UNIT II

FUNCTIONS, POINTERS, STRUCTURES AND UNIONS

FUNCTIONS AND POINTERS

- Functions are created when the same process or an algorithm to be repeated several times in various places in the program.
- Function has a self-contained block of code, that executes certain task. A function has a name, a list of arguments which it takes when called, and the block of code it executes when called.
- Functions are two types:
    1. Built-in / Library function.
        Example: printf(), scanf(), getch(), exit(), etc.
    2. User defined function.

User-Defined Functions

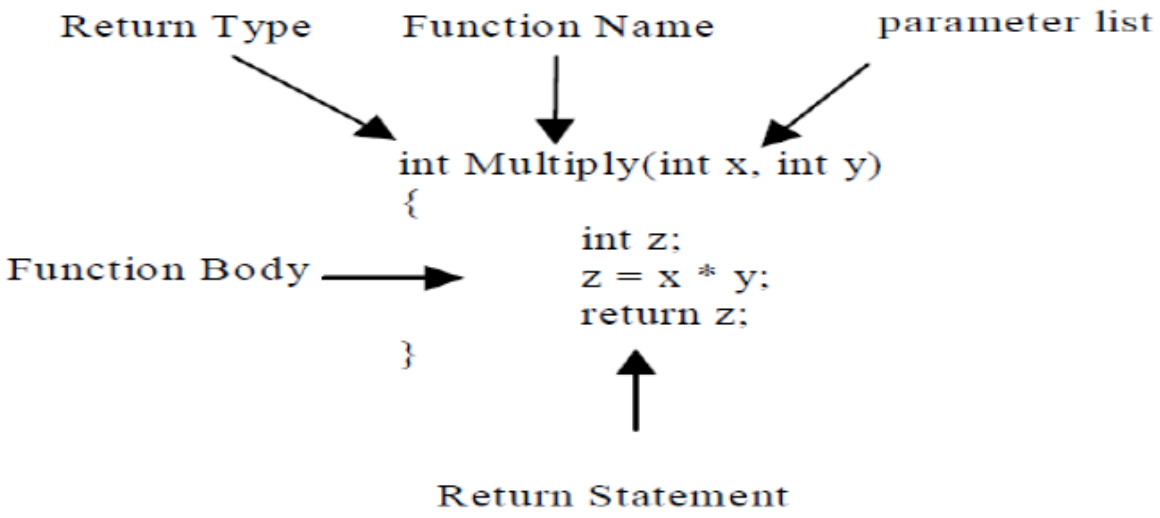
Functions defined by the users according to their requirements are called user-defined functions. These functions are used to break down a large program into a small functions.

Advantage of User defined function:

1. Reduce the source code
2. Easy to maintain and modify
3. It can be called anywhere in the program.

Body of user defined function:

```
return_type f_name (argument1,argument 2)
{
local variables;
statements;
return_type;
}
```



The body of user-defined shows that an user-defined functions has following characteristics:

1. Return type
2. Function name
3. Parameter list of arguments
4. Local variables
5. Statements
6. Return value

S.No	C function aspects	Syntax
1	function definition	return_type function_name (arguments list) { Body of function; }
2	function call	function_name (arguments list);
3	function declaration	return_type function_name (argument list);

Note: The function with return value must be the data type, that is return type and return value must be of same data type.

User defined function has three parts:

1. Declaration part

    ret_type f_name (arguments)

2. Calling part

    f_name(arguments);

3. Definition part

Function Parameters:

Parameters provide the data communication between calling function and called function.

Two types:

 Actual parameters:

These are the parameters transferred from the calling function[main function] to the called function[user defined function].

 Formal parameters:

Passing the parameters from the called functions[user defined function] to the calling functions[main function].

Note:

 Actual parameter– This is the argument which is used in function call.

 Formal parameter– This is the argument which is used in function definition.

4.1 Categories of User defined function/Function Prototypes:

- A function prototype declaration consists of the functions return type, name and arguments list.
- Always terminated with semicolon.

The following are the function prototypes:

1. Function without return value and without argument.

2. Function without return value and with argument.

3. Function with return value and with argument.

4. Function with return value and without argument.

Note: Function with more than one return value will not have return type and return statement in the function definition.

Consider the following example to multiply two numbers:

```
void main( )
{
int x,y,z;
scanf(“%d%d”, &x,&y);
z=x* y;
printf(“The result is %d”, z); }
```

1. Function without return value and without argument

In this prototype, no data transfers takes place between the calling function and the called function. They read data values and print result in the same block.

Syntax:

```
void f_name(); //function declaration
void f_name (void) //function definition
{
local variables;
statements;
}
void main()
{
f_name(); //function call
}
```

The above example can be rewritten by creating function:

```
void f_mult(); //function definition
void f_mult(void )
{
int x,y,z;
scanf("%d%d", &x,&y);
z=x* y;
printf("The result is %d", z);
}
void main()
{
f_mult();
}
```

2. Function without return value and with argument

In this prototype, data is transferred from the calling function to called function. The called function receives some data from the calling function and does not send back any values to the calling functions.

Syntax:

```
void f_name(int x, int y ); //Function declaration
void f_name (int x, int y) //Function Definition //Formal Parameters
{
local variables;
statements;
}
void main()
{
//variable declaration
//input statement
f_name(c, d); //Function call //Actual Parameters
}
```

The above example can be rewritten by creating function

```
void f_mult(int x, int y);
void f_mult(int x, int y )
{
int z;
z=x* y;
printf("The result is %d", z);
}
void main()
{  int c, d;
printf("Enter any two number");
scanf("%d%d", &c, &d);
f_mult(c, d); //Function call
}
```

3. Function with return value and without argument

The calling function cannot pass any arguments to the called function but the called function may send some return value to the calling function.

Syntax:

```
int f_name(); //Function declaration
int f_name (void) //Function Definition
{
local variables;
statements;
return int;
}
void main()
{
//variable declaration
ret_var=f_name(); //Function call
}
```

The above example can be rewritten by creating function

```
int f_mult();
int f_mult(void )
{
int x,y,z;
scanf("%d%d", &x,&y);
z=x* y;
printf("The result is %d", z);
return(z); }
void main()
{
int c;
c=f_mult();
getch();
}
```

4. Function with return value and with argument

In this prototype, the data is transferred between the calling function and called function. The called function receives some data from the calling function and send back a value return to the calling function.

Syntax:

```
int f_name (int x, int y); //Function declaration
int f_name (int x, int y) //Function definition //Formal Parameters
{
local variables;
statements;
return int;
}
void main()
{
//variable declaration
//Input Statement
ret_value=f_mult(a, b); //Function Call //Actual Parameters
}
```

The above example can be rewritten by creating function:

```
int f_mult(int x, int y);
int f_mult(int x, int y)
{
int z;
z=x* y;
printf("The result is %d", z);
return(z);
}
void main()
{
int a,b,c;
printf("Enter any two value:");
```

```
scanf("%d%d", &a, &b);
c=f_mult(a, b);
}
```

Note:

- If the return data type of a function is "void", then, it can‟t return any values to the calling function.
- If the return data type of the function is other than void such as "int, float, double etc", then, it can return values to the calling function.

return statement:

It is used to return the information from the function to the calling portion of the program.

Syntax*:*

```
return;
return();
return(constant);
return(variable);
return(exp);
return(condn_exp);
```

By default, all the functions return int data type.

Do you know how many values can be return from C functions?

- Always, only one value can be returned from a function.
- If you try to return more than one values from a function, only one value will be returned that appears at the right most place of the return statement.
- For example, if you use "return a,b,c" in your function, value for c only will be returned and values a, b won‟t be returned to the program.
- In case, if you want to return more than one values, pointers can be used to directly change the values in address instead of returning those values to the function.

Function Call:

A function can be called by specifying the function_name in the source program with parameters, if presence within the paranthesis.

Syntax:

```
Fn_name();
```

Fn_name(parameters);

Ret_value=Fn_name(parameters);

Example:

```
#include<stdio.h>
#include<conio.h>
int add(int a, int b); //function declaration
void main()
{
int x,y,z;
printf("\n Enter the two values:");
scanf("%d%d",&x,&y);
z=add(x,y); //Function call(Actual parameters)
printf("The sum is .%d", z);
}
int add(int a, int b) //Function definition(Formal parameters)
{
int c;
c=a+b;
return(c); //return statement
}
```

4.2 Parameter Passing Methods/Argument Passing Methods

Call by Value/Pass by Value:

When the value is passed directly to the function it is called call by value. In call by value only a copy of the variable is only passed so any changes made to the variable does not reflects in the calling function.

Example:

```
#include<stdio.h>
#include<conio.h>
swap(int,int);
void main()
{
```

```
int x,y;
printf("Enter two nos");
scanf("%d %d",&x,&y);
printf("\nBefore swapping : x=%d y=%d",x,y);
swap(x,y);
getch();
}
swap(int a,int b)
{
int t;
t=a;
a=b;
b=t;
printf("\nAfter swapping :x=%d y=%d",a,b);
}
```

System Output:

Enter two nos 12 34

Before swapping :12 34

After swapping : 34 12

Call by Reference/Pass by Reference:

When the address of the value is passed to the function it is called call by reference. In call by reference since the address of the value is passed any changes made to the value reflects in the calling function**.**

Example:

```
#include<stdio.h>
#include<conio.h>
swap(int *, int *);
void main()
{
int x,y;
```

```
printf("Enter two nos");
scanf("%d %d",&x,&y);
printf("\nBefore swapping:x=%d y=%d",x,y);
swap(&x,&y);
printf("\nAfter swapping :x=%d y=%d",x,y);
getch();
}
swap(int *a,int *b)
{
int t;
t=*a;
*a=*b;
*b=t;
}
```

System Output:

Enter two nos 12 34

Before swapping :12 34

After swapping : 34 12

Call by Value	Call by Reference
This is the usual method to call a function in which only the value of the variable is passed as an argument	In this method, the address of the variable is passed as an argument
Any alternation in the value of the argument passed does not affect the function.	Any alternation in the value of the argument passed affect the function.
Memory location occupied by formal and actual arguments is different	Memory location occupied by formal and actual arguments is same and there is a saving of memory location
Since a new location is created, this method is slow	Since the existing memory location is used through its address, this method is fast
There is no possibility of wrong data manipulation since the arguments are directly used in an application	There is a possibility of wrong data manipulation since the addresses are used in an expression. A good skill of programming is required here

Library Functions:

C language provides built-in-functions called library function compiler itself evaluates these functions.

List of Functions

- Sqrt(x)□ (x)0.5
- Log(x)
- Exp(x)
- Pow(x,y)
- Sin(x)
- Cos(x)
- Rand(x)-> generating a positive random integer.

4.3 Recursion in C:

Recursion is calling function by itself again and again until some specified condition has been satisfied.

Syntax:

```
int f_name (int x)
{
local variables;
f_name(y); // this is recursion
statements;
}
```

Example_1: Factorial using Recursion

```
#include<stdio.h>
#include<conio.h>
int factorial(int n);
void main()
{
int res,x;
printf("\n Enter the value:");
```

```
scanf(“%d”, &x);
res=factorial(x);
printf(“The factorial of %d is ..%d”, res);
}
int factorial(int n)
{
int fact;
if (n==1)
return(1);
else
fact = n*factorial(n-1);
return(fact);
 }
```

Example_2: Fibonacci using Recursion

```
#include<stdio.h>
#include<conio.h>
int Fibonacci(int);
int main()
{
int n, i = 0, c;
scanf("%d",&n);
printf("Fibonacci series\n");
for ( c = 1 ; c <= n ; c++ )
{
printf("%d\n", Fibonacci(i));
i++;
}
return 0;
}
int Fibonacci(int n)
{
```

```
if ( n == 0 )
return 0;
else if ( n == 1 )
return 1;
else
return ( Fibonacci(n-1) + Fibonacci(n-2) );
}
```

Example_3: Sum of n Natural Numbers

```
#include<stdio.h>
#include<conio.h>
int add(int n);
void main()
{
int m, x;
printf("Enter an positive integer: ");
scanf("%d",&m);
x=add(m);
printf("Sum = %d", x);
getch();
}
int add(int n)
{
if(n!=0)
return n+add(n-1); /* recursive call */
}
```

4.4 POINTERS

Definition:

- C Pointer is a variable that stores/points the address of the another variable.
- C Pointer is used to allocate memory dynamically i.e. at run time.
- The variable might be any of the data type such as int, float, char, double, short etc.
- Syntax : data_type *var_name;

  Example : int *p; char *p;

  Where, * is used to denote that "p" is pointer variable and not a normal variable.

Key points to remember about pointers in C:

- Normal variable stores the value whereas pointer variable stores the address of the variable.
- The content of the C pointer always be a whole number i.e. address.
- Always C pointer is initialized to null, i.e. int *p = null.
- The value of null pointer is 0.
- & symbol is used to get the address of the variable.
- * symbol is used to get the value of the variable that the pointer is pointing to.
- If pointer is assigned to NULL, it means it is pointing to nothing.
- The size of any pointer is 2 byte (for 16 bit compiler).
- No two pointer variables should have the same name.
- But a pointer variable and a non-pointer variable can have the same name.

4.4.1 Pointer –Initialization:

Assigning value to pointer:

It is not necessary to assign value to pointer. Only zero (0) and NULL can be assigned to a pointer no other number can be assigned to a pointer. Consider the following examples;

int *p=0;

int *p=NULL; The above two assignments are valid.

int *p=1000; This statement is invalid.

Assigning variable to a pointer:

int x; *p;

p = &x;

This is nothing but a pointer variable p is assigned the address of the variable x. The address of the variables will be different every time the program is executed.

Reading value through pointer:

int x=123; *p;

p = &x;

Here the pointer variable p is assigned the address of variable x.

printf("%d", *p); will display value of x 123. This is reading value through pointer

printf("%d", p); will display the address of the variable x.

printf("%d", &p); will display the address of the pointer variable p.

printf("%d",x); will display the value of x 123.

printf("%d", &x); will display the address of the variable x.

Note: It is always a good practice to assign pointer to a variable rather than 0 or NULL.

Pointer Assignments:

We can use a pointer on the right-hand side of an assignment to assign its value to another variable.

Example:

```
int main()
{
int var=50;
int *p1, *p2;
p1=&var;
p2=p1;
}
```

Chain of pointers/Pointer to Pointer:

A pointer can point to the address of another pointer. Consider the following example.

```
int x=456, *p1, **p2; //[pointer-to-pointer];
p1 = &x;
p2 = &p1;
printf("%d", *p1); will display value of x 456.
printf("%d", *p2); will also display value of x 456.
```

This is because p2 point p1, and p1 points x.

Therefore p2 reads the value of x through pointer p1. Since one pointer is points towards another pointer it is called chain pointer. Chain pointer must be declared with ** as in **p2.

Manipulation of Pointers

We can manipulate a pointer with the indirection operator „*", which is known as dereference operator. With this operator, we can indirectly access the data variable content.

Syntax:

*ptr_var;

Example:

```
#include<stdio.h>
void main()
{
int a=10, *ptr;
ptr=&a;
printf("\n The value of a is ",a);
*ptr=(*ptr)/2;
printf("The value of a is.",(*ptr));
}
```

Output:

The value of a is: 10

The value of a is: 5

4.4.2 Pointer Expression & Pointer Arithmetic

- C allows pointer to perform the following arithmetic operations:
- A pointer can be incremented / decremented.
- Any integer can be added to or subtracted from the pointer.

A pointer can be incremented / decremented.

In *16 bit machine*, size of all types[data type] of pointer always 2 bytes.

Eg:

```
int a;
int *p;
p++;
```

Each time that a pointer p is incremented, the pointer p will points to the memory location of the next element of its base type. Each time that a pointer p is decremented, the pointer p will points to the memory location of the previous element of its base type.

```
int a,*p1, *p2, *p3;
p1=&a;
p2=p1++;
p3=++p1;
printf("Address of p where it points to %u", p1); 1000
printf("After incrementing Address of p where it points to %u", p1); 1002
printf("After assigning and incrementing p %u", p2); 1000
printf("After incrementing and assigning p %u", p3); 1002
```

In *32 bit machine*, size of all types of pointer is always 4 bytes.

The pointer variable p refers to the base address of the variable a. We can increment the pointer variable,

p++ or ++p

This statement moves the pointer to the next memory address. let p be an integer pointer with a current value of 2,000 (that is, it contains the address 2,000). Assuming 32-bit integers, after the expression

p++

the contents of p will be 2,004, not 2,001! Each time p is incremented, it will point to the next integer. The same is true of decrements. For example,

p –

will cause p to have the value 1,996, assuming that it previously was 2,000. Here is why: Each time that a pointer is incremented, it will point to the memory location of the next element of its base type. Each time it is decremented, it will point to the location of the previous element of its base type.

Any integer can be added to or subtracted from the pointer.

Like other variables pointer variables can be used in expressions. For example if p1 and p2 are properly declared and initialized pointers, then the following statements are valid.

```
y=*p1**p2;
sum=sum+*p1;
```

```
z= 5* - *p2/p1;
*p2= *p2 + 10;
```

C allows us to add integers to or subtract integers from pointers as well as to subtract one pointer from the other. We can also use short hand operators with the pointers p1+=; sum+=*p2; etc., we can also compare pointers by using relational operators the expressions such as p1 >p2 , p1==p2 and p1!=p2 are allowed.

/*Program to illustrate the pointer expression and pointer arithmetic*/

```
#include< stdio.h >
#include<conio.h>
void main()
{
int ptr1,ptr2;
int a,b,x,y,z;
a=30;b=6;
ptr1=&a;
ptr2=&b;
x=*ptr1+ *ptr2 –6;
y=6*- *ptr1/ *ptr2 +30;
printf(“\nAddress of a + %u”,ptr1);
printf(“\nAddress of b %u”, ptr2);
printf(“\na=%d, b=%d”, a, b);
printf(“\nx=%d,y=%d”, x, y);
ptr1=ptr1 + 70;
ptr2= ptr2;
printf(“\na=%d, b=%d”, a, b);
}
```

/* Sum of two integers using pointers*/

```
#include <stdio.h>
int main()
{
int first, second, *p, *q, sum;
```

```
printf("Enter two integers to add\n");
scanf("%d%d", &first, &second);
p = &first;
q = &second;
sum = *p + *q;
printf("Sum of entered numbers = %d\n",sum);
return 0;
}
```

4.4.3 Pointers and Arrays

Array name is a constant pointer that points to the base address of the array[i.e the first element of the array]. Elements of the array are stored in contiguous memory locations. They can be efficiently accessed by using pointers.

Pointer variable can be assigned to an array. The address of each element is increased by one factor depending upon the type of data type. The factor depends on the type of pointer variable defined. If it is integer the factor is increased by 2. Consider the following example:

```
int x[5]={11,22,33,44,55}, *p;
p = x; //p=&x; // p = &x[0];
```

Remember, earlier the pointer variable is assigned with address (&) operator. When working with array the pointer variable can be assigned as above or as shown below:

Therefore the address operator is required only when assigning the array with element.

Assume the address on x[0] is 1000 then the address of other elements will be as follows

x[1] = 1002

x[2] = 1004

x[3] = 1006

x[4] = 1008

The address of each element increase by factor of 2. Since the size of the integer is 2 bytes the memory address is increased by 2 bytes, therefore if it is float it will be increase 4 bytes, and for double by 8 bytes. This uniform increase is called scale factor.

```
p = &x[0];
```

Now the value of pointer variable p is 1000 which is the address of array element x[0].

To find the address of the array element x[1] just write the following statement.

p = p + 1;

Now the value of the pointer variable p is 1002 not 1001 because since p is pointer variable the increment of will increase to the scale factor of the variable, since it is integer it increases by 2.

The p = p + 1; can be written using increment or decrement operator ++p; The values in the array element can be read using increment or decrement operator in the pointer variable using scale factor.

Consider the above example.

printf("%d", *(p+0)); will display value of array element x[0] which is 11.

printf("%d", *(p+1)); will display value of array element x[1] which is 22.

printf("%d", *(p+2)); will display value of array element x[2] which is 33.

printf("%d", *(p+3)); will display value of array element x[3] which is 44.

printf("%d", *(p+4)); will display value of array element x[4] which is 55.

/*Displaying the values and address of the elements in the array*/

```
#include<stdio.h>
void main()
{
int a[6]={10, 20, 30, 40, 50, 60};
int *p;
int i;
p=a;
for(i=0;i<6;i++)
{
printf("%d", *p); //value of elements of array
printf("%u",p); //Address of array
}
getch();
}
```

/* Sum of elements in the Array*/

```
#include<stdio.h>
#include<conio.h>
void main()
```

```
{
int a[10];
int i,sum=0;
int *ptr;
printf("Enter 10 elements:n");
for(i=0;i<10;i++)
scanf("%d",&a[i]);
ptr = a; /* a=&a[0] */
for(i=0;i<10;i++)
{
sum = sum + *ptr; //*p=content pointed by 'ptr'
ptr++;
}
printf("The sum of array elements is %d",sum);
}
```

/*Sort the elements of array using pointers*/

```
#include<stdio.h>
int main(){
int i,j, temp1,temp2;
int arr[8]={5,3,0,2,12,1,33,2};
int *ptr;
for(i=0;i<7;i++){
for(j=0;j<7-i;j++){
if(*(arr+j)>*(arr+j+1)){
ptr=arr+j;
temp1=*ptr++;
temp2=*ptr;
*ptr--=temp1;
*ptr=temp2;
}}}
for(i=0;i<8;i++){
```

printf(" %d",arr[i]); } }

4.4.4 Pointers and Multi-dimensional Arrays

The array name itself points to the base address of the array.

Example:

```
int a[2][3];
int *p[2];
p=a; //p points to a[0][0]
```

/*Displaying the values in the 2-d array*/

```
#include<stdio.h>
void main()
{
int a[2][2]={{10, 20},{30, 40}};
int *p[2];
int i,j;
p=a;
for(i=0;i<2;i++)
{
for(j=0;j<2;j++)
{
printf("%d", *(*(p+i)+j)); //value of elements of array
}
}
getch();
}
```

4.5 Dynamic Memory Allocation

The process of allocating memory during program execution is called dynamic memory allocation.

Dynamic memory allocation functions

Sl.No	Function	Syntax	Use
1	malloc()	ptr=(cast-type*)malloc(byte-size)	Allocates requested size of bytes and returns a pointer first byte of allocated space.
2	calloc()	ptr=(cast-type*)calloc(n,element-size)	Allocates space for an array elements, initializes to zero and then returns a pointer to memory.
3	free()	free(ptr);	dellocate the previously allocated space.
4	realloc()	ptr=realloc(ptr,newsize);	Change the size of previously allocated space.

STRUCTURES, UNIONS AND PREPROCESSOR DIRECTORIES

Structure

- Single name that contains a collection of data items of same data type, Single name that contains a collection of data items of different data types.
- Individual entries in a array are called elements. Individual entries in a structure are called members.
- No Keyword Keyword: **struct**
- Members of an array are stored in sequence of memory locations.
- Members of a structure are not stored in sequence of memory locations.

5.1 STRUCTURES

Structure is a collection of variables under the single name, which can be of different data type. In simple words, Structure is a convenient way of grouping several pieces of related information together.

A structure is a derived data type, The scope of the name of a structure member is limited to the structure itself and also to any variable declared to be of the structure's type.

Variables which are declared inside the structure are called “members of structure”.

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

```
struct <tag>
{
member 1;
member 2;
….
…
member m;
}
```

where, struct is a keyword, <tag> is a structure name.

For example:

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

Now we need an interface to access the members declared inside the structure, it is called structure variable. we can declare the structure variable in two ways:

i) within the structure definition itself.

ii) within the main function.

i] within the structure definition itself.

```
struct tag
{
member 1;
```

member 2;

____

__

member

m;

} variable 1, variable 2 variable

n; //Structure variables

Eg:

struct student

{

char name [80];

int roll_no;

float marks;

}s1;

ii] within the main function.

struct tag

{

member 1;

member 2;

____

__

member

m;

};

void main()

{

struct tag var1, var2,...,var_n; //structure variable

}

Eg:

```
struct student
{
char name [80];
int roll_no;
float marks;
};
void main()
{
struct student s1, s2; //declaration of structure variable.
};
```

Note: Structure variable can be any of three types:

1. normal variable
2. array_variable
3. pointer_variable

Initialization of Structure members:

A structure variable, like an array can be initialized in two ways:

I.

```
struct student
{
char name [80];
int roll_no;
float marks ;
} s1={“Saravana”,34,469};
```

II.

```
struct student
{
char name [80];
int roll_no;
float marks ;
} s1;
struct student s1= {“Saravana”,34,469};
```

[OR]

s1.name={“Saravana”};

s1.roll_no=34;

s1.marks=500;

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

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

} st [100];
```

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

Accessing the Structure Members

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

structure_variable.member_name //[normal variable]

Eg:

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

e.g. if we want to get the detail of a member of a structure then we can write as

scanf(“%s”,st.name); or scanf(“%d”, &st.roll_no) and so on.

if we want to print the detail of a member of a structure then we can write as

printf("%s",st.name); or printf("%d", st.roll_no) and so on.

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

array [expression].member_name //[array variable]

Eg:

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

float marks ;
}st[10];
```

e.g. if we want to get the detail of a member of a structure then we can write as

scanf("%s",st[i].name); or scanf("%d", &st[i].roll_no) and so on.

if we want to print the detail of a member of a structure then we can write as

printf("%s",st[i].name); or printf("%d", st[i].roll_no) and so on.

The use of the period operator can be extended to pointer of structure by writing:

array [expression]>

member_name //[array variable]

Eg:

```
struct student
{
char name [80];
int roll_no ;
float marks ;
}*st;
void main()
{
Struct student s;
st=&s;
```

e.g. if we want to get the detail of a member of a structure then we can write as

scanf("%s",st>

name); or scanf("%d", &st>

roll_no) and so on.

if we want to print the detail of a member of a structure then we can write as

printf("%s",st>

name); or printf("%d", st>

roll_no) and so on.

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

PROGRAM'S USING STRUCTURE

Example 1://To print the student details// normal structure variable

```
#include<stdio.h>
struct student
{
char name[30];
int reg_no[15];
char branch[30];
};
void main()
{
struct student s1;
printf("\n Enter the student name::");
scanf("%s",s1.name);
printf("\n Enter the student register number::");
scanf("%s",&s1.reg_no);
printf("\n Enter the student branch::");
scanf("%s",s1.branch);
printf("\n Student Name::",s1.name);
printf("\n Student Branch::",s1.branch);
printf("\n Student reg_no::",s1.reg_no);
getch();
}
```

Example 2://To print the student details// pointer structure variable

```
#include <stdio.h>
struct name{
int a;
float b;
};
int main()
{
struct name *ptr,p;
ptr=&p; /* Referencing pointer to memory address of p */
printf("Enter integer: ");
scanf("%d",&(*ptr).a);
printf("Enter number: ");
scanf("%f",&(*ptr).b);
printf("Displaying: ");
printf("%d%f",(*ptr).a,(*ptr).b);
return 0;
}
```

Example 3://To print the Mark sheet for a student//

```
#include<stdio.h>
struct student
{
char name[30];
int reg_no[15];
char branch[30];
int m1,m2,m3,total;
float avg;
};

void main()
{
int total;
```

```
float avg;
struct student s1;
printf("\n Enter the student name::");
scanf("%s",s1.name);
printf("\n Enter the student register number::");
scanf("%s",&s1.reg_no);
printf("\n Enter the student branch::");
scanf("%s",s1.branch);
printf("\n Enter the 3 subjects Marks:");
scanf("%d%d%d", &s1.m1,&s1.m2,&s1.m3);
s1.total=s1.m1+s1.m2+s1.m3;
printf("\n Ur Total mark is.%d", s1.total);
s1.avg=a1.total/3;
printf("\n Ur Average mark is.%f", s1.avg);
getch();
}
```

Example_4: /* To Print Students Mark Sheet's using Structures*/

```
#include"stdio.h"
#include"conio.h"
struct student
{
char name[25],grade;
int reg_no[15];
int s1,s2,s3,s4,s5,total;
float avg;

}sa[20];

void main()
{
```

```
int i,n,total;
float avg;
printf("\n Enter the count of Students need marksheet::");
scanf("%d",&n);
printf("\n Enter the name of students:,reg_no, 5 sub marks::");
for(i=0;i<n;i++)
{
printf("\n Enter the name of students,reg_no, 5 sub marks::%d",i+1);
scanf("%s%d%d%d%d%d%d", &sa[i].name, &sa[i].reg_no, &sa[i].s1, &sa[i].s2, &sa[i].s3,
&sa[i].s4, &sa[i].s5);
sa[i].total=sa[i].s1+sa[i].s2+sa[i].s3+sa[i].s4+sa[i].s5;
sa[i].avg=sa[i].total/5;
if((sa[i].s1<50)||(sa[i].s2<50)||(sa[i].s3<50)||(sa[i].s4<50)||(sa[i].s5<50))
{
sa[i].grade='U';
printf("Ur grade is %c", sa[i].grade);
}

else
{
if(sa[i].avg==100)
{
sa[i].grade='S';
printf("Ur Grade is %c", sa[i].grade);
}
if((sa[i].avg>90)&&(sa[i].avg<=99))
{

sa[i].grade='A';
printf("Ur Grade is %c", sa[i].grade);
}
```