppose you are given an array of n integers. Assume each integer in the array appear either one or twice in the array. Display all the integers that appear twice.

Example input/output #1:

```
Enter the length of the array: 10
Enter the elements of the array: 3 5 1 4 4 3 9 1 8 11
Output: 3 1 4
```

Example input/output #2:

```
Enter the length of the array: 8
Enter the elements of the array: 5 0 13 4 1 7 3 5
Output: 5
```

1) Name your program `arrays.c`
2) Include the following function to find elements that appear twice in the array:

```
void find_duplicates(int *a, int n, int *b, int *size);
```

`a` represents the input array with length `n`, and `b` represents the output array. The `size` parameter points to a variable in which the function will store the number of integers that appear twice (the actual length of array b).

This function should use pointer arithmetic– not subscripting – to visit array elements. In other words, eliminate the loop index variables and all use of the [] operator in the function.

3) In the main function, ask the user to enter the length of the input array, declare the input array. Then ask the user to enter the elements of the array, and declare the output array with half of the length of the input array.

4) In the main function, call the `find_duplicates` function to find elements that appear twice and store the elements in array b.

5) In the main function, display the output array.

6) Pointer arithmetic in NOT required in the main function.

Before you submit

1. Compile both programs with –Wall. –Wall shows the warnings by the compiler. Be sure it compiles on ***student cluster*** with no errors and no warnings.

   gcc –Wall indices.c
   gcc –Wall arrays.c

2. Be sure your Unix source file is read & write protected. Change Unix file permission on Unix:

   chmod 600 indices.c
   chmod 600 arrays.c

3. Test your programs with the shell scripts on Unix:

   chmod +x try_indices
   ./try_indices

   chmod +x try_arrays
   ./try_arrays

4. Submit *indices.c* and *arrays.c* on Canvas.

Grading

Total points: 100 (50 points each problem)

1. A program that does not compile will result in a zero.
2. Runtime error and compilation warning 5%
3. Commenting and style 15%
4. Functionality 80% (**Including functions implemented as required)**

Programming Style Guidelines

The major purpose of programming style guidelines is to make programs easy to read and understand. Good programming style helps make it possible for a person knowledgeable in the application area to quickly read a program and understand how it works.

1. Your program should begin with a comment that briefly summarizes what it does. This comment should also include your **name**.
2. In most cases, a function should have a brief comment above its definition describing what it does. Other than that, comments should be written only *needed* in order for a reader to understand what is happening.

3. Information to include in the comment for a function: name of the function, purpose of the function, meaning of each parameter, description of return value (if any), description of side effects (if any, such as modifying external variables)
4. Variable names and function names should be sufficiently descriptive that a knowledgeable reader can easily understand what the variable means and what the function does. If this is not possible, comments should be added to make the meaning clear.
5. Use consistent indentation to emphasize block structure.
6. Full line comments inside function bodies should conform to the indentation of the code where they appear.
7. Macro definitions (#define) should be used for defining symbolic names for numeric constants. For example: **`#define PI 3.141592`**
8. Use names of moderate length for variables. Most names should be between 2 and 12 letters long.
9. Use underscores to make compound names easier to read: **`tot_vol`** or **`total_volumn`** is clearer than totalvolumn.