FUNCTIONS

A function is itself a block of code which can solve simple or complex task/calculations.

A function performs calculations on the data provided to it is called "parameter" or "argument".

A function always returns single value result.

Types of function:

1. Built in functions(Library functions)

a.) Inputting Functions.

b.) Outputting functions.

2. User defined functions.

a.) fact();

b.) sum();

Parts of a function:

1. Function declaration/Prototype/Syntax.

2. Function Calling.

3. Function Definition.

1.)Function Declaration:

Syntax: <return type > <function name>(<type of argument>)

The declaration of function name, its argument and return type is called function declaration.

2.) Function Calling:

The process of calling a function for processing is called function calling.

Syntax: <var_name>=<function_name>(<list of arguments>).

3.) Function defination:

The process of writing a code for performing any specific task is called function defination.

Syntax:

<return type><function name>(<type of arguments>)

{

<statement-1>

<statement-2>

return(<vlaue>)

}

Example: program based upon function:

WAP to compute cube of a no. using function.

```
#include<stdio.h>
#include<conio.h>
void main()
{
 int c,n;
int cube(int);
printf("Enter a no.");
scanf("%d",&n);
c=cube(n);
printf("cube of a no. is=%d",c);
}
int cube(int n)
{
c=n*n*n;
return(c);
}
```

WAP to compute factorial of a no. using function:

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

```
{
int n,f=1;
int fact(int)
printf("Enter a no.");
scanf("%d",&n);
f=fact(n);
printf("The factorial of a no. is:=%d",f);
}
int fact(int n)
int f=1;
{
 for(int i=n;i>=n;i--)
{
 f=f*i;
}
return(f);
}
```

Recursion

Firstly, what is nested function?

When a function invokes another function then it is called nested function.

But,

When a function invokes itself then it is called recursion.

NOTE: In recursion, we must include a terminating condition so that it won't execute to infinite time.

Example: program based upon recursion:

WAP to compute factorial of a no. using Recursion:

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

```
{
int n,f;
int fact(int)
printf("Enter a no.");
scanf("%d",&n);
f=fact(n);
printf("The factorial of a no. is:=%d",f);
}
int fact(int n)
int f=1;
{
 if(n=0)
return(f);
else
return(n*fact(n-1));
}
```

Passing parameters to a function:

Firstly, what are parameters?

parameters are the values that are passed to a function for processing.

There are 2 types of parameters:

a.) Actual Parameters.

b.) Formal Parameters.

a.) Actual Parameters:

These are the parameters which are used in main() function for function calling.

Syntax: <variable name>=<function name><actual argument>

Example: f=fact(n);

b.) Formal Parameters.

These are the parameters which are used in function defination for processing.

Methods of parameters passing:

1.) Call by reference.

2.) Call by value.

1.) Call by reference:

In this method of parameter passing , original values of variables are passed from calling program to function.

Thus,

Any change made in the function can be reflected back to the calling program.

2.) Call by value.

In this method of parameter passing, duplicate values of parameters are passed from calling program to function defination.

Thus,

Any change made in function would not be reflected back to the calling program.

Example: Program based upon call by value:

```
# include<stdio.h>
# include<conio.h>
void main()
{
int a,b;
a=10;
b=20;
void swap(int,int)
printf("The value of a before swapping=%d",a);
printf("The value of b before swapping=%d",b);
void swap(a,b);
printf("The value of a after swapping=%d",a);
printf("The value of b after swapping=%d",b);
```

```
}
void swap(int x, int y)
{
int t;
t=x;
x=y;
y=t;
}
```

STORAGE CLASSES

Every Variable in a program has memory associated with it.

Memory Requirement of Variables is different for different types of variables.

In C, Memory is allocated & released at different places

Term	**Definition**
Scope	Region or Part of Program in which Variable is accessible
Extent	Period of time during which memory is associated with variable
Storage Class	Manner in which memory is allocated by the Compiler for Variable **Different Storage Classes**

Storage class of variable Determines following things

Where the variable is stored

Scope of Variable

Default initial value

Lifetime of variable

A. Where the variable is stored:

Storage Class determines the location of variable, where it is declared. Variables declared with auto storage classes are declared inside main memory whereas variables declared with keyword register are stored inside the CPU Register.

B. Scope of Variable

Scope of Variable tells compile about the visibility of Variable in the block. Variable may have Block Scope, Local Scope and External Scope. A scope is the context within a computer program in which a variable name or other identifier is valid and can be used, or within which a declaration has effect.

C. Default Initial Value of the Variable

Whenever we declare a Variable in C, garbage value is assigned to the variable. Garbage Value may be considered as initial value of the variable. C Programming have different storage classes which has different initial values such as Global Variable have Initial Value as 0 while the Local auto variable have default initial garbage value.

D. Lifetime of variable

Lifetime of the = Time Of variable Declaration - Time of Variable Destruction

Suppose we have declared variable inside main function then variable will be destroyed only when the control comes out of the main .i.e end of the program.

Different Storage Classes:

Auto Storage Class

Static Storage Class

Extern Storage Class

Register Storage Class

Automatic (Auto) storage class

This is default storage class

All variables declared are of type Auto by default

In order to Explicit declaration of variable use 'auto' keyword

auto int num1 ; // Explicit Declaration

Features:

Storage	Memory
Scope	Local / Block Scope

Life time	Exists as long as Control remains in the block
Default initial Value	Garbage

Example

```
void main()
{
 auto mum = 20 ;
 {
    auto num = 60 ;
        printf("nNum : %d",num);
 }
 printf("nNum : %d",num);
}
```

Output :

Num : 60

Num : 20

Note :

Two variables are declared in different blocks , so they are treated as different variables

External (extern) storage class in C Programming

Variables of this storage class are "Global variables"

Global Variables are declared outside the function and are accessible to all functions in the program

Generally , External variables are declared again in the function using keyword extern

In order to Explicit declaration of variable use 'extern' keyword

extern int num1 ; // Explicit Declaration

Features :

Storage	Memory
Scope	Global / File Scope
Life time	Exists as long as variable is running Retains value within the function
Default initial Value	Zero

Example

```
int num =  75 ;

void display();

void main()
{
 extern int num ;
          printf("nNum : %d",num);
 display();
}

void display()
{
 extern int num ;
          printf("nNum : %d",num);
}
```

Output :

Num : 75

Num : 75

Note :

Declaration within the function indicates that the function uses external variable

Functions belonging to same source code , does not require declaration (no need to write extern)

If variable is defined outside the source code , then declaration using extern keyword is required

Static Storage Class

The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

In C programming, when **static** is used on a class data member, it causes only one copy of that member to be shared by all the objects of its class.

```
#include <stdio.h>

/* function declaration */
void func(void);

static int count = 5; /* global variable */

main() {

  while(count--) {
    func();
  }

  return 0;
}

/* function definition */
void func( void ) {

  static int i = 5; /* local static variable */
  i++;

  printf("i is %d and count is %d\n", i, count);
}
```

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

```
i is 6 and count is 4
i is 7 and count is 3
i is 8 and count is 2
i is 9 and count is 1
i is 10 and count is 0
```

Register Storage Class

register keyword is used to define local variable.

Local variable are stored in register instead of **RAM**.

As variable is stored in register, the **Maximum size of variable = Maximum Size of Register**

unary operator [&] is not associated with it because Value is not stored in RAM instead it is stored in Register.

This is generally used for **faster access**.

Common use is "**Counter**"

Syntax

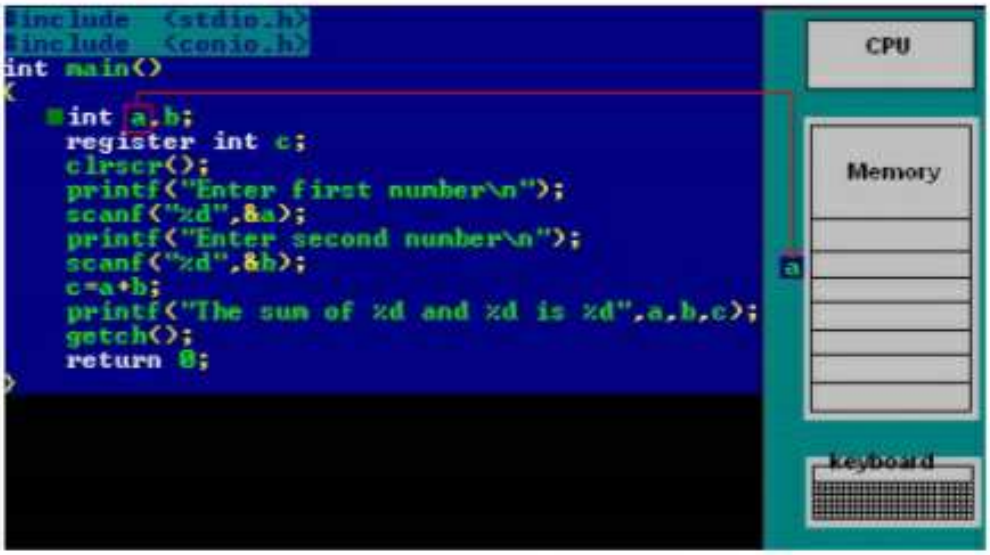
```
{
register int count;
}
```

Register storage classes example

```
#include<stdio.h>

int main()
{
int num1,num2;
register int sum;

printf("\nEnter the Number 1 : ");
scanf("%d",&num1);

printf("\nEnter the Number 2 : ");
scanf("%d",&num2);

sum = num1 + num2;

printf("\nSum of Numbers : %d",sum);

return(0);
}
```

Explanation of program

Refer below animation which depicts the register storage classes –



In the above program we have declared two variables num1,num2. These two variables are stored in RAM.

Another variable is declared which is stored in register variable.Register variables are stored in the register of the microprocessor.Thus memory access will be faster than other variables.

If we try to declare more register variables then it can treat variables asAuto storage variables as memory of microprocessor is fixed and limited.

Why we need Register Variable ?

Whenever we declare any variable inside C Program then memory will be randomly allocated at particular memory location.

We have to keep track of that memory location. We need to access value at that memory location using ampersand operator/Address Operator i.e (&).

If we store same variable in the register memory then we can access that memory location directly without using the Address operator.

Register variable will be accessed faster than the normal variable thus increasing the operation and program execution. Generally we use register variable as Counter.

Note : It is not applicable for arrays, structures or pointers.

Summary of register Storage class

Keyword	**register**
Storage Location	CPU Register

Keyword	register
Initial Value	Garbage
Life	Local to the block in which variable is declared.
Scope	Local to the block.

Preprocessor directives

Before a C program is compiled in a compiler, source code is processed by a program called preprocessor. This process is called preprocessing.

Commands used in preprocessor are called preprocessor directives and they begin with "#" symbol.

Below is the list of preprocessor directives that C language offers.

S.no	Preprocessor	Syntax	Description
1	Macro	#define	This macro defines constant value and can be any of the basic data types.
2	Header file inclusion	#include <file_name>	The source code of the file "file_name" is included in the main program at the specified place
3	Conditional compilation	#ifdef, #endif, #if, #else, #ifndef	Set of commands are included or excluded in source program before compilation with respect to the

			condition
4	Other directives	#undef, #pragma	#undef is used to undefine a defined macro variable. #Pragma is used to call a function before and after main function in a C program

A program in C language involves into different processes. Below diagram will help you to understand all the processes that a C program comes across.

EXAMPLE PROGRAM FOR #DEFINE, #INCLUDE PREPROCESSORS IN C:

#define – This macro defines constant value and can be any of the basic data types.
#include <file_name> – The source code of the file "file_name" is included in the main C program where "#include <file_name>" is mentioned.

```
#include <stdio.h>

#define height 100
#define number 3.14
#define letter 'A'
#define letter_sequence "ABC"
#define backslash_char '\?'

void main()
{
  printf("value of height   : %d \n", height );
  printf("value of number : %f \n", number );
  printf("value of letter : %c \n", letter );
  printf("value of letter_sequence : %s \n", letter_sequence);
  printf("value of backslash_char  : %c \n", backslash_char);

}
```

OUTPUT:

```
value of height : 100
value of number : 3.140000
value of letter : A
value of letter_sequence : ABC
value of backslash_char : ?
```

EXAMPLE PROGRAM FOR CONDITIONAL COMPILATION DIRECTIVES:

A) EXAMPLE PROGRAM FOR #IFDEF, #ELSE AND #ENDIF IN C:

"#ifdef" directive checks whether particular macro is defined or not. If it is defined, "If" clause statements are included in source file.

Otherwise, "else" clause statements are included in source file for compilation and execution.

```
#include <stdio.h>
#define RAJU 100

int main()
{
  #ifdef RAJU
  printf("RAJU is defined. So, this line will be added in " \
        "this C file\n");
  #else
  printf("RAJU is not defined\n");
  #endif
  return 0;
}
```

OUTPUT:

```
RAJU is defined. So, this line will be added in this C file
```

B) EXAMPLE PROGRAM FOR #IFNDEF AND #ENDIF IN C:

#ifndef exactly acts as reverse as #ifdef directive. If particular macro is not defined, "If" clause statements are included in source file.

Otherwise, else clause statements are included in source file for compilation and execution.

```
#include <stdio.h>
#define RAJU 100
```

```
int main()
{
  #ifndef SELVA
  {
    printf("SELVA is not defined. So, now we are going to " \
        "define here\n");
    #define SELVA 300
  }
  #else
  printf("SELVA is already defined in the program");

  #endif
  return 0;

}
```

OUTPUT:

```
SELVA is not defined. So, now we are going to define here
```

C) EXAMPLE PROGRAM FOR #IF, #ELSE AND #ENDIF IN C:

"If" clause statement is included in source file if given condition is true.

Otherwise, else clause statement is included in source file for compilation and execution.

```
#include <stdio.h>
#define a 100
int main()
{
  #if (a==100)
  printf("This line will be added in this C file since " \
        "a \= 100\n");
  #else
  printf("This line will be added in this C file since " \
        "a is not equal to 100\n");
  #endif
  return 0;
}
```

OUTPUT:

```
This line will be added in this C file since a = 100
```

EXAMPLE PROGRAM FOR UNDEF IN C:

This directive undefines existing macro in the program.

```
#include <stdio.h>

#define height 100
void main()
{
  printf("First defined value for height    : %d\n",height);
  #undef height          // undefining variable
  #define height 600     // redefining the same for new value
  printf("value of height after undef \& redefine:%d",height);
}
```

OUTPUT:

```
First defined value for height : 100
value of height after undef & redefine : 600
```

EXAMPLE PROGRAM FOR PRAGMA IN C:

Pragma is used to call a function before and after main function in a C program.

```
#include <stdio.h>

void function1( );
void function2( );

#pragma startup function1
#pragma exit function2

int main( )
{
  printf ( "\n Now we are in main function" ) ;
  return 0;
}

void function1( )
{
  printf("\nFunction1 is called before main function call");
}

void function2( )
{
  printf ( "\nFunction2 is called just before end of " \
        "main function" ) ;"
}
```

OUTPUT:

```
Function1 is called before main function call
Now we are in main function
Function2 is called just before end of main function
```

MORE ON PRAGMA DIRECTIVE IN C:

S.no	Pragma command	description
1	#Pragma startup <function_name_1>	This directive executes function named "function_name_1" before
2	#Pragma exit <function_name_2>	This directive executes function named "function_name_2" just before termination of the program.
3	#pragma warn – rvl	If function doesn't return a value, then warnings are suppressed by this directive while compiling.
4	#pragma warn – par	If function doesn't use passed function parameter , then warnings are suppressed
5	#pragma warn – rch	If a non reachable code is written inside a program, such warnings are suppressed by this directive.