Unit 5
Functions

Syllabus : Introduction to Functions, Function Declaration and Definition, Function call Return Types and Arguments,modifying parameters inside functions using pointers, arrays as parameters. Storage classes, Basics of File Handling

* Introduction to Functions:

Function Definition :

function is a **group of statements** that together perform a task. Every C program has **at least one main function**.

Or

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

Any 'C' program contain at least one function i.e main().

Types of Functions:

There are 2 types of functions in C programming

1. Library functions:

These functions are defined in the **library of C compiler** which are used frequently in the C program. functions which are declared in the C header files such as scanf(), **printf(), gets(), puts(), ceil(), floor() etc.**

- These functions are written by designers of c compiler.
- C supports many built in functions like
  - ☐ Mathematical functions
  - ☐ String manipulation functions
  - ☐ Input and output functions
  - ☐ Memory management functions

EXAMPLE:

- ☐ pow(x,y)-computes x^y
- ☐ sqrt(x)-computes square root of x
- ☐ printf()- used to print the data on the screen
- ☐ scanf()-used to read the data from keyboard.

2. User defined functions:

- ☐ The functions written by the programmer /user to do the specific tasks are called user defined function(UDF's).
- ☐ the functions which are created by the C programmer, so that he/she can use it many times. It reduces the complexity of a big program and optimizes the code.
- ☐ The user can construct their own functions to perform some specific task. This type of functions created by the user is termed as User defined functions.

Elements of User Defined Function:

The Three Elements of User Defined function structure consists of :

1. Function Definition
2. Function Declaration
3. Function call

1. Function Definition:

➢ It contains the actual statements which are to be executed. It is the most important aspect to which the control comes when the function is called. Here, we must notice that only one value can be returned from the function.

➢ A program Module written to achieve a specific task is called as function definition.

Syntax:

```
Return_type   function_name( parameter list )
{
          body of the function
}
```

Return Type: A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the **keyword void**.

Function Name: This is the actual name of the function. The function name and the parameter list together constitute the function signature.

Parameters: A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

Function Body: The function body contains a collection of statements that define what the function does.

2. Function Declarations:

A function declaration tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts:

Syntax: return_type function_nam e(param eter list);

Example: int m ax(int num 1, int num 2);

3. Calling a Function:

- While creating a C function, you give a definition of what the function has to do. To use a function,you will have to call that function to perform the defined task.
- When a program calls a function, program control is transferred to the called function. A called function performs defined task and when its return statement is executed or when its function ending closing brace is reached, it returns program control back to the main program.
- To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value.

Syntax: function_nam e(param eter list);

Example: max(int num 1, int num 2);

For example:

```
int m ax(int num 1, int num 2);              /* function declaration * /

int m ain ()
{
int a = 100;          /* local variable definition * /

int b = 200;
int ret;

ret = m ax(a, b);          /* calling a function to get m ax value * /

printf( "Max value is : %d\n", ret );
return 0;
}

int m ax(int num 1, int num 2)               /* function returning the m ax between two numbers * /
{
int result;                    /* local variable declaration * /
if (num 1 > num 2)
result = num 1;
else
result = num 2;
return result;
}                                        output : max value is 200
```

Advantage of functions in C :

There are the following advantages of C functions.

- ☐ By using functions, we can avoid rewriting same logic/code again and again in a program.
- ☐ We can call C functions any number of times in a program and from any place in a program. We can track a large C program easily when it is divided into multiple functions.
- ☐ Reusability is the main achievement of C functions.
- ☐ However, Function calling is always a overhead in a C program.

*Function call Return Types and Arguments: (or)

* Categories of Functions:

A function may or may not accept any argument. It may or may not return any value. Based on these facts,

There are 4 different aspects of function calls.

1. **Function with no parameters and no return values**
2. **Function with no parameters and return values.**
3. **Function with parameters and no return values**
4. **Function with parameters and return values**

1. Function with no parameters and no return values:

In this category **no data** is transferred from **calling function to called function**, hence called function cannot receive any values.

- ➢ In the above example,no arguments are passed to user defined function **add( ).**
- ➢ Hence no parameter are defined in function header.
- ➢ When the control is transferred from calling function to called function a ,and b values are read,they are added,the result is printed on monitor.
- ➢ When return statement is executed ,control is transferred from called function/add to calling function/main

Calling function	Called function

/*program to find sum of two numbers
using function*/
#include<stdio.h>
void add( );
void main()
{
add();
}
	void add ( )
	{
	int sum;
	printf("enter a and b values\n");
	scanf("%d%d",&a,&b);
	sum=a+b;
	printf("\n The sum is %d", sum);
	return;
	}

2. Function with no parameters and return values.

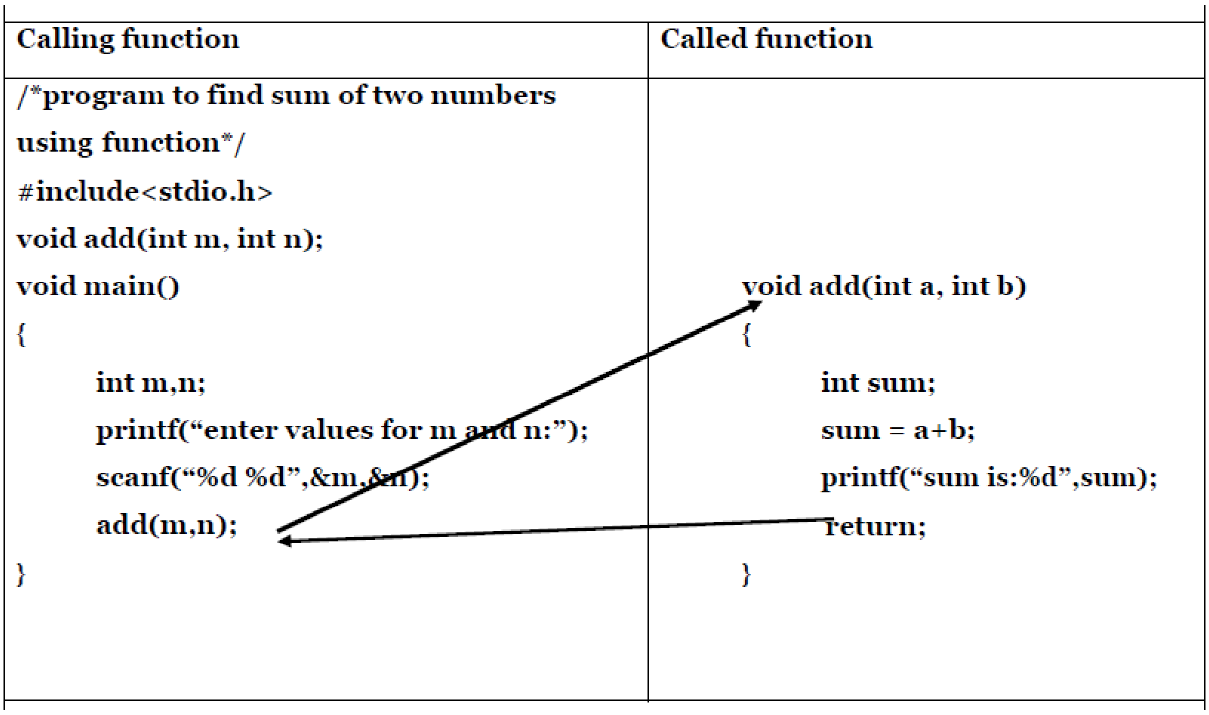
- In this category **there is no data transfer from the calling function to the called function.**
- But, there is data transfer from called function to the calling function.
- No arguments are passed to the function add( ). So, no parameters are defined in the function header
- When the function returns a value, the calling function receives one value from the called function and assigns to variable result.
- The result value is printed in calling function

Calling function	Called function

/*program to find sum of two numbers
using function*/
#include<stdio.h>
int add();
void main()
{
int result;
result=add( );
printf("sum is:%d",result);
}
	int add( ) /* function header */
	{
	int a,b,sum;
	printf("enter values for a and
	b:");
	scanf("%d %d",&a,&b);
	sum= a+b;
	return sum;
	}

3. Function with parameters and no return values

- In this category, **there is data transfer** from the calling function to the called function using parameters.
- **But there is no data transfer from** called function to the calling function.
- The values of actual parameters m and n are copied into formal parameters a and b.
- The value of a and b are added and result stored in sum is displayed on the screen in called function itself.



Calling function

```
/*program to find sum of two numbers
using function*/
#include<stdio.h>
void add(int m, int n);
void main()
{
        int m,n;
        printf("enter values for m and n:");
        scanf("%d %d",&m,&n);
        add(m,n);
}
```

Called function

```
void add(int a, int b)
{
        int sum;
        sum = a+b;
        printf("sum is:%d",sum);
        return;
}
```

4. Function with parameters and return values

- In this category, there is data transfer between the calling function and called function.
- When Actual parameters values are passed, the formal parameters in called function can receive the values from the calling function.
- When the add function returns a value, the calling function receives a value from the called function.
- Sum is computed and returned back to calling function which is assigned to variable result.

Calling function

```
/*program to find sum of two numbers
using function*/
        #include<stdio.h>
        int add();
        void main()
        {
                int result,m,n;
                printf("enter values for m and
                n:");
                scanf("%d %d",&m,&n);
                result=add(m,n);
                printf("sum is:%d",result);
        }
```

Called function

```
int add(int a, int b) /* function header */
{
        int sum;
        sum= a+b;
        return sum;
}
```

Difference between Actual parameters & Formal Parameters:

Actual Parameters	Formal Parameters
Actual parameters are also called as **argument list.** Ex: add(m,n)	Formal parameters are also called as **dummy parameters.** **Ex:**int add(int a, int b)
The variables used in function call are called as actual parameters	The variables defined in function header are called formal parameters
Actual parameters are used in calling function when a function is called or invoked Ex: add(m,n) Here, m and n are called actual parameters	Formal parameters are used in the function header of a called function. Example: int add(int a, int b) { ……….. } Here, a and b are called formal parameters.
Actual parameters sends data to the formal parameters Example:	Formal parameters receive data from the actual parameters.

* Modifying parameters inside functions using pointers: (Call by reference) or

* Passing parameters to functions or Types of argument passing:

The different ways of passing parameters to the function are:

1. **Pass by value or Call by value**
2. **Pass by address or Call by address**

1. Call by value or (pass by value):

- In call by value, the values of actual parameters are copied into formal parameters.
- The formal parameters contain only a copy of the actual parameters.
- So, even if the values of the formal parameters changes in the called function, the values of the actual parameters are not changed.
- The concept of call by value can be explained by considering the following program.

Example:

```
#include<stdio.h>
 void swap(int a,int b);
void main()
{
int m,n;
printf("enter values for a and b:");
scanf("%d %d",&m,&n);
printf("the values before swapping are m=%d n=%d \n",m,n);
swap(m,n);
printf("the values after swapping are m=%d n=%d \n",m,n);
}
void swap(int a, int b)
{
int temp;
temp=a;
a=b;
b=temp;
}
```

Output:

```
enter values for a and b  10   20
the values before swapping are  10  20
the values after swapping are   10  20
```

Execution starts from function main( ) and we will read the values for variables m and n, assume we are reading 10 and 20 respectively.

- We will print the values before swapping it will print 10 and 20.
- The function swap( ) is called with actual parameters m=10 and n=20.
- In the function header of function swap( ), the formal parameters a and b receive the values 10 and 20.
- In the function swap( ), the values of a and b are exchanged.
  - But, the values of actual parameters m and n in function main( ) have not been exchanged.
- The change is not reflected back to calling function.

2. Call by Address or (pass by reference):

- In Call by **Address,** when a function is called, the addresses of actual parameters are sent.
- In the called function, the formal parameters should be declared as pointers with the same type as the actual parameters.
- The addresses of actual parameters are copied into formal parameters.
- Using these addresses the values of the actual parameters can be changed.
- This way of changing the actual parameters indirectly using the addresses of actual parameters is known as pass by address.

Example:

```
#include<stdio.h>
void swap(int a,int b);
void main()
{
int m,n;
printf("enter values for a and b:");
scanf("%d %d",&m,&n);
printf("the values before swapping are m=%d n=%d \n",m,n);
swap(&m,&n);
printf("the values after swapping are m=%d n=%d \n",m,n);
}


void swap(int*a, int*b)
{

int temp;
temp=*a;
*a=*b;
*b=temp;

}
```

```
Output: enter values for a and b   10   20
           the values before swapping are    10  20
            the values after swapping are   20  10
```

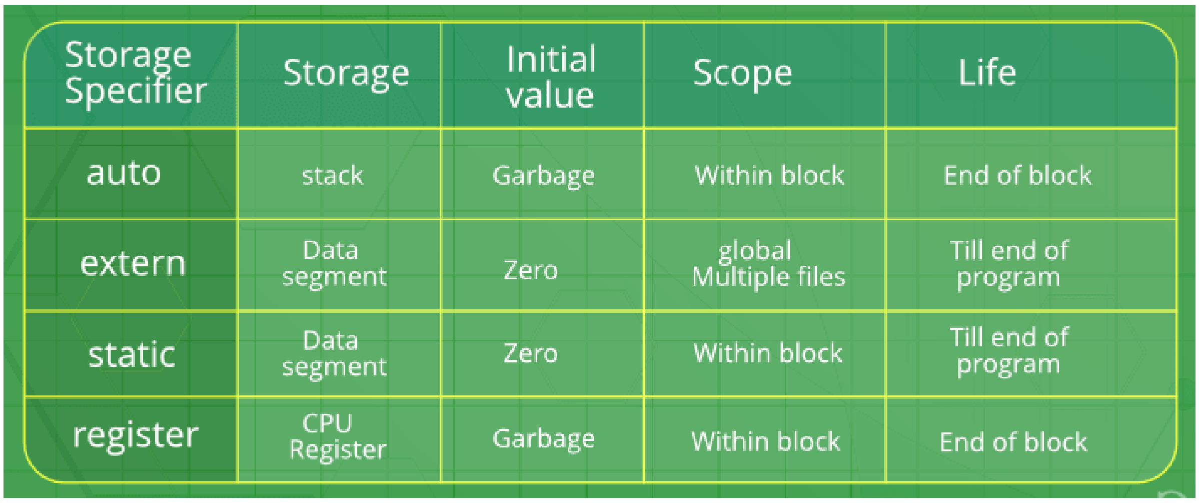
Differences between Call by Value and Call by reference

Call by Value	Call by Address
When a function is called the **values of variables are passed**	When a function is called the **addresses of variables are passed**
The type of formal parameters should be same as type of actual parameters	The type of formal parameters should be same as type of actual parameters, but they have to be declared as **pointers.**
Formal parameters contains the values of actual parameters	Formal parameters contain the addresses of actual parameters.

* Storage classes:

- C Storage Classes are used to describe the features of a variable/function.
- These **features** basically include the **scope, visibility, and lifetime** which help us to trace the existence of a particular variable during the runtime of a program.

C language uses 4 storage classes, namely:



Storage Specifier	Storage	Initial value	Scope	Life
auto	stack	Garbage	Within block	End of block
extern	Data segment	Zero	global Multiple files	Till end of program
static	Data segment	Zero	Within block	Till end of program
register	CPU Register	Garbage	Within block	End of block

1. auto :

- The **auto** storage class is the default storage class for all local variables.
- The variables defined using auto storage class are called as local variables. Auto stands for automatic storage class.
- A variable is in auto storage class by default if it is not explicitly specified.
- The scope of an auto variable is limited with the particular block only.
- Once the control goes out of the block, the access is destroyed.
- This means only the block in which the auto variable is declared can access it.

- A **keyword <u>auto</u>** is used to define an auto storage class.
- By default, an auto variable contains a garbage value.

```
Example: auto int age;
```

```
#include <stdio.h>
int main( )
{
 auto int j = 1;
 {
  auto int j= 2;
  {
   auto int j = 3;
   printf ( " %d ", j);
  }
  printf ( "\t %d ",j);
 }
 printf( "%d\n", j);
}
Output:3 2 1
```

2. <u>Extern :</u>

- Extern stands for external storage class.
- Extern storage class is used when we have global functions or variables which are shared between two or more files.
- Keyword **extern** is used to declaring a global variable or function in another file to provide the reference of variable or function which have been already defined in the original file.
- The variables defined using an extern keyword are called as global variables. These variables are accessible throughout the program.
- Notice that the extern variable cannot be initialized it has already been defined in the original file.

```
Example: extern void display();
```

<u>First File: main.c</u>

```
#include <stdio.h>
extern i;
main()
{
   printf("value of the external integer is = %d\n", i);
   return 0;
}
```

Second File: original.c

```
#include <stdio.h>
i=48;
```

Result:

```
 value of the external integer is = 48
```

3. Static :

- The static variables are used within function/ file as local static variables. They can also be used as a global variable.
- Static local variable is a local variable that retains and stores its value between function calls or block and remains visible only to the function or block in which it is defined.
- Static global variables are global variables visible **only to the file in which it is declared.** **Example: static int count = 10**;

Program:

```
void display();
int main()
{
   display();
   display();
}
void display()
{
   static int c = 1;
   c += 5;
   printf("%d  ",c);
}
Output: 6   11
```

4. Register :

- You can use the register storage class when you want to store local variables within functions or blocks in CPU registers instead of RAM to have quick access to these variables.

  Example: register int age;
- The keyword **register** is used to declare a register storage class.

- The variables declared using register storage class has lifespan throughout the program.

Arrays as parameters:

- Single array elements can also be passed as arguments. This can be done in exactly the same way as we pass variables to a function.
- If you want to pass a single-dimension **array as an argument in a function**, you would have to declare a formal parameter in one of following **three ways** and all three declaration methods

- Way-1. **Formal parameters as a pointer** − void myFunction(int *param)
```
{
  statements
}
```
- Way-2. **Formal parameters as a sized array** − void myFunction(int param[10])
```
{
        statements
}
```
- Way-3. **Formal parameters as an unsized array** − void myFunction(int param[])
```
{
  Statements
}
```

Syntax : ***return_type functionName*** `( array_type array_name[size], ...);`

Example:

Now, consider the following function, which takes an array as an argument along with another argument and based on the passed arguments, it returns the average of the numbers passed through the array as follows −

```
Int getAverage(int arr[], int size)
{
   int i;
   double avg;
   double sum = 0;

   for (i = 0; i < size; ++i)
   {
      sum += arr[i];
   }
   avg = sum / size;

   return avg;
}
```

Now, let us call the above function as follows –

```
#include <stdio.h>
int getAverage(int arr[], int size);

int main ()
{
   /* an int array with 5 elements */
   int balance[5] = {1000, 2, 3, 17, 50};
   double avg;

   /* pass pointer to the array as an argument */
   avg = getAverage( balance, 5 ) ;

   /* output the returned value */
   printf( "Average value is: %f ", avg );

   return 0;
}
```

Output : Average value is: 214.400000