Programming in C

code: 18K5MELM1

Semester : V̄

Major Based Elective.

Unit : 3

Decision making and looping - while, DO and FOR statement.

(Chapter 6 omitting Section 6.5)

Unit 4 :

Arrays - one dimensional, two dimensional and multi dimensional groups - Structures - definition giving Values to members - Initialization - Comparison - Arrays of Structures - Arrays within Structures - structures within structures.

(Chapter 7 & 10 Section 10.1 - 10.8)

Unit 5 :

User defined functions - The form of c functions, return Values & their types - calling a function - category of functions - No arguments and No return Values - arguments but no return values argument with return Values - Nesting of functions - Recursion (chapter 9 - Section 9.1 to 9.13)

Text Book:

Programming in ANSI C - E. Balagurusamy, II Edition 2001

Decision Making And Looping

It is possible to execute a segment of a program repeatedly by introducing a counter and later testing it using the if statement. While this method is quite satisfactory for all practical purposes we need to initialize and increment a counter and test its value at an appropriate place in the program for the completion of the loop. For example, suppose we want to calculate the sum of squares of all integers between 1 and 10. we can write a program using the if statement as follows:

```
 -- ----
 ---- ----
 sum = 0;
 n = 1;
→loop:
 sum = sum + n*n;           L
  if (n==10)                o
  goto print;               o
 else                       p
 {                                  n=10,
 n=n+1;                             end of loop
 goto Loop;
 }
 print :
 -----
```

This program does the following things:

1. Initializes the variable n.
2. Computes the square of n and adds it to sum.
3. Tests the value of n to see whether it is equal to 10 or not. If it is equal to 10, then the program prints the results
4. If n is less than 10, then it is incremented by one and the control goes back to compute the sum again.

The program evaluates the statement

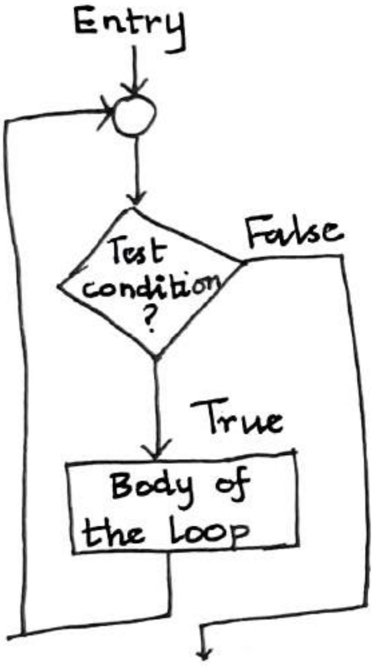
sum = sum + n * n ;

10 times. That is. the loop executed 10 times.

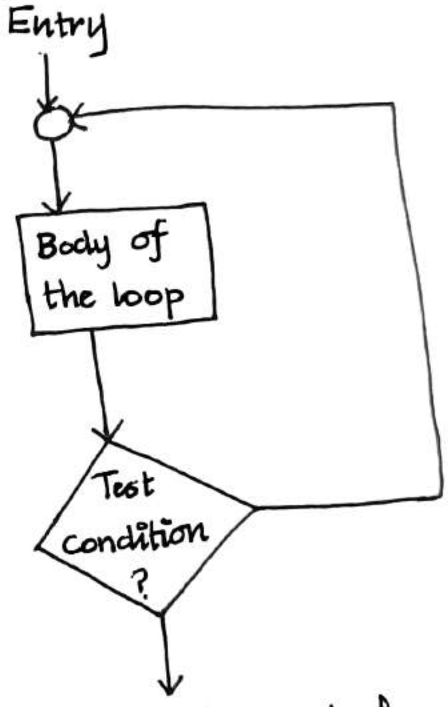
In Looping, a sequence of statements are executed until some conditions for termination of the loop are satisfied. A program loop therefore consists of two segments, one known as the body of the loop and the other known as the control statement. The control statement tests certain conditions and then directs the repeated execution of the statements contained in the body of the loop.

Depending on the position of the control statement in the loop, a control structure may be clasified either as entry-controlled loop or as the exit-controlled loop.

In the entry-controlled loop, the control conditions are tested before the start of the loop execution. If the conditions are not satisfied, then the body of the loop will not be executed. In the case of an exit-controlled loop, the test is performed at the end of the body of the loop and therefore the body is executed unconditionally for the first time.



Entry control



Exit control.

A looping process in general, would include the following four steps.

1. Setting and Initialization of a counter.
2. Execution of the statements in the loop.
3. Test for a specified condition for execution of the loop.
4. Incrementing the counter.

The test may be either to determine whether the loop has been repeated the specified number of times or to determine whether a particular condition has been met.

The C Language provides for three loop constructs for performing loop operations. They are

1. The while statement 2. The do statements. 3. The for statement

The while Statement :

The simplest of all the looping structures in C is the while statement. The basic format of the while statement

```
while (test condition)
{
 body of the loop
}
```

The while is an entry-controlled loop statement. The test condition is evaluated and if the condition is true, then the body of the loop is executed. After execution of the body, the test-condition is once again evaluated and if it is true, the body is executed once again. This process of repeated execution of the body continues until the test-condition finally becomes false and the control is transferred out of the loop. On exit, the program continues with the statement immediately after the body of the loop. The body of the loop may have one or more statements. The braces are needed only if the body contains two or more statements.

For eg :

```
-------
Sum = 0;
 n = 1;
while (n <= 10)
{
 sum = sum + n * n;
 n = n+1;
}
 printf ("sum = %d\n", sum);
 - - - - - - -
 - - - - - - -
```

The body of the loop is executed 10 times for n = 1, 2, ...10 each time adding the square of the value of n, which is incremented inside the loop. The test condition may also be written as n < 11; the result would be the same.

The Do Statement :

The while loop construct that we have discussed, a test of condition before the loop is executed. Therefore, the body of the loop may not be executed at all if the condition is not satisfied at the very first attempt. On some occassions it might be necessary to execute the body of the loop before the test is performed. Such situations can be handled with the help of the do statement.

```
do
{
 body of the loop
}
while (test-condition);
```

On reaching the do statement, the program proceeds to evaluate the body of the loop first. At the end of the loop, the test-condition in the while statement is evaluated. If the condition is true, the program continues to evaluate the body of the loop once again. This process continues as long as the condition is true: when the condition become false, the loop will be terminated and the control goes to the statement that appears immediately after the while statement. Since the test-condition is evaluated at the bottom of the loop, the do.... while construct provides an exit-controlled loop and therefore the body of the loop is always executed at least once.

```
- - - - - -
do
 {
 printf("Input 'a number \n");
 number = getnum();
 }
 while (number >0);
 - - - - - - - -
```

This segment of a program reads a number from the keyboard until a zero or a negative number is keyed in.

The test conditions may have compound relations as well. For instance, the statement

while (number > 0 && number<100); in this example would cause the loop to be executed as long as the number keyed in lies between 0 and 100.

The FOR Statement:

The for loop is another entry-controlled loop that provides a more concise loop control structure. The general form of the for loop is

```
for ( initialization ; test-condition ; increment)
{
   body of the loop
}
```

The execution of the for statement is as follows:

1. Initialization of the control variable is done first, using assignment statement such as i = 1 and count = 0. The variable i and count are known as loop-control variables.

2. The value of the control variable is tested using the test-condition. The test-condition is a relational expression such as i < 10. that determines when the loop will exit.

If the condition is true, the body of the loop is executed; otherwise the loop is terminated and the execution continues with the statement that immediately follows the loop.

3. when the body of the loop is executed, the control is transferred back to the for statement after evaluating the last statement in the loop. Now, the control variable is incremented using an assignment statement such as i= i+1 and the new value of the ~~condition is satisfied~~ control variable is again tested to see whether it satisfies the loop condition. If the condition is satisfied, the body of the loop is again executed. This process continues till the value of the control variable fails to satisfy the test condition. consider the following segment of a program:

```
for (x = 0 ; x <= 9 ; x = x+1)
{
 printf ("%d ", x);
}
 printf ("\n");
```

This loop is executed 10 times and prints the digits 0 to 9 in one line. The three sections enclosed within parentheses must be seperated by semicolons. Note that there is no semicolon at the end of the increment section, x = x+1.

The for statement allows for negative increments. For example, the loop discussed above can be written as follows:

```
for (x = 9 ; x >= 0; x = x-1)
 printf ("%d", x);
 printf ("\n");
```

This loop is also executed 10 times, but the output would be from 9 to 0. instead of 0 to 9. Since the conditional test is always performed at the beginning of the loop, the body of the loop may not be executed at all, if the condition fails at the start. For example,

```
for(x = 9 ; x<9; x = x-1)
 printf ("%d ", x);
```

will never be executed because the Test condition fails at the very beginning itself.

One of the important points about the for loop is that all the three actions, namely initialization, testing, and incrementing, are placed in the for statement itself, thus making them visible to the programmers and users, in one place

Example :

The program for loop to print the "Powers of 2" table for the power 0 to 20, both positive and negative.

The program evaluates the value

$$p = 2^n$$

successively by multiplying 2 by itself n times.

$$q = 2^{-n} = 1/p$$

Note that we have declared p as a long int and q as a double.

Additional Features of for Loop :

The for loop in C has several capabilities that are not found in other loop constructs. For example, more than one variable can be initialized at a time in the for statement. The statements

```
 p=1;
for(n=0;n<17;++n)   can be rewritten as

 for (p=1,n=0;n<17;++n)
```

Notice. that the initialization section has two parts p=1 and n=1 Seperated by a comma.

Like the initialization section, the increment section may also have more than one part. For example, the loop

```
for (n=1, m=50; n<m; n =n+1, m=m-1)
{
 p=m/n;
 printf("%d %d %d\n",n,m,p);
}
```

is perfectly valid.

The third feature is that the test-condition may have any compound relation and the testing need not be limited only to the loop control variable. Consider the example below.

```
Sum=0;
for(i=1;i<20 && Sum<100;++i)
{
 Sum=Sum+i;
 printf("%d %d\n", Sum);
}
```

The loop uses a compound test condition with the control variable i and external variable sum. The loop is executed as long as both the conditions i<20 and sum <100 are true. The sum is evaluated inside the Loop.

It is also permissible to use expressions in the assignment statements of initialization and increment Sections. For example a statement of the type for(x = (m+n)/2; x>0; x=x/2) is perfectly valid.

Another unique aspect of for loop is that one or more section can be omitted

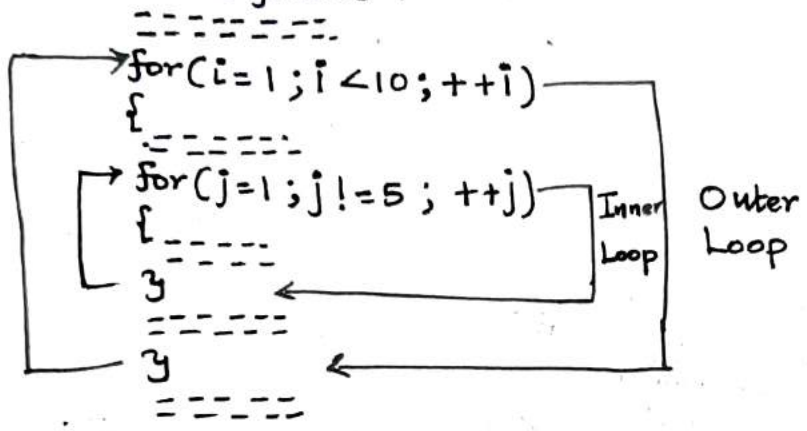
```
--------
m=5;
for(;m!=100;)
{ printf("%d\n", m);
  m=m+5;
}
--------
```

Both the initialization and increment Sections are omitted in the for statement.

The initialization has been done before the for statement and the control variable is incremented inside the loop. In such cases, the sections are left blank. However, the semicolons separating the sections must remain. If the test-condition is not present; the for statement sets up an infinite loop. such loops can be broken using break or goto statements in the loop.

Nesting of for Loops:

Nesting of loops, that is, one for statement within another for statement, is allowed in C. For example, two loops can be nested as follows:



```
--------
for(i=1;i<10;++i)
{------
------
for(j=1;j!=5; ++j)
{-----
-----
}
------
}
------
```

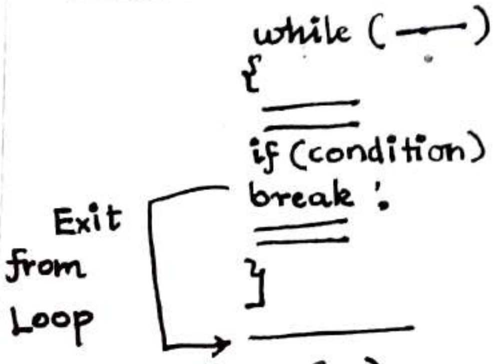
The nesting may continue upto 15 levels in ANSI C; many compilers allow more. The loops should be properly indented so as to enable the reader to easily determine which statements are contained within each for statement.

Jumping Out of a Loop:

when the break statement is encountered inside a loop. the loop is immediately exited and the program continues with the statement immediately following the loop. when the loops are nested, the break would only exit from the loop containing it. That is, the break will exit only a single loop.

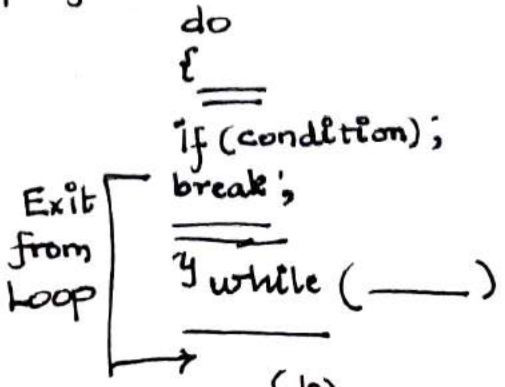
Since a goto statement can transfer the control to any place in a program, it is useful to provide branching within a loop Another important use of goto is to exit from deeply nested loops when an error occurs. A simple break statement would not work here.

Break statement in a C program:



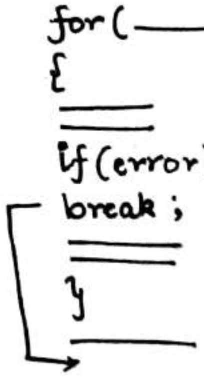
```
while (——)
{
——
——
if(condition)
break;
——
}
——
```

(a)

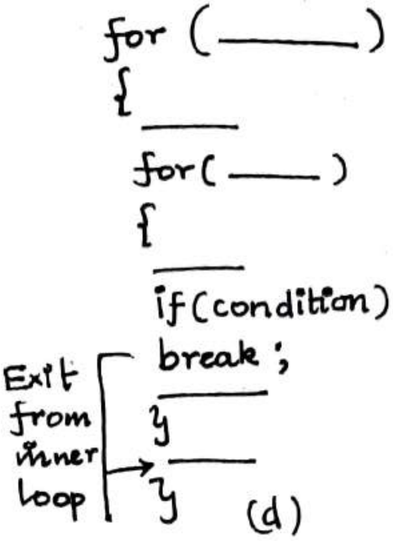


```
do
{
——
if(condition);
break;
——
——
} while (——)
——
```

(b)

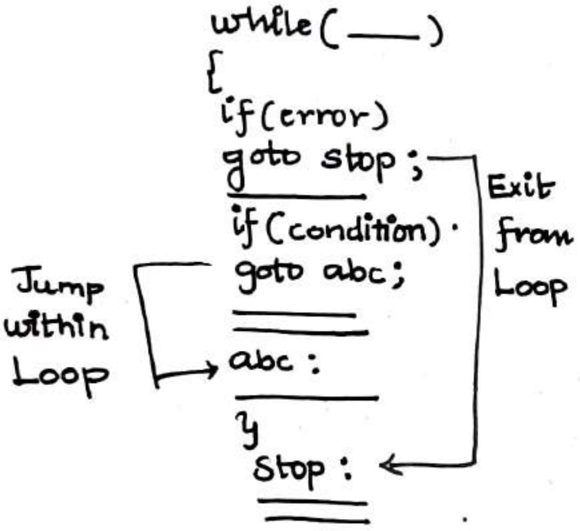


(c)

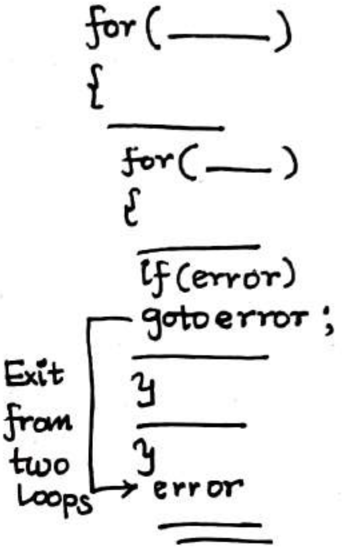


(d)

Goto statement:



(a)



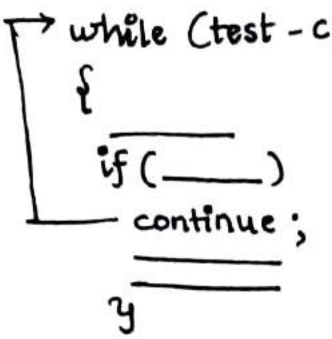
(b)

Skipping a part of a Loop:

Like the break statement, c supports another similar statement called the continue statement. However, unlike the break which causes the loop to be determinated, the continue, as the name implies, causes the loop to be contimued with the next iteration after skipping any statements in between. The continue statement tells the compiler, "SKIP THE FOLLOWING STATEMENTS AND CONTINUE WITH THE NEXT ITERATION". The format of the continue statement is simply

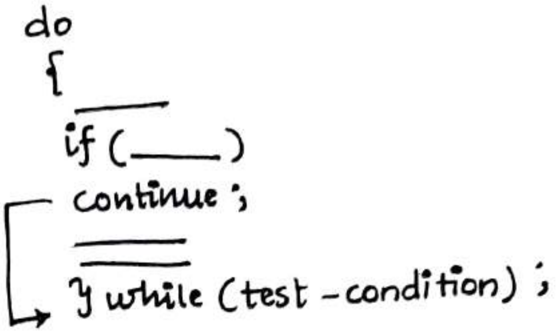
$\boxed{\text{Continue ;}}$

The use of the continue statement in loops. In while and do loops, continue causes the control to go directly to the test-condition and then to continue the iteration process. In the case of forloop, the increment section of the loop is executed before the test-condition is evaluated.



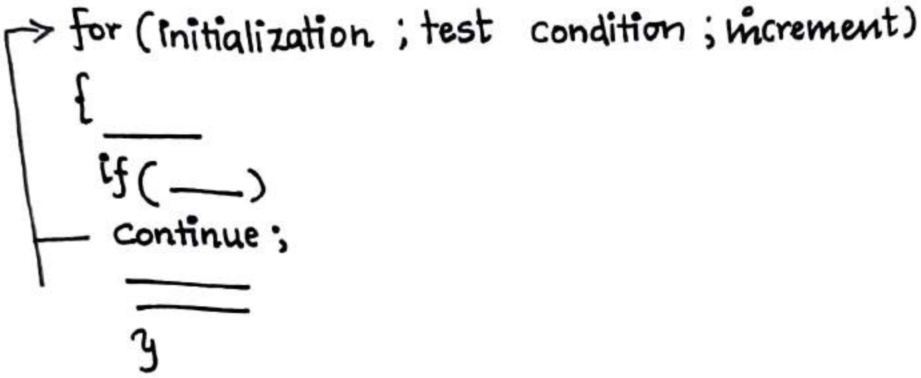
```
while (test-condition)
{
  ______
  if (______)
   continue ;
   ______
   ______
 }
```

(a)



```
do
{ ______
  if (______)
  continue ;
  ______
  ______
} while (test-condition) ;
```

(b)



```
for (initialization ; test  condition ; increment)
{ ______
  if (______)
  continue ;
   ______
   ______
 }
```

(c)

Chapter- 7

ARRAYS

An array is a group of related data items that share a common name.

For eg: Define an array name **Salary** to represent a set of Salaries of a group of employees.

A particular value is indicated by writing a number called index number or subscipt in brackets after the array name.

For example: Salary[0]

represents the salary of the 10th employee. while the complete set of Values is referred to as an array, the individual values are called elements.

One-Dimensional Arrays:-

A list of items can be given one variable name using only one Subscipt and such a Variable is called a Single-Subscipted Variable or one-dimensional array

For eg: $A = \sum_{i=1}^{n} x_i / n$ - to calculate the average of n values of x. The subscipted Variable x_i, refers to the ith element of x.

In C, a single-subscipted variable x_i can be expressed as $x[1], x[2], x[3] \ldots x[n]$

The Subscipt can begein with number 0.

(ie) $x[0]$ is allowed.

For eg: Represent a set of five numbers say (35,40,20,57,19 by an array variable number, then we may declare the variable name as follows:

int number [5] and the computer reserves five storage location as shown below.

□ number [0]

□ number [1]

□ number [2]

□ number [3]

□ number [4]

The values to the array elements can be assigned as follows:

```
number [0] = 35;
number [1] = 40;
number [2] = 20;
number [3] = 57
number [4] = 19
```

Another eg:

```
a = number [0] + 10;
number [4] = number [0] + number [2];
number [2] = X [5] + y [10];
Value [6] = number [i] * 3
```

Declaration of Arrays

The general form of array declaration is

type variable-name [size];

The type specifies the type of element that will be contained in the array, such as int, float or char and the size indicates the maximum number of elements that can be stored inside the array.

For eg: float height [50]

declares the height to be an array containing 50 real elements.

The size in a character string represents the maximum number of characters that the string can hold.

For eg: char name [10] - declares the name as a character ~~that~~ array (string) that can hold a maximum of 10 characters

Initialization of Arrays

The general form of Array:

```
Static type array-name [size] = {list of values}
```

The values in the list are separated by commas.

For eg: static int number [3] = {0,0,0};
will declare the variable number as an array of size 3 and will assign zero to each element.

If the number of values in the list is less than the number of elements, then only that many elts will be initialized. The remaining elements will be set to zero automatically.

For eg: static float total [5] = {0.0, 15.75, -10};
will initialize the first three elements to 0.0, 15.75, -10.0 and the remaining two elements to zero.

Initialization of arrays in C suffers two drawbacks:

1. There is no convenient way to initialize only selected elements.

2. There is no shortcut method for initiazing a large number of arrays elements like the one available in Fortran.

Two dimensional Arrays.

The general form is

type array-name [row_size] [column_size]

For eg: The value of sales of three items by four salesgirl

	item 1	item 2	item 3
Salesgirl 1	310	275	365
Salesgirl 2	210	190	325
Salesgirl 3	405	235	240
Salesgirl 4	260	300	380

Two dimensional arrays are stored in memory as shown below.

	Column0 ↓	column1 ↓	Column 2 ↓
Row0 →	[0][0] 310	[0][1] 275	[0][2] 365
Row 1 →	[1][0] 210	[1][1] 190	[1][2] 325
Row 2 →	[2][0] 405	[2][1] 235	[2][2] 240
Row 3 →	[3][0] 266	[3][1] 300	[3][2] 380

Initializing two-dimensional Arrays:

For eg: static int table [2][3] = {0,0,0,1,1,1};

initializes the elements of the first row to zero and the second row to one. The intialization is done row by row.

The above statement can be equivalently written as static int tables [2][3] = {{0,0,0}, {1,1,1}}; by surrounding the elements of each row by braces.

If the values are missing in an intializer, they are automatically seti to zero.

For eg: static int table [2][3] = {
{1,1},
{2}
};

will initialize the first two elements of the first row to one, the first element of the second row to two and all the other elements to zero.

Multidimensional Arrays:

The general form of a multidimensional array is

type array-name [s1][s2][s3].... [sm];

where S_j is the size of the i^{th} dimension.

Some example are:

```
int survey [3][5][12];
float table [5][4][5][3];
```

Survey is a three dimensional array declared to contain 180 integer type elements. Similarly table is a four dimensional array containing 300 eleme of floating-point-type.

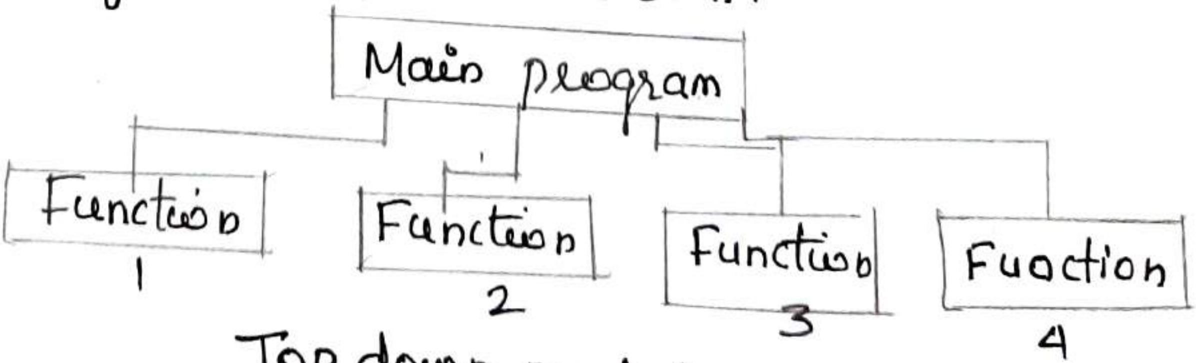
Chapter - 9

USER - DEFINED FUNCTIONS

Need for user - Defined functions:

Number of advantages:

1. It facilitates top-down moduler programming. In this programming style, the high level logic of the overall problem is solved first while the lower level function are adrresed later.
2. The length of the Source program can be reduced by using functions at appropriate places.
3. As mentioned earlier, it is easy to locate and isolate a faulty functions for further investig
4. A function may be used by many other program This means that a C programmer can build on what others have already done, instead of Starting over, from scratch.



Top down modular programming using functions

A Multi-Function Program

- * A function is a self-contained block of code that performs a particular task.

- * Once a function has been designed and packed, It can be treated as a black-box that takes some data from the main program and return its Value.

Every C program can be designed using a collection of these black boxes.

Consider a set of statements

```
Printline ( )
{ int i
  for(i=1; i<40; i++)
     Printf("— ")
  Printf("\n");
}
```

The above set of statements defines a function called printline which could print a line of 39-character length.

This function can be used in a program as follows.

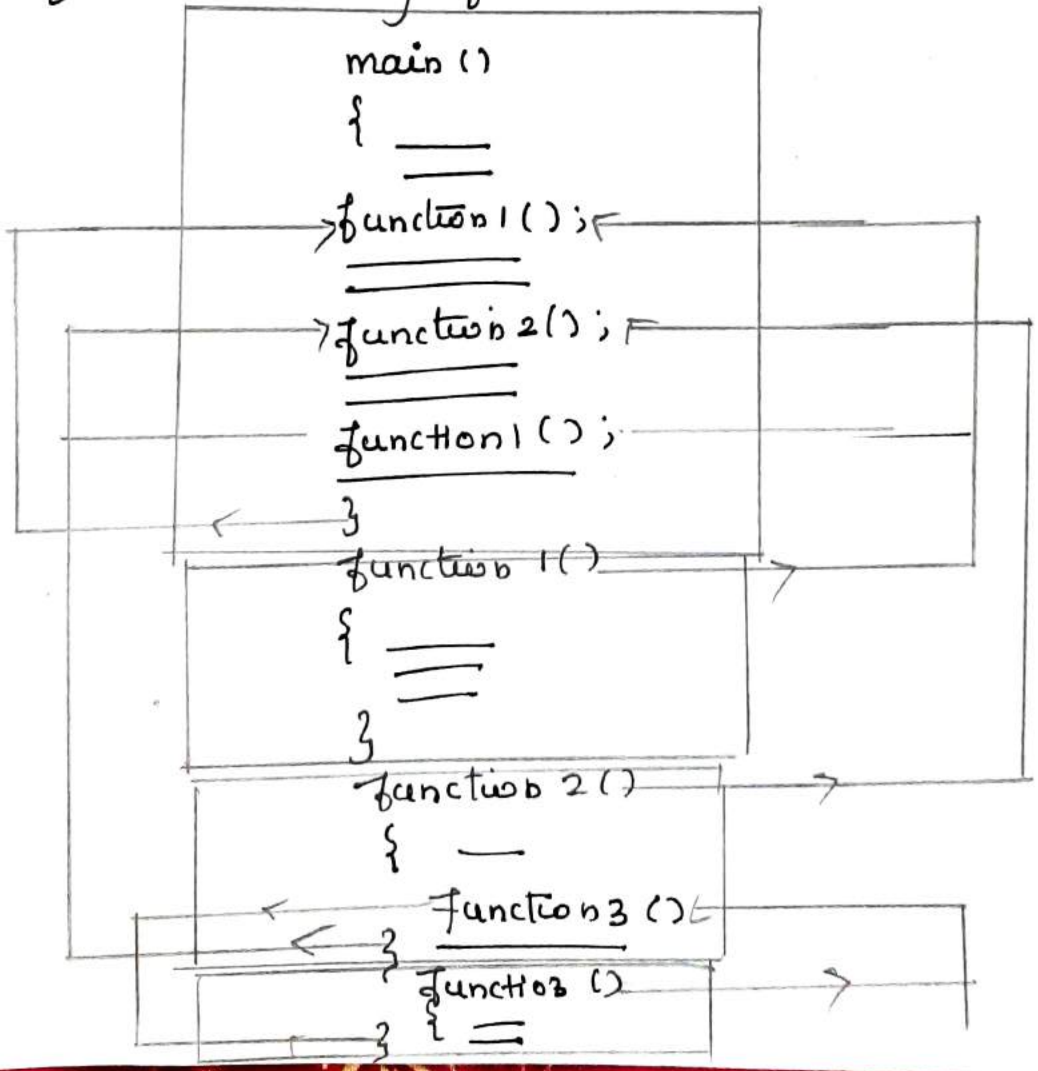
```
main ( )
{ printline ( ) ;
  Printf ("This illustrates the use of
          c functions\n")
  Printline ();
}
Printline ( )
{
```

```
int i;
for(i=1; i<40, i++)
 Printf("— ");
 Printf("\n");
}
```

This program will print the following output:

```
________________________________________
This illustrates the use of c functions
________________________________________
```

- * Any function can call any other function
- * A called function can also call another function
- * A function can be called more than once.
- * The functions can be placed in any order.
- * A called function can be placed either before or after the calling function.



```
main()
{ ___
  ___
function1();
  ___
  ___
function2();
  ___
  ___
function1();
  ___
}
function1()
{ ___
  ___
  ___
}
function2()
{ ___
  function3()
}
function3()
{ ___
}
```

The form of C Functions

All functions have the form

```
function-name (argument list)
argument declaration
{
local variable declarations ;
executable statement 1 ;
executable Statement 2 ;
. . . . . . . . . . .
. . . . . . . . . . .
return (expression) ;
}
```

* All parts are not essential. Some may be absent.

For eg: argument list and its associated argument declaration parts are optional.

Function Name :

* A function must follow the same rules of formation as other variable names in C.

Arrgument List :

* The argument list contains valid variable names separated by commas.

* List must be surrounded by paranthesis.

* No semicolon follows the closing parenthesis.

* The argument variables receive values from the calling function, thus providing a means for data communication from the calling function to the called function.

Some examples of functions with argument are

```
Quadratic(a,b,c)
power(x,n)
mul(a,b)
square(y)
copy(name1,name2)
```

* All argument Variables must be declared for their types after the function header and before the opening brace of the function body.

For Eg:
```
Powe(x,n)
float x;
 int n;
 {
   .....
   .....
   ....
 }
```

Return Values and their Types:

* The return statement can take one of the following forms

```
return;
or
return(expression);
```

* The first, the plain return does not return any Value.

* It acts as the closing brace of the function.

* when a return is encountered, the control is immediately passed back to the calling function

For eg:
```
if (error)
    return;
```

* The Second form of return with an expression returns the value of the expression.

For eg:
```
mul (x,y)
int x,y;
{ int p;
  p = x*y;
 return (p);
}
```
returns the value of p which is the product of the values of x and y.

* The last two statement can be combined into one statem
```
return (x*y)
```

* A function may have more than one return statements.

For eg:
```
if (x <=0)
   return (0);
else
return (1);
```

Calling A Function:

* A function can be called by simply using the function name in a statement.

For eg:

```
    main ( )
    {
      int p ;
      P = mul (10,5 );
      printf (" %d\n", P);
    }
```

when the compiler encounters a function call, the control is transfered to the function mul(x,y)

This function is executed line by line as described and a value is returned when a return statement is encountered. This value is assigned to p

Each of the following statement is valid:

```
    Printf(" %d\n", mul(p,q));
    Y = mul (P,q) / P+q);
    if (mul (m,n) > total ) printf ("large");
```

A function cannot be used on the right side of an assignment statement.

For eg: mul (a,b) = 15; is invalid.

Category of Functions:

A function, depending on whether arguments are present or not and whether a value is returned or not, may belong to one of the following categories.

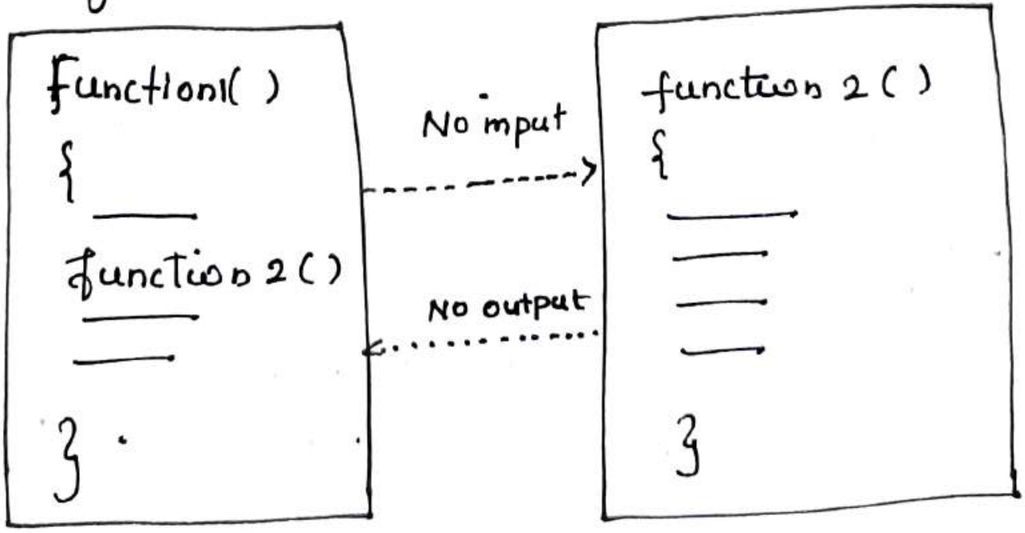
category 1 : Functions with no aruguments and no return values.

category 2 : Functions with arguments and no return values.

category 3 : Functions with arguments and returns values.

No Arguments and No Return values :

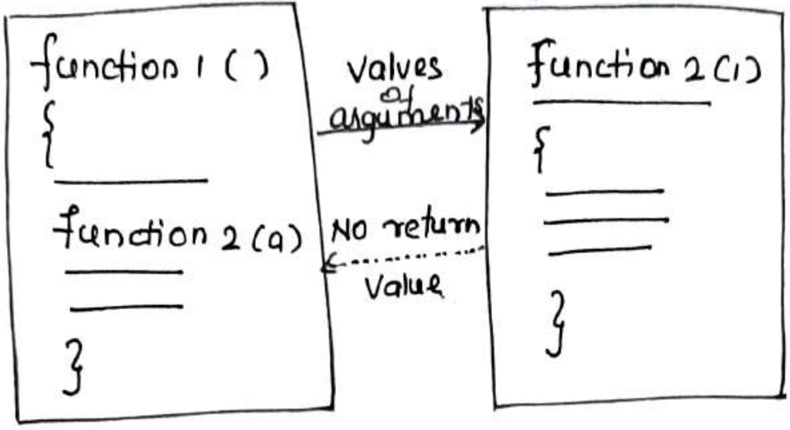
- * When a function has no arguments, it does not receive any data from the calling functions
- * Similarly it does not return a value, the calling function does not receive any data from the called function.
- * In effect, there is no data transfer between the calling function and the called function.



* The dotted lines indicate that there is only a transfer of control but not data.

Arguments but no Return Values:

* Data Communication between the calling function and the called function with arguments but no return value.



One way data communication

For eg: printline(ch)
Value (p, r, n)

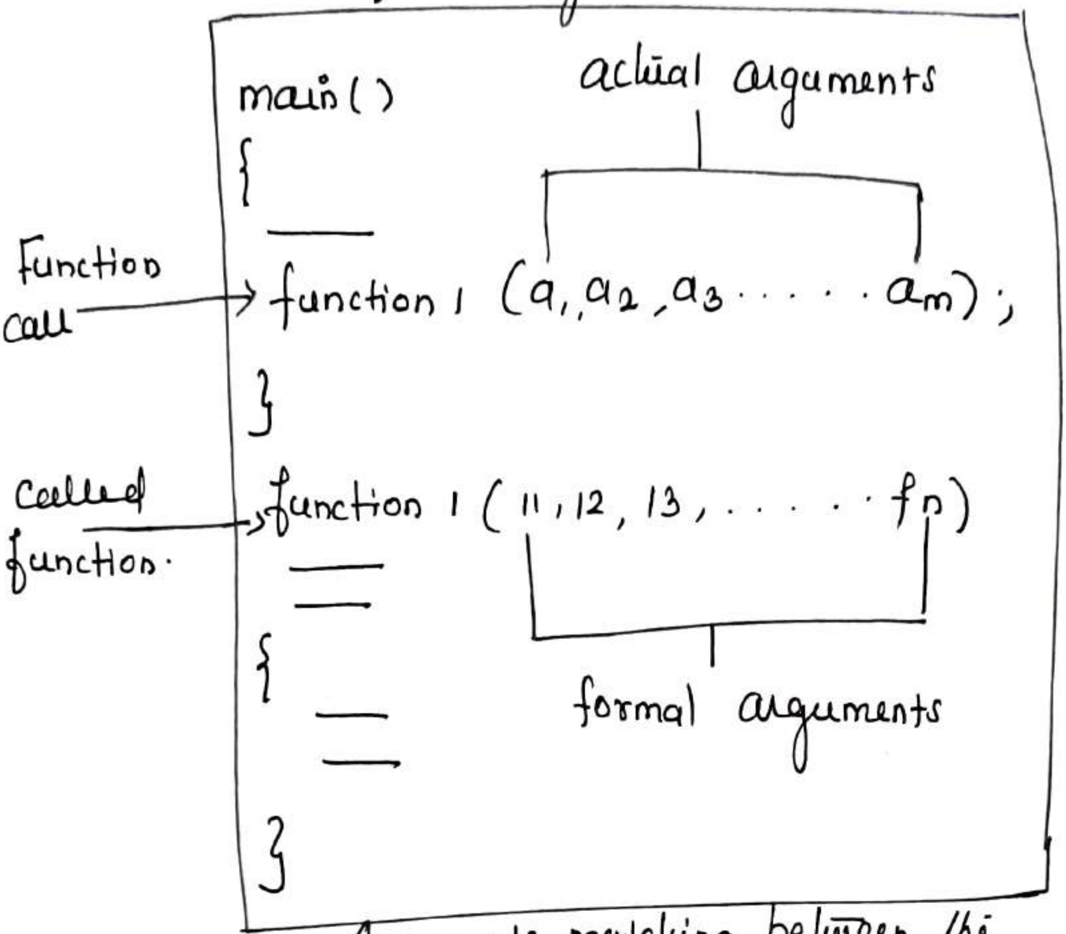
The arguments ch, p, r and n are called the formal arguments. The calling function can now send values to these arguments using functions calls containing appropriate arguments.

For eg: The function call
Value (500, 0.12, 5) would send the values 500, 0.12 and 5 to the function
Value (p, r, n) and assign 500 to p 0.12 to r and 5 to n.

* The values 500, 0.12 and 5 are the actual arguments which become the values of the formal arguments inside the called function

* The actual and formal arguments should match in number, type and order.

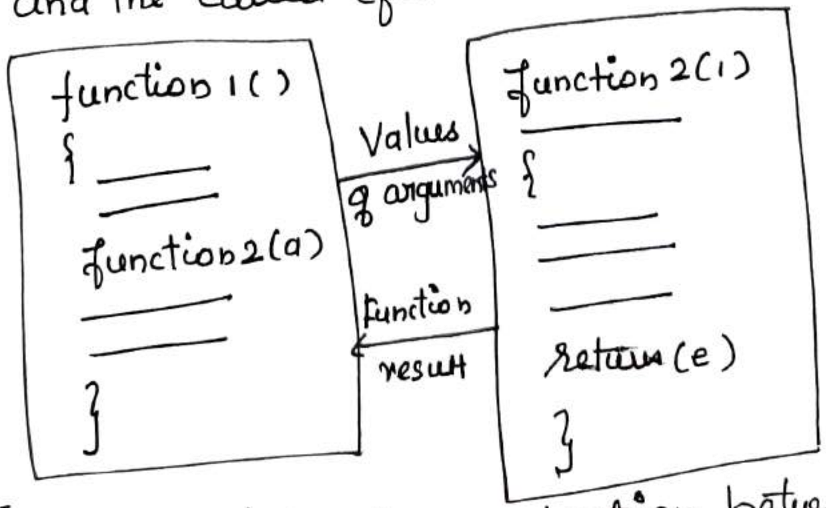
* The values of actual arguments are assigned to the formal arguments on a one to one basis, starting with the first argument.



Arguments matching between the function call and the called function

Arguments with Return Values:

The use of two-way data communication between the calling and the called function



Two-way data Communication between functions

Handling of Non-integer Functions

* Two things to enable a calling function to receive a non-integer value from a called function:

1. The explicit type-specifier, corresponding to the data type required must be mentioned in the function header.

The general form is

```
type - specifier function-name (argument list)
argument declaration ;
{
 function statements;
 }
```

The type-specifier tells the compiler, the type of data the function is to return.

2. The called function must be declared at the start of the body in the calling function, like any other variable. This is to tell the calling function the type of data that the function is actually returning.

Functions Returning Nothing

```
main ()
{                         Value ()
 Printline ();            {
 Value ();
 Printline ();              . . . .
}                           . . .
 Printline ()             }
 {
  . . . .
 }
```

* The function printline() and value() do not return any Value and therefore they were not declared in the main.

* we may declare with the qualifier Void

* This states explicitly that the function do not return Values.

Nesting of Functions:

* main can call function 1, which calls function 2, which calls function 3, ... and so on.

* There is in principle no limits as to how deeply functio can be nested.

Consider the following program:

```
main()
{int a,b,c;
 float ratio();
 Scanf("%d %d %d", &a,&b,&c);
 Printf("%f\n", ratio(a,b,c));
}
 float ratio(x,y,z)
 int x,y,z
 {
  if (difference(y,z))
     return (x/y-z));
  else
     return(0,0);
 }
  difference(p,q)
  int p,q;
  { If(p!=q)
      return(1);
    else
      return(0);
  }
```

Returns float

Returns integer

Nesting of Functions calls is also possible.

For eg, a statement like p= mul(mul(5,2),6); is Valid.

Recursion:

* When a called function in turn calls another function a process of chaining occurs.

* Recursion is a Special case of this process, where a func- calls itself.

* A very simple example of recursion is

```
main ()
{
 Printf("This is an example of recursion\n");
 main();
}
```

When executed, this program will produce an output something like this

```
This is an example of recursion
This is an example of recursion
This is an example of recursion
This is an ex
```

* Another example :

The factorial of a number n is expressed as a series of repetitive multiplications as shown below.

factorial of n = n(n-1)(n-2) . . .   1.

For eg: factorial of 4 = 4×3×2×1 = 24

A function to evaluate factorial of n is as follows:

```
factorial(n)
int n;
{
 int fact
 if (n==1)
   return(1);
 else
   fact = n* factorial(n-1);
 return (fact);
}
```

Assume n = 3. Since the value of n is not 1, the statement `fact = n * factorial(n-1);` will be executed with n=3.

(i) `fact = 3 * factorial(2);` will be evaluated.

With n = 2, the statement is `2 * factorial(1)`

once again, factorial is called with n=1. This time, the function returns 1.

The sequence of operations can be summarized as follows:

```
fact = 3 * factorial(2)
     = 3 * 2 * factorial(1)
     = 3 * 2 * 1
     = 6 .
```

Structures And Unions

If we want to represent a collection of data items of different types using a single name, then we cannot use an array. C Supports a constructed data type known as structure Structures help to organize complex, data in a more meaningful way. It is a powerful concept that we may often need to use in our program design.

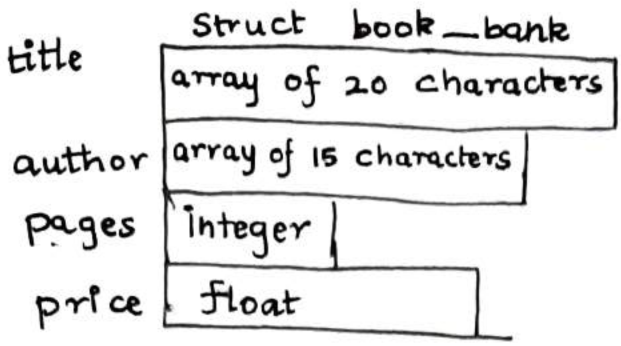
Structure Definition:

A Structure definition creates a format that may be used to declare structure variable. Let us use an example to illustrate the process of structure definition and the creation of structure variables. Consider a book database consisting of book name, author, number of pages, and price. we can define a structure to hold this information as follows:

```
Struct book_bank
{
 Char title [20];
 char author [15];
 int pages;
 float price;
 };
```

The keyword struct declares a structure to hold the details of four fields, namely title, author, pages and price. These fields are called Structure elements or members. Each member may belong to a different type of data. book_bank is the name of the structure and is called the structure tag. The tag name may be used subsequently to declare variables that have the tag's structure.

It simply describes a format called template to represent the information as shown below.



The general format of a structure definition as follows:

```
Struct tag_name
{
 data_type  member 1;
 data_type  member 2;
 .....       ....
 ......      .....
 };
```

we can declare structure variables using the tag name anywhere in the program. For example, the statement

```
Struct book_bank  book1, book2, book 3;
```

declares book1, book2 and book 3 as variables of type struct book_bank.

Each one of these variables has four members as specified by the template. The complete declaration might look like this:

```
Struct  book_bank
{
 char  title [20];
 char   author [15];
 int    pages ;
 float  price ;
};
Struct  book_bank  book1, book2, book 3 ;
```

Remlember that the members of a structure themselves are not variables. They do not occupy any memory until they are associated with the structure variables such as book1

In defining a structure you may note the following syntax:

1. The template is terminated with a semicolon.
2. while the entire declaration is considered as a statement, each member is declared independently for its name and type in a seperate statement inside the template.

3. The tag name such as book_bank can be used to declare structure variables of its type, later in the program

Giving Values to Members :

Thise link between a member and a variable is established using the member operator '.' which is also known as 'dot operator' or 'period operator'. For example

```
book1.price
```

is the variable representing the price of book1 and can be treated like any other ordinary variable.

```
strcpy (book1.title, "BASIC");
strcpy (book1.author, " Balagurusamy");
book1. pages = 250;
book1. price = 28.50;
```

we can also use scanf to give the values through the keyboard.

```
Sa scanf ("%s \n", book1. title);
   scanf ("%d \n", &book1. pages);
```

are valid input statements.

Structure Initialization :

A structure must be declared as static if it is to be initiallized inside a function (similar to arrays).

```
main ()
{ static struct
  {int weight;
  float height;
  }
  Student = {60, 180.75} ;
  .......
  }
```

This assings the value 60 to student, weight and 180.75 to student.height. There is a one-to-one correspondence between the members and their initializing values.

A lot of variation is possible in initializing a structure. The following statements initialize two structure variables.

Here, it is essential to use a tag name.

```
main ()
{struct st_record
{
int weight ;
float height ;
};
static struct st_record student1 = {60, 180.75}
Static Struct st_record student2 = {53, 170.60}
----
----
}
```

Another method is to initialize a structure variable outside the function as shown below:

```
Struct st_record /* No static word */
{
int weight ;
float height ;
} student 1 = {60, 180.75}
main ()
{
Static struct. st_record student 2 = {53, 170.60};
----
----
}
```

c language does not permit the initialization of individual structure members within the template. The initialization must be done only in the declaration of the actual ~~values~~. variables.

Comparison of Structure Variables:

Two Variables of the same structure type can be compared the same way as ordinary variables. If Person 1 and person 2 belong to the same structure, then the following operations are valid.

operation	Meaning
person1 = person2	Assign person2 to person1.
person1 == person2	Compare all members of person1 and person2 and return 1 if they are equal, 0 otherwise.
person1 != person2	Return 1 if all the members are not equal, 0 otherwise.

Arrays of Structures :

For example, in analysing the marks obtained by a class of students, we may use a template to describe student name and marks obtained in various subjects and then declare all the students as structure variables. In such cases, we may declare an array of structures, each element of the array representing a structure variable. For example,

```
struct class student [100];
```

defines an array called Student, that consists of 100 elements. Each element is defined to be of the type struct class. Consider the following declaration.

```
Struct marks
{
 int subject 1 ;
 int subject 2 ;
 int subject 3 ;
};
main ()
{
Static struct marks student [3] =
   {{45,68,81},{75,53,69},{57,36,71}};
```

This declares the student as an array of three elements student [0], Student [1], and student [2] and initializes their members as follows :

```
Student [0] . subject 1 = 45 ;
Student [0] . Subject 2 = 68 ;
        . . . .
        . . . .
Student [2] . subject 3 = 71 ;
```

student [0]. subject 1	45
. subject 2	68
. subject 3	81
student [1]. subject 1	75
. subject 2	53
. subject 3	69
student [2]. subject 1	57
. subject 2	36
. subject 3	71

The array student inside memory.

Arrays within Structures:

C permits the use of arrays as structure members. we can use single - or multi-dimensional arrays of type int or float

For example, the following structure declaration is valid :

```
struct marks
{
 int number;
 float subject [3];
}student [2];
```

Here, the member subject contains three elements subject [0], subject [1] and subject [2]. These elements can be accessed using appropriate subscripts. For example, the name

student [1]. subject [2];

would refer to the mark obtained in the third subject by the second student.

Structures within Structures:

structures within a structure means nesting of structures. Nesting of structures is permitted in C.

```
Struct  Salary
{
 char  name [20];
 char  department [10];
 Int    basic_pay;
 int    dearness_allowance;
 Int     city _allowance;
}employee;
```

```
struct  Salary
{
 char  name [20];
 char  department [10];
 Struct
  {
   int  dearness;
   int  house_rent;
   int  city;
  }
  allowance;
 }
  employee;
```

```
employee.allowance.dearness
employee.allowance.house_rent
employee.allowance.city
```

An inner-most member in a nested structure can be accessed by chaining all the concerned structure variables (from outer-most to inner-most) with the member using dot operator.