Project 4, Program Design

1. Write a program `array_compare.c` that determines the number of elements are different between the two integer arrays (same size) when compared element by element (first element with first element, second with second, and so on). The program should include the following function. **Do not modify the function prototype.**

   ```
   void count_diff(int *a1, int *a2, int n, int *num_diff);
   ```

   The function takes two integer array parameter `a1` and `a2` of size `n`. The function stores the number of elements that are different to the variable pointed by the pointer variable `num_diff`.

   The 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.

   The `main` function should ask the user to enter the size of the arrays and then the elements of the arrays, and then declare the arrays. The `main` function calls the `count_diff` function. The `main` function displays the result.

   ```
   Example input/output #1:

   Enter the length of the array: 5

   Enter the elements of the first array:  -12 0 4 2 36

   Enter the elements of the second array: -12 0 4 2 36

   Output: The arrays are identical


   Example input/output #2:

   Enter the length of the array: 4

   Enter the elements of the first array:  -12 0 4 36

   Enter the elements of the second array: 12 0 41 36

   Output: The arrays are different by 2 elements
   ```

2. A simple way to encrypt a number is to replace each digit of the number with another digit. Write a C program that asks the user to enter a positive integer and replace each digit by *adding*

6 to the digit and calculate the remainder by 10. For example, if a digit is 6, then the digit is replaced by (6+6)%10, which is 2. After all the digits are replaced, the program then swap the first digit with the last digit. The main function reads in input and displays output. A sample input/output:

```
Enter the number of digits of the input number: 3

Enter the number: 728

Output: 483
```

1) Name your program **`encrypt_digits.c`**.
2) The user will enter the total number of digits before entering the number.
3) You can use format specifier "%1d" in scanf to read in a single digit into a variable (or an array element). For example, for input 101011, scanf("%1d", &num) will read in 1 to num.
4) As part of the solution, write and call the function `encrypt()` with the following prototype. The function assumes that the digits are stored in the array `a` and computes the encrypted digits and store them in the array `b`. c represents the size of the arrays.

   ```
   void encrypt(int *a, int *b, int n);
   ```

   The 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.

5) As part of the solution, write and call the function `swap()` with the following prototype.

   ```
   void swap(int *p, int *q);
   ```

   When passed the addresses of two variables, the `swap()` function should exchange the values of the variables:

   swap(&i, &j); /* exchange values of i and j */

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 array_compare.c

   gcc –Wall encrypt_digits.c

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

   chmod 600 array_compare.c

   chmod 600 encrypt_digits.c

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

   chmod +x try_compare

   ./try_compare

   chmod +x try_encrypt

   ./try_encrypt

4. Submit *array_compare.c* and *encrypt_digits.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%

   -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.