Pointers:

→ A pointer is a memory variable that stores a memory address

→ Pointer can have any name as of other variable and it is declared in the same fashion like other vari but it is always denoted by '*' operator.

Features of Pointers:

1. Pointers save the memory space.
2. Execution time with pointer is faster becau the data is manipulated with the address ie direct access to memory location.
3. The memory is accessed efficiently with the pointe The pointers assigns memory space and it also releases Dynamically memory is allocated.
4. Pointers are used with data structures. They are useful to represent two dimensional and multidimensional ar

Pointer Declaration:

→ Pointer variables can be declared as follows.

Ex: `int *x;`
`float *f;`
`char *ch;`

→ In the first statement x is an integer pointer and it tells the compiler that it holds the address of any integ variable. In the same way f is a floating pointer and ch is character pointer that stores the address of fl and character variables respectively.

→ The indirection operator (*) is also called as deferencing operator. When a pointer is dereferenced, the value at that address stored by the pointer is retrieved.

→ Normal variable provides direct access to their values. Whereas pointer provides indirect access to value of variable whose address it stores

→ The indirection operator (*) is used in two distinct ways with the pointers declaration and deference.

→ When a pointer is declared, the star indicates that it is a pointer, not a normal variable.

→ when a pointer is ~~defe~~ dereferenced. the ~~star~~ ~~dec~~ indicate that value at that memory location stored in the pointer is to be accessed rather than address.

→ The indirection operator (*) is the same operator as multiplication operator. The Compiler knows which operator to call based on the Content.

→ The & is the address operator and it represents the address of the variable. The %u is used with printf() function for printing the address of a variable along with the & operator immediately proceeding with variable to return the address of variable which is a whole number

Example Write a program to display the value of variable and it location using pointers

```
#include <stdio.h>
#include <conio.h>
main()
{  int v=10, *p;
   clrscr();
```

```
printf("\n Address of v = %u", &v);
printf("\n value of v = %d", *p);
printf("\n Address of p = %u", &p);
getch();
}
```

output : Address of v = 4060
Value of v = 10
Address of p = 4062

Explanation:

→ In the above program v is an integer variable and its value is 10. The variable p is declared as pointer variable

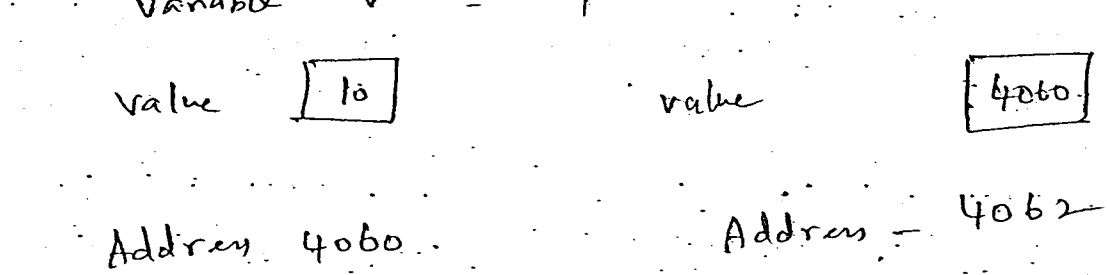
→ The statement p = &v assigns the address of v to p i.e p is the pointer to the variable v.

→ To access the address and value of v pointer 'p' can be used.

→ The value of p is nothing but address of variable v and display the value *p is used.

→ The pointer variable also have an address and are displayed using & operator.

→ The above explanation can be given by diagram as shown below.



Write a program to add two numbers through variable and their pointers

```
#include<stdio.h>
#include<conio.h>
main()
{ int a, b, c, d, *p1, *p2;
  clrscr();
 printf("Enter two numbers:");
 scanf("%d %d", &a, &b);
    p1 = &a;
    p2 = &b;
    c = a+b;
    d = *p1 + *p2;
 printf("\n Sum of A & B using variable: %d", c);
 printf("\n Sum of A & B using pointers: %d", d);
 getch();
}
```

output: Enter two numbers: 5 6
sum of A & B using variable: 11
sum of A & B using pointers: 11

Explanation:
→ In the above program sum of two numbers is obtained by two ways.
1. Adding variables a and b directly
2. Through pointer of a and b i.e p_1 and p_2 respectively
→ Address of a and b are stored in p_1 and p_2 respectively. Thus adding *p1 and *p2 gives the sum of a, b

Arithmetic Operations with pointers:

→ Arithmetic operation on pointer variables like increase, decrease, prefix and postfix operations can be performed

→ The increase of pointer variable for integer, the address is increased by two because integer requires 2 byte

→ Similarly for characters, floating point and double requires 1, 4 and 8 bytes respectively.

→ The following table shows the increase and decrease of pointer variable for all data type variables

Initial Data type	Address	operation		Address after operation		Require bytes
int i = 2	4046	++	--	4048	4044	2
char c = 'x'	4053	++	--	4054	4052	1
float f = 2.2	4058	++	--	4062	4054	4
double d = 2.6	4060	++	--	4068	4052	8

Write a program to show the effect of increment on pointer variables. Display the memory location of integer character and floating point numbers before and after increment of pointers

```
#include <stdio.h>
#include <conio.h>
void main()
{ int x, *x1;
  char y, *y1;
  float z, *z1;
```

```
clrscr();
printf("Enter integer, character, float values: ");
scanf("%d %c %f", &x, &y, &z);
x1 = &x;
y1 = &y;
z1 = &z;
printf("Address of x = %u\n", x1);
printf("Address of y = %u\n", y1);
printf("Address of z = %u\n", z1);
x1++;
y1++;
z1++;
printf("After Increment in pointers:\n");
printf("Address of x = %u\n", x1);
printf("Address of y = %u\n", y1);
printf("Address of z = %u\n", z1);
getch();
}
```

output: Enter integer, character, floating value: 2

Address of x = 4046
Address of y = 4053
Address of z = 4058
After Increment in pointers
Address of x = 4048
Address of y = 4054
Address of z = 4062

Explanation: In the above Program 4046 is the addr of integer x, 4053 for character value y and 4058 for float

→ On increasing of pointer the address of integer, character, and float will be 4048, 4054 and 4062 respectively.

→ This is because every time a pointer points immediate to or a previous location of its type after increase or decrease in the operation respectively.

Write a program to perform different arithmetic operation using pointers

```
#include<stdio.h>
#include<conio.h>
main()
{ int a=25, b=10, *p, *q;
  p=&a;
  q=&b;
  clrscr();
  printf("\n Addition of a+b = %d", *p+b);
  printf("\n Subtraction of a-b = %d", *p-b);
  printf("\n Product of a*b = %d", *p**q);
  printf("\n Division of a/b = %d", *p/(*q));
  printf("\n a Mod b = %d", *p%(*j));
  getch();
}
```

output: Addition of a+b = 35
Subtraction of a-b = 15
Product of a*b = 250
Division of a/b = 2
a Mod b = 5

Explanation

→ The various arithmetic operation can be perform on a pointer such as addition, subtraction, multiplicat division and mod.

→ The value stored at the address is taken for operation but not the address.

→ The arithmetic operations are impossible with addresses.

Ex: Two addres location are not possible to add.

Pointers and Arrays:

→ Array name itself is an address or pointer.

→ It points to the address of the first element of the array.

→ The elements of array together with their addresses can be displayed by using array name itself.

→ Array elements are always stored in contiguous memory location.

Write a program to display array element with their addre using array name as pointers

```
#include<stdio.h>
#include<conio.h>
main()
{ int x[5] = {2, 4, 6, 8, 10}, k=0;
  clrscr();
  printf("Element Number \t Element\t Address \n");
```

```
    while (k<5)
    {
        printf (" x[%d] = \t %d \t %u \n", k, *(x+k), x+k
        k++;
    }
}
```

output:-

Element Number	Element	Address
x[0]	2	4056
x[1]	4	4058
x[2]	6	4060
x[3]	8	4062
x[4]	10	4064

Explanation:

→ In the above program variable k acts as an element number and its value varies from 0 to 4.

→ When it is added with array name x ie with address of first element, it points to consecutive memory location.

→ Thus the element no., element and their address are displayed.

Write a program to find sum of all the elements of an array by using array name as a pointer.

```
#include<stdio.h>
#include<conio.h>
main()
{ int sum=0, i=0, a[]={1,2,3,4,5};
  clrscr();
```

```
printf ("Elements \t Values \t Address \n");
while (i<5)
{
    printf ("a[%d] \t %d \t %u \n", i, *(a+i), (a+i));
    sum = sum + *(a+i);
    i++;
}
printf ("Sum of Array elements = %d", sum);
getch();
}
```

output:

Elements	Values	Address
a[0]	1	4056
a[1]	2	4058
a[2]	3	4060
a[3]	4	4062
a[4]	5	4064

Sum of array elements = is

Explanation:

→ In this program array name 'a' act as pointer and variable 'i' is used for referring elem numbers

→ Using the while loop and expression *(a+i) and (a+i) various elements and their addresses are displayed respectively.

→ In the sum variable sum of all the elements is obtained.

Pointers and two dimensional arrays:

→ A matrix can represent two dimensional array. of an element where first argument is row numb and second as column number.

→ To display the elements of two dimensional array using pointers it is essential to have & operator as prefix with an array name followed by element number, otherwise compiler shows an error.

Write a program to display array elements and their address using pointers

```
#include <stdio.h>
#include <conio.h>
main()
{
    int i, j=1, *p;
    int a[3][3] ={{1,2,3}, {4,5,6}, {7,8,9}};
    clrscr();
    printf("Elements of an array with their address \n");
    p= &a[0][0];
    for(i=0; i<9; i++, j++)
    {
        printf("%5d [%5u] ", *p, p);
        p++;
        if(j==3)
        { printf("\n");
          j=0;
        }
    }
    getch();
}
```

output:

Elements of an array with their addresses.

1 [4052] 2 [4054] 3 [4056]

4 [4058] 5 [4060] 6 [4062]

7 [4064] 8 [4064] 9 [4064]

Explanation:

→ In the above program two dimensional arr is declared and initialized.

→ The base address of an array is assigned t integer pointer p, while assigning & operator is to b prefixed with array name followed by element number

→ The ~~statement~~ pointer p is printed and increased in for loop till it prints the entire array elements

→ The if statement splits a line when the elements in each row are printed.

Array of pointers:

→ Arrays of different standard data types such as array of int, float, character etc are possible in c.

→ In the same way 'c' language also supports arr of pointers which ~~before~~ is nothing but collection of addresses.

→ Here, we store address of variables for which we have to declare an array as a pointer.

Write a program to store addresses of different elements of an array using array of pointers.

```
#include <stdio.h>
#include <conio.h>
main()
{ int *arrp[3];
  int a[3] = {5, 10, 15}, k;
  clrscr();
  for(k=0; k<3; k++)
    { arrp[k] = a+k;
    }
  printf("Address \t Element \n");
  for(k=0; k<3; k++)
    { printf("\t%u", arrp[k]);
      printf("\t%d", *(arrp[k]));
    }
  getch();
}
```

arrp[0] = &a[0]
arrp[1] = &a[1]
arrp[2] = &a[2]

output:

Address	Element
4060	5
4062	10
4064	15

Explanation:
→ In the above program *arrp[3] is declared as an array of pointers.
→ using first for loop the address of various ele of array 'a' are assigned to *arrp[].
→ The second for loop pick up the addresses fr arrp[]. and display value present at that location along wi the address.

Pointers to Pointers:

→ Pointer is known as a variable containing address of another variable

→ The pointer variables also have an address.

→ The pointer variable containing address of another pointer variables is called as Pointer to Pointer.

→ This chain can be continued to any extent.

Write a program to print value of a variable through pointer and pointer to pointer.

variable	Pointer *	Pointer to pointer **
(4056) a	(4060) p	(4064) q
2	(4056)	(4060)

q = 4060
*q = 4056
**q = 2

```
#include<stdio.h>
#include<conio.h>
main()
{ int a=2, *p, **q;
  p = &a;
  q = &p;
  clrscr();
  printf("value of a = %d; Address of a = %u\n", a,
printf("Through *p the value of a = %d, Address of a = %u\n", *p, p
printf("Through **q the value of a = %d, Address of a = %u\n", **q, q
  getch();
}
```

output:

value of a = 2, Address of a = 4056.
Through *p value of a = 2, Address of a = 4056
Through **q value of a = 2, Address of a = 4056

Exploration:

→ In the above program variable p is declared as a pointer. The variable q is declared as pointer to another pointer.

→ The address of variable a is assigned to pointer p.

→ The address of pointer p is assigned to q.

→ The variable q contains the address of the pointer variable *p. Hence q is pointer to pointer.

→ The program displays value and address of variable a, my variable itself and pointers p and q.

Void pointers:

→ Pointers can also be declared as void type

→ Void pointers cannot be ~~def~~ dereferenced without explicit type conversion because being void the compiler cannot determine the size of the object that pointer points to.

→ Void pointer declaration is possible but void variable declaration is not allowed.

→ The declaration void p displays the error message "size of p is unknown or zero" after compilation

Write a program to declare as void pointer. Assign address of int, float and char variables to the void pointers using type casting method. Display the contents of various variables

```
#include<stdio.h>
#include<conio.h>
 int p;
 float d;
 char c;
void *pt = &p;
main()
{
 *(int*) pt = 12;
   printf("p = %d\n", p);
    pt = &d;
 *(float*) pt = 2.3;
   printf("d = %f\n", d);
   pt = &c;
 *(char*) pt = 'S';
   printf("c = %c", c);
 getch();
}
```

output: p = 12
d = 2.3
c = S

Explanation:

→ In the above example variable p, d and c are declared as int, float and char respectively

→ pointer pt is a pointer of type void

→ The pointer is initialized with address of integer variable p ~~the pointer 'p' points to variable~~

→ The statement *(int*) pt = 12 assigns the integer value 12 to pointed pt i.e. to variable p.

→ The declaration *(int*) tells the compiler the value assigned is of integer type.

→ ~~assignment of float and~~ char type are ca

Pointer to function:

→ In c language every variable has address except register variables. We can access the address of variable using pointers.

→ C function also have an address. ~~which we~~

→ ~~invoke~~ ~~using~~ we invoke the function using its address.

Write a program to display address of user defined function

```
#include <stdio.h>
# include <conio.h>
~~int~~ show (void);
main()
{
    clrscr;
     show();
    printf("%u", show);
}
show()
{ printf (" Address of function show() is :");
}
```

output: Address of function show() is : 530

Explanation: → In the above program the show() is a user defined function.

→ The function name without parenthesis displays the address of the function.

→ In the output the address of function show is displayed.

→ To assign the obtained address of function, we need to declare a pointer which can hold the address of the function.

→ The following statement declare the pointer that can hold the address of the function.

```
int (*p) ();
```

Here 'p' is the pointer name and the bracket indicates that it is a pointer to function.

Example: Write a program to call a function using pointer

```
#include <stdio.h>
#include <conio.h>
  show();
main()
{ int (*p) ();
   p = show;
   (*p) ();
  printf ("%u", show);
  getch();
}
 show()
 { clrscr();
  printf("Address of function show() is:");
 }
```

Output: Address of function show() is : 531

Explanation → In the above program variable p is pointer to function → Address of function show() is assigned to pointer p → Using function pointer the function show() is invoked and the output of program is displayed.

```
#include <stdio.h>
#include <conio.h>
main()
{
    int x[50], i, n;
    clrscr();
    printf("Enter the size of array:\n");
    scanf("%d", &n);
    printf("Enter the elements of array:");
    for(i=0; i<n; i++)
    {
        scanf("%d", (x+i));
    }
    printf("The elements of array are:");
    for(i=0; i<n; i++)
    {
        printf("%5d", *(x+i));
    }
    getch();
}
```

100 10
3
106
104

(20)

Write a program using pointer to read an array of integer and prints its elements in reverse order.

```
#include <stdio.h>
#include <conio.h>
main()
{
    int x[50], i, n, *y;
    clrscr();
    printf(" Enter the size of array : ");
    scanf("%d", &n);
    y = x;
    printf(" Enter the elements of array : ");
    for(i=0; i<n; i++)
      { scanf("%d", (y+i));
      }
    printf(" The elements of array in reverse order is
    for(i=n-1; i>=0; i--)
      {
        printf("%5d", *(y+i));
      }
    getch();
}
```

Write a c program ~~that~~ ~~way~~ ~~of~~ ~~pointer~~ to show that (11)
pointers of any datatype occupies same space.

```
#include<stdio.h>
#include<conio.h>
main()
{
    int *i;
    float *f;
    char *c;
    double *d;
    clrscr();
    printf("The size of integer pointer is %d\n", sizeof(i));
    printf("The size of character pointer is %d\n", sizeof(c));
    printf("The size of float pointer is %d\n", sizeof(f));
    printf("The size of double pointer is %d\n", sizeof(d));
    getche();
}
```

free function is used to free the memory allocation allocated by malloc, calloc and realloc when they are no longer in use.

→ The declaration of free function is

```
void free (void *ptr);
```

Dynamic Memory allocation:

→ In static memory allocation, memory is allocated to the variables and arrays at compile time

→ Allocating memory to the object at runtime is called as dynamic memory allocation.

Memory Management functions:

→ four types of memory have been defined for dynamic memory allocation.

1. malloc()
2. calloc()
3. realloc()
4. free()

→ First 3 functions are used to allocate memory and last one is to free the memory that is not in use

→ All these function are included in the standard library function, stdlib.h.

Malloc():

→ malloc function is used to allocate a block of memory and it returns void pointer to the address of first byte in the memory.

→ The declaration of malloc function is as follows:

```
void *malloc(size_t size);
```

→ size_t represents the type, it is usually an unsigned integer.

→ The size of the malloc's actual parameters is defined by using sizeof operator.

Example:

int m = void *malloc (sizeof (int))

→ The malloc function with zero size may return a null pointer or some other value.

Calloc:

→ Calloc function is used to allocate memory dynamical for arrays.

→ The only difference between calloc and malloc is that calloc sets memory to null character but malloc doesn't

→ Declaration of calloc is as follows.

void *calloc (size_t element_count, size_t elem size);

calloc function with zero size may return a null pointer.

Ex: int .c[] = (int *) calloc (50, sizeof (int));

→ Here array of 50 integer is allocated.

Realloc:

→ Realloc resizes the block of memory either by deleting or extending the memory block

→ It allocates new block, if the existing block cannot be extended and copies the existing memory allocation to new block and deletes the old one.

void *realloc (void *ptr, size_t newsize);

free:

→ free function is used to free the memory allocation allocated by malloc, calloc and realloc when they are no longer in use.

→ The declaration of free function is

```
void free (void *ptr);
```

UNIT-4

STRUCTURES, UNIONS and FILES

Structures and Unions:

→ Introduction
→ Defining a structure
→ Declaring structure variables
→ Accessing structure members
→ Structure initialization
→ Copying and comparing structure variables
→ Operations on individual members
→ Arrays of structures
→ Arrays within structures
→ Structures within structures
→ Structures and functions
→ Unions
→ Size of structures
→ Bit fields
→ Type def
→ Enum

1. INTRODUCTION:- (OR) SIGNIFICANCE OF STRUCTURES (OR) NEED FOR STRUCTURES:-

⇒ A variable can store a single value at a time.
⇒ Arrays can be used to represent a collection of similar data items.
⇒ We can't use arrays if we want to represent a collection of different data items using a single name.
⇒ For example consider, we want to maintain employee information.
→ Employee information may include employee name, age, qualification, salary etc.
→ To maintain employee information different data types are required.
→ Name and qualification are char data type, age is integer and salary is float.
→ All these data types can't be represented in a single array. We need to declare different arrays for each data type. Hence, arrays can't be useful here.
→ For handling these mixed data types, C provides a feature known as structures

DEFINITION:-

⇒A structure is a collection of hetrogeneous elements.

(OR)

⇒A structure is a collection of different data items grouped under a single name.

EXAMPLES (OR) USES:-

⇒Structures can be used to represent the following:

1. Customer details : name, telephone, city, category
2. Address : name, door-number, street, city
3. Employee details : name, age, qualification, salary.
4. City : name, country, population
5. Book details : author, title, price, year
6. Date : day, month, year
7. Time : seconds, minutes, hours

⇒By using structures, the data can be organized in a more meaningful way.

FEATURES OF STRUCTURES:-

1. Nesting of structures is possible. ie. one can create structure within a structure. Using this feature one can handle complex data types.
2. To copy elements of one array to another array, the elements are copied one by one. It is not possible to copy all the elements at a time.

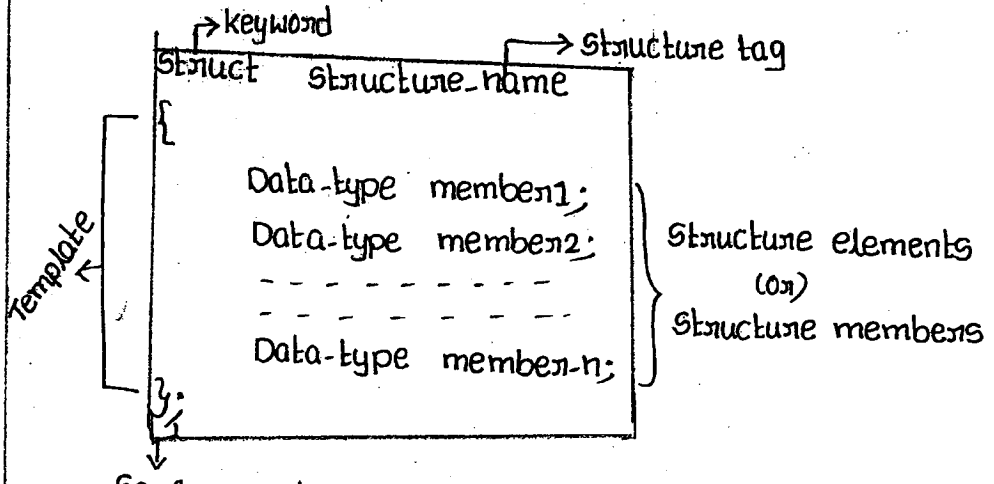
   Where as in structure, it is possible to copy the all the structure elements at a time using = operator.
3. It is also possible to pass structure elements to a function.
4. It is also possible to create structure pointers.

2. DEFINING A STRUCTURES:-

⇒ Each and every structure must be defined and declared before they are used in a program.

⇒ Structure definition creates a format that may be used to declare structure variables.

Syntax: The general format of structure definition is as follows:



→ Structure declaration always starts with struct keyword.

→ Structure_name is known as a tag, it may be used to declare structure variables. Structure name is optional in some situations.

→ The structure definition must be enclosed within a pair of curly braces.

→ The body of the structure should end with semi-colon.

→ Members of structures are not variables. They don't occupy any memory until they are associated with structure variables.

→ Each member of a structure contains its own memory.

Example1:-

⇒ Let us consider, we want to represent a book data-base.

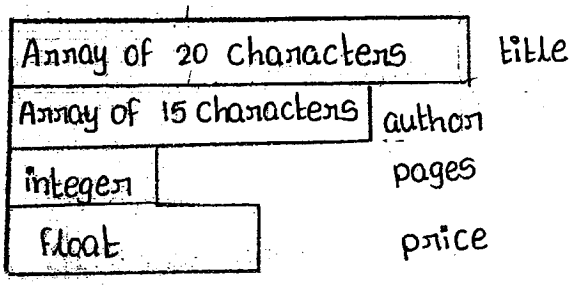
The book database consists of bookname, author, number of pages, and price. We can define a structure to represent this as follows:

```
struct book_bank
{
    char title[20];
    char author[30];
    int pages;
    float price;
};
```

→ The keyword struct declares a structure to hold the details of 4 data fields namely title, author, pages and price. These fields are called structure members (or) structure elements.

→ book_bank is the name of the structure.

→ The structure definition describes a format called template to represent the following information:



Example2:-

⇒ Define a structure to represent student information.

The student information may include roll number, student name, branch and semester.

```
struct student
{
    int roll_no;
    char name[20];
    int semester;
    char branch[10];
};
```

Example3:-

⇒ Define a structure to hold employee information (name, age, qualification and salary).

```
struct employee
{
    char name[30];
    int age;
    char qualification[10];
    float salary;
};
```

3. DECLARING STRUCTURE VARIABLES:-

⇒ Structure is a user-defined data type.

⇒ We can declare the structure variables using structure name any where in the program.

Syntax: `Struct structure_name Variable1, variable2, ---- variablen;`

Example:

```
Struct book_bank      →/* Structure definition */
{
    Char title[20];
    Char author[15];
    int pages;
    float price;
};
Struct book_bank b1, b2, b3; /* Declaration of structure variables */
```

Here b1, b2, b3 are structure variables of type book_bank. Each variable will have 4 fields namely title, author, pages and price.

After declaring the structure variables, memory is allocated.

⇒ It is also possible to combine both the structure definition and variables declaration in one statement as follows:

```
Struct book_bank
{
    Char title[20];
    Char authors[15];
    int pages;
    float price;
} b1, b2, b3;
```

⇒ The use of structure name (or) tag name is optional

Ex:

```
Struct
{
    Char title[20];
    Char author[15];
    int pages;
} b1, b2, b3;
```

NOTE: This method is not recommended because without a tag name, we can't use it for future declaration

ACCESSING STRUCTURE MEMBERS:- (OR) USE OF DOT OPERATOR;

⇨ Structure members themself are not variables. They should be linked to the structure variables in order to make them meaningful members.

→ The link between the member and a variable is established using the . (dot) operator.

→ dot operator (.) is also known as member operator (or) period operator.

⇒ We can access structure members by using . dot operator.

Syntax: `Structure variable . structure member`

Ex:
```
Struck book_bank
{
    Char title[40];
    Char author[30];
    float price;
    int year;
};
Struct book_bank b1,b2;
```
⇒ structure members

⇒ Structure variables

Now we can access the structure members as follows:

i.e. b1.title → Represents title of book1

b1.author → Represents author of book1

b1.price

b1.year

b2.title → Represents title of book2

b2.author

b2.price

b2.year

STRUCTURE INITIALIZATION:-

⇒There are many ways to initialize the structure members.

1. Structure variables can be initialized at compile-time.

```
struct book_bank
{
    char title[40];
    char author[30];
    float price;
    int year;
} b1 = {"CDS", "Balaguruswamy", 300.00, 1987};
```

This initialization assign CDS to b1.title
Balaguruswamy to b1.author
300.00 to b1.price
1987 to b1.year

NOTE:- The order of values enclosed in the braces must match the order of members in the structure definition.

Ex: We can also declare more than one structure variable and initialize them.

```
struct book_bank
{
    char title[40];
    char author[30];
    float price;
    int year;
};
struct book_bank b1 = {"CDS", "Balaguruswamy", 300.00, 1987};
struct book_bank b2 = {"SE", "Pressman", 450.00, 1990};
struct book_bank b3 = {"Networks", "Pressman", 500.00 1986};
```

2. We can also use assignment operator to initialize structure variables.

```
Ex: struct book_bank b1;
    b1.price = 300.00          strcpy(b1.author, "Balaguruswamy")
    b1.year = 1987
    strcpy(b1.title, "CDS")
```

3. Scanf function can be used to read the values through keyboard.

Ex:
```
struct book_bank
{
    char title[30];
    char author[40];
    float price;
    int year;
};
struct book_bank b;
```
⇒ `scanf("%s", b.title);`
`scanf("%s", b.author);`
`scanf("%f", &b.price);`
`scanf("%d", &b.year);`

NOTE:-

⇨ We cann't initialize individual members inside structure definition.

Ex:
```
struct book_bank
{
    float price = 300.00;
    char title[30] = "COS";
};
```
⇒ Invalid

⇨ It is permitted to have a partial initialization. we can initialize only the first few members and leave the remaining blank.

⇨ The uninitialized members will be assigned default values.

→ 0 for integer & floating-point members

→ '\0' for characters and strings.

EXAMPLE PROGRAMS:-

⇨ Define a structure book bank which contains title, author, price and year of publication. Write a c program to read this information from the keyboard and display the same on the screen.

```
#include<stdio.h>
#include<conio.h>
Void main()
{
    Struct book_bank   /* Defininig a structure */
    {
        Char title[40];
        Char author[30];
        Float price;
        Int year;
    };
    Struct book_bank b; /* Declaring a structure variable */
    Clrscr();
    Printf("Enter title of the book:");
    Scanf("%s", b.title);
    Printf(" Enter author name:");
    Scanf("%s", b.author);
    Printf("Enter price of the book:");
    Scanf("%f", &b.price);
    Printf(" Enter year of publication:");
    Scanf("%d", &b.year);
    Printf(" Book details are:");
    Printf(" Book name : %s", b.title);
    Printf("\n Author : %s", b.author);
    Printf("\n Price : %f", b.price);
    Printf("\n Year : %d", b.year);
    getch();
}
```

Explanation:-

The above program defines a structure book_bank that contains 4 members namely title, author, price and year. It reads the values through the keyboard and display the information using printf function.

Example Program 3:-

⇒ Define a structure Employee which contains employee id, name, qualification and salary. Write a C program to read the values through the keyboard and display the content of the structure.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    struct Employee  /* Defining a structure */
    {
        int id;
        char name[10];
        char qualification[20];
        float salary;
    };
    struct Employee e;  /* Declaring a structure variable */
    clrscr();
    /* Reading the values for structure members */
    printf("Enter employee details[id, name, qualification, salary]:");
    scanf("%d %s %s %f",&e.id,&e.name,&e.qualification, &e.salary);
    /* Displaying the values */
    printf("Employee ~~num~~ id : %d", e.id);
    printf("\n Employee name : %s", e.name);
    printf("\n Qualification : %s", e.qualification);
    printf("\n Salary : %f", e.salary);
    getch();
}
```

INPUT:-

Enter employee details[id, name, qualification, salary]: 12 Ajit managen 5000

OUTPUT:-

Employee id : 12
Employee name : Ajit
Qualification : Managen
Salary : 50000

Example Program3:-

⇨ Write a program to print the marks obtained by two students using structures

```
#include<stdio.h>
#include<conio.h>
Void main()
{
    Struct student
    {
       int marks1, marks2, marks3;
    };
    Struct student s1 ={90, 80, 70}; /* Compile time initialization */
    Struct student s2 ={95,80, 75};
    clrscr();
    printf(" marks obtained by st student: %d %d %d", s1.marks1, s1.marks2,
                                                        s1.marks3);
    printf("\n Marks obtained by 2nd student: %d %d %d", s2.marks1, s2.marks2,
                                                        s2.marks3);
    getch();
}
```

OUTPUT:-

Marks obtained by st student : 90 80 70
Marks obtained by 2nd student : 95 80 75

SIZE OF STRUCTURES:-

⇨ Structure is a collection of different data items.

⇨ **Size of structure = sum of size of all its members**

Ex:
```
Struct student
{
    int roll_no;
    char name[20];
    char branch[10];
    float marks;
};
```

Size of student structure = sizeof(roll_no) + sizeof(name) + sizeof(branch) + sizeof(marks)

= 2 + 20 + 10 + 4

= 36 bytes.

⇨ We can use sizeof operator to know the size of a structure.

⇨ There are two ways:

1. Syntax: `sizeof(struct structure_name);`

   Ex: sizeof(struct student);

   The above statement returns size of student structure.

2. Syntax: `sizeof(structure_variable);`

   Ex: struct student s;

   sizeof(s);
   ↳ structure variable.

Example program:-

⇨ Define a structure book which contains title, author, price & year of publication. Write a C program to display the size of the structure.

```
#include<stdio.h>
#include<conio.h>
void main()
{
   struct book
   {
      char title[20];
      char author[20];
      float price;
      int year;
   };
   struct book b;
   clrscr();
   printf("Size of structure book = %d", sizeof(struct book));
   printf("\n Size of structure book = %d", sizeof(b));
   getch();
}
```

OUTPUT:-

```
Size of structure book = 46
Size of structure book = 46
```

COPYING AND COMPARING STRUCTURE VARIABLES:-

1. COPYING STRUCTURE VARIABLES:-

⇨ Two variables of same structure type can be copied using '=' operator i.e. we can copy the content of one structure variable to another structure variable.

⇨ If S1 and S2 are two structure variables of same type then the following statements are valid: S1=S2 (OR) S2=S1.

Ex:
```
struct student
{
    int id;
    char name[50];
    float marks;
};
struct student S1={135, "RAM", 90.00};
struct student S2;
```

→ The statement S2=S1 copies the content of S1 to S2.

∴ Now S2.id = 135
S2.name = RAM
S2.marks = 90.00

2. COMPARING STRUCTURE VARIABLES:-

⇨ It is not possible to compare two structure variables directly.

⇨ i.e. if S1 & S2 are two structure variables then S1==S2, S1!=S2 are not valid.

⇨ If we want to compare the structure variables, we may compare the individual members.

Ex:
```
S1.id == S2.id
strcmp(S1.name, S2.name)
S1.marks == S2.marks
```

NOTE: C doesn't permit any logical operations on structure variables.

Example Program:-

⇨ Write a C program to illustrate copying and comparision of structure variables.

```
#include<stdio.h>
#include<conio.h>
void main()
{
   struct student
   {
      char name[50];
      int id;
      float marks;
   };
   struct student s1={"DWARAK", 12, 80.00};
   struct student s2;
   clrscr();
   s2 = s1; /* copying */
   printf("Student2 name is: %s", s2.name);
   printf("\n Student2 id is: %d", s2.id);
   printf("\n Student2 marks: %f", s2.marks);
   if(s1.id==s2.id && s1.marks==s2.marks) /* Comparing */
   {
      printf("\n Both students are same");
   }
   else
   {
      printf("\n Students are not same");
   }
   getch();
}
```

OUTPUT:- Student2 name is: DWARAK
Student2 id is: 12
Student2 marks: 80.00
Both students are same

OPERATIONS ON INDIVIDUAL MEMBERS:-

⇨ A member with • operator along with its structure variable can be treated like a normal variable.

⇨ So we perform all the arithmetic operations on individual structure members.

Ex: Struct marks
```
{
  int m1;
  int m2;
  int m3;
};
struct marks s={90,80,75};
total = s.m1+s.m2+s.m3;
```

⇨ We can also apply increment and decrement operators to numeric type (i.e. integer type) members.

Ex:
```
s.m1++
++s.m1
--s.m1
```

⇨ We can also apply all relational operators to compare individual structure members.

Ex: If s1 & s2 are two structure variables of same type, then the following operations are valid

1. s1.age > s2.age
2. s1.m1 == s2.m1
3. s1.total != s2.total

⇨ We can also use individual structure members in expressions.

Ex:
```
Struct marks
{
  int m1;
  int m2;
  int m3;
};
struct marks s={90, 70, 45};
s.m3 = s.m3*10;
avg = (s.m1+s.m2+s.m3)/3;
```

Example program:-

⇒Write a program to read the marks obtained by a student in 3 subjects and calculates its sum and average.

```
#include<stdio.h>
#include<conio.h>
void main()
{
  struct marks
  {
    int m1;
    int m2;
    int m3;
  };
  struct marks s;
  int sum;
  float avg;
  clrscr();
  printf("Enter marks obtained by a student in 3 subjects:");
  scanf("%d %d %d",&s.m1,&s.m2,&s.m3);
  sum=s.m1+s.m2+s.m3;
  avg=sum/3;
  printf("\n Sum is: %d", sum);
  printf("\n Average is: %f", avg);
  getch();
}
```

OUTPUT:-

```
Enter marks obtained by a student in 3 subjects: 70 80 90
Sum is: 240
Average is: 80.00
```

ARRAY OF STRUCTURES:-

⇒ Array is a collection of similar data items. In the same way we can also define array of structures.

⇒ Whenever the same structure is to be applied to a group of people or items, in such situations array of structures will be defined.

⇒ In array of structures each element is of structure type.

⇒ For example, if we want to maintain information of 60 students in a class, we would be required to use 60 different structure variables from student 1 to student 60, which is not convenient.

A better approach would be to use an array of structures.

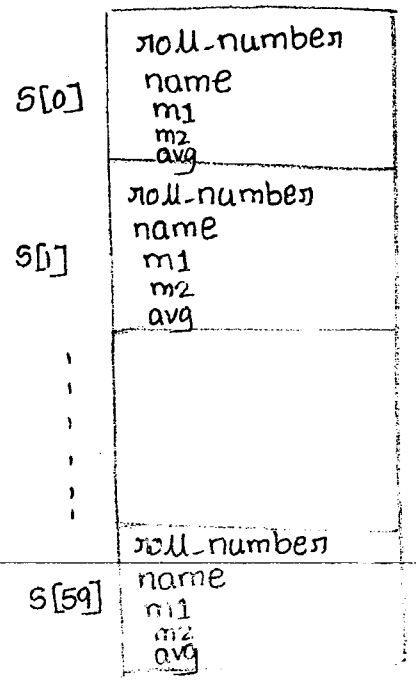
⇒ Array of structures can be declared as follows:

```
Struct student_info
{
    int roll_number;
    Char name[10];
    int m1,m2;
    float avg;
};
Struct student_info s[60];
```

→ array of structures.

In the above example, s[60] is an array of structures each containing 5 members namely roll number, name, m1, m2 and avg.

⇒ An array of structures is stored inside the memory as follows:



⇒We can use the usual array accessing methods to access individual elements and then member operator to access members.

For example, s[0].name represents first student name

s[4].avg represents 42nd student avg marks and so on..

Example program:-

```
/* Write a C-program to accept roll number, name and marks obtained
   in 3 subjects of 3 students in a class and display roll-number,
   name, marks in 3 subjects and average marks */
#include<stdio.h>
#include<conio.h>
void main()
{
    struct student
    {
       int roll_no;
       char name[30];
       int sub1, sub2, sub3;
       float avg;
    };
    struct student s[3]; /* array of structures */
    int i;
    clrscr();
    printf("Enter roll number, name, marks in 3 subjects for 3 stude");
    for(i=0;i<3;i++)
    {
       scanf("%d %s %d %d %d", &s[i].roll_no, &s[i].name,
                               &s[i].sub1, &s[i].sub2, &s[i].sub3);
    }
    /* Calculating average marks */
    for(i=0;i<3;i++)
    {
       s[i].avg = (s[i].sub1 + s[i].sub2+ s[i].sub3)/3;
    }
```

```
printf("\n Roll Number \t Name \t sub1 \t sub2 \t sub3 \t avg");
 for(i=0; i<3; i++)
 {
   printf("%d %s %d %d %d %f", s[i].roll_no, s[i].name, s[i].sub1,
                                s[i].sub2, s[i].sub3, s[i].avg );
 }
 getch();
}
```

Input:

Enter Roll number, name, marks in 3 subjects for 3 students

111	Vinod kumar	90	80	95
121	Usha	80	70	80
131	Uma	90	95	85

Output:

Roll Number	Name	sub1	sub2	sub3	avg
111	Vinod kumar	90	80	95	88.33
121	Usha	80	70	80	76.66
131	Uma	90	95	85	90.00

Explanation:-

In the above program array of structures is used to store the details of 3 students in a class.

ARRAYS WITHIN STRUCTURES:-

⇒ C allows arrays as structure members. i.e. we can use single-dimensional or multi-dimensional arrays of type int, float or char.

Ex:
```
struct student
{
    int number;
    char name[30];
    float subject[6];
};
```
→ arrays within structures

Here, the member subject contains 6 elements, subject[0], subject[1], subject[3], subject[4], subject[5]. These elements can be accessed by using appropriate subscripts.

For example S.subject[1] → represents marks obtained in 2nd subject

S.subject[5] → represents marks obtained in 6th subject

Example program:-

/* Define a structure student which contains Roll-number and marks obtained by a student in 3 subjects. Use array to represent marks. Calculate and display total marks */

```
#include<stdio.h>
#include<conio.h>
void main()
{
   struct student
   {
      int roll_no;
      int sub[3];
      int total;
   };
   struct student S;
   int i;
   printf("Enter Student Roll-Number:");
   scanf("%d", &S.roll_no);
   printf("Marks obtained in 3 subjects:");
   for(i=0; i<3; i++)
   scanf("%d", &S.sub[i]);
   S.total=0;
   for(i=0; i<3; i++)
   {
      S.total=S.total+S.sub[i];
   }
   printf("Total marks=%d", S.total);
   getch();
}
```

STRUCTURES WITHIN STRUCTURES:-

⇒ Structures within structures means nesting of structures.

⇒ Nesting of structures is allowed in C.

⇒ For example consider the following structure to store a ~~person~~ employee details.

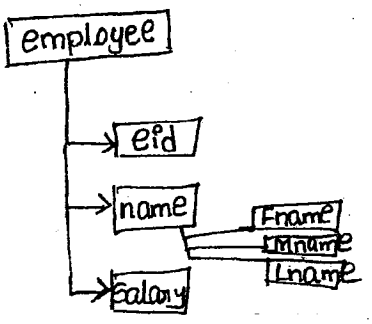
```
Struct employee
{
    int eid;
    Char Fname[20];
    Char Mname[30];
    Char Lname[20];
    float salary;
}e;
```

The above structure defines employee id, First name, middle name, Last name and salary. We can group all the members related to name together and declare them as a sub-structure as follows:

```
Struct employee
{
    int eid;
    Struct ename
    {
        Char Fname[20];
        Char Mname[30];
        Char Lname[20];
    }name;
    float salary;
}e;
```

Outer Structure

Substructure (or) Inner-Structure.



Here the structure employee contains name as its member, which itself is a structure with 3 members.

⇒ Accessing inner structure members:

outer-structure variable . inner-structure variable . inner structure member

⇒ The inner structure members can be accessed as follows:

```
e.name.Fname
e.name.Mname
e.name.Lname
```

⇒ An inner structure can have more than one variable.

⇒ We can also use tagname (or) structure name to declare inner structure.

Ex:
```
struct ename
{
    char Fname[20];
    char Mname[30];
    char Lname[30];
};
struct employee
{
    int eid;
    struct ename name; → /* we are using structure name to declare inner structure */
    float salary;
};
```

⇒ It is also possible to nest more than one structure.

Ex:
```
struct employee
{
    struct ename name;
    struct date dob;
    struct address address_part;
    ----------
    ----------
};
```
⇒ sub structures (or) inner structures.

Note: C permits nesting upto 15 levels.

C99 allows 63 levels of nesting

Example program:-

```
/* Write a program to enter full name and date of birth of an employee
   person and display the same. Use nested structures */
#include<stdio.h>
#include<conio.h>
void main()
{
   struct employee
   {
      struct ename
      {
         char Fname[20];
         char Mname[30];
         char Lname[30];
      }name;
      struct dob
      {
         int date;
         int month;
         int year;
      }d;
   };
   struct employee e;
   clrscr();
   printf("Enter name (First name, middle name and Last name):");
   scanf("%s %s %s",&e.name.Fname,&e.name.Mname,&e.name.Lname);
   printf("Enter Date of birth (Date, month, year):");
   scanf("%d %d %d",&e.d.date,&e.d.month,&e.d.year);
   printf("Name: %s %s %s",e.name.Fname,e.name.Mname,e.name.Lname);
   printf("\n Date of Birth: %d %d %d",e.d.date,e.d.month,e.d.year);
   getch();
}
```

employee Full name

Date of birth

Explanation:-

In the above program employee name and date of birth i.e dob are declared as sub structures.

Input:-

Enter name (First name, middle name, Last name): Ram Sham Pande

Enter date of birth (Date, month, year): 1 1 1989.

output:-

Name: Ram Sham Pande

Date of Birth: 1 1 1989

DIFFERENCE BETWEEN STRUCTURE and ARRAY:-

ARRAY

1. Array is a collection of similar data items

Ex:
```
int a[100];
float m[10];
```

2. Arrays can be used to represent list of similar data items.

3. To copy elements of one array to another array of same datatype, elements are copied one by one. It is not possible to copy all elements at a time.

Ex:
```
int a[3]={1,2,3};
int b[3];
b[0]=a[0]
b[1]=a[1]
b[2]=a[2]
```
} elements are copied one by one.

4. Uses of arrays:
To represent
→ List of employees in a company
→ List of products
→ List of student marks etc

STRUCTURE

1. Structure is a collection of different data items.

Ex:
```
Struct student
{
    int number;
    Char name[20];
    float marks;
};
```

2. Structures can be used to handle mixed data items (or) complex data

3. In structures, it is possible to copy the entire content of one structure variable to another structure variable of same type at a time using '=' operator

Ex:
```
Struct student
{
    int number;
    char name[20];
    float marks
};
Struct student S1={1,"usha",90.00};
Struct student S2;
S2=S1; → copying entire structure at a time.
```

4. Structures can be used to represent:
→ Customer details: name, phone number, city
→ Address : door no, street, city
→ Book details : title, author, price etc.

POINTER TO STRUCTURES:-

Pointer is a variable, which stores the address of another variable. The variable may be of any data type i.e. int, float, double etc.

In the same way we can also define a pointer to structure:

We can have a pointer pointing to a structure. Such pointers are known as "structure pointers".

Ex:
```
struct book
{
   char name[30];
   char author[20];
   int pages;
};
struct book *ptr; → ptr is a pointer to structure book.
struct book b; ⟶ structure variable.
```

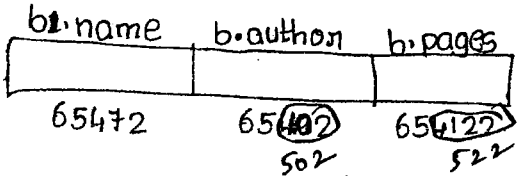
⇒ Address of structure variable can be ~~use~~ assigned to structure pointer as follows:

```
ptr = &b;
```

⇒ Now we can use "→" operator to access structure members using pointer.

Ex:
```
ptr->name        (*ptr).name
ptr->author  (OR) (*ptr).author
ptr->pages       (*ptr).pages
```

⇒ The arrangement of structure variable and pointer to structure is shown below:



ptr = &b ⇒ ptr: 65472 → address of structure member.

65524

Example program:-

/* Write a program to define a structure book. Display book details using pointer to structure */

```
#include<stdio.h>
#include<conio.h>
Void main()
{
    Struct book
    {
       Char name[25];
       Char author[30];
       Int pages;
    };
    Struct book b={"CDS", "Balaguruswamy", 300};
    Struct book *ptr;
    Clrscr();
    ptr = &b;
    Printf("Book name : %s", ptr->name);
    Printf("\n Author : %s", ptr->author);
    Printf("\n Pages : %d", ptr->pages);
    getch();
}
```

Output: Book name: CDS
Author: Balaguruswamy
Pages: 300

STRUCTURES AND FUNCTIONS:-

⇨ Like any other variables, structure variables can also be passed as arguments to functions

⇨ There are 3 methods to pass structure members as arguments to functions.

1. Passing each member of the structure as an actual arguments of the function call.
2. Passing a copy of entire structure to the called function.
3. Passing address of entire structure to the called function.

First method : Passing each member of the structure as an actual argume

⇨ We can pass each member of the structure as an actual argument to the function call.

→ The actual arguments are then treated like ordinary variables.

Syntax: functionname(structure_members);

Ex:
```
struct book
{
    char title[10];
    char author[10];
    float price;
};
struct book b={"COS","Balaguru",30.000};
display(b.title,b.author,b.price);
```
↳ Here we are passing each member of structure as an argument.

Disadvantage:-

⇨ This method is inefficient when the structure size is large.

Example Program:-

⇨ Write a program to pass structure members as arguments to a user-defined function and display the content of structure.

```
#include<stdio.h>
#include<conio.h>
Struct book
{
    char name[10];
    char author[20];
    float price;
    int pages;
};
Void main()
{
   Struct book b={"CDS","Balaguruswamy",300.00,250};
   Void display(char t[10],char a[20], float p, int pages);
   clrscr();
   printf("Book details are:");
   display(b.name, b.author, b.price, b.pages); →/* we are passing each
                                                    member as argument */
   getch();
}
void display(char t[10], char a[20], float p, int pages)
{
    printf("\n Book name is: %s", t);
    printf("\n Author is : %s", a);
    printf("\n Price : %f", p);
    printf("\n Number of pages : %d", pages);
}
```

Explanation:-

In the above example, the structure book contains 4 members namely name, author, price and pages. we are defining one structure variable b. The function display is called by passing each member of the structure as arguments i.e. display(b.name, b.author, b.price, b.pages);

SECOND METHOD: Passing entire structure to the function

⇨ We can pass a copy of entire structure to the called function.

⇨ Since the function is working on a copy of the structure, any changes made to structure members within the function will not reflect the original structure.

Syntax: function_name(structure_variable);

⇨ The function definition should take the following form:

```
return_type function_name(struct structure_name structure_variable)
{
    statement 1;
    statement 2;
    -------
    -------
    return (expression);
}
```

⇨ The function declaration is similar to function header.

⇨ The structure variable used in the function call and in the function definition must be of same struct type.

⇨ The return statement is necessary only when the function is returning some data back to the calling function.

⇨ When a function returns a structure, it must be assigned to structure of identical type in the calling function.

Ex:
```
struct book
{
    char title[20];
    char author[30];
    float price;
    int pages;
};
struct book b={"CDS", "Balaguruswamy", 300.00};
```

→ We can pass above structure to a user-defined function as follows:

```
display(b);
```

→ We are passing entire structure.

Example Program:-

⇨ Write a program to pass entire structure to a user-defined function.

```
/* Passing entire structure to the function */
#include<stdio.h>
#include<conio.h>
struct book
{
    char title[10];
    char author[20];
    float price;
    int pages;
};
void main()
{
   struct book b={"CDS", "Balaguruswamy", 300.00, 250};
   void display(struct book b1); /* Function Declaration */
   clrscr();
   printf("Book details are:");
   display(b); --> /* Passing structure variable */
   getch();
}
void display(struct book b1)  --> /* Function definition */
{
    printf("\n Book name is: %s", b1.title);
    printf("\n Author is: %s", b1.author);
    printf("\n Price : %f", b1.price);
    printf("\n Number of pages: %d", b1.pages);
}
```

THIRD METHOD: Passing address of a structure as an argument

⇨ Third method is to pass address location of the structure as an argument to the called function.

⇨ When we are passing address of a structure as an argument, the receiving parameter must be a pointer to the structure. The function can access the structure members using "->" operator.

⇨ Any changes made to the structure members within the function will reflect the original array.

⇨ This method is more efficient as compared to the second method.

Syntax: `function_name(&structure_variable);`

Example program:-

⇨ Write a program to pass address of a structure as an argument to a user-defined function and display the contents of the structure.

```
#include<stdio.h>
struct book
{
    char title[10];
    char author[20];
    float price;
    int pages;
};
void main()
{
   struct book b={"CDS", "Balaguruswamy", 300.00, 250};
   void display(struct book *b1);
   clrscr();
   display(&b); → /* passing address of a structure variable */
   getch();
}
void display(struct book *b1)  ——→ /* The receiving parameter must be a pointer */
{
    printf("Book name is: %s", b1->title);
    printf("Author is: %s", b1->author);
    printf("Price: %f", b1->price);
    printf("Number of pages: %d", b1->pages);
}
```

function returning structure variable:

Ex:

```
struct product
{
    char name[20];
    int no;
};
struct product items;
struct product fun(void);

main()
{
    clrscr();
    items = fun();
    printf("-product Name is %s\n", items.name);
    printf("No. & product is %d", items.no);
    getch();
}
struct product fun()
{
    struct product items1;
    printf("Enter product name:");
    gets(items1.name);
    printf("Enter no. & products:");
    scanf("%d", &items1.no);
    return(items1);
}
```

UNIONS:-

Union is a collection of heterogeneous elements. ie. it is a collection of different data items.

⇒ Unions are similar to structures except in terms of storage.

⇒ In structure each member has its own memory location, whereas in unions all the members use same memory location.

⇒ Unions may contain many members, but it can handle only one member at a time.

Defining & declaring unions:-

⇒ Unions can be declared using the keyword union as follows:

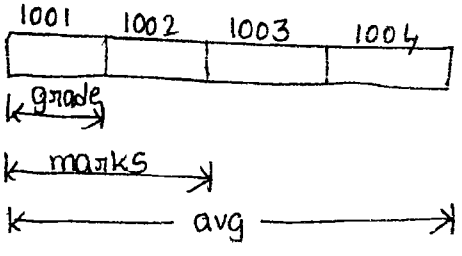
Syntax:
```
union union_name
{
    datatype member1;
    datatype member2;
    - - - - - - - - - -
    - - - - - - - - - -
};
```

Ex:
```
union results
{
    int marks;
    char grade;
    float avg;
};
```

⇒ Size of union is equal to the size of its largest member.

In the above example number of bytes reserved in the memory for the union results is 4 bytes [i.e. size of its largest member float].

⇒ All the union members shares the same memory location.



⇒ Unions can access only one member at a time because there is only one memory location.

⇒ Accessing union members is equal to accessing structure members.

Ex: r.marks,
r.grade
r.avg

⇒ Unions may be used in all case where the structure is allowed.

⇒ During accessing we should make sure that we are accessing the member whose value is currently stored in the memory.

Example program:-

⇒ Write a program that uses sizeof operator to differentiate between structure and union.

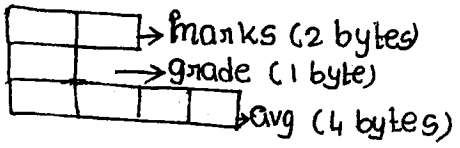
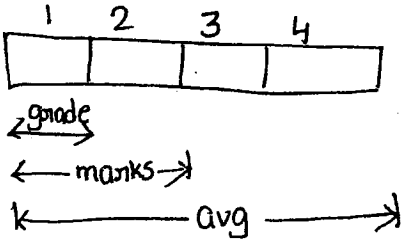
```
#include<stdio.h>
void main()
{
    union result
    {
        int marks;
        char grade;
        float avg;
    }r1;
    struct res
    {
        int marks;
        char grade;
        float avg;
    }r2;
    clrscr();
    printf("Size of union: %d bytes", sizeof(r1));
    printf("\n Size of structure: %d bytes", sizeof(r2));
    getch();
}
```

Output: Size of union: 4 bytes
Size of structure: 7 bytes

DIFFERENCES BETWEEN STRUCTURE and UNION:-

Structure	Union.

1. To define a structure the keyword 'struct' is used.
Ex: struct result
{
int marks;
char grade;
float avg;
};
	1. To define a union, the keyword union is used.
	Ex: union result
	{
	int marks;
	char grade;
	float avg;
	}u;

2. In structure each member has its own memory location 	2. In unions, all the members shares the same memory location 
3. The size of structure is equal to the sum of size of all its members. Ex: sizeof(result) = Sizeof(marks) + Sizeof(grade) + Sizeof(avg) = 2+1+4 = 7 bytes	3. The size of union is equal to the size of its largest member Ex: sizeof(result) = sizeof(avg) = 4 bytes
4. We can access all structure members at a time.	4. In unions, we can access only one member at a time.

BIT FIELDS:-

We know that integer numbers requires 16 bits to store data. There are some situations, where data items requires less than 16 bits space. In such cases we waste the memory space.

~~To overcome this C-supports a concept known as "bit-fields".~~

⇒ Using bitfields we can tell the compiler to allocate only specified number of bits.

Bit field:- A bit field is a set of adjacent bits whose size can be from 1 to 16 bits in length.

⇒ The name and sizeof bitfields can be defined using a structure.

Defining bitfields:-

Syntax:
```
struct tag_name
{
   data-type name1 : bit_length;
   data-type name2 : bit_length;
   - - - - - - - - - - - -
   - - - - - - - - - - - -
   data-type nameN : bit_length;
};
```

where data-type is either int (or) unsigned int (or) signed int.

bit_length is the number of bits used for specified name.

⇒ The bit_length is decided by the range of value to be stored. The largest value that can be stored is (2^{n-1}), where n is bit_length.

Ex:
```
struct personal
{
   unsigned age:3
   int result:1
   unsigned count:4
} p;
```

⇒ The range of values is $2^n - 1$, as follows:

Bit field	Bit_length	Range of values
age	3	0 to 2^3-1
result	1	0 to 2^1-1
count	4	0 to 2^4-1

2^4-1

The above example defines a variable p with 3 bit fields.

⇒ Once bit fields are defined, they can be accessed using '.' operator.

Ex: p.age, p.result, p.count.

⇒ We can't use scanf to read values into a bitfield.

⇒ We can use assignment operator to assign values to the bitfields

Ex: p.age = 17
p.result = 1

⇒ Bitfields can be used in expressions

Ex: p.age = p.age * 2
if(p.age == 18)

⇒ Bitfields can't be arrays.

⇒ We can't take address of a bitfield variable.

⇒ We can't use pointer to access the bit fields.

⇒ There can be unnamed bitfields with declared size.

Ex: unsigned : bit_length.

⇒ It is possible to combine normal structure elements with bitfield elements.

Ex:
```
struct personal
{
    char name[10];
    float salary;          } → Normal structure members
    int age : 3;
    int result : 1;
    unsigned count : 4;    } → Bit-fields.
};
```

Ex: Write a C program to display the examination result of student using bit field.

```
#define pass 1
#define fail 0
#define A 0
#define B 2
#define C 3

main()
{
    struct stud
    {
        char name[20];
        unsigned result : 1;
        unsigned grade : 2;
    } s;

    strcpy(s.name, "rachu");
    s.result = PASS;
    s.grade = C;
    printf("Name : %s", s.name);
    printf("result : %d", s.result);
    printf("Grade : %d", s.grade);
}
```

OM VINAYAKA
OM SAI RAM