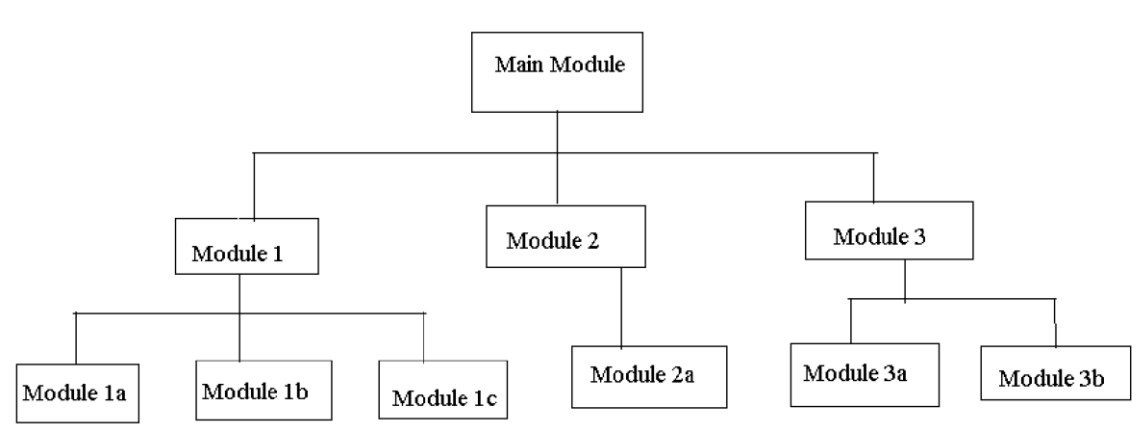
UNIT – V

FUNCTIONS-DESIGNING STRUCTURED PROGRAMS:

The planning for large programs consists of first understanding the problem as a whole, second breaking it into simpler, understandable parts. We call each of these parts of a program a **module** and the process of subdividing a problem into manageable parts **top-down design**.

The principles of top-down design and structured programming dictate that a program should be divided into a main module and its related modules. Each module is in turn divided into sub-modules until the resulting modules are intrinsic; that is, until they are implicitly understood without further division.

Top-down design is usually done using a visual representation of the modules known as a structure chart. The structure chart shows the relation between each module and its submodules. The structure chart is read top-down, left-right. First we read Main Module. Main Module represents our entire set of code to solve the problem.



Structure Chart

Moving down and left, we then read Module 1. On the same level with Module 1 are Module 2 and Module 3. The Main Module consists of three sub-modules. At this level, however we are dealing only with Module 1. Module 1 is further subdivided into three modules, Module

1a, Module 1b, and Module 1c. To write the code for Module 1, we need to write code for its three sub-modules.

The Main Module is known as a calling module because it has sub-modules. Each of the sub-modules is known as a called module. But because Modules 1, 2, and 3 also have submodules, they are also calling modules; they are both called and calling modules.

Communication between modules in a structure chart is allowed only through a calling module. If Module 1 needs to send data to Module 2, the data must be passed through the calling module, Main Module. No communication can take place directly between modules that do not have a calling-called relationship.

How can Module 1a send data to Module 3b?

It first sends data to Module 1, which in turn sends it to the Main Module, which passes it to Module 3, and then on to Module 3b.

The technique used to pass data to a function is known as parameter passing. The parameters are contained in a list that is a definition of the data passed to the function by the caller. The list serves as a formal declaration of the data types and names.

FUNCTIONS :

A function is a self contained program segment that carries out some specific well defined tasks.

Advantages of functions:

1. Write your code as collections of small functions to make your program modular

2.Structured programming

3.Code easier to debug

4.Easier modification

5.Reusable in other programs

Function Definition :

type func_name(parameter list)

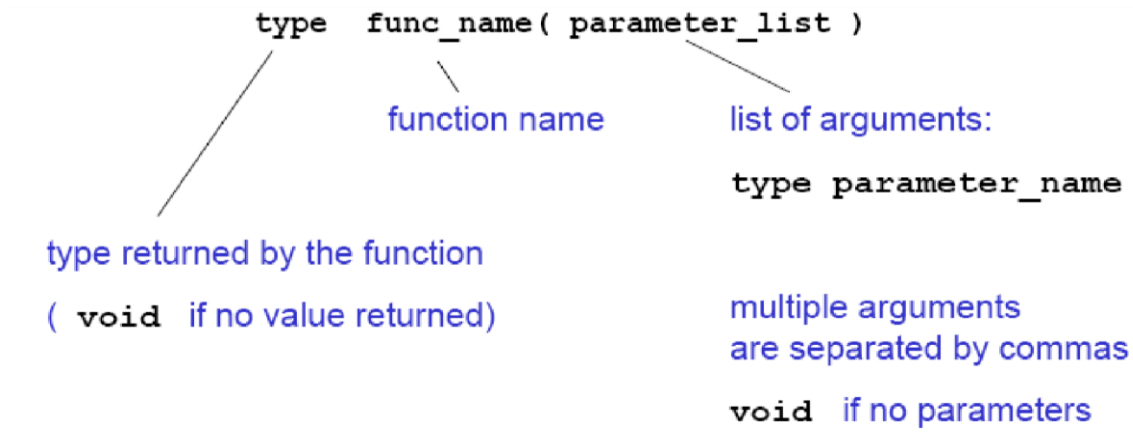
{

declarations;

statements;

}

FUNCTION HEADER :



Examples:

```
int fact(int n)
void error_message(int errorcode)
double initial_value(void)
int main(void)
```

Usage:

```
a = fact(13);
error_message(2);
x=initial_value();
```

FUNCTION PROTOTYPES

If a function is not defined before it is used, it must be declared by specifying the return type and the types of the parameters.

double sqrt(double);

Tells the compiler that the function **sqrt()** takes an argument of type double and returns a double. This means, incidentally, that variables will be cast to the correct type; so **sqrt(4)** will return the correct value even though **4** is **int** not **double**. These function prototypes are placed at the top of the program, or in a separate header file, **file.h**, included as

#include "file.h"

Variable names in the argument list of a function declaration are optional:

void f (char, int); void f (char c, int i); /*equivalent but makes code more readable */

If all functions are defined before they are used, no prototypes are needed. In this case, **main()** is the last function of the program.

SCOPE RULES FOR FUNCTIONS :

Variables defined within a function (including main) are local to this function and no other function has direct access to them. The only way to pass variables to function is as parameters. The only way to pass (a single) variable back to the calling function is via the return statement

Ex: int func

(int n)

{

```
printf("%d\n",b);

return n;


}//b not defined locally!
int main (void)

{

int a = 2, b = 1,

c; c = func(a);

return 0;


}
```

FUNCTION CALLS :

When a function is called, expressions in the parameter list are evaluated (in no particular order!) and results are transformed to the required type. Parameters are copied to local variables for the function and function body is executed when return is encountered, the function is terminated and the result (specified in the return statement) is passed to the calling function (for example main).

```
Ex: int fact

(int n)


{

int i, product = 1;

for (i = 2; i <= n;

++i) product *= i;

return product;


}

int main (void)
```

```
{ int i = 12;

printf("%d",fact

(i)); return 0;

}
```

TYPES OF FUNCTIONS:

USER DEFINED FUNCTIONS:

Every program must have a main function to indicate where the program has to begin its execution. While it is possible to code any program utilizing only main function, it leads to a number of problems. The program may become too large and complex and as a result task of debugging, testing and maintaining becomes difficult. If a program is divided into functional parts, then each part may be independently coded and later combined into a single unit, these subprograms called "functions" are much easier to understand debug and test.

There are times when some types of operation for calculation is repeated many times at many points through out a program. For instance, we might use the factorial of a number at several points in the program. In such situations, we may repeat the program statements whenever they are needed. Another approach is to design a function that can be called and used whenever required.

The advantages of using functions are

1. It facilitates top-down modular programming.
2. The length of a source program can be reduced by using functions at appropriate places.
3. It is easy to locate and isolate a faculty function for further investigations.
4. A function may be used by many other programs.

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 main program and returns a value. The inner details of the program are invisible to the rest of program.

The form of the C functions.

```
function-name ( argument list )  argument

declaration;

        {

                local variable declarations;

                executable statement 1;
                executabie       statement 2;

                - - - - - - - - - -

                - - - - - - - - - -

                - - - - - - - - - -

                return (expression) ;

         }
```

The return statement is the mechanism for returning a value to the calling function. All functions by default returns int type data. we can force a function to return a particular type of data by using a type specifier in the header.

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

STANDARD LIBRARY FUNCTIONS AND HEADER FILES:

C functions can be classified into two categories, namely, library functions and userdefined functions .Main is the example of user-defined functions.Printf and scanf belong to the category of library functions. The main difference between these two categories is that library functions are not required to be written by us where as a user-defined function has to be developed by the user at the time of writing a program.However, a user-defined function can later become a part of the c program library.

ANSI C Standard Library Functions :

The C language is accompanied by a number of library functions that perform various tasks.The ANSI committee has standardized header files which contain these functions.

Some of the Header files are:

<ctype.h> character testing and conversion functions.

<math.h> Mathematical functions

<stdio.h> standard I/O library functions

<stdlib.h> Utility functions such as string conversion routines memory allocation routines , random number generator,etc.

<string.h> string Manipulation functions

<time.h> Time Manipulation functions

MATH.H

The math library contains functions for performing common mathematical operations. Some of the functions are :

abs : returns the absolute value of an integer x

cos : returns the cosine of x, where x is in radians

exp: returns "e" raised to a given power

fabs: returns the absolute value of a float x

log: returns the logarithm to base e

log10: returns the logarithm to base 10 **pow :** returns a given number raised to another number **sin :** returns the sine of x, where x is in radians **sqrt :** returns the square root of x

tan : returns the tangent of x, where x is in radians **ceil :** The ceiling function rounds a number with a decimal part up to the next highest integer (written mathematically as ⌈x⌉) **floor :** The floor function rounds a number with a decimal part down to the next lowest integer (written mathematically as ⌊x⌋)

STRING.H

There are many functions for manipulating strings. Some of the more useful are:

strcat : Concatenates (i.e., adds) two

strings **strcmp:** Compares two strings

strcpy: Copies a string

strlen: Calculates the length of a string (not including the

null) **strstr:** Finds a string within another string **strtok:**

Divides a string into tokens (i.e., parts)

STDIO.H

Printf: Formatted printing to stdout

Scanf: Formatted input from stdin

Fprintf: Formatted printing to a file

Fscanf: Formatted input from a file

Getc: Get a character from a stream (e.g, stdin or a

file) **putc:** Write a character to a stream (e.g, stdout

or a file) **fgets:** Get a string from a stream **fputs:**

Write a string to a stream **fopen:** Open a file **fclose:**

Close a file

STDLIB.H

Atof: Convert a string to a double (not a float)

Atoi: Convert a string to an int

Exit: Terminate a program, return an integer value **free:** Release allocated memory **malloc:** Allocate memory **rand:** Generate a pseudo-random number **system:** Execute an external command

TIME.H

This library contains several functions for getting the current date and time.

Time: Get the system time

Ctime: Convert the result from time() to something meaningful

The following table contains some more header files:

<assert.h>	Contains the assert macro, used to assist with detecting logical errors and other types of bug in debugging versions of a program.
<complex.h>	A set of functions for manipulating complex numbers. (New with **C99**)
<ctype.h>	Contains functions used to classify characters by their types or to convert between upper and lower case in a way that is independent of the used character set (typically ASCII or one of its extensions, although implementations utilizing EBCDIC are also known).
<errno.h>	For testing error codes reported by library functions.
<fenv.h>	For controlling floating-point environment. (New with **C99**)

<float.h>	Contains defined constants specifying the implementation-specific properties of the floating-point library, such as the minimum difference between two different floating-point numbers (_EPSILON), the maximum number of digits of accuracy (_DIG) and the range of numbers which can be represented (_MIN, _MAX).
<inttypes.h>	For precise conversion between integer types. (New with **C99**)
<iso646.h>	For programming in ISO 646 variant character sets. (New with **NA1**)
<limits.h>	Contains defined constants specifying the implementation-specific properties of the integer types, such as the range of numbers which can be represented (_MIN, _MAX).
<locale.h>	For setlocale() and related constants. This is used to choose an appropriate locale.
<math.h>	For computing common mathematical functions
<setjmp.h>	Declares the macros setjmp and longjmp, which are used for non-local exits
<signal.h>	For controlling various exceptional conditions
<stdarg.h>	For accessing a varying number of arguments passed to functions.
<stdbool.h>	For a boolean data type. (New with **C99**)
<stdint.h>	For defining various integer types. (New with **C99**)
<stddef.h>	For defining several useful types and macros.

<stdio.h>	Provides the core input and output capabilities of the C language. This file includes the venerable `printf` function.
<stdlib.h>	For performing a variety of operations, including conversion, pseudo-random numbers, memory allocation, process control, environment, signalling, searching, and sorting.
<string.h>	For manipulating several kinds of strings.
<tgmath.h>	For type-generic mathematical functions. (New with **C99**)
<time.h>	For converting between various time and date formats.
<wchar.h>	For manipulating wide streams and several kinds of strings using wide characters - key to supporting a range of languages. (New with **NA1**)
<wctype.h>	For classifying wide characters. (New with **NA1**)

CATEGORIES OF FUNCTIONS :

A function depending on whether arguments are present or not and whether a value is returned or not may belong to.

1. Functions with no arguments and no return values.
2. Functions with arguments and no return values.
3. Functions with arguments and return values.

1. Functions with no arguments and no return values :

When a function has no arguments, it does not receive any data from calling function. In effect, there is no data transfer between calling function and called function.

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

{
        printline();
        value();
printline();

}
printline()
{
        int i;
        for(i=1;i<=35;i++0)
        printf("%c","-");
        printf("\n");

}
value()
{
        int year, period;       float
inrate,sum,principal;  printf("Enter Principal,
Rate,Period");
scanf("%f%f%f",&principal,&inrate,&period);
sum=principal;          year=1;
```

```
        while(year<=period)

        {

                sum=sum*(1+inrate);

                year=year+1;


        }
        printf("%f%f%d%f",principal,inrate,period,sum);

}
```

2. Arguments but no return values :

The function takes argument but does not return value.

The actual (sent through main) and formal(declared in header section) should match in number, type and order.

In case actual arguments are more than formal arguments, extra actual arguments are discarded. On other hand unmatched formal arguments are initialized to some garbage values.

```
#include<stdio.h>

main()


{

        float

prin,inrate;    int

period;


        printf("Enter principal amount, interest");

printf("rate and period\n");

scanf("%f%f%d",&principal,&inrate,&period);
```

```
printline('z');   value(principal, inrate, peiod);

printline('c');


}

printline(ch)

char ch;


{

        int i;
        for(i=1;i<=52;i++)

                printf("%c",ch);

                printf("\n");


}
```

PARAMETER PASSING TECHNIQUES:

Parameter passing mechanism in „C" is of two types.

1. Call by Value
2. Call by Reference.

The process of passing the actual value of variables is known as Call by Value.

The process of calling a function using pointers to pass the addresses of variables is known as Call by Reference. The function which is called by reference can change the value of the variable used in the call.

Example of Call by Value:

```
        #include <stdio.h>
        void swap(int,int);
        main()
        {
                int a,b;
                printf("Enter the Values of a and
b:");           scanf("%d%d",&a,&b);
        printf("Before Swapping \n");
                printf("a = %d \t b = %d", a,b);
```

```
                swap(a,b);
                printf("After Swapping \n");
                printf("a = %d \t b = %d", a,b);
        }


        void swap(int a, int b)
        {
                int
temp;           temp
= a;            a = b;
        b = temp;
        }
```

Example of Call by Reference:

```
        #include<stdio.h>
        main()
        {
                int
a,b;            a =
10;             b =
20;
                swap (&a, &b);
                printf("After Swapping \n");
                printf("a = %d \t b = %d", a,b);
        }
        void swap(int *x, int *y)
        {
                int temp;
        temp = *x;
                *x = *y;
                *y = temp;
        }
```

STORAGE CLASSES :

In 'C' a variable can have any one of four Storage Classes.

1. Automatic Variables
2. External Variables 3. Static Variables
4. Register Variables

SCOPE :

The Scope of variable determines over what parts of the program a variable is actually available for use.

LONGEVITY :

Longevity refers to period during which a variable retains a given value during execution of a program (alive). So Longevity has a direct effect on utility of a given variable.

The variables may also be broadly categorized depending on place of their declaration as internal(local) or external(global). Internal variables are those which are declared within a particular function, while external variables are declared outside of any function.

AUTOMATIC VARIABLES :

They are declared inside a function in which they are to be utilized. They are created when function is called and destroyed automatically when the function is exited, hence the name automatic. Automatic Variables are therefore private (or local) to the function in which they are declared. Because of this property, automatic variables are also referred to as local or internal variables.

By default declaration is automatic. One important feature of automatic variables is that their value cannot be changed accidentally by what happens in some other function in the program. #include<stdio.h> main()

```
{
        int m=1000;
func2();
printf("%d\n",m);

}
```

```
func1()

{

        int m=10;

printf("%d\n",m);


}
func2()

{

        int m=100;

func1();

printf("%d",m);


}
```

First, any variable local to main will normally live throughout the whole program, although it is active only in main.

Secondly, during recursion, nested variables are unique auto variables, a situation similar to function nested auto variables with identical names.

EXTERNAL VARIABLES :

Variables that are both alive and active throughout entire program are known as external variables. They are also known as Global Variables. In case a local and global have same name local variable will have precedence over global one in function where it is declared.

```
#include<stdio.h>
```

```
int x;
main()
{
        x=10;
        printf("%d",x);
printf("x=%d",fun1());
printf("x=%d",fun2());
printf("x=%d",fun3());



}
fun1()
{
        x=x+10;
return(x);

}
fun2()
{
        int x;
x=1;

        return(x);
}
```

```
fun3()

{

x=x+10;

return(x);

}
```

An extern within a function provides the type information to just that one function.

STATIC VARIABLES :

The value of Static Variable persists until the end of program. A variable can be declared Static using Keyword Static like Internal & External Static Variables are differentiated depending whether they are declared inside or outside of auto variables, except that they remain alive throughout the remainder of program.

```
#include<stdio.h>

main()

{

        int I;

        for (I=1;I<=3;I++)

stat();

}

stat()

{

        static int x=0;
```

```
    x=x+1;

    printf("x=%d\n",x);

}
```

REGISTER VARIABLES :

We can tell the Compiler that a variable should be kept in one of the machines registers, instead of keeping in the memory. Since a register access is much faster than a memory access, keeping frequently accessed variables in register will lead to faster execution

Syntax:

```
register int Count.
```

RECURSIVE FUNCTIONS:

Recursion is a repetitive process in which a function calls itself (or) A function is called recursive if it calls itself either directly or indirectly. In C, all functions can be used recursively.

Example: Fibonacci Numbers

A recursive function for Fibonacci numbers (0,1,1,2,3,5,8,13...)

```
/* Function with
recursion*/ int fibonacci(int
n)

{

if (n <= 1) return n; else return

(fibonacci(n-1) + fibonacci(n-2));


}
```

With recursion 1.4×10^9 function calls needed to find the 43rd Fibonacci number(which has the value 433494437) .If possible, it is better to write iterative functions.

```
int factorial (int n)  /* iterative version */

{ for ( ; n > 1;

--n) product

*= n; return

product;


}
```