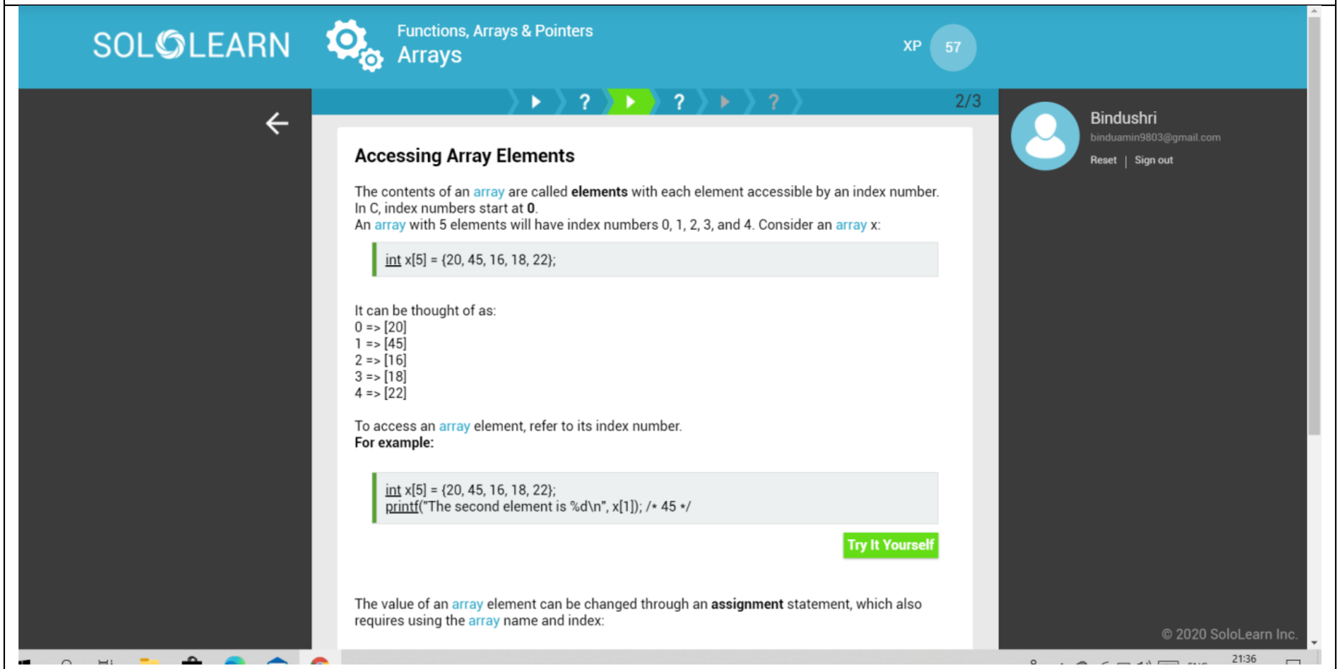
DAILY ASSESSMENT FORMAT

Date:	**19-06-2020**	**Name:**	**BINDUSHRI**
Course:	**C programming**	**USN:**	**4AL17EC011**
Topic:	**Pointers** **Functions** **Memory management** **Accessing structure member**		**6th A**
Github Repository:	**Bindushri**		

FORENOON SESSION DETAILS



Accessing Array Elements

The contents of an array are called **elements** with each element accessible by an index number.
In C, index numbers start at **0**.
An array with 5 elements will have index numbers 0, 1, 2, 3, and 4. Consider an array x:

```
int x[5] = {20, 45, 16, 18, 22};
```

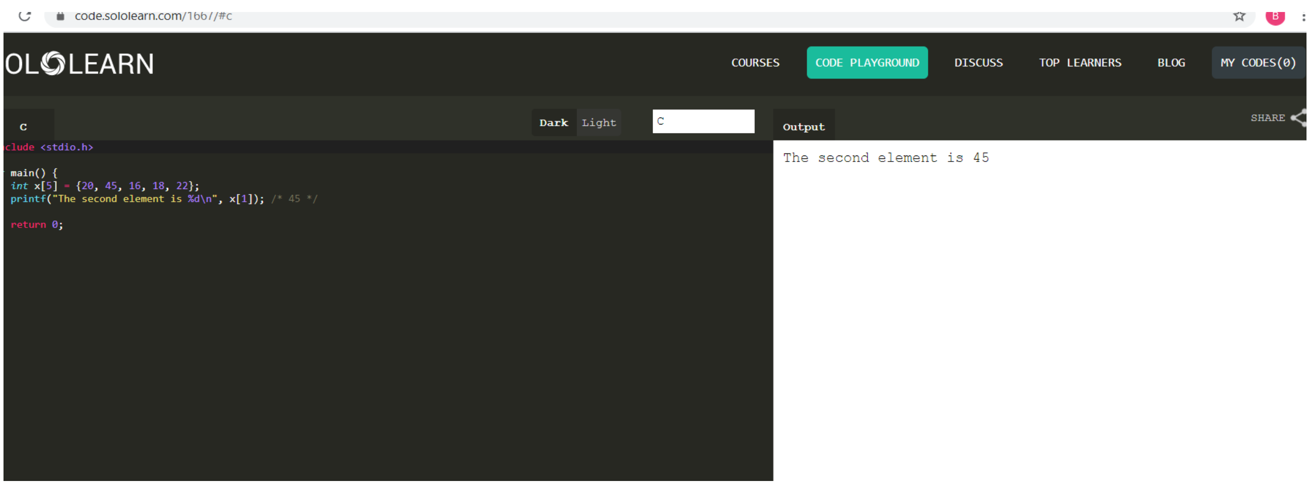
It can be thought of as:
0 => [20]
1 => [45]
2 => [16]
3 => [18]
4 => [22]

To access an array element, refer to its index number.
For example:

```
int x[5] = {20, 45, 16, 18, 22};
printf("The second element is %d\n", x[1]); /* 45 */
```

Try It Yourself

The value of an array element can be changed through an **assignment** statement, which also requires using the array name and index:



```
clude <stdio.h>

 main() {
 int x[5] = {20, 45, 16, 18, 22};
 printf("The second element is %d\n", x[1]); /* 45 */

 return 0;
```

```
The second element is 45
```

19/06/2020

Function parameters.

function parameters are used to receive value required by the function

```
#include<stdio.h>
int sum_up(int x, int y);
int main(){
  int x, y, result;
   x=3;
   y= 12).
  result = sum_up (x, y);
 printf ("%d+%d=%d", x, y, result);
  ~~result~~
  return0;
}
 int sum_up(int x, int y) {
   x += y;
   return (x);
 }
```

variable Scope

variable scope refers to the visibility of variables with in a program.

```
#include<stdio.h>
  int global =0;
  int main(){
  int local1, local2;
   local1 =5;
   local2=10;
  global= local1 + local2;
  printf("%d \n", global);
  return0;
}
```

→ Static variables

Static variables have a local scope but are not destroyed when a function is exited.

```
#include <stdio.h>
void say_hello ();
 int main(){
 int i;
for (i=0; i<5; i++){
  say_hello();
 }
 return0;
 }
void say_hello (){
 static int num_calls=1;
 printf("Hello number %d\n",
    num_calls);
 num_calls++;
 }
```

Recursive function.

An algorithm for solving a problem may be best implemented using a process called recursion.

Ex!- 5! = 5x4x3x2x1

```
#include<stdio.h>
int factorial (int num);
int main(){
 int x=5;
 printf ("the factorial of %d is %d",
         x, factorial (x));
 return0;
}
```

```
int factorial (int num) {
 if (num == 1)
  return (1);
 else
 return (num * factorial(num-1));
}
```

Arrays in C

A array is a data structure that stores collection of related values that are all same type.

```
int test_Scores [25];
float prices [5] = {3.2, 6.2, 10.2,
                    1.0, 2.33};
```

Accessing Array Elements

the contents of an array are elements.

```
#include <stdio.h>
int main() {
   int x[5] = {20, 45, 16, 18, 22};
   printf ("the seccond element
          is %d\n", x[1]);
   return 0;
 }
```

o/p 45.

Using loops with Array

many algorithm require accessing every element of an array!

```
float purchases [3] =
{10.99, 14.25, 90.50};
float total=0;
int k;
for (k=0; k<3; k++){
  total += purchase [k];
}
printf ("purchase total
 is %6.2f\n", total);
```

Two-dimensional Arrays

two dimensional array is an array of arrays and can be thought of as a table.

```
int a [2] [3];
```

ex:-
```
int [2] [3] = {
     {3, 2, 6},
     {4, 5, 20}
     };
```

Pointers

using memory

As c is base-level-language that can easily acess memory locations and perform memory related operation.

what is pointer

Pointers are very important in c programming because they allow you to easily work with memory locations.

```
Pointer-type * identifier

int j=63
int *p=NULL
P = &j;

Print ("The address of j is
     %x\n", &j);

Printf ("p contains address %x\n",
     p);

Printf ("the value of j is %d\n",
     j);

Printf ("p is pointing to the value
     %d\n", *p);
```

Pointers and Arrays.

Pointers are especially useful with Arrays. An array declaration reserves a block of contiguous elements.

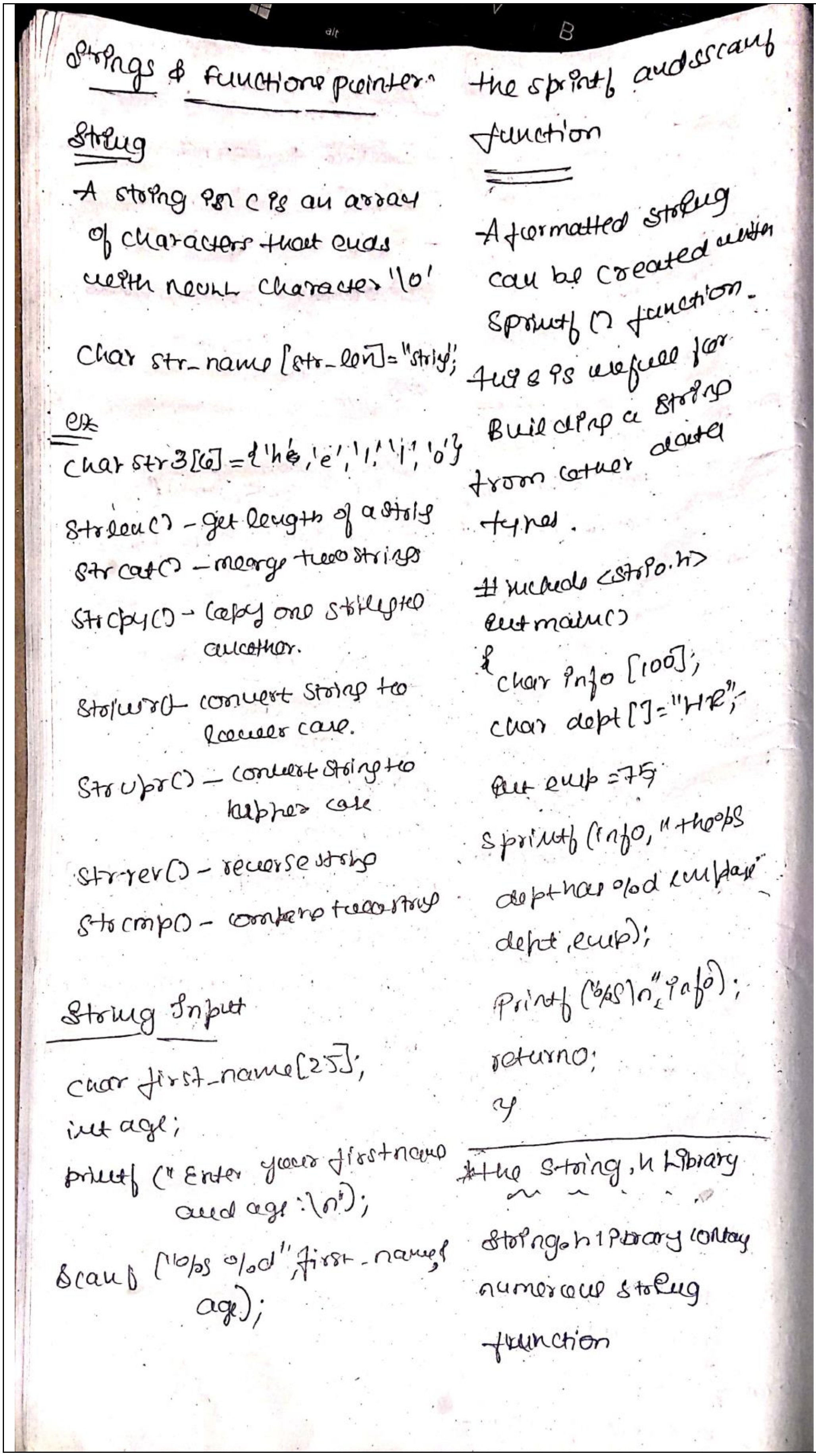
+ used to forward to a memory location

- used to move backward

```
int a[5]={22,33, 44,55,66};
int *ptr = NULL;
int i;
Ptr = a;
for( i=0; i<5; i++){
Printf ("%d", *(Ptr+i));
}
```

Pointers and functions.

```
void swap (int* num1, int* num2)
int main() {
int x=25;
int y=100;
Printf ("x is %d, y is %d\n", x,y);
swap (&x, &y)
Printf ("x is %d, y is %d\n", x,y);
return 0;
}
void swap (int* num1, int* num2)
int temp
temp = *num1;
*num1 = *num2;
*num2 = temp;
```



Strings & functions pointer

String

A string in c is an array of characters that ends with null character '\0'

```
Char str_name [str_len]="string";
```

ex

```
char str3[6] = {'he','e','l','l','o'}
```

strlen() - get length of a string

strcat() - merge two strings

strcpy() - copy one string to another.

strlwr() - convert string to lower case.

strupr() - convert string to higher case

strrev() - reverse string

strcmp() - compare two string

String Input

```
char first_name[25];
int age;
printf ("Enter your firstname
        and age:\n");
scanf ("%s %d", first_name,
        age);
```

the sprintf and sscanf function

A formatted string can be created with sprintf () function. this is useful for Building a string from other data types.

```
#include <stdio.h>
int main()
{
char info [100];
char dept[]="HR";
int emp =75
sprintf (info, "the %s
dept has %d employees",
dept, emp);
printf ("%s\n", info);
return0;
}
```

the string.h Library

string.h library contain numerous string function

```
#include <stdio.h>
#include <string.h>
int main()
{
 char s1[] = "the grey fox";
 char s2[] = "jumped";
 strcat (s1, s2);
 printf ("%s\n", s1);
 printf ("Length of s1 is %d\n",
        strlen (s1));

 strcpy (s1, s2);
 printf ("s1 is now %s\n", s1);
 return0;
 }
```

→ Converting a string to a number

Converting a string of number characters to a numeric value is a common task in C program

int atoi (str) stands for ASCII to integer.

```
#include <stdio.h>
 int main()
{
 char input[10];
 int num;
 printf ("Enter a number");
 gets(input);
 num = atoi(input);
 return0;
}
```

Array of string

A two-dimensional array can be used to store related strings.

```
char trip [3][15] = {
   "suitcase",
   "passport",
   "ticket"
 };
```

* Function Pointers.

Since pointers can point to an address in any memory location, they can also point to the start of executable code

```
return_type (*func_name)
            (parameters)
```

```
#include <stdio.h>
void say_hello (int num_times);
int main(){
 void (*funptr) (int);
 funptr = say_hello;
 funptr (3);
 return0;
}
void say_hello (int num_times){
int k;
for (k=0; k< num_times; k++)
 printf ("Hello\n");
}
```

The void pointer

A void pointer is used to refer to any address type in memory and has a declaration that look like void *ptr.

```
int x=33;
float y=12.4;
char c='a';

void *ptr;
ptr = &x;
Printf ("void ptr points to
        %d\n", *(int *) ptr));

ptr = &y;
printf ("void ptr points
        to %f\n", *(float *)
        ptr));

ptr = &c;
printf ("void ptr points to
        %c ", *((char *) ptr));
```

Function using void pointer

```
void* square (const void*);
```

Function pointers as argument

* Structures & unions

=> A Structure is user-defined data types that groups related variables of different data types

=> A structure declaration includes the keyword struct, a structure tag for referencing the structure, & curly braces { } with a list of variable declarations called members.

```
struct course{
  int id;
  char title [40];
  float hours;
};
```

Declaractions using struct

the statements below declare a structure data type and then uses the student struct to declare variables s1 and s2.

```
struct student{
int age;
int grade;
char name[40];
};
struct student s1;
struct student s2;
struct student s1={19, 4, "John"};
struct student s2={22, 10, "Batman"};
```

Accessing structure Members

```
s1.age = 19;

#include <stdio.h>
#include <string.h>
struct course{
int id;
char title [40];
float hours;
}
int main(){
struct course cs1={341279,"Intro",12.5},
struct course cs2;
cs2.id = 341281;
strcpy(cs2.title,"advanced C++");
  cs2.hours = 14.25;
printf("%d\t%s\t% 4.2f\n",
          cs1.id, cs1.title, cs1.hrs);
return0;
}
```

using typedef

```
typedef struct {
   int id;
    char title [40];
    float hours;
   } course;
    course cs1;
    course cs2;
```

working with structures

```
typedef struct{
  int x;
  int y;
} point;
typedef struct{
float radius;
point center;
} circle;
```

Pointers to the structure

```
struct my struct *struct_ptr;
                 ↳ points.
struct_ptr = &struct_var;
             ↳ stores.
struct_ptr -> struct_mem;
               ↓
               access the value of
                struct_mem.
```

```
struct student {
      char name [50];
      int number;
      int age;
   };
void showstudentData(struct
                          student *st){
printf("\n student :\n");
printf("name:%s\n", st->name);
printf("number: %d\n", st->number);
printf("Age :%d\n", st->age);
}
struct student st1 = {"Krishna",5,21};
showstudentData(&st1);
```

Unions

A union allows to store different data types in the Same memory location.

Union -> Key record.

```
union val{
   int int_num;
   float fl_num;
   char str[20];
 };
```

Pointers to unions

Pointer to a union points to the memory location allocated to the union.

```
union val{
   int int_num;
   float fl_num;
   char str[20];
 };
union val info;
 union val *ptr=NULL;
   ptr = &info;
   ptr -> int_num=10;
Printf("info.int_num is
        %d", info.int_num);
```

Unions as function Parameters

```
union id {
   int id_num;
   char name[20];
 };
void set_id(union id *
                item){
 item->id_num=42;
}
void show_id(union id
                item){
 printf("ID is %d",
 item.id_num);
}
```

Array unions

```
union val{
 int int_num;
 float fl_num;
 char str[20];
};
union val nums[10];
int k;
for(k=0;k<10;k++){
 nums[k].int_num=k;
}
for(k=0;k<10;k++){
 printf("%d", nums[k].
   int_num);
}
```

Memory Management

```
int x
printf ("%d", sizeof (x));
```

Memory management functions

the stdlib.h library include memory management functions.

→ #include <stdlib.h>
↓
gives access to.

malloc (bytes) → Returns a pointer to contiguous block of memory

calloc (num_items, item_size) Returns a pointer to a contiguous block of memory!

realloc (ptr, bytes) resizes the memory pointed to by ptr to size bytes.

free (ptr) releases the block of memory pointed to by ptr.

* The Malloc function

malloc() function allocates a specified number of contiguous bytes in memory

```
#include<stdlib.h>
int *ptr
ptr = malloc(10 * sizeof(*ptr));
if (ptr != NULL) {
*(ptr+2) = 50;
}
```

Malloc function can be allocate memory is contiguous and can be treated as an array instead of using brackets [] to refer to elements, pointer arithmetic is used to traverse the array.

The free function

free() function is a memory management function that is called release memory.

```
int *ptr = malloc (10 * sizeof(*ptr))
if (ptr != NULL)
*(ptr+2) = 50;
printf ("%d \n", *(ptr+2));
free (ptr);
```

* Calloc function

calloc() function allocates memory based on the size of a specific item, such as a structure.

realloc

realloc() function expands a current block to include additional memory.

Allocating Memory for strings.

when allocating memory for a string pointer, you may want to use string length rather than the size of operator for calculating bytes.

```
char str20[20];
char *str=NULL;
strcpy(str20,"12345");
str = malloc (strlen (str20)+1);
strcpy(str, str20);
printf("%s", str);
```

Dynamic Arrays

many algorithms implement a dynamic array because this allows the number of elements to grow as needed.

```
typedef struct{
    int * elements;
    int size;
    int cap;
} dyn_array;
  dyn_array arr;
  arr.size=0
          e =calloc(1,sizeof
arr.cap=1;
```

Files & Error Handling

working with files

Accessing files

An external file can be opened, read