COMPUTER PROGRAMMING
POWER POINT PRESENTATION

- **Year : 2017 - 2018**
- **Subject Code : ACS001**
- **Regulations : IARE-R16**
- **Class : II Semester**
- **Branch : AE / ME / CE**

Team of Instructors

Mr. N. Ramanjaneya Reddy, Associate Professor, CSE

Mr. N. Poorna Chandra Rao, Assistant Professor, CSE

Mr. S. Lakshman Kumar, Assistant Professor, CSE

Ms. A. Uma Datta, Assistant Professor, IT

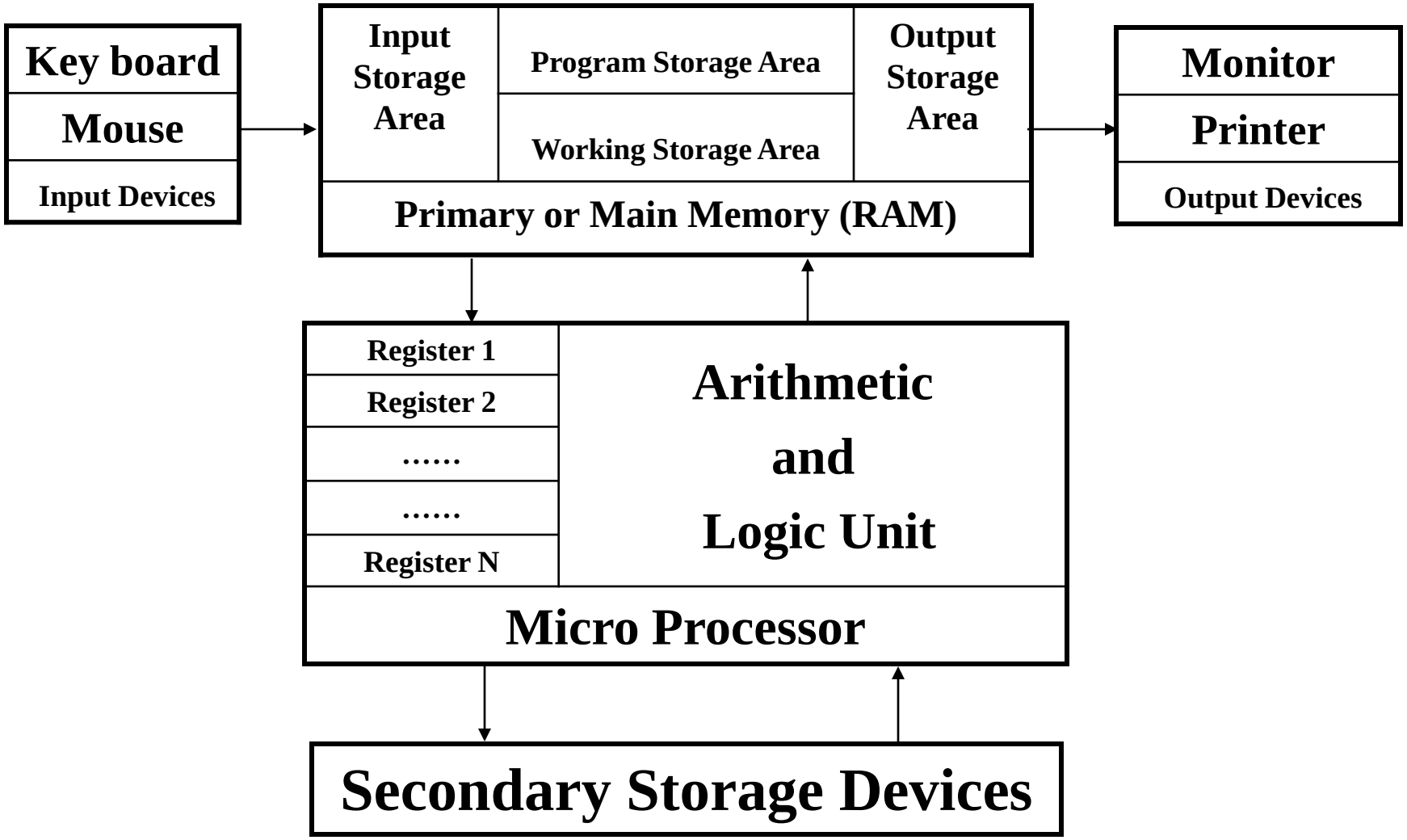
Ms. A. Swapna, Assistant Professor, IT

Ms. A. Lakshmi, Assistant Professor, IT

UNIT-I

Introduction to computers: Computer systems, computing environments, computer languages, creating and running programs, algorithms, flowcharts; Introduction to C language: History of C, basic structure of C programs, process of compiling and running a C program, C tokens, keywords, identifiers, constants, strings, special symbols, variables, data types; Operators and expressions: Operators, arithmetic, relational and logical, assignment operators, increment and decrement operators, bitwise and conditional operators, special operators, operator precedence and associativity, evaluation of expressions, type conversions in expressions, formatted input and output.

Computer -- Hardware



Algorithm: Step by step procedure of solving a particular problem.
Pseudo code: Artificial informal language used to develop algorithms.
Flow chart: Graphical representation of an algorithm.
Algorithm to find whether a number even or odd:

Step1: Begin
Step2: Take a number
Step3: if the number is divisible by2 then
print that number is even
otherwise print that number is odd

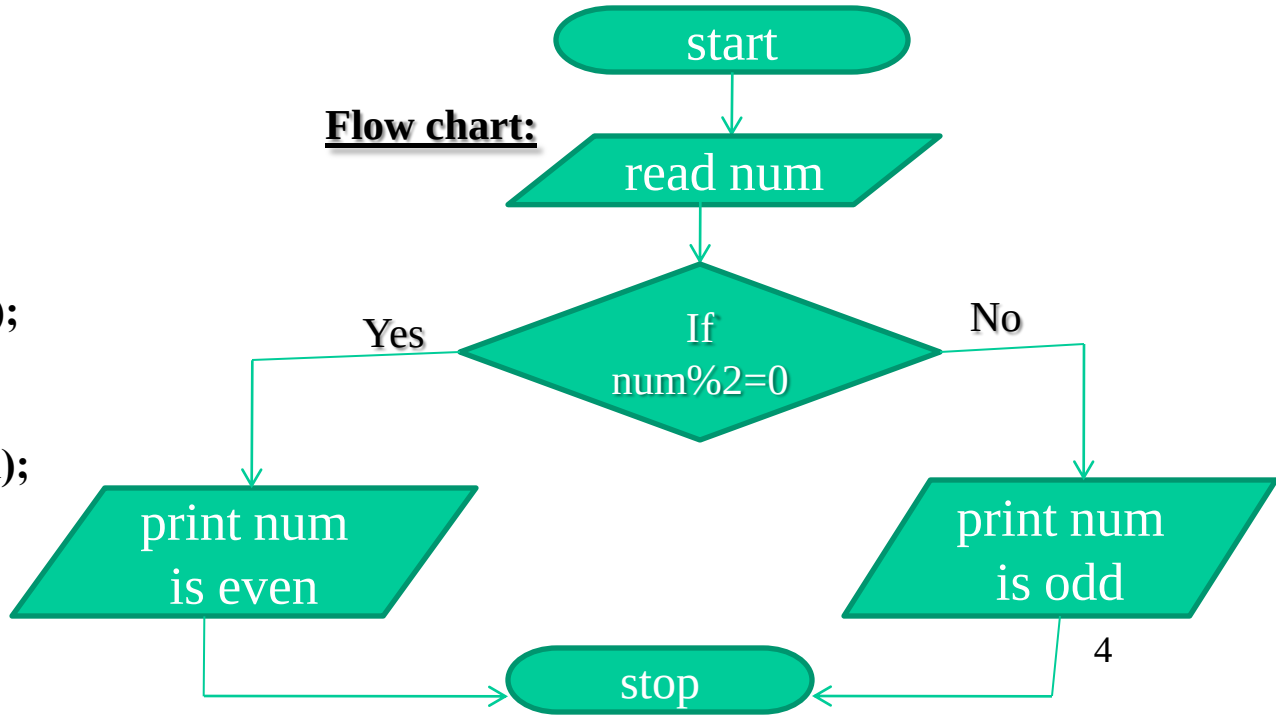
Step4: End
(Algorithm in natural language)

Step1: START
Step2: Read num
Step3: if(num%2=0) then
print num is even
otherwise
print num is odd
Step4: STOP
(Algorithm by using pseudo code)

```
#include<stdio.h>
#include<conio.h>
main()
{
    int num;
    printf("Enter any number");
    scanf("%d",&num);
    if(num%2==0)
        printf("%d is even",num);
    else
        printf(%d is odd",num);
}
```

(Program in C language)

Flow chart:

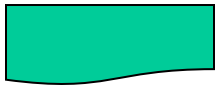


Flow chart symbols

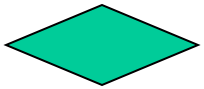
Oval — **Terminal**

Parallegram — **Input/output**

Rectangle — **Process**



Document — **Hard copy**



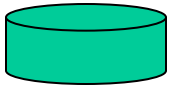
Diamond — **Decision**

Circle — **Connector**

Double sided Rectangle — **Sub program**

Hexagon — **Iteration**

Trapezoid — **Manual Operation**



Cylinder — **Magnetic Disk Storage**

Machine Language – Assembly Language – High-Level Language

```
 1              00000000  00000100  0000000000000000
 2   01011110  00001100  11000010  0000000000000010
 3              11101111  00010110  0000000000000101
 4              11101111  10111110  0000000000001011
 5   11111000  10101101  11011111  0000000000010010
 6              01100010  11011111  0000000000010101
 7   11101111  00000010  11111011  0000000000010111
 8   11110100  10101101  11011111  0000000000011110
 9   00000011  10100010  11011111  0000000000100001
10   11101111  00000010  11011111  0000000000100100
11   01111110  11110100  10101101
12   11111000  10101110  11000101  0000000000101011
13   00000110  10100010  11111011  0000000000110001
14   11101111  00000010  11111011  0000000000110100
15              01010000  11010100  0000000000111011
16                        00000100  0000000000111101
```

```
1        entry     main,^m<r2>
2        sub12     #12,sp
3        jsb       C$MAIN_ARGS
4        moveb     $CHAR_STRING_CON
5
6        pusha1    -8(fp)
7        pusha1    (r2)
8        calls     #2,SCANF
9        pusha1    -12(fp)
10       pusha1    3(r2)
11       calls     #2,SCANF
12       mull3     -8(fp),-12(fp),-
13       pusha     6(fp)
14       calls     #2,PRINTF
15       clrl      r0
16       ret
```

```
1   #include<stdio.h>
2   int main(void)
3   {
4       int n1, n2,product;
5       printf(“Enter two numbers : “);
6       scanf(“%d  %d”,&n1,&n2);
7       product  =  n1  *  n2;
8       printf(“%d”,product);
9       return 0;
10  }
```

The only language the computer can understand is machine language (binary language).

A high level language is an English like language where one instruction typically translates into a series of machine-language instructions.

A low level language corresponds closely to machine code so that a single low-level language instruction translates to a single machine language instruction.

Structure of C program

```
/*Program to find
   area and perimeter of Circle */
#include<stdio.h>
#define PI 3.1415
float radius;
float area();
float perimeter();
int main()
{
    float a, p;
    printf("Enter radius : ");
    scanf("%f",&radius);
    a = area();
    p = perimeter();
    printf("Area of Circle : %f",a);
    printf("Perimeter : %f",p);
}
float area()
{
   return (PI * radius * radius);
}
float perimeter()
{
    return (2 * PI * radius);
}
```

Documentation Section

Linkage Section

Definition Section

Global Declaration Section

Main Function Section

Local Declaration Part
Executable Code Part

Sub Program Section

Function1()
Function2()
……
FunctionN()

Program Development Steps

1)Statement of Problem

a) Working with existing system and using proper questionnaire, the problem should be explained clearly.

b) What inputs are available, outputs are required and what is needed for creating workable solution should be understood clearly.

2)Analysis

a) The method of solutions to solve the problem can be identified.

b) We also judge that which method gives best results among different methods of solution.

3)Designing

a) Algorithms and flow charts will be prepared.

b) Keep focus on data, architecture, user interfaces and program components.

4)Implementation

The algorithms and flow charts developed in the previous steps are converted into actual programs in the high level languages like C.

4.a)Compilation

Translate the program into machine code. This process is called as Compilation. Syntactic errors are found quickly at the time of compiling the program. These errors occur due to the usage of wrong syntaxes for the statements.

Eg: x=a*y+b

There is a syntax error in this statement, since, each and every statement in C language ends with a semicolon (;).

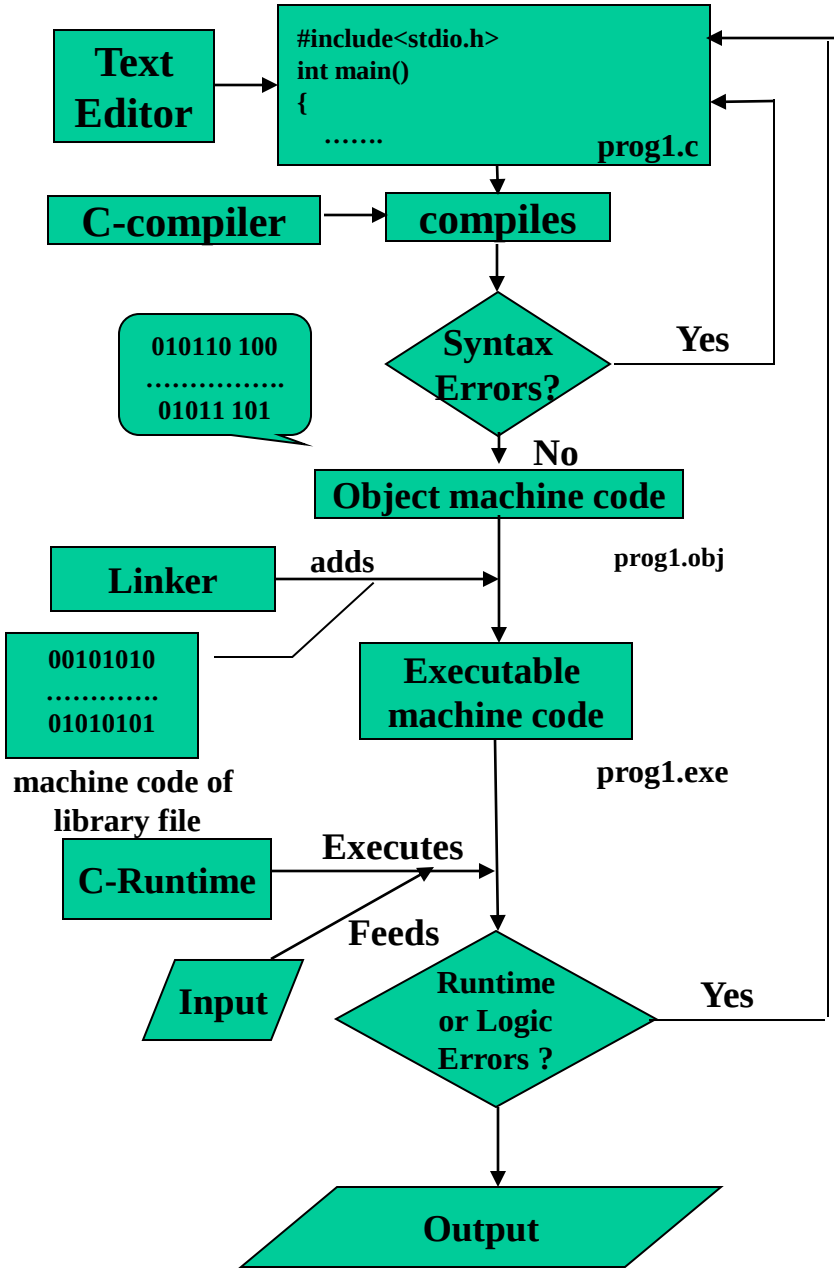
4.b)Execution

The next step is Program execution. In this phase, we may encounter two types of errors.

Runtime Errors: these errors occur during the execution of the program and terminates the program abnormally.

Logical Errors: these errors occur due to incorrect usage of the instructions in the program. These errors are neither detected during compilation or execution nor cause any stoppage to the program execution but produces incorrect output.

Executing a C program



Translators are system software used to convert high-level language program into machine-language code.

Compiler : Coverts the entire source program at a time into object code file, and saves it in secondary storage permanently. The same object machine code file will be executed several times, whenever needed.

Interpreter : Each statement of source program is translated into machine code and executed immediately. Translation and execution of each and every statement is repeated till the end of the program. No object code is saved. Translation is repeated for every execution of the source program.

Character Set of C-Language

Alphabets : A-Z and a-z
Digits : 0-9
Special Symbols : ~ ! @ # $ % ^ & ( ) _ - + = | \ { } [ ] : ; “ ‘
< > , . ? /
White Spaces : space , Horizontal tab, Vertical tab, New Line
Form Feed.

C-Language Keywords(C99)

auto	**double**	**int**	**struct**
break	**else**	**long**	**switch**
case	**enum**	**register**	**typedef**
char	**extern**	**return**	**union**
const	**float**	**short**	**unsigned**
continue	**for**	**signed**	**void**
default	**goto**	**sizeof**	**volatile**
do	**if**	**static**	**while**
_Bool	**_Imaginary**	**restrict**	**_Complex**
inline			

C-Tokens

Tokens : **The smallest individual units of a C- program are called Tokens. Key words, Identifiers, Constants, Operators, Delimiters.**

Key words : **have a predefined meaning and these meanings cannot be changed. All keywords must be written in small letters (except additional c99 keywords).**

Identifiers : **names of variables, functions, structures, unions, macros, labels, arrays etc.,**

Rules for define identifiers :

a) First character must be alphabetic character or under score

b) Second character onwards alphabetic character of digit or under score.

c) First 63 characters of an identifier are significant.

d) Cannot duplicate a key word.

e) May not have a space or any other special symbol except under score.

f) C – language is Case-sensitive.

C-Tokens

Constants : fixed values that do not change during execution of a program.

Boolean constants : 0 (false) and 1 (true)

Character constants :

only one character enclosed between two single quotes (except escape characters).

wide character type - wchar_t - for Unicode characters.

Integer constants : +123, -3454 , 0235 (octal value),
0x43d98 (hexa - decimal value)
54764U, 124356578L, 124567856UL

Float constants : 0.2 , 876.345, .345623 , 23.4E+8, 47.45e+6

String Constants : “Hello world” , “Have a nice day!”

Complex Constants : real part + imaginary part * I ex : 12.3 + 3.45 * I

Operators : a symbol, which indicates an operation to be performed.
Operators are used to manipulate data in program.

Delimiters : Language Pattern of c-language uses special kind of symbols

: (colon, used for labels) ; (semicolon terminates statement) ( ) parameter list
[ ] (array declaration and subscript), { } (block statement)
(hash for preprocessor directive) , (comma variable separator)

Data Types (pre defined)

Type	Typical Size in Bits	Minimal Range
char	8	–127 to 127
unsigned char	8	0 to 255
signed char	8	–127 to 127
int	16 or 32	–32,767 to 32,767
unsigned int	16 or 32	0 to 65,535
signed int	16 or 32	Same as int
short int	16	–32,767 to 32,767
unsigned short int	16	0 to 65,535
signed short int	16	Same as short int
long int	32	–2,147,483,647 to 2,147,483,647
long long int	64	$-(2^{63})$ to $2^{63} - 1$ (Added by C99)
signed long int	32	Same as long int
unsigned long int	32	0 to 4,294,967,295
unsigned long long int	64	$2^{64} - 1$ (Added by C99)
float	32	3.4e-38 to 3.4e+38
double	64	1.7e-308 to 1.7e+308
long double	80	3.4e-4932 to 1.1e+4932
void	--	data type that not return any value

Conversion Specifiers

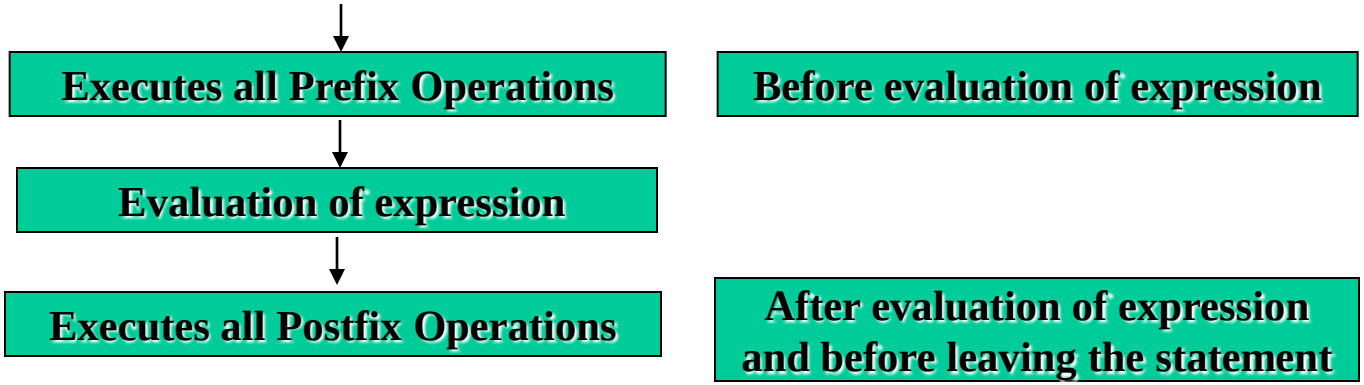
Code	Format
%a	Hexa decimal output in the form of 0xh.hhhhp+d(C99 only)
%s	String of characters (until null zero is reached)
%c	Character
%d	Decimal integer
%f	Floating-point numbers
%e	Exponential notation floating-point numbers
%g	Use the shorter of %f or %e
%u	Unsigned integer
%o	Octal integer
%x	Hexadecimal integer
%i	Signed decimal integer
%p	Display a pointer
%n	The associated argument must be a pointer to integer, This sepecifier causes the number of characters written in to be stored in that integer.
%hd	short integer
%ld	long integer
%lf	long double
%%	Prints a percent sign (%)

Back Slash (Escape Sequence) Characters

Code	Meaning
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\"	Double quote
\'	Single quote
\ \	Backslash
\v	Vertical tab
\a	Alert
\?	Question mark
\N	Octal constant (N is an octal constant)
\xN	Hexadecimal constant (N is a hexadecimal constant)

Increment and Decrement Operators

prefix increment (++a) **postfix increment (a++)** **prefix decrement(- -a)** **postfix decrement (a- -)**




```
/* prefix operators */
#include<stdio.h>
int main() {
    int  a = 7, b = 12, c;
    c = b * (++a) + 5 * (++a);
    printf(" a = %d", a);
    printf("\n b = %d",b);
    printf("\n c = %d",c);
}
```

Output:

```
    a = 9
    b = 12
    c = 153 ( 12 * 9 + 5 * 9)
```

```
/* prefix and postfix operators */
#include<stdio.h>
int main() {
    int  a = 7, b = 12, c;
    c = b * (a++) + 5 * (++a);
    printf(" a = %d", a);
    printf("\n b = %d",b);
    printf("\n c = %d",c);
}
```

Output:

```
    a = 9
    b = 12
    c = 136 ( 12 * 8 + 5 * 8)
```

```
/* postfix operators */
#include<stdio.h>
int main() {
    int  a = 7, b = 12, c;
    c = b * (a++) + 5 * (a++);
    printf(" a = %d", a);
    printf("\n b = %d",b);
    printf("\n c = %d",c);
}
```

Output:

```
    a = 9
    b = 12
    c = 119 ( 12 * 7 + 5 * 7)
```

Bitwise Logical Operators

& -- Bitwise AND

| -- Bitwise OR

^ -- Bitwise XOR

~ -- Bitwise NOT

A	B	A & B	A | B	A ^ B	~A
1	1	1	1	0	0
1	0	0	1	1	0
0	1	0	1	1	1
0	0	0	0	0	1

Bitwise AND

A (42) : 00000000 00101010
B (15) : 00000000 00001111

& (10) : 00000000 00001010

Bitwise OR

A (42) : 00000000 00101010
B (15) : 00000000 00001111

| (47) : 00000000 00101111

Bitwise XOR

A (42) : 00000000 00101010
B (15) : 00000000 00001111

& (37) : 00000000 00100101

Bitwise NOT

A (42) : 00000000 00101010

~ (-43) : 11111111 11010101

BITWISE SHIFT OPERATORS

Bitwise Left Shift (<<)

A (43) : 00000000 00101011

A << 2 : 00000000 10101100

Bitwise Right Shift (>>)
(positive values)

A (43) : 00000000 00101011

A >> 2 : 00000000 00001010

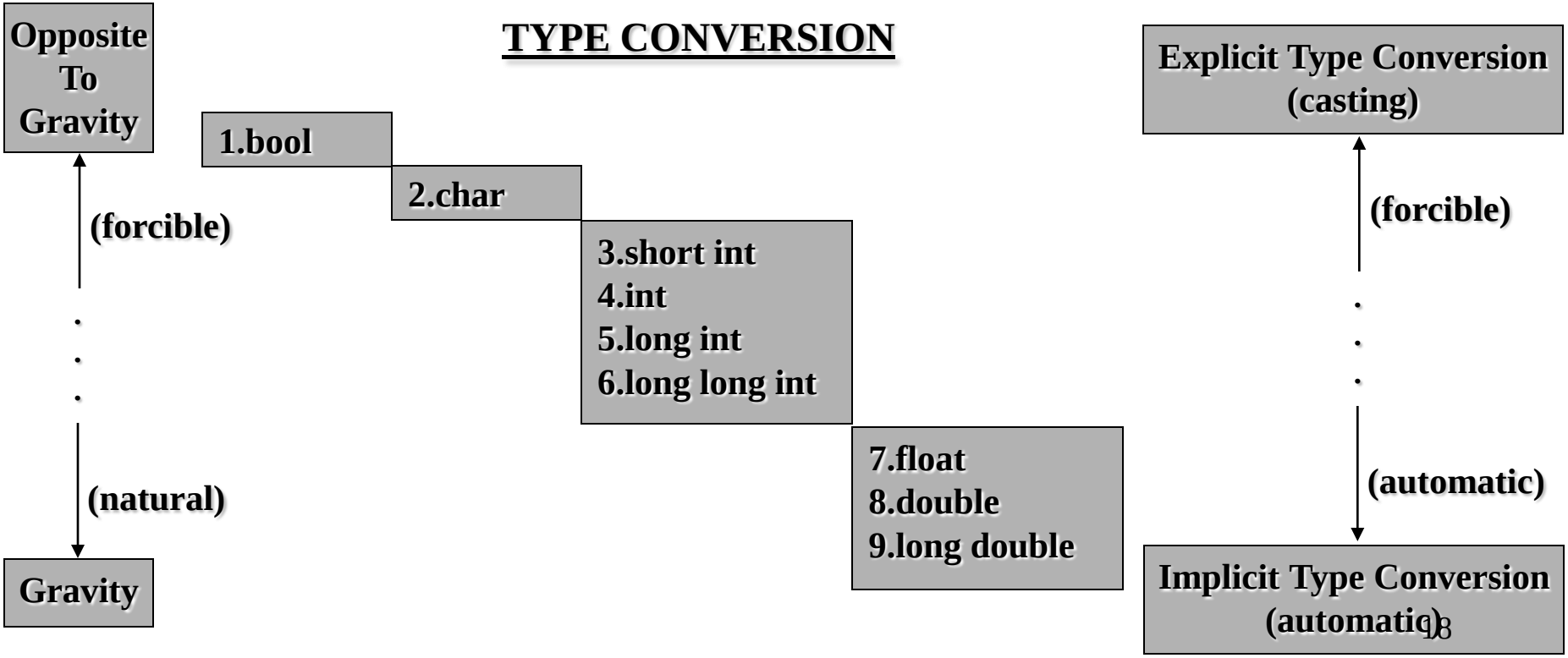
Bitwise Right Shift (>>)
(negetive values)

A (-44) : 11111111 11010100

A >> 2 : 11111111 11110101

Note : Right shift operator fills the left vacant fields with 'zeros' for positive numbers, with 'ones' for negative numbers.

TYPE CONVERSION



Precedence and Associativity of Operators

Precdence Group (Highest to Lowest)	Operators	Associativity
(param) subscript etc.,	( ) [ ] –>.	L → R
Unary operators	- + ! ~ ++ – – (type) * & sizeof	R → L
Multiplicative	* / %	L → R
Additive	+ –	L → R
Bitwise shift	<< >>	L → R
Relational	< <= > >=	L → R
Equality	= = !=	L → R
Bitwise AND	&	L → R
Bitwise exclusive OR	^	L → R
Bitwise OR	|	L → R
Logical AND	&&	L → R
Logical OR	| |	L → R
Conditional	?:	R → L
Assignment	= += –= *= /= %= &= ^= |= <<= >>=	R → L
Comma	,	L → R

Important Functions in math.h

abs(x)	**absolute value of integer x**
ceil(x)	**rounds up and returns the smallest integer greater than or equal to x**
floor(x)	**rounds down and returns the largest integer less than or equal to x**
log(x)	**returns natural logarithm**
pow(x,y)	**returns the value of x^y**
sqrt(x)	**returns square root of x**
exp(x)	**returns natural anti logarithm**
sin(x)	**returns sine value where x in radians**
cos(x)	**returns cosine value where x in radians**
tan(x)	**returns tangent values where x in radians**
fmod(x,y)	**calculate x modulo y, where x and y are double**
hypot(x,y)	**calculate hypotenuse of right angle where x,y are sides.**
log10(x)	**returns logarithm base 10**

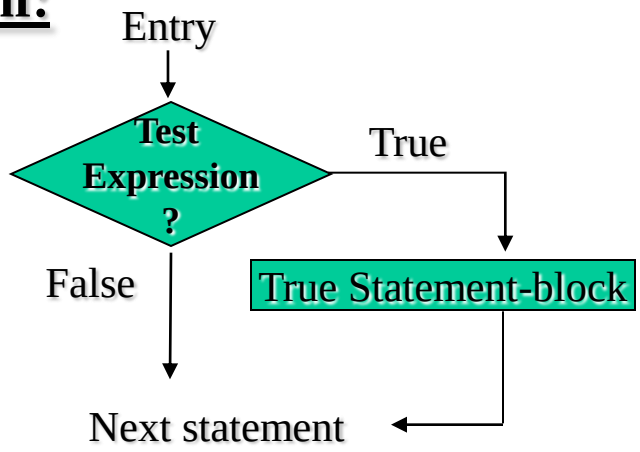
UNIT-II

CONTROL STRUCTURES, ARRAYS AND STRINGS

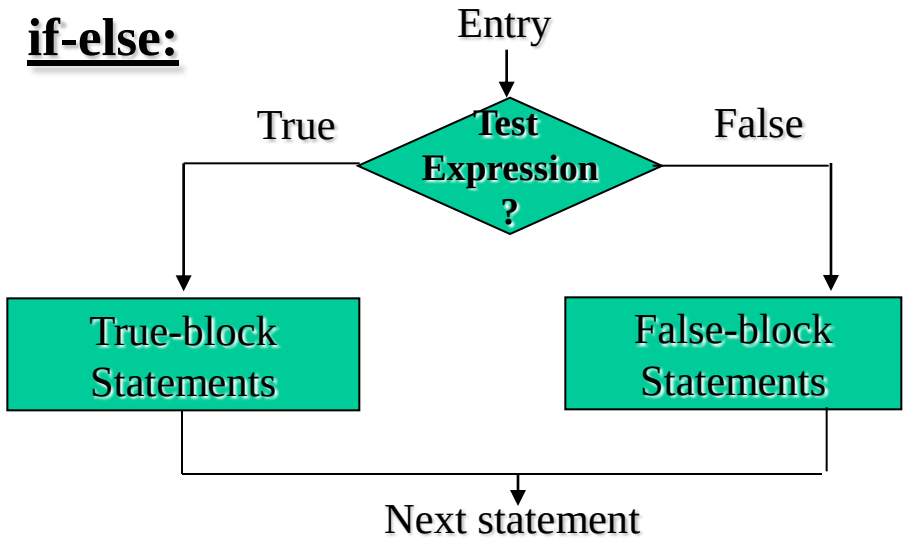
Control structures: Decision statements; if and switch statement; Loop control statements: while, for and do while loops, jump statements, break, continue, goto statements; Arrays: Concepts, one dimensional arrays, declaration and initialization of one dimensional arrays, two dimensional arrays, initialization and accessing, multi dimensional arrays; Strings concepts: String handling functions, array of strings.

Prepared by
Dr. K. Srinivasa Reddy,
HOD-IT,
Institute of Aeronautical Engineering, Hyderabad-090

simple if:



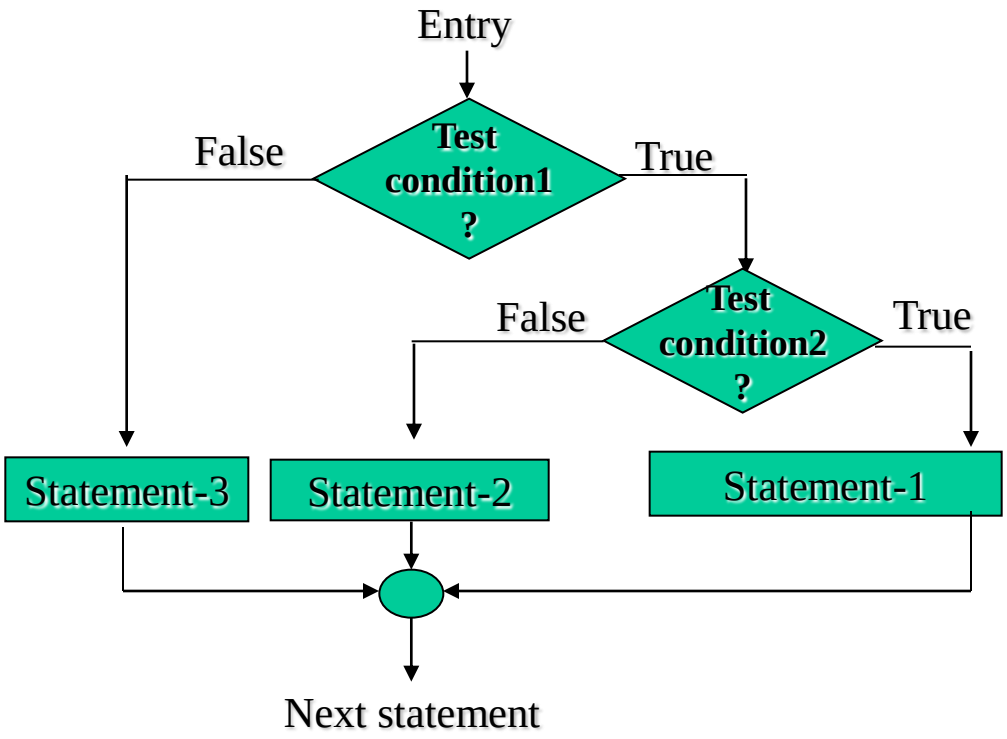
if-else:




```
/* check a citizen is eligible for voting */
#include<stdio.h>
int main()
{
int age;
printf("Enter the age : ");
scanf("%d",&age);
if(age >= 18)
printf("Eligible for voting…");
getch();
}
```

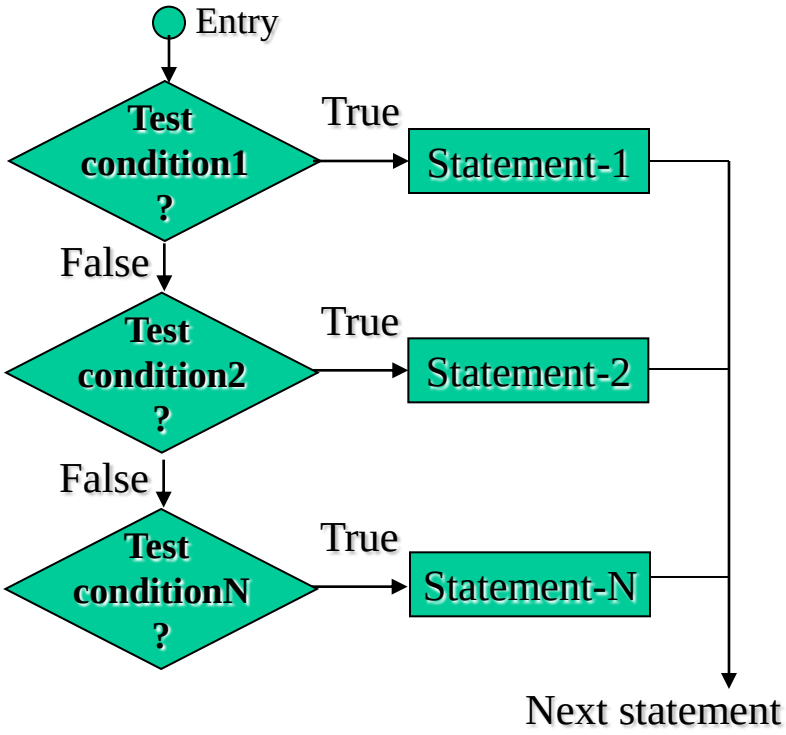
```
/* print a number is even or odd */
#include<stdio.h>
int main()
{
int number;
printf("Enter a number : ");
scanf("%d", &number);
if((number %2) == 0)
printf("%d is even number.",number);
else
printf("%d is odd number.",number);
}
```

nested if…else:



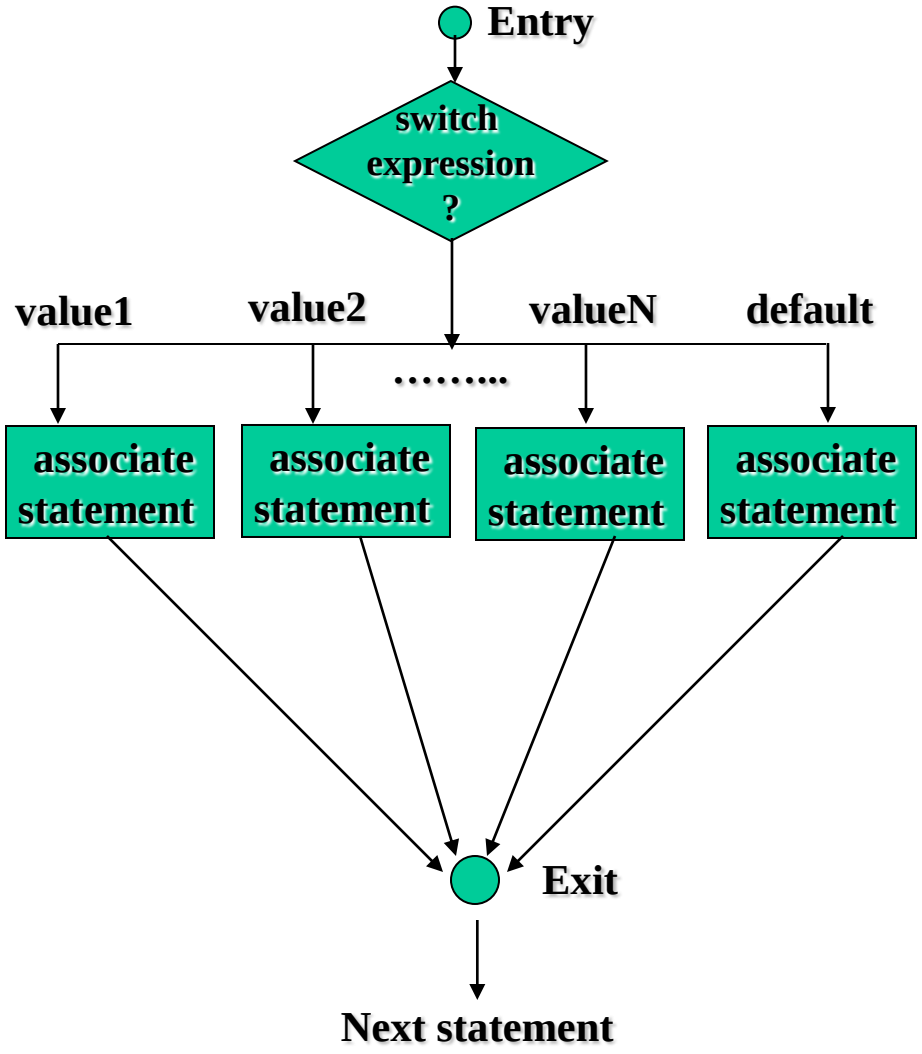

```
/* check whether a year is leap year or not */
#include<stdio.h>
int main() {
int year;
printf("Enter the year ?");
scanf("%d",&year);
if((year %100) == 0)
{
if((year % 400) == 0)
printf("%d is leap year.",year);
else
printf("%d is not leap year.",year);
} else {
if((year % 4) == 0)
printf("%d is leap year.",year);
else
printf("%d is not leap year.",year);
}
getch();
}
```

if…else…if :




```
/* program to print the grade of student */
#include<stdio.h>
int main() {
int marks;
printf("Enter marks ? ");
scanf("%d", &marks);
if(marks >= 75)
printf("Distinction");
else if(marks >= 60)
printf("First class");
else if(marks >= 50)
printf("Second class");
else if(marks >= 35)
printf("Third class");
else
printf("Failed");
}
```

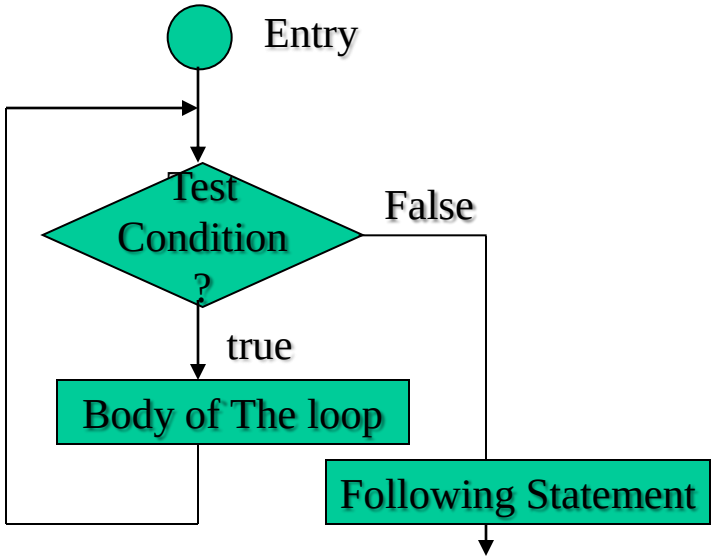
switch statement :




```
/* program to simulate a simple calculator */
#include<stdio.h>
int main() {
float a,b;
char opr;
printf("Enter number1 operator number2 : ");
scanf("%f %c %f",&a,&opr,&b);
switch(opr)
{
case '+':
printf("Sum : %f",(a + b));
break;
case '-':
printf("Difference : %f",(a - b));
break;
case '*':
printf("Product : %f",(a * b));
break;
case '/':
printf("Quotient : %f",(a / b));
break;
default:
printf("Invalid Operation!");
}
}
```

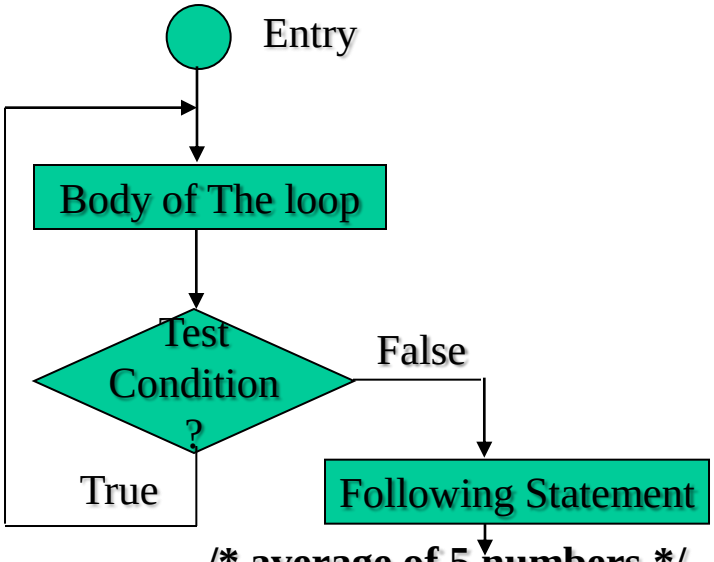
Loop Statements

while – (Entry controlled)



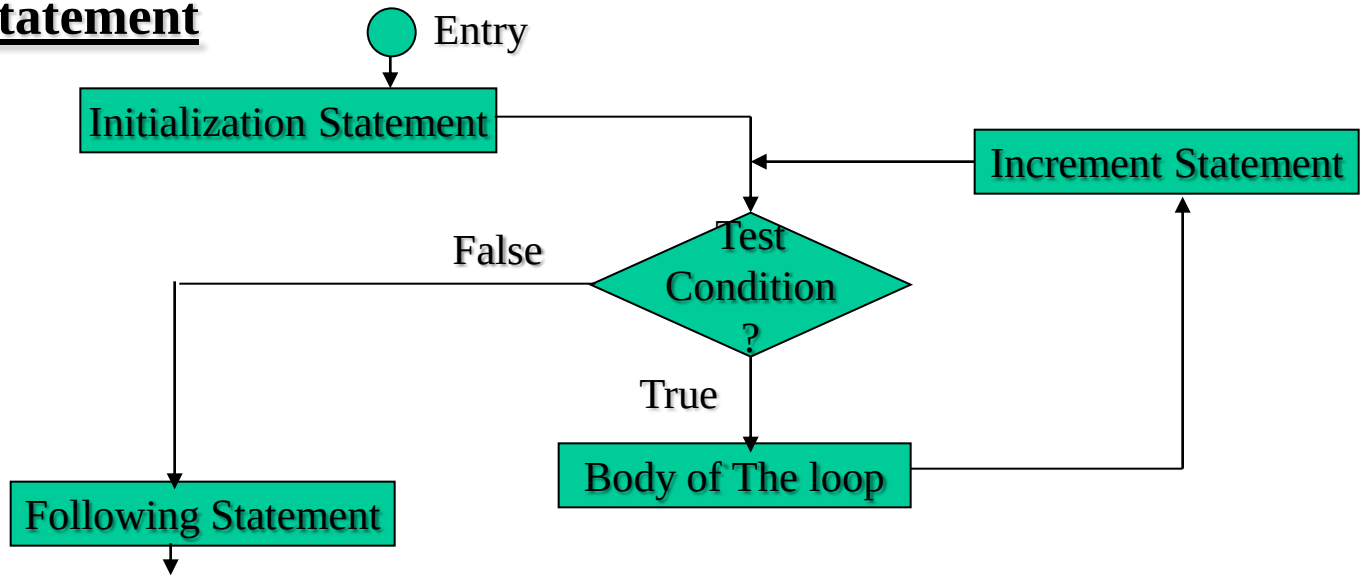

```
/* sum of 1 to 10 numbers */
#include<stdio.h>
int main() {
int i = 1,sum = 0;
while(i<=10){
sum = sum + i;
i = i + 1;
}
printf("Total : %d ",sum);
}
```

do-while – (Exit controlled)




```
/* average of 5 numbers */
#include<stdio.h>
int main() {
int count = 1;
float x,  sum = 0;
do {
printf("x = ");
scanf("%f",&x);
sum += x;
++ count;
}  while(count <= 5);
printf("Average = %f ", (sum/5))
}
```

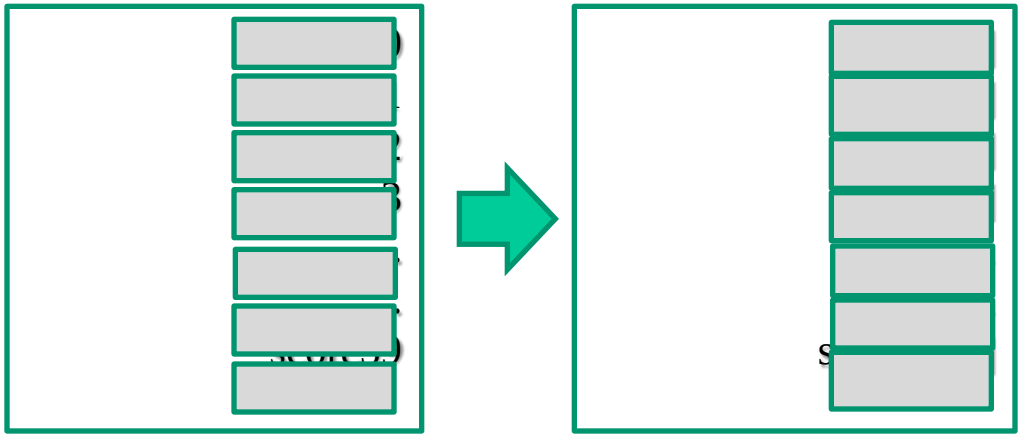
for -- Statement




```
/* check whether a number is prime or not */
#include<stdio.h>
int main() {
int n,i,factors = 0;
printf("Enter a number : ");
scanf("%d",&n);
for(i = 1; i <= n; i++) {
if((n % i)==0)  ++factors;
}
if (factors == 2)
printf("%d is prime number.",n);
else
printf("%d is not prime number.",n);
}
```

Array & its Advantage

```
/* Ranking  of 60 students in a class */
int main() {
        /*declaring 60 varialbes */
    int  score0, score1,score2,……,score59;
        /* Reading scores for sixty times */
    printf("Enter  the score : ");
    scanf("%d", &score0);
        …. …. …. ….
    printf("Enter  the score : ");
    scanf("%d", &score59);
          /* comparing & swapping for 1770 times
                * to arrange in descending order  */
                                swap( score0, score1);
                                swap( score1, score2);
                                swap( score2, score3);
                                    …. …. …. ….
                                swap( score0,score1);
                                swap( score1,score2);
                                swap( score0,score1);
                    /*printing 60 scores after sorting */
                                printf("%4d", score0);
                                printf("%4d", score1);
                                    … … … …
                                }
                    void swap ( int  a, int b)  {
                                int temp;
                                if( a < b) {
                    temp = a  ;  a = b  ;  b = temp;
                                }
                                }
```



Sixty variables are replaced by one **Array**

Sixty input statements are called by one loop statement

1770 comparing statements are included in one loop statement

Mean can be calculated only after reading all scores. Each deviation is difference of individual score and mean. To calculate deviations of all scores, scores must be stored in an **ARRAY.**

```
#include<stdio.h>
#include<math.h>
#define SIZE 10
int main() {
   int scores[SIZE],sum=0,i;
   float deviation, mean, total=0;
   float variance , stddev;
   for(i=0; i < SIZE ;i++) {
     printf("Enter score : “);
     scanf("%d",   &scores[i]     );
     sum = sum + scores[i];
   }
   mean =
   printf("\
```

Declaration of Array

Initialization of Array

Input to an element

```
   printf("\nDeviations : ");
   for(i=0;i<SIZE ; i++) {
      deviation =   scores[i]    - mean;
      printf("%.2f\t", deviation);
      total=total + deviation*deviation;
   }
   variance = total / SIZE;
   printf("\nVariance = %.2f\n", variance);
   stddev = sqrt(variance);
   printf("Standard Deviation : %f", stddev);
}
```

Accessing an element

Processing on Array

Scalar variable for single data item & Vector variable for multiple data items

Scalar Variables :

- A variable represents a data item and it can be used to store a single atomic value at a time. These are also called scalar variables.
- Integer takes 2 bytes memory as a single unit to store its value. i.e.,the value of a scalar variable cannot be subdivided into a more simpler data items.
- The address of first byte is the address of a variable .

Vector Variables (arrays):

- In contrast, an **array** is multivariable (an aggregate data type), which is also referred to a data structure. It represents a collection of related data items of same type.
- An individual data item of an array is called as 'element'. Every element is accessed by index or subscript enclosed in square brackets followed after the array name.
- All its elements are stored in consecutive memory locations, referred under a common array name.

  Ex : int marks[10] ; /* declaration of array */
- '0' is first number in computer environment. The first element of array marks is marks[0] and last element is marks[9]. (the address of first element is the address of the array)
- An array is a derived data type. It has additional operations for retrieve and update the individual values.
- The lowest address corresponds to the first element and the highest address to the last element.
- Arrays can have from one to several dimensions. The most common array is the *string*, which is simply an array of characters terminated by a null.

Declaration of One Dimensional Arrays

Syntax :

arrayType arrayName [numberOfElements];

Example :

```
int scores [60];
float salaries [20];
```

Initialization of Array while Declaration :

```
int numbers [ ] = { 9, 4, 2, 7, 3 };
char name[ ] ={'J','N','T','U',' ','H','Y','D','\0' };
char greeting[ ] = "Good Morning";
```

Declaration of Multi Dimensional Arrays

Syntax :

arrayType arrayName [Rows][Columns];

arrayType arrayName [Planes][Rows][Columns];

Example :

```
/* Each student for seven subjects */
int marks[60][7];
/* Matrix with 3 planes and 5 rows and 4 columns */
float matrix[3][5][4];
```

Initialization of Array while Declaration :

```
int matrix [ ][ ] = { { 4, 2, 7, 3 } ,
                      { 6, 1, 9, 5 } ,
                      { 8, 5, 0, 1 } };
```

Elements of
Array [3] by [4]

[0][0]	[0][1]	[0][2]	[0][3]
[1][0]	[1][1]	[1][2]	[1][3]
[2][0]	[2][1]	[2][2]	[2][3]

```
/*passing an array to function */
#define SIZE 10
int main() {
    float list[SIZE] ,avg;
        … … … … …
    avg = average(SIZE , list );
        … … … … …
}
float average( int n , float x[]) {
    float sum=0,i;
     for( i = 0; i < n ; i++)
        sum = sum + x[i];
     return ( sum / n ) ;
}
```

Strings - One Dimensional Character Arrays

A String is sequence of characters. In 'C' strings are implemented by an array of characters terminated with a null character '\0'(back slash followed by zero).

```
char city[] = "HYDERABAD";
```

'CITY' is an array of characters has size of 10 characters including a null character '\0'(ascii code is zero).

```
char name[25] ;
scanf("%s", name); /*reading a string until a white space is encountered ( & operator is not required )*/
printf("%s", name);  /*printing a string in input window */
gets(name) ; /* reading a string including white spaces until '\n' is encountered. */
puts(name); /* printing a string and moves cursor to new line */
```

String Manipulation Functions in <string.h>

strlen(s1) - returns the length of string excluding the last 'null' character.
strcpy(s1,s2) - copies characters in s2 into s1.
strcat(s1,s2)- concatenates s2 to s1.
strcmp(s1,s2) -compares s1 with s2 lexicographically and returns '0' if two strings are same , returns -1 if s1 is before s2 and returns +1 if s1 is after s2.
strcmpi(s1,s2) -compares s1 with s2 like strcmp() but case of characters is ignored.
strchr(s1,ch) -returns pointer to first occurrence of the character 'ch' in s1.
strstr(s1,s2)-returns pointer to first occurrence s2 in s1.
strrev(s1) -returns pointer to the reversed string.

Memory Address : Bit is a smallest unit of memory to store either '0' or '1' in memory. Byte is unit of memory of 8 bits. Memory is a sequence of a large number of memory locations , each of which has an address known as byte. Every byte in memory has a sequential address number to recognized by processor.

Memory Sections of C-Runtime

RAM is temporary storage place to run programs. C-Language runtime also utilizes an allotted memory block in RAM to run its programs.

Text Section : Memory-area that contains the machine instructions(code).It is read only and is shared by multiple instances of a running program.

Data Section : Memory image of a running program contains storage for initialized global variables, which is separate for each running instance of a program.

BSS (Below Stack Segment) : Memory area contains storage for uninitialized global variables. It is also separate for each running instance of a program.

Stack : Area of memory image of a running program contains storage for automatic variables of a function. It also stores memory address of the instruction which is the function call, to return the value of called function.

Heap : This memory region is reserved for dynamically allocating memory for variables at run time. **Dynamic Memory Allocation** calculate the required memory size while program is being executed.

Shared Libraries: This region contains the executable image of shared libraries being used by a program.

UNIT-III

FUNCTIONS AND POINTERS

Functions: Need for user defined functions, function declaration, function prototype, category of functions, inter function communication, function calls, parameter passing mechanisms, recursion, passing arrays to functions, passing strings to functions, storage classes, preprocessor directives.

Pointers: Pointer basics, pointer arithmetic, pointers to pointers, generic pointers, array of pointers, pointers and arrays, pointers as functions arguments, functions returning pointers.

Prepared by
Dr. K. Srinivasa Reddy,
HOD-IT,
Institute of Aeronautical Engineering, Hyderabad-090

Modularizing and Reusing of code through Functions

Calculation of area of Circle is separated into a separate module from Calculation of area of Ring and the same module can be reused for multiple times.

```
/* program to find area of a ring
*/
#include<stdio.h>
int   main()
{
   float a1,a2,a,r1,r2;
   printf("Enter the radius : ");
   scanf("%f",&r1);
   a1 = 3.14*r1*r1;
   printf("Enter the radius : ");
   scanf("%f",&r2);
   a2 = 3.14*r2*r2;
   a = a1- a2;
   printf("Area of Ring : %.3f\n",
a);
}
```

Repeated & Reusable blocks of code

```
/* program to find area of a ring */
#include<stdio.h>
float area();
int   main()
{
   float a1,a2,a;
   a1 = area();
   a2 = area();
   a = a1- a2;
   printf("Area of Ring : %.3f\n", a);
}
float area()
{
   float r;
   printf("Enter the radius : ");
   scanf("%f", &r);
   return (3.14*r*r);
}
```

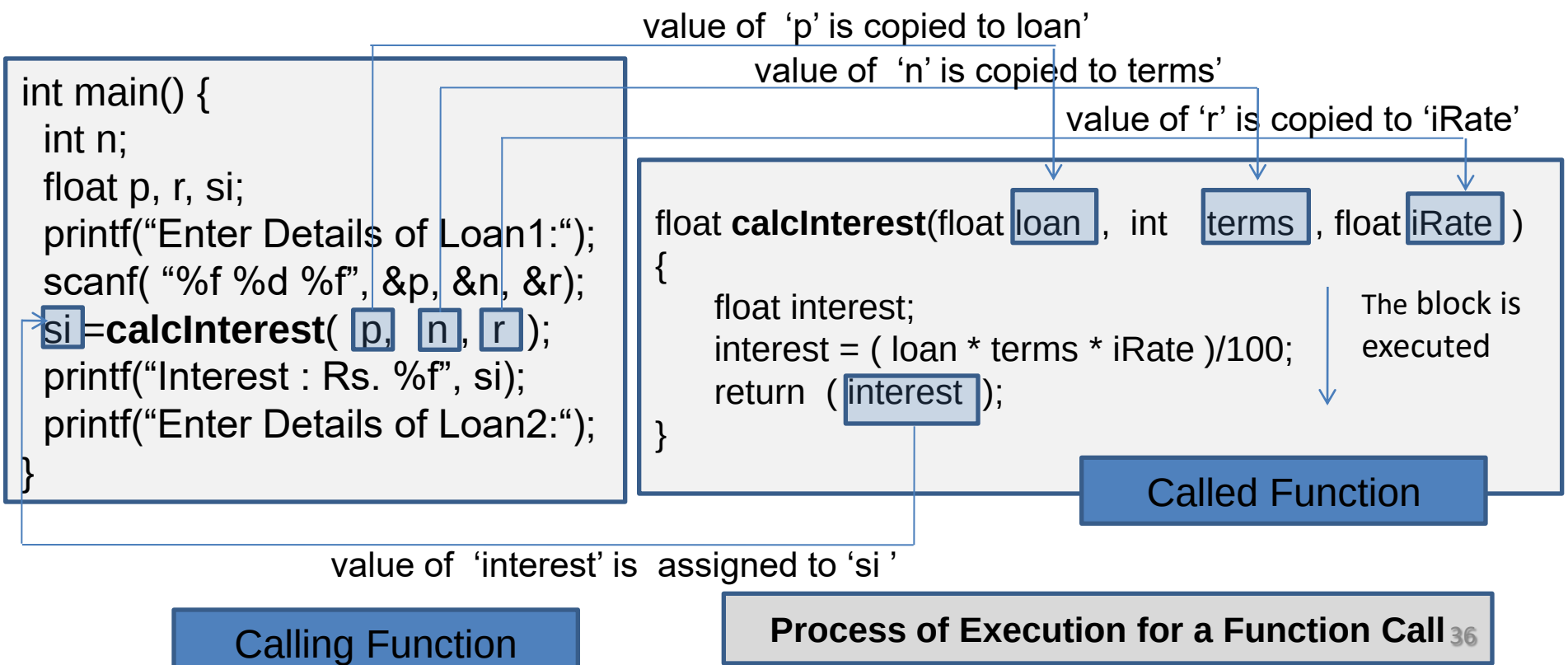
Function Declaration

Function Calls

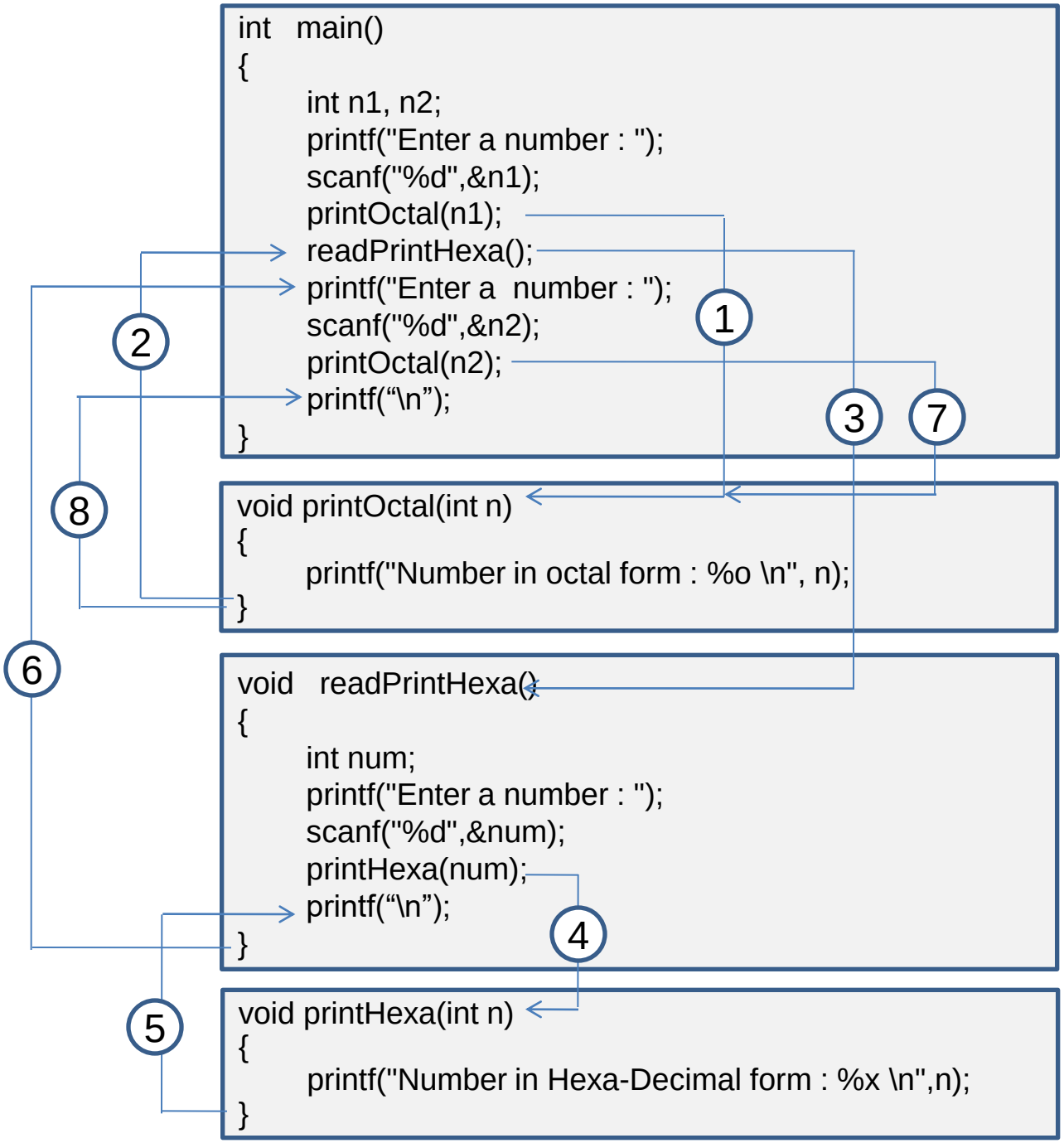
Function Definition

A **Function** is an **independent, reusable module** of statements, that specified by a name. This module (sub program) can be called by it's name to do a specific task. We can call the function, for any number of times and from anywhere in the program. The purpose of a function is to receive zero or more pieces of data, operate on them, and return at most one piece of data.

A **Called Function** receives control from a **Calling Function**. When the called function completes its task, it returns control to the calling function. It may or may not return a value to the caller. The function main() is called by the operating system; main() calls other functions. When main() is complete, control returns to the operating system.



Process of Execution for a Function Call



Flow of Control in Multi-Function Program

Function-It's Terminology

```
/* Program demonstrates function calls */
#include<stdio.h>
int add ( int n1, int n2 ) ;
int main(void)
{
    int a, b, sum;
    printf("Enter two integers : ");
    scanf("%d %d", &a, &b);

    sum = add ( a , b ) ;
    printf("%d +  %d = %d\n", a, b, sum);
    return 0;
}

/* adds two numbers and return the sum */

int  add  ( int  x ,  int  y  )
{
    int s;
    s = x + y;
    return ( s );
}
```

- Function Name
- Declaration (proto type) of Function
- Formal Parameters
- Function Call
- Actual Arguments
- Return Type
- Definition of Function
- Parameter List used in the Function
- Return statement of the Function
- Return Value

Categories of Functions

```
/* using different functions */
int main()
{
    float radius, area;
    printMyLine();
    printf("\n\tUsage of functions\n");
    printYourLine('-',35);
    radius = readRadius();
    area = calcArea ( radius );
    printf("Area of Circle = %f",
area);
}
```

```
void  printMyLine()
{
    int i;
    for(i=1; i<=35;i++) printf("%c",  '-');
    printf("\n");
}
```

Function with No parameters and No return value

```
void printYourLine(char ch, int n)
{
     int i;
    for(i=1;  i<=n ;i++) printf("%c",  ch);
    printf("\n");
}
```

Function with parameters and No return value

```
float  readRadius()
{
    float r;
    printf("Enter the radius :  ");
    scanf("%f", &r);
    return  ( r );
}
```

Function with return value & No parameters

```
float  calcArea(float r)
{
    float  a;
    a = 3.14 * r * r ;
    return  ( a ) ;
}
```

Function with return value and parameters

Note: 'void' means "Containing nothing"

```
#include<stdio.h>
float length, breadth;
int main()
{
   printf("Enter length, breadth : ");
   scanf("%f %f",&length,&breadth);
   area();
   perimeter();
   printf("\nEnter length, breadth: ");
   scanf("%f %f",&length,&breadth);
   area();
   perimeter();
}
```

External Global Variables

Scope: Visible across multiple functions Lifetime: exists till the end of the program.

```
void area()
{
   static int num = 0;
   float a;
   num++;
   a = (length * breadth);
   printf("\nArea of Rectangle %d : %.2f", num, a);
}
```

Static Local Variables

Visible with in the function, created only once when function is called at first time and exists between function calls.

```
void perimeter()
{
   int no = 0;
   float p;
   no++;
   p = 2 *(length + breadth);
   printf("Perimeter of Rectangle %d: %.2f",no,p);
}
```

```
Enter length, breadth : 6  4
Area of Rectangle 1 : 24.00
Perimeter of Rectangle 1 : 20.00
Enter length, breadth : 8  5
Area of Rectangle 2 : 40.00
Perimeter of Rectangle 1 : 26.00
```

Automatic Local Variables

Scope : visible with in the function.
Lifetime: re-created for every function call and destroyed automatically when function is exited.

File1.c

```
#include<stdio.h>
float length, breadth;

static float base, height;
int main()
{
   float peri;
   printf("Enter length, breadth : ");
   scanf("%f %f",&length,&breadth);
   rectangleArea();
   peri = rectanglePerimeter();
   printf("Perimeter of Rectangle : %f", peri);
   printf("\nEnter base , height: ");
   scanf("%f %f",&base,&height);
   triangleArea();
}
void rectangleArea() {
  float a;
  a = length * breadth;
  printf("\nArea of Rectangle : %.2f",  a);
}
void triangleArea() {
  float a;
  a = 0.5 * base * height ;
  printf("\nArea of Triangle : %.2f",  a);
}
```

File2.c

```
extern float length, breadth ;
/*  extern base , height ;  --- error  */
float rectanglePerimeter()
{
   float p;
   p = 2 *(length + breadth);
   return ( p );
}
```

External Global Variables

Scope: Visible to all functions across all files in the project.

Lifetime: exists till the end of the program.

Static Global Variables

Scope: Visible to all functions with in the file only.

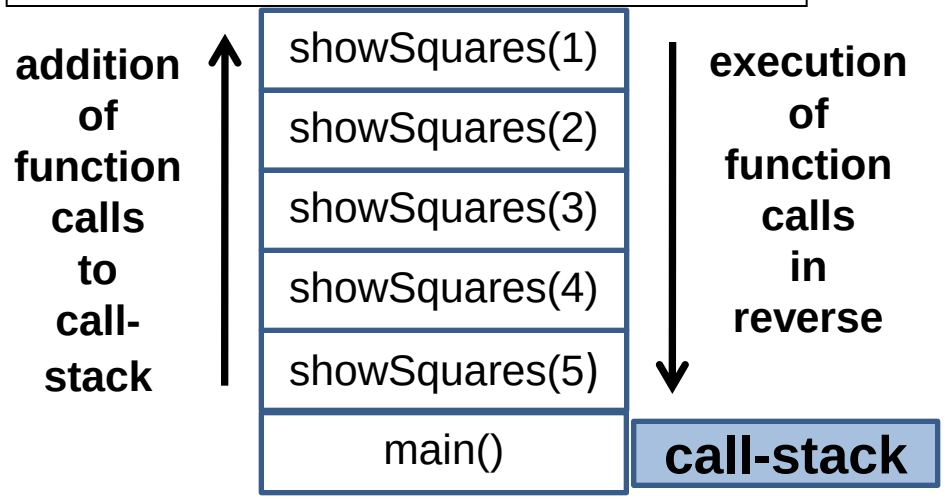
Lifetime: exists till the end of the program.

Storage Classes – Scope & Lifetime

```
#include<stdio.h>
void showSquares(int n)
{
    if(n == 0)
      return;
    else
      showSquares(n-1);
    printf("%d  ", (n*n));
}
int main()
{
   showSquares(5);
}
```

A function calling itself is Recursion

Output : 1 4 9 16 25



Preprocessor Directives

#define	- Define a macro substitution
#undef	- Undefines a macro
#ifdef	- Test for a macro definition
#ifndef	- Tests whether a macro is not defined
#include	- Specifies the files to be included
#if	- Test a compile-time condition
#else	- Specifies alternatives when **#if** test fails
#elif	- Provides alternative test facility
#endif	- Specifies the end of **#if**
#pragma	- Specifies certain instructions
#error	- Stops compilation when an error occurs
#	- Stringizing operator
##	- Token-pasting operator

Preprocessor is a program that processes the source code before it passes through the compiler.

Memory Address : Bit is a smallest unit of memory to store either '0' or '1' in memory. Byte is unit of memory of 8 bits. Memory is a sequence of a large number of memory locations , each of which has an address known as byte. Every byte in memory has a sequential address number to recognized by processor.

Memory Sections of C-Runtime

RAM is temporary storage place to run programs. C-Language runtime also utilizes an allotted memory block in RAM to run its programs.

Text Section : Memory-area that contains the machine instructions(code).It is read only and is shared by multiple instances of a running program.

Data Section : Memory image of a running program contains storage for initialized global variables, which is separate for each running instance of a program.

BSS (Below Stack Segment) : Memory area contains storage for uninitialized global variables. It is also separate for each running instance of a program.

Stack : Area of memory image of a running program contains storage for automatic variables of a function. It also stores memory address of the instruction which is the function call, to return the value of called function.

Heap : This memory region is reserved for dynamically allocating memory for variables at run time. **Dynamic Memory Allocation** calculate the required memory size while program is being executed.

Shared Libraries: This region contains the executable image of shared libraries being used by a program.

Two or more Permanent Manipulations using one Function

Passing Parameters By Value

```
/* program to swap two numbers */
#include<stdio.h>
void swap(int x, int y)
{
     int temp;
     temp = x; x = y; y = temp;
     printf("\nIn swap()  :  %d  %d ",x,y);
}
int main()
{
   int  a = 25,b = 37;
   printf("Before swap() : %d %d",a,b);
   swap (a,b);
   printf("\nAfter swap() : %d %d",a,b);
}
```

```
Output :
Before swap()  25 37
In swap ()        37 25
After swap()     25 37
```

Passing Parameters By Reference

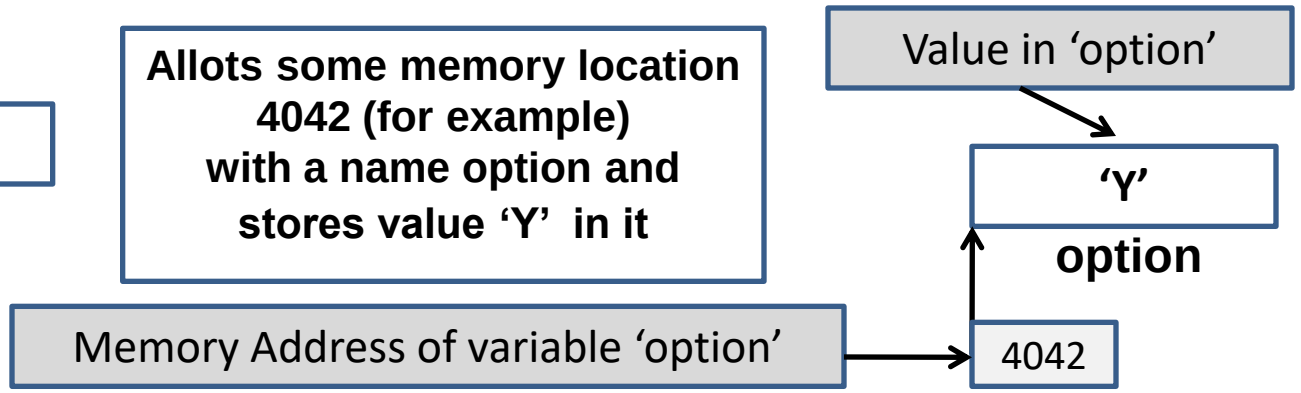
```
/* program to swap two numbers */
#include<stdio.h>
void swap(int *x, int *y)
{
     int temp;
     temp = *x; *x = *y; *y = temp;
     printf("\nIn swap()  :  %d  %d ",*x,*y);
}
int main()
{
   int  a = 25,b = 37;
   printf("Before swap() : %d %d",a,b);
   swap (&a , &b);
   printf("\nAfter swap() : %d %d",a,b);
}
```

```
Output :
Before swap()  25 37
In swap ()        37 25
After swap()     37 25
```

Pointer variable – A variable holds the address of another variable

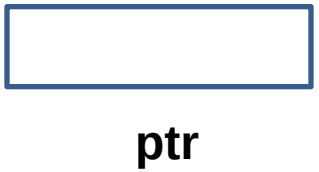
char option = 'Y';

Allots some memory location 4042 (for example) with a name option and stores value 'Y' in it



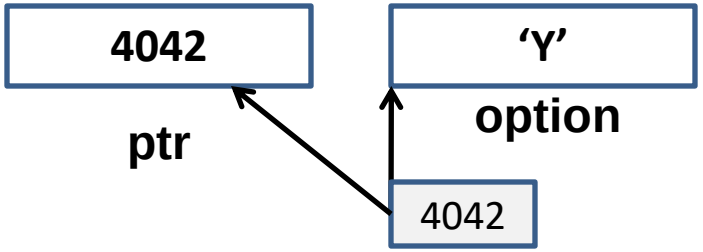
char *ptr = NULL;

Creates a pointer variable with a name 'ptr' Which can hold a Memory address



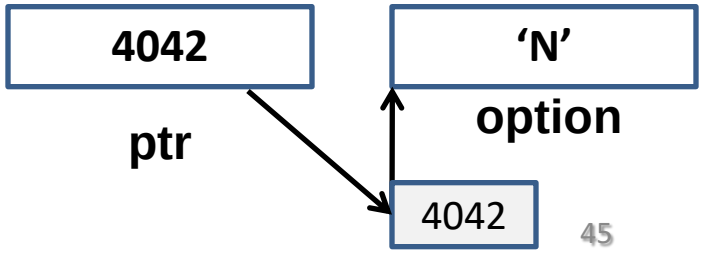
ptr = &option;

Memory address of Variable 'option' Is copied to the Pointer 'ptr'



***ptr = 'N';**

The value 'N' is stored in the variable which has the memory address 4042



Program with Using Pointers

```
int main() {
    int  n1, n2 ;
    int   *p = NULL, *q = NULL;
    n1  =  6 ;
    p    =  & n1;
    printf ("%d  %d", n1,*p  );

    printf ("%ld  %ld",&n1, p );

    q    =  & n2;
    *q   =  3 ;
    printf (" %d  %d ", *p , *q ) ;

    p   =  q ;
    printf (" %d  %d ", *p , *q ) ;


    *p  = 7 ;
    printf (" %d  %d ", *p , *q ) ;
}
```

pointer variables are declared

Prints 6 6

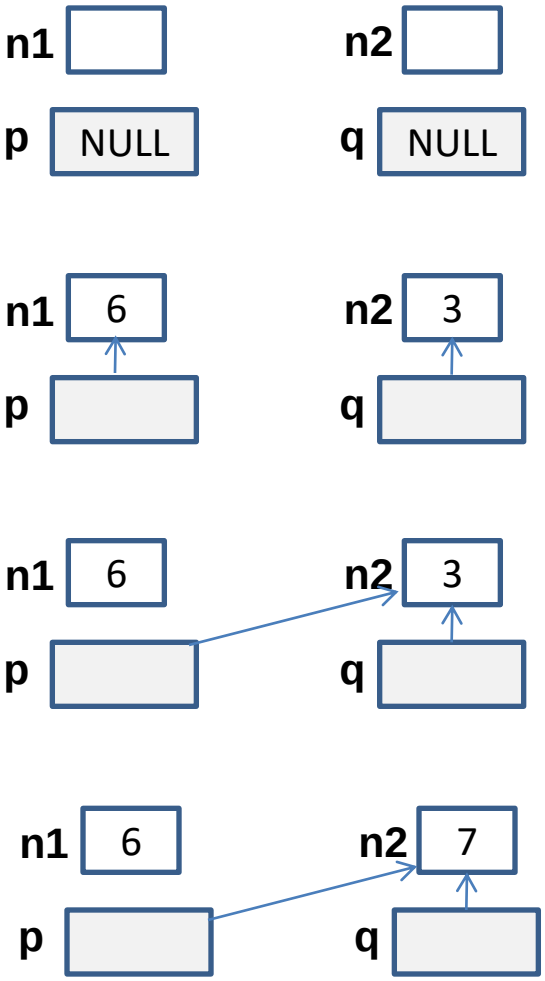
Prints address of n1

Prints 6 3

pointer 'q' assigned with pointer 'q'

Prints 3 3

Prints 7 7



When two pointers are referencing with one variable, both pointers contains address of the same variable, and the value changed through with one pointer will reflect to both of them.

Pointer and Arrays

Even though pointers and arrays work alike and strongly related, they are not synonymous. When an array is assigned with pointer, the address of first element of the array is copied into the pointer.

```
#include<stdio.h>
int main()
 {
    int  a[3] = { 12, 5 ,7}, b[3];
    int *p ,*q;

    p = a;
    printf("%d %d\n", *p, *a);

    q = p;
    printf("%d %d",*p,*q);

    b = a;  /*  error  */
}
```

Prints 12 12

Prints 12 12

Pointer is an address variable, having no initialized value by default. The address stored in the pointer can be changed time to time in the program.

Array name is an address constant, initialized with the address of the first element (base address)in the array. The address stored in array name cannot be changed in the program.

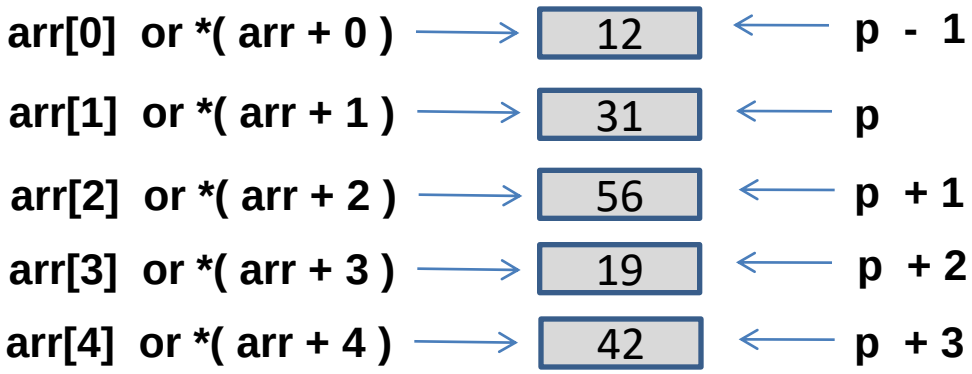
Pointer Arithmetic and Arrays

```
#include <stdio.h>
int main() {
    int arr [5]  = {  12, 31, 56, 19, 42 };
    int *p;
    p = arr + 1;
    printf("%d \n", *p);
    printf("%d %d %d\n", *(p-1), *(p), *(p + 1));
    --p;
    printf("%d", *p);
```

Prints 31

Prints 12 31 56

Prints 12



Subscript operator [ ] used to access an element of array implements address arithmetic, like pointer.

Array of Pointers

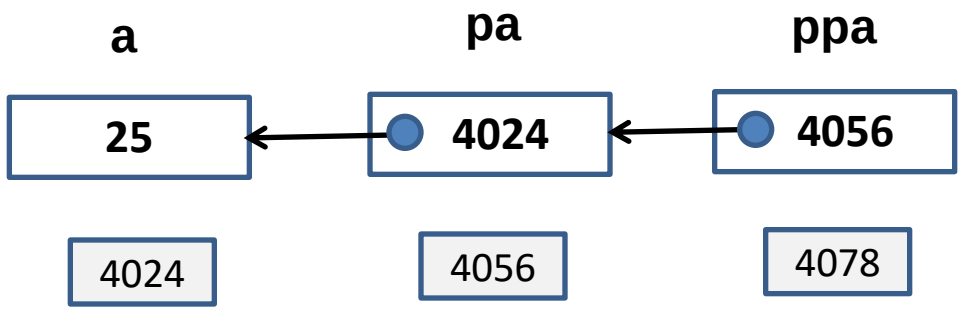
The advantage of pointer array is that the length of each row in the array may be different. The important application of pointer array is to store character strings of different length. Example :

```
char  *day[ ] = { "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday",
                  "Friday", "Saturday" };
```

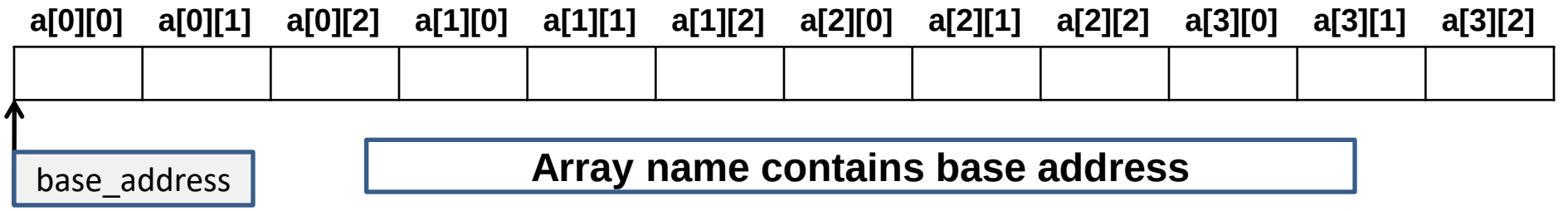
Pointer to Pointer (Double indirection)

Example :

```
int   a = 25;
int  *pa = &a;
int  **ppa ;
*ppa  =  &pa;
printf("%d", *pa); → prints  25
printf("%d", **ppa); → prints 25
```



Two Dimensional Array -- Pointers



Array name contains base address

Address of a[i] [j] = *(* (base_address + i) + j) = * (* (a + i) + j)

void Pointer

'void' type pointer is a generic pointer, which can be assigned to any data type without cast during compilation or runtime. 'void' pointer cannot be dereferenced unless it is cast.

```
int main( )  {
    void*  p;
    int x = 7;
    float y = 23.5;
    p = &x;
    printf("x contains : %d\n", *( ( int *)p) );
    p = &y;
    printf("y contains : %f\n", *( ( float *)p) );
}
```

Output :
x contains 7
y contains 23.500000

Function Pointers

Function pointers are pointers, which point to the address of a function.
Declaration :
<return type> (* function_pointer)
(type1 arg1, type2 arg2, …….);

```
int add ( int a, int b ) { return (a + b) ; }
int sub ( int a, int b ) { return (a – b) ; }

int (*fp ) (int, int ) ; /* function pointer */

int main( )  {
    fp = add;
    printf("Sum = %d\n", fp( 4, 5 )   ) ;
    fp = sub;
    printf("Difference = %d\n", fp( 6 , 2 )   ) ;
}
```

Output :
Sum = 9
Difference = 4

UNIT-IV

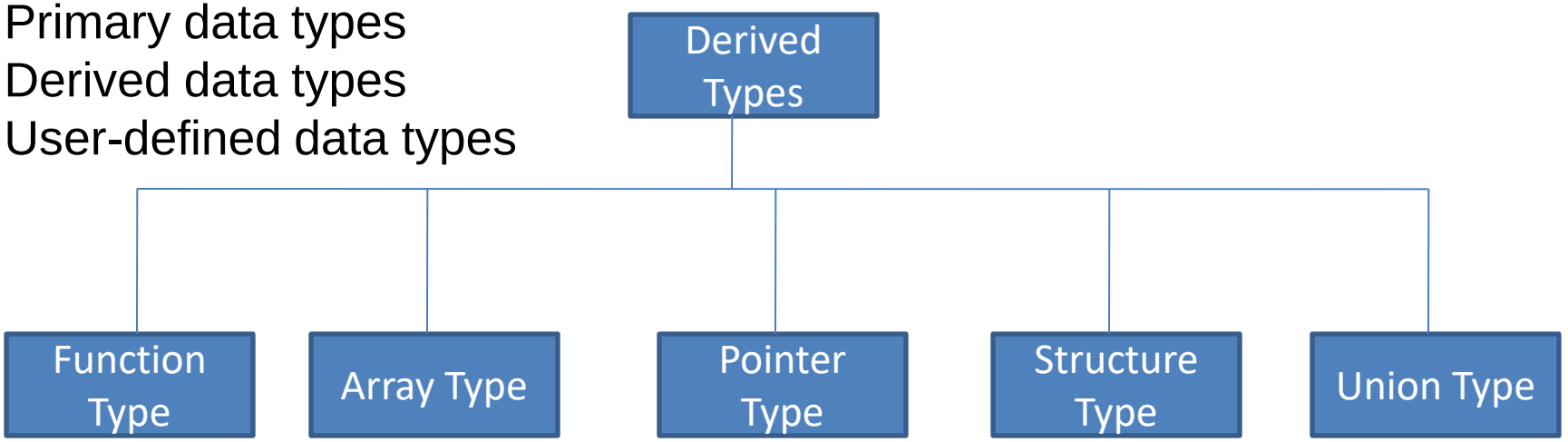
STRUCTURES AND UNIONS

Structures and unions: Structure definition, initialization, accessing structures, nested structures, arrays of structures, structures and functions, passing structures through pointers, self referential structures, unions, bit fields, typedef, enumerations; Dynamic memory allocation: Basic concepts, library functions.

Prepared by
Dr. K. Srinivasa Reddy,
HOD-IT,
Institute of Aeronautical Engineering, Hyderabad-090

C Data Types:

- Primary data types
- Derived data types
- User-defined data types



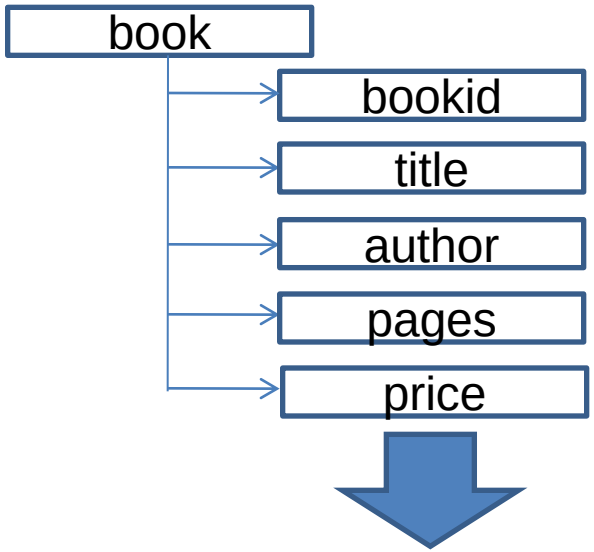
Array – Collection of one or more related variables of similar data type grouped under a single name
Structure – Collection of one or more related variables of different data types, grouped under a single name

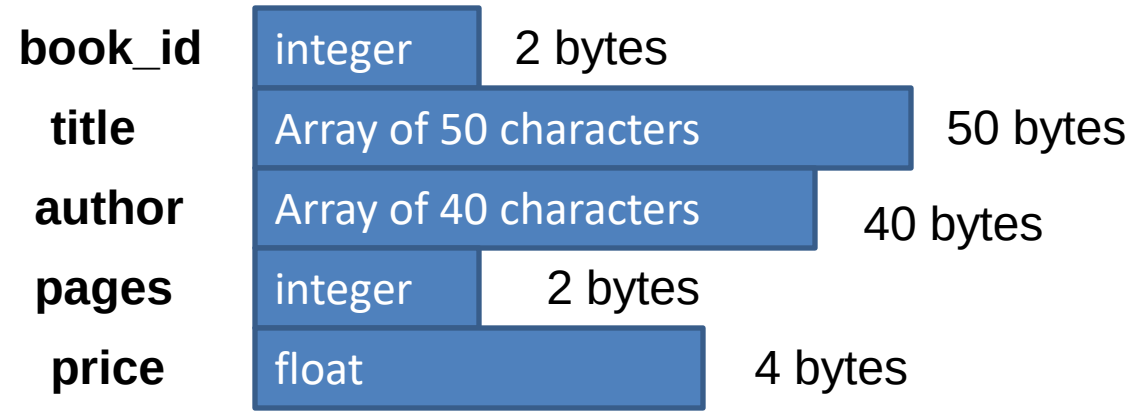
In a Library, each book is an **object**, and its **characteristics** like title, author, no of pages, price are grouped and represented by one **record**.

The characteristics are different types and grouped under a aggregate variable of different types.

A **record** is group of **fields** and each field represents one characteristic. In C, a record is implemented with a derived data type called **structure**. The characteristics of record are called the **members** of the structure.

Book-1	Book-2	Book-3
BookID: 1211	BookID: 1212	BookID: 1213
Title : C Primer Plus	Title : The ANSI C Programming	Title : C By Example
Author : Stephen Prata	Author : Dennis Ritchie	Author : Greg Perry
Pages : 984	Pages : 214	Pages : 498
Price : Rs. 585.00	Price : Rs. 125.00	Price : Rs. 305.00





Memory occupied by a Structure variable

STRUCTURE- BOOK

```
struct   book {
        int  book_id ;
        char title[50] ;
        char author[40] ;
        int pages ;
        float price ;
};
```

Structure tag

```
struct   < structure_tag_name >
 {
        data type < member 1 >
        data type < member 2 >
             ....  ....  ....  ....
        data type < member N >
} ;
```

Declaring a Structure Type

```
struct student
{
        int roll_no;
        char name[30];
        float percentage;
};
```

Declaring a Structure Variable

```
struct student s1,s2,s3;
        (or)
struct student
{
        int roll_no;
        char name[30];
        float percentage;
}s1,s2,s3;
```

Initialization of structure

Initialization of structure variable while declaration :

```
struct  student  s2 =  {  1001, " K.Avinash ",
                    87.25 } ;
```

Initialization of structure members individually :

```
s1. roll_no = 1111;
strcpy ( s1. name , " B. Kishore " ) ;
s1.percentage = 78.5 ;
```

membership operator

Reading values to members at runtime:

```
struct student s3;
printf("\nEnter the roll no");
scanf("%d",&s3.roll_no);
printf("\nEnter the name");
scanf("%s",s3.name);
printf("\nEnter the percentage");
scanf("%f",&s3.percentage);
```

Implementing a Structure

```
struct employee {
        int empid;
        char name[35];
        int age;
        float salary;
};
int main()  {
      struct employee emp1,emp2 ;

      struct employee emp3 = { 1213 , " S.Murali " , 31 , 32000.00 } ;
      emp1.empid=1211;
      strcpy(emp1.name, "K.Ravi");
      emp1.age = 27;
      emp1.salary=30000.00;
      printf("Enter the details of employee 2");
      scanf("%d %s %d %f " , &emp2.empid, emp2.name, &emp2.age, &emp2.salary);
      if(emp1.age > emp2.age)
            printf( " Employee1 is senior than Employee2\n" );
      else
            printf("Employee1 is junior than Employee2\n");

      printf("Emp ID:%d\n  Name:%s\n  Age:%d\n    Salary:%f",
              emp1.empid,emp1.name,emp1.age,emp1.salary);
}
```

Declaration of Structure Type

Declaration of Structure variables

Declaration and initialization of Structure variable

Initialization of Structure members individually

Reading values to members of Structure

Accessing members of Structure

Arrays And structures

```
struct student
{
   int sub[3] ;
   int total ;
} ;


int  main( )  {
  struct student s[3];
  int i,j;
   for(i=0;i<3;i++) {
      printf("\n\nEnter student %d marks:",i+1);
      for(j=0;j<3;j++) {
          scanf("%d",&s[i].sub[j]);
       }
    }
    for(i=0;i<3;i++) {
         s[i].total =0;
         for(j=0;j<3;j++) {
               s[i].total +=s[i].sub[j];
       }
       printf("\nTotal marks of student %d is: %d",
                          i+1,s[i].total );
    }
}
```

Nesting of structures

```
struct date {
   int day ;
   int month ;
   int year ;
} ;
struct  person {
   char  name[40];
   int  age ;
   struct  date  b_day ;
};
int  main( )  {
  struct  person  p1;
  strcpy ( p1.name , "S. Ramesh " ) ;
  p1. age =  32 ;
  p1.b_day.day  = 25 ;
  p1.b_day. month = 8 ;
  p1.b_day. year = 1978 ;
}
```

Outer Structure

Inner Structure

Accessing Inner Structure members

```
OUTPUT:
 Enter student 1 marks: 60 60 60
 Enter student 2 marks: 70 70 70
 Enter student 3 marks: 90 90 90

Total marks of  student 1 is: 180
Total marks of  student 2 is: 240
Total marks of  student 3 is: 270
```

structures and functions

```
struct   fraction  {
      int numerator ;
      int denominator ;
};

void  show ( struct fraction f  )
{
    printf ( " %d / %d ", f.numerator,
                        f.denominator ) ;
}

int  main ( )  {
      struct fraction f1  =  {  7, 12 } ;
      show  ( f1 )  ;
}
```

OUTPUT:

7 / 12

Self referential structures

```
struct student_node   {
    int roll_no ;
    char  name [25] ;
    struct student_node  *next ;
} ;
int  main( )
 {
   struct  student_node  s1 ;
   struct  student_node  s2 = { 1111, "B.Mahesh", NULL } ;
   s1. roll_no  = 1234 ;
   strcpy ( s1.name , "P.Kiran " ) ;

    s1. next  =  & s2 ;

    printf ( " %s ", s1. name  ) ;

    printf ( " %s " , s1.next - > name  ) ;
}
```

s2 node is linked to s1 node

Prints P.Kiran

Prints B.Mahesh

A self referential structure is one that includes at least one member which is a pointer to the same structure type.

With self referential structures, we can create very useful data structures such as linked -lists, trees and graphs.

Pointer to a structure

```
struct product
{
   int prodid;
   char name[20];
};
int main()
{
   struct product inventory[3];
   struct product  *ptr;
   printf("Read Product Details : \n");
   for(ptr = inventory;ptr<inventory +3;ptr++) {
      scanf("%d %s", &ptr->prodid, ptr->name);
   }
   printf("\noutput\n");
   for(ptr=inventory;ptr<inventory+3;ptr++)
   {
      printf("\n\nProduct ID :%5d",ptr->prodid);
      printf("\nName:  %s",ptr->name);
   }
}
```

Accessing structure members through pointer :

i) Using . (dot) operator :

```
( *ptr ) . prodid  =  111 ;
strcpy ( ( *ptr ) . Name, "Pen") ;
```

ii) Using - > (arrow) operator :

```
ptr - > prodid =  111 ;
strcpy( ptr - > name , "Pencil") ;
```

```
Read Product Details :

111 Pen
112 Pencil
113 Book

Print Product Details :

Product ID : 111
Name : Pen
Product ID : 112
Name : Pencil
Product ID : 113
Name : Book
```

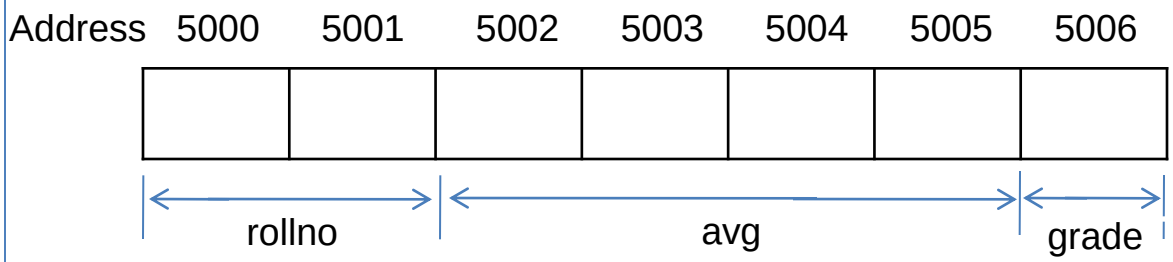
A union is a structure all of whose members share the same memory

Union is a variable, which is similar to the **structure** and contains number of members like structure.

In the structure each member has its own memory location whereas, members of union share the same memory. The amount of storage allocated to a union is sufficient to hold its largest member.

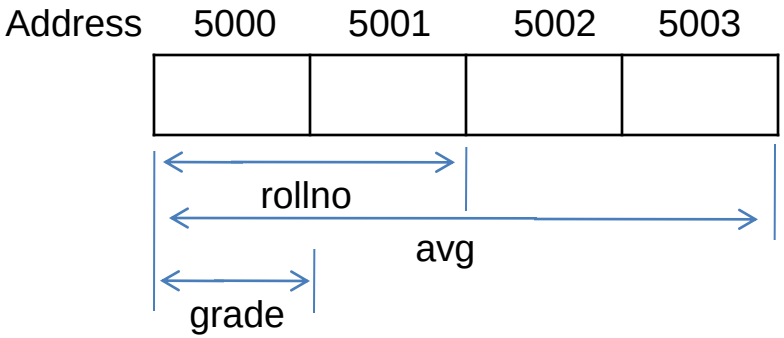
```
struct  student  {
    int  rollno;
    float  avg ;
    char  grade ;
};
union  pupil {
    int  rollno;
    float  avg ;
    char  grade;
} ;
int main()   {
    struct  student  s1 ;
    union pupil  p1;
    printf ( " %d bytes ",
         sizeof ( struct student ) ) ;
     printf ( " %d bytes ",
          sizeof ( union pupil ) ) ;
}
Output :
   7 bytes   4 bytes
```

Memory allotted to structure student



Total memory occupied : 7 bytes

Memory allotted to union pupil



Total memory occupied : 4 bytes

Dynamic Memory Allocation (DMA) of pointers

Static memory allocation means allocating memory by compiler. When using address operator, the address of a variable is assigned to a pointer. Ex : int a = 20 ; int *p = &a ;

Dynamic memory allocation means allocating memory using functions like malloc() and calloc(). The values returned by these functions are assigned to pointer variables only after execution of these functions. Memory is assigned at run time.

```
int main()
{
    int *p, *q ;
    p = (int  *) malloc ( sizeof( int ) );
    if( p == NULL )
    {
       printf("Out of memory\n");
       exit(-1);
    }
    printf("Address  in  p : %d", p );

    free ( p );
    p = NULL;
}
```

Allocates memory in bytes and returns the address of first byte to the pointer variable

Releases previously allocated memory space.

calloc ( ) is used for allocating memory space during the program execution for derived data types such as arrays, structures etc.,

Example :

```
struct book {
   int no ;  char name[20] ; float price ;
};
struct  book  b1 ;
 b1  *ptr ;
 ptr  = (book *) calloc ( 10, sizeof ( book ) );
```

ptr = (book *) realloc (ptr , 35 * sizeof (book));
Modifies the size of previously allocated memory to new size.

typedef – to define new datatype

‘ **typedef** ’ is a keyword,which allows you to specify a new name for a datatype which is already defined in c language program.

Syntax:

```
    typedef <datatype> <newname>
```

```
  /* Re-defining int type as Integer type */
typedef int Integer;
int main( ) {
    Interger a ,b , sub;
    a = 20,b = 10;
    sub = a - b;
    printf("%d - %d = %d", a, b, sub);
}
 /* Defining structure with typedef to avoid
   repeated usage of  struct keyword */

typedef struct  {
  int hours;
  int minutes;
} TIME ;
int main( ) {
   TIME t1, t2 , *t;
   t = (TIME *) calloc (10, sizeof( TIME ));
}
```

bitfieds

```
struct  playcard  {
  unsigned pips ;
  unsigned suit ;
};
```

Above structure occupies 4 bytes of memory. But the member pips accepts a value between 1 to 13 and the member suit accepts any value of 0, 1, 2 and 3 .

So we can create a more packed representation of above structure with bitfields.

```
struct  playcard {
  unsigned   pips : 4;
  unsigned   suit : 2;
};
```

A bitfield is a set of adjacent bits within a single machine word.

4-bit field called pips that is capable of storing the 16 values 0 to 15, and a 2-bit field called suit that is capable of storing values 0, 1, 2, and 3. So the entire structure variable occupies only one byte.

Note : arrays of bit fields and a pointer to address a bit field is not permitted.

Enumeration – a set of named integers, makes program more readable

Declaration of enumeration :

enum <enum_name> { member1, member2, …. …. …. } ;

Example :

enum option { YES, NO, CANCEL } ;

By default YES has value 0, NO has value 1 and CANCEL has 2.

enum direction { EAST = 1, SOUTH, WEST = 6, NORTH } ;

Now EAST has value 1, SOUTH has value 2, WEST has value 6, and NORTH has value 7.

Enumerated types can be converted implicitly or cast explicitly.

int x = WEST ; /* Valid. x contains 6. */

enum direction y ; y = (enum direction) 2 ; /* Valid. Y contains SOUTH */

```
#include<stdio.h>
int main()  {
    int signal;
    printf ("\t\t\t MENU \n\t1.RED \n");
    printf ("\t2.ORANGE\n\t3.GREEN \n“ );
    printf ("\n\t Enter the signal : “ );
    scanf (“%d”, &signal );
    switch(signal)
    {
       case 1:
         printf(“\t Stop and Wait!"); break;
       case 2:
         printf(“\t Ready to start!"); break;
       case 3:
         printf(“\t Start and go!");  break;
    }
}
```

```
#include<stdio.h>
enum color {RED = 1,ORANGE,GREEN };
int main()  {
    enum color signal;
    printf ("\t\t\t MENU \n\t1.RED \n");
    printf ("\t2.ORANGE\n\t3.GREEN\n");
    printf ("\n\t Enter the signal : ");
    scanf ("%d", &signal);
    switch(signal) {
       case RED:
         printf("\t Stop and Wait!"); break;
       case ORANGE:
         printf("\t Ready to start!"); break;
       case GREEN:
         printf("\t Start and go!");    break;
    }
}
```

Standard C-Library Functions

<stdlib.h>	
int atoi(s)	Converts string s to an integer
long atol(s)	Converts string s to a long integer.
float atof(s)	Converts string s to a double-precision quantity.
void* calloc(u1,u2)	Allocate memory to an array u1, each of length u2 bytes.
void exit(u)	Closes all files and buffers, and terminate the program.
void free (p)	Free block of memory.
void* malloc (u)	Allocate u bytes of memory.
int rand(void)	Return a random positive integer.
void* realloc(p,u)	Allocate u bytes of new memory to the pointer variable p.
void srand(u)	Initialize the random number generator.
void systerm(s)	Pass command string to the operating system.
<time.h>	
clock_t clock()	Returns clock ticks since program starts.
char *asctime(stuct tm)	Converts date and time into ascii.
int stime(time_t *tp)	Sets time.
time_t time(time_t *timer)	Gets time of day.
double difftime(t1,t2)	Returns difference time between two times t1 and t2.

UNIT-V

FILES

Files: Streams, basic file operations, file types, file opening modes, file input and output functions, file status functions, file positioning functions, command line arguments.

Prepared by
Dr. K. Srinivasa Reddy,
HOD-IT,
Institute of Aeronautical Engineering, Hyderabad-090

Console I / O Vs File I / O

- **scanf( ) and printf( ) functions read and write data which always uses the terminal (keyboard and screen) as the target.**
- **It becomes confusing and time consuming to use large volumes of data through terminals.**
- **The entire data is lost when either program terminates or computer is turned off.**
- **Some times it may be necessary to store data in a manner that can be later retrieved and processed.**

  This leads to employ the concept of FILES to store data permanently in the system.

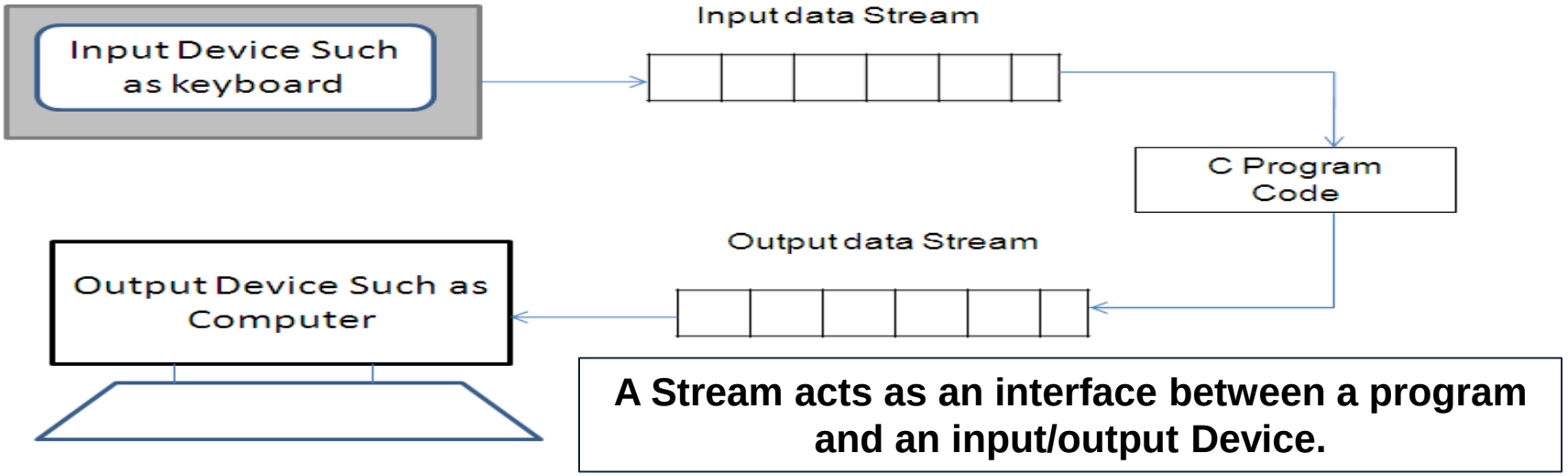
Record is logical group of data fields that comprise a single row of information, which describes the characteristics of an object.

File is a set of records that can be accessed through the set of library functions.

A **File** is a place on disk where a group of related data (records) can be stored

File Operations

1. **Creating a new file**
2. **Opening an existing file**
3. **Reading from a file**
4. **Writing to a file**
5. **Moving to a specific location in a file (seek)**
6. **Closing a file**



Stream is a Sequence of data bytes, which is used to read and write data to a file.

The streams that represent the input data of a program are known as **Input Streams,** where as the streams that represent the output data of a program are known as **Output Streams.**

Input streams gets the data from different **input devices** such as keyboard and mouse and provide input data to the program.

Output Streams obtain data from the program and write that on different **Output Devices** such as Memory or print them on the Screen.

Types of Files

1.Text file : It can be thought of as a stream of characters that can be processed sequentially and in forward direction only.

2.Binary file : It is collection of bytes like images.

3.Sequential File: Data stored sequentially, to read the last record of the file, we need to traverse all the previous records before it. Ex: files on magnetic tapes.

4.Random Access File: Data can be accessed and modified randomly. We can read any record directly. Ex : files on disks.

Steps involved using files

```
/*program to write and read data from file*/
#include<stdio.h>
void  main() {
    FILE *fp;
    char ch;
    fp = fopen("data.txt", "w");
    if(fp == NULL) {
        printf("Cannot open file.");
        exit(0);
     }
     printf("Type text ( to stop press '.' ) : ");
     while(ch != '.') {
        ch = getche();
        fputc(ch,fp);
     }
     fclose(fp);
     printf("\nContants read : ");
     fp = fopen("data.txt","r");
     while(!feof(fp))
        printf("%d", fgetc(fp));
    fclose(fp);
}
```

1. Declaring FILE pointer variable :

Syntax :

```
FILE *file_pointer1;
```

2. Open a file using fopen() function :

Syntax :

```
fp= fopen("filename","mode of access");
```

3. fputc() – Used to write a character to the file.

Syntax :

```
fputc(character, file_pointer);
```

4. fgetc() – Used to read a character to the file.

Syntax :

```
fgetc(file_pointer);
```

5. Close a file using fclose() function :

Syntax :

```
fclose(file_pointer);
```

```
  /* creating a new file */
int main(){
   char ch;FILE *fp;
   printf("\nEnter the text\n");
   printf("\n\t(Press ctrl+Z after
                    completing text)\n");
   fp=fopen("str.txt","w");
   while((ch=getchar())!=EOF)
        putc(ch,fp);
   fclose(fp);
}
```

file pointer used to handle files

filepointer=fopen("filename","**mode**");

putc(character,filepointer);

fclose(filepointer);

```
/* Reading the contents of existing file */
#include<stdio.h>
int main()  {
    FILE *fp;
    char ch;
    fp=fopen("str.txt","r");
    while((ch=getc(fp))!=EOF)
         printf("%c",ch);
    fclose(fp);
}
```

```
/* appending data to an existing file */
int main()  {
    FILE *fp; char ch;
    printf("\nEnter the text\n");
    printf("\n\t(Press ctrl+Z after
                  completing text)\n");
    fp=fopen("str.txt","a");
    while((ch=getchar())!=EOF)
              putc(ch,fp);
    fclose(fp);
}
```

r -- open a file in read mode -- if file exits, the marker is positioned at beginning. -- if file does not exist, error returned.	r+ -- open a file in read and write mode -- if file exits, the marker is positioned at beginning. -- if file does not exist, NULL returned.
w -- open a file in write mode -- if file exits, all its data is erased. -- if file does not exist, it is created.	w+ -- open a file in read and write mode -- if file exits, all its data is erased. -- if file does not exist, it is created.
a -- open a file in append mode -- if file exits, the marker is positioned at end. -- if file does not exist, it is created.	a+ -- open a file in read and append mode -- if file exits, the marker is positioned at end. -- if file does not exist, it is created.

rb , wb , ab, rb+ , wb+ , ab+ are modes to operate a file as binary file.

```
int main( ) {   /* Without using w+ */
   FILE *fp; char ch;
   printf("\nEnter the text\n");
   fp=fopen("str1.txt","w");
   while((ch=getchar())!='\n‘)putc(ch,fp);
   fclose(fp);
   fp=fopen("str1.txt","r");
   while((ch=getc(fp))!=EOF)
      printf("%c",ch);
   fclose(fp);
}
```

```
  /* open a file in read and write mode */
int main( ) {
   FILE *fp; char ch;
   printf("\nEnter the text\n");
   fp=fopen("str1.txt","w+");
   while((ch=getchar())!='\n') putc(ch,fp);
   rewind(fp);
   while((ch=getc(fp))!=EOF)
      printf("%c",ch);
   fclose(fp);
}
```

File Input / Output Functions	
fopen(fp, mode)	Open existing file / Create new file
fclose(fp)	Closes a file associated with file pointer.
closeall ( )	Closes all opened files with fopen()
fgetc(ch, fp)	Reads character from current position and advances the pointer to next character.
fprintf( )	Writes all types of data values to the file.
fscanf()	Reads all types of data values from a file.
gets()	Reads string from a file.
puts()	Writes string to a file.
getw()	Reads integer from a file.
putw()	Writes integer to a file.
fread()	Reads structured data written by fwrite() function
fwrite()	Writes block of structured data to the file.
fseek()	Sets the pointer position anywhere in the file
feof()	Detects the end of file.
rewind()	Sets the record pointer at the beginning of the file.
ferror()	Reports error occurred while read/write operations
fflush()	Clears buffer of input stream and writes buffer of output stream.
ftell()	Returns the current pointer position.

Text files Vs Binary Files

```
/* Copying one binary file to other */

#include<stdio.h>
int main( )
{
   FILE *fs,*ft;
   char ch;
   fs=fopen("pr1.exe","rb");
   if(fs==NULL){
      printf("\nCannot Open the file");
      exit(0);
   }
   ft=fopen("newpr1.exe","wb");
   if(ft==NULL) {
       printf("\nCannot open the file");
       fclose(fs);
       exit( 0);
    }
    while((ch=getc(fs))!=EOF)
              putc(ch,ft);
    fclose(fs);
    fclose(ft);
}
```

"rb" → open a file in read mode
"wb" → open a file in write mode
"ab" → open a file in append mode
"rb+" → open a pre-existing file in read and write mode
"wb+" →open a file in read and write mode
"ab+" → open a file in read and append mode

Text File :

i) Data are human readable characters.
ii) Each line ends with a newline character.
iii) Ctrl+z or Ctrl+d is end of file character.
iv) Data is read in forward direction only.
v) Data is converted into the internal format before being stored in memory.

Binary File :

i) Data is in the form of sequence of bytes.
ii) There are no lines or newline character.
iii) An EOF marker is used.
iv) Data may be read in any direction.
v) Data stored in file are in same format that they are stored in memory.

Random Access File

```
int main() {
  int n,i;
  char *str="abcdefghijklmnopqrstuvwxyz";
  FILE *fp = fopen("notes.txt","w");
  if(fp==NULL){
     printf("\nCannot open file."); exit(0);
  }
  fprintf(fp,"%s",str);
  fclose(fp);
  fp = fopen("notes.txt","r");
  printf("\nText from position %d : \n\t“,ftell(fp));
  fseek(fp, 3 ,SEEK_SET);
  for(i=0; i < 5; i++)  putchar(getc(fp));
  printf("\nText from position %d : \n\t“,ftell(fp));
  fseek(fp, 4 ,SEEK_CUR);
  for(i=0; i < 6; i++)  putchar(getc(fp));
  fseek(fp,  - 10 , SEEK_END);
  printf("\nText from position %d : \n\t“,ftell(fp));
  for(i=0; i < 5; i++)  putchar(getc(fp));
  printf("\nCurrent position : %d",ftell(fp));
  rewind(fp);
  printf("\nText from starting : \n\t");
  for(i=0;i < 8 ; i++) putchar(getc(fp));
  fclose(fp);
}
```

ftell(file_pointer)
-- returns the current position of file pointer in terms of bytes from the beginning.

rewind(file-pointer)
-- moves the file pointer to the starting of the file, and reset it.

fseek(fileptr, offset, position)
– moves the file pointer to the location (position + offset)

position :
- **SEEK_SET –** beginning of file
- **SEEK_CUR –** current position
- **SEEK_END –** end of the file

output :

```
Text from position 3 :
      defgh
Text from position 12 :
      mnopqr
Text from position 16 :
      qrstu
Current position : 21
Text from starting :
      abcdefgh
```

Formatted I / O

```
/* using fscanf() and fprintf() functions */
#include<stdio.h>
int main( ) {
   FILE *fp;
   int rno , i;
   float avg;
   char  name[20] ,  filename[15];
   printf("\nEnter the filename\n");
   scanf("%s",filename);
   fp=fopen(filename,"w");
   for(i=1;i<=3;i++)  {
      printf("Enter rno,name,average
         of student no:%d",i);
      scanf("%d %s %f",&rno,name,&avg);
      fprintf(fp,"%d %s %f\n",rno,name,avg);
   }
   fclose(fp);
   fp=fopen ( filename, "r“ );
   for(i=1;i<=3;i++)  {
      fscanf(fp,"%d %s %f",&rno,name,&avg);
      printf("\n%d %s %f",rno,name,avg);
   }
   fclose(fp);
}
```

```
/*Receives strings from keyboard
  and writes them to file
  and prints on screen*/
#include<stdio.h>
int main( )  {
  FILE *fp;
  char s[80];
  fp=fopen(“poem.txt","w");
  if(fp==NULL)  {
     puts("Cannot open file");exit(0);
   }
   printf("\nEnter a few lines of text:\n");
   while(strlen(gets(s))>0){
      fputs(s,fp);
      fputs("\n",fp);
   }
   fclose(fp);
   fp=fopen(“poem.txt","r");
    if(fp==NULL){
      puts("Cannot open file"); exit(0);
   }
   printf("\nContents of file:\n");
   while(fgets(s,79,fp)!=NULL)
      printf("%s",s);
   fclose(fp);
}
```

```
/* using putw() and getw() functions */
#include<stdio.h>
int main( )  {
   FILE *fp1,*fp2; int i,n;
   char *filename;
   clrscr();
   fp1=fopen("test.txt","w");
   for(i=10;i<=50;i+=10)
     putw(i,fp1);
   fclose(fp1);
   do {
     printf("\nEnter the filename : \n");
     scanf("%s",filename);
     fp2=fopen(filename,"r");
     if(fp2==NULL)
      printf("\nCannot open the file");
   } while(fp2==NULL);
   while(!feof(fp2)) {
     n=getw(fp2);
     if(n==-1) printf("\nRan out of data");
     else printf("\n%d",n);
   }
   fclose(fp2);
   getch();
}
```

Standard I / O

fputc()	fgetc()	Individual characters
fputs()	fgets()	Character Strings
fprintf()	fscanf()	Formatted ASCII
fwrite()	fread()	Binary files
write()	read()	Low-level binary

Predefined Streams

NAME	MEANING
stdin	Standard input (from keyboard)
stdout	Standard output (to monitor)
stderr	Standard error output (to monitor)
stdaux	Standard auxiliary (both input and output)
stdprn	Standard printer output(to printer)

Handling Records (structures) in a File

```
struct player {
  char name[40]; int age; int runs;
} p1,p2;
void main() {
     int i ; FILE *fp = fopen ( "player.txt", "w");
     if(fp == NULL)  {
         printf ("\nCannot open file."); exit(0);
     }
     for(i=0;i<3;i++) {
       printf("Enter name, age, runs of a player : ");
       scanf("%s %d %d",p1.name, &p1.age,&p1.runs);
       fwrite(&p1,sizeof(p1),1,fp);
     }
    fclose(fp);
    fp = fopen("player.txt","r");
    printf("\nRecords Entered : \n");
    for(i=0;i<3;i++) {
       fread(&p2,sizeof(p2),1,fp);
       printf("\nName : %s\nAge : %d\nRuns : %d",p2.name,p2.age,p2.runs);
   }
   fclose(fp);
}
```

Error Handling:

While operating on files, there may be a chance of having certain errors which will cause abnormal behavior in our programs.

1)Opening an file that was not present in the system.
2)Trying to read beyond the end of file mark.
3)Device overflow.
4)Trying to use a file that has not been opened.
5)Trying to perform an operation on a file when the file is opened for another type of operation.
6)Attempting to write to a write-protected file.

feof(fp)→ returns non-zero integer value if we reach end of the file otherwise zero.
ferror(fp)→ returns non-zero integer value if an error has been detected otherwise zero
perror(string)→prints the string, a colon and an error message specified by the compiler
file pointer (fp) will return NULL if it cannot open the specified file.

```
/* program on ferror( ) and perror ( ) */
#include<stdio.h>
int main(){
   FILE *fp;
   char ch;
   fp=fopen("str.txt","w");
   ch=getc(fp);
   if(ferror(fp))
        perror("Error Raised : ");
   else
       printf("%c",ch);
   fclose(fp);
}
```

```
#include<stdio.h>
main(){
FILE *fp1,*fp2;
int i,number;
char *filename;
fp1=fopen("TEST.txt","w");
for(i=10;i<=50;i+=10)
                putw(i,fp1);
fclose(fp1);
file:
printf("\nEnter the filename\n");
scanf("%s",filename);
fp2=fopen(filename,"r");
if(fp2==NULL){
     printf("\nCannot open the file");
     printf("\nType File name again");
      goto file;}
else{
        for(i=1;i<=10;i++){
                number=getw(fp2);
                                if(feof(fp2)){
                    printf("\nRan out of data");
                    break;}
else
    printf("\n%d",number); }  }
fclose(fp2);}
```

fp will return NULL if unable to open the file

feof(fp) returns 1 if it reaches end of file otherwise 0.

Output:

```
Enter the filename
TETS.txt
Cannot open the file
Type the File name again
Enter the filename
TEST.txt
10
20
30
40
50
Ran out of data.
```

Structure of FILE pointer

```
Type: FILE
   File control structure for streams.
 typedef struct {
   short   level;
   unsigned      flags;
   char      fd;
   unsigned char  hold;
   short         bsize;
   unsigned char *buffer, *curp;
   unsigned      istemp;
   short         token;
 } FILE;
```

File status functions

feof(file_pointer)
  -- to check if end of file has been reached.
ferror(file_pointer)
  -- to check the error status of the file
clearerr(file_pointer)
  -- to reset the error status of the file

File Management Functions

rename("old-filename","new-filename");
  -- It renames the file with the new name

remove("filename")
  -- It removes the file specified (macro)

unlink("filename");
  -- It also removes the file name

fcloseall();
  -- Closes all the opened streams in the program except standard streams.

fflush(file_pointer)
  -- Bytes in the buffer transferred to file.

tmpfile ()
  -- creates a temporary file, to be deleted when program is completed.

tmpnam("filename")
  -- creates a unique file name