C PROGRAMMING FOR PROBLEM SOLVING (18CPS23)

Module 4
User defined functions and Recursion

Prof. Prabhakara B. K.
Associate Professor
Department of Computer Science and Engineering
Vivekananda College of Engineering and Technology, Puttur.

C PROGRAMMING FOR PROBLEM SOLVING (18CPS23)

Module 4
User defined functions and Recursion

User defined functions and Recursion

Example programs, Finding factorial of a positive integer and Fibonacci series.

FUNCTIONS

Definition

Function is a independent program module (group of statements) to perform a specific task.

General categories functions

1) Built-in functions / Library functions /Pre-defined functions.

- These functions are grouped together and placed in a common location called library.
- Each function here performs a specific operations.
- The compiler will access these functions from the header files.
- User is not required to write it.

Examples printf( ), scanf( ), gets( ), fopen( ), sqrt ( ), pow( ), exit( ) etc.

2) User defined functions

- These are written by the user.
- Header files are not required.
- main( ) is a user defined function.

USER DEFINED FUNCTIONS

ADVANTAGES OF USER DEFINED FUNCTIONS

1. Modularity (Divide and Conquer technique)

A Complex program can be decomposed into small sub-programs or user defined functions. These decompositions are made based up on functionalities to give solutions.

2. Easy for Debugging (Removing errors)

During debugging it is very easy to locate and isolate faulty functions. It is also easy to maintain program that uses user defined functions.

3. Readability (Understand the logic)

A complex problem is divided in to different sub-programs with clear objective and interface which makes easy to understand the logic behind the program.

ADVANTAGES OF USER DEFINED FUNCTION

4. Reduced code (Reduced Program size)

The sequence of statements those are repeatedly used in a program can be combined together to form a user defined functions. And these user defined functions can be used as many times as required. This avoids writing of same code again and again reducing program size.

5. Reusability

Once user defined function is implemented, it can be used as many times as required which reduces code repeatability and increases code reusability.

6. Build library

User can create personal library (header files), which helps in reduce effort required in implementation.

7. Protects Data

By declaring variables inside user defined functions we can protect data.

ELEMENTS OF USER DEFINED FUNCTION

Three elements are possible in a user defined function. They are

1. Function Definition

- It contains the group of statements to do specific tasks.
- It is a mandatory element.

2. Function Call

- It is a statement used to invoke the function definition at required place in a program.
- It is a mandatory element.

3. Function Declaration (Function Prototype)

- Function is an example for an identifier
- While naming a function we must follow rules to form an identifier.
- It is a statement which is not mandatory when user defined function definition appears before function main( )
- It is a statement which is mandatory when user defined function definition appears after function main( )

FUNCTION DEFINITION

There are 6 elements possible in a function definition. They are

1. Return data type
2. Function name
3. List of formal parameters
4. Declaration of local variables
5. Executable statements
6. return statement

In the above function name and executable statements are mandatory.

Syntax

```
return_type  function_name (list of formal parameters)
{
    declaration of local variable;
    executable statements;
    return statement;
}
```

ELEMENTS OF FUNCTION DEFINITION

Syntax

```
return_type  function_name (list of formal parameters)
{
    declaration of local variable;
    executable statements;
    return statement;
}
```

Example

```
int add (int  a,  int  b)
{
    int sum;
    sum=a+b;
    return(sum);
}
```

ELEMENTS OF FUNCTION DEFINITION

1. Return data type

- It is data type of the result returned / sent back from function definition.
- If function definition is not returning any value the return data type is void.
- This element of function definition is not mandatory.
- If return data type is not specified the default return data type is int.

2. Function name

- It is a name given to the function.
- While naming functions, we must follow the rules to form an identifier. Because function is also an identifier.
- This element of function definition is mandatory.

Example

```
int add (int  a,  int  b)
{
        int sum;
        sum=a+b;
        return(sum);
}
```

3. List of Formal Parameter

- These parameters are inputs to do specified task.
- They are **variables** received from function call.
- The order, data type and number of parameters received from function call must match with list of formal parameters.
- Each parameter must be declared individually and separated by comma.

ELEMENTS OF FUNCTION DEFINITION

1. Return data type 2. Function name 3. List of Formal Parameter

Above three elements together known as function header.

Below three elements are know as function body.

Example

```
int add (int  a,  int  b)
{
        int sum;
        sum=a+b;
        return(sum);
}
```

4. Declaration of local variables

The variable declared inside a function and used only inside a function are local variables. They are not possible to access outside the function.

5. Executable statements

These statements are used to do specific task inside function definition.

6. return statement

This statement returns /sends the result back to the calling function.

return statement is a unconditional branching statement.

Syntax `return;` or `return(expression);`

Where expression may be constant / variable or an expression itself.

Examples return; return(5); return(a) ; return(a+b); return(b*6);

FUNCTION CALL

- It is a statement used to invoke the function definition at required place in a program.
- It is a mandatory element.
- There are three elements in a function call.

Syntax

```
Variable = function_name(list of actual parameters);
```

1. Variable

- This parameter is used to store the result returned from function definition.
- This element of the function call is not mandatory.

2. Function name

- It is a name given to the function.
- The function name specified here must be same as that of function definition.
- This element of function call is mandatory.

FUNCTION CALL

Syntax

```
Variable = function_name(list of actual parameters);
```

3. List of actual parameters

- These parameters are used to give inputs to the function definition.
- These parameters may be constants / variables / expressions and they are separated by comma.
- This element of the function call is not mandatory.
- If actual parameters are not present then we can keep empty parenthesis or we can write void inside.

Examples

```
sum = addition(5);
      addition(5);
sum = addition(5,6);
      addition(5,6);
```

```
sum = addition(a);
      addition(a);
sum = addition(a,b);
      addition(a,b);
```

```
sum = addition(1+2, 3-4);
      addition(1+2,3-4);
sum = addition(a+b, a-b);
      addition(a+b,a-b);
```

FUNCTION DEFINITION and FUNCTION CALL

```
/* Program to find product of given two integers */
#include<stdio.h>
int mul(int a, int b)
{
    return(a*b);
}

void main()
{
    int a, b, c;
    printf("Enter two integers\n");
    scanf("%d%d",&a,&b);
    c=mul(a,b);
    printf("The Product is = %d",c);
}
```

Called Function / Function definition

Calling Function

Function call

FUNCTION DECLARATION

(FUNCTION PROTOTYPE)

Function is an example for an identifier While naming a function we must follow rules to form an identifier.

- It is a statement which is not mandatory when user defined function definition appears before function main( )
- It is a statement which is mandatory when user defined function definition appears after function main( )

Syntax

```
data_type  function_name(list formal parameters);
```

- The simplest way of declaring a function is keeping the function header before calling function with semicolon.
- In good practice they are commonly written after pre-processor directives. i.e., after #include statements.
- In the parameter list, it is enough if we mention the data types of the parameters. (variables are not mandatory)

FUNCTION DECLARATION

(FUNCTION PROTOTYPE)

```
/* Program to find product of given two integers */
#include<stdio.h>
int mul(int a, int b);          // int mul(int, int);
void main()
{
    int a, b, c;
     printf("Enter two integers\n");
     scanf("%d%d",&a,&b);
     c=mul(a,b);
     printf("The Product is = %d",c);
}
int mul(int a, int b)
{
     return(a*b);
}
```

Function Declaration

Calling Function

Function Call

Function header

Called Function / Function definition

CATEGORIES OF USER DEFINED FUNCTIONS
BASE ON PARAMETERS AND RETURN VALUE

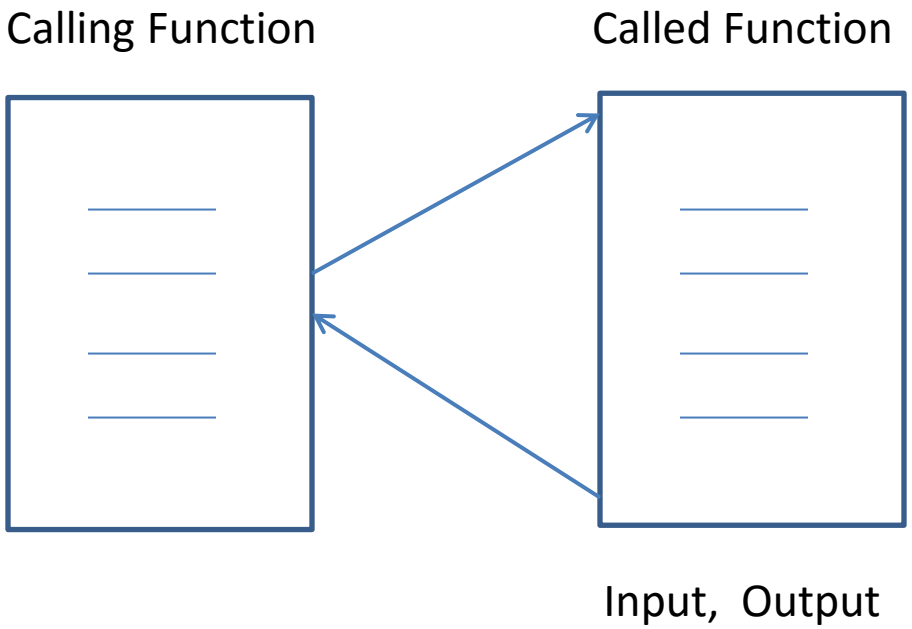
There are four categories of user defined function based on parameters and return value. They are

1. Function with no arguments and no return value
2. Function with arguments and no return value
3. Function with no arguments and return value
4. Function with arguments and return value

TYPES OF USER DEFINED FUNCTIONS
BASE ON PARAMETERS AND RETURN VALUE

1. Function with no arguments and no return value

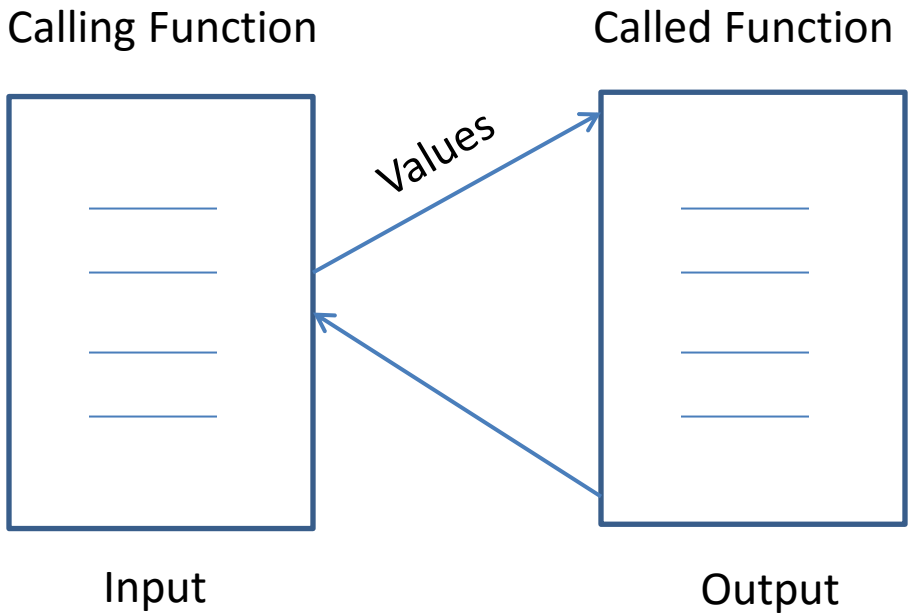
   (void function without parameters / No communication)



CATEGORIES OF USER DEFINED FUNCTIONS
BASE ON PARAMETERS AND RETURN VALUE

2. Function with arguments and no return value

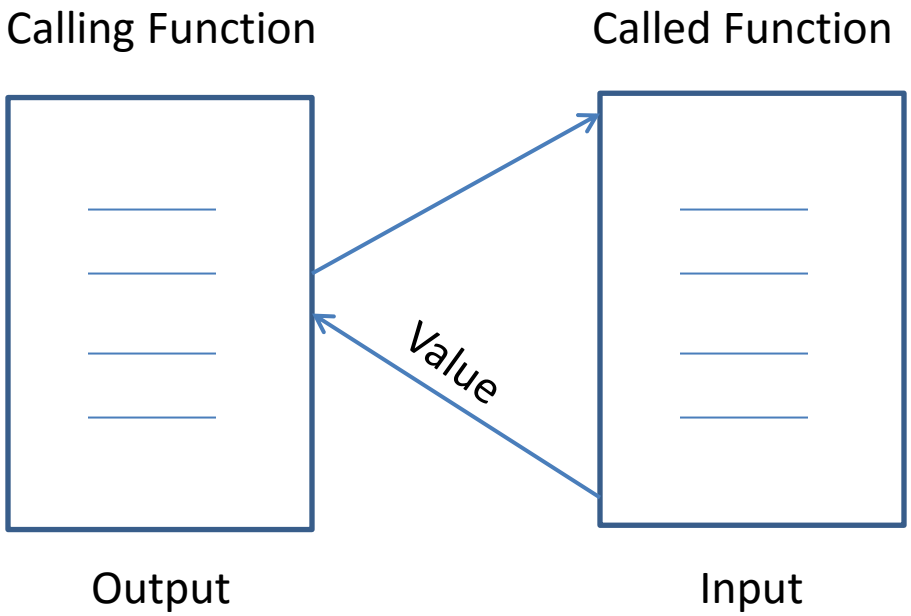
   (void function with parameters / Downward communication)



TYPES OF USER DEFINED FUNCTIONS
BASE ON PARAMETERS AND RETURN VALUE

3. Function with no arguments and with return value

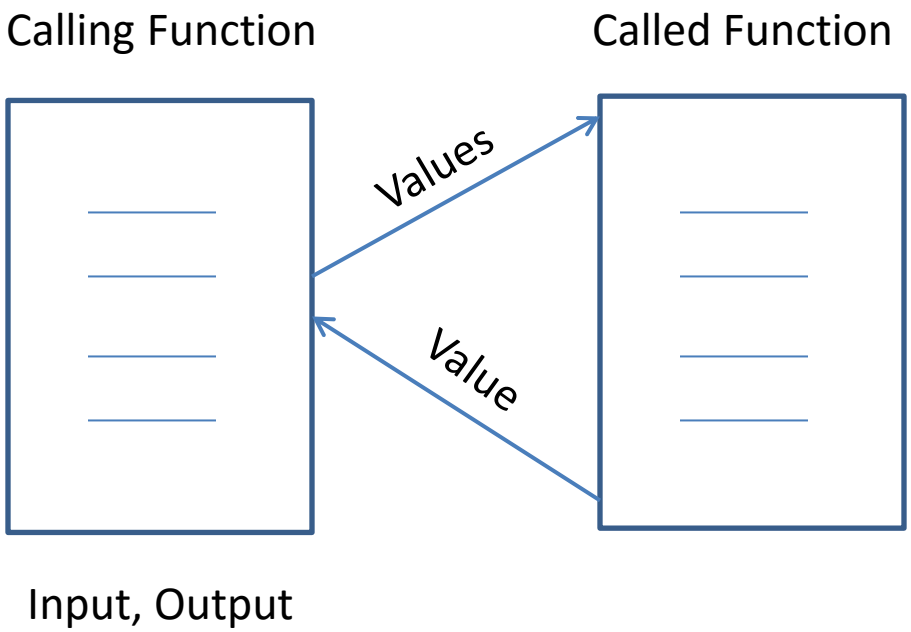
(non-void function without parameters / Upward communication)



TYPES OF USER DEFINED FUNCTIONS
BASE ON PARAMETERS AND RETURN VALUE

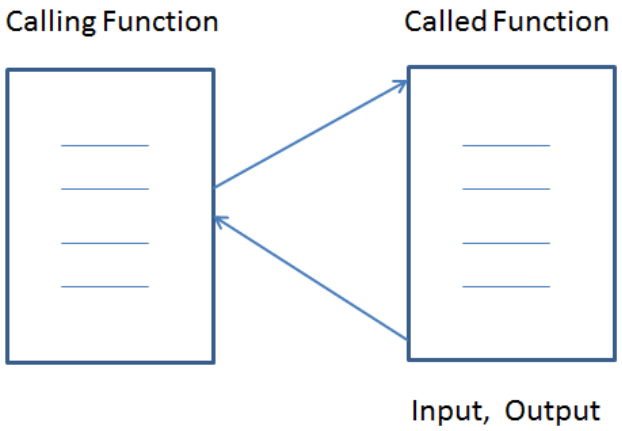
4. Function with arguments and with return value

   (non-void function with parameters / Up-down communication)



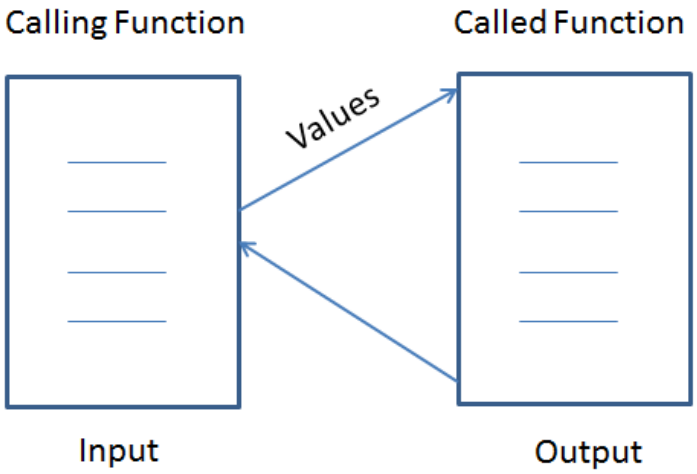
1. Function with no arguments and no return value (Void function without parameters / No communication)

```
#include<stdio.h>
void add();
void main()
{
      add( );
}
void add()
{
       int a, b, c;
       printf("Enter two integers\n");
       scanf("%d%d",&a,&b);
       c=a+b;
       printf("Sum is =%d",c);
}
```



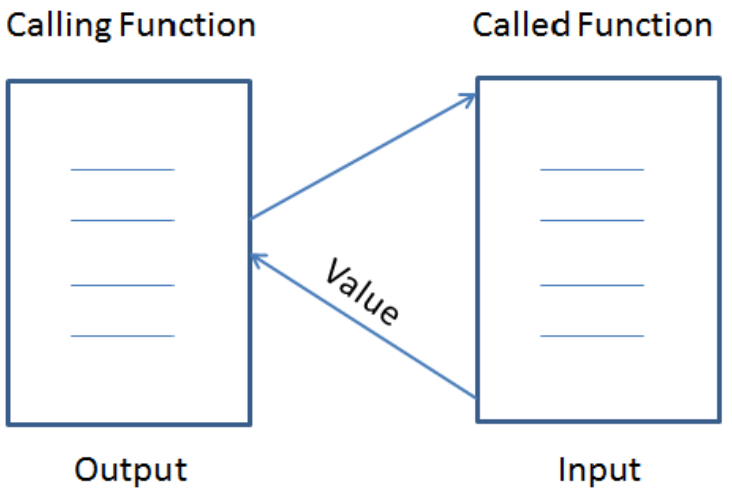
2. Function with arguments and no return value (Void function with parameters/Downward communication)

```
#include<stdio.h>
void add(int a, int b);
void main()
{
      int a,b;
      printf("Enter two integers\n");
      scanf("%d%d",&a,&b);
      add(a,b);
}
void add(int a, int b)
{
       int c;
       c=a+b;
       printf("Sum is =%d",c);
}
```



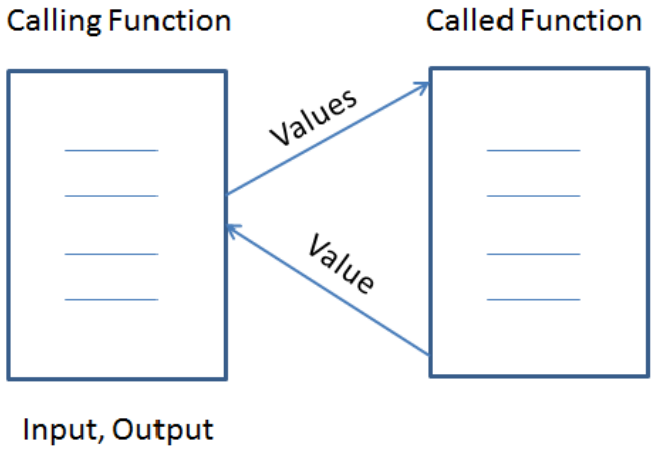
3. Function with no arguments and with return value
(Non-void function without parameters/Upward communication)

```
#include<stdio.h>
int add();
void main()
{
      int c;
      c=add();
      printf("Sum is =%d",c);
}
int add()
{
       int a,b,c;
       printf("Enter two integers\n");
       scanf("%d%d",&a,&b);
       c=a+b;
       return(c);
}
```



4. Function with arguments and with return value (Non-void function with parameters/Up-down communication)

```
#include<stdio.h>
int add(int a, int b);
void main()
{
      int a, b, c;
      printf("Enter two integers\n");
      scanf("%d%d",&a,&b);
      c=add(a, b);
      printf("Sum is =%d",c);
}
int add(int a, int b)
{
       int c;
       c=a+b;
       return(c);
}
```



TYPES OF USER DEFINED FUNCTIONS
BASE ON FUNCTION CALL

There are two types of user defined function based on function call /passing parameters.

1. **Call by value**

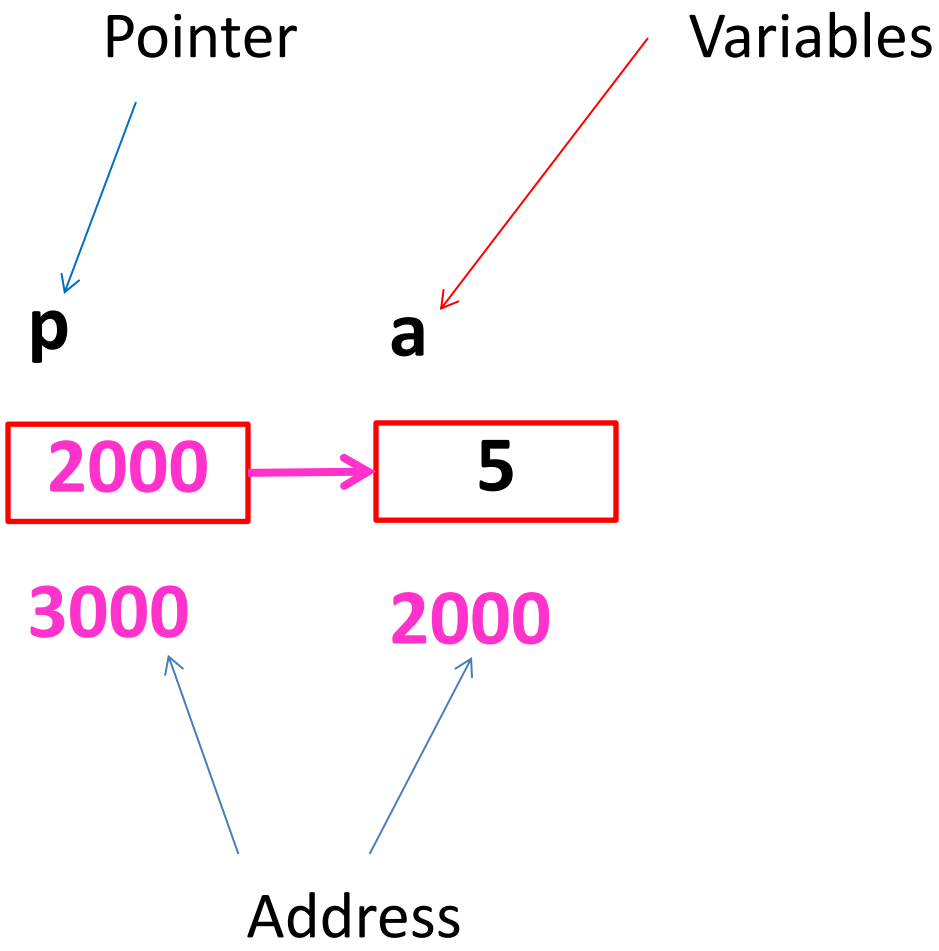
   This type of user defined functions are invoked by sending values of the parameters form function call to function definition.

2. **Call by reference**

   This type of user defined functions are invoked by sending address of the parameters form function call to function definition.

POINTER

Pointer is a variable which holds address of another variable.



POINTERS

When a pointer is declared, it points to the garbage value.

Such a pointer is called stray pointer. Hence it is required to be initialized.

Declaration & Initialization

```
datatype variable;
datatype *pointer;
pointer=&variable;
```

OR

```
datatype variable;
datatype *pointer=&variable;
```

Example

```
int a=10;
int *p;
p=&a;
```

```
int a=10;
int *p=&a;
```

CALL BY VALUE

```
/*Sum of 2 integers without Pointers */
#include<stdio.h>
int add(int a, int b);
void main()
{
      int a, b, c;
      printf("Enter two integers\n");
      scanf("%d%d",&a,&b);
      c=add(a, b);
      printf("Sum is =%d",c);
}
int add(int a, int b)
{
       int c;
       c=a+b;
       return(c);
}
```

CALL BY REFERENCE

```
/* Sum of 2 integers with Pointers */
#include<stdio.h>
int add(int *a, int *b);
void main()
{
      int a, b, c;
      printf("Enter two integers\n");
      scanf("%d%d",&a,&b);
      c=add(&a, &b);
      printf("Sum is =%d",c);
}
int add(int *a, int *b)
{
       int c;
       c=*a+*b;
       return(c);
}
```

CALL BY VALUE	CALL BY REFERENCE
1. When the function is called the values of the variables are passed.	1. When the function is called the address of the variables are passed.
2. Pointers are not required.	2. Pointers are required.
3. Slow in execution.	3. Fast in execution.
4. Changes in the formal parameters will not effect to the actual parameters.	4. Changes in the formal parameters indirectly changes the actual parameters.
5. Actual and formal parameters are created in different memory locations.	5. Actual and formal parameters are created in same memory locations.

6. Example
#include<stdio.h>
int add(int a, int b);
void main()
{
int a, b, c;
printf("Enter two integers\n");
scanf("%d%d",&a,&b);
c=add(a, b);
printf("Sum is =%d",c);
}
int add(int a, int b)
{
int c;
c=a+b;
return(c);
}
	6. Example
	#include<stdio.h>
	int add(int *a, int *b);
	void main()
	{
	int a, b, c;
	printf("Enter two integers\n");
	scanf("%d%d",&a,&b);
	c=add(&a, &b);
	printf("Sum is =%d",c);
	}
	int add(int *a, int *b)
	{
	int c;
	c=*a+*b;
	return(c);
	}

RECURSION

Recursion is a programming technique that allow the programmer to express an operation in terms of themselves.

Definition

Recursion is a user defined function which calls itself until terminating condition.

- Recursive function is a user defined function.
- Where Function Call lies inside a Function Definition.
- The Calling Function and Called Functions are same.
- The Stack data structure is used in recursion.
- When recursive function is called, the stack is used to hold a new set of local variables and parameters.
- When recursive call returns, the old local variables and parameters are removed from the stack.

RECURSION

Example

Function Definition

(Called Function/Calling function)

```
#include<stdio.h>
void main()
{
    printf("C Programming");
    main();
}
```

Function Call

Output

Prints "C Programming" infinite times.

The above program is not having terminating condition to stop the execution.

Write a C program to find factorial of given integer using recursive function.

```
#include<stdio.h>
int fact(int n)
{
    if(n==0)
        return(1);
    else
        return(n*fact(n-1));
}
void main()
 {
    int n;
    printf("Enter an integer\n");
    scanf("%d",&n);
    printf("Factorial of %d is %d\n",n,fact(n));
}
```

Terminating Condition

TRACE

n=4

fact(4)

4*fact(3)

4*3*fact(2)

4*3*2*fact(1)

4*3*2*1*fact(0)

4*3*2*1*1

24

Write a C program to print Fibonacci numbers of given limit using recursive function.

```
#include <stdio.h>
int fib(int i)
{
    if(i<=1)   // Terminating Condition
        return i;
    else
        return(fib(i-1)+fib(i-2));
}
void main()
{
    int i,n;
    printf("Enter a number\n");
    scanf("%d",&n);
    printf("Fibonacci numbers are\n");
    for(i=0;i<n;i++)
        printf("%d\t",fib(i));
}
```

TRACE n=4	
i=0,0<4	fib(0) 0
i=1,1<4	fib(1) 1
i=2,2<4	fib(2) fib(1) + fib(0) 1 + 0 1
i=3,3<4	fib(3) fib(2) + fib(1) fib(1) + 1 1 + 1 2
i=4,4<4	

STORAGE CLASSES

- Storage classes are used to describe the features of variables or functions.
- These features are Scope (Visibility) and Life-time.

Storage classes are qualifiers used to define the Scope (Visibility) and Life-time of variables or functions.

There are four storage classes in C

1. auto 2. static 3. extern 4. register

Qualifier	Storage	Initial Value	Scope (Visibility)	Life-time
auto	Stack	Garbage	Local / Inside function	End of block
extern	Data Segment	0	Global / Through out	End of program
static	Data Segment	0	Local / Inside function	End of program
register	CPU Registers	Garbage	Local / Inside function	End of block

AUTOMATIC VARIABLES

- These variables are declared and used inside the functions.
- By default automatic variables are initialized to garbage (Junk Value).
- Memory is allocated automatically when they are declared.
- Memory is deallocated automatically when the control of execution exit from the function where they are declared.
- **Since because the memory allocation and deallocation for these variables are done automatically they are called automatic variables.**
- The keyword used for defining automatic variables is auto
- By default all local variables are automatic variables.

Syntax

```
auto datatype variable;
```

Example auto int a;

Qualifier	Storage	Initial value	Scope (Visibility)	Life-time
auto	Stack	Garbage	Local	End of block
extern	Data Segment	0	Global	End of program
static	Data Segment	0	Local	End of program
register	CPU Registers	Garbage	Local	End of block

Programming example for AUTOMATIC VARIABLES

```
#include<stdio.h>
void f2()
{
    auto int a=10;
    printf("%d\t",a);
}
void f1()
{
    auto int a=100;
    f2();
    printf("%d\t",a);
}
```

```
void main()
{
    auto int a=1000;
    f1();
    printf("%d",a);
}
```

OUTPUT

10 100 1000

EXTRNAL VARIABLES

- They are also known as global variable.
- Default initial value of the external variable is 0 (If integer otherwise NULL).
- External variable cannot be initialized with in functions.
- Extern variables are always initialized globally (Only once).
- The variables declared as extern are not allocated any memory.
- **These declaration are used to specify that the variable is declared elsewhere in the program.**
- The keyword used to define external variable is extern

Syntax

```
extern datatype variable;
```

Example extern int a;

Qualifier	Storage	Initial value	Scope (Visibility)	Life-time
auto	Stack	Garbage	Local	End of block
extern	Data Segment	0	Global	End of program
static	Data Segment	0	Local	End of program
register	CPU Registers	Garbage	Local	End of block

Programming example for EXTERNAL VARIABLES

Below C program is saved in a filename **file2.c**

```
#include<stdio.h>
#include"file1.c"
int a=5;
void main()
{
   printf("%d\n",a);
   fun();
   printf("%d\n",a);
}
```

Below C program is saved in a filename **file1.c**

```
void fun()
{
      extern int a;
      a=a+3;
}
```

OUTPUT

5

8

STATIC VARIABLES

- Static variables tell compiler to persists (saves) the variable until the end of the program.
- Static variables are visible only to the function where they are declared.
- Default initial value of the static variable is 0 (If integer otherwise NULL).
- Static variables are initialized only once.
- The keyword used to define static variable is static

Two types of static variables

- Internal static variable - declared inside a function
- External static variable - declared outside a function

Syntax

```
static datatype variable=value;
```

Example static int x=10;

Qualifier	Storage	Initial value	Scope (Visibility)	Life-time
auto	Stack	Garbage	Local	End of block
extern	Data Segment	0	Global	End of program
static	Data Segment	0	Local	End of program
register	CPU Registers	Garbage	Local	End of block

Programming example for STATIC VARIABLES

```
#include<stdio.h>
void fun()
{
    static int x=10;
    x=x+1;
    printf("x=%d\n",x);
}
void main()
{
    fun();
    fun();
}
```

OUTPUT

x=11

x=12

REGISTER VARIABLES

- The register variables are stored in CPU registers.
- We can not use dereference operator (&) for the register variable.
- Because these variables are not having address.
- The access time of the register variables is faster than other variables.
- Default initial value of the register variable is garbage (Junk value).
- Frequently used variables (looping variables) will be kept in CPU registers.
- Most compiler allows int and char variables in CPU registers.
- The keyword used to define register variable is register

Syntax

```
register datatype variable;
```

Example `register int a;`

Qualifier	Storage	Initial value	Scope (Visibility)	Life-time
auto	Stack	Garbage	Local	End of block
extern	Data Segment	0	Global	End of program
static	Data Segment	0	Local	End of program
register	CPU Registers	Garbage	Local	End of block

Programming code for REGISTER VARIABLES

Example

```
int i, n, sum=0;
//register int i, n, sum=0;
……..
……..
……..

for(i=0;i<n;i++)
{
  sum=sum+i;
}
printf(“Sum=%d”,sum);
```

In this example

The declaration is

int i, n, sum=0;

If we modify above declaration like below

register int i, n, sum=0;

The loop executes faster.

Because the looping variables i and sum are frequently changing variables inside the loop. By storing i and sum variables in CPU registers, it is possible to execute the loop faster than earlier.

LOCAL VARIABLE	GLOBAL VARIABLE
1. They are declared inside a functions.	1. They are declared outside a functions.
2. These variables are created when control of execution enter into the function where they are declared and destroyed when that function is terminated.	2. These are created when a program execution starts and destroyed when program execution is terminated.
3. They are accessible inside the function where they are declared.	3. They are accessible throughout a program in which they are declared.
4. Stack memory is allocated.	4. Stack memory is not allocated.
5. Initialization is mandatory.	5. Initialization is not mandatory.
6. Parameter passing is required.	6. Parameter passing is not necessary.

REVIEW QUESTIONS

1. Explain elements of function with an example for each.
2. What is a function. Explain types of function based on parameters/arguments.
3. Write a program to find cube of a number using function.
4. Write a C program to find sum of two numbers using function.
5. Explain two types of function calls with a program example for each.
6. Write a note on benefits of functions.
7. What is recursion? Write a C program to calculate factorial of a number using recursion.
8. Write a C program to print Fibonacci series using recursion.
9. Compare local variables and global variables.
10. Compare actual parameters with formal parameters.