CHARUTAR VIDYA MANDAL'S
SEMCOM
Vallabh Vidyanagar

Faculty Name: Ami D. Trivedi
Class: FYBCA
Subject: US02CBCA01
(Advanced C Programming and Introduction to Data Structures)

*UNIT – 2 (Structures, Unions and File Handling)

*STRUCTURES

A structure is a data storage method designed by the programmer, to suit your programming needs exactly. Structures are useful whenever information of different variable types needs to be treated as a group. Array can not be used in this situation.

*SIMPLE STRUCTURES

Definition:
A structure is a convenient tool for handling a group of logically related data items.

OR

A structure is a collection of one or more variables grouped under a single name for easy manipulation.

Example of structure:

student	:	rollno, name, total, percentage
book	:	author, title, price, pages
employee	:	name, address, city, basic_salary, designation

DEFINING A STRUCTURE

Format of structures must be defined first that will be used in future to declare structure variables. The general format of structure definition is as follows:

```
struct tag_name
{
   data_type member1;
   data_type member2;
   -----------------------
   -----------------------
} ;
```

The struct keyword identifies the beginning of a structure definition. It must be followed immediately by the structure name, or tag (which follows the same rules as other C variable names).

Structure name is followed by pair of braces which contains list of the structure's member variables. You must give a data type and name for each member.

The variables in a structure can be of different variable types. Structures also can hold arrays, pointers, and other structures. Each variable / data field within a structure is called a **member of the structure OR structure elements**.

```
struct emp
{
   int eno;
   char enm[20];
   float salary;
};
```

This will define a structure to hold details of 3 data filed: employee number, name and salary. These fields are called members. emp is name of structure and it is called **tag or structure tag**.

Here we have defined the structure without the instance. It describes a format called **template** that can be used later in a program to declare structures.

Above template emp do not create any instances of the structure emp. In other words, they don't declare any structures. Structure variables are known as an **instance**.

DECLARING STRUCTURE VARIABLES

After defining a structure format (template), we can declare variable (instance) of that type.

There are 2 ways to declare structure variables.

1. Declaration with the definition:

```
struct emp
    {
       int eno;
       char enm[20];
       float salary;
    } e1,e2;
```

These statements define the structure type emp and declare two structures, e1 and e2 of type emp. e1 and e2 are instances of type emp. e1 contains integer variable eno, character array enm and float variable salary, and e2 also contains integer variable eno, character array enm and float variable salary.

2. Declare structure variables at a different location from the definition.

```
struct emp                 | void main()
    {                      | {
        int eno;           |    struct emp e1, e2;
        char enm[20];      |    ----------------------
        float salary;      | }
    };                     |
```

Use of tag name is optional.

```
struct
    {
        int eno;
        char enm[20];
        float salary;
    }e1,e2;
```

will declare 2 structure variable e1 and e2 but there is no tag name. This is not recommended because

1. Without tag name, we can not use this template in future to declare variables of this struct type.
2. Generally, structure definition comes at beginning before main. Here definition is global and without tag we are forced to declare variables with definition. So they also will become global and used by any other user defined function.

Note:

1. Members of structure are not variables.
2. They do not occupy any memory until they are linked with structure variable. When compiler comes to declaration statement then only it reserves memory space for structure variables.
3. Size of structure variable will be sum of size of all members of that structure.

ACCESSING STRUCTURE MEMBERS

Individual structure members can not be accessed like other variables because they are not variable. They should be linked with structure variable to make them meaningful member. For e.g. eno has no meaning but eno of e1 has a meaning.

Link between member and variable is created by **member operator (.)**, also called the **dot operator.** So, structure members are accessed using dot operator between the structure name and the member name.

Thus, for structure e1 refer to a first employee, you could write

```
e1.eno = 100;
strcpy(e1.enm,"Anuj");
e1.salary=5000;
```

To display the employee details stored in the structure e1, you could write

```
printf("%d %s %f", e1.eno, e1.enm, e1.salary);
```

INITIALIZING STRUCTURES

Like other C variable types, structures can be initialized when they're declared. After a structure is declared, its members must be initialized. Otherwise, they will contain “garbage”.

C does not allow initialization of individual structure members within the template. The initialization must be done only in declaration of actual structure variables.

A structure can be initialized at either of the following stages:

1. At compile time (at time of declaration of a structure)
2. At run time (input form user and with assignment statement)

1. Compile time initialization (at time of declaration of a structure)

We can initialize the members of a structure when they are declared.

Initializing structure which is declared with definition

Structure declaration is followed by an equal sign and a list of initialization values separated by commas and enclosed in braces. For example,

```
struct sale
{
    char customer[20];
    char item[20];
    float amount;
} mysale = { "Acme Industries","Table",1000.00};
```

When these statements are executed, they perform the following actions:

1. Define a structure type named sale
2. Declare an instance of structure type sale named mysale
3. Initialize the structure member mysale.customer to the string "Acme Industries"
4. Initialize the structure member mysale.item to the string "Table"
5. Initialize the structure member mysale.amount to the value 1000.00

Initializing structure which is declared at different location from the definition

```
struct sale
{
    char customer[20];
    char item[20];
    float amount;
};
void main()
{
   struct sale mysale = { "Acme Industries","Table",1000.00};
          ----------------------
}
```

Note:

1. The order of values enclosed in braces must match the order of members in structure definition.
2. Partial initialization is allowed. We can initialize first few members and leave the remaining. Uninitialized members should be at the end of list only.
3. Uninitialized members will be assigned default value as: 0 for integers and floats and '\0' for character and strings.

<u>2. Run time initialization (input form user and with assignment statement)</u>

Structure can be initialized at run time. Run time initialization can be done by:

(1) Taking input from user or
(2) By assignment statement

<u>(1) Taking input from user</u>

We can also initialize structure by taking input from user. This concept is known as run time initialization. E.g.

```
struct sale                 void main()
{                           {
    char customer[20];              struct sale mysale;
    char item[20];                  printf("Enter customer : ");
    float amount;                   gets(mysale.customer);
};                                  printf("Enter item : ");
                                    gets(mysale.item);
                                    printf("Enter amount : ");
                                    scanf("%f",&mysale.customer);
                            }
```

(2) By assignment statement

Members of structure can be initialized individually also. E.g.

```
struct sale                 void main()
{                           {
    char customer[20];        struct sale mysale;
    char item[20];            strcpy (mysale.customer," Acme Industries");
    float amount;             strcpy (mysale.item,"Table");
};                            mysale.customer = 1000;
                            }
```

This approach is time consuming. It will increase the length of program. It will also make program static. But it can be used to change value as per the requirement.

COPYING AND COMPARING STRUCTURE VARIABLES

One major advantage of structure is that you can copy information between structures of the same type with a simple equation statement. For example, the statement

e2 = e1;	is equivalent to this statement:	e2.eno = e1.eno;
		strcpy(e2.enm,e1.enm);
		e2.salary=e1.salary;

When your program uses complex structures with many members, this notation can be a great time-saver.

Statements like: e1 == e2 or e1 != e2 are not allowed for comparison between two structure variable. **C does not allow any logical operation on structure variable.** So we need to compare members individually.

*MORE-COMPLEX STRUCTURES

These are structures that contain other structures as members and structures that contain arrays as members.

NESTED STRUCTURE (STRUCTURES WITHIN STRUCTURES)

C structure can contain any of C's data types. For example, a structure can contain other structures. The previous example can be extended to illustrate this.

Assume that your employee salary program needs to deal with salary details. Salary is calculated as – allowances minus deductions. You need a structure to define salary.

```
struct emp
{
   int eno;
   char enm[20];
   struct
          { float allowance;
            float deduction;
          } sal;
}employ1;
```

This statement defines a structure of type emp that contains structure named sal. It also declares a structure variable employ1.

To access the members of inner structure, you must apply the member operator (.) twice. Thus, the expression **employ1.sal.allowance** refers to the allowance member of the sal member of the type emp structure named employ1.

An inner structure can have more than variable. For e,g. sal and salary. Now we can refer members of inner structure as **employ1.sal.allowance OR employ1.salary.allowance**

Listing 1. A demonstration of structures that contain other structures.

```
#include <stdio.h>
struct emp
{
   int eno;
   char enm[20];
   struct
           { float allowance;
             float deduction;
           }sal;
};

void main()
{
    struct emp employ1;
    float salary;
    printf("\nEnter employee number: ");
    scanf("%d", &employ1.eno);
    printf("\nEnter employee name: ");
    scanf("%s", employ1.enm);
    printf("\nEnter allowance: ");
    scanf("%f", &employ1.sal.allowance);
    printf("\nEnter deduction: ");
    scanf("%f", &employ1.sal.deduction);
    /* Calculate salary */
    salary = employ1.sal.allowance - employ1.sal.deduction;
    printf("\nSalary is %f \n", esalary);
}
```

```
Enter employee number: 100
Enter employee name: Anuj
Enter allowance: 1000
Enter deduction: 50
Salary is 950
```

STRUCTURES THAT CONTAIN ARRAYS (ARRAY WITHIN STRUCTURE)

C allows use of array as structure members. You can define a structure that contains one or more arrays as members. We can use single or multi dimensional arrays of type int, float or char etc. For e.g.

```
struct marks
{
    int number;
    float subject[3];
}student;
```

Here, the member subject contains 3 elements – subject[0], subject[1] and subject[2]. Thses elements can be accessed using appropriate subscripts.

For example, the statements

```
struct data{
   int  x[4];
   char y[10];
};
```

define a structure of type data that contains a four-element integer array member named x and a 10-element character array member named y. You can then declare a structure named record of type data as follows:
struct data record;

The organization of this structure is shown in Figure 1.

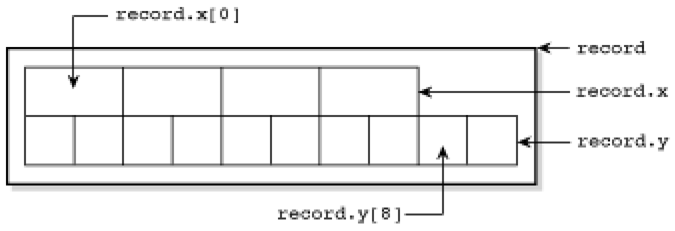

Figure 1: The organization of a structure that contains arrays as members.

You access individual elements of arrays that are structure members using a combination of the member operator and array subscripts:
record.x[2] = 100;
record.y[1] = `x';

You probably remember that character arrays are most frequently used to store strings. You can input in y using the statement
gets(record.y);

Now look at another example. Listing 2 uses a structure that contains a type float variable and two type char arrays.

Listing 2. A structure that contains array members.

```
#include <stdio.h>
struct stud
{
    int rollno, m[3], total;
    char sname[30]'
} ;
void main()
{
    struct stud  s1;
    printf("Enter student roll number : ");
    scanf("%d” ,s1.rollno);

    printf("Enter student name : ");
    gets(s1.sname);
    printf(“Enter marks of subject 1 : “);
```

```
#include <stdio.h>
struct stud
{
    int rollno, m[6], total;
    char sname[30]'
} ;
void main()
{
    struct stud  s1;
    int i;
    printf("Enter student roll number : ");
    scanf("%d” ,s1.rollno);
    printf("Enter student name : ");
    gets(s1.sname);
    s1.total=0;
    scanf("%d", &s1.m1);
    printf("Enter marks of subject 2 : ");
    scanf("%d", &s1.m2);
    printf("Enter marks of subject 3 : ");
    scanf("%d", &s1.m3);
    s1.total=s1.m1+s1.m2+s1.m3;
    printf("\nTotal mark = %d",s1.total);
}

Enter student roll number : 10
Enter student name : XYZ
Enter marks of subject 1 : 25
Enter marks of subject 2 : 30
Enter marks of subject 3 : 40
Total mark = 95
```

```
    for (i=0; i<9; i++)
    {
        printf("Enter marks of subject %d : ",i+1);
        scanf("%d", &s1.m[i]);
        s1.total=s1.total+s1.m[i];
    }
    printf("\nTotal mark = %d",s1.total);
}

Enter student roll number : 10
Enter student name : XYZ
Enter marks of subject 1 : 25
Enter marks of subject 2 : 30
Enter marks of subject 3 : 40
Enter marks of subject 4 : 50
Enter marks of subject 5 : 30
Enter marks of subject 6 : 40
Total mark =215
```

*ARRAYS OF STRUCTURES

We use structure to represent group of related variables. For example, we may use structure student to have members like student roll number, name and marks of 2 subjects. We need to declare one structure variable of type struct student to store data for one student.

But if we want to store such data for entire class then we need to declare many structure variables. In such cases, we can declare array of structures. Each element of array is a structure variable. Arrays of structures are very powerful programming tools.

For example,

```
struct student
{
   int rollno;
   char name[10];
   int m1,m2;
};

struct student stud[100];
```

defines an array called stud that contains 100 elements. Each element is defined to be of type struct student.

Listing 3. Arrays of structures.

```
#include <stdio.h>
/* Define a structure to hold entries. */
struct student
{
   int rollno;
   char name[10];
   int m1,m2;
};

void main()
{
    struct student stud[3];
    int i;
    for (i = 0; i < 3; i++)
        {
          printf("\nEnter roll number: ");
          scanf("%d", &stud[i].rno);
          printf("Enter student name: ");
          scanf("%s", stud[i].name);
          printf("Enter marks of subject 1 : ");
          scanf("%d", &stud[i].m1);
          printf("Enter marks of subject 2 : ");
          scanf("%d", &stud[i].m2);
        }
    printf("\n\n");
    /* Loop to display data. */
    for (i = 0; i < 3; i++)
      {
         printf("Roll number : %d", stud[i].rno);
         printf("Name : %s",stud[i].name);
         printf("Subject 1 marks : %d", stud[i].m1);
         printf("Subject 2 marks : %d", stud[i].m1);
      }
}
```

```
Enter roll number: 1
Enter student name: ABC
Enter marks of subject 1 : 10
Enter marks of subject 2 : 20
Enter roll number: 2
Enter student name: PQR
Enter marks of subject 1 : 30
Enter marks of subject 2 : 40
Enter roll number: 3
Enter student name: XYZ
Enter marks of subject 1 : 50
Enter marks of subject 2 : 60

Roll number : 1
Name : ABC
Subject 1 marks : 10
Subject 2 marks : 20
Roll number : 2
Name : PQR
Subject 1 marks : 30
Subject 2 marks : 40
Roll number : 3
Name : XYZ
Subject 1 marks : 50
Subject 2 marks : 60
```

*POINTERS TO STRUCTURES

We know that the name of an array stands for the address of its zeroth element. The same thing is true of the names of arrays of structure variables. Suppose product is an array variable of struct type. The name product represents the address of its zeroth element. Consider the following declaration:

```
struct inventory
{
    char name[30];
    int number;
    float price;
} product[2], *ptr;
```

This statement declares product as an array of two elements, each of the type struct inventory and ptr as a pointer to data objects of the type struct inventory. The assignment

```
ptr = product;
```

Would assign the address of the zeroth element of product to ptr. That is , the pointer ptr will now point to product[0]. Its members can be accessed using the following notation.

```
ptr -> name
ptr -> number
ptr -> price
```

The symbol -> is called the arrow operator (also known as member selection operator) and is made up of a minus sign and a greater that sign. Note that ptr -> is simply another way of writing product[0].

Would assign the address of the zeroth element of **product** to **ptr**. That is, the pointer **ptr** will now point to **product [0].** Its members can be accessed using the following notation.

```
ptr -> name
ptr ->number
ptr ->price
```

The symbol -> is called the arrow operator (also known as member selection operator) and is made up a minus sign and a greater then sign. Note that **ptr->** is simply another way of writing **product [0]**.

When the pointer **ptr** is incremented by one, it is made to point to the next recode, i.e., product [1]. The following **for** statement will print the values of members of all the elements of product array.

```
for (ptr = product; ptr < product + 2; ptr++)
printf (“%s %d %f\n”, ptr->name, ptr->number, ptr->price);
```

We could also use the notation

```
(*ptr) .number
```

to access the member number. The parentheses around ***ptr** are necessary because the member operator ‘.’ has a higher precedence than the operator*.

Listing 4. Write a program to illustrate the use of structure pointer.

A program to illustrate the use of a structure pointer to manipulate the elements of an array of structure is shown in following figure. Note that the pointer ptr (of type **struct invent**) can be used as the loop control index in **for** loop.

Program

```
struct invent
       {
          char *name[20];
          int number;
          float price;
        };
main ()
{
   struct invent product[3], *ptr;
   printf(“INPUT\n\n”);
   for (ptr = product; ptr < product+3; ptr++)
       {
          scanf(“%s” , ptr->name);
          scanf(“%d” , &ptr->number);
          scanf(“%f” , &ptr->price);
       }
    printf (“\n output \n\n”);
    ptr = product;
    while (ptr < product + 3)
          {
               printf(“%-20s”,ptr->name);
               printf(“%5d”,ptr->number);
               printf(“%10.2f \n”, ptr->price);
               ptr++;
           }
}
```

INPUT

```
WASHING_MACHINE  5   7500
ELECTRIC_IRON      12    350
TWO_IN_ONE           7    1250
```

OUTPUT

```
WASHING MACHINE 5  7500.00
EIECTRIC_IRON    12     350.00
TWO_IN_ONE          7   1250.00
```

POINTERS TO ARRAY OF STRUCTURE / POINTER TO STRUCTURE ARRAY

Name of array represents address of 0th element of the array. It is also true for array of structure. Suppose **s** is an array of type struct stud. Then **s** represents address of its 0th element.

```
struct stud
{
   int rollno, total;
   char sname[30];
   float per;
} s[50], *p;
```

Here s is an array of 50 elements. Each element is of type struct stud. And p is a pointer to data of type struct stud.

To assign address of 0th element address of structure array to pointer variable following method is used.
p=s;   OR p=&s[0];

Its members can be accessed using following notation:
p->total, p_m[i], p->total etc.

The symbol **->** is known as **arrow operator**. It is also known as **member selection operator**.

Note: We can also use the notation **(*p).rollno** to access the member rollno. The parentheses around *p are necessay because the the meber operator **"."** Has higher precedence than the operator **"*"**.

When pointer p is incremented by one, it will point to next element of structure array. i.e. s[1]. We can use for loop to access values of members of all elements of s array.

Example: Accessing successive array elements by incrementing a pointer.

```
#include <stdio.h>
struct stud
{
   int rollno, total;
   char sname[30];
} s[3] = { {1,100,"ABC"}, {2,200,"PQR"}, {3,300,"XYZ"} }, *p;

void main()
{
  /*Initialize the pointer to the first array element. */
  p=s;
  /* Loop through the array, incrementing the pointer with each iteration. */
  for (i=0 ; i < n ; i++)
      {
         printf("%d %s %d",p->rollno,p->sname,p->total);
         p++;
      }
```

```
Output:
1   ABC   100
2   PQR   200
3   XYZ    300
```

If you want to print all the array elements then use for loop. Here for loop will print one array element with each iteration of the loop.

To access the members using pointer notation, you must change the pointer p so that with each iteration of the loop it points to the next array element (that is, the next structure in the array).

C's pointer arithmetic comes to your help. The unary increment operator (++) has a special meaning when applied to a pointer: It means "increment the pointer by the size of the object it points to."

So, if you have a pointer p that points to a data object of type obj, the statement
ptr++; has the same effect as **ptr += sizeof(obj);**

This aspect of pointer arithmetic is particularly relevant to arrays because array elements are stored sequentially in memory. If a pointer points to array element n, incrementing the pointer with the (++) operator causes it to point to element n + 1.

This is illustrated in Figure 3, which shows an array named x[] that consists of four-byte elements (for example, a structure containing two type int members, each two bytes long). The pointer ptr was initialized to point to x[0]; each time ptr is incremented, it points to the next array element.

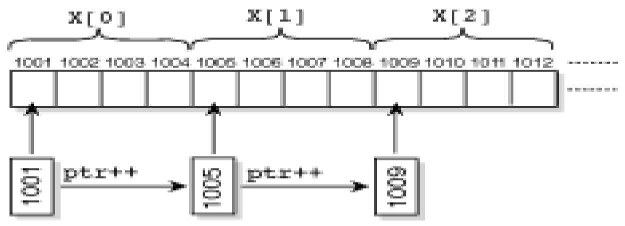

Fig. 3 With each increment, a pointer steps to the next array element.

This means that your program can step through an array of structures by incrementing a pointer. This sort of notation is usually easier to use and more concise than using array subscripts to perform the same task. Following example shows how you do this.

*STRUCTURE AND FUNCTION

C supports passing of structure values as arguments to functions. Tare 3 methods through which the values of structure can be transferred from one function to another.

1. Pass each member of the structure as an actual argument
2. Pass copy of entire structure
3. Pass address of structure

1. Pass each member of the structure as an actual argument

Each member of structure is passed as an actual argument in function call. It is exactly the way by which we pass the variables to function.

```
int get_total(int a, int b, int c);
struct stud { int m1,m2,m3,total; };
void main()
{
   struct stud s1 = { 10,20,30,0} ;
   s1.total=get_total(s1.m1,s1.m2,s1,m3);
   printf("%d",s1.total);
}
```

```
int get_total (int a, int b, int c)
{
     int tot;
     tot = a + b + c;
     return(tot);
}
```

Here get_total function will receive values of actual parameters s1.m1, s1.m2 and s1.m3 into formal parameters a, b and c respectively. And it returns total of 3 formal parameters. This method is helpful when number of parameters to be passed to function are less in number.

2. Pass copy of entire structure

When number of parameters to be passed to function are more in number then we can use this method. Here, copy of entire structure is passed to called function. Any changes done to structure members within called function are not reflected in original structure in the calling function. Because we use concept of call by value.

If we want changes done to structure members within called function to be reflected in original structure in the calling function then we need to return entire structure back to the calling function.

General format of sending a copy of structure to the called function is:

function_name (structure_variable_name);

Format of called function would be:

```
data_type function_name (struct_type st_name)
{
   ---------------------------
   ---------------------------
   return (expression);
}
```

For E.g.

**#include <stdio.h>
#include <conio.h>
void cal(struct stud t);
struct stud
{ int m[6],total;
   float per;
};
void main()
{
   clrscr();
   struct stud s1 = { 10, 20, 15, 25, 18, 22};
   cal(s1);
   getch();
}
 	
void cal(struct stud t)
{
     int i;
     t.total=0;
     for (i=0; i<6; i++)
          t.total = t.total + t.m[i];
     t.per=t.total/6.0;
     printf("%d   %f",t.total, t.per);
}**
	**#include <stdio.h>
	#include <conio.h>
	struct stud cal(struct stud t);
	struct stud
	{ int m[6],total;
	   float per;
	};
	void main()
	{
	   clrscr();
	   struct stud s1 = { 10, 20, 15, 25, 18, 22};
	   s1=cal(s1);
	   printf("%d %f",s1.total, s1.per);
	   getch();
	}
	
	struct stud cal(struct stud t)
	{
	     int i;
	     t.total=0;
	     for (i=0; i<6; i++)
	          t.total = t.total + t.m[i];
	     t.per=t.total/6.0;
	     return(t);
	}**

Here, s1.total and s1.per will not receive the changes done in total and per in cal function.	Here, s1.total and s1.per will receive the changes done in total and per in cal function.

<u>3. Pass address of structure</u>

Here we pass address of structure to the called function. This method is more efficient as compared to the second method.

```
#include <stdio.h>
#include <conio.h>
void cal(struct stud *p);
struct stud
{  int m[6],total;
   float per;
};
void main()
{
   clrscr();
   struct stud s1 = { 10, 20, 15, 25,
   18, 22};
   cal(&s1);
   printf("("%d   %f",s1.total, s1.per);
   getch();
}
```

```
void cal(struct stud *p)
{
   int i;
   p->total=0;
   for (i=0; i<6; i++)
        p->total = p->total + p->m[i];
   p->per=p->total/6.0;
}
```

In this example, address of structure variable s1 is passed to function cal and it is stored in pointer variable p. Now pointer p will point s1. Its members can be accessed using following notation:
p->total, p_m[i], p->total etc.

The symbol **->** is known as **arrow operator**. It is also known as **member selection operator**.
Note: p-> is another way of writing **s1**.

*UNIONS

Unions are similar to structures. A union is declared and used in the same ways a structure is used.

A union differs from a structure in that only one of its members can be used at a time. The reason for this is simple. All the members of a union occupy the same area of memory. They are laid on top of each other.

DEFINING, DECLARING, AND INITIALIZING UNIONS

Unions are defined and declared in the same fashion as structures. The only difference in the declarations is that the keyword union is used instead of struct.

A union is a collection of one or more variables (union_members) that have been grouped under a single name. In addition, each of these union members occupies the same area of memory location. While in structures, each member has its own storage location. So, a union may contain many members of different types, it can handle only one member at a time.

The keyword union identifies the beginning of a union definition. It's followed by a tag that is the name given to the union. Following the tag are the union members enclosed in braces. An instance, the actual declaration of a union, also can be defined. If you define the structure without the instance, it's just a template that can be used later in a program to declare structures.

The following is a template's format:

```
union tag {
          union_member(s);
        };
```

To use the template, you would use the following format:

```
union tag instance;
```

To define a simple union of a char variable and an integer variable, you would write the following:

```
union shared {
            char c;
            int i;
          } instance;
```

This union, named as shared, can be used to create instances of a union that can hold either a character value c or an integer value i. This is an OR condition. A structure would hold both values, the union can hold only one value at a time. Figure 3 illustrates how the shared union would appear in memory.

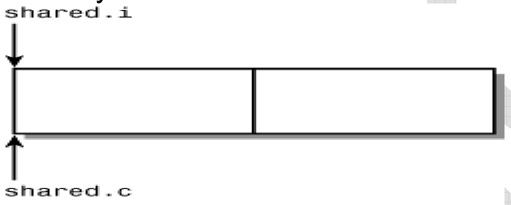

Fig 4. The union can hold only one value at a time.

Example 1
/* Declare a union template called tag */
union test {
   int nbr;
   char character;
}
/* Use the union template */
union test t;
	Example 2
	/* Declare a union and instance together */
	union testing {
	   char c;
	   int i;
	   float f;
	   double d;
	} g;

Example 3
/* Initialize a union. */
union date_tag {
   char full_date[9];
   struct part_date_tag {
      char month[2];
      char break_value1;
      char day[2];	

```
        char break_value2;
        char year[2];
    } part_date;
} date = {"01/01/97"};
```

A union can be initialized on its declaration. Because only one member can be used at a time, only one can be initialized. To avoid confusion, only the first member of the union can be initialized. The following code shows an instance of the shared union being declared and initialized:
union shared g = {`A'};
Notice that the g union was initialized just as the first member of a structure would be initialized.

ACCESSING UNION MEMBERS

Individual union members can be used in the same way that structure members can be used--by using the member operator (.).

However, there is an important difference in accessing union members. Only one union member should be accessed at a time. Because a union stores its members on top of each other, it's important to access only one member at a time. Listing 4 presents an example.

Listing 5. An example of the wrong use of unions.
/* Example of using more than one union member at a time */

```
#include <stdio.h>
void main()
{
     union shared_tag
     {
        char   c;
        int    i;
        long   l;
        float  f;
        double d;
      } shared;
     shared.c = `$';
     printf("\nchar    c          =    %c",
shared.c);
     printf("\nint i    = %d",  shared.i);
     printf("\nlong l   = %ld", shared.l);
     printf("\nfloat f  = %f",  shared.f);
     printf("\ndouble      d      =      %f",
shared.d);

     shared.d = 123456789.8765;
     printf("\n\nchar   c          =   %c",
shared.c);
```

```
char c   = $
int i    = 4900
long l   = 437785380
float f  = 0.000000
double d = 0.000000
char c   = 7
int i    = -30409
long l   = 1468107063
float f  = 284852666499072.000000
double d = 123456789.876500
```

printf("\nint i = %d", shared.i);
printf("\nlong l = %ld", shared.l);
printf("\nfloat f = %f", shared.f);
printf("\ndouble d = %f\n", shared.d);
}	

A union named shared is defined and declared. shared contains five members, each of a different type.

Character variable, c, was initialized before first bunch of printf statements so it is the only value that should be used until a different member is initialized. The results of printing the other union member variables (i, l, f, and d) can be unpredictable. Then we put value into the double variable, d.

Notice that the printing of the variables again is unpredictable for all but d. The value entered into c has been lost because it was overwritten when the value of d was entered. This is evidence that the members all occupy the same space.

DON'T try to initialize more than the first union member.
DO remember which union member is being used. If you fill in a member of one type and then try to use a different type, you can get unpredictable results.
DON'T forget that the size of a union is equal to its largest member.

USER-DEFINED TYPE DECLARATION (typedef)

C supports a feature known as "type definition" that allows users to define an identifier that would represent an existing data type.

The user-defined data type identifier can later be used to declare variables. It takes the general form:

typedef type identifier;

Where type refers to an existing data type and "identifier" refers to the "new" name given to the data type. The existing data type may belong to any class of type, including the user-defined ones.

Remember that the new type is 'new' only in name, but not the data type. typedef cannot create a new type. Some examples of type definition are:

typedef int units;
typedef float marks;

Here, units symbolizes int and marks symbolizes float. They can be later used to declare variables as follows:

units batch1, batch2;
marks name1[50], name2[50];

batch1 and batch2 are declared as int variable and name1[50] and name2[50] are declared as 50 element floating point array variables.

The main advantage to typedef is that we can create meaningful data type names for increasing the readability of the program.

***FILE HANDLING**

***INTRODUCTION AND USAGE**

FILE

Definition: A file is a collection of related data stored in secondary storage device.

We use functions like scanf and printf to read and write (display) data. These Input / Output functions always use the terminal (keyboard and screen) as the target place. This works fine as long as the data is small.

However, many real life problems involve large volumes of data and in such situations the terminal oriented I/O operations create two major problems:

1. It becomes cumbersome and time consuming to handle large volumes of data through terminals.
2. The entire data is lost when either the program is terminated or the computer is turned off.

So more flexible method is necessary where data can be stored on the disks and can be read whenever necessary, without destroying the data. This method uses the concept of files to store data.

A file is a place on the disk where a group of related data is stored. We frequently use files for storing information which can be processed by our programs. In order to store information permanently and retrieve it we need to use files.

Files are not only used for data. Our programs are also stored in files.

Like most other languages, C supports a number of functions that have the ability to perform basic file operations. Like

- Naming a file
- Opening a file
- Reading data from a file
- Writing data to a file and
- Closing a file

Important file handling functions available in the C library are listed in the following table – 1.

fopen()	Creates a new file for use Opens an existing file for use
fclose()	Cloases a file which has been opened for use

getc()	Reads a character from file
putc()	Writes a character to a file
fprintf()	Writes a set of data values to a file
fscanf()	Reads a set of data values from a file
getw()	Reads an integer from a file
putw()	Writes an integer to a file

*DEFINING AND OPENING FILE

If we want to store data in a file in the secondary memory, we must specify certain things about the file to the operating system. Like

1. Filename	Filename is a string of characters that make up a valid filename for the operating system. It may contain two parts, a primary name and an optional period with the extension. Examples: input.dat, prog.c
2. Data structure	Structure of a file is defined as **FILE** in the library of standard I/O function definitions. So all files should be declared as type FILE before they are used. FILE is a defined data type.
3. Purpose	When we open a file, we must specify that what we want to do with the file. For example, we may write the file or read the already existing data.

General format for declaring and opening a file:

```
FILE *fp;
fp = fopen(“filename”,”mode”);
```

First statement declares the variable **fp** as “pointer to the data type FILE”. FILE is s structure in the I/O library.

The second statement opens the file named filename and assigns an identifier to the FILE type pointer fp. This pointer contains all the information about the file. And it is subsequently used as a communication link between the system and the program.

The second statement also specifies the purpose of opening this file. The **mode** does this job.

File Access Modes can be one of the following:

r	Read mode	Opens the file for reading only
w	Write mode	Opens the file for writing only
a	Append mode	Opens the file for appending (or adding) data to it

Both the filename and mode are specified as strings. They should be enclosed in double quotation marks.

When you try to open a file, one of the following things may happen:

1. When the mode is ‘writing’, a file with the specified name is created if the file does not exist. The contents are deleted, if the file already exists.
2. When the purpose is ‘appending’, the file is opened with the current contents safe. A file with the specified name is created if the file does not exist.
3. If te purpose is ‘reading’, and if it exists, then the file is opened with current contents safe otherwise an error occurs.

Consider the following statements:

```
FILE *p1, *p2;
p1 = fopen (“data”, ”r”);
p2 = fopen (“results”, ”w”);
```

The file **data** is opened for reading and results is opened for writing. In case, the **results** file already exists, its contents are deleted and the file is opened as a new file. If data file does not exist an error will occur.

Many recent compilers include **additional File Access Modes** of operation. Like

r+	The existing file is opened for both reading and writing
w+	Same as w except both for reading and writing
a+	Same as a except both for reading and writing

We can open and use a number of files at a time. This number however depends on the system we use.

<u>*CLOSING A FILE</u>

A file must be opened as soon as all operations performed on it have been completed.

So,
1. This gives guarantee that information associated with the file is flushed out from the buffers.
2. Also all links to the file are broken.
3. It also prevents any unintentional misuse of the file.
4. Suppose there is a limit on number of files that can be kept open simultaneously. Then closing unwanted files may help to open required files.

We have to close a file when we want to reopen the same file in a different mode. The I/O library supports a function to do this. It takes the following form:

fclose (file_pointer);

This would close the file associated with the FILE pointer file_pointer. Look at the following statements of a program.

```
FILE *p1, *p2;
p1 = fopen (“INPUT”, ”w”);
p2 = fopen (“OUTPUT”, “r”);
fclose (p1);
fclose (p2);
```

This program opens two files and closes them after all operations on them are completed. Once a file is closed, is file pointer can be reused for another file.

Actually all files are closed automatically whenever a program terminates. However, closing a file as soon as you are done with it is a good programming habit.

*INPUT / OUTPUT OPERATIONS ON FILES

Once a file is opened, reading or writing is done using the standard I/O functions mentioned in Table 1.

getc and putc Functions

Full form	getc: get character
	putc: put character

The simplest file I/O functions are getc and putc. These are similar to getchar and putchar functions. They handle one character at a time.

Assume that a file is opened with mode “w” and file pointer is fp. Then, the statement

putc (c, fp1);

writes the character stored in the character variable c to the file associated with FILE pointer fp1.

Same way, getc is used to read a character from a file that has been opened in read mode.

For example, statement

c = getc (fp2);

will read a character from the file whose file pointer is fp2.

File pointer moves by one character position for every operation of getc or putc. The getc will return **end-of-file marker EOF**, when end of the file has been reached. So reading should be terminated when EOF is encountered.

getw and putw Functions

Full form	getw: get word (word means 2 bytes)
	putw: put word

getw and putw are integer-orinted functions. They are similar to the getc and putc functions. They are used to read and write integer values.

These functions would be useful when we deal with only integer data. The general forms of getw and putw are:

> **putw (integer, fp);**
> **getw (fp);**

fprintf and fscanf Functions

Full form	fprintf: file print formatted
	fscanf: file scan formatted

getc, putc, getw and putw functions can handle only one character or integer at a time. Most compilers support two other functions fprintf and fscanf which can handle a group of mixed data simultaneously.

The functions fprintf and fscanf perform I/O operations similar to printf and scanf functions but they work only on files.

The first argument of these functions is a file pointer which specifies the file to be used.

The general form of fprintf is
> **fprintf (fp, "control string", list);**

Where **fp** is file pointer associated with a file that has been opened for writing. The control string contains output specifications (format specifiers) for the items in the list. The list may include variables, constants and strings.

Example: fprintf (f1, "%s %d %f", name, age, 7.5);

Here, name is an array variable of type char and age is an int variable.

The general format of fscanf is
> **fscanf (fp, "control string", list);**

This statement will read the items in the list from the file specified by **fp** according to the specifications contained in the control string.

Example: fscanf (f2, "%s %d", item, &quantity);

Like scanf, fscanf also returns the number of items that are successfully read. When the end of file is reached, it returns the value EOF.

ERROR HANDLING DURING I/O OPERATIONS

It is possible that an error may occur during I/O operations on a file. Typical error situations include:

1. Trying to read beyond the end-of-file mark.
2. Trying to use a file that has not been opened.
3. Trying to perform an operation on a file, when the file is opened for another type of operation.
4. Opening a file with an invalid filename.
5. Attempting to writr to a write-protected file.

If we fail to check such read and write errors, a program may behave abnormally when an error occurs. An unchecked error may result in a premature termination of the program or incorrect output.

We have two status-inquiry library functions: feof and ferror that can help us to detect I/O errors in the files.

feof function

The feof function can be used to test for and end of file condition. It takes a FILE pointer as its argument.

It returns a nonzero integer value if all of the data from the specified file has been read. It returns zero otherwise.

If fp is pointer to file that has just opened for reading, then the statement

```
if ( feof (fp) )
   printf (“End of data”);
```

will display the message “End of data” on reaching the end of file condition.

ferror function

The ferror function reports the status of the file indicated. It takes a FILE pointer as its argument.

It returns a nonzero integer value if an error has been detected up to that point during processing. It returns zero otherwise.

The statement

```
if ( ferror (fp) != 0 )
   printf (“An error has occurred”);
```

will display the message “An error has occurred”, if the reading is not successful.

We know that whenever a file is opened using fopen function, a file pointer is returned. If the file cannot be opened for some reason, then the function returns a NULL pointer. This facility can be used to test whether a file has been opened or not.

Example:

```
if ( fp == NULL )
   printf(“File could not be opened”);
```

Disclaimer

The study material is compiled by Ami D. Trivedi. The basic objective of this material is to supplement teaching and discussion in the classroom in the subject. Students are required to go for extra reading in the subject through library work.