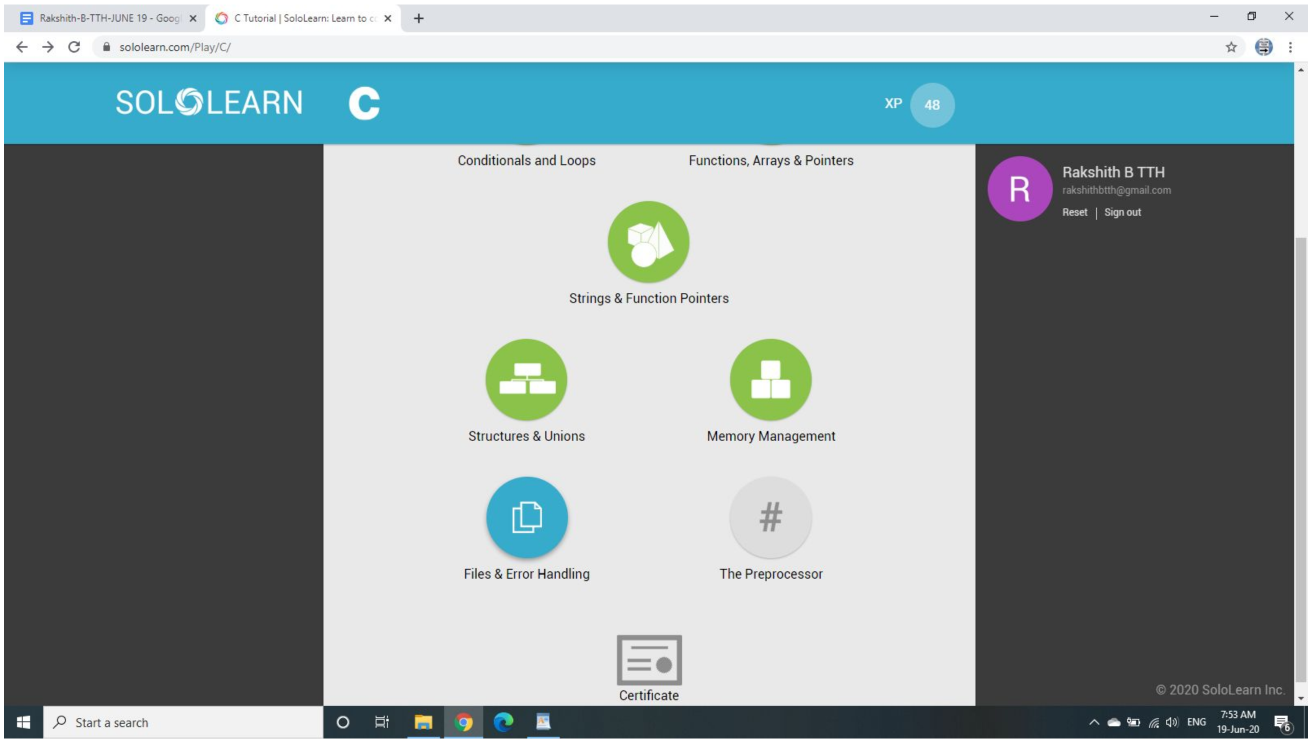
REPORT JUNE 19

Date:	19 JUNE 2020	Name:	Rakshith B
Course:	Solo Learning	USN:	4AL16EC409
Topic:	Structure and Union,Memory Management.	Semester & Section:	6th SEM B
Github Repository:	Rakshith-B		

FORENOON SESSION DETAILS

Image of session



Report –

Structures

A structure is a user-defined data type that groups related variables of different data types.

A structure declaration includes the keyword struct, a structure tag for referencing the structure, and curly braces { } with a list of variable declarations called members.
For example:

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

```
 float hours;
};
```

This struct statement defines a new data type named course that has three members.
Structure members can be of any data type, including basic types, strings, arrays, pointers, and even other structures, as you will learn in a later lesson.

Declarations Using Structures

To declare variables of a structure data type, you use the keyword struct followed by the struct tag, and then the variable name.
For example, the statements below declares 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];
};

/* declare two variables */
struct student s1;
struct student s2;
```

Accessing Structure Members

You access the members of a struct variable by using the . (dot operator) between the variable name and the member name.
For example, to assign a value to the age member of the s1 struct variable, use a statement like:

```
s1.age = 19;
```

You can also assign one structure to another of the same type:

```
struct student s1 = {19, 9, "Jason"};

struct student s2;

//....

s2 = s1;
```

Using typedef

The typedef keyword creates a type definition that simplifies code and makes a program easier to read.
typedef is commonly used with structures because it eliminates the need to use the keyword struct when declaring variables.
For example:

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

course cs1;
course cs2;
```

Pointers to Structures

Just like pointers to variables, pointers to structures can also be defined.

struct myStruct *struct_ptr;
defines a pointer to the *myStruct* structure.

struct_ptr = &struct_var;
stores the address of the structure variable *struct_var* in the pointer *struct_ptr*.

struct_ptr -> struct_mem;
accesses the value of the structure member *struct_mem*.

For example:

```
struct student{
 char name[50];
 int number;
 int age;
};

// Struct pointer as a function parameter
void showStudentData(struct student *st) {
 printf("\nStudent:\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);
```

Array of Structures

An array can store elements of any data type, including structures.
After declaring an array of structures, an element is accessible with the index number.
The dot operator is then used to access members of the element, as in the program:

```
#include <stdio.h>

typedef struct {
 int h;
 int w;
 int l;
} box;

int main() {
 box boxes[3] = {{2, 6, 8}, {4, 6, 6}, {2, 6, 9}};
 int k, volume;

 for (k = 0; k < 3; k++) {
   volume = boxes[k].h*boxes[k].w*boxes[k].l;
   printf("box %d volume %d\n", k, volume);
 }
 return 0;
}
```

Unions

A union allows to store different data types in the same memory location.
It is like a structure because it has members. However, a union variable uses the same memory location for all its member's and only one member at a time can occupy the memory location.

A union declaration uses the keyword union, a union tag, and curly braces { } with a list of members.

Union members can be of any data type, including basic types, strings, arrays, pointers, and structures.

Accessing Union Members

You access the members of a union variable by using the . dot operator between the variable name and the member name.
When assignment is performed, the union memory location will be used for that member until another member assignment is performed.

Trying to access a member that isn't occupying the memory location gives unexpected results.

The following program demonstrates accessing union members:

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

union val test;

test.int_num = 123;
test.fl_num = 98.76;
strcpy(test.str, "hello");

printf("%d\n", test.int_num);
printf("%f\n", test.fl_num);
```

```
    printf("%s\n", test.str);
```

Pointers to Unions

A pointer to a union points to the memory location allocated to the union.
A union pointer is declared by using the keyword union and the union tag along with * and the pointer name.
For example, consider the following statements:

```
    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);
```

Memory Management

Understanding memory is an important aspect of C programming. When you declare a variable using a basic data type, C automatically allocates space for the variable in an area of memory called the stack.

An int variable, for example, is typically allocated 4 bytes when declared. We know this by using the sizeof operator:

```
int x;
printf("%d", sizeof(x)); /* output: 4 */
```

Memory Management Functions

The stdlib.h library includes memory management functions.
The statement #include <stdlib.h> at the top of your program gives you access to the following:

malloc(*bytes*) Returns a pointer to a contiguous block of memory that is of size *bytes*.

calloc(*num_items, item_size*) Returns a pointer to a contiguous block of memory that has *num_items* items, each of size *item_size* bytes. Typically used for arrays, structures, and other derived data types. The allocated memory is initialized to 0.

realloc(*ptr, bytes*) Resizes the memory pointed to by ptr to size bytes. The newly allocated memory is not initialized.

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

The malloc Function

The malloc() function allocates a specified number of contiguous bytes in memory.
For example:

```
#include <stdlib.h>

int *ptr;
/* a block of 10 ints */
ptr = malloc(10 * sizeof(*ptr));

if (ptr != NULL) {
 *(ptr + 2) = 50;  /* assign 50 to third int */
}
```

The malloc Function

The allocated 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. You are advised to use + to refer to array elements. Using ++ or += changes the address stored by the pointer.

If the allocation is unsuccessful, NULL is returned. Because of this, you should include code to check for a NULL pointer.

The calloc Function

The calloc() function allocates memory based on the size of a specific item, such as a structure.
The program below uses calloc to allocate memory for a structure and malloc to allocate memory for the string within the structure:

```
typedef struct {
 int num;
 char *info;
} record;

record *recs;
int num_recs = 2;
int k;
char str[ ] = "This is information";

recs = calloc(num_recs, sizeof(record));
if (recs != NULL) {
 for (k = 0; k < num_recs; k++) {
   (recs+k)->num = k;
   (recs+k)->info = malloc(sizeof(str));
   strcpy((recs+k)->info, str);
 }
}
```

The realloc Function

The realloc() function expands a current block to include additional memory.
For example:

```
int *ptr;
ptr = malloc(10 * sizeof(*ptr));
if (ptr != NULL) {
 *(ptr + 2) = 50;  /* assign 50 to third int */
}
ptr = realloc(ptr, 100 * sizeof(*ptr));
*(ptr + 30) = 75;
```

Allocating Memory for Strings

When allocating memory for a string pointer, you may want to use string length rather than the sizeof operator for calculating bytes.
Consider the following program:

```
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.
Because elements are not allocated all at once, dynamic arrays typically use a structure to keep track of current array size, current capacity, and a pointer to the elements, as in the following program.

```
typedef struct {
 int *elements;
 int size;
 int cap;
} dyn_array;

dyn_array arr;

/* initialize array */
arr.size = 0;
arr.elements = calloc(1, sizeof(*arr.elements) );
arr.cap = 1;  /* room for 1 element */
```