Chapter 5 Structures

In this chapter you will learn all about:

Structures - definition and declaration

Operations with structures

Arrays of structures

Passing members of the structures to functions

Annexure - Command Line arguments processing

==========

A.1 Structures - Definition

Object containing a set of members; Members are of varying data types

General form:

```
struct name {  /* name is the name of this structure */
    <members here>
};  /* observe the semi-colon */
```

Various forms:

```
struct name { <members> };
struct { <members> }x,y,z;
```

Example 6.1 Sample structure definitions

```
struct sample1 {
        int tel_no;
        float salary;
        char name[10];
};
Alternatively,
struct {
        int tel_no;
        float salary;
        char name[10];
} sample1;
```

How do I declare more structures of type sample1?

(a) struct sample1 JoYew, Francis, John;

(b) Alternatively,

```
struct {
        int tel_no;
        float salary;
        char name[10];
} JoYew, Francis, John;
```

The above means JoYew, Francis and John are structures of type "sample1" defined above. Each of the structures has 3 arrays as indicated;

Example 6.2 Nested structure representations

```
struct Temployee {
        struct {
            char surname[15];
            char other[15];
        } name; /* the "name" is a structure */
        struct {
            char house_no[6];
            char block_no[6];
            char street[30];
            char postcode[7];
        } address;  /* address is a structure */
        char tel_no[9];
        char ic_no[9];
        char birthdate[9];
        int age;
        float salary;
};
```

In the above definition of a structure, we observe that there are 2 structures in turn within structure "T_employee". The first structure describes one attribute "name" and the second one describes "address". The rest are all arrays.

Usage:

struct Temployee pers_emp, engg_emp;

Interpretation:

pers_emp and engg_emp are structures of type "Temployee".

==========

/* Following form is referred to as user-defined data type */

typedef operator

typedef **struct** {

int month;

int year;

int day;

} **date**;

date lastpayment, currentpay, lastyearpayment; /* convenient to write */

The typedef is a casting operator that treats "date" structure as a specific data-type (like int, char, float) and hence can be used to define structures of type "date" as shown above.

typedef int age;

We can use then "age" equivalently as: ***age male, female;*** instead of int male, female;

typedef float height[100];

height MaleHeight, FemaleHeight; /* defines two 100 element arrays of float type */

Where to define the structure in the program?

- Look at our examples provided. Outside main() function a structure is defined and then inside the main() function we can declare the required number of instances of that defined structure.

A.2. Natural Binding

Example 6.3 – Employee Table – Name, age, salary information. Each employee's entry forms one record.

(1) Using arrays for each of these members for an employee is a possible solution; However, when we sort w.r.t say, age, then only the age information gets sorted and overall output could be a wrong record when one reads as a record!

Smith 44 9000

Tom 32 4000

Amanda 30 3500

If the above is the data to be entered and if we use arrays then when we sort w.r.t age the output will be:

Smith 30 9000

Tom 32 4000

Amanda 44 3500

Which is completely wrong as far as a record is concerned. This happens because there is not "natural binding" between the members!

Structure solves this problem.

```
struct employee {
        char name[10];
        int age;
        float salary;
};
```

Using the structure definition above for a employee record creation, if we sort using age member then the output would be:

Amanda	30	3500
Tom	32	4000
Smith	44	9000

which is the correct expected output. This happens only with the use of structure and due to the binding process.

B.1 Operations with structures

(i) Access a member; (ii) Change a member's value;
(iii) Obtain the address of a structure (*next chapter, stay tuned!)*

How to access a member in a structure?

A member is accessed by:

structure-name**.**member [structure name DOT member]

Example 6.4 (using employee structure):

employee**.**age

employee.salary

employee**.**name[5]

The members are variables and hence they can be used in computation as normal variables; See the following examples;

- Customer**.**lastpay.month = 10;
- if(Customer.balance – payment >0) **...**
- printf("Account number is: %d\n", customer**.**account_no);

Assigning the information of one structure to another identical structure by assignment operator is allowed in most new versions of C; Older versions demand copying every variable of old structure to the new structure explicitly;

- newCustomer = oldCustomer; /* RHS is a structure and LHS is of identical structure type as that of RHS */
- ++Customer.balance;

B. 2 Arrays of structures

Example 6.5 - Arrays of structures can be defined as follows.

```
struct book {
        char title[40];  /* book has this title array */
        char author[40]; /* book has this author array */
        float cost;
        struct {
            char publisher[20];
            char ISBN[25];
            int Edition;
            } pubinfo;  /* this is a structure */
    };
struct book mylibbook[100];  /* This would declare an array of 100 structures (called mylibbook) each of which is a structure of type book */
```

B.3 Accessing members of an array of structures

Some examples are as follows.

```
strcpy(mylibbook[5].title, "my dear ghost!");
strcpy(mylibbook[5].author, "Jonathan Smith");
mylibbook[5].cost = 23.45;
mylibbook[20].pubinfo.Edition = 1997;
if(mylibbook[j].pubinfo.Edition <1995)
        printf("No Stocks available\n");
strcmp("McGrawHill", mylibbook[20].pubinfo.publisher);
```

B.4 Initialization

Structures and arrays of structures can be initialized in the same manner as other data types

Example 6.6:

```
struct book {
    char title[40];
    char author[40];
    float cost;
};

struct book mylib[] = {
    { "C Programming Language", "Kernighan", 45.0 },
    { "C Primer Plus", "Waite", 20.0 },
    { "10,000 Ways To Get Rich", "DieHard", 1.0}
};  /* fixed and pre-assigned data */
```

B.4 Passing Structures to-and-from Functions

Three methods available –

(i) Members of structures can be passed individually (*after all they are variables!*)

(ii) Entire structure can be passed – use the **struct** as a data type;

(iii) Entire structure can be passed – using pointers(address of the structure)!

Example 6.7: *Follow this example carefully – Passing certain members to functions for processing (Identical to passing by value!)*

```
float adjust(char name[], int Acc_no, float Balance); /* function
prototype */
void main() {
        typedef struct {
                int month;
                int year;
                int day;
                } date; /* date structure */
        struct {
                int accnt_no;
                char accnt_type:
                char name[80];
                float balance;
                date lastpayment;
                } customer;
        . . . . . .
        customer.balance = adjust(customer.name,  customer.accnt_no,
                                  customer.balance);
        /* observe how we pass the values pertaining to a single customer */
        . . . . .
        } /* end of main  */
/*------------------------ */
float adjust(char name[], int Accnt_no, float Balance)  /*  func
definition here */
{
        float newbalance, currentDep; /* local variable */
        . . . .
        newbalance = Balance + currentDep;
        . . . . . .
        return(newbalance);
}
```

NOTE: Method by which entire structure can be passed is via pointers(out of scope of this module).

Example 6.8: *Follow this example carefully – Passing the entire structure using* ***struct*** *data type.*

```
#include<stdio.h>
 typedef struct data{
        int x;
        float y;
};

struct data mydata[10]; /* array of structures */

void myfunc(struct data B[]); /* func prototype */

int main() {
  int i;
  myfunc(mydata);  /* Passing the array of structures! - just like array. No difference!*/
  for(i=0;i<10;i++)
      printf("Data: mydata[%d].x = %d,mydata[%d].y = %f\n",i, mydata[i].x,i,mydata[i].y);
  return 0;
}
/* ==========  */
void myfunc(struct data B[]){
  int j;
  for(j=0;j<10;j++){
    mydata[j].x = j;
    mydata[j].y = j+1.0;
    printf("From myfunc: mydata[%d].x = %d,mydata[%d].y = %f\n",i,
        mydata[i].x,i,mydata[i].y);
  }
  printf("\n");
  return;
}
```

NOTE: Third method by which entire structure can be passed is via pointers (out of scope of this module).

Tutorial Q 6.1: *Revise all the demonstrated examples in the lecture. Codes are provided.*

Annexure

Interacting via Command Line Parameters

Until now we have been using the main() without any arguments. Actually, main can accommodate two arguments **argc** and **argv**. The first one is an integer variable and the second one is an array of strings (pointers to characters!!). Each string in this array will represent a parameter that is passed to the main function. The value of argc denotes the number of parameters passed.

Definition:

```
void main(int argc, char *argv[]) {
…
}
```

A program is executed usually by specifying its name (filename.exe). In order to pass the parameters to the program from the command line, we have:

Programme-name parameter1 parameter2 parameter 3 . . parameter n
/ note that there is no comma between the paramters */*

Program name will be stored as the first item in **argv** array followed by each parameter. Note that the value of **argc** is not supplied explicitly from the command line.

Example 6.9: Consider the following example of reading a set of parameters from the command line.

```
#include <stdio.h>
main(int argc, char *argv[]) {
        int count;
        printf('argc = %d\n",argc);
        for(count=0; count < argc; ++count)
                printf("argv[%d] = %s\n", count, argv[count]);
}
```

What does this program do? Allows unspecified number of parameters to be passed to the main function for processing via the command line. If the program name is say "sample" and the command line initiating the program execution is:

sample red white blue

Then the program will be executed resulting in the following output.

argc = 4 argv[0]=sample.exe argv[1]=red argv[2]=white argv[3]=blue

The output tells us that 4 separate items have been entered from the command line. First is the programme name followed by three parameters. Each item is an element in the array argv. Suppose we enter as,

sample red "white blue"

then the output will be: argc = 3argv[0]=sample.exe argv[1]=red argv[2]=white blue