LECTURE NOTES

ON

PROGRAMMING IN ‘C’ AND DATA STRUCTURE

1ST YEAR

Sushree Sangeeta Sahoo

ASST. PROFESSOR

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

GANDHI INSTITUTE OF TECHNOLOGY AND MANAGEMENT(GITAM)

Affiliated to BPUT & SCTE&VT, Govt. of Odisha

Approved by AICTE, New Delhi

SYLLABUS

Module –I

Fundamentals of C : Algorithms an Flow chart, C as a middle level language, Structure of C program, Character set, Identifiers, keyword, data types, Constants and variables, statements, expression, operators, precedence of operators, Input-output, Assignments, control structures decision making and branching.

Module –II

Arrays, Functions and Structure and Union : functions, user defined vs. standard functions, formal vs. actual arguments, function – category, function prototypes, parameter passing, recursion, Array: 1-D ,2-D, Matrix Operations, String, Structure, Union

Module –III

Pointers & Dynamic Memory Allocation : pointer arithmetic, passing parameters, Call by Value vs. Call by Reference, pointer to pointer, pointer to structure, pointer to function, dynamic memory allocation

Module –IV

Data Structures : pointer Introduction to Data structure, Linear Linked list : Creation, Insertion, Deletion, Stack, Stack applications(Infix to postfix), Queue(Linear & Circular)

Module –V

Tree, Introduction to sorting & Searching : point Binary tree, Binary Search Tree, Sorting(Bubble sort, Quick sort), Searching(Linear search, Binary search)

<u>Lecture Note: 1</u>

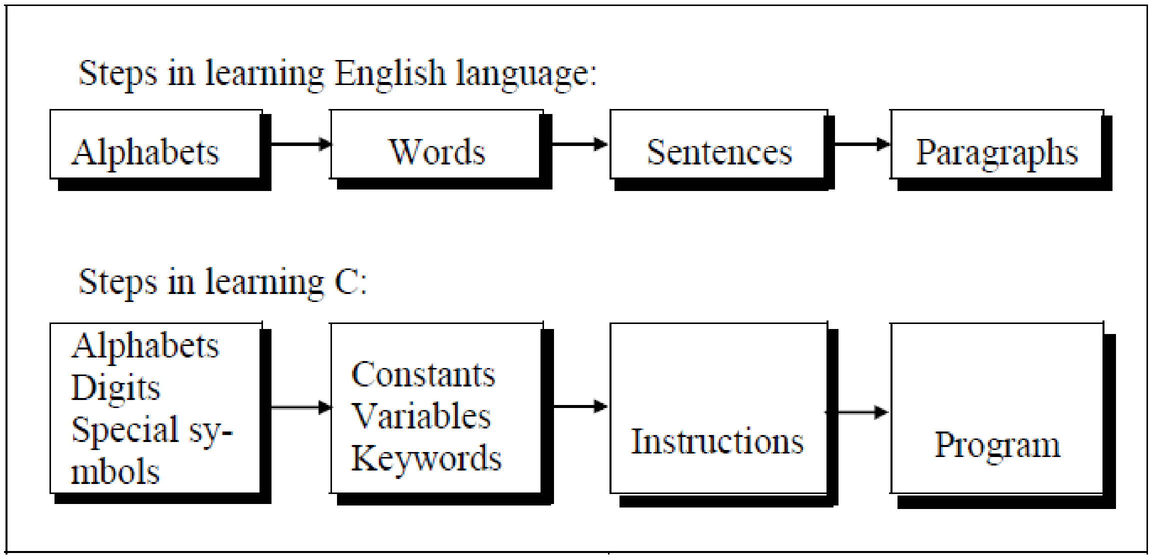
Introduction to C

C is a programming language developed at AT & T's Bell Laboratories of USA in 1972. It was designed and written by a man named Dennis Ritchie. In the late seventies C began to replace the more familiar languages of that time like PL/I, ALGOL, etc

ANSI C standard emerged in the early 1980s, this book was split into two titles: The original was still called ***Programming in C***, and the title that covered ANSI C was called ***Programming in ANSI C***. This was done because it took several years for the compiler vendors to release their ANSI C compilers and for them to become ubiquitous. It was initially designed for programming UNIX operating system. Now the software tool as well as the C compiler is written in C. Major parts of popular operating systems like Windows, UNIX, Linux is still written in C. This is because even today when it comes to performance (speed of execution) nothing beats C. Moreover, if one is to extend the operating system to work with new devices one needs to write device driver programs. These programs are exclusively written in C. C seems so popular is because it is **reliable**, **simple** and **easy** to use. often heard today is – "C has been already superceded by languages like C++, C# and Java.

Program

There is a close analogy between learning English language and learning C language. The classical method of learning English is to first learn the alphabets used in the language, then learn to combine these alphabets to form words, which in turn are combined to form sentences and sentences are combined to form paragraphs. Learning C is similar and easier. Instead of straight-away learning how to write programs, we must first know what alphabets, numbers and special symbols are used in C, then how using them constants, variables and keywords are constructed, and finally how are these combined to form an **instruction**. A group of instructions would be combined later on to form a **program**.


Steps in learning English language:
Alphabets
Words
Sentences
Paragraphs
Steps in learning C:
Alphabets Digits Special sy-mbols
Constants Variables Keywords
Instructions
Program

So a computer ***program*** is just a collection of the instructions necessary to solve a specific problem. The basic operations of a computer system form what is known as the computer's ***instruction set.*** And the approach or method that is used to solve the problem is known as an ***algorithm***.

So for as programming language concern these are of two types.

1) Low level language

2) High level language

LOW LEVEL LANGUAGE

Low level languages are **machine level** and **assembly level language**. In machine level language computer only understand digital numbers i.e. in the formof 0 and 1. So, instruction given to the computer is in the form binary digit, whichis difficult to implement instruction in binary code. This type of program is not portable, difficult to maintain and also error prone. The **assembly language** is on other hand modified version of machine level language. Where instructions are given in English like word as ADD, SUM, MOV etc. It is easy to write and understand but not understand by the machine. So the translator used here is assembler to translate into machine level. Although language is bit easier, programmer has to know low level details related to low level language. In the assembly level language the data are stored in the computer register, which variesfor different computer. Hence it is not portable.

High level language:

These languages are machine independent, means it is portable. The language in this category is Pascal, Cobol, Fortran etc. High level languages are understood by the machine. So it need to translate by the translator into machine level. A translator is software which is used to translate high level language as well as low level language in to machine level language.

Three types of translator are there: Compiler, Interpreter,Assembler
Compiler and interpreter are used to convert the high level language into machine level language. The program written in high level language is known as source program and the corresponding machine level language program is called as object program. Both compiler and interpreter perform the same task but there working is different. Compiler read the program at-a-time and searches the error and lists them. If the program is error free then it is converted into object program. When program size is large then compiler is preferred. Whereas interpreter read only one line of the source code and convert it to object code. If it check error, statement by statement and hence of take more time.

Integrated Development Environments (IDE)

The process of editing, compiling, running, and debugging programs is often managed by a single integrated application known as an Integrated Development Environment, or IDE for short. An IDE is a windows-based program that allows us to easily manage large software programs, edit files in windows, and compile, link, run, and debug programs.

On Mac OS X, CodeWarrior and Xcode are two IDEs that are used by many programmers. Under Windows, Microsoft Visual Studio is a good example of a popular IDE. Kylix is a popular IDE for developing applications under Linux. Most IDEs also support program development in several different programming languages in addition to C, such as C# and C++.

Structure of C Language program

1) Comment line

2) Preprocessor directive

3) Global variable declaration

4) main function( )

```
            {
              Local variables;
  Statements;
  }
  User defined function
 }
}
```

Comment line

It indicates the purpose of the program. It is represented as

/*……………………………..*/

Comment line is used for increasing the readability of the program. It is useful in explaining the program and generally used for documentation. It is enclosed within the decimeters. Comment line can be single or multiple line but should not be nested. It can be anywhere in the program except inside string constant & character constant.

Preprocessor Directive:
#include<stdio.h> tells the compiler to include information about the standard input/output library. It is also used in symbolic constant such as #define PI 3.14(value). The stdio.h (standard input output header file) contains definition &declaration of system defined function such as printf( ), scanf( ), pow( ) etc. Generally printf() function used to display and scanf() function used to read value

Global Declaration:

This is the section where variable are declared globally so that it can be access by all the functions used in the program. And it is generally declared outside the function :

main()

It is the user defined function and every function has one main() function from where actually program is started and it is encloses within the pair of curly braces.

The main( ) function can be anywhere in the program but in general practice it is placed in the first position.

Syntax :

```
  main()
{
……..
……..
……..
}
```

The main( ) function return value when it declared by data type as

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

The main function does not return any value when void (means null/empty) as

```
void  main(void ) or  void main()

{

 printf (“C language”);

}
```

Output: C language

The program execution start with opening braces and end with closing brace.

And in between the two braces declaration part as well as executable part is mentioned. And at the end of each line, the semi-colon is given which indicates statement termination.

/*First c program with return statement*/

```
#include <stdio.h>

int main (void)

{

printf ("welcome to c Programming language.\n");

return 0;

}
```

Output: welcome to c programming language.

Steps for Compiling and executing the Programs

A compiler is a software program that analyzes a program developed in a particular computer language and then translates it into a form that is suitable for execution on a particular computer system. Figure below shows the steps that are involved in entering, compiling, and executing a

computer program developed in the C programming language and the typical Unix commands that would be entered from the command line.

Step 1: The program that is to be compiled is first typed into a *file* on the computer system. There are various conventions that are used for naming files, typically be any name provided the last two characters are **".c"** or file with extension .c. So, the file name **prog1.c** might be a valid filename for a C program. A text editor is usually used to enter the C program into a file. For example, vi is a popular text editor used on Unix systems. The program that is entered into the file is known as the ***source program*** because it represents the original form of the program expressed in the C language.

Step 2: After the source program has been entered into a file, then proceed to have it compiled. The compilation process is initiated by typing a special command on the system. When this command is entered, the name of the file that contains the source program must also be specified. For example, under Unix, the command to initiate program compilation is called **cc**. If we are using the popular GNU C compiler, the command we use is **gcc**.

Typing the line

```
gcc prog1.c or cc prog1.c
```

In the first step of the compilation process, the compiler examines each program

statement contained in the source program and checks it to ensure that it conforms to the syntax and semantics of the language. If any mistakes are discovered by the compiler during this phase, they are reported to the user and the compilation process ends right there. The errors then have to be corrected in the source program (with the use of an editor), and the compilation process must be restarted. Typical errors reported during this phase of compilation might be due to an expression that has unbalanced parentheses (**syntactic error**), or due to the use of a variable that is not "defined" (**semantic error**).

Step 3: When all the syntactic and semantic errors have been removed from the program, the compiler then proceeds to take each statement of the program and translate it into a “lower” form that is equivalent to assembly language program needed to perform the identical task.

Step 4: After the program has been translated the next step in the compilation process is to translate the assembly language statements into actual machine instructions. The assembler takes each assembly language statement and converts it into a binary format known as ***object code***, which is then written into another file on the system. This file has the same name as the source file under Unix, with the last letter an “**o**” (**for *object***) instead of a “**c**”.

Step 5: After the program has been translated into object code, it is ready to be ***linked.*** This process is once again performed automatically whenever the cc or gcc command is issued under Unix. The purpose of the linking phase is to get the program into a final form for execution on the computer.

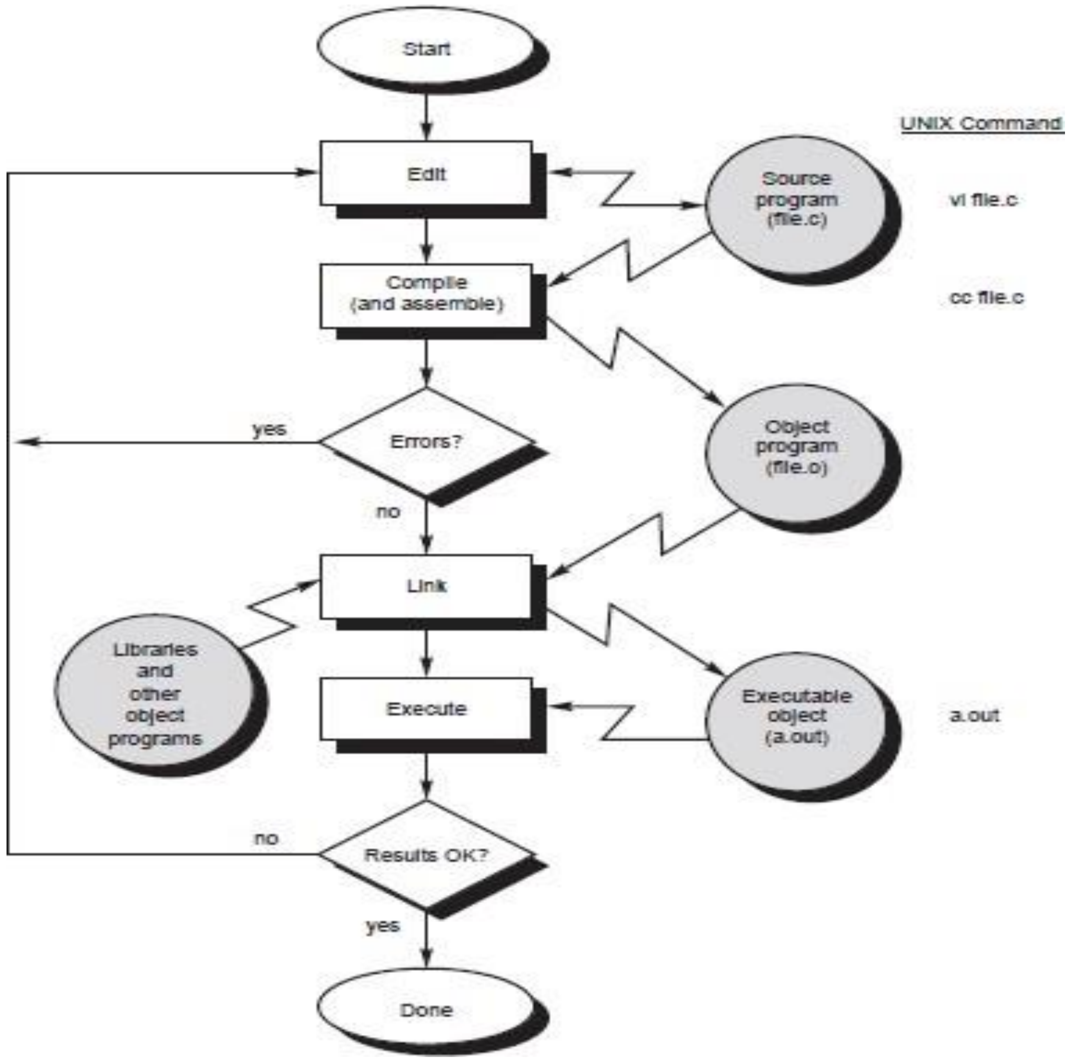
If the program uses other programs that were previously processed by the compiler, then during this phase the programs are linked together. Programs that are used from the system’s program ***library*** are also searched and linked together with the object program during this phase.

The process of compiling and linking a program is often called ***building.***

The final linked file, which is in an executable ***object*** code format, is stored in another file on the system, ready to be run or ***executed.*** Under Unix, this file is called **a.out** by default. Under Windows, the executable file usually has the same name as the source file, with the c extension replaced by an exe extension.

Step 6: To subsequently execute the program, the command **a.out** has the effect of *loading* the program called **a.out** into the computer's memory and initiating its execution.

When the program is executed, each of the statements of the program is sequentially executed in turn. If the program requests any data from the user, known as ***input***, the program temporarily suspends its execution so that the input can be entered. Or, the program might simply wait for an ***event***, such as a mouse being clicked, to occur. Results that are displayed by the program, known as ***output,*** appear in a window, sometimes called the ***console***. If the program does not produce the desired results, it is necessary to go back and reanalyze the program's logic. This is known as the ***debugging phase***, during which an attempt is made to remove all the known problems or ***bugs*** from the program. To do this, it will most likely be necessary to make changes to original source program.



/*Simple program to add two numbers*/

```
#include <stdio.h>int
main (void)
        {
        int  v1, v2, sum;                 //v1,v2,sum are variables and int is data type declared
        v1 = 150;
        v2 = 25;
        sum = v1 + v2;
```

```c
printf ("The sum of %i and %i is= %i\n", v1, v2, sum);

return 0;

}
```

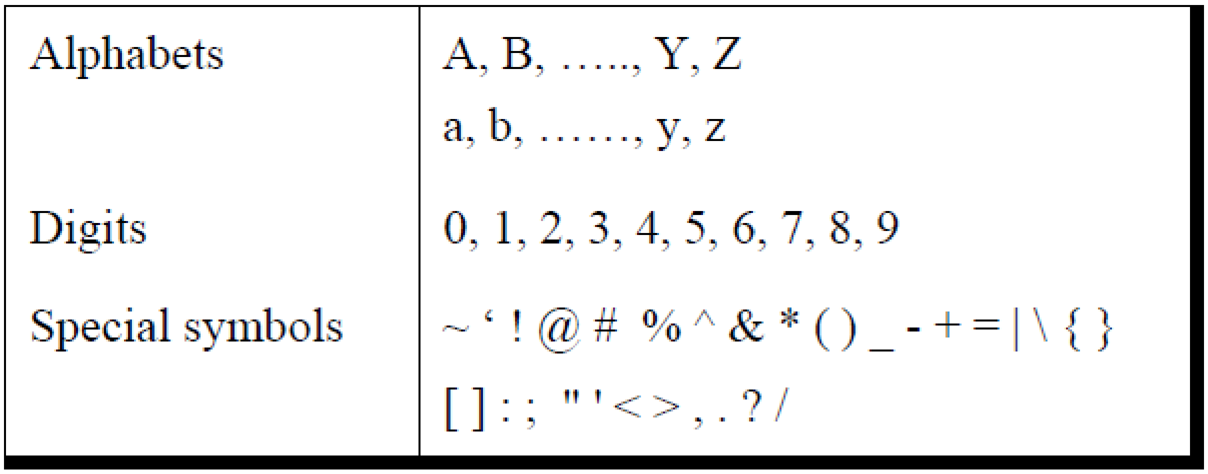
Output:

```
    The sum of 150 and 25 is=175
```

Lecture Note-3

Character set

A character denotes any alphabet, digit or special symbol used to represent information. Valid alphabets, numbers and special symbols allowed in C are



Alphabets	A, B, ….., Y, Z a, b, ……, y, z
Digits	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Special symbols	~ ‘ ! @ # % ^ & * ( ) _ - + = | \ { } [ ] : ; " ' < > , . ? /

The alphabets, numbers and special symbols when properly combined form constants, variables and keywords.

Identifiers

Identifiers are user defined word used to name of entities like variables, arrays, functions, structures etc. Rules for naming identifiers are:

1) name should only consists of alphabets (both upper and lower case), digits and underscore (_) sign.

2) first characters should be alphabet or underscore

3) name should not be a keyword

4) since C is a case sensitive, the upper case and lower case considered differently, for example code, Code, CODE etc. are different identifiers.

5) identifiers are generally given in some meaningful name such as value, net_salary, age, data etc. An identifier name may be long, some implementation recognizes only first eight characters, most recognize 31 characters. ANSI standard compiler recognize 31 characters. Some invalid identifiers are 5cb, int, res#, avg no etc.

Keywords

There are certain words reserved for doing specific task, these words are known as **reserved word** or **keywords.** These words are predefined and always written in lower case or small letter. These keywords cann't be used as a variable name as it assigned with fixed meaning. Some examples are **int, short, signed, unsigned, default, volatile, float, long, double, break, continue, typedef, static, do, for, union, return, while, do, extern, register, enum, case, goto, struct, char, auto, const etc.**

Data type

Data types refer to an extensive system used for declaring variables or functions of different types before its use. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted. The value of a variable can be changed any time.

C has the following 4 types of data types

basic built-in data types: int, float, double, char

Enumeration data type: enum

Derived data type: pointer, array, structure, union

Void data type: void

A variable declared to be of type int can be used to contain integral values only—that is, values that do not contain decimal places. A variable declared to be of type float can be used for storing floating- point numbers (values containing decimal places). The double type is the same as type float, only with roughly twice the precision. The char data type can be used to store a single character, such as the letter *a*, the digit character *6*, or a semicolon similarly A variable declared char can only store character type value.

There are two types of type qualifier in c

Size qualifier: short, long

Sign qualifier: signed, unsigned

When the qualifier unsigned is used the number is always positive, and when signed is used number may be positive or negative. If the sign qualifier is not mentioned, then by default sign qualifier is assumed. The range of values for signed data types is less than that of unsigned data type. Because in signed type, the left most bit is used to represent sign, while in unsigned type this bit is also used to represent the value. The size and range of the different data types on a 16 bit machine is given below:

Basic data type	Data type with type qualifier	Size (byte)	Range
char	char or signed char Unsigned char	1 1	-128 to 127 0 to 255
int	int or signed int unsigned int short int or signed short int unsigned short int long int or signed long int unsigned long int	2 2 1 1 4 4	-32768 to 32767 0 to 65535 -128 to 127 0 to 255 -2147483648 to 2147483647 0 to 4294967295
float	float	4	-3.4E-38 to 3.4E+38
double	double Long double	8 10	1.7E-308 to 1.7E+308 3.4E-4932 to 1.1E+4932

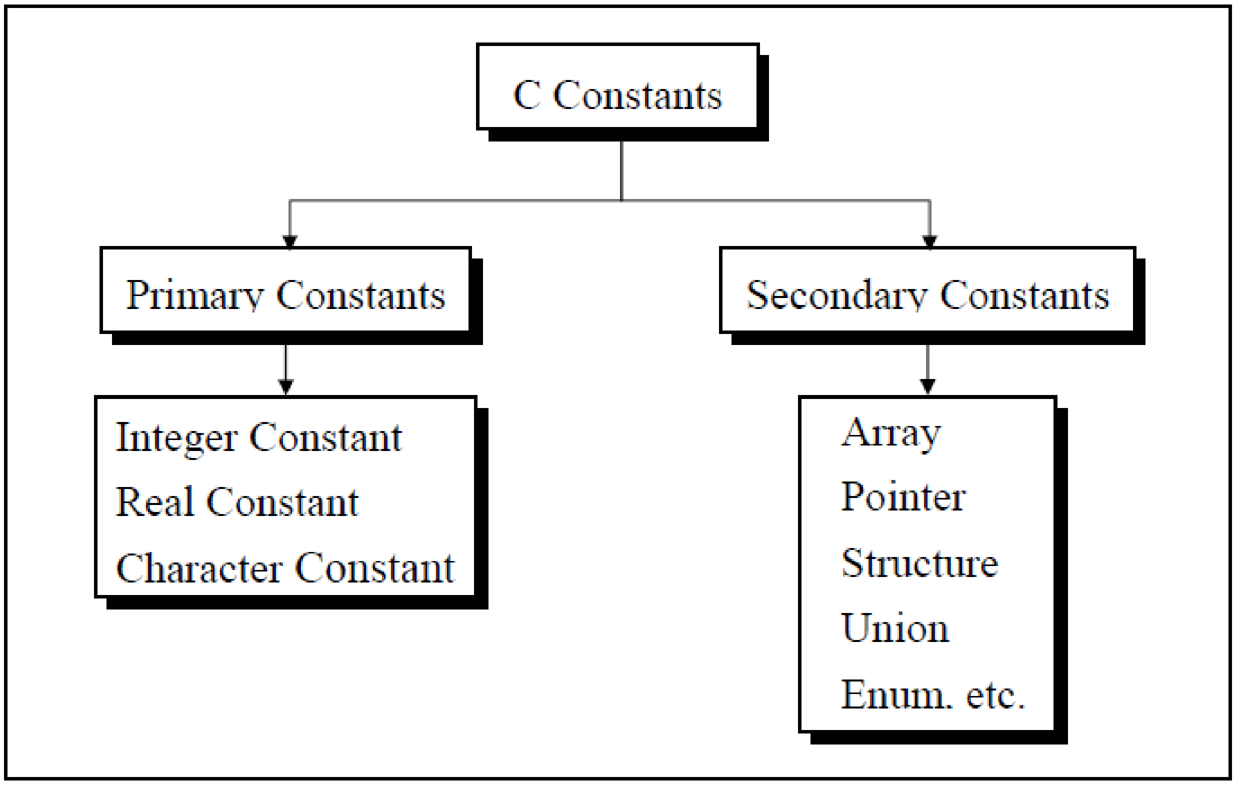
Lecture Note: 4

Constants

Constant is a any value that cannot be changed during program execution. In C, any number, single character, or character string is known as a *constant*. A constant is an entity that doesn't change whereas a variable is an entity that may change. For example, the number 50 represents a constant integer value. The character string "Programming in C is fun.\n" is an example of a constant character string. C constants can be divided into two major categories:

Primary Constants
Secondary Constants

These constants are further categorized as



Numeric constant
Character constant
String constant

Numeric constant: Numeric constant consists of digits. It required minimum size of 2 bytes and max 4 bytes. It may be positive or negative but by default sign is always positive. No comma or space is allowed within the numeric constant and it must have at least 1 digit. The allowable range for integer constants is -32768 to 32767. Truly speaking the range of an Integer constant depends upon the compiler. For a 16-bit compiler like Turbo C or Turbo C++ the range is –32768 to 32767. For a 32-bit compiler the range would be even greater. Mean by a 16-bit or a 32-bit compiler, what range of an Integer constant has to do with the type of compiler.

It is categorized a **integer constant** and **real constant.** An integer constants are whole number which have no decimal point. Types of integer constants are:

Decimal constant:	0------ 9(base 10)
Octal constant:	0------ 7(base 8)
Hexa decimal constant:	0----9, A ----- F(base 16)

In decimal constant first digit should not be zero unlike octal constant first digit must be zero(as 076, 0127) and in hexadecimal constant first two digit should be 0x/ 0X (such as 0x24, 0x87A). By default type of integer constant is integer but if the value of integer constant is exceeds range then value represented by integer type is taken to be unsigned integer or long integer. It can also be explicitly mention integer and unsigned integer type by suffix l/L and u/U.

Real constant is also called floating point constant. To construct real constant we must follow the rule of ,

-real constant must have at least one digit.
-It must have a decimal point.
-It could be either positive or negative.
-Default sign is positive.
-No commas or blanks are allowed within a real constant. Ex.: +325.34
426.0
-32.76

To express small/large real constant exponent(scientific) form is used where number is written in mantissa and exponent form separated by e/E. Exponent can be positive or negative integer but mantissa can be real/integer type, for example $3.6*10^5$=3.6e+5. By default type of floating point constant is double, it can also be explicitly defined it by suffix of f/F.

Character constant : Character constant represented as a single character enclosed within a single quote. These can be single digit, single special symbol or white spaces such as ‘9’,’c’,’$’, ‘ ’ etc. Every character constant has a unique integer like value in machine’s character code as if machine using ASCII (American standard code forinformation interchange). Some numeric value associated with each upper and lower case alphabets and decimal integers are as:

A ------------ Z ASCII value (65-90)

a ------------ z ASCII value (97-122)

0-------------9 ASCII value (48-59)

; ASCII value (59)

String constant

Set of characters are called string and when sequence of characters are enclosed within a double quote (it may be combination of all kind of symbols) is a string constant. String constant has zero, one or more than one character and at the end of the string null character(\0) is automatically placed by compiler. Some examples are “,sarathina” , “908”, “3”,” ”, “A” etc. In C although same characters are enclosed within single and double quotes it represents different meaning such as “A” and ‘A’ are different because first one is string attached with null character at the end but second one is character constant with its corresponding ASCII value is 65.

Symbolic constant

Symbolic constant is a name that substitute for a sequence of characters and, characters may be numeric, character or string constant. These constant are generally defined at the beginning of the program as

#define name value , here name generally written in upper case for example

```
#define MAX 10
#define CH 'b'
  #define NAME "sony"
```

Variables

Variable is a data name which is used to store some data value or symbolic names for storing program computations and results. The value of the variable can be change during the execution. The rule for naming the variables is same as the naming identifier. Before used in the program it must be declared. Declaration of variables specify its name, data types and range of the value that variables can store depends upon its data types.

Syntax:

```
int a;
char c;
float f;
```

Variable initialization

When we assign any initial value to variable during the declaration, is called initialization of variables. When variable is declared but contain undefined value then it is called garbage value. The variable is initialized with the assignment operator such as

Data type variable name=constant;

```
Example: int a=20;
         Or int a;
            a=20;
```

Expressions

An expression is a combination of variables, constants, operators and function call. It can be arithmetic, logical and relational for example:-

```
int z=  x+y    // arithmatic expression

 a>b      //relational

a==b         // logical
func(a, b)   // function call
```

Expressions consisting entirely of constant values are called *constant expressions*. So, the expression
121 + 17 - 110
is a constant expression because each of the terms of the expression is a constant value. But if i were declared to be an integer variable, the expression
180 + 2 – j
would not represent a constant expression.

Operator

This is a symbol use to perform some operation on variables, operands or with the constant. Some operator required 2 operand to perform operation or Some required single operation.

Several operators are there those are, arithmetic operator, assignment, increment , decrement, logical, conditional, comma, size of , bitwise and others.

1. **Arithmatic Operator**

This operator used for numeric calculation. These are of either Unary arithmetic operator, Binary arithmetic operator. Where Unary arithmetic operator required only one operand such as +,-, ++, --,!, tiled. And these operators are addition, subtraction, multiplication, division. Binary arithmetic operator on other hand required two operand and its operators are +(addition), -(subtraction), *(multiplication), /(division), %(modulus). But modulus cannot applied with floating point operand as well as there are no exponent operator in c.

Unary (+) and Unary (-) is different from addition and subtraction.

When both the operand are integer then it is called integer arithmetic and the result is always integer. When both the operand are floating point then it is called floating arithmetic and when operand is of integer and floating point then it is called mix type or mixed mode arithmetic . And the result is in float type.

2.Assignment Operator

A value can be stored in a variable with the use of assignment operator. The assignment operator(=) is used in assignment statement and assignment expression. Operand on the left hand side should be variable and the operand on the right hand side should be variable or constant or any expression. When variable on the left hand side is occur on the right hand side then we can avoid by writing the compound statement. For example,

```
  int x= y;
int  Sum=x+y+z;
```

3.Increment and Decrement

The Unary operator ++, --, is used as increment and decrement which acts upon single operand. Increment operator increases the value of variable by one .Similarly decrement operator decrease the value of the variable by one. And these operator can only used with the variable, but cann't   use with expression and constant as ++6 or ++(x+y+z).

It again categories into prefix post fix . In the prefix the value of the variable is incremented 1st, then the new value is used, where as in postfix the operator is written after the operand(such as m++,m--).

EXAMPLE

let y=12;

z= ++y;

y= y+1;

z= y;

Similarly in the postfix increment and decrement operator is used in the operation . And then increment and decrement is perform.

EXAMPLE

let x= 5;

y= x++;

y=x;

x= x+1;

4.Relational Operator

It is use to compared value of two expressions depending on their relation. Expression that contain relational operator is called relational expression.

Here the value is assign according to true or false value.

a.(a>=b) || (b>20)

b.(b>a) && (e>b)

c. 0(b!=7)

5. Conditional Operator

It sometimes called as ternary operator. Since it required three expressions as operand and it is represented as (? , :).

SYNTAX

exp1 ? exp2 :exp3

Here exp1 is first evaluated. It is true then value return will be exp2 . If false then exp3.

EXAMPLE

```
void main()

{

 int a=10, b=2

  int s= (a>b) ? a:b;

  printf("value is:%d");

 }
```

Output:

```
    Value is:10
```

6. Comma Operator

Comma operator is use to permit different expression to be appear in a situation where only one expression would be used. All the expression are separator by comma and are evaluated from left to right.

EXAMPLE

```
int i, j, k, l;

for(i=1,j=2;i<=5;j<=10;i++;j++)
```

7. Sizeof Operator

Size of operator is a Unary operator, which gives size of operand in terms of byte that occupied in the memory. An operand may be variable, constant or data type qualifier.

Generally it is used make portable program(program that can be run on different machine) . It determines the length of entities, arrays and structures when their size are not known to the programmer. It is also use to allocate size of memory dynamically during execution of the program.

EXAMPLE

```
main( )

{

int sum;

float f;

printf( "%d%d" ,size of(f), size of (sum) );

printf("%d%d", size of(235 L), size of(A));


}
```

8. Bitwise Operator

Bitwise operator permit programmer to access and manipulate of data at bit level. Various bitwise operator enlisted are

one's complement	(~)
bitwise AND	(&)
bitwise OR	(|)
bitwise XOR	(^)
left shift	(<<)
right shift	(>>)

These operator can operate on integer and character value but not on float and double. In bitwise operator the function showbits( ) function is used to display the binary representation of any integer or character value.

In one's complement all 0 changes to 1 and all 1 changes to 0. In the bitwise OR its value would obtaining by 0 to 2 bits.

As the bitwise OR operator is used to set on a particular bit in a number. Bitwise AND the logical AND.

It operate on 2operands and operands are compared on bit by bit basic. And hence both the operands are of same type.

Logical or Boolean Operator

Operator used with one or more operand and return either value zero (for false) or one (for true). The operand may be constant, variables or expressions. And the expression that combines two or more expressions is termed as logical expression. C has three logical operators :

Operator	Meaning
&&	AND
||	OR
!	NOT

Where logical NOT is a unary operator and other two are binary operator. Logical AND gives result true if both the conditions are true, otherwise result is false. And logial OR gives result false if both the condition false, otherwise result is true.

Precedence and associativity of operators

Operators	**Description**	**Precedence level**	**Associativity**
()	function call	**1**	left to right
[]	array subscript		
→	arrow operator		
.	dot operator		
+	unary plus	2	right to left
-	unary minus		
++	increment		
- -	decrement		
!	logical not		
~	1's complement		
*	indirection		
&	address		
(data type)	type cast		
sizeof	size in byte		
*	multiplication	3	left to right
/	division		
%	modulus		
+	addition	4	left to right

-	subtraction		
<<	left shift	**5**	left to right
>>	right shift		
<=	less than equal to	6	left to right
>=	greater than equal to		
<	less than		
>	greater than		
==	equal to	7	left to right
!=	not equal to		
&	bitwise AND	8	left to right
^	bitwise XOR	9	left to right
|	bitwise OR	10	left to right
&&	logical AND	11	
||	logical OR	12	
?:	conditional operator	13	
=, *=, /=, %= **&=**, ^=, <<= **>>=**	assignment operator	14	right to left
,	comma operator	15	

Control Statement

Generally C program statement is executed in a order in which they appear in the program. But sometimes we use decision making condition for execution only a part of program, that is called control statement. Control statement defined how the control is transferred from one part to the other part of the program. There are several control statement like if...else, switch, while, do. while, for loop, break, continue, goto etc.

Loops in C

Loop:-it is a block of statement that performs set of instructions. In loops

Repeating particular portion of the program either a specified number of time or until a particular no of condition is being satisfied.

There are three types of loops in c

1. While loop

2.do while loop

3.for loop

While loop

Syntax:-

```
while(condition)

{

Statement 1;

Statement 2;

`

Or      while(test
        condition)
```

The test condition may be any expression .when we want to do something a fixed no of times but not known about the number of iteration, in a program then while loop is used.

Here first condition is checked if,   it is true body of the loop is executed else, If condition is false control will be come out of loop.

Example:-

```
/* wap to print 5 times welcome to C" */
#include<stdio.h>
void main()
{
int p=1;
While(p<=5)
{
printf("Welcome to C\n");
P=p+1;
}
}
```

```
Output:  Welcome to C
          Welcome to C
         Welcome to C
         Welcome to C
         Welcome to C
```

So as long as condition remains true statements within the body of while loop will get executed repeatedly.

do while loop

This (do while loop) statement is also used for looping. The body of this loop may contain single statement or block of statement. The syntax for writing this statement is:

Syntax:-

```
Do
{
Statement;
}
while(condition);
```

Example:-

```
#include<stdio.h>
void main()
{
int X=4;
 do
{
 Printf("%d",X);
X=X+1;
```

```
  }whie(X<=10);
    Printf(" ");
}
```

Output: 4 5 6 7 8 9 10

Here firstly statement inside body is executed then condition is checked. If the condition is true again body of loop is executed and this process continue until the condition becomes false. Unlike while loop semicolon is placed at the end of while.

There is minor difference between while and do while loop, while loop test the condition before executing any of the statement of loop. Whereas do while loop test condition after having executed the statement at least one within the loop.

If initial condition is false while loop would not executed it's statement on other hand do while loop executed it's statement at least once even If condition fails for first time. It means do while loop always executes at least once. **Notes:**

Do while loop used rarely when we want to execute a loop at least once.

Lecture Note: 8

for loop

In a program, for loop is generally used when number of iteration are known in advance. The body of the loop can be single statement or multiple statements. Its syntax for writing is:

Syntax:-

```
for(exp1;exp2;exp3)

{

Statement;

}
```

Or

```
for(initialized counter; test counter; update counter)

{

Statement;

}
```

Here exp1 is an initialization expression, exp2 is test expression or condition and exp3 is an update expression. Expression 1 is executed only once when loop started and used to initialize the loop variables. Condition expression generally uses relational and logical operators. And updation part executed only when after body of the loop is executed.

Example:-

```
void main()

{

int i;

for(i=1;i<10;i++)

{
 Printf(“ %d ”, i);

  }

}
```

Output:-1 2 3   4 5 6 7 8 9

Nesting of loop

When a loop written inside the body of another loop then, it is known as nesting of loop. Any type of loop can be nested in any type such as while, do while, for. For example nesting of for loop can be represented as :

```
void main()

{

int i,j;

for(i=0;i<2;i++)

for(j=0;j<5; j++)

printf(“%d %d”, i, j);

 }
```

```
Output: i=0

     j=0 1 2 3 4

      i=1

     j=0 1 2 3 4
```

Break statement(break)

Sometimes it becomes necessary to come out of the loop even before loop condition becomes false then break statement is used. Break statement is used inside loop and switch statements. It cause immediate exit from that loop in which it appears and it is generally written with condition. It is written with the keyword as **break.** When break statement is encountered loop is terminated and control is transferred to the statement, immediately after loop or situation where we want to jump out of the loop instantly without waiting to get back to conditional state.

When break is encountered inside any loop, control automatically passes to the first statement after the loop. This break statement is usually associated with **if** statement.

Example :

```
void main()

{

int j=0;

for(;j<6;j++)

if(j==4)

break;

}
```

Output:

```
0 1 2 3
```

}

Continue statement (key word continue)

Continue statement is used for continuing next iteration of loop after skipping some statement of loop. When it encountered control automatically passes through the beginning of the loop. It is usually associated with the if statement. It is useful when we want to continue the program without executing any part of the program.

The difference between break and continue is, when the break encountered loop is terminated and it transfer to the next statement and when continue is encounter control come back to the beginning position.

In while and do while loop after continue statement control transfer to the test condition and then loop continue where as in, for loop after continue control transferred to the updating expression and condition is tested.

Example:-

```
void main()

{

int n;

for(n=2; n<=9; n++)

{

if(n==4)

continue;

printf("%d", n);

  }

}

Printf("out of loop");

}
```

Output: 2 3 5 6 7 8 9 out of loop

Lecture Note: 9

if statement

Statement execute set of command like when condition is true and its syntax is

```
      If (condition)
   Statement;
```

The statement is executed only when condition is true. If the if statement body is consists of several statement then better to use pair of curly braces. Here in case condition is false then compiler skip the line within the if block.

```
void main()
{
   int n;
    printf  (" enter a number:");
     scanf("%d",&n);
        If (n>10)
      Printf(" number is grater");
    }
```

Output:

```
        Enter a number:12
       Number is greater
```

if…..else ... Statement

it is bidirectional conditional control statement that contains one condition & two possible action. Condition may be true or false, where non-zero value regarded as true & zero value regarded as false. If condition are satisfy true, then a single or block of statement executed otherwise another single or block of statement is executed.

Its syntax is:-

```
if (condition)
 {
 Statement1;
  Statement2;
 }
    else
      {
       Statement1;
       Statement2;
        }
```

Else statement cannot be used without if or no multiple else statement are allowed within one if statement. It means there must be a if statement with in an else statement.

Example:-

/* To check a number is eve or odd */

```
 void main()
{
  int n;
      printf (“enter a number:”);
     sacnf (“%d”, &n);
      If (n%2==0)
         printf (“even number”);
else
         printf(“odd number”);
}
```

```
Output: enter a number:121
odd number
```

Nesting of if …else

When there are another if else statement in if-block or else-block, then it is called nesting of if-else statement.

Syntax is :-

```
        if (condition)
         {
             If (condition)
              Statement1;
         else
                statement2;
           }
                 Statement3;
```

If….else LADDER

In this type of nesting there is an if else statement in every else part except the last part. If condition is false control pass to block where condition is again checked with its if statement.

Syntax is :-

```
         if (condition)
          Statement1;
     else if (condition)
```

```
          statement2;
      else if (condition)
          statement3;
      else
             statement4;
```

This process continue until there is no if statement in the last block. if one of the condition is satisfy the condition other nested "else if" would not executed.

But it has disadvantage over if else statement that, in if else statement whenever the condition is true, other condition are not checked. While in this case, all condition are checked.

<u>Lecture Note: 11</u>

ARRAY

Array is the collection of similar data types or collection of similar entity stored in contiguous memory location. Array of character is a string. Each data item of an array is called an element. And each element is unique and located in separated memory location. Each of elements of an array share a variable but each element having different index no. known as subscript.

An array can be a single dimensional or multi-dimensional and number of subscripts determines its dimension. And number of subscript is always starts with zero. One dimensional array is known as vector and two dimensional arrays are known as matrix.

ADVANTAGES: array variable can store more than one value at a time where other variable can store one value at a time.

Example:

```
int  arr[100];
```

int mark[100];

DECLARATION OF AN ARRAY :

Its syntax is :

Data type array name [size];

```
int arr[100];

int mark[100];

int a[5]={10,20,30,100,5}
```

The declaration of an array tells the compiler that, the data type, name of the array, size of the array and for each element it occupies memory space. Like for int data type, it occupies 2 bytes for each element and for float it occupies 4 byte for each element etc. The size of the array operates the number of elements that can be stored in an array and it may be a int constant or constant int expression.

We can represent individual array as :

```
int ar[5];

 ar[0], ar[1], ar[2], ar[3], ar[4];
```

Symbolic constant can also be used to specify the size of the array as:

```
#define SIZE 10;
```

INITIALIZATION OF AN ARRAY:

After declaration element of local array has garbage value. If it is global or static array then it will be automatically initialize with zero. An explicitly it can be initialize that

Data type array name [size] = {value1, value2, value3…}

Example:

```
in ar[5]={20,60,90, 100,120}
```

}

Array subscript always start from zero which is known as lower bound and upper value is known as upper bound and the last subscript value is one less than the size of array. Subscript can be an expression i.e. integer value. It can be any integer, integer constant, integer variable, integer expression or return value from functional call that yield integer value.

So if i & j are not variable then the valid subscript are

ar [i*7],ar[i*i],ar[i++],ar[3];

The array elements are standing in continuous memory locations and the amount of storage required for hold the element depend in its size & type.

Total size in byte for 1D array is:

Total bytes=size of (data type) * size of array.

Example : if an array declared is:

int [20];

Total byte= 2 * 20 =40 byte.

ACCESSING OF ARRAY ELEMENT:

```
/*Write a program to input values into an array and display them*/

#include<stdio.h>

int main()

{

int arr[5],i;

for(i=0;i<5;i++)

{

printf("enter a value for arr[%d] \n",i);

scanf("%d",&arr[i]);

}
printf("the array elements are: \n");

for (i=0;i<5;i++)

{

printf("%d\t",arr[i]);

}

return 0;

}
```

OUTPUT:

```
Enter a value for arr[0] = 12

Enter a value for arr[1] =45

Enter a value for arr[2] =59

Enter a value for arr[3] =98

Enter a value for arr[4]  =21

The array elements are 12  45  59  98  21
```

Example: From the above example value stored in an array are and occupy its memory addresses 2000, 2002, 2004, 2006, 2008 respectively.

a[0]=12, a[1]=45, a[2]=59, a[3]=98, a[4]=21

ar[0]	ar[1]	ar[2]	ar[3]	ar[4]
12	45	59	98	21
2000	2002	2004	2006	2008

Example 2:

```
     /* Write a program to add 10 array elements */
      #include<stdio.h>
       void main()
{
 int i ;
int arr [10];
int sum=o;
for (i=0; i<=9; i++)
{
printf (“enter the %d element \n”, i+1);
scanf (“%d”, &arr[i]);
}
for (i=0; i<=9; i++)
{
sum = sum + a[i];
}
printf (“the sum of 10 array elements is %d”, sum);
}
```

OUTPUT:

```
Enter a value for arr[0] =5
Enter a value for arr[1] =10
Enter a value for arr[2] =15
Enter a value for arr[3]  =20
Enter a value for arr[4] =25

Enter a value for arr[5] =30

Enter a value for arr[6] =35

Enter a value for arr[7] =40

Enter a value for arr[8] =45

Enter a value for arr[9] =50

Sum = 275
```

while initializing a single dimensional array, it is optional to specify the size of array. If the size is omitted during initialization then the compiler assumes the size of array equal to the number of initializers.

For example:-

```
    int  marks[]={99,78,50,45,67,89};
```

If during the initialization of the number the initializers is less then size of array, then all the remaining elements of array are assigned value zero .

For example:-

```
      int marks[5]={99,78};
```

Here the size of the array is 5 while there are only two initializers so After this initialization, the value of the rest elements are automatically occupied by zeros such as

Marks[0]=99 , Marks[1]=78 , Marks[2]=0, Marks[3]=0, Marks[4]=0

Again if we initialize an array like

```
int array[100]={0};
```

Then the all the element of the array will be initialized to zero. If the number of initializers is more than the size given in brackets then the compiler will show an error.

For example:-

```
 int arr[5]={1,2,3,4,5,6,7,8};//error
```

we cannot copy all the elements of an array to another array by simply assigning it to the other array like, by initializing or declaring as

```
     int a[5] ={1,2,3,4,5};
   int b[5];
   b=a;//not valid
```

(**note**:-here we will have to copy all the elements of array one by one, using for loop.)

Single dimensional arrays and functions

```
/*program to pass array elements to a function*/
#include<stdio.h>
void main()
{
int arr[10],i;
printf("enter the array elements\n");
for(i=0;i<10;i++)
{
scanf("%d",&arr[i]);
check(arr[i]);
}
}
```

```
void check(int num)

{

if(num%2=0)

{

printf(”%d is even \n”,num);

}

else

{

printf(”%d is odd \n”,num);

}

}
```

Lecture Note: 12

Two dimensional arrays

Two dimensional array is known as matrix. The array declaration in both the array i.e.in single dimensional array single subscript is used and in two dimensional array two subscripts are is used.

Its syntax is

Data-type array name[row][column];

Or we can say 2-d array is a collection of 1-D array placed one below the other.

Total no. of elements in 2-D array is calculated as **row*column**

Example:-

int a[2][3];

Total no of elements=row*column is 2*3 =6

It means the matrix consist of 2 rows and 3 columns

For example:-

```
 20    2     7

8      3     15
```

Positions of 2-D array elements in an array are as below

```
00     01     02

10     11     12
```

a [0][0]	a [0][0]	a [0][0]	a [0][0]	a [0][0]	a [0][0]
20	2	7	8	3	15
2000	2002	2004	2006		2008

Accessing 2-d array /processing 2-d arrays

For processing 2-d array, we use two nested for loops. The outer for loop corresponds to the row and the inner for loop corresponds to the column.

For example

int a[4][5];

for reading value:-

```
        for(i=0;i<4;i++)
        {
                for(j=0;j<5;j++)
                {
                        scanf("%d",&a[i][j]);
                }
        }
```

For displaying value:-

```
        for(i=0;i<4;i++)
        {
        for(j=0;j<5;j++)
                {
                        printf("%d",a[i][j]);
                }
        }
```

Initialization of 2-d array:

array can be initialized in a way similar to that of 1-D array. for example:-int

mat[4][3]={11,12,13,14,15,16,17,18,19,20,21,22};

These values are assigned to the elements row wise, so the values of elements after this initialization are

Mat[0][0]=11,	Mat[1][0]=14,	Mat[2][0]=17	Mat[3][0]=20
Mat[0][1]=12,	Mat[1][1]=15,	Mat[2][1]=18	Mat[3][1]=21
Mat[0][2]=13,	Mat[1][2]=16,	Mat[2][2]=19	Mat[3][2]=22

While initializing we can group the elements row wise using inner braces.

for example:-

    int mat[4][3]={{11,12,13},{14,15,16},{17,18,19},{20,21,22}};

And while initializing , it is necessary to mention the 2nd dimension where 1st dimension is optional.

int mat[][3];

int mat[2][3];

int mat[][];

int mat[2][];   invalid

If we **initialize an array** as

    int mat[4][3]={{11},{12,13},{14,15,16},{17}};

Then the compiler will assume its all rest value as 0,which are not defined.

```
Mat[0][0]=11,    Mat[1][0]=12,    Mat[2][0]=14,    Mat[3][0]=17
Mat[0][1]=0,     Mat[1][1]=13,    Mat[2][1]=15     Mat[3][1]=0
Mat[0][2]=0,     Mat[1][2]=0,     Mat[2][2]=16, Mat[3][2]=0
```

In memory map whether it is 1-D or 2-D, elements are stored in one contiguous manner.

We can also give the size of the 2-D array by using symbolic constant

Such as

    #define ROW 2;

#define COLUMN 3;

int mat[ROW][COLUMN];

String

Array of character is called a string. It is always terminated by the NULL character. String is a one dimensional array of character.

We can initialize the string as

char name[]={‘j’,’o’,’h’,’n’,’\o’};

Here each character occupies 1 byte of memory and last character is always NULL character. Where ’\o’ and 0 (zero) are not same, where **ASCII** value of ‘\o’ is 0 and ASCII value of 0 is 48. Array elements of character array are also stored in contiguous memory allocation.

From the above we can represent as;

J	o	h	N	‘\o’

The terminating NULL is important because it is only the way that the function that work with string can know, where string end.

String can also be **initialized** as;

char name[]=”John”;

Here the NULL character is not necessary and the compiler will assume it automatically.

String constant (string literal)

A string constant is a set of character that enclosed within the double quotes and is also called a literal. Whenever a string constant is written anywhere in a program it is stored somewhere in a memory as an array of characters terminated by a NULL character ('\o').

Example – "m"

"Tajmahal"

"My age is %d and height is %f\n"

The string constant itself becomes a pointer to the first character in array.

Example-char crr[20]="Taj mahal";

1000	1001	1002	1003	1004	1005	1006	1007	100	1009
T	a	j		M	A	H	a	l	\o

It is called base address.

Lecture Note: 13

String library function

There are several string library functions used to manipulate string and the prototypes for these functions are in header file "string.h". Several string functions are

strlen()

This function return the length of the string. i.e. the number of characters in the string excluding the terminating NULL character.

It accepts a single argument which is pointer to the first character of the string.

For example-

strlen(“suresh”);

It return the value 6.

In array version to calculate legnth:-

```
        int str(char str[])

{

        int i=0;

        while(str[i]!=’\o’)

        {

                i++;

        }

        return i;

}
```

Example:-

```
        #include<stdio.h>

        #include<string.h>

        void main()

{

        char str[50];

        print(”Enter a string:”);
        gets(str);

        printf(“Length of the string is %d\n”,strlen(str));

}
```

Output:

```
Enter a string: C in Depth

Length of the string is 8
```

strcmp()

This function is used to compare two strings. If the two string match, strcmp() return a value 0 otherwise it return a non-zero value. It compare the strings character by character and the comparison stops when the end of the string is reached or the corresponding characters in the two string are not same.

```
        strcmp(s1,s2)

        return a value:

                <0 when s1<s2

                =0 when s1=s2

                >0 when s1>s2
```

The exact value returned in case of dissimilar strings is not defined. We only know that if s1<s2 then a negative value will be returned and if s1>s2 then a positive value will be returned.

For example:

```
/*String comparison… ........................... */
#include<stdio.h>
#include<string.h>
void main()
{
        char str1[10],str2[10];
        printf("Enter two strings:");
        gets(str1);
        gets(str2);
        if(strcmp(str1,str2)==0)
{
        printf("String are same\n");
}
else
{
        printf("String are not same\n");
}
}
```

strcpy()

This function is used to copying one string to another string. The function strcpy(str1,str2) copies str2 to str1 including the NULL character. Here str2 is the source string and str1 is the destination string.

The old content of the destination string str1 are lost. The function returns a pointer to destination string str1.

Example:-

```
        #include<stdio.h>
        #include<string.h>
        void main()
{
        char str1[10],str2[10];
        printf(“Enter a string:”);
        scanf(“%s”,str2);
        strcpy(str1,str2);
        printf(“First string:%s\t\tSecond string:%s\n”,str1,str2);
        strcpy(str,”Delhi”);
        strcpy(str2,”Bangalore”);
printf(“First string :%s\t\tSecond string:%s”,str1,str2);
```

strcat()

This function is used to append a copy of a string at the end of the other string. If the first string is “”Purva” and second string is “Belmont” then after using this function the string becomes “PusvaBelmont”. The NULL character from str1 is moved and str2 is added at the end of str1. The 2nd string str2 remains unaffected. A pointer to the first string str1 is returned by the function.

Example:-

```
        #include<stdio.h>

        #include<string.h>

        void main()

{

        char str1[20],str[20];

        printf(“Enter two strings:”);

        gets(str1);

        gets(str2);

        strcat(str1,str2);

        printf(“First string:%s\t second string:%s\n”,str1,str2);

        strcat(str1,”-one”);

        printf(“Now first string is %s\n”,str1);

}
```

```
        Output

        Enter two strings: data

        Base

        First string: database second string: database

`       Now first string is: database-one
```

FUNCTION

A function is a self contained block of codes or sub programs with a set of statements that perform some specific task or coherent task when it is called.

It is something like to hiring a person to do some specific task like, every six months servicing a bike and hand over to it.

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

There are basically two types of function those are

1. Library function

2. User defined function

The user defined functions defined by the user according to its requirement

System defined function can’t be modified, it can only read and can be used. These function are supplied with every C compiler

Source of these library function are pre complied and only object code get used by the user by linking to the code by linker

Here in system defined function description:

Function definition : predefined, precompiled, stored in the library

Function declaration : In header file with or function prototype.

Function call : By the programmer

User defined function

Syntax:-

```
    Return type      name of function (type 1 arg 1, type2 arg2, type3 arg3)
     Return type       function name      argument list of the above syntax
```

So when user gets his own function three thing he has to know, these are.

Function declaration

Function definition

Function call

These three things are represented like

```
     int function(int, int, int);    /*function declaration*/
     main()    /* calling function*/
      {
     function(arg1,arg2,arg3);
      }
  int function(type 1 arg 1,type2 arg2,type3, arg3)   /*function definition/*
  {
  Local variable declaration;
 Statement;
 Return value;
  }
```

Function declaration:-

Function declaration is also known as function prototype. It inform the compiler about three thing, those are name of the function, number and type of argument received by the function and the type of value returned by the function.

While declaring the name of the argument is optional and the function prototype always terminated by the semicolon.

Function definition:-

Function definition consists of the whole description and code of the function.

It tells about what function is doing what are its inputs and what are its out put

It consists of two parts function header and function body

Syntax:-

```
return type function(type 1 arg1, type2 arg2, type3  arg3)  /*function header*/
  {
    Local variable declaration;
    Statement 1;
    Statement 2;
    Return value
  }
```

The return type denotes the type of the value that function will return and it is optional and if it is omitted, it is assumed to be int by default. The body of the function is the compound statements or block which consists of local variable declaration statement and optional return statement.

The local variable declared inside a function is local to that function only. It can't be used anywhere in the program and its existence is only within this function.

The arguments of the function **definition** are known as **formal arguments**.

Function Call

When the function get called by the calling function then that is called, function call. The compiler execute these functions when the semicolon is followed by the function name.

Example:-

```
  function(arg1,arg2,arg3);
```

The argument that are used inside the function call are called **actual argument**

Ex:-

```
   int S=sum(a, b);                    //actual arguments
```

Actual argument

The arguments which are mentioned or used inside the function call is knows as actual argument and these are the original values and copy of these are actually sent to the called function

It can be written as constant, expression or any function call like

```
Function (x);

 Function (20, 30);

Function (a*b, c*d);

Function(2,3,sum(a, b));
```

Formal Arguments

The arguments which are mentioned in function definition are called formal arguments or dummy arguments.

These arguments are used to just hold the copied of the values that are sent by the calling function through the function call.

These arguments are like other local variables which are created when the function call starts and destroyed when the function ends.

The basic difference between the formal argument and the actual argument are

1) The formal argument are declared inside the parenthesis where as the local variable declared at the beginning of the function block.

2). The **formal argument** are automatically initialized when the copy of actual arguments are passed while other local variable are assigned values through the statements.

Order number and type of actual arguments in the function call should be match with the order number and type of the formal arguments.

Return type

It is used to return value to the calling function. It can be used in two way as

```
              return
      Or      return(expression);
       Ex:-    return (a);
              return (a*b);
             return (a*b+c);
```

Here the 1st return statement used to terminate the function without returning any value

```
Ex:-   /*summation of two values*/
     int sum (int a1, int a2);
    main()
```

```
{
   int a,b;
    printf("enter two no");
    scanf("%d%d",&a,&b);
    int S=sum(a,b);
     printf("summation is = %d",s);
      }
     int sum(intx1,int y1)
      {
    int z=x1+y1;
     Return z;
     }
```

Advantage of function

By using function large and difficult program can be divided in to sub programs and solved. When we want to perform some task repeatedly or some code is to be used more than once at different place in the program, then function avoids this repeatition or rewritten over and over.

Due to reducing size, modular function it is easy to modify and test

Notes:-

C program is a collection of one or more function.

A function is get called when function is followed by the semicolon.

A function is defined when a function name followed by a pair of curly braces

Any function can be called by another function even main() can be called by other function.

```
main()
{

function1()

}

function1()

{

Statement;

function2;

}

function 2()

{


}
```

So every function in a program must be called directly or indirectly by the main() function. A function can be called any number of times.

A function can call itself again and again and this process is called **recursion**.

A function can be called from other function **but** a function can’t be defined in another function

Category of Function based on argument and return type

i) Function with no argument & no return value

Function that have no argument and no return value is written as:-

```
 void  function(void);
main()
 {
void  function()
{
Statement;
}
```

Example:-

```
void  me();
    main()
  {
     me();
    printf("in  main");
   }
  void   me()
 {
    printf("come  on");
 }
```

Output: come on

In main

ii) Function with no argument but return value

Syntax:-

```
  int   fun(void);
  main()
 {
      int r;
      r=fun();
   }
      int   fun()
    {
    reurn(exp);
    }
```

Example:-

```
 int   sum();
 main()
 {
 int   b=sum();
 printf(“entered   %d\n, b”);
  }
int   sum()
   {
 int  a,b,s;
```

```
     s=a+b;
    return  s;
       }
```

Here called function is independent and are initialized. The values aren’t passed by the calling function .Here the calling function and called function are communicated partly with each other.

iii) function with argument but no return value

Here the function have argument so the calling function send data to the called function but called function dose n’t return value.

Syntax:-

```
            void   fun (int,int);
            main()
           {
          int (a,b);
         }
  void   fun(int x, int y);
   {
  Statement;
   }
```

Here the result obtained by the called function.

iv) function with argument and return value

Here the calling function has the argument to pass to the called function and the called function returned value to the calling function.

Syntax:-

```
          fun(int,int);
           main()
        {
            int r=fun(a,b);
         }
              int  fun(intx,inty)
         {
                     return(exp);

         }
```

Example:

```
    main()
   {
    int fun(int);
    int  a,num;
    printf("enter value:\n");
    scanf("%d",&a)
```

```
int num=fun(a);
  }
 int  fun(int x)
 {
   ++x;
            return x;
  }
```

Call by value and call by reference

There are two way through which we can pass the arguments to the function suchas **call by** value and **call by reference**.

1. Call by value

In the call by value copy of the actual argument is passed to the formal argument and the operation is done on formal argument.

When the function is called by 'call by value' method, it doesn't affect content of the actual argument.

Changes made to formal argument are local to block of called function so when the control back to calling function the changes made is vanish.

Example:-

```
  main()
 {
 int x,y;
change(int,int);
```

```
          printf("enter two values:\n");
           scanf("%d%d",&x,&y);
           change(x ,y);
       printf("value of x=%d and y=%d\n",x ,y);
          }
       change(int a,int b);
     {
       int k;
       k=a;
       a=b;
       b=k;
     }
```

```
Output: enter two values: 12
 23
Value of x=12 and y=23
```

2. Call by reference

Instead of passing the value of variable, address or reference is passed and the function operate on address of the variable rather than value.

Here formal argument is alter to the actual argument, it means formal arguments calls the actual arguments.

Example:-

```
        void main()
```

```
{
        int a,b;
        change(int *,int*);
        printf("enter two values:\n");
        scanf("%d%d",&a,&b);
        change(&a,&b);
        printf("after  changing two value of a=%d and b=%d\n:"a,b);
}
        change(int *a, int *b)
        {
        int k;
          k=*a;
        *a=*b;
      *b= k;
printf("value in this function  a=%d and b=%d\n",*a,*b);
}
```

```
Output: enter two values: 12

32

Value in this function a=32  and b=12

After changing two value of  a=32 and b=12
```

So here instead of passing value of the variable, directly passing address of the variables. Formal argument directly access the value and swapping is possible even after calling a function.

Lecture Note: 17

Local, Global and Static variable

Local variable:-

variables that are defined with in a body of function or block. The local variables can be used only in that function or block in which they are declared. Same variables may be used in different functions such as

```
function()
{
        int a,b;
        function 1();
}
function2 ()
{
        int a=0;
        b=20;
}
```

Global variable:-

The variables that are defined outside of the function is called global variable. All functions in the program can access and modify global variables. Global variables are automatically initialized at the time of initialization.

Example:

```
        #include<stdio.h>

        void function(void);

        void function1(void);

        void function2(void);

        int a, b=20;

void main()

{

        printf("inside main a=%d,b=%d \n",a,b);

function();

function1();

function2();

        }

         function()

        {

                Prinf("inside function a=%d,b=%d\n",a,b);

        }

        function 1()

        {
```

```
        prinf("inside function a=%d,b=%d\n",a,b);
    }
    function 2()
    {
        prinf("inside function a=%d,b=%d\n",a,);
    }
```

Static variables: static variables are declared by writing the key word static.

-syntax:-

```
    static data type variable name;
    static int a;
```

-the static variables initialized only once and it retain between the function call. If its variable is not initialized, then it is automatically initialized to zero.

Example:

```
    void fun1(void);
    void fun2(void);
    void main()
    {
        fun1();
        fun2();
    }
    void fun1()
    {
        int a=10, static int b=2;

        printf("a=%d, b=%d",a,b);

        a++;

        b++;

    }
```

```
Output:a= 10  b=  2

            a=10   b= 3
```

Recursion

When function calls itself (inside function body) again and again then it is called as recursive function. In recursion calling function and called function are same. It is powerful technique of writing complicated algorithm in easiest way. According to recursion problem is defined in term of itself. Here statement with in body of the function calls the same function and same times it is called as circular definition. In other words recursion is the process of defining something in form of itself.

Syntax:

```
    main ()

    {

            rec(); /*function call*/

            rec();

            rec();
```

Ex:-

```
    /*calculate factorial of a no.using recursion*/

    int fact(int);

    void main()
```

```
        {
                int num;
                printf(“enter a number”);
                scanf(“%d”,&num);
                f=fact(num);
                printf(“factorial is =%d\n”f);
        }
        fact (int num)
        {
                If  (num==0||num==1)
return 1;
else
return(num*fact(num-1));
        }
```

POINTER

A pointer is a variable that store memory address or that contains address of another variable where addresses are the location number always contains whole number. So, pointer contain always the whole number. It is called pointer because it points to a particular location in memory by storing address of that location.

Syntax-

Data type *pointer name;

Here * before pointer indicate the compiler that variable declared as a pointer.

e.g.

```
int *p1; //pointer to integer type

float *p2; //pointer to float type

char *p3; //pointer to character type
```

When pointer declared, it contains garbage value i.e. it may point any value in the memory.

Two operators are used in the pointer i.e. **address operator(&)** and **indirection operator or dereference operator (*).**

Indirection operator gives the values stored at a particular address.

Address operator cannot be used in any constant or any expression.

Example:

```
    void main()
    {
     int i=105;
     int *p;
     p=&i;
t
printf("value of i=%d",*p);
printf("value of i=%d",*/&i);
printf("address of i=%d",&i);
printf("address of i=%d",p);
printf("address of p=%u",&p);
}
```

Pointer Expression

Pointer assignment

```
int i=10;
int *p=&i;//value assigning to the pointer
```

Here declaration tells the compiler that P will be used to store the address of integer value or in other word P is a pointer to an integer and *p reads the **value at the address contain in p.**

P++;

printf("value of p=%d");

We can assign value of 1 pointer variable to other when their base type and data type is same or both the pointer points to the same variable as in the array.

Int *p1,*p2;

P1=&a[1];

P2=&a[3];

We can assign constant 0 to a pointer of any type for that symbolic constant '**NULL**' is used such as

*p=NULL;

It means pointer doesn't point to any valid memory location.

Pointer Arithmetic

Pointer arithmetic is different from ordinary arithmetic and it is perform relative to the data type(base type of a pointer).

Example:-

If integer pointer contain address of 2000 on incrementing we get address of 2002 instead of 2001, because, size of the integer is of 2 bytes.

Note:-

When we move a pointer, somewhere else in memory by incrementing or decrement or adding or subtracting integer, it is not necessary that, pointer still pointer to a variable of same data, because, memory allocation to the variable are done by the compiler.

But in case of array it is possible, since there data are stored in a consecutive manner.

Ex:-

```
void main( )

{

static int a[ ]={20,30,105,82,97,72,66,102};

int *p,*p1;

P=&a[1];

P1=&a[6];

printf("%d",*p1-*p);

printf("%d",p1-p);

}
```

Arithmetic operation never perform on pointer are:

addition, multiplication and division of two pointer.

multiplication between the pointer by any number.

division of pointer by any number

-add of float or double value to the pointer.

Operation performed in pointer are:-

/* Addition of a number through pointer */

Example

```
int i=100;

int *p;
```

p=&i;

p=p+2;

p=p+3;

p=p+9;

ii /* Subtraction of a number from a pointer'*/

Ex:-

int i=22;

*p1=&a;

p1=p1-10;

p1=p1-2;

iii- Subtraction of one pointer to another is possible when pointer variable point to an element of same type such as an array.

Ex:-

in tar[ ]={2,3,4,5,6,7};

int *ptr1,*ptr1;

ptr1=&a[3]; //2000+4

ptr2=&a[6]; //2000+6

Pointer to pointer

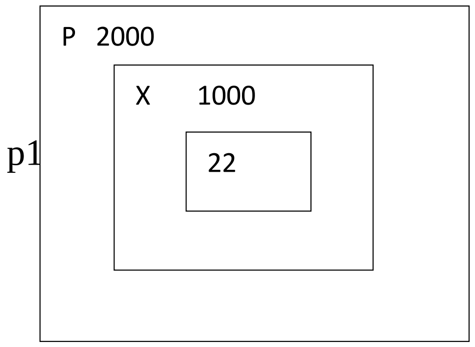
Addition of pointer variable stored in some other variable is called pointer to pointer variable.

Or

Pointer within another pointer is called pointer to pointer.

Syntax:-

```
        Data type **p;

        int x=22;

        int *p=&x;

        int **p1=&p;

printf("value of x=%d",x);

printf("value of x=%d",*p);

printf("value of x=%d",*&x);

printf("value of x=%d",**p1);

printf("value of p=%u",&p);

printf("address of p=%u",p1);

printf("address of x=%u",p);

printf("address of p1=%u",&p1);

printf("value of p=%u",p);

printf("value of p=%u",&x);
```



*Under revision

3000

Pointer vs array

Example :-

```
   void main()
{
static char arr[]=”Rama”;
char*p=”Rama”;
printf(“%s%s”, arr, p);
```

In the above example, at the first time printf( ), print the same value array and pointer.

Here array arr, as **pointer to character** and **p act as a pointer to array of character .** When we are trying to increase the value of arr it would give the error because its known to compiler about an array and its base address which is always printed to base address is known as constant pointer and the base address of array which is not allowed by the compiler.

```
printf(“size of (p)”,size of (ar));
size of (p)          2/4 bytes
size of(ar)            5 bytes
```

Sructure

It is the collection of dissimilar data types or heterogenous data types grouped together. It means the data types may or may not be of same type.

Structure declaration-

```
struct tagname
{
Data type member1;
Data type member2;
Data type member3;
………
………
Data type member n;
};
```

OR

```
struct
{
Data type member1;
Data type member2;
```

Data type member3;

………

………

Data type member n;

};

OR

struct tagname

{

struct element 1;

struct element 2;

struct element 3;

………

………

struct element n;

};
Structure variable declaration;

struct student

{

int age; char

name[20]; char

branch[20];

```
}; struct student s;
```

Initialization of structure variable-

Like primary variables structure variables can also be initialized when they are declared. Structure templates can be defined locally or globally. If it is local it can be used within that function. If it is global it can be used by all other functions of the program.

We cant initialize structure members while defining the structure

```
struct student
{
        int age=20;
char name[20]=”sona”;
}s1;
```

The above is **invalid.**

A structure can be initialized as

```
struct student
{
        int age,roll;
char name[20];
} struct student s1={16,101,”sona”};
  struct student s2={17,102,”rupa”};
```

If initialiser is less than no.of structure variable, automatically rest values are taken as zero.

Lecture Note: 20

Accessing structure elements-

Dot operator is used to access the structure elements. Its associativety is from left to right.

structure variable ;

s1.name[];

s1.roll;

s1.age;

Elements of structure are stored in contiguous memory locations. Value of structure variable can be assigned to another structure variable of same type using assignment operator.

Example:

```
#include<stdio.h>

#include<conio.h>

void main()

{

int roll, age;

char branch;

} s1,s2;

printf("\n enter roll, age, branch=");

scanf("%d %d %c", &s1.roll, &s1.age, &s1.branch);

s2.roll=s1.roll;

printf(" students details=\n");

printf("%d %d %c", s1.roll, s1.age, s1.branch);
printf("%d", s2.roll);

}
```

Unary, relational, arithmetic, bitwise operators are not allowed within structure variables.

Size of structure-

Size of structure can be found out using sizeof() operator with structure variable name or tag name with keyword.

```
sizeof(struct student); or

sizeof(s1);

sizeof(s2);
```

Size of structure is different in different machines. So size of whole structure may not be equal to sum of size of its members.

Array of structures

When database of any element is used in huge amount, we prefer Array of structures.

Example: suppose we want to maintain data base of 200 students, Array of structures is used.

```
#include<stdio.h>

#include<string.h>

struct student

{
```

```
char name[30];

char branch[25];

int roll;

};

void main()

{

struct student s[200];

int i;

s[i].roll=i+1;

printf("\nEnter information of students:");

for(i=0;i<200;i++)

{

printf("\nEnter the roll no:%d\n",s[i].roll);

printf("\nEnter the name:");

scanf("%s",s[i].name);

printf("\nEnter the branch:");

scanf("%s",s[i].branch);

printf("\n");

}

printf("\nDisplaying information of students:\n\n");

for(i=0;i<200;i++)

{

printf("\n\nInformation for roll no%d:\n",i+1);
printf("\nName:");
puts(s[i].name);
printf("\nBranch:");
puts(s[i].branch);
}
}
```

Lecture Note: 21

UNION

Union is derived data type contains collection of different data type or dissimilar elements. All definition declaration of union variable and accessing member is similar to structure, but instead of keyword struct the keyword union is used, the main difference between union and structure is;

Each member of structure occupy the memory location, but in the unions members share memory. Union is used for saving memory and concept is useful when it is not necessary to use all members of union at a time.

Where union offers a memory treated as variable of one type on one occasion where (struct), it read number of different variables stored at different place of memory.

Syntax of union:

```
union student

{

datatype member1;

datatype member2;

};
```

Like structure variable, union variable can be declared with definition or separately such as

```
union union name

{

Datatype member1;

}var1;
```

Example:-   union student s;

Union members can also be accessed by the dot operator with union variable and if we have pointer to union then member can be accessed by using (arrow) operator as with structure.

Example:- struct student

```
struct student

{

int i;

char  ch[10];

};struct student s;
```

Here datatype/member structure occupy 12 byte of location is memory, where as in the union side it occupy only 10 byte.

Lecture Note: 22

Dynamic memory Allocation

The process of allocating memory at the time of execution or at the runtime, is called dynamic memory location.

Two types of problem may occur in static memory allocation.

If number of values to be stored is less than the size of memory, there would be wastage of memory.

If we would want to store more values by increase in size during the execution on assigned size then it fails.

Allocation and release of memory space can be done with the help of some library function called dynamic memory allocation function. These library function are called as **dynamic memory allocation function.** These library function prototype are found in the header file, “alloc.h” where it has defined.

Function take memory from memory area is called heap and release when not required.

Pointer has important role in the dynamic memory allocation to allocate memory.

malloc():

This function use to allocate memory during run time, its declaration is void*malloc(size);

malloc ()

returns the pointer to the 1st byte and allocate memory, and its return type is void, which can be type cast such as:

int *p=(datatype*)malloc(size)

If memory location is successful, it returns the address of the memory chunk that was allocated and it returns null on unsuccessful and from the above declaration a pointer of type**(datatype)** and size in byte.

And **datatype** pointer used to typecast the pointer returned by malloc and this typecasting is necessary since, malloc() by default returns a pointer to void.

Example int*p=(int*)malloc(10);

So, from the above pointer p, allocated IO contigious memory space address of 1st byte and is stored in the variable.

We can also use, the size of operator to specify the the size, such as *p=(int*)malloc(5*size of int) Here, 5 is the no. of data.

Moreover , it returns null, if no sufficient memory available , we should always check the malloc return such as, **if(p==null)**

printf("not sufficient memory");

Example:

```
/*calculate the average of mark*/

void main()

{

int n , avg,i,*p,sum=0;
printf("enter the no. of marks ”);

scanf(“%d”,&n);

p=(int *)malloc(n*size(int));

if(p==null)

printf(“not sufficient”);

exit();

}

for(i=0;i<n;i++)

scanf(“%d”,(p+i));

for(i=0;i<n;i++)

Printf(“%d”,*(p+i));

sum=sum+*p;

avg=sum/n;

printf(“avg=%d”,avg);
```

Lecture Note: 23

calloc()

Similar to malloc only difference is that calloc function use to allocate multiple block of memory .

two arguments are there

1st argument specify number of blocks
2nd argument specify size of each block.

Example:-

```
        int *p= (int*) calloc(5, 2);
int*p=(int *)calloc(5, size of (int));
```

Another difference between malloc and calloc is by default memory allocated by malloc contains garbage value, where as memory allocated by calloc is initialised by zero(but this initialisation) is not reliable.

realloc()

The function realloc use to change the size of the memory block and it alter the size of the memory block without loosing the old data, it is called reallocation of memory.

It takes two argument such as;

```
int *ptr=(int *)malloc(size);
int*p=(int *)realloc(ptr, new size);
```

The new size allocated may be larger or smaller.

If new size is larger than the old size, then old data is not lost and newly allocated bytes are uninitialized. If old address is not sufficient then starting address contained in pointer may be changed and this reallocation function moves content of old block into the new block and data on the old block is not lost.

Example:

```
#include<stdio.h>

#include<alloc.h>void

main()

        int i,*p;
        p=(int*)malloc(5*size of (int));

        if(p==null)

        {
        printf("space not available");exit();
        printf("enter 5 integer");

        for(i=0;i<5;i++)

        {

        scanf("%d",(p+i));

        int*ptr=(int*)realloc(9*size of (int) );

        if(ptr==null)

        {

        printf("not available");

        exit();

        }
        printf("enter 4 more integer");
        for(i=5;i<9;i++)
         scanf("%d",(p+i));
         for(i=0;i<9;i++)
        printf("%d",*(p+i));

        }
```

LECTURE NOTE-24

Introduction to Data Structure

In computer terms, a data structure is a Specific way to store and organize data in a computer's memory so that these data can be used efficiently later. Data may be arranged in many different ways such as the logical or mathematical model for a particular organization of data is termed as a data structure. The variety of a particular data model depends on the two factors -

- Firstly, it must be loaded enough in structure to reflect the actual relationships of the data with the real world object.
- Secondly, the formation should be simple enough so that anyone can efficiently process the data each time it is necessary.

Categories of Data Structure:

The data structure can be sub divided into major types:

- Linear Data Structure
- Non-linear Data Structure

Linear Data Structure:

A data structure is said to be linear if its elements combine to form any specific order. There are basically two techniques of representing such linear structure within memory.

- First way is to provide the linear relationships among all the elements represented by means of linear memory location. These linear structures are termed as arrays.
- The second technique is to provide the linear relationship among all the elements represented by using the concept of pointers or links. These linear structures are termed as linked lists.

The common examples of linear data structure are:

- Arrays
- Queues
- Stacks
- Linked lists

Non linear Data Structure:

This structure is mostly used for representing data that contains a hierarchical relationship among various elements.

Examples of Non Linear Data Structures are listed below:

- Graphs
- family of trees and
- table of contents

Tree: In this case, data often contain a hierarchical relationship among various elements. The data structure that reflects this relationship is termed as rooted treegraph or a tree.

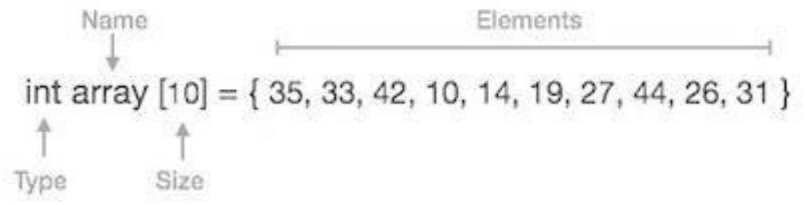
Graph: In this case, data sometimes hold a relationship between the pairs of elements which is not necessarily following the hierarchical structure. Such data structure is termed as a Graph.

Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of Array.

- **Element** − Each item stored in an array is called an element.
- **Index** − Each location of an element in an array has a numerical index, which is used to identify the element.

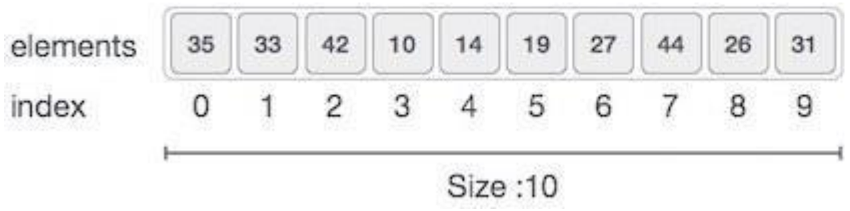
Array Representation:(Storage structure)

Arrays can be declared in various ways in different languages. For illustration, let's



take C array declaration.

Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.



As per the above illustration, following are the important points to be considered.

- Index starts with 0.
- Array length is 10 which means it can store 10 elements.
- Each element can be accessed via its index. For example, we can fetch anelement at index 6 as 9.

Basic Operations

Following are the basic operations supported by an array.

- **Traverse** − print all the array elements one by one.
- **Insertion** − Adds an element at the given index.
- **Deletion** − Deletes an element at the given index.
- **Search** − Searches an element using the given index or by the value.
- **Update** − Updates an element at the given index.

In C, when an array is initialized with size, then it assigns defaults values to its elements in following order

Data Type	Default Value
bool	false

char	0
int	0
float	0.0
double	0.0f
void	
wchar_t	0

Insertion Operation

Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.

Here, we see a practical implementation of insertion operation, where we add data at the end of the array −

Algorithm

Let **LA** be a Linear Array (unordered) with **N** elements and **K** is a positive integer suchthat **K<=N**. Following is the algorithm where ITEM is inserted into the K^th^ position of LA

−

```
1. Start
2. Set J = N
3. Set N = N+1
4. Repeat steps 5 and 6 while J >= K
5. Set LA[J+1] = LA[J]
6. Set J = J-1
7. Set LA[K] = ITEM
```

Example

Following is the implementation of the above algorithm −

```
#include <stdio.h>

main() {
  int LA[] = {1,3,5,7,8};
  int item = 10, k = 3, n =
  5;int i = 0, j = n;
  printf("The original array elements are
  :\n");for(i = 0; i<n; i++) {
    printf("LA[%d] = %d \n", i, LA[i]);
```

```
n = n + 1;
while( j >=
k) {
  LA[j+1] = LA[j];
  j = j - 1;
}
LA[k] = item;
printf("The array elements after insertion
:\n");for(i = 0; i<n; i++) {
  printf("LA[%d] = %d \n", i, LA[i]);
```

When we compile and execute the above program, it produces the following result −

Output

```
The original array elements
are :LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after
insertion :LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 10
LA[4] = 7
```

Deletion Operation

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to delete an element available at the Kth position of LA.

```
1. Start
2. Set J = K
3. Repeat steps 4 and 5 while J < N
4. Set LA[J] = LA[J + 1]
5. Set J = J+1
6. Set N = N-1
```

Example

Following is the implementation of the above algorithm −

```
#include <stdio.h>

void main() {
   int LA[] = {1,3,5,7,8};
   int k = 3, n =
   5;int i, j;
     printf("The original array elements are
   :\n");for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }

   j = k;
   while( j < n) {
      LA[j-1] =
      LA[j];
      j = j + 1;
   }
   n = n -1;
     printf("The array elements after deletion
   :\n");for(i = 0; i<n; i++) {
```

When we compile and execute the above program, it produces the following result −

Output

```
The original array elements
are :LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after
deletion :LA[0] = 1
LA[1] = 3
LA[2] = 7
```

Search Operation

You can perform a search for an array element based on its value or its index.

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to find an element with a value of ITEM using sequential search.

```
1. Start
2. Set J = 0
3. Repeat steps 4 and 5 while J < N
4. IF LA[J] is equal ITEM THEN GOTO STEP 6
5. Set J = J +1
6. PRINT J, ITEM
```

Example

Following is the implementation of the above algorithm −

```
#include <stdio.h>

void main() {
   int LA[] = {1,3,5,7,8};
   int item = 5, n =
   5;int i = 0, j = 0;
      printf("The original array elements are
         :\n");for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
     while( j < n){
      if( LA[j] ==
         item ) {break;
      }
     j = j + 1;
   }
```

When we compile and execute the above program, it produces the following result −

Output

```
The original array elements
are :LA[0] = 1
LA[1] = 3
LA[2] = 5
```

```
LA[3] = 7
LA[4] = 8
Found element 5 at position 3
```

Update Operation

Update operation refers to updating an existing element from the array at a given index.

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to update an element available at the Kth position of LA.

```
1. Start
2. Set LA[K-1] = ITEM
3. Stop
```

Example

Following is the implementation of the above algorithm –

```
#include <stdio.h>

void main() {
  int LA[] = {1,3,5,7,8};
  int k = 3, n = 5, item =
  10;int i, j;
    printf("The original array elements are
  :\n");for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
  }

  LA[k-1] = item;
 printf("The array elements after updation
  :\n");for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
```

When we compile and execute the above program, it produces the following result –

Output

```
The original array elements
are :LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
```

```
The array elements after
updation :LA[0] = 1
LA[1] = 3
LA[2] = 10
LA[3] = 7
LA[4] = 8
```

Lecture-26

Sparse Matrix and its representations

A matrix is a two-dimensional data object made of m rows and n columns, therefore having total m x n values. If most of the elements of the matrix have **0 value**, then it is called a sparse matrix.

Why to use Sparse Matrix instead of simple matrix ?

- **Storage:** There are lesser non-zero elements than zeros and thus lessermemory can be used to store only those elements.
- **Computing time:** Computing time can be saved by logically designing a data structure traversing only non-zero elements..

Example:

0 0 3 0 4

0 0 5 7 0

0 0 0 0 0

0 2 6 0 0

Representing a sparse matrix by a 2D array leads to wastage of lots of memory as zeroes in the matrix are of no use in most of the cases. So, instead of storing zeroes with non-zero elements, we only store non-zero elements. This means storing non-zero elements with **triples- (Row, Column, value).**

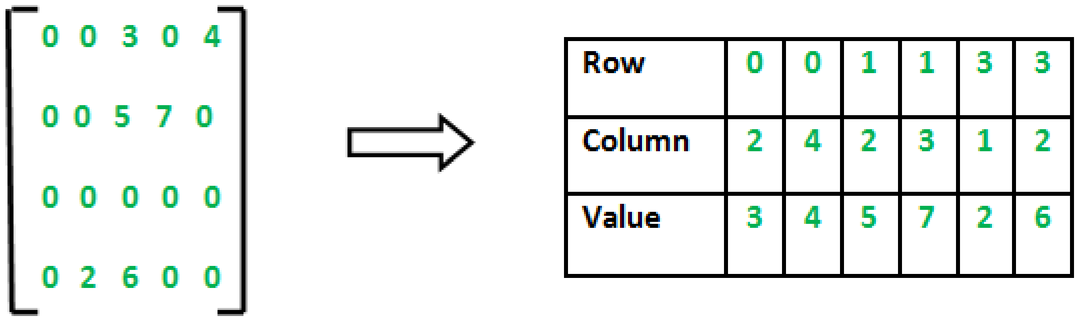
Sparse Matrix Representations can be done in many ways following are two common representations:

1. Array representation
2. Linked list representation

Method 1: Using Arrays

```
#include<stdio.h>
 int main()
{
   // Assume 4x5 sparse
   matrixint
   sparseMatrix[4][5] =
   {
      {0 , 0 , 3 , 0 , 4 },
      {0 , 0 , 5 , 7 , 0 },
      {0 , 0 , 0 , 0 , 0 },
      {0 , 2 , 6 , 0 , 0 }
   };

   int size = 0;
   for (int i = 0; i < 4;
      i++)
```

```
    for (int j = 0; j < 5;j++)
            if (sparseMatrix[i][j]
                != 0)size++;
     int compactMatrix[3][size];
     // Making of new matrix
    int k = 0;
    for (int i = 0; i < 4;
        i++) for (int j = 0; j
        < 5; j++)
            if (sparseMatrix[i][j] != 0)
            {
                compactMatrix[0][k] = i;
                compactMatrix[1][k] = j;
                compactMatrix[2][k] =
                sparseMatrix[i][j];k++;
            }
     for (int i=0; i<3; i++)
    {
        for (int j=0; j<size; j++)
            printf("%d ",
         compactMatrix[i][j]);
         printf("\n");
    }
    return 0;
}
```



$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix} \Rightarrow$$

Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

STACK

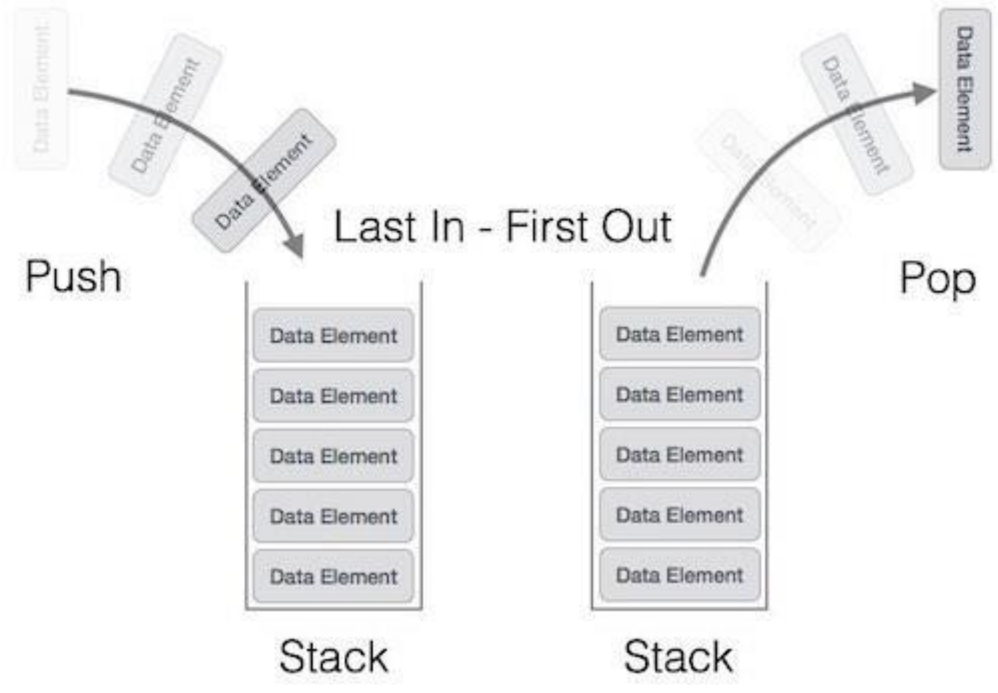
A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example — a deck of cards or a pile of plates, etc.

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

Stack Representation

The following diagram depicts a stack and its operations −



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

Basic OperationsStack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

- **push()** − Pushing (storing) an element on the stack.
- **pop()** − Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose,the following functionality is added to stacks −

- **peek()** − get the top data element of the stack, without removing it.
- **isFull()** − check if stack is full.
- **isEmpty()** − check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

First we should learn about procedures to support stack functions −

peek()

Algorithm of peek() function −

```
begin procedure peek
   return stack[top]
```

Implementation of peek() function in C programming language −

Example

```
int peek() {

   return stack[top];
```

isfull()

Algorithm of isfull() function −

```
begin procedure isfull



   if top equals to MAXSIZE
      return true

   else

      return false
```

Implementation of isfull() function in C programming language −

Example

```
bool isfull() {

  if(top == MAXSIZE)
    return true;

  else
```

isempty()

Algorithm of isempty() function −

```
begin procedure isempty



   if top less than 1
      return true

   else

      return false
```

Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1to determine if the stack is empty. Here's the code −

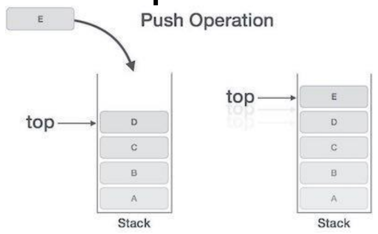
Example

```
bool isempty() {
   if(top == -1)

      return true;
   else

      return false;
```

Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps −

- **Step 1** − Checks if the stack is full.
- **Step 2** − If the stack is full, produces an error and exit.
- **Step 3** − If the stack is not full, increments **top** to point next empty space.
- **Step 4** − Adds data element to the stack location, where top is pointing.
- **Step 5** − Returns success.



If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

Algorithm for PUSH Operation

A simple algorithm for Push operation can be derived as follows − begin procedure push: stack, data

```
if stack is full
   return null
  endif



  top ← top + 1
  stack[top] ← data
```

Implementation of this algorithm in C, is very easy. See the following code −

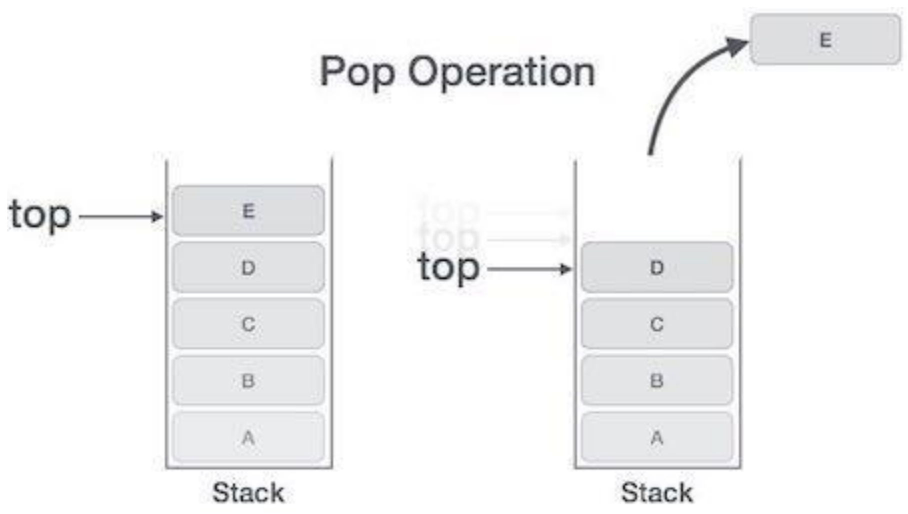
Example

```
void push(int data) {
  if(!isFull()) {

    top = top + 1;
    stack[top] = data;

  } else {

    printf("Could not insert data. Stack is full.\n");
```

Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space. A Pop operation may involve the following steps −

- **Step 1** − Checks if the stack is empty.
- **Step 2** − If the stack is empty, produces an error and exit.
- **Step 3** − If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** − Decreases the value of top by 1.
- **Step 5** − Returns success.



Algorithm for Pop Operation

A simple algorithm for Pop operation can be derived as follows −

```
begin procedure pop: stack



  if stack is empty
    return null

  endif



  data ← stack[top]
```

Implementation of this algorithm in C, is as follows −

Example

```
int pop(int data) {



  if(!isempty()) {

    data = stack[top];
    top = top - 1;
    return data;

  } else {
```

Stack Applications

Three applications of stacks are presented here. These examples are central to many activitiesthat a computer must do and deserve time spent with them.

1. Expression evaluation
2. Backtracking (game playing, finding paths, exhaustive searching)
3. Memory management, run-time environment for nested language features.

Expression evaluation

In particular we will consider arithmetic expressions. Understand that there are boolean and logical expressions that can be evaluated in the same way. Control structures can also be treated similarly in a compiler.

This study of arithmetic expression evaluation is an example of problem solving where you solvea simpler problem and then transform the actual problem to the simpler one.

Aside: *The NP-Complete problem*. There are a set of apparently intractable problems: finding the shortest route in a graph (Traveling Salesman Problem), bin packing, linear programming, etc. that are similar enough that if a polynomial solution is ever found (exponential solutions abound) for one of these problems, then the solution can be applied to all problems.

Infix, Prefix and Postfix Notation

We are accustomed to write arithmetic expressions with the operation between the two operands: a+b or c/d. If we write a+b*c, however, we have to apply precedence rules to avoidthe ambiguous evaluation (add first or multiply first?).

There's no real reason to put the operation between the variables or values. They can just as well precede or follow the operands. You should note the advantage of prefix and postfix: theneed for precedence rules and parentheses are eliminated.

Infix	Prefix	Postfix
a + b	+ a b	a b +
a + b * c	+ a * b c	a b c * +
(a + b) * (c - d)	* + a b - c d	a b + c d - *
b * b - 4 * a * c		
40 - 3 * 5 + 1		

Postfix expressions are easily evaluated with the aid of a stack.

Infix	Prefix	Postfix
a + b	+ a b	a b +

a + b * c	+ a * b c	a b c * +
(a + b) * (c - d)	* + a b - c d	a b + c d - *
b * b - 4 * a * c	- * b b * * 4 a c	b b * 4 a * c * -
40 - 3 * 5 + 1 = 26	+ - 40 * 3 5 1	40 3 5 * - 1 +

Postfix Evaluation Algorithm

Assume we have a string of operands and operators, an informal, by hand process is

1. Scan the expression left to right
2. Skip values or variables (operands)
3. When an operator is found, apply the operation to the preceding two operands
4. Replace the two operands and operator with the calculated value (three symbols arereplaced with one operand)
5. Continue scanning until only a value remains--the result of the expression

The time complexity is O(n) because each operand is scanned once, and each operation is performed once.

A more formal algorithm:

```
create a new stack
while(input stream is not
  empty){token =
  getNextToken(); if(token
  instanceof operand){
     push(token);
  } else if (token instance of
     operator)op2 = pop();
     op1 = pop();
     result = calc(token, op1,
     op2);push(result);
  }
}
return pop();
```

Demonstration with 2 3 4 + * 5 -

Infix transformation to Postfix

This process uses a stack as well. We have to hold information that's expressed inside parentheses while scanning to find the closing ')'. We also have to hold information on operations that are of lower precedence on the stack. The algorithm is:

1. Create an empty stack and an empty postfix output string/stream
2. Scan the infix input string/stream left to right
3. If the current input token is an operand, simply append it to the output string (note the examples above that the operands remain in the same order)
4. If the current input token is an operator, pop off all operators that have equal or higher precedence and append them to the output string; push the operator onto the stack. Theorder of popping is the order in the output.
5. If the current input token is '(', push it onto the stack
6. If the current input token is ')', pop off all operators and append them to the output stringuntil a '(' is popped; discard the '('.
7. If the end of the input string is found, pop all operators and append them to the outputstring.

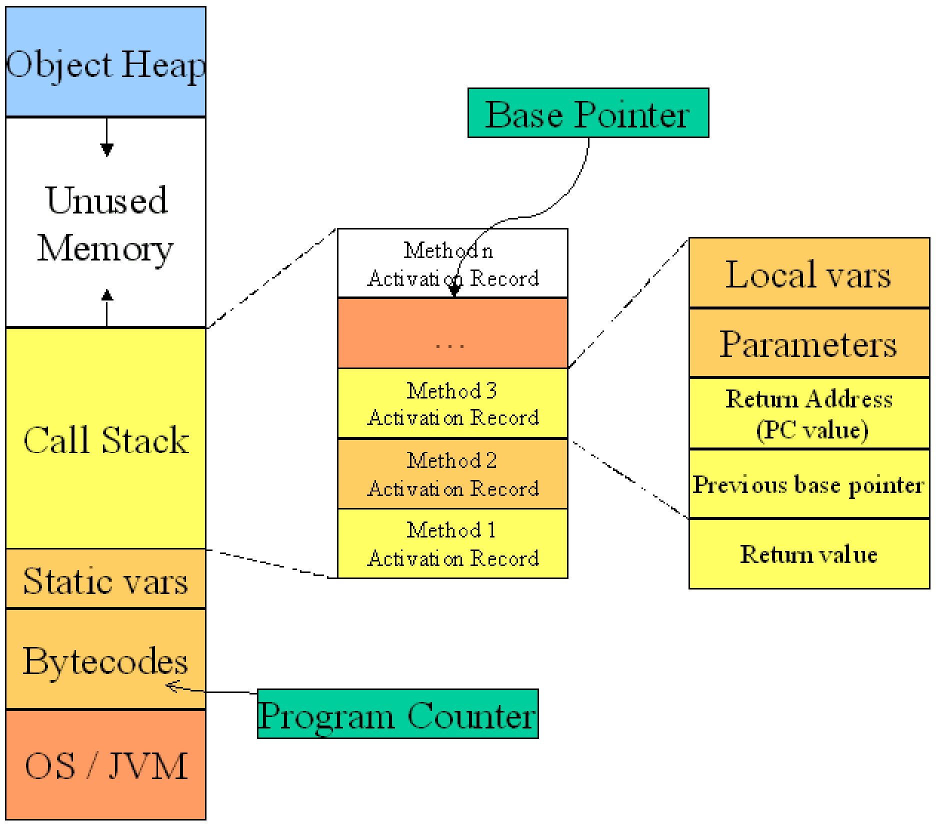
This algorithm doesn't handle errors in the input, although careful analysis of parenthesis or lackof parenthesis could point to such error determination.
Apply the algorithm to the above expressions.

Backtracking
Backtracking is used in algorithms in which there are steps along some path (state) from somestarting point to some goal.

- Find your way through a maze.
- Find a path from one point in a graph (roadmap) to another point.
- Play a game in which there are moves to be made (checkers, chess).

In all of these cases, there are choices to be made among a number of options. We need some way to remember these decision points in case we want/need to come back and try the alternative
Consider the maze. At a point where a choice is made, we may discover that the choice leads to a dead-end. We want to retrace back to that decision point and then try the other (next) alternative.
Again, stacks can be used as part of the solution. Recursion is another, typically more favored, solution, which is actually implemented by a stack.

Memory Management
Any modern computer environment uses a stack as the primary memory management model for a running program. Whether it's native code (x86, Sun, VAX) or JVM, a stack is at the center of the run-time environment for Java, C++, Ada, FORTRAN, etc.
The discussion of JVM in the text is consistent with NT, Solaris, VMS, Unix runtime environments.
Each program that is running in a computer system has its own memory allocation containingthe typical layout as shown below.



Call and return process

When a method/function is called

1. An activation record is created; its size depends on the number and size of the localvariables and parameters.
2. The Base Pointer value is saved in the special location reserved for it
3. The Program Counter value is saved in the Return Address location
4. The Base Pointer is now reset to the new base (top of the call stack prior to the creationof the AR)
5. The Program Counter is set to the location of the first bytecode of the method beingcalled
6. Copies the calling parameters into the Parameter region
7. Initializes local variables in the local variable region

While the method executes, the local variables and parameters are simply found by adding aconstant associated with each variable/parameter to the Base Pointer.

When a method returns

1. Get the program counter from the activation record and replace what's in the PC
2. Get the base pointer value from the AR and replace what's in the BP
3. Pop the AR entirely from the stack.

<u>**Lecture Note-29**</u>

QUEUE

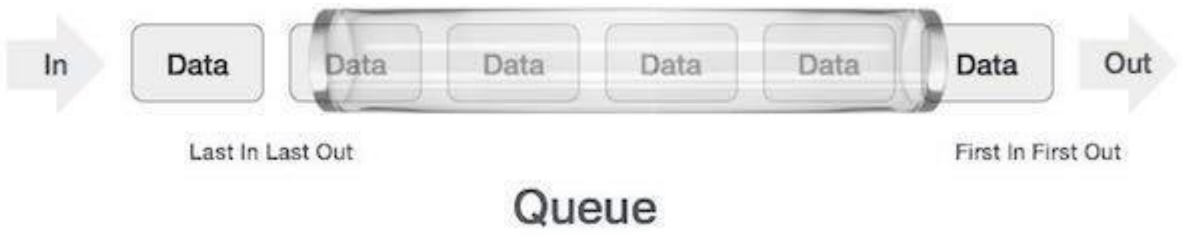
Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

LAST IN
LAST OUT
ONE WAY
FIRST IN
FIRST OUT

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus- stops.

Queue Representation

As we now understand that in queue, we access both ends for different reasons. The



following diagram given below tries to explain queue representation as data structure −
As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array. **Basic Operations**

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associatedwith queues −

- **enqueue()** − add (store) an item to the queue.
- **dequeue()** − remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. Theseare −

- **peek()** − Gets the element at the front of the queue without removing it.
- **isfull()** − Checks if the queue is full.
- **isempty()** − Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueing

(orstoring) data in the queue we take help of **rear** pointer.
Let's first learn about supportive functions of a queue –

peek()

This function helps to see the data at the **front** of the queue. The algorithm of peek() function isas follows –

Algorithm

```
begin procedure peek
   return queue[front]
```

Implementation of peek() function in C programming language –

Example

```
int peek() {

   return queue[front];
```

isfull()

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function –

Algorithm

```
begin procedure isfull



   if rear equals to MAXSIZE
      return true

   else

      return false
```

Implementation of isfull() function in C programming language –

Example

```
bool isfull() {

   if(rear == MAXSIZE - 1)
      return true;

   else
```

Example

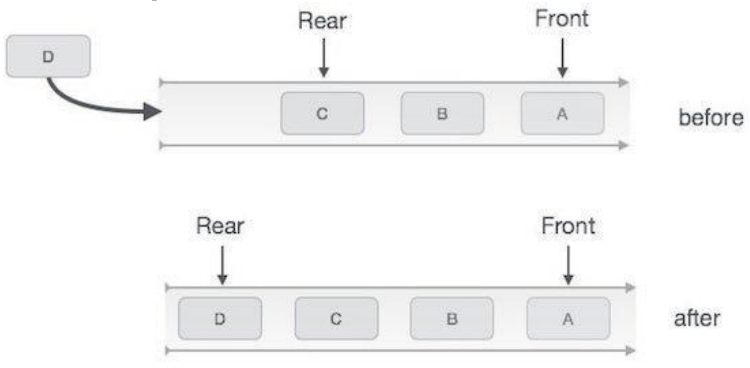
```
bool isempty() {

  if(front < 0 || front > rear)
    return true;

  else
```

Enqueue Operation

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparativelydifficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue −

- **Step 1** − Check if the queue is full.
- **Step 2** − If the queue is full, produce overflow error and exit.
- **Step 3** − If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** − Add data element to the queue location, where the rear is pointing.
- **Step 5** − return success.



Queue Enqueue

Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

Algorithm for enqueue operation

```
procedure enqueue(data)
```

```
endif


rear ← rear + 1
queue[rear] ← data
return true
```

Implementation of enqueue() in C programming language −

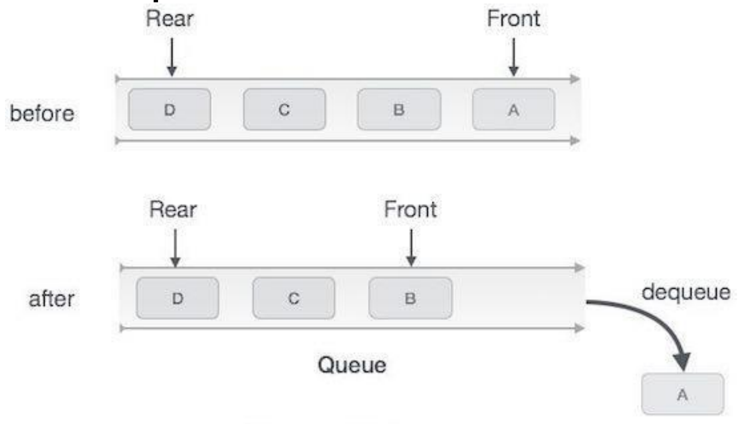
Example

```
int enqueue(int data)
  if(isfull())

    return 0;



  rear = rear + 1;
  queue[rear] = data;
```

Dequeue Operation

Accessing data from the queue is a process of two tasks − access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation −

- **Step 1** − Check if the queue is empty.
- **Step 2** − If the queue is empty, produce underflow error and exit.
- **Step 3** − If the queue is not empty, access the data where **front** is pointing.
- **Step 4** − Increment **front** pointer to point to the next available data element.
- **Step 5** − Return success.



Queue Dequeue

Algorithm for dequeue operation

```
procedure dequeue
```

```
 if queue is empty
    return underflow

 end if



 data = queue[front]
 front ← front + 1
```

Implementation of dequeue() in C programming language −

Example

```
int dequeue() {
   if(isempty())

      return 0;



   int data = queue[front];
   front = front + 1;
```

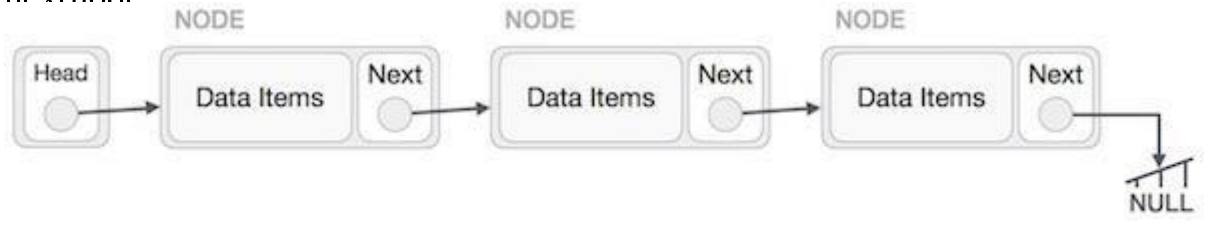
LINKED LIST

A linked list is a sequence of data structures, which are connected together via links. Linked List is a sequence of links which contains items. Each link contains a connectionto another link. Linked list is the second most-used data structure after array. Followingare the important terms to understand the concept of Linked List.

- **Link** − Each link of a linked list can store a data called an element.
- **Next** − Each link of a linked list contains a link to the next link called Next.
- **LinkedList** − A Linked List contains the connection link to the first link called First.

Linked List Representation

Linked list can be visualized as a chain of nodes, where every node points to the nextnode.



As per the above illustration, following are the important points to be considered.

- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.

Types of Linked List

Following are the various types of linked list.

- **Simple Linked List** − Item navigation is forward only.
- **Doubly Linked List** − Items can be navigated forward and backward.
- **Circular Linked List** − Last item contains link of the first element as next andthe first element has a link to the last element as previous.

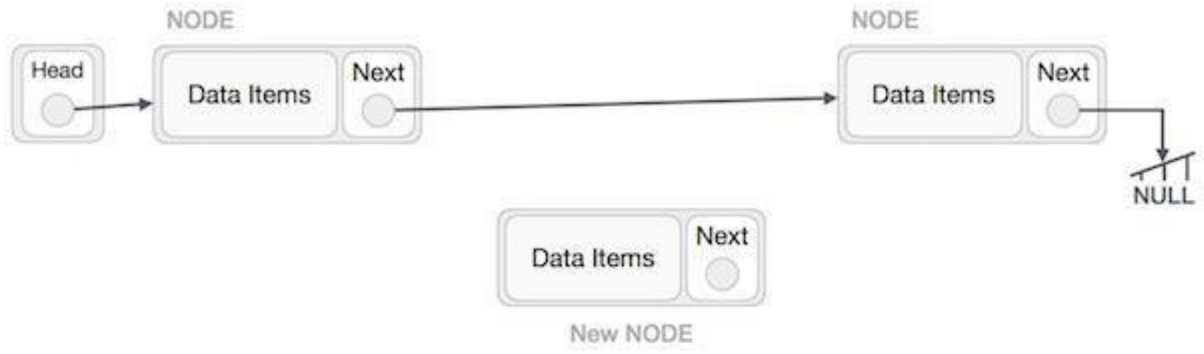
Basic Operations

Following are the basic operations supported by a list.

- **Insertion** − Adds an element at the beginning of the list.
- **Deletion** − Deletes an element at the beginning of the list.
- **Display** − Displays the complete list.
- **Search** − Searches an element using the given key.
- **Delete** − Deletes an element using the given key.

Insertion Operation

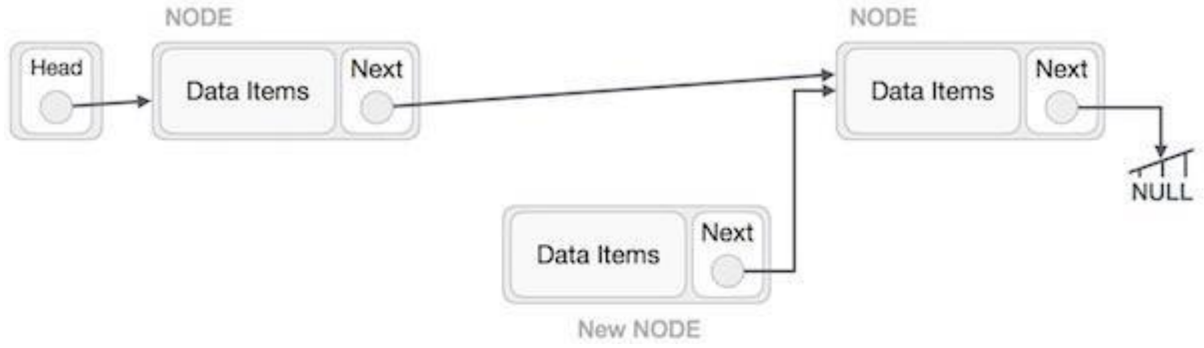
Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.



Imagine that we are inserting a node **B** (NewNode), between **A** (LeftNode)and **C** (RightNode). Then point B.next to C −

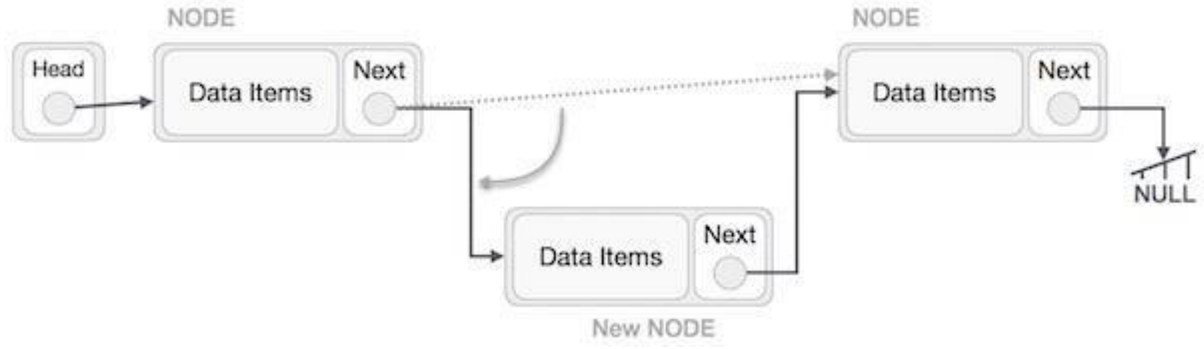
```
NewNode.next −> RightNode;
```

It should look like this −

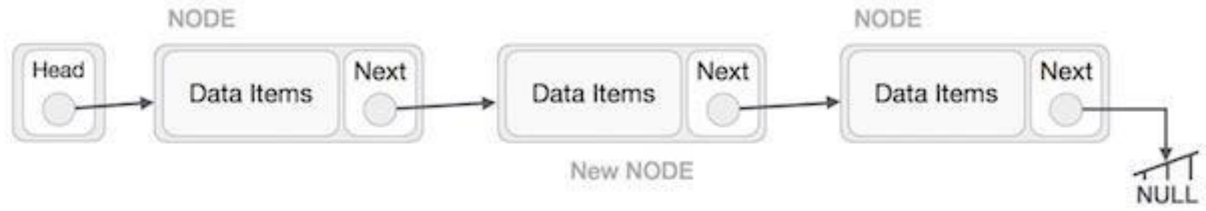


Now, the next node at the left should point to the new node.

```
LeftNode.next −> NewNode;
```



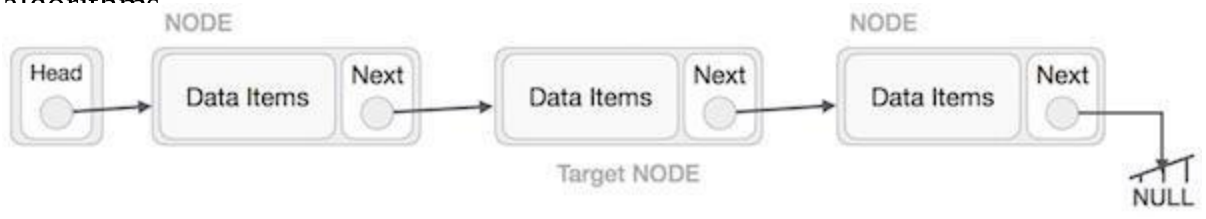
This will put the new node in the middle of the two. The new list should look like this −



Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the newnode and the new node will point to NULL.

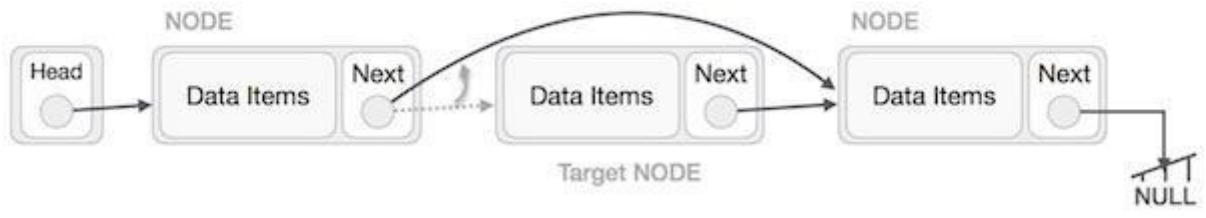
Deletion Operation

Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.



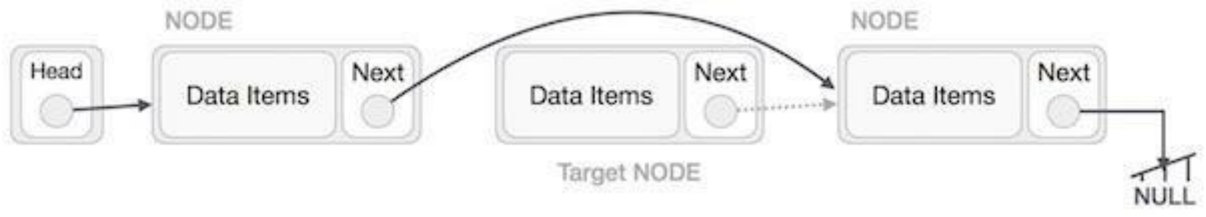
The left (previous) node of the target node now should point to the next node of the target node –

```
LeftNode.next –> TargetNode.next;
```

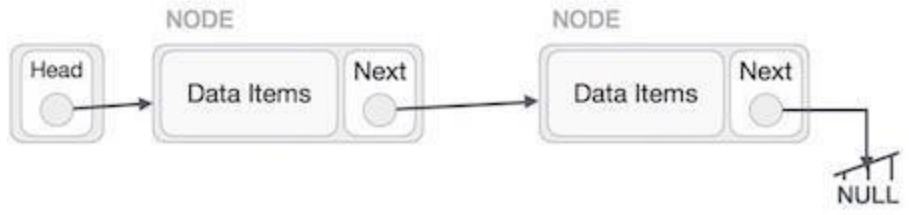


This will remove the link that was pointing to the target node. Now, using the

```
TargetNode.next –> NULL;
```

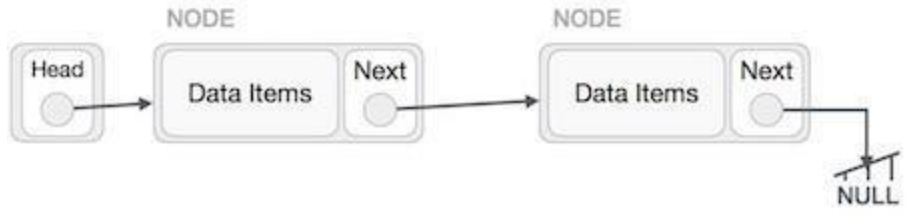


We need to use the deleted node. We can keep that in memory otherwise we cansimply deallocate memory and wipe off the target node completely.



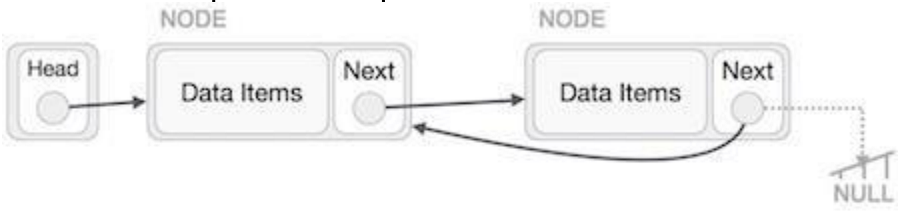
Reverse Operation

This operation is a thorough one. We need to make the last node to be pointed by thehead node and reverse the whole linked list.

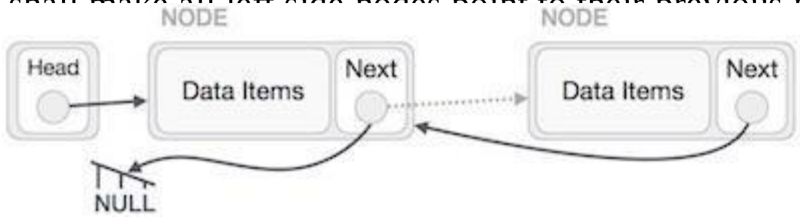


*Under revision

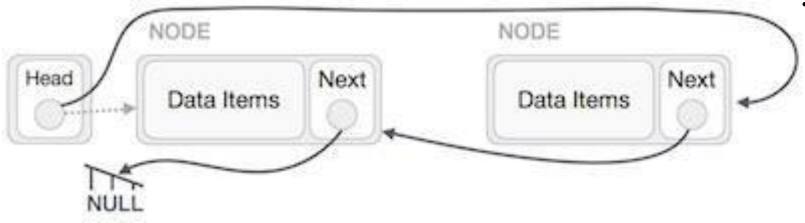
First, we traverse to the end of the list. It should be pointing to NULL. Now, we shall make it point to its previous node –



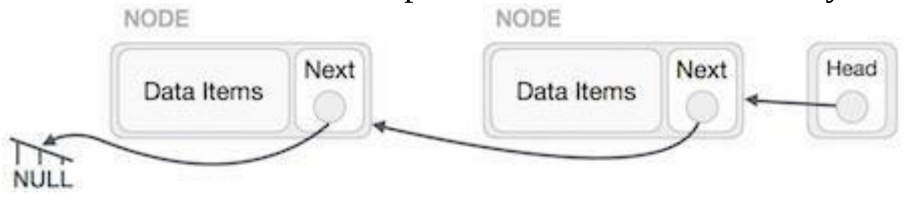
We have to make sure that the last node is not the lost node. So we'll have some temp node, which looks like the head node pointing to the last node. Now, we shall make all left side nodes point to their previous nodes one by one.



Except the node (first node) pointed by the head node, all nodes should point to their predecessor, making them their new successor. The first node will point to



We'll make the head node point to the new first node by using the temp node.



The linked list is now reversed.

Program:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include
<stdbool.h>

struct node
  {int  data;
  int key;
  struct node *next;
};
```

```
struct node *head =
NULL; struct node
*current = NULL;

//display the
listvoid
printList() {
  struct node *ptr =
  head;printf("\n[ ");

  //start from the
  beginningwhile(ptr !=
  NULL) {
    printf("(%d,%d) ",ptr->key,ptr-
    >data);ptr = ptr->next;
  }

  printf(" ]");
}

//insert link at the first location
void insertFirst(int key, int
data) {
  //create a link
  struct node *link = (struct node*) malloc(sizeof(struct node));

  link->key =
  key; link->data
  = data;

  //point it to old first
  nodelink->next =
  head;

  //point first to new first
  nodehead = link;
}
```

```
//delete first item
struct node* deleteFirst() {
```

```
//save reference to first link
struct node *tempLink =
head;

//mark next to first link as
firsthead = head->next;

//return the deleted link
```

```
   return tempLink;
}

//is list empty
bool isEmpty()
{
   return head == NULL;
}

int length() {
   int length = 0;
   struct node *current;

   for(current = head; current != NULL; current = current->next) {
      length++;
   }

   return length;
}

//find a link with given
keystruct node* find(int
key) {

   //start from the first link
   struct node* current =
   head;

   //if list is empty
   if(head ==
   NULL) {
      return NULL;
   }

   //navigate through list
   while(current->key !=
   key) {
```

```
   //if it is last node
   if(current->next ==
     NULL) {return NULL;
   } else {
     //go to next link
     current = current->next;
   }
}
```

```
  //if data found, return the current
  Linkreturn current;
}

//delete a link with given
keystruct node* delete(int
key) {

  //start from the first link
  struct node* current =
  head;
  struct node* previous = NULL;

  //if list is empty
  if(head ==
  NULL) {
    return NULL;
  }

  //navigate through list
  while(current->key !=
  key) {

    //if it is last node
    if(current->next ==
      NULL) {return NULL;
    } else {
      //store reference to current
      linkprevious = current;
      //move to next link
      current = current-
      >next;
    }
  }

  //found a match, update the
  linkif(current == head) {
    //change first to point to next
    linkhead = head->next;
  } else {
    //bypass the current link
    previous->next = current-
    >next;
  }

  return current;
}
```

```
void sort() {

  int i, j, k, tempKey,
  tempData;struct node
  *current;
  struct node *next;

  int size =
  length();k = size
  ;

  for ( i = 0 ; i < size - 1 ; i++, k-
    - ) {current = head;
    next = head->next;

    for ( j = 1 ; j < k ; j++ ) {

      if ( current->data > next->data
        ) {tempData = current-
        >data; current->data = next-
        >data; next->data =
        tempData;

        tempKey = current-
        >key; current->key =
        next->key;next->key =
        tempKey;
      }

      current = current-
      >next;next = next-
      >next;
    }
  }
}

void reverse(struct node**
  head_ref) {struct node* prev =
  NULL;
```

```
struct node* current =
*head_ref;struct node* next;

while (current !=
  NULL) {next =
  current->next;
  current->next = prev;
  prev = current;
  current = next;
}
```

```
   *head_ref = prev;
}

void main() {
   insertFirst(1,10
   );
   insertFirst(2,20
   );
   insertFirst(3,30
   );
   insertFirst(4,1);
   insertFirst(5,40
   );
   insertFirst(6,56
   );

   printf("Original List: ");

   //print
   list
   printList(
   );

   while(!isEmpty()) {
      struct node *temp =
      deleteFirst();printf("\nDeleted
      value:");
      printf("(%d,%d) ",temp->key,temp->data);
   }

   printf("\nList after deleting all
   items: ");printList();
   insertFirst(1,10
   );
   insertFirst(2,20
   );
   insertFirst(3,30
   );
```

```
insertFirst(4,1);
insertFirst(5,40
);
insertFirst(6,56
);

printf("\nRestored List:
");printList();
printf("\n");

struct node *foundLink = find(4);

if(foundLink != NULL)
  { printf("Element
  found: ");
  printf("(%d,%d) ",foundLink->key,foundLink-
  >data);printf("\n");
```

```
   } else {
     printf("Element not found.");
   }

  delete(4);
  printf("List after deleting an
  item: ");printList();
  printf("\n");
  foundLink =
  find(4);

  if(foundLink != NULL)
     { printf("Element
     found: ");
     printf("(%d,%d) ",foundLink->key,foundLink-
     >data);printf("\n");
   } else {
     printf("Element not found.");
   }

  printf("\n
  ");sort();

  printf("List after sorting the
  data: ");printList();

  reverse(&head);
  printf("\nList after reversing the
  data: ");printList();
}
```

If we compile and run the above program, it will produce the following result −Output

```
Original List:
[ (6,56) (5,40) (4,1) (3,30) (2,20) (1,10) ]
Deleted
value:(6,56)
Deleted
value:(5,40)
Deleted
value:(4,1)
Deleted
value:(3,30)
Deleted
```