UNIT-4 (STRUCTURES) (RAM GOPAL GUPTA- http://ramgopalgupta.com/)

PART-2

Complete Simple Code example1:

```
//Program to accept a student record from the user and display it using structure.
#include <stdio.h>
#include <string.h>
Step 1-/* Created a structure here. The name of the structure is StudentData. */
struct StudentData{
    char stu_name[50];
    int stu_roll;
    int stu_age;
};

int main()
{
Step 2-/* stud is the variable of structure StudentData*/
     struct StudentData stud;

Step 3-/*Accepting the values of each struct member here*/
     printf("Enter Student Name: ");
     gets(stud.stu_name);                    //scanf("%s", & stud.stu_name);
     printf("Enter Student Roll Number: ");
     scanf("%d", & stud.stu_roll);
     printf("Enter Student Age: ");
     scanf("%d", & stud.stu_age);

Step 4-/* Displaying the values of struct members */
     printf("Student Name is: %s", stud.stu_name);
     printf("\nStudent Id is: %d", stud.stu_roll);
     printf("\nStudent Age is: %d", stud.stu_age);
     return 0;
}
```

Explanation:

If you observe the above program:

Step 1-In this step a structure "**StudentData**" has been created with three members' stu_name, stu_roll, stu_age for student name, student roll number and student age respectively.

Step 2-In this step a structure variable "**stud**" has been declared.

Step 3-In this step we are accepting student name, roll number and age from the user. Here we used gets(....) function to accept student name from the user. gets(.....) function can accept a string with space and will store into the string variable passed into parameter.

In this case statement ***gets(stud.stu_name);*** will accept a string i.e. student name from the user with spaces (if have) and will store in **stu_name** member of the structure **StudentData**.

Step 4-It written to display the values stored into the structure members.

Complete Simple Code example2:

//Program to accept 5 student records from the user and display it using structure.

```
#include <stdio.h>

#include <string.h>

Step 1-/* Created a structure here. The name of the structure is StudentData. */

struct StudentData{

   char stu_name[50];

   int stu_roll;

   int stu_age;

};

int main(){

Step 2-/* student is the variable of structure StudentData*/

    struct StudentData stud[5];

    int i;

Step 3-/*Accepting the 5 student records with values of each struct member here*/

    for (i=0; i<5; i++){

    printf("Enter Student Name: ");

    gets(stud[i].stu_name);//scanf("%s", &stud.stu_name);

    fflush(stdin);

    printf("Enter Student Roll Number: ");

    scanf("%d", &stud[i].stu_roll);

    fflush(stdin);

    printf("Enter Student Age: ");

    scanf("%d", &stud[i].stu_age);

    fflush(stdin);

          }

Step 4-/* Displaying the 5 student records with values of each struct members */

    for (i=0; i<5; i++){

    printf("\nStudent Name is: %s", stud[i].stu_name);

    printf("\nStudent Roll is: %d", stud[i].stu_roll);

    printf("\nStudent Age is: %d", stud[i].stu_age);

          }

    return 0;

}
```

Explanation:

The above program is written to accept the records of 5 students therefore we have done some changes in each step.

Step 1-Explanation is same as in previous program.

Step 2-In this step an array of structure **StudentData** has been declared with size 5 "**stud[5]**" as we want to accept the record of 5 students.

struct **StudentData stud[5];**

stud[0]	**stud[1]**	**stud[2]**	**stud[3]**	**stud[4]**
stu_name stu_roll stu_age	stu_name stu_roll stu_age	stu_name stu_roll stu_age	stu_name stu_roll stu_age	stu_name stu_roll stu_age

Step 3-In this step we are accepting record of 5 students including student name, roll number and age from the user; therefore for loop is used "**for (i=0; i<5; i++)**". Explanation of "***gets(stud.stu_name);***" is same as in previous program. I have used "**fflush(stdin)**" function to flush the buffer memory after accepting the input from the user so that next input can be stored into the memory. You do one thing remove the "fflush(stdin)" from all three place in Step-3 and see the result then you can understand its purpose.

Each record of student is stored in separate subscript of **stud** array.

Step 4-It written to display the record of 5 students which are stored from **stud[0]..**to ..**stud[4].**

NESTED STRUCTURE:

A structure is called a nested structure when one of the members of the structure is itself a structure. ***For instance:***

```
struct StEx{
        int i;
        char ch;
        struct StNested svar;
};
```

In this example we have declared a "**svar"** nested structure of type "***StNested***" as one of the member of structure "**StEx**".

Complete code Example:

```
Line 1-    #include<stdio.h>
Line 2-    struct address                 // defined a structure
Line 3-    {
Line 4-    char house_no[20];
Line 5-    char street_nm[50];
Line 6-    char city[20];
Line 7-    };
Line 8-    struct employee                // defined a structure
Line 9-    {
Line 10-   char name[20];
Line 11-   struct address add;   //declared a nested structure as a member of another structure
Line 12-   };
Line 13-   void main ()
Line 14-   {
Line 15-   struct employee emp;
Line 16-   printf("Enter employee information?\n");
Line 17-   scanf("%s %s %s %s",emp.name, emp.add.house_no, emp.add.street_nm, emp.add.city);
Line 18-   printf("Printing the employee information....\n");
Line 19-   printf("Name:%s\nHouse No:%s\nStreet Name: %s\nCity: %s", emp.name, emp.add. house_no, emp.add.street_nm, emp.add.city);
Line 20-   }
```

Output:

```
Enter employee information?
RGG
23A
Sunderpur
Varanasi

Printing the employee information....
Name: RGG
House No: 23A
Street Name: Sunderpur
City: Varanasi
```

Explanation:

In the above program

Nested structure variable **add** at **Line 11** has been declared inside another structure named **employee**.

Then nested structure variable **add** is used in Line 17 & 19 to access the members (house_no, street_nm, city) of nested structure **address**. Rest code working is same in previous code examples.

UNION

A union is block of memory that is used to hold data items of different types. In C, a union is similar to a structure, except that data items saved in the union are overlaid in order to share the same memory location.

Or

A union is a special data type available in C that allows to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple-purpose.

Declaration:

To define a union, we must use the union statement in the same way as we did while defining a structure. The union statement defines a new data type with more than one member for our program.

```
union Data {
   int i;
   float f;
   char str[20];
};
```

Now, a variable of **Data** can store an integer, a floating-point number, or a string of characters. It means a single variable, i.e., same memory location, can be used to store multiple types of data. We can use any built-in or user defined data types inside a union based on our requirement.

The memory occupied by a union will be large enough to hold the largest member of the union. For example, in the above example, Data type will occupy ***20 bytes*** of memory space because this is the maximum space which can be occupied by a character string. The following example displays the total memory size occupied by the above union.

```
#include <stdio.h>
#include <string.h>
 union Data {
   int i;
   float f;
   char str[20];
};

int main( ) {
   union Data data;
   printf( "Memory size occupied by data : %d\n", sizeof(data));
   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Memory size occupied by data : 20
```

Explanation:

sizeof (data) = sizeof (union member which need largest memory)

sizeof (data) = sizeof (str)

sizeof (data) = 20 bytes

Same program if run with structure instead of union, see the effect:

```
#include <stdio.h>
#include <string.h>
 struct Data {
   int i;
   float f;
   char str[20];
};

int main( ) {
   struct Data data;
   printf( "Memory size occupied by data : %d\n", sizeof(data));
   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Memory size occupied by data : 26
```

Explanation: in case of structure the memory occupied by a structure will be sum of memory storage capacity of each data type variable declared inside the structure. As in our case

sizeof (data) = sizeof(int) + sizeof(f) + sizeof(str)

sizeof (data) = 2 + 4 + 20

sizeof (data) = 26 bytes

Accessing Union Members:

To access any member of a union, we use the member access operator (.). The member access operator is coded as a period between the union variable name and the union member that we wish to access.

The following example shows how to use unions in a program –

```
#include <stdio.h>
#include <string.h>
 union Data {
   int i;
   float f;
   char str[20];
};
 int main( ) {
   union Data data;
   data.i = 10;
   data.f = 220.5;
   strcpy( data.str, "C Programming");

   printf( "data.i : %d\n", data.i);
   printf( "data.f : %f\n", data.f);
   printf( "data.str : %s\n", data.str);
   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
data.i : 1917853763
data.f : 4122360580327794860452759994368.000000
data.str : C Programming
```

Important note: Here, we can see that the values of i and f members of union got corrupted because the final value assigned to the variable has occupied the memory location and this is the reason that the value of str member is getting printed very well.

Now let's look into the same example once again where we will use one variable at a time which is the main purpose of having unions –

```
#include <stdio.h>
#include <string.h>

union Data {
   int i;
   float f;
   char str[20];
};

int main( ) {
   union Data data;
   data.i = 10;
   printf( "data.i : %d\n", data.i);

   data.f = 220.5;
   printf( "data.f : %f\n", data.f);

   strcpy( data.str, "C Programming");
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
data.i : 10

data.f : 220.500000

data.str : C Programming
```

Here, all the members are getting printed very well because one member is being used at a time.