Lab 06 : Arrays

I. Overview and Objectives

Arrays are a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index. All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

The learning objective of this lab is to practice working with arrays of one and two dimensions and to use them with functions.

Reading and related topics: Course slides lesson 07. Book chapter 7.

II. Lab Tasks and Submission Guideline

Write complete C programs to solve the following three problems. Save the code you wrote to solve them, together with the result of it in a report. Make sure you include enough comments in your code.

For each problem, copy/paste the source code (in text, not image) and copy/ paste the execution results of each of the outputs (high-resolution screenshots) into your lab report. Save your report in .pdf format and submit it on D2L. You should submit your lab at the end of your lab session or soon after. In all cases it must be submitted before the deadline indicated in the D2L dropbox or it will not be accepted for marking.

Problem 1: A set of numbers x_i in the range x_{low} to x_{high} can be "normalized" into a new range *min* to *max*, by calculating each normxi from each x_i as follows:

$$\mathbf{\mathit{normx}_i = min + (x_i - x_{low}) * (max - min) / (x_{high} - x_{low})}$$

Write a complete C program to read from a data file an integer *count* (between 3 and 20) and two real numbers (min and max). Then read the set of real numbers (there are *count* numbers) for xi into an array. Normalize the xi values into the range min to max, placing the normalized values into a second array. Finally print the original and normalized values in a two-column table, with appropriate column headings.

Use the following sets of data to test your program. Put all three high-resolution screenshots in your report.

7 0.0 10.0 67.9 45.2 33.3 66.1 83.5 14.3 50.5

11 0.0 1.0 6.9 4.2 3.3 6.1 8.5 1.3 5.5 9.9 8.0 3.6 2.8

5 0.0 100.0 -34.3 50.9 0.0 43.2 -77.7

Problem 2: Figure 7.16 in the textbook (slide 38 / lesson 07 of the generic slides) shows a program that sorts an array. It is incomplete as it lacks the definition of the **get_min_range** function.

Complete the get_min_range function to make the program work to sort the array specified in the main program.

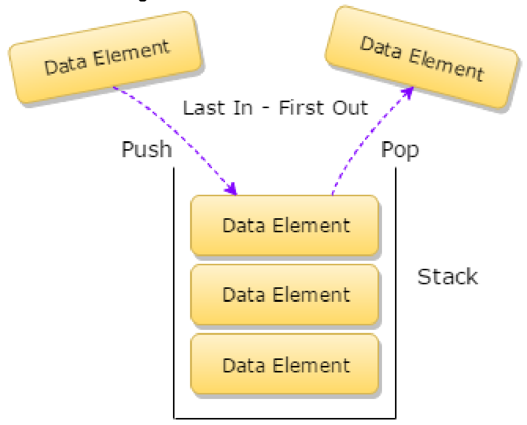
```
#include <stdio.h>
#define ARRAY_SIZE 8

// finds the position of the smallest element in the subarray
// list[first] through list[last].
// Pre: first < last and elements 0 through last of array list are
defined.
```

```
// Post: Returns the subscript k of the smallest element in the
subarray;
// i.e., list[k] <= list[i] for all i in the subarray
int get_min_range (int list[], int first, int last)
{


   /* complete the function here */


}

// sorts the data in array list
void
select_sort(int list[], int n)
{
     int fill,           /* index of first element in unsorted subarray
*/
         temp,           /* temporary storage
*/
         index_of_min;   /* subscript of next smallest element
*/

     for (fill = 0; fill < n-1; ++fill) {
          /* Find position of smallest element in unsorted subarray */
          index_of_min = get_min_range (list, fill, n-1);

          /* Exchange elements at fill and index_of_min */
          if (fill != index_of_min) {
                temp = list[index_of_min];
                list[index_of_min] = list[fill];
                list[fill] = temp;
          }
     }
}

int
main (void) {
    int array[] = {67, 98, 23, 11, 47, 13, 94, 58};
    int i;

    select_sort (array, ARRAY_SIZE);

    for (i=0; i < 8; ++i)
        printf ("%d ", array[i]);

    return (0);
}
```

Problem 3: Figure 7.13 in the textbook (slide 34 / lesson 07 of the generic slides) shows the two functions pop and push that deal with a stack of characters. Write the program by completing the main function that calls the push function three times, prints out the updated stack, then calls the pop function once and finally prints out the updated stack again.

```
#include <stdio.h>
#define STACK_EMPTY '0'
#define STACK_SIZE 20

void
push(char stack[],    /* input/output - the stack */
     char item,       /* input - data being pushed onto the stack */
     int  *top,       /* input/output - pointer to top of stack */
     int  max_size)   /* input - maximum size of stack */
{
     if (*top < max_size-1) {
         ++(*top);
         stack[*top] = item;
     }
}

char
pop (char stack[],    /* input/output - the stack */
    int *top)        /* input/output - pointer to top of stack */
{
    char item;       /* value popped off the stack */

    if (*top >= 0) {
        item = stack[*top];
        --(*top);
    } else {
         item = STACK_EMPTY;
    }

    return (item);
}

int
main (void)
{
   char s [STACK_SIZE];
   int s_top = -1; // stack is empty

   /* complete the program here */

   return (0);
}
```

Have fun!


Data Element
Data Element
Last In - First Out
Push
Pop
Data Element
Data Element
Data Element
Stack