CL-101

INTRODUCTION TO COMPUTING

LAB – 13

STRUCTURES in C

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

INTRODUCTION

We studied earlier that array is a data structure whose element are all of the same data type. Now we are going towards structure, which is a data structure whose individual elements can differ in type. Thus a single structure might contain integer elements, floating– point elements and character elements. Pointers, arrays and other structures can also be included as elements within a structure. The individual structure elements are referred to as members. This lesson is concerned with the use of structure within a 'c' program. We will see how structures are defined, and how their individual members are accessed and processed within a program. The relationship between structures and pointers, arrays and functions will also be examined. Closely associated with the structure is the union, which also contains multiple members.

OBJECTIVES

After going through this lesson you will be able to

- explain the basic concepts of structure
- process a structure
- use **typedef** statement
- explain the between structures and pointers
- relate structure to a function

STRUCTURE

In general terms, the composition of a structure may be defined as

```
struct tag
{ member 1;
member 2;
------------
-------------
member m; }
```

In this declaration, struct is a required key-word; tag is a name that identifies structures of this type. The individual members can be ordinary variables, pointers, arrays or other structures. The member names within a particular structure must be distinct from one another, though a member name can be same as the name of a variable defined outside of the structure.
A storage class, however, cannot be assigned to an individual member, and individual members cannot be initialized within a structure-type declaration.

For example:

```
struct student
{
char name [80];
int roll_no;
float marks;
};
```

We can now declare the structure variable s1 and s2 as follows:

```
struct student s1, s2;
```

s1 and s2 are structure type variables whose composition is identified by the tag student.
It is possible to combine the declaration of the structure composition with that of the structure variable as shown below.

```
storage- class struct tag
{
member 1;
member 2;
- ——
- —-
- member m;
} variable 1, variable 2 --------- variable n;
```

The tag is optional in this situation.

```
struct student {
char name [80];
int roll_no;
float marks;
} s1, s2;
```

The s1, s2, are structure variables of type student. Since the variable declarations are now combined with the declaration of the structure type, the tag need not be included. As a result, the above declaration can also be written as

```
struct {
char name [80];
int roll_no;
float marks ;
} s1, s2;
```

A structure may be defined as a member of another structure. In such situations, the declaration of the embedded structure must appear before the declaration of the outer structure. The members of a structure variable can be assigned initial values in much the same manner as the elements of an array. The initial values must appear in the order in which they will be assigned to their corresponding structure members, enclosed in braces and separated by commas. The general form is

```
storage-class struct tag variable = { value1, value 2,-------, value m};
```

A structure variable, like an array can be initialized only if its storage class is either external or static. e.g. suppose there are one more structure other than student.

```
struct dob
{ int month;
int day;
int year;
};

struct student
{ char name [80];
int roll_no;
float marks;
struct dob d1;
};

static struct student st = { “param”, 2, 99.9, 17, 11, 01};
```

It is also possible to define an array of structure, that is an array in which each element is a structure. The procedure is shown in the following example:

```
struct student{
char name [80];
int roll_no ;
float marks ;
} st [100];
```

In this declaration st is a 100- element array of structures. It means each element of st represents an individual student record.

PROCESSING A STRUCTURE

The members of a structure are usually processed individually, as separate entities. Therefore, we must be able to access the individual structure members. A structure member can be accessed by writing

variable.member name

This period (.) is an operator, it is a member of the highest precedence group, and its associativity is left-to-right.
E.g. if we want to print the detail of a member of a structure then we can write as printf("%s",st.name); or printf("%d", st.roll_no) and so on. More complex expressions involving the repeated use of the period operator may also be written. For example, if a structure member is itself a structure, then a member of the embedded structure can be accessed by writing:

variable.member.submember

Thus in the case of student and dob structure, to access the month of date of birth of a student, we would write

st.d1.month

The use of the period operator can be extended to arrays of structure, by writing

array [expression].member

Structures members can be processed in the same manner as ordinary variables of the same data type. Single-valued structure members can appear in expressions. They can be passed to functions and they can be returned from functions, as though they were ordinary single-valued variables.
E.g. suppose that s1 and s2 are structure variables having the same composition as described earlier. It is possible to copy the values of s1 to s2 simply by writing

s2=s1;

It is also possible to pass entire structure to and from functions though the way this is done varies from one version of 'C' to another. Let us consider an example of structure:

```
#include <stdio.h>
struct date {
int month;
int day;
int year;
};
struct student{
char name[80];
char address[80];
int roll_no;
char grade;
float marks;
struct date d1;
}st[100];
main()
{
int i,n;
void readinput (int i);
void writeoutput(int i);
printf("Student system");
printf("How many students are there ?");
scanf("%d" &n);
for (i=0; i<n; ++i){
readinput (i);
if( st[i].marks <80)
st[i].grade='A';
else
st[i].grade='A'+;
}
for (i=0; i<n; ++i)
writeoutput(i);
}
void readinput (int i)
{
printf("\n student no % \n", i+1);
printf("Name:");
scanf("%[^\n]", st[i].name);
printf("Address:");
scanf("%[^\n]", st[i].address);
printf("Roll number");
scanf("%d", &st[i].roll_no);
printf("marks");
scanf("%f",&st[i].marks);
printf("Date of Birth {mm/dd/yyyy)");
scanf("%d%d%d", & st[i].d1.month & st[i].d1.day, & st[i].d1.year);
return;
}
void writeoutput(int i)
{
printf("\n Name:%s",st[i].name);
printf("Address %s\n", st[i].address);
printf("Marks % f \n", st[i].marks);
printf("Roll number %d\n", st[i].roll_no);
printf("Grade %c\n",st[i].grade);
return;
}
```

It is sometimes useful to determine the number of bytes required by an array or a structure. This information can be obtained through the use of the **sizeof** operator.

USER-DEFINED DATA TYPES (Typedef)

The typedef feature allows users to define new data types that are equivalent to existing data types. Once a user-defined data type has been established, then new variables, arrays, structure and so on, can be declared in terms of this new data type. In general terms, a new data type is defined as

typedef type new type;

Where type refers to an existing data type and new-type refers to the new user-defined data type.

e.g. typedef int age;

In this declaration, age is user- defined data type equivalent to type int. Hence, the variable declaration

age male, female;
is equivalent to writing
int age, male, female;

The typedef feature is particularly convenient when defining structures, since it eliminates the need to repeatedly write struct tag whenever a structure is referenced. As a result, the structure can be referenced more concisely. In general terms, a user-defined structure type can be written as

```
typedef struct
{ member 1;
member 2:
- - - -
- - - -
member m;
} new-type;
```

The typedef feature can be used repeatedly, to define one data type in terms of other user-defined data types.

STRUCTURES AND POINTERS

The beginning address of a structure can be accessed in the same manner as any other address, through the use of the address (&) operator. Thus, if variable represents a structure type variable, then & variable represents the starting address of that variable. We can declare a pointer variable for a structure by writing

```
type *ptr;
```

Where type is a data type that identifies the composition of the structure, and ptr represents the name of the pointer variable. We can then assign the beginning address of a structure variable to this pointer by writing

```
ptr= &variable;
```

Let us take the following example:

```
typedef struct {
char name [ 40];
int roll_no;
float marks;
}student;
student s1,*ps;
```

In this example, s1 is a structure variable of type student, and ps is a pointer variable whose object is a structure variable of type student. Thus, the beginning address of s1 can be assigned to ps by writing.

```
ps = &s1;
```

An individual structure member can be accessed in terms of its corresponding pointer variable by writing ptr -> member

Where ptr refers to a structure- type pointer variable and the operator -> is comparable to the period (.) operator. The associativity of this operator is also left-to-right.
The operator -> can be combined with the period operator (.) to access a submember within a structure. Hence, a submember can be accessed by writing

ptr -> member.submember

PASSING STRUCTURES TO A FUNCTION

There are several different ways to pass structure–type information to or from a function. Structure member can be transferred individually, or entire structure can be transferred. The individual structures members can be passed to a function as arguments in the function call; and a single structure member can be returned via the return statement. To do so, each structure member is treated the same way as an ordinary, single- valued variable.
A complete structure can be transferred to a function by passing a structure type pointer as an argument. It should be understood that a structure passed in this manner will be passed by reference rather than by value. So, if any of the structure members are altered within the function, the alterations will be recognized outside the function. Let us consider the following example:

```
# include <stdio.h>
typedef struct{
char *name;
int roll_no;
float marks ;
} record ;
main ( )
{
void adj(record *ptr);
static record stduent={“Param”, 2,99.9};
printf(“%s%d%f\n”, student.name, student.roll_no, student.marks);
adj(&student);
printf(“%s%d%f\n”, student.name, student.roll_no, student.marks);
}
void adj(record*ptr)
{
ptr -> name= “Ali”;
ptr -> roll_no=4;
ptr -> marks=95.0;
return;
}
```

Let us consider an example of transferring a complete structure, rather than a structure-type pointer, to the function.

```
# include <stdio.h>
typedef struct{
char *name;
int roll_no;
float marks;
}record;
main()
{
void adj(record stduent); /* function declaration */
static record student={“Param,” 2,99.9};
printf(“%s%d%f\n”, student.name,student.roll_no,student.marks);
adj(student);
printf(“%s%d%f\n”, student.name,student.roll_no,student.marks);
}
void adj(record stud) /*function definition */
{
stud.name= “Ali”;
stud.roll_no=5;
stud.marks=96.0;
return;
}
```