



# FILE HANDLING

# FILE HANDLING

- **PROBLEM:-**if we need same data to be processed again and again, we have to enter it again and again and if volume of data is large, it will be time consuming process.
- **SOLUTION:-**data can be permanently stored, a program can read it directly at a high speed.
- The permanent storage space on a disk is called FILE.
- **FILE** is a set of records that can be accessed through the set of library functions.

# FILE TYPES

**1. SEQUENTIAL FILE:-** in this data is kept permanently. If we want to read the last record of file then we need to read all the records before that record. it means if we want to read 10<sup>th</sup> record then the first 9 records should be read sequentially for reaching to the 10<sup>th</sup> record.

**2. RANDOM ACCESS FILE:-** in this data can be read and modified randomly. If we want to read the last record of the file, we can read it directly. It takes less time as compared as sequential file.



# Text Files vs. Binary Files

- Text files are human readable files. They are universal and can be edited with many different programs such as NOTEPAD.
  - **contains ASCII codes only**
- Binary files are not human readable. They contain information encoded into the numbers which make up the file which are ultimately turned into ones and zeros, thus the name “binary.”

# HOW TO DEFINE A FILE

Before using a file in a program, we must open it, which establishes a link between the program and the operating system. A file must be defined as:-

`FILE *fp;`

structure

file pointer

(user data type)

where FILE is the data structure which is included in the header file, `stdio.h` and `fp` is declared as a pointer which contains the address of a character which is to be read.



# Operation on a file

```
fp = fopen ("ABC.txt", "r");
```

Here      fp =file pointer

fopen= function to open a file

ABC.txt=name of the file

r=mode in which file is to be opened

fopen is a function which contains two arguments:-

1. File name ABC.txt
2. Mode in which we want to open the file.
3. Both these are of string types.

**fopen():-**

- **This function** loads the file from disk to memory and stores the address of the first character to be read in the pointer variable fp.
- If file is absent, it returns a NULL pointer variable.

**fclose(fp):-** the file which is opened need to be closed and only can be done by library function fclose.

# File opening modes

Mode	Description
r	Opens an existing text file for reading purpose.
w	Opens a text file for writing, if it does not exist then a new file is created. Here your program will start writing content from the beginning of the file.
a	Opens a text file for writing in appending mode, if it does not exist then a new file is created. Here your program will start appending content in the existing file content.
r+	Opens a text file for reading and writing both.
w+	Opens a text file for reading and writing both. It first truncate the file to zero length if it exists otherwise create the file if it does not exist.
a+	Opens a text file for reading and writing both. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended.

- If you are going to handle binary files then you will use below mentioned access modes instead of the above mentioned:
- "rb", "wb", "ab", "ab+", "a+b", "wb+", "w+b"



```
void main()
{
FILE *fp;
fp=fopen("data.txt","r");
If(fp == NULL)
{
printf("cannot open file");
}
getch();
}
```

# Closing a file

- To reopen it in some other mode
- To prevent any misuse with the file
- To make the operations you have performed on the files successful.

To close a file, we must use function `fclose()`

**Syntax:**

```
int fclose(FILE *stream);
```

# fclose()

- It closes the named stream. We can pass pointer variable that points to a file to be closed. All buffers associated with the stream are flushed before closing.
- On success, fclose() returns 0 otherwise, it returns end of function (EOF).

```
#include<stdio.h>
#include<conio.h>
void main()
{
    FILE *fptr;
    clrscr();
    fptr=fopen("a.txt","w");
    if(fptr==null)
        printf("\n file cannot be opened for creation");
    else
        printf("\n file created successfully");
    fclose(fptr);
}
```

## Binary modes:-

- **wb(write):-** it opens a binary file in write mode.

```
fp=fopen("data.dat","wb");
```

Here data.dat file is opened in binary mode for writing.

- **rb(read):-** it opens a binary file in read mode.

```
fp=fopen("data.dat","rb");
```

Here data.dat file is opened in binary mode for reading.

# Reading from a file

- © Different functions to read a char are `fgetc` and `getc`. Both perform similar functions and have the same syntax.

`ch = fgetc (fp);`

char type      function      file pointer  
variable      name

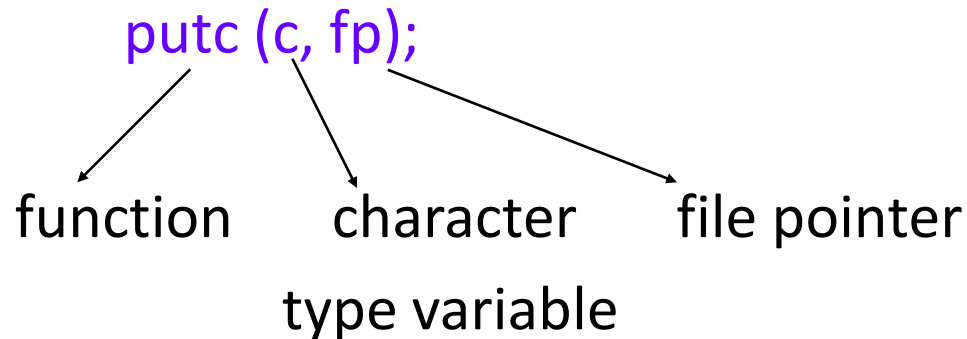
- © `fgets()` or `gets()`:- used to read strings from a file and copy string to a memory location which is referenced by array.

`fgets(str, n, fptr);`

array of chars      max length      file pointer

# Writing chars to a file

- `fputc()` or `putc()`:- both these function write the character in the file at the position specified by pointer.



The file pointer automatically moves 1 position forward after printing characters in the file.

# Read & write char into file and prints them

```
#include<conio.h>
void main()
{
    FILE *fptr;
    char c;
    clrscr();
    fptr=fopen("a.txt","r");
    printf("the line of text is");
    while((c=getc(fptr))!=EOF)
    {
        printf("%c",c);
    }
    fclose(fptr);
    getch();
}
```

```
#include<conio.h>
#include<stdio.h>
void main()
{
    FILE *fptr;
    char c;
    clrscr();
    fptr=fopen("a.txt","w");
    printf("Enter the line of text,press ^z to stop ");
    while((c=getchar())!=EOF)
    {
        putc(c,fptr);
    }
    fclose(fptr);
}
```



# Program to understand use of fgetc()

```
main( )  
{  
FILE *fptr;  
char ch;  
fptr= fopen("rec.dat","r");  
if(fptr == NULL)  
printf("file doesn't exist");  
else  
{  
while( ch = fgetc(fptr) !=EOF)  
printf("%c",ch);  
}  
fclose(fptr);  
}
```

## EOF:-END OF FILE

EOF is an integer value sent to prog.by OS. it's value is predefined to be -1 in STDIO.H, No char with the same value is stored on disk. while creating file, OS detects that last character to file has been sent, it transmits EOF signal.

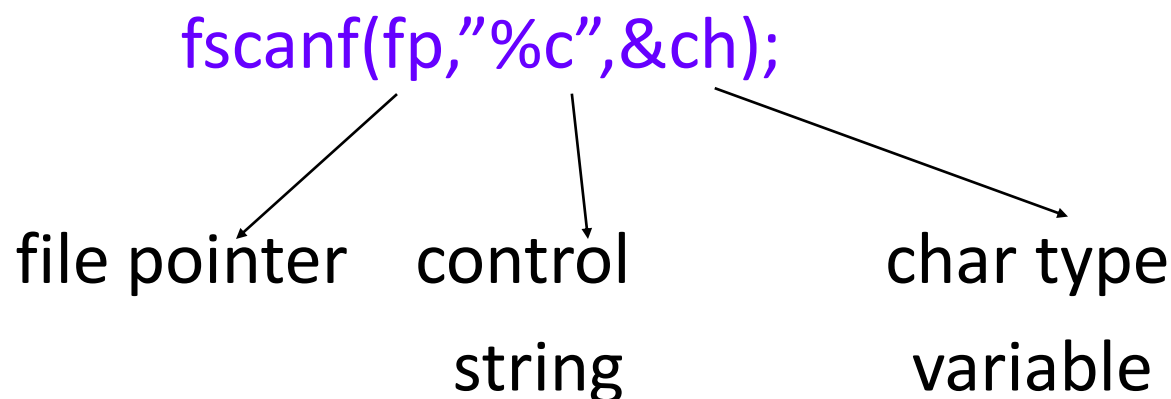
# Writing to file using fputs (sptr, fptr)

```
main()
{
FILE *fp;
char name[20],arr[50];
printf("enter the file name");
scanf("%s",name);
fp=fopen(name,"w");
if(fp == NULL)
{
printf("file can't be opened");
exit(0);
}
```

```
else
{
printf("the string is ");
gets(arr);
fputs(arr,fp);
}
fclose(fp);
}
```

- **fscanf( )**:- it is used to read to any type of variable  
i.e.,char,int,float,string

**fscanf(fp,"%c",&ch);**



file pointer      control  
                                 string

char type  
variable

**fscanf(fp,"%c%d%f",&a,&b,&c);**

We write **stdin**, this func call will work similar to simple scanf  
Because stdin means standard i/p i.e.keyboard.

**fscanf(stdin,"%c",&ch);**

Is equivalent to

**scanf("%c",&ch);**



```
#include<conio.h>
void main()
{
FILE *fptr;
char name[10];
int sal;
fptr=fopen("rec.dat","r");
fscanf(fptr,"%s%d",name,&sal);
while(!feof(fptr))
{
printf("%s%d",name,sal);
fscanf(fptr,"%s%d",name,&sal);
}
    fclose(fptr);
    getch();
}
```

- **fprintf()**:- it is used to print chars,numbers or strings in a file.

```
fprintf (fp,"%d",a);
```

With this function,we can print a no. of variables with single call.

```
fprintf (fp,"%c %d %d",a,b,c);
```

We write **stdout**,this func call will work similar simple printf

Because stdout means standard o/p i.e.keyboard.

```
fprintf(stdout,"%c",ch);
```

Is equivalent to

```
printf("%c",ch);
```

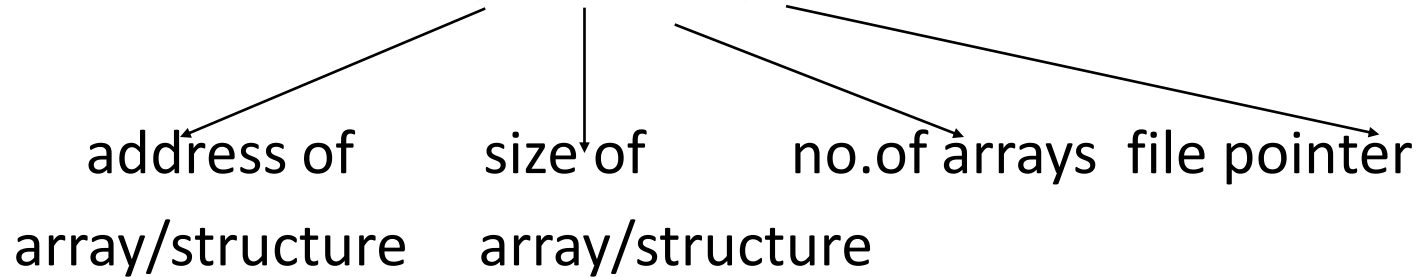


# Program for fprintf()

```
main()
{
FILE *fp;
char name[10];
fp=fopen("rec.dat","w");
printf("enter your name");
scanf("%s",name);
fprintf(fp,"%s",name);
fclose(fp);
}
```

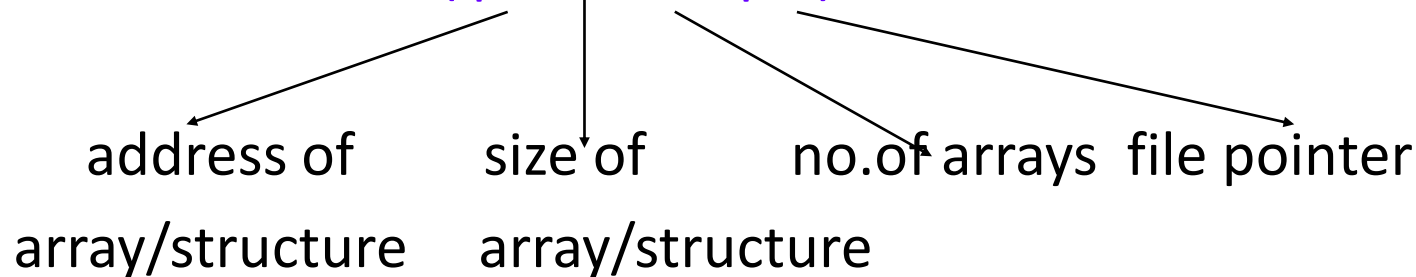
- **fread()**:-reads a block(array or structure)from a file.

**fread( ptr, m, n ,fptr);**



- **fwrite()**:-writes a block(array or structure)from a file.

**fread( ptr, m, n ,fptr);**





# Prog for fread()

```
#include<conio.h>
struct student
{
    int rollno;
    char name[20];
};
void main()
{
    struct student st;
    file *fptr;
    fptr=fopen("d.txt","r");
    clrscr();
    fread(&st,sizeof(st),1,fptr);
    printf("roll number is %d",st.rollno);
    printf("name is %s",st.name);
    fclose(fptr);
    getch();
}
```





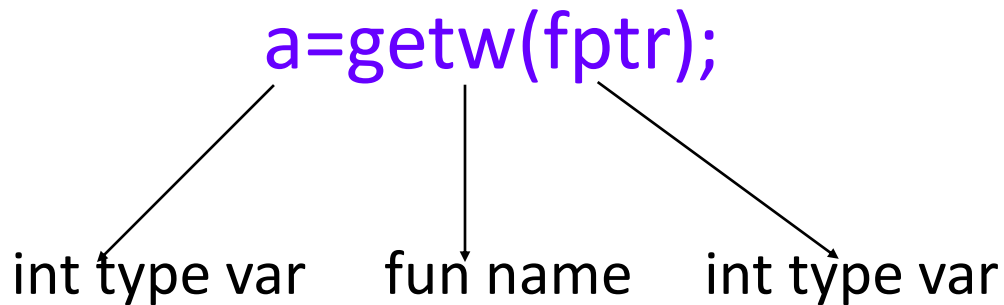
# Prog for fwrite()

```
#include<conio.h>
struct student
{
    int rollno;
    char name[20];
};
void main()
{
    struct student st;
    file *fptr;
    fptr=fopen("d.txt","w");
    clrscr();
    printf("enter roll number");
    scanf("%d",&st.number);
    printf("enter name");
    gets(st.name);
    fwrite(&st,sizeof(st),1,fptr);
    fclose(fptr);
    getch();
}
```

- **getw()**:-it is an integer oriented function.it is used to read an integer from file in which numbers are stored.

**a=getw(fptr);**

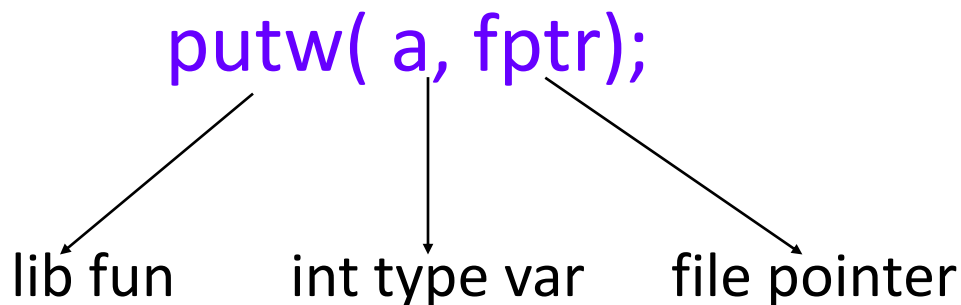
int type var      fun name      int type var



- **putw()**:-it prints a number in the file.

**putw( a, fptr);**

lib fun      int type var      file pointer





# Prog for getw()

```
#include<conio.h>
void main()
{
    file *fptr;
    int i,n;
    clrscr();
    fptr=fopen("b.txt","r");
    printf("\n the number are")
    for(i=1;i<=10;i++)
    {
        n=getw(fptr);
        printf("%d\t",n);
    }
    fclose(fptr);
    getch();
}
```



# Prog for putw()

```
#include<conio.h>
void main()
{
    file *fptr;
    int i,n;
    clrscr();
    fptr=fopen("b.txt","w");
    for(i=1;i<=10;i++)
    {
        printf("enter number");
        scanf("%d",&n);
        putw(n,fptr);
    }
    fclose(fptr);
    getch();
}
```

- **fseek():**-it is used for setting the file position pointer at the specified byte.

`fseek( FILE *fp,long offset,int origin);`

file pointer

long int can  
be +ve/-ve

no.of bytes skipped  
backward(-ve)  
forward(+ve)

Constant	Value	position
SEEK_SET	0	Beginning of file
SEEK_CUR	1	Current position
SEEK_END	2	End of file

## EXAMPLE

`fseek(p,10L,0);`

origin is 0, means that displacement will be relative to beginning of file, so position pointer is skipped 10 bytes forward from beginning of file. 2<sup>nd</sup> argument is long integer so L is attached with it.

`fseek(p,8L,SEEK_SET);`

position pointer is skipped 8 bytes forward from beginning of file.

`fseek(p,-6L,SEEK_END);`

position pointer is skipped 6 bytes backward from the end of file.



# Replace a char x by char y

```
main()
{
FILE *fp;
char temp,name[];
printf("enter name of file");
scanf("%s",name);
fp=fopen("name","r");
while((temp=getc(fp))!=EOF)
{
if(temp== 'x')
{
fseek(fp, -1,1);
putc('y',fp);
}
}
fclose(fp);
}
```

**ftell():**-it is required to find the current location of the file pointer within the file

**ftell(fptr);**

Fptr is pointer for currently opened file.

```
main()
{
FILE *fp;
char ch;
fp=fopen("text","r");
fseek(fp,241,0);
ch=fgetc(fp);
while(!feof (fp))
{
printf("%c",ch);
printf("%d",ftell(fp));
ch=fgetc(fp);
}
fclose(fp);
}
```



● **Rewind():** - it is used to move the file pointer to the beginning of the given file.

```
rewind(fp);
```

```
main()
{
FILE *fp;
fp=fopen("stu.dat","r");
if(fp==NULL)
{
printf("file can not open");
exit(0);
}
printf("position pointer %d",ftell(fp));
fseek(fp,0,2);
printf("position pointer %d",ftell(fp));
rewind(fp);
fclose(fp);
}
```

- **error()**:-it is used for detecting error in the file when file read write operation is carried out. It returns the value nonzero if an error occurs otherwise returns zero value.

**error(fp);**

WAP that writes the contents entered by user at runtime to a file, then read the contents of file using function fgetc() and fputc()

```
void main()
{
    FILE *fp;
    int c;
    clrscr();
    fp=fopen("ABC.doc","w");
    if(fp==NULL)
    {
        printf("file not opened in write mode");
        getch();
        exit(0);
    }
    printf("file opened successfully, enter ctrl +z to exit\n");
    while(c=getchar()!=EOF)
    {
```

```
fputc(c,fp);
}
fclose(fp);
fp=open("ABC.doc","r");
if(fp==NULL)
{
printf("file not opened in read mode");
getch();
exit(0);
}
printf("file opened for reading");
c=fgetc(fp);
while(c!=EOF)
{
printf("%c",c);
c=fgetc(fp);
}
fclose(fp);
getch();
}
```

# CSE101-lec#12

- Recursive function(or Recursion)
- Mathematical functions in C(<math.h> header file)

# Outline

- Recursion
- Examples of recursion
  - Finding factorial of a number
  - Finding sum of all numbers from 1 to n
  - Printing n terms of Fibonacci series using recursion
- Recursion Vs Iteration
- Mathematical functions in C

# Recursion

- It is a process in which function calls itself again and again and the function which is called is known as recursive function
- Recursion is supported with two cases: **base case and recursive case**
- **Base case** is one which helps to stop the recursion once the required operation is done
- **Recursive case** is one which keeps on repeating and function will keep on calling itself

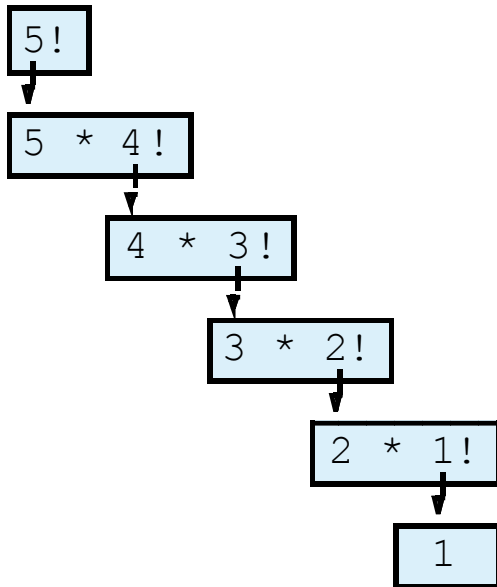


# Example

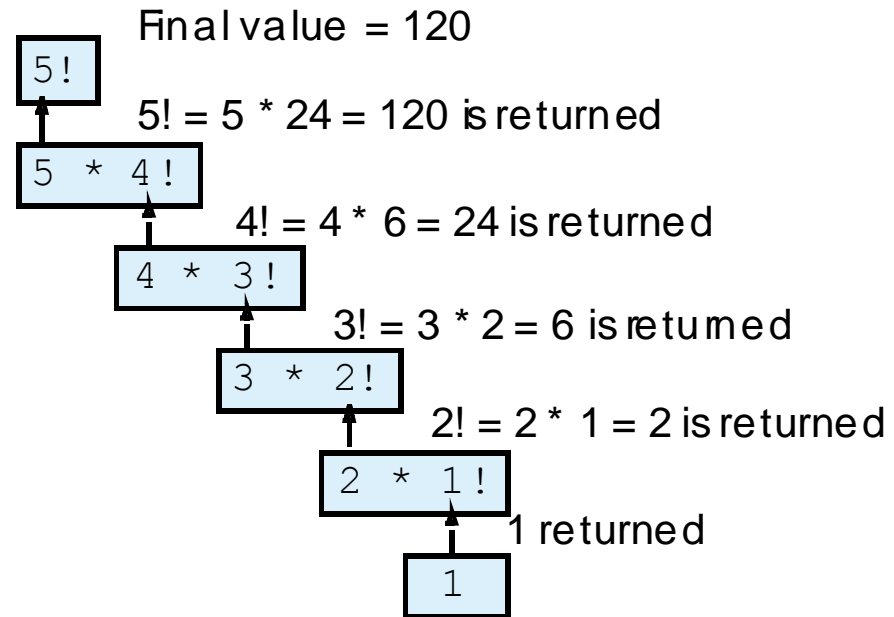
```
int fact(int n)
{
    if (n <= 1) // base case( Here the recursion will stop)
        return 1;
    else
        return n*fact(n-1); //recursive case where fact() is calling itself
}
```



# Recursion example (factorial)



(a) Sequence of recursive calls



(b) Values returned from each recursive call.



## Program example 1-Write a program to find the factorial of a number using recursion(or recursive function)

```
#include <stdio.h>
long long int factorial(int);
int main()
{
    int n;
    printf("Enter a positive integer: ");
    scanf("%d", &n);
    printf("Factorial of %d = %lld", n, factorial(n));
    return 0;
}
long long int factorial(int n)
{
    if (n <=1)
        return 1;
    else
        return n*factorial(n-1);
}
```



## Program example 2-Write a program to find the sum of all numbers from 1 to n using recursion(or recursive function)

```
#include <stdio.h>
int addNumbers(int n);
int main()
{
    int num;
    printf("Enter a positive integer: ");
    scanf("%d", &num);
    printf("Sum = %d",addNumbers(num));
    return 0;
}
int addNumbers(int n)
{
    if(n == 0)
        return n;
    else
        return n + addNumbers(n-1);
}
```

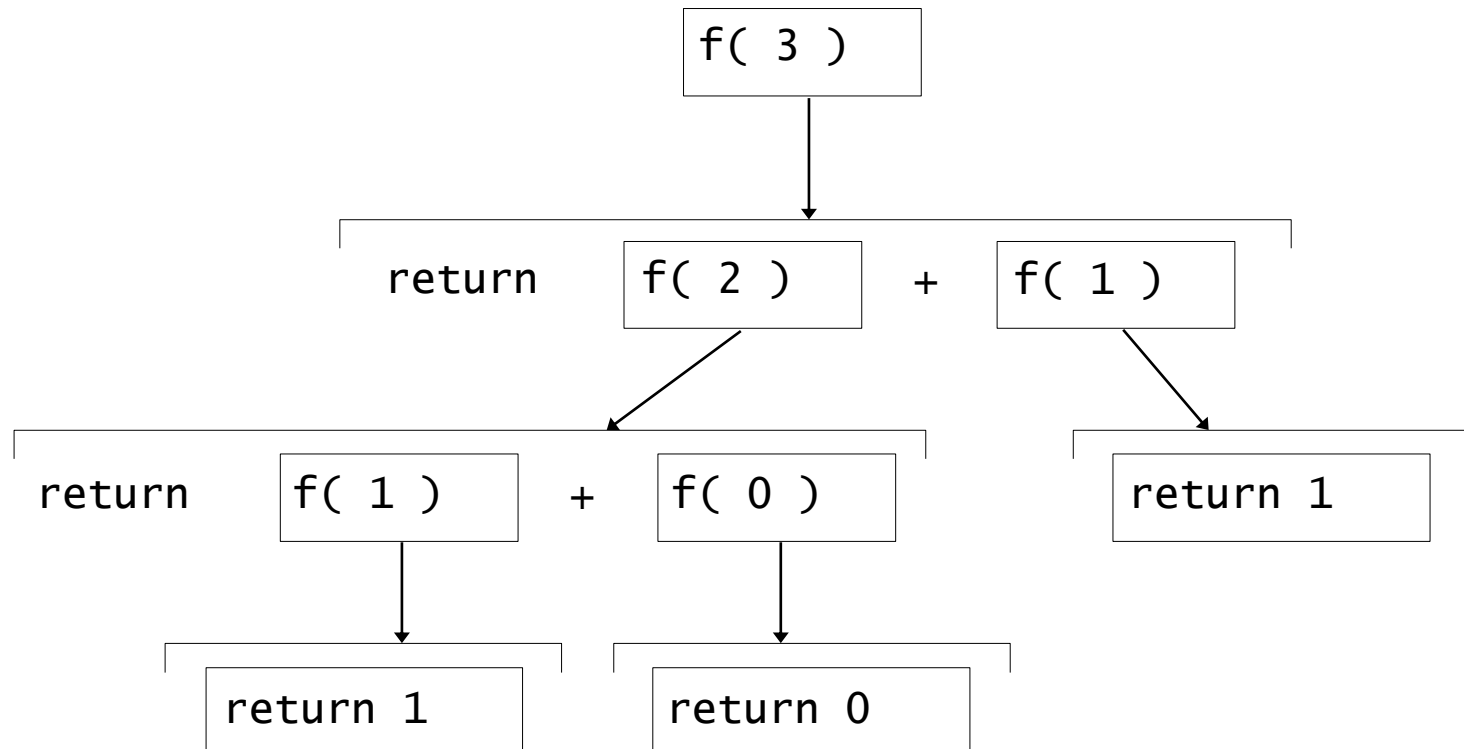
# Recursion example-3 (Fibonacci series)

- What is Fibonacci series: ...??
- 0, 1, 1, 2, 3, 5, 8...
  - Each number is the sum of the previous two
  - Can be solved recursively:
    - $\text{fib}(n) = \text{fib}(n - 1) + \text{fib}(n - 2)$
  - Code for the fibonacci function

```
long fibonacci( long n )
{
    if (n == 0 || n == 1)  // base case
        return n;
    else
        return fibonacci( n - 1)+fibonacci( n - 2 );
}
```

# Recursion example (fibonacci)

- Set of recursive calls to fibonacci() function



## Program example 3-WAP to display n terms of Fibonacci series using recursion(or recursive function)



```
#include<stdio.h>
int Fibonacci(int);
int main()
{
    int n, i;
    printf("\n Enter number of terms you want to print:");
    scanf("%d",&n);
    printf("Fibonacci series\n");
    for(i=0;i<n;i++)
    {
        printf("%d\n", Fibonacci(i));
    }
    return 0;
}
int Fibonacci(int n)
{
    if ( n == 0 )
        return 0;
    else if ( n == 1 )
        return 1;
    else
        return ( Fibonacci(n-1) + Fibonacci(n-2) );
}
```

# Recursion vs. Iteration

PROPERTY	RECURSION	ITERATION
<b>Definition</b>	Function calls itself.	A set of instructions repeatedly executed.
<b>Application</b>	For functions.	For loops.
<b>Termination</b>	Through base case, where there will be no function call.	When the termination condition for the iterator ceases to be satisfied.
<b>Usage</b>	Used when code size needs to be small, and time complexity is not an issue.	Used when time complexity needs to be balanced against an expanded code size.
<b>Code Size</b>	Smaller code size	Larger Code Size.
<b>Time Complexity</b>	Very high(generally exponential) time complexity.	Relatively lower time complexity(generally polynomial-logarithmic).



# Rules for recursive function

1. In recursion, it is essential to call a function itself
2. Only the user defined function can be involved in the recursion. Library function cannot be involved in recursion because their source code cannot be viewed
3. A recursive function can be invoked by itself or by other function.
4. To stop recursive function, it is necessary to have a proper base case, otherwise infinite recursion can happen
5. `main()` function can be invoked recursively.





# Math Library Functions(or Mathematical functions in C)

- Math library functions
  - perform common mathematical calculations
  - `#include <math.h>`
- Format for calling maths functions
  - `functionName( argument );`
    - If multiple arguments, use comma-separated list
- Example:  

```
printf( "%.2f", sqrt( 900.0 ) );
```

  - Calls function `sqrt`, which returns the square root of its argument
  - All math functions return data type **double**
  - Arguments may be constants, variables, or expressions

# Math Library Functions

Function	Description	Example
<code>sqrt( x )</code>	square root of $x$	<code>sqrt( 900.0 )</code> is 30.0 <code>sqrt( 9.0 )</code> is 3.0
<code>exp( x )</code>	exponential function $e^x$	<code>exp( 1.0 )</code> is 2.718282 <code>exp( 2.0 )</code> is 7.389056
<code>log( x )</code>	natural logarithm of $x$ (base $e$ )	<code>log( 2.718282 )</code> is 1.0 <code>log( 7.389056 )</code> is 2.0
<code>log10( x )</code>	logarithm of $x$ (base 10)	<code>log10( 1.0 )</code> is 0.0 <code>log10( 10.0 )</code> is 1.0 <code>log10( 100.0 )</code> is 2.0
<code>fabs( x )</code>	absolute value of $x$	<code>fabs( 5.0 )</code> is 5.0 <code>fabs( 0.0 )</code> is 0.0 <code>fabs( -5.0 )</code> is 5.0
<code>ceil( x )</code>	rounds $x$ to the smallest integer not less than $x$	<code>ceil( 9.2 )</code> is 10.0 <code>ceil( -9.8 )</code> is -9.0
<code>floor( x )</code>	rounds $x$ to the largest integer not greater than $x$	<code>floor( 9.2 )</code> is 9.0 <code>floor( -9.8 )</code> is -10.0
<code>pow( x, y )</code>	$x$ raised to power $y$ ( $x^y$ )	<code>pow( 2, 7 )</code> is 128.0 <code>pow( 9, .5 )</code> is 3.0
<code>fmod( x, y )</code>	remainder of $x/y$ as a floating point number	<code>fmod( 13.657, 2.333 )</code> is 1.992
<code>sin( x )</code>	trigonometric sine of $x$ ( $x$ in radians)	<code>sin( 0.0 )</code> is 0.0
<code>cos( x )</code>	trigonometric cosine of $x$ ( $x$ in radians)	<code>cos( 0.0 )</code> is 1.0
<code>tan( x )</code>	trigonometric tangent of $x$ ( $x$ in radians)	<code>tan( 0.0 )</code> is 0.0



# Math Library Functions: `pow()`

- The **power** function, `pow(x, y)`, calculates  $x^y$ ; that is, the value of

$$\text{pow}(x, y) = x^y.$$

- `pow(2, 3) = 23 = 8.0` and `pow(2.5, 3) = 15.625`.
- The function `pow` is of the type **double** or that the function `pow` returns a value of the type **double**.
- `x` and `y` are called the **parameters** (or **arguments**) of the function `pow`.
- Function `pow` has two parameters.

# sqrt()

- The **square root** function, `sqrt(x)`, calculates the non-negative square root of `x` for `x >= 0.0`  
`sqrt(2.25)` is 1.5  
`sqrt(25)` is 5.0
- The function `sqrt` is of the type **double** and has only one parameter.

# `fabs()`

- `fabs` calculates the absolute value of a float argument.

`fabs (2.25)` is 2.25

`fabs (-25.0)` is 25.0

- The function `fabs` is of the type `double` and has only one parameter.

# floor()

- The ***floor*** function, `floor`, calculates the largest whole number that is not greater than  $x$ .

`floor(48.79)` is `48.0`

`floor(48.03)` is `48.0`

`floor(47.79)` is `47.0`

- The function `floor` is of the type `double` and has only one parameter.

# ceil()

- The ***ceil*** function, `ceil`, calculates the smallest whole number that is not less than  $x$ .

`ceil(48.79)` is 49

`ceil(48.03)` is 49

`ceil(47.79)` is 48

- The function `ceil` is of the type `double` and has only one parameter.

Function	Standard Header File	Purpose	Parameter(s) Type	Result
<code>ceil(x)</code>	<code>&lt;cmath&gt;</code>	Returns the smallest whole number that is not less than x: <code>ceil(56.34) = 57.0</code>	double	double
<code>cos(x)</code>	<code>&lt;cmath&gt;</code>	Returns the cosine of angle x: <code>cos(0.0) = 1.0</code>	double (radians)	double
<code>exp(x)</code>	<code>&lt;cmath&gt;</code>	Returns $e^x$ , where $e = 2.718$ : <code>exp(1.0) = 2.71828</code>	double	double
<code>fabs(x)</code>	<code>&lt;cmath&gt;</code>	Returns the absolute value of its argument: <code>fabs(-5.67) = 5.67</code>	double	double
<code>floor(x)</code>	<code>&lt;cmath&gt;</code>	Returns the largest whole number that is not greater than x: <code>floor(45.67) = 45.00</code>	double	double
<code>pow(x, y)</code>	<code>&lt;cmath&gt;</code>	Returns $x^y$ ; if x is negative, y must be a whole number: <code>pow(0.16, 0.5) = 0.4</code>	double	double



# Program example



```
#include<stdio.h>
#include<math.h>
int main()
{
    double x=9.0,y=8.0,z=7.0;
    printf("\nLog value is:%lf",log(x));
    printf("\nLog value with base 10 is:%lf",log10(x));
    printf("\nExponential value is:%lf",exp(x));
    printf("\n Ceil value is:%lf",ceil(8.94));
    printf("\n Floor value is:%lf",floor(2.34));
    printf("\n Power:%lf",pow(3.0,2.0));
    printf("\n Floating absolute is:%lf",fabs(-2.9));
    printf("\n Square root value is:%lf",sqrt(9));
    printf("\n Sin:%f,cos:%f,tan:%lf",sin(x),cos(y),tan(z));
    printf("\n fMod:%f",fmod(2.0,1.5));
    return 0;
}
```

# Q1



What is the output of the following code?

```
void my_recursive_function(int n)
```

```
{
```

```
    if(n == 0)
```

```
        return;
```

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

```
    my_recursive_function(n-1);
```

```
}
```

```
int main()
```

```
{
```

```
    my_recursive_function(5);
```

```
    return 0;
```

```
}
```

A. 5

B. 1

C. 5 4 3 2 1 0

D. 5 4 3 2 1

Predict output of following program

```
#include <stdio.h>
int fun(int n)
{
    if (n == 4)
        return n;
    else return 2*fun(n+1);
}
int main()
{
    printf("%d ", fun(2));
    return 0;
}
```

- A. 4
- B. 8
- C. 16
- D. 10

# Q3



Consider the following recursive function fun(x, y). What is the value of fun(4, 3)

```
int fun(int x, int y)
{
    if (x == 0)
        return y;
    return fun(x - 1, x + y);
}
```

- A. 13
- B. 12
- C. 9
- D. 10

What does the following function print for  $n = 25$ ?

```
void fun(int n)
{
    if (n == 0)
        return;
    printf("%d", n%2);
    fun(n/2);
}
```

- A. 11001
- B. 10011
- C. 11111
- D. 00000

# CSE101-lec#13

- Storage Classes and Scope Rules

# Outline

- Storage Classes
  - auto
  - static
  - extern
  - register
- Scope Rules

# Storage Classes

- Storage Classes are used to describe the features of a variable. These features basically include the scope, visibility and life-time which help us to trace the existence of a particular variable during the runtime of a program.
- Storage class specifies four things-
  - I. Storage location*: Where the variable will be stored?[ In memory /or CPU register]
  - II. Scope*: Block in which the variable is accessible.
  - III. Lifetime*: How long would the variable exist?
  - IV. Default initial value*: What will be the initial value of the variable, if initial value is not specifically assigned ?



# Storage Classes: Auto

- Automatic storage
  - `auto int x, y;`
  - It is the default storage class for a local variable
  - This is the default storage class for all the variables declared inside a function or a block
- Storage – **Memory(RAM).**
- Default initial value – An unpredictable value, which is often called a **garbage value.**
- Scope – **Local to the block** in which the variable is defined.
- Lifetime – Till the control remains within the block in which the variable is defined.



# Program example-auto storage class

```
#include<stdio.h>
void func1()
{
    auto int a=10;  // Local variable of func1()
    printf("\n a=%d",a);
}
void func2()
{
    auto int a=20; //Local variable of func2()
    printf("\n a=%d",a);
}
int main()
{
    auto int a=30;//Local variable of main()
    func1();
    func2();
    printf("\n a=%d",a);
    return 0;
}
```

# Storage Classes: Register

- register: tries to put variable into high-speed registers.
  - `register int counter = 1;`
- Storage - **CPU registers**.
- Default initial value - **Garbage value**.
- Scope - **Local** to the block in which the variable is defined.
- Lifetime - Till the control remains within the block in which the variable is defined.

# More points in relation to register storage class

- This storage class declares register variables which have the same functionality as that of the auto variables. The only difference is that the compiler tries to store these variables in the register of the microprocessor if a free register is available.
- This makes the use of register variables to be much faster than that of the variables stored in the memory during the runtime of the program. If a free register is not available, these are then stored in the memory only.
- Usually few variables which are to be accessed very frequently in a program are declared with the register keyword which improves the running time of the program. An important and interesting point to be noted here is that we cannot obtain the address of a register variable using pointers.



# Program example-register storage class

```
#include<stdio.h>
int main()
{
    register int i; // i will be used frequently so, it can be given register storage class
    for(i=1;i<=20;i++)
    {
        printf("\n%d",i);
    }
    return 0;
}
```

# Storage Classes: Static

- Static storage
  - Storage – **Memory(RAM)**.
  - Default initial value – **Zero**.
  - Scope – **Local** to the block in which the variable is defined.
  - Life time – variable will retain its value throughout the program



# More points in relation to static storage class

- Static variables have a property of preserving their value even after they are out of their scope! Hence, static variables preserve the value of their last use in their scope.
- So we can say that they are initialized only once and exist till the termination of the program. Thus, no new memory is allocated because they are not re-declared.
- Their scope is local to the function to which they were defined. Global static variables can be accessed anywhere in the program. By default, they are assigned the value 0 by the compiler.

# Program example-static storage class



```
#include<stdio.h>
void function();
int main()
{
    function();
    function();
    function();
    return 0;
}
void function()
{
    int a=10;
    static int b=10;
    printf("\n Value of a:%d, Value of
b:%d",a,b);
    a++;
    b++;
}
```

Output:

Value of a:10, Value of b:10

Value of a:10, Value of b:11

Value of a:10, Value of b:12



# Storage Classes: extern

Extern storage class simply tells us that the variable is defined elsewhere and not within the same block where it is used. Basically, the value is assigned to it in a different block and this can be overwritten/changed in a different block as well

- Storage – **Memory(RAM)**.
- Default initial value – **Zero**.
- Scope – **Global**.
- Life – As long as the program's execution doesn't come to an end.



# More points in relation to extern storage class

- extern variable is nothing but a global variable initialized with a legal value where it is declared in order to be used elsewhere. It can be accessed within any function/block.
- Also, a normal global variable can be made extern as well by placing the 'extern' keyword before its declaration/definition in any function/block.
- This basically signifies that we are not initializing a new variable but instead we are using/accessing the global variable only. The main purpose of using extern variables is that they can be accessed between two different files which are part of a large program.

# Program example 1-extern storage class

## External variable in the same file



```
#include<stdio.h>
void first();
int main()
{
    extern int x; /* declaration in main() */
    printf("\nx=%d",x); // x is used before its definition[Possible because of extern]
    first();
    printf("\nx=%d",x);// Changes done by first are visible here
    return 0;
}
void first()
{
    extern int x; /* declaration in first() */
    printf("\nx=%d",x); // x is used again before its definition[Possible because of extern]
    x=x+10;
}
int x=10; /* definition of external variable, here x is global variable */
```



# Program example 2-extern storage class

## External variable in different file

### extern1.c file

```
#include<stdio.h>
#include"extern2.c"
//Global variable declared in extern1.c
int x=30;
int main()
{
    print();
    printf("%d",x);//Changes done by extern2.c
    file are also reflected
}
```

### extern2.c file

```
void print()
{
    extern int x;//Taking reference of
    global variable in different file or
    Declaration
    printf("%d\n",x);
    x=x+10;
}
//Output:
30
40
```

# Summary

Storage class	Auto	Static	Register	Extern
Default Initial Value	Garbage value	0 (Zero)	Garbage	0(Zero)
Location	RAM	RAM	CPU registers	RAM
Scope	Local to the variable where the variable is defined	Local to the variable where the variable is defined	Local to the variable where the variable is defined	Entire Program
Life	As long as the control is within the block where the variable is defined	As long as the program is under execution	As long as the control is within the block where the variable is defined	As long as the program is under execution



# Scope Rules

- The *scope* of a variable is the portion of a program where the variable has meaning (where it exists).
- A global variable has global (unlimited) scope.
- A local variable's scope is restricted to the function that declares the variable.
- A block variable's scope is restricted to the block in which the variable is declared.

# Local variables

- Parameters and variables declared inside the definition of a function are *local*.
- They only exist inside the function body.
- Once the function returns, the variables no longer exist!



# Example-local variable

```
#include<stdio.h>
void function();
int main()
{
    int a=1,b=2;
    printf("\n a is:%d,b is:%d",a,b);//a,b are local variables of main()
    function();
    return 0;
}
void function()
{
    int c=3;
    printf("\n Value of c is:%d",c);// c is a local variable of function
}
```



# Block Variables

- You can also declare variables that exist only within the *body* of a compound statement (*a block*):

```
{  
  int f;  
  ...  
  ...  
}
```



# Example-block scoped variable

```
#include<stdio.h>
int main()
{
    int b=2;
    {
        int a=1; // a is block variable
        printf("\nValue of a is:%d",a);
    }
    printf("\nValue of b is:%d",b);
    return 0;
}
```

# Global variables

- You can declare variables outside of any function definition – these variables are *global variables*.
- Any function can access/change global variables.



# Example-global variable

```
#include<stdio.h>
int a=1;// a is a global variable
void print();
int main()
{
    printf("\nValue of a is:%d",a);
    print();
    return 0;
}
void print()
{
    printf("\nValue of a is:%d",a);
}
```



# More examples-Scope rules

```
#include<stdio.h>
int a=10;// a is a global variable
void print();
int main()
{
    int a=1;
    printf("\nValue of a is:%d",a);// It will
access local a
    print();
    return 0;
}
void print()
{
    printf("\nValue of a is:%d",a); //It will
access global a
}
```

Output:

Value of a is:1

Value of a is:10

Note:

When we have same named local and global variables, priority is always given to local variable first, that is why in main() function value of local a is printed, whereas in function definition, there is no local variable so preference is given to global version of a



# More examples-Scope rules

```
#include<stdio.h>
int a=10;// a is a global variable
void print();
int main()
{
    printf("\nValue of a is:%d",a);
    print();
    printf("\nValue of a is:%d",a);// Change
done by print to global a is reflected here
    return 0;
}
void print()
{
    printf("\nValue of a is:%d",a);
    a=20;
}
```

Output:

Value of a is:10  
Value of a is:10  
Value of a is:20



# More examples-Scope rules

```
#include<stdio.h>
int main()
{
    int a=5;
    {
        int a=50;
        {
            int a=500;
            printf("\na:%d",a);
        }
        printf("\na:%d",a);
    }
    printf("\na:%d",a);
    return 0;
}
```

Output:

a:500  
a:50  
a:5

# Q1



What will be the output of the following C code?

```
#include <stdio.h>
int x;
void m();
int main()
{
    m();
    printf("%d", x);
    return 0;
}
void m()
{
    x = 4;
}
```

A. 0

B. 4

C. Compile time error

D. Runtime error



# Q2



What will be the output of the following C code?

```
#include <stdio.h>

int x = 5;
void m();
void n();
int main()
{
    int x = 3;
    m();
    printf("%d", x);
    return 0;
}

void m()
{
    x = 8;
    n();
}

void n()
{
    printf("%d", x);
}
```

A. 8 3

B. 3 8

C. 8 5

D. 5 3

# Q3

What will be the output of following code?

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

```
int main()
```

```
{
```

```
    int x=1;
```

```
    {
```

```
        x=2;
```

```
        {
```

```
            int x=3;
```

```
        }
```

```
    }
```

```
    printf("%d",x);
```

```
    return 0;
```

```
}
```

A. 1

B. 2

C. 3

D. Compile time error

In case of a conflict between the names of a local and global variable what happens?

- A. The global variable is given a priority.
- B. The local variable is given a priority.
- C. Which one will get a priority depends upon which one is defined first.
- D. The compiler reports an error.

What will be the storage class of variable i in the code written below?

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

```
int main()
```

```
{
```

```
int i = 10;
```

```
printf("%d",i);
```

```
return 0;
```

```
}
```

- A. Automatic storage class
- B. Extern storage class
- C. Static storage class
- D. Register storage class

What will be the behaviour of following code?

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

```
int main()
```

```
{
```

```
register int a;
```

```
printf("\nEnter value of a:");
```

```
scanf("%d",&a);
```

```
return 0;
```

```
}
```

A. Program will work normally

B. Compile time error

C. Runtime error

D. None of the above

# Q7



```
#include<stdio.h>
int incr(int i)
{
    static int count = 0;
    count = count + i;
    return (count);
}
int main()
{
    int i,j;
    for (i = 0; i <=2; i++)
        j = incr(i);
    printf("%d",j);
return 0;
}
```

- A. 10
- B. 4
- C. 3
- D. 2

# Q8



```
#include<stdio.h>
void update();
int main()
{
    update();
    update();
    return 0;
}
void update()
{
    auto int a=1;
    static int b=1;
    a++;
    b++;
    printf("%d,%d\n",a,b);
}
```

A. 2,2

2,3

B. 2,2

2,2

C. 1,1

1,2

D. 2,1

2,2

```
#include<stdio.h>
int main()
{
extern int a;
printf("%d",++a);
return 0;
}
int a;
```

- A. 0
- B. 1
- C. -1
- D. Compile time error





# CSE101-Lec 10 and 11

User defined functions(Introduction to Functions)

Parameter Passing mechanisms(Call by value and Call by reference)

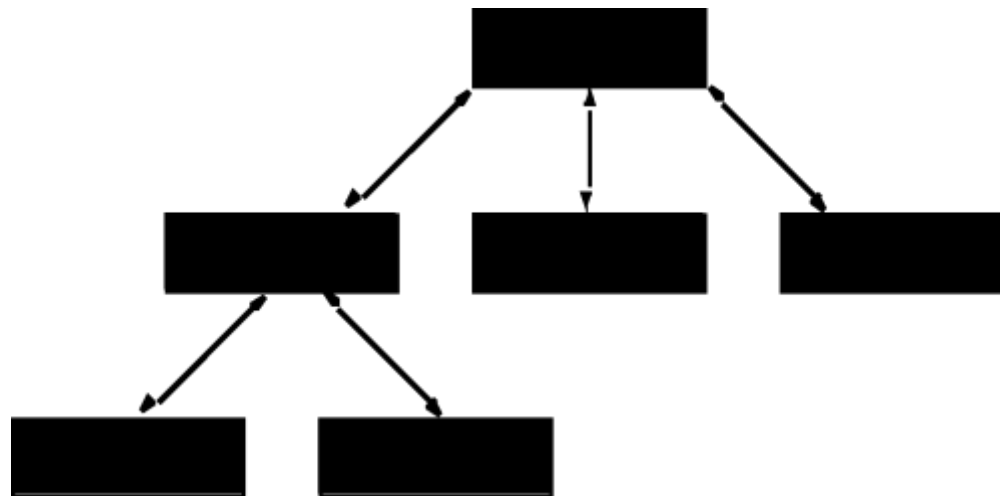
# Functions

# Divide and Conquer

- Best way to solve a problem is by dividing the problem and solving it.
- **Divide and conquer**
  - Construct a program from smaller pieces or components
    - These smaller pieces are called **modules**
  - Each module more manageable than the original program

# Program Modules in C

Hierarchical boss function/worker function relationship.



# Program Modules in C

- Functions
  - Modules in C are called functions.
  - **Two categories of functions: Pre-defined(or library functions) and User defined**
  - Programs combine user-defined functions with library functions
    - C standard library has a wide variety of functions for performing common *mathematical calculations, string manipulations, character manipulations, input/output* and many more.
    - C standard library makes your job easier.
    - Functions like `printf()`, `scanf()`, `pow()` are standard library functions.
    - We can also write functions to define some specific task in a program and these functions are called **user-defined functions.**

# What are functions??

- Functions
- A function is a set of statements that take inputs, do some specific computation and produces output.
- The idea is to put some commonly or repeatedly done task together and make a function so that instead of writing the same code again and again for different inputs, we can call the function.

# Benefits of functions

- Functions help us in reducing code redundancy. If functionality is performed at multiple places in software, then rather than writing the same code, again and again, we create a function and call it everywhere. This also helps in maintenance as we have to change at one place if we make future changes to the functionality.
- Functions make code modular. Consider a big file having many lines of codes. It becomes really simple to read and use the code if the code is divided into functions.
- Functions provide abstraction. For example, we can use library functions without worrying about their internal working.



Three important parts/ or components of any user defined function

- Function declaration/ or Function prototype/ or Function signature
- Function definition
- Function calling



# Function declaration/ or Function prototype/ or Function signature

- A function declaration tells the compiler about the number of parameters function takes, data-types of parameters and return type of function. Putting parameter names in function declaration is optional in the function declaration, but it is necessary to put them in the definition. Below are an example of function declarations. (parameter names are not there in below declarations)

## Syntax:

return\_type function\_name( parameter list/or arguments );

## Example 1:

int example(int,int); // Here function is returning integer value and accepting two integer arguments

## Example 2:

void example(); // Here function is returning nothing and it is accepting no argument

# Function definition

- Function definition consists of the set of statements that are required to be executed when the function is used.
- It consists of body of the function along with function header
- Function header is same as function declaration but there is no semicolon after it
- Function header is followed by function body where actual functionality of the function is placed



# Function definition-Example

## Syntax:

```
return_type function_name( parameter list ) //Function Header
{
    //body of the function    // Function body
}
```

## Example:

```
int sum(int x,int y) // sum is a user defined function which takes two integer parameters
{
    int z;
    z=x+y;
    return z;          //After adding two integers it is returning the result
}
```

# Function Calling

- Once the function is declared and defined, we wish to use it. So, for using the function we need to call it
- When the function is called in the main() function/ or some other function, the control is transferred to the called function, once the function definition is executed completely, the control will come back to the main()/ or the other function in which the calling statement has been placed, and the execution will resume from the same point where it was halted.
- Syntax of function calling:

`function_name(<List_of_argument_values>);`

Example: `function(2,3);`

# Points to remember during function calling

- Function name and the number and type of arguments in the function call must be same as that given in the function declaration and function header of the function definition
- Names (and not the types) of variables in function declaration, function call and header of function definition may vary
- Arguments may be passed in the form of expressions to the called function. In such a case, arguments are first evaluated and converted to the type of formal parameter and then the body of the function gets executed.
- If the return type of the function is not void, then the value returned by the called function may be assigned to some variable as given below.

**variable\_name = function\_name(variable1, variable2, ...);**

# Program example of function



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

```
// FUNCTION DEFINITION
```

```
int sum(int a, int b);// FUNCTION DECLARATION
```

```
int sum ( int a, int b)// FUNCTION HEADER
```

```
int main()
```

```
{
```

```
{
```

```
// FUNCTION BODY
```

```
int num1, num2, total = 0;
```

```
return (a + b);
```

```
printf("\n Enter the first number : ");
```

```
}
```

```
scanf("%d", &num1);
```

```
printf("\n Enter the second number : ");
```

```
scanf("%d", &num2);
```

```
total = sum(num1, num2);
```

```
//
```

```
FUNCTION CALL
```

```
printf("\n Total = %d", total);
```

```
return 0;
```

```
}
```

# return statement in functions

- The **return** statement is used to terminate the execution of a function and return control to the calling function. When the return statement is encountered, the program execution resumes in the calling function at the point immediately following the function call.
- Calling function is one in which the function is called(usually main()), and the called function is one whose definition is placed outside main() or/ some other function
- A **return** statement returns a value to the calling function. The syntax of return statement can be given as

**return** *<expression>* ;

- Here expression is placed in between angular brackets because specifying an expression is optional. The value of *expression*, if present, is returned to the calling function. However, in case *expression* is omitted, the return value of the function is undefined.
- Programmer may or may not place the *expression* within parentheses.
- For functions that has no **return** statement, the control automatically returns to the calling function after the last statement of the called function is executed.

# Types of user defined functions

- 1) **Function with no argument and no return value** :When a function has no arguments, it does not receive any data from the calling function. Similarly when it does not return a value, the calling function does not receive any data from the called function. (input, output statements and actual logic ,all will be placed in the function definition)
- 2) **Function with arguments but no return value** : When a function has arguments, it receive data from the calling function but it returns no values.(input will be taken in the calling function, whereas output statement and logic will be placed in the function definition)
- 3) **Function with no arguments but returns a value** : There could be occasions where we may need to design functions that may not take any arguments but returns a value to the calling function. (input statement and logic will be placed in function definition and output statement will be placed in calling function)
- 4) **Function with arguments and return value**: This kind of function returns a value as well as it can accept arguments also.(input and output statements will be placed in calling function and logic will be placed in function definition)



## Program example: **Function with no argument and no return value**

Write a program to add two numbers using function with no argument and no return value

```
#include<stdio.h>
void add(); //Function declaration
int main()
{
    add(); //Function calling
    return 0;
}
//Function definition
void add()
{
    int a,b,sum;
    printf("\n Enter two numbers:");
    scanf("%d%d",&a,&b);
    sum=a+b;
    printf("\n Sum of two numbers is:%d",sum);
}
```

- Here input, output statements and logic all are placed in the function definition

## Program example: **Function with arguments but no return value**

Write a program to add two numbers using function with argument but no return value

```
#include<stdio.h>
void add(int,int); //Function declaration
int main()
{
    int a,b;
    printf("\n Enter two numbers:");
    scanf("%d%d",&a,&b);
    add(a,b); //Function calling //a, b are actual arguments
    return 0;
}
//Function definition
void add(int x,int y) //x and y are formal arguments
{
    int sum;
    sum=x+y;
    printf("\n Sum of two numbers is:%d",sum);
}
```

- Here input is taken in calling function(i.e. `main()`) whereas output statement and logic is placed in the function definition
- Note: Arguments in function call statements are actual arguments, whereas arguments in function definition are formal arguments

## Program example: **Function with no arguments but returns a value**

Write a program to add two numbers using function no arguments but return a value

```
#include<stdio.h>
int add(); //Function declaration
int main()
{
    int result;
    result=add(); //Function calling
    printf("\n Sum is:%d",result);
    return 0;
}
//Function definition
int add()
{
    int a,b,sum;
    printf("\n Enter two numbers:");
    scanf("%d%d",&a,&b);
    sum=a+b;
    return sum;
}
```

- Here input statement and logic is placed in the function definition and then it returns sum, and output statement is written in the calling function(i.e. main())

## Program example: **Function with arguments and return value**

Write a program to add two numbers using function with arguments and return a value

```
#include<stdio.h>
int add(int,int); //Function declaration
int main()
{
    int a,b,result;
    printf("\n Enter two numbers:");
    scanf("%d%d",&a,&b);
    result=add(a,b); //Function calling //a, b are actual arguments
    printf("\n Sum is:%d",result);
    return 0;
}
//Function definition
int add(int x,int y) //x and y are formal arguments
{
    int sum;
    sum=x+y;
    return sum;
}
```

- Here input and output statements are placed in the calling function (i.e. main()) and logic is placed in the function definition
- Note: Arguments in function call statements are actual arguments, whereas arguments in function definition are formal arguments



## Parameter Passing mechanisms in C/ or Argument Passing techniques in C/ or Calling mechanisms in C

There are two ways in which arguments or parameters can be passed to the called function.

- **Call by value** in which values of the variables are passed by the calling function to the called function.
- **Call by reference** in which address of the variables are passed by the calling function to the called function

# Call by value



- In the Call by Value method, the called function creates new variables to store the value of the arguments passed to it. Therefore, the called function uses a copy of the actual arguments to perform its intended task.
- If the called function is supposed to modify the value of the parameters passed to it, then the change will be reflected only in the called function. In the calling function no change will be made to the value of the variables.

```
#include<stdio.h>
void add( int n);
int main()
{
    int num = 2;
    printf("\n The value of num before calling the function = %d", num);
    add(num);
    printf("\n The value of num after calling the function = %d", num);
    return 0;
}
void add(int n)
{
    n = n + 10;
    printf("\n The value of num in the called function = %d", n);
}
```

## **The output of this program is:**

The value of num before calling the function = 2

The value of num in the called function = 12

The value of num after calling the function = 2



# Advantages and Disadvantages of Call by value

## Advantages:

- The method doesn't change the original variable, so it is preserving data.
- Whenever a function is called it, never affect the actual contents of the actual arguments.
- The other advantage could be arguments can be variables(e.g. x), literals(e.g. 6,0...), or expressions(e.g. x+1)

## Disadvantages:

- Copy of the variable is created hence it can consume additional storage space
- Along with more space, it can take a lot of time to copy, thereby resulting in performance penalty, especially if the function is called many times

# Call by reference

- When the calling function passes arguments to the called function using call by value method, the only way to return the modified value of the argument to the caller is explicitly using the return statement. The better option when a function can modify the value of the argument is to pass arguments using call by reference technique.
- In call by reference, we declare the function parameters as pointers rather than normal variables. When this is done, any changes made by the function to the arguments, are visible to the calling program.
- To indicate that an argument is passed using call by reference, address of the variable is passed as an actual argument during function calling, and pointers will act as formal arguments in the function definition to which the addresses will be assigned



## Call by reference(Program example)

```
#include<stdio.h>
void add( int *n);
int main()
{
    int num = 2;
    printf("\n The value of num before calling the function = %d", num);
    add(&num);
    printf("\n The value of num after calling the function = %d", num);
    return 0;
}
void add( int *n)
{
    *n = *n + 10;
    printf("\n The value of num in the called function = %d", *n);
}
```

**The output of this program is:**

The value of num before calling the function = 2

The value of num in the called function = 12

The value of num after calling the function = 12

# Advantages and Disadvantages of Call by reference

## Advantages:

- It provides greater time and space efficiency as duplicate copies of the arguments are not created
- In this method, there is no copy of the argument made. Therefore it is processed very fast.

## Disadvantages:

- Original values are not safe, as they can be modified
- We can only pass addresses as actual arguments (we cannot pass: literals(1,2,3...) / or expressions(x+1,y+2..) as actual arguments

# Call by value vs Call by reference

Parameters	Call by value	Call by reference
Definition	While calling a function, when you pass values by copying variables, it is known as "Call By Values."	While calling a function, in programming language instead of copying the values of variables, the address of the variables is used it is known as "Call By References.
Arguments	In this method, a copy of the variable is passed.	In this method, address of the variable is passed.
Effect	Changes made in a copy of variable never modify the value of variable outside the function.	Change in the variable also affects the value of the variable outside the function.
Alteration of value	Does not allow you to make any changes in the actual variables.	Allows you to make changes in the values of variables by using function calls.
Passing of variable	Values of variables are passed using a straightforward method.	Pointer variables are required to store the address of variables.
Value modification	Original value not modified.	The original value is modified.

# Q1

Which of the following is a correct format for declaration of function?

- A. return-type function-name(argument type);
- B. return-type function-name(argument type){}
- C. return-type (argument type)function-name;
- D. all of the mentioned

## Q2

Which function definition will run correctly?

a) `int sum(int a, int b)`

`return (a + b);`

b) `int sum(int a, int b)`

`{return (a + b);}`

c) `int sum(a, b)`

`return (a + b);`

d) none of the mentioned

## Q3

Which of the following is a valid function declaration statement?

- A. `function1(int,int);`
- B. `void function1(a,b);`
- C. `void 1function(int,int);`
- D. `void function1(int,int);`

# Q4

Which of the following is a valid function call statement for user defined function: abc

- a) `abc(int);`
- b) `void abc();`
- c) `abc();`
- d) `int abc(int);`

What will be the output of the following C code?

```
#include <stdio.h>

void m()
{
    printf("hi");
}

int main()
{
    m();
    return 0;
}
```

- A. hi
- B. Run time error
- C. Nothing will be displayed
- D. Compile time error



What will be the output of the following C code?

```
#include <stdio.h>

void m();
int main()
{
    m();
    return 0;
}

void m()
{
    printf("hi");
    m();
}
```

- A. Compile time error
- B. hi
- C. Infinite hi
- D. Nothing