

CC-112L

Programming Fundamentals

Laboratory 09

Pointers – II

Version: 1.0.0

Release Date: 16-09-2022

Department of Information Technology

University of the Punjab

Lahore, Pakistan

Contents:

- Learning Objectives
- Required Resources
- General Instructions
- Background and Overview
 - Relationship Between Pointers and Arrays
 - Array of Pointers
 - Function Pointers
- Activities
 - Pre-Lab Activity
 - Comparing Pointer
 - Key points about pointer comparison
 - Relationship Between Pointers and Arrays:
 - Pointer/Subscript Notation
 - Example code of Subscript and pointer notation
 - Exercise 1
 - String Copying with Arrays and Pointers
 - Copying with Array Subscript Notation
 - Copying with Pointers and Pointer Arithmetic
 - Task 01: Pointers & Addresses
 - In-Lab Activity
 - Arrays of Pointers
 - Example code of Arrays of pointers
 - Function Pointers
 - Sorting in Ascending or Descending Order
 - Function Pointer Parameter
 - Task 01: Copying strings
 - Task 02: Remove Duplicates from sorted array
 - Task 03: Plus one in Integer
 - Post-Lab Activity
 - Task 01: Array Addressing
- Submissions
- Evaluations Metric
- References and Additional Material
- Lab Time and Activity Simulation Log

Learning Objectives:

- Relationship Between Pointers and Arrays
- Array of Pointers
- Function Pointers

Resources Required:

- Desktop Computer or Laptop
- Microsoft ® Visual Studio 2022

General Instructions:

- In this Lab, you are **NOT** allowed to discuss your solution with your colleagues, even not allowed to ask how is s/he doing, this may result in negative marking. You can **ONLY** discuss with your Teaching Assistants (TAs) or Lab Instructor.
- Your TAs will be available in the Lab for your help. Alternatively, you can send your queries via email to one of the followings.

Teachers:		
Course Instructor	Prof. Dr. Syed Waqar ul Qounain	swjaffry@pucit.edu.pk
Lab Instructor	Madiha Khalid	madiha.khalid@pucit.edu.pk
Teacher Assistants	Usman Ali	bitf19m007@pucit.edu.pk
	Saad Rahman	bsef19m021@pucit.edu.pk

Background and Overview:

Relationship Between Pointers and Arrays:

Array in C is used to store elements of same types whereas Pointers are address variables which stores the address of a variable. Now array variable is also having a address which can be pointed by a pointer and array can be navigated using pointer. In simple words, array names are converted to pointers. That's the reason why you can use pointers to access elements of arrays. However, you should remember that pointers and arrays are not the same. There are a few cases where array names don't decay to pointers.

Array of Pointers:

An array of pointers is an indexed set of variables, where the variables are pointers (referencing a location in memory). Pointers are an important tool in computer science for creating, using, and destroying all types of data structures. Pointers are variables which stores the address of another variable. When we allocate memory to a variable, pointer points to the address of the variable. Unary operator (*) is used to declare a variable and it returns the address of the allocated memory.

Function Pointers:

Function Pointers point to code like normal pointers. In Functions Pointers, function's name can be used to get function's address. A function can also be passed as an argument and can be returned from a function. Function pointers can be useful when you want to create callback mechanism, and need to pass address of a function to another function. They can also be useful when you want to store an array of functions, to call dynamically.

Activities:

Pre-Lab Activities:

Comparing Pointers:

We can compare pointers if they are pointing to the same array. Relational pointers can be used to compare two pointers. Pointers can't be multiplied or divided. If pointers are compared using equality and relational operators it can be done, but such comparisons are meaningful only if the pointers point to elements of the same array; otherwise, such comparisons are logic errors. Pointer comparisons compare the addresses stored in the pointers. Such a comparison could show, for example, that one pointer points to a higher-numbered array element than the other. A common use of pointer comparison is determining whether a pointer is NULL

Example Code:

```
1  #include <stdio.h>
2
3  int main() {
4      int* p2;
5      int* p1;
6      p2 = (int*)300;
7      p1 = (int*)200;
8      if (p1 > p2) {
9          printf("P1 is greater than p2\n");
10     }
11     else {
12         printf("P2 is greater than p1\n");
13     }
14     return 0;
15 }
16
```

Fig. 01 (Comparing Pointer Example Code)

Output:



```
Microsoft Visual Studio Debug Console
P2 is greater than p1
```

Fig. 02 (Comparing Pointer Example Code Output)

Some Key points about pointer comparison:

- $p1 \leq p2$ and $p1 \geq p2$ both yield true and $p1 < p2$ and $p1 > p2$ both yield false, if two pointers $p1$ and $p2$ of the same type point to the same object or function, or both point one past the end of the same array, or are both null.
- $p1 < p2$, $p1 > p2$, $p1 \leq p2$ and $p1 \geq p2$ are unspecified, if two pointers $p1$ and $p2$ of the same type point to different objects that are not members of the same object or elements of the same array or to different functions, or if only one of them is null.
- If two pointers point to non-static data members of the same object, or to sub objects or array elements of such members, with same access control then the result is specified.
- The result is unspecified, if two pointers point to non-static data members of the same object with different access control.

Relationship Between Pointers and Arrays:

Array in C is used to store elements of same types whereas Pointers are address variables which stores the address of a variable. Now array variable is also having a address which can be pointed by a pointer and array can be navigated using pointer. In simple words, array names are converted to pointers. That's the reason why you can use pointers to access elements of arrays. However, you should remember that pointers and arrays are not the same. There are a few cases where array names don't decay to pointers.

Arrays and pointers are intimately related and often may be used interchangeably. You can think of an array name as a constant pointer to the array's first element. Pointers can be used to do any operation involving array subscripting. Assume the following definitions:

```
int b [5];  
int *bPtr;
```

Because the array name b (without a subscript) is a pointer to the array's first element, we can set bPtr to the address of the array b's first element with the statement:

```
bPtr = b;
```

This is equivalent to taking the address of array b's first element as follows:

```
bPtr = &b [0];
```

Pointer/Subscript Notation:

Pointers can be subscripted like arrays. If bPtr has the value b, the expression bPtr [1]

refers to the array element b [1]. This is referred to as pointer/subscript notation.

Example Code:

```
1  #include <stdio.h>  
2  
3  int main()  
4  {  
5      int b[] = { 10, 20, 30, 40 };  
6      int* bPtr = b;  
7      int i;  
8      int offset;  
9  
10     printf("\nPointer subscript notation\n");  
11  
12     for (i = 0; i < 4; i++) {  
13         printf("bPtr[ %d ] = %d\n", i, bPtr[i]);  
14     }  
15  
16     return 0;  
17 }
```

Fig. 03 (Pointer & Subscript notation)

Output:

```
Microsoft Visual Studio Debug Console

Pointer subscript notation
bPtr[ 0 ] = 10
bPtr[ 1 ] = 20
bPtr[ 2 ] = 30
bPtr[ 3 ] = 40
```

Fig. 04 (Pointer & Subscript Notation Output)

Exercise 1:

```
1  #include <stdio.h>
2
3
4  int main() {
5      int a[] = { 1,2,3,4,5 }, * p;
6      p = a;
7      ++* p;
8      printf("%d ", *p);
9      p += 2;
10     printf("%d ", *p);
11 }
```

Fig. 05 (Exercise 1)

Write expected output of above code:

Output >

String Copying with Arrays and Pointers:

To further illustrate array and pointer interchangeability, let's look at two string copying functions copy1 and copy2 in below example code. Both functions copy a string into a character array, but they're implemented differently.

```

1  #include <stdio.h>
2  #define SIZE 10
3  void copy1(char* const s1, const char* const s2); // prototype
4  void copy2(char* s1, const char* s2); // prototype
5
6  int main(void) {
7      char string1[SIZE]; // create array string1
8      char* string2 = "Hello"; // create a pointer to a string
9      copy1(string1, string2);
10     printf("string1 = %s\n", string1);
11     char string3[SIZE]; // create array string3
12     char string4[] = "Good Bye"; // create an array containing a string
13     copy2(string3, string4);
14     printf("string3 = %s\n", string3);
15 }
16 // copy s2 to s1 using array notation
17 void copy1(char* const s1, const char* const s2) {
18     // loop through strings
19     for (size_t i = 0; (s1[i] = s2[i]) != '\0'; ++i) {
20         // do nothing in body
21     }
22 }
23 // copy s2 to s1 using pointer notation
24 void copy2(char* s1, const char* s2) {
25     // loop through strings
26     for (; (*s1 = *s2) != '\0'; ++s1, ++s2) {
27         // do nothing in body
28     }
29 }
30 }

```

Fig. 06 (Copying string Using Pointer)

Copying with Array Subscript Notation:

Function `copy1` uses array subscript notation to copy the string in `s2` to the character array `s1`. The function defines counter variable `i` as the array subscript. The for-statement header (line 17) performs the entire copy operation. The statement's body is the empty statement. The header specifies that `i` is initialized to zero and incremented by one during each iteration. The expression `s1[i] = s2[i]` copies one character from `s2` to `s1`. When the null character is encountered in `s2`, it's assigned to `s1`. Since the assignment's value is what gets assigned to the left operand (`s1`), the loop terminates when an element of `s1` receives the null character, which has the value 0 and therefore is false.

Output:



```

Microsoft Visual Studio Debug Console
string1 = Hello
string3 = Good Bye

```

Fig. 07 (Coping with subscript notation)

Copying with Pointers and Pointer Arithmetic:

Function `copy2` uses pointers and pointer arithmetic to copy the string in `s2` to the character array `s1`. Again, the for-statement header (line 34) performs the copy operation. The header does not include any variable initialization. The expression `*s1 = *s2` performs the copy operation by dereferencing `s2` and assigning that character to the current location in `s1`. After the assignment, line 34 increments `s1` and `s2` to point to each string's next character. When the assignment copies the null character into `s1`, the loop terminates.

Task 01: Pointer & Addresses

[Estimated 20 minutes / 20 marks]

Add the appropriate lines of code in the program given above. Print the following things and fill in the table.

Given Code:

```

1  #include <stdio.h>
2
3
4  int main()
5  {
6      int var1 = 10, var2 = 5;
7      int* p1, * q1, * r1;
8      int var3;
9      var3 = ++var2;
10     p1 = &var1;
11     q1 = &var2;
12     r1 = &var3;
13     *p1 = var3++;
14     *q1 = ++var1;
15     //add appropriate lines of code to print the things mentioned in the table below
16
17
18     return 0;
19 }
    
```

Fig. 08 (Pre-Lab Task)

Address of variable var1:	
Address of variable var2:	
Address of variable var3:	
Address of pointer variable p1:	
Address of pointer variable q1:	
Address of pointer variable r1:	
Value at location pointed by variable p1:	
Value at location pointed by variable q1:	
Value at location pointed by variable r1:	
Value at location that is pointed by p1	
Address of location that is pointed by p1	

Submit “.c” file named your “**Table**” on Google Classroom.

Task 02: Pointer & Addresses

[Estimated 20 minutes / 20 marks]

Write a program in C to demonstrate the use of & (address of) and *(value at address) operator.

Expected Output:

```
Select Microsoft Visual Studio Debug Console

Pointer : Demonstrate the use of & and * operator :
-----
m = 300
fx = 300.600006
cht = z

Using & operator :
-----
address of m = 0000007A883EF804
address of fx = 0000007A883EF824
address of cht = 0000007A883EF844

Using & and * operator :
-----
value at address of m = 300
value at address of fx = 300.600006
value at address of cht = z

Using only pointer variable :
-----
address of m = 0000007A883EF804
address of fx = 0000007A883EF824
address of cht = 0000007A883EF844

Using only pointer operator :
-----
value at address of m = 300
value at address of fx= 300.600006
value at address of cht= z
```

Fig. 08 (Pre-Lab Task)

In-Lab Activities:**Arrays of Pointers:**

Let us consider the following example, which uses an array of 3 integers.

```
1  #include <stdio.h>
2
3  int main() {
4
5      int var[] = { 10, 100, 200 };
6      int i, * ptr[3];
7
8      for (i = 0; i < 3; i++) {
9          ptr[i] = &var[i]; /* assign the address of integer. */
10     }
11
12     for (i = 0; i < 3; i++) {
13         printf("Value of var[%d] = %d\n", i, *ptr[i]);
14     }
15
16     return 0;
17 }
```

Fig. 09 (Arrays of Pointers)

Output:

```
Microsoft Visual Studio Debug Console
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

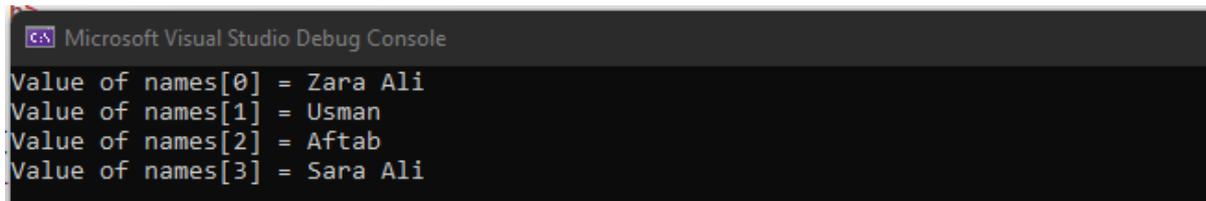
Fig. 10 (Arrays of Pointers Output)

You can also use an array of pointers to character to store a list of strings as follows:

Example Code:

```
1  #include <stdio.h>
2
3  int main() {
4
5      char* names[] = {
6          "Zara Ali",
7          "Usman",
8          "Aftab",
9          "Sara Ali"
10     };
11     int i = 0;
12     for (i = 0; i < 4; i++) {
13         printf("Value of names[%d] = %s\n", i, names[i]);
14     }
15
16     return 0;
17 }
```

Fig. 11 (Example Code of Arrays of Pointers)

Output:A screenshot of the Microsoft Visual Studio Debug Console. The title bar reads "Microsoft Visual Studio Debug Console". The console output shows four lines of text: "Value of names[0] = Zara Ali", "Value of names[1] = Usman", "Value of names[2] = Aftab", and "Value of names[3] = Sara Ali".

```
Value of names[0] = Zara Ali
Value of names[1] = Usman
Value of names[2] = Aftab
Value of names[3] = Sara Ali
```

Fig. 12 (Example Code of Arrays of Pointers Output)

Function Pointers:

In previous examples an array name is really the address in memory of the array's first element. Similarly, a function's name is really the starting address in memory of the code that performs the function's task. A pointer to a function contains the address of the function in memory. Pointers to functions can be passed to functions, returned from functions, stored in arrays, assigned to other function pointers of the same type and compared with one another for equality or inequality.

Sorting in Ascending or Descending Order:

To demonstrate pointers to functions, below code presents a modified version of bubble-sort program. The new version consists of main and functions bubbleSort, swap, ascending and descending. Function bubbleSort receives a pointer to a function as an argument either function ascending or function descending in addition to an int array and the array's size. The user chooses whether to sort the array in ascending (1) or descending (2) order. If the user enters 1, main passes a pointer to function ascending to function bubbleSort. If the user enters 2, main passes a pointer to function descending to function bubbleSort.

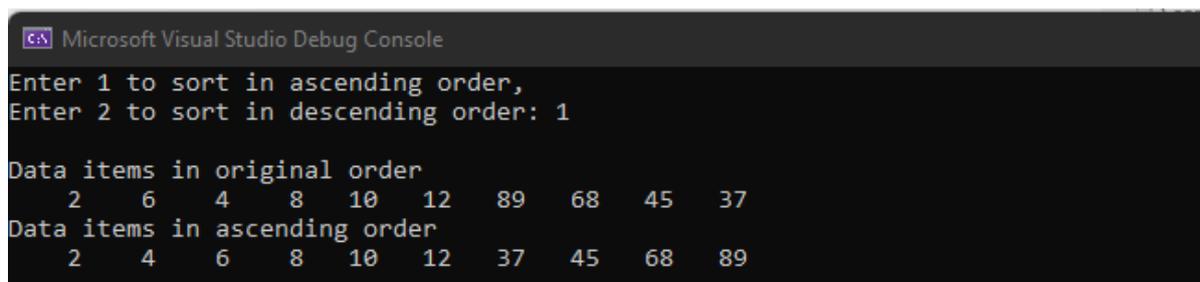
```

1  #include <stdio.h>
2  #define SIZE 10
3  // prototypes
4  void bubbleSort(int work[], size_t size, int (*compare)(int a, int b));
5  int ascending(int a, int b);
6  int descending(int a, int b);
7
8  int main(void) {
9      // initialize unordered array a
10     int a[SIZE] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
11     printf("%s", "Enter 1 to sort in ascending order,\n" "Enter 2 to sort in descending order: ");
12     int order = 0;
13     scanf("%d", &order);
14     puts("\nData items in original order");
15     // output original array
16     for (size_t counter = 0; counter < SIZE; ++counter) {
17         printf("%5d", a[counter]);
18     }
19     if (order == 1) {
20         bubbleSort(a, SIZE, ascending);
21         puts("\nData items in ascending order");
22     }
23     else { // pass function descending
24         bubbleSort(a, SIZE, descending);
25         puts("\nData items in descending order");
26     }
27     // output sorted array
28     for (size_t counter = 0; counter < SIZE; ++counter) {
29         printf("%5d", a[counter]);
30     }
31     puts("\n");
32 }
33
34 // multipurpose bubble sort; parameter compare is a pointer to
35 // the comparison function that determines sorting order
36 void bubbleSort(int work[], size_t size, int (*compare)(int a, int b)) {
37     void swap(int* element1Ptr, int* element2Ptr); // prototype
38     // loop to control passes
39     for (int pass = 1; pass < size; ++pass) {
40         // loop to control number of comparisons per pass
41         for (size_t count = 0; count < size - 1; ++count) {
42             // if adjacent elements are out of order, swap them
43             if ((*compare)(work[count], work[count + 1])) {
44                 swap(&work[count], &work[count + 1]);
45             }
46         }
47     }
48 }
49
50
51 void swap(int* element1Ptr, int* element2Ptr) {
52     int hold = *element1Ptr;
53     *element1Ptr = *element2Ptr;
54     *element2Ptr = hold;
55 }
56
57 // determine whether elements are out of order for an ascending order sort
58 int ascending(int a, int b) {
59     return b < a; // should swap if b is less than a
60 }
61
62 // determine whether elements are out of order for a descending order sort
63 int descending(int a, int b) {
64     return b > a; // should swap if b is greater than a
65 }
66

```

Fig. 13 (Example Code of Function Pointers)

Output 1:



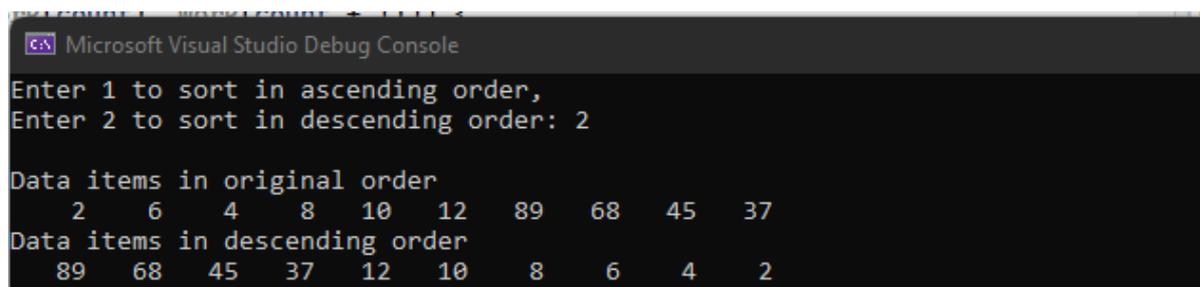
```

Microsoft Visual Studio Debug Console
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1

Data items in original order
 2  6  4  8 10 12 89 68 45 37
Data items in ascending order
 2  4  6  8 10 12 37 45 68 89

```

Fig. 14 (Output of code of Function Pointers)

Output 2:


```

Microsoft Visual Studio Debug Console
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 2

Data items in original order
 2  6  4  8 10 12 89 68 45 37
Data items in descending order
89 68 45 37 12 10 8 6 4 2

```

Fig. 15 (Output of Code of Function Pointers)

Function Pointer Parameter:

The following parameter appears in the function header for bubbleSort (line 36):

```
int (*compare) (int a, int b)
```

This tells bubbleSort to expect a parameter (compare) that's a pointer to a function, specifically for a function that receives two int and returns an int result. The parentheses around *compare is required to group the * with compare and indicate that compare is a pointer. Without the parentheses, the declaration would have been:

```
int *compare (int a, int b)
```

which declares a function that receives two integers as parameters and returns a pointer to an integer.

To call the function passed to bubbleSort via its function pointer, we dereference it, as shown in the if statement at line 43:

```
if ((*compare) (work[count], work [count + 1])).
```

The call to the function could have been made without dereferencing the pointer as in

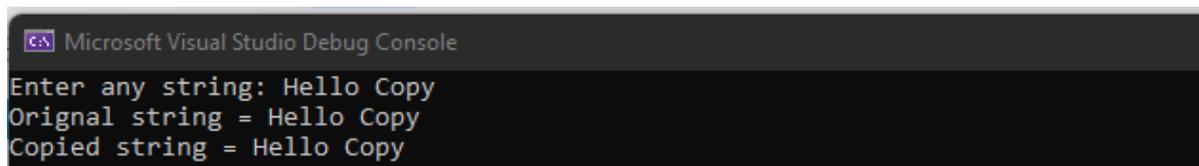
```
if (compare(work[count], work [count + 1])),
```

which uses the pointer directly as the function name. The first method of calling a function through a pointer explicitly shows that compare is a pointer to a function that's dereferenced to call the function. The second technique makes it appear that compare is an actual function name. This may confuse someone reading the code who'd like to see compare's function definition and finds that it's never defined.

Task 01: Copy string**[30 minutes / 20 marks]**

Write a C program to copy one string to another string using loop. You are not allowed to use inbuilt function strcpy () in C.

- Input a string from the user.
- Output original and copied string on the console.

Sample Output:

```
Microsoft Visual Studio Debug Console
Enter any string: Hello Copy
Original string = Hello Copy
Copied string = Hello Copy
```

Fig. 16 (In-Lab Task 01)

Submit “.c” file named your “CopyStr” on Google Classroom.

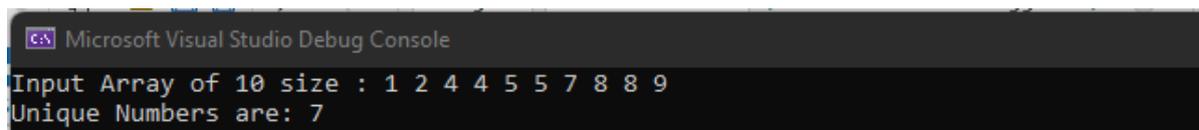
Task 02: Remove Duplicates from Sorted Array**[40 minutes / 30 marks]**

Given an integer pointer to array nums of size 10 sorted in increasing order, remove the duplicates in-place such that each unique element appears only once. The relative order of the elements should be kept the same.

Since it is impossible to change the length of the array in some languages, you must instead have the result be placed in the first part of the array nums. More formally, if there are k elements after removing the duplicates, then the first k elements of nums should hold the final result. It does not matter what you leave beyond the first k elements.

Return k after placing the final result in the first k slots of nums.

Do not allocate extra space for another array. You must do this by modifying the input array in-place with O (1) extra memory.

Sample Output:

```
Microsoft Visual Studio Debug Console
Input Array of 10 size : 1 2 4 4 5 5 7 8 8 9
Unique Numbers are: 7
```

Fig. 17 (In-Lab Task 02)

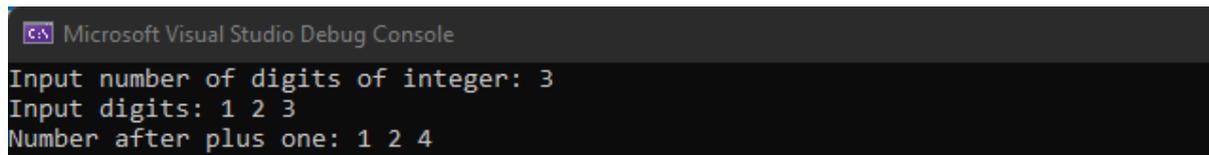
Submit “.c” file named your “Removing” on Google Classroom.

Task 03: Plus One in a integer**[40 minutes / 30 marks]**

You are given a large integer represented as an integer array to pointer representing digits, where each digit [i] is the ith digit of the integer. The digits are ordered from most significant to least significant in left-to-right order. The large integer does not contain any leading 0's.

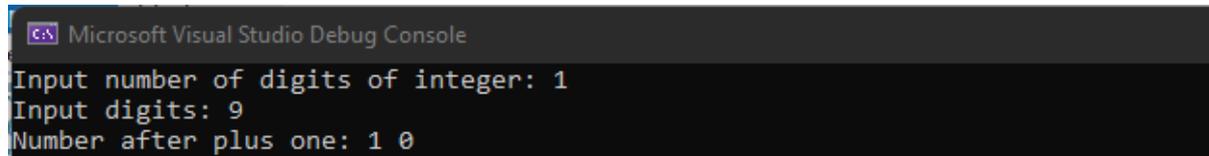
Increment the large integer by one and return the resulting array of digits.

Sample Output I:



```
Microsoft Visual Studio Debug Console
Input number of digits of integer: 3
Input digits: 1 2 3
Number after plus one: 1 2 4
```

Fig. 18 (In-Lab Task 03)

Sample Output II:

```
Microsoft Visual Studio Debug Console
Input number of digits of integer: 1
Input digits: 9
Number after plus one: 1 0
```

Fig. 19 (In-Lab Task 01)

Submit “.c” file named your “**PlusOne**” on Google Classroom.

Post-Lab Activities:

Task 01: Array Addressing

[Estimated 60 minutes / 60 marks]

Part (a):

In main (), write a logic that declares two arrays A and B with sizes as shown below. Array B will be of type int i.e., it stores integers.

Print the address of location of cells of both arrays.

Your next task is to create a following link between them. You can create a link / point locations to each other by using pointers concept. That is, A [0] is pointing towards B [1] and so on. At the end, display the contents of both the arrays.

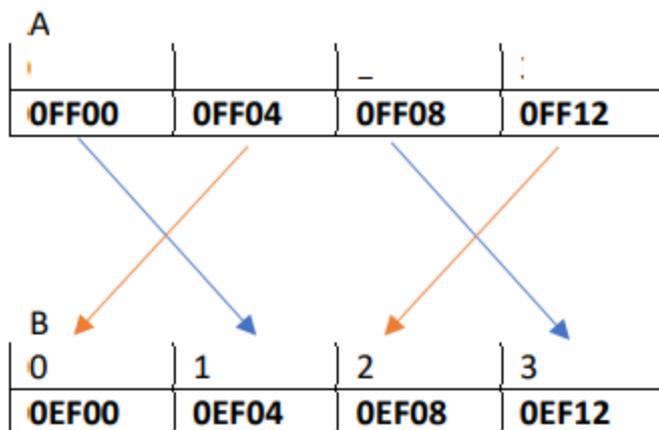


Fig. 20 (Post-Lab Task 01)

Part (b)

1. Consider an array A (take input from user) consisting of some fixed number of integers.
2. Iterate through the array (w/o pointer) using a loop, find out and store the indices of elements that have even integers. The indices must be stored in another array B.
3. 3. Declare an array of pointers C. The cells of this array should point to the respective entries of array A at indices specified in array B.

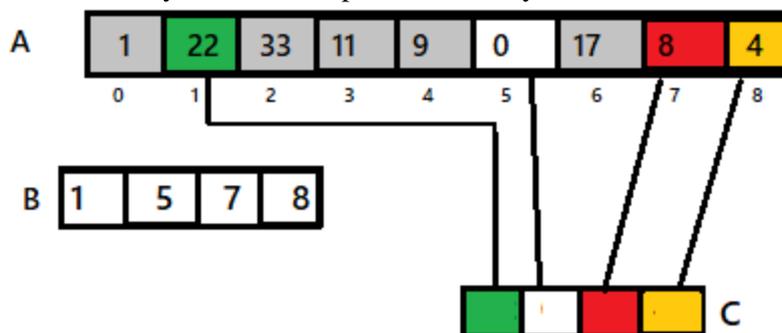


Fig. 20 (Post-Lab Task 01)

Submit “.c” file named your “Addressing” on Google Classroom.

Submissions:

- For In-Lab Activity:
 - Save the files on your PC.
 - TA's will evaluate the tasks offline.
- For Pre-Lab & Post-Lab Activity:
 - Submit the .c file on Google Classroom and name it to your roll no.

Evaluations Metric:

- All the lab tasks will be evaluated offline by TA's
- **Division of Pre-Lab marks:** [40 marks]
 - Task 01: Pointers & Addresses [20 marks]
 - Task 02: Pointers & Addresses [20 marks]
- **Division of In-Lab marks:** [80 marks]
 - Task 01: Copying strings [20 marks]
 - Task 02: Remove Duplicates from sorted array [30 marks]
 - Task 03: Plus one Integer [30 marks]
- **Division of Post-Lab marks:** [30 marks]
 - Task 01: Array Addressing [30 marks]

References and Additional Material:

- Relationship Between arrays and pointers
<https://www.programiz.com/c-programming/c-pointers-arrays>
- Array of pointers
<https://www.programiz.com/cpp-programming/pointers-arrays>
- Function Pointers
<https://www.programiz.com/c-programming/c-pointer-functions>

Lab Time Activity Simulation Log:

- Slot – 01 – 00:00 – 00:15: Class Settlement
- Slot – 02 – 00:15 – 00:30: Execute C on Visual Studio
- Slot – 03 – 00:30 – 00:45: Execute C on Visual Studio
- Slot – 04 – 00:45 – 01:00: In-Lab Task
- Slot – 05 – 01:00 – 01:15: In-Lab Task
- Slot – 06 – 01:15 – 01:30: In-Lab Task
- Slot – 07 – 01:30 – 01:45: In-Lab Task
- Slot – 08 – 01:45 – 02:00: In-Lab Task
- Slot – 09 – 02:00 – 02:15: In-Lab Task
- Slot – 10 – 02:15 – 02:30: In-Lab Task
- Slot – 11 – 02:30 – 02:45: Evaluation of Lab Tasks
- Slot – 12 – 02:45 – 03:00: Discussion on Post-Lab Task