UNIT -3Syllabus:
Introduction to structured programming, Functions – basics, user defined functions, inter functions communication, Standard functions, Storage classes- auto, register, static, extern,scope rules, arrays to functions, recursive functions, example C programs.
String – Basic concepts, String Input / Output functions, arrays of strings, string handling functions, strings to functions, C programming examples.

1. Introduction to structured programming

Software engineering is a discipline that is concerned with the construction of robust and reliable computer programs. Just as civil engineers use tried and tested methods for the construction of buildings, software engineers use accepted methods for analyzing a problem to be solved, a blueprint or plan for the design of the solution and a construction method that minimizes the risk of error.
The structured programming approach to program design was based on the following method.

i. To solve a large problem, break the problem into several pieces and work on each piece separately.
ii. To solve each piece, treat it as a new problem that can itself be broken down into smaller problems;
iii. Repeat the process with each new piece until each can be solved directly, without further decomposition.

2. Functions - Basics

In programming, a function is a segment that groups code to perform a specific task. A C program has at least one function main().Without main() function, there is technically no C program.

Types of C functions
There are two types of functions in C programming:

1. Library functions
2. User defined functions

1 Library functions

Library functions are the in-built function in C programming system.
For example:
main() - The execution of every C program starts form this main() function.
printf() – is used for displaying output in C.
scanf() – is used for taking input in C.

2 User defined functions

C allows programmer to define their own function according to their requirement. These types of functions are known as user- defined functions. Suppose, a programmer wants to find

factorial of a number and check whether it is prime or not in same program. Then, he /she can create tow separate user-defined functions in that program: one for finding factorial and other for checking whether it is prime or not.

How user-defined function works in C Programming?

```
#include<stdio.h>
voidfunction_name(){
            ……………….
            ……………….
}
int main(){
           ……………….
           ……………….
function_name();
           ……………….
           ……………….
}
```

As mentioned earlier, every C program begins form main() and program starts executing the codes inside main() function. When the control of program reaches to function_name() inside main() function. The Control of program jumps to void function_name() and executes the codes inside it. When all the codes inside that user-defined function are executed, control of the program jumps to the statement just after function_name() from where it is called. Analyze the figure below for understanding the concept of function in C programming. Visit this page to learn in detail about user-defined functions.

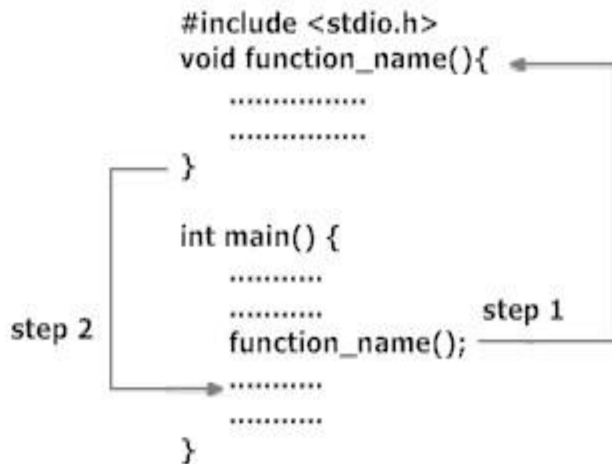


Fig: Working of Functions

Remember, the function name is an identifier and should be unique.

Advantages of user defined functions

1. User defined functions helps to decompose the large program into small segments which makes programmer easy to understand, maintain and debug.

2. If repeated code occurs in a program. Function can be used to include those codes and execute when needed by calling that function.
3. Programmer working on large project can divide the workload by making different functions.

3. C Programming User-defined functions

Example of user-defined function

Write a C program to add two integers. Make a function **add** to add integers and display sum int**main()** function.

```
/* program  to demonstrate the working of user defined function*/
#include<stdio.h>
#include<conio.h>
int add(int a, int b);                          /*  Function Declaration  */
int main()
{
        int num1,num2,sum;
        clrscr();
        printf("enter two numbers  : \n ");
        scanf("%d %d",&num1,&num2);
        sum=add(num1,num2);                     /* Function Call*/
        printf("\n sum is : %d",sum);
        getch();
}
int add(inta,int b)                             /* Function Definition */
{
        int add;
        add=a+b;
        return add;
}
```

3.1 Function Declaration

Every function in C programming should be declared before they are used. These type of declaration are also called function prototype. Function prototype gives compiler information about function name, type of arguments to be passed and return type.

Syntax:

return_typefunction_name(type(1) argument(1),....,type(n) argument(n));

In the above example,int add(int a, int b); is a function prototype which provides following information to the compiler:name of the function is add()

1. return type of the function is int.
2. two arguments of type int are passed to function.

Function prototype are not needed if user-definition function is written before main() function.

3.2 Function call

Control of the program cannot be transferred to user-defined function unless it is called invoked.

Syntax:

```
function_name(argument(1),....argument(n));
```

In the above example, function call is made using statement add(num1,num2); from main(). This make the control of program jump from that statement to function definition and executes the codes inside that function.

3.3 Function Definition:

Function definition contains programming codes to perform specific task.

Syntax:

```
return_typefunction_name(type(1) argument(1),..,type(n) argument(n))
{
        //body of function
}
```

Function definition has two major components:

3.3.1Function declarator or function header

Function declarator is the first line of function definition. When a function is called, control of the program is transferred to functiondeclarator.

Syntax

```
return_typefunction_name(type(1) argument(1),....,type(n) argument(n))
```

Syntax of function declaration and declarator are almost same except, there is no semicolon at the end of declarator and function declarator is followed by function body.
In above example, intadd(inta,int b) in line 12 is a function declarator.

3.3.2 Function body

Function declarator is followed by body of function inside braces.

3.4 Passing Arguments to functions

In programming, argument(parameter) refers to data this is passed to function(function definition) while calling function.
In above example two variable, num1 and num2 are passed to function during function call and

these arguments are accepted by arguments a and b in function definition.
Arguments that are passed in function call and arguments that are accepted in function definition should have same data type. For example:

If argument *num1* was of int type and *num2* was of float type then, argument variable *a* should be of type int and b should be of type float,i.e., type of argument during function call and function definition should be same.
A function can be called with or without an argument.

3.5 Return Statement

Return statement is used for returning a value from function definition to calling function.

Syntax:

return (expression);

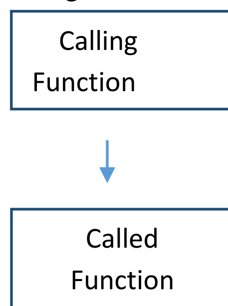
For example:

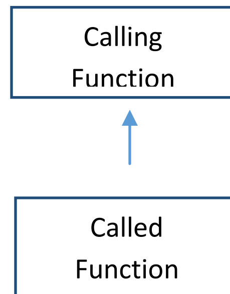
```
return a;
returna+b;
```

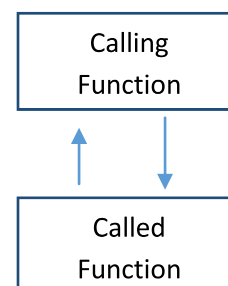
In above example, value of variable add in add() function is returned and that value is stored in variable *sum* in main() function. The data type of expression in return statement should also match the return type of function.

4. Inter Function communication

The Calling and called function are two separate entities, they need to communicate to exchange data. The data flow between the calling and called functions can be divided into three strategies







DOWNWARD FLOW UPWARD FLOW BI-DIRECTIONAL FLOW

Downward Flow:

In downward communication, the calling function sends data to the called function. No data flows in the opposite direction. In this strategy, copies of the data items are passed from the calling function to the called function. The called function may change the values passed, but the original values in the calling function remain untouched.

Example: **call-by-value**

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

```
                int  x=2,y=3,z=0;
                add(x,y);
        }
    void add(intx,int y)
        {       int z;
                z=x+y;
                printf("%d",z);
        }
```

Upward Flow:

Upward communication occurs when the called function sends data back to the called function without receiving any data from it.

Example: **call-by-reference**

```
        #include<stdio.h>
        void main()
        {
                int  x=2,y=3,z=0;
                modify(&x, &y);
                printf(("%d%d",x,y);
        }
    void modify(int *x,int *y)
        {
                *x=10;
                *y=20;
        }
```

Bi-directional Flow:

The strategy described for the upward direction can easily be augmented to allow the communication in both directions. The only difference is that the indirect reference must be used in both sides of the assignment variable.

```
        #include<stdio.h>
        void main()
        {
                int  x=2,y=3,z=0;
                modify(&x, &y);
                printf(("%d%d",x,y);
        }
    void modify(int *x,int *y)
        {
                *x=*x+10;
                *y=*y+20;
        }
```

5. Storage classes – auto, register, static, extern

The concept of storage class modifiers is used for the purpose of efficient utilization of variables and economisation of variables. These are used to tell the compiler how the variable that follows should be stored.

The General declaration form is

Storage-specifier Data type variable name;

There are four storage class specifiers supported by C. They are

1) Auto
2) Register
3) Static
4) Extern

auto: The variables generally declared in any function are called automatic variables. Even if the storage class is not specified they become auto by default.

For example :inta,b,c; is equal to auto inta,b,c;

The scope of these variables is that they are active only within the function in which they are declared. These are also called as 'Local Variables'.

Ex Pgm:

```
#include<stdio.h>
void main()
  {
autoint  x=2;
printf("%d",x);
  }
```

static: The variables that are declared and initialized in a function are normally initializing the variables for every function call and process further, but if we expect the function should initialize only once and process further then we go to the declaration of variable to be static.

Example: static int x;

The scope of these variables is within the function in which they are declared.

Ex pgm: :

```
#include<stdio.h>
void main()
  {
        add();
        add();
  }
void add()
{ staticint x=0;
x++;
Printf("%d",x);
}
```

Output: 1 2 3

register: The variables generally declared in functions normally allocate the memory in the RAM type of memory. The variables declared using register allocate the memory in a 'register' type of memory. The speed of register type of memory is more than the speed of RAM type of memory. The declaration looks like this.

Example: register inta,b,c;

The scope of these variables is that they are active only within the functions in which they are declared.

Example Program:
```
#include<stdio.h>
void main()
  {
    Register int  x=2;
printf("%d",x);
  }
```

extern: The variables declared in a file are to be referenced by the variables in another files then such kind of variables can be accessed among the files by declaring the variable to be extern;

Example program:

First.c
```
 #include<stdio.h>
#include<second.h>
Extern int m;
void main()
   {
int  m=2;
        add();
  }
```

second.h
```
int m;
void add()
{
        printf("%d", m);
}
```

6. Recursion

Recursion is a repetitive process in which a function calls itself. The process of repetition can be done in two ways using programming. They are

- Iterative Definition
- Recursive Definition

Iterative Definition:

Repeating a set of statements using loops is referred as Iteration.

Recursive Definition:

A repetitive function is defined recursively whenever the function appears within the definition itself. The recursive function has two elements : each call either solves one part of the problem or it reduces the size of the problem. The statement that solves the problem is known as the **base case**. The rest of the function is known as the **general case**. Each recursive function must have a base case.

Example: Finding Factorial of a number using recursive function.

Let the number be 4, then

4! =4x3x2x1

If n is the number then

n!=nx(n-1)x(n-2)x(n-3)x(n-4)……x(n-n)

or

n!=nx(n-1)!
(n-1)!=(n-1) x (n-2)!
(n-2)!=(n-2)x(n-3)!
………………………………………
(n-n)!=1

4!=4x3!   n=nx(n-1)!   If   n=4
3!=3x2!   n=nx(n-1)!   If   n=3
2!=2x1!   n=nx(n-1)!   If   n=2
1!=1x0!   n=nx(n-1)!   if   n=1
0!=1   n1=1   if   n=0

In the above example ,

n=nx(n-1)! Is considered as general case as it is valid for all n values

n!=1 is considered as base case as the value of 0! and 1! are 1 with which calculating the factorial terminates.

Base case: factorial(0)

General case: nxfactorial(n-1).

The execution order is as follows

4! = 4x3! (4x6=24)
3! = 3x2! (3x2=6)
2! = 2!x1 (2x1=2)
1! = 1x0! (1x1=1)
0!=1 (1=1)

The following are the rules for designing a recursive function:

1. First, determine the base case
2. Then, determine the general case
3. Finally, combine the base case and general case into a function

In combining the base and general cases into a function, we must pay careful attention to the logic. The base case, when reached, must terminate without a call to the recursive function; that is, it must execute a return.

Program:

```
#include<stdio.h>
int factorial(int );
void main()
{
        Intn,result;
        Printf("enter the number");
        Scanf("%d",&n);
        Result=factorial(n);
        Printf("%d",result);
}
```

```
Intfactorial(int n)
{
        if(n==0)
                Return 1;
        else
                Return(nxfactorial(n-1));
}
```

Limitations of Recursion:

1. Recursive solutions may involve extensive overhead because they use function calls.
2. Each time you make a call, you use up some of your memory allocation.

Example - Towers of Hanoi:

```
#include<stdio.h>
#include<conio.h>
void TOH(intn,char x, char y, char z);
void main()
{
int n;
printf("\n enter the number of plates :");
scanf("%d",&n);
TOH(n-1,'A','B','C');
getch();
}
void TOH(int n, char x, char y, char z)
{
        if(n>0)
        {
                TOH(n-1,x,z,y);
                printf("\n %c  -> %c",x,y);
                TOH(n-1,z,y,x);
        }
}
```

7. Passing Arrays to Functions:

An entire array can be transferred to a function as a parameter. To transfer an array to a function, the array name is enough without subscripts as an actual parameters within the function call.The corresponding formal parameters are written in same fashion, and must be declared as an array in formal parameters declaration.

Syntax:

```
voidfunccall(int,int[]);
main()                          void funccall(x,arr[])
{                               {
int a[10],n;
……                              }
funccall(n,a);
…..
}
```

Example Program demonstrating passing an array of 5 elements to function printarray;

```
#include<stdio.h>
void main()
    {
                inti,a[5];
                Printf("enter the elements");
                For(i=0;i<5;i++)
                Scanf("%d",&a[i]);
                Printarray(a);
    }
    voidprintarray(intarr[5])
    {
                For(i=0;i<5;i++)
                printf("%d",arr[i]);
    }
```

8. Strings - Basics

Strings are actually one-dimensional array of characters terminated by a **null**character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a **null**.

The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

```
char greeting[6]={ 'H', 'e', 'l', 'l', 'o', '\0'};
```

If you follow the rule of array initialization then you can write the above statement as follows

```
char greeting[] = "Hello";
```

Following is the memory presentation of the above defined string in C/C++

Index	0	1	2	3	4	5
Variable	H	e	l	l	o	\0
Address	0x23451	0x23452	0x23453	0x23454	0x23455	0x23456

Actually, you do not place the *null* character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print the above mentioned string –

```
#include<stdio.h>
#include<conio.h>
int main()
{
  char greeting[6]=”Hello”;
printf(“\n Greeting Message : %s \n”,greeting);
getch();
}
```

When the above code is compiled and executed, it produces the following result –

Greeting message: Hello

9. String input /output functions

C programming language provides many of the built-in functions to read given input and write data on screen, printer or in any file.

i. scanf() and printf() functions

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int i;
        printf(“\n Enter a value :”);
        scanf(“%d”,&i);
        printf(“\n You entered : %d”,i);
        getch();
}
```

When you will compile the above code, it will ask you to enter a value. When you will enter the value, it will display the value you have entered.

NOTE : printf() function returns the number of characters printed by it, and scanf() returns the number of characters read by it.

```
int i = printf("studytonight");
```

In this program i will get 12 as value, because studytonight has 12 characters.

ii. getchar() &putchar() functions

The getchar()function reads a character from the terminal and returns it as an integer. This function reads only single character at a time. You can use this method in the loop in case you want to read more than one characters. Theputchar()function prints the character passed to it on the screen and returns the same character. This function puts only single character at a time. In case you want to display more than one characters, use putchar() method in the loop.

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int c;
        printf("Enter a character :");
        c=getchar();
        putchar(c);
        getch();
}
```

When you will compile the above code,it will ask you to enter a value. When you will enter the value, it will display the value you have entered.

iii. gets() & puts() functions

The gets() function reads a line from **stdin** into the buffer pointed to by **s** until either a terminating newline or EOF (end of file). The puts() function writes the string **s** and a trailing newline to stdout.

```
#include<stdio.h>
#include<conio.h>
void main()
{
        charstr[100];
        printf("Enter a String :");
        gets(str);
        puts(str);
        getch();
}
```

When you will compile the above code,it will ask you to enter a string. When you will enter the string, it will display the value you have entered.

Difference between scanf() and gets() functions

The main difference between these two functions is that scanf() stops reading characters when it encounters a space, but gets() reads space as character too.
If you enter name as **Study Tonight** using scanf() it will only read and store **Study** and will leave the part after space. But gets() function will read it complete.

10. String handling functions:

C supports a wide range of functions that manipulate null-terminated strings −

S.N.	Function & Purpose
1	**strcpy(s1, s2);** Copies string s2 into string s1.
2	**strcat(s1, s2);** Concatenates string s2 onto the end of string s1.
3	**strlen(s1);** Returns the length of string s1.
4	**strcmp(s1, s2);** Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	**strchr(s1, ch);** Returns a pointer to the first occurrence of character ch in string s1.
6	**strstr(s1, s2);** Returns a pointer to the first occurrence of string s2 in string s1.

The following example uses some of the above-mentioned functions –

/* Write C program to reverse a string */

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
intn,i,j=0;
char str1[10],str2[10];
clrscr();
printf("\n Enter the string  :");
gets(str1);
```

```
 n=strlen(str1)-1;
for(i=n;i>=0;i--)
 {
str2[j]=str1[i];
j++;
 }
str2[j]='\0';
printf("\n reverse of the string is : %s",str2);
getch();
}
```

/* Write a C program copying from one string to another string */

```
#include<stdio.h>
#include<conio.h>
void main()
{
char str1[20],str2[20];
clrscr();
printf("\n enter the string  : ");
scanf("%s",str1);
strcpy(str2,str1);
printf("\n The string after copying is : ");
printf("\n %s",str2);
getch();
}
```

/* Write a C program to get the sub string from the given string by using index values */

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
intn,start,end,i=0;
char str1[10],str2[10];
clrscr();
printf("\n enter the string  :");
gets(str1);
printf("\n enter the starting index and ending index value : ");
scanf("%d %d",&start,&end);
for(n=start;n<=end;n++)
 {
str2[i]=str1[n];
i++;
 }
str2[i]='\0';
printf("sub string is %s",str2);
getch();
```

```
}
```

/* Write a C program to concatenating two strings

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
char str1[10],str2[10],c;
clrscr();
printf("enter the first string :");
scanf("%s",str1);
printf("enter the second string :");
scanf("%s",str2);
strcat(str1,str2);
printf("\n The string after concatenation is : %s ",str1);
getch();
}
```

11. Strings to functions

String can be passed to functions in similar manner as arrays as, string is also an array.

Example:

```
#inlcude<stdio.h>
#include<conio.h>
void display(char ch[]);
void main()
{
        char c[50];
        printf("Enter the string :");
        gets(c);
        display(c);
}
 void display(char ch[])
 {
        printf("\n String is : ");
        puts(ch);
 }
```

Here string c is passed from main() function to user-defined function display(). In function declaration, ch[] is the formal argument.