MLR Institute ofTechnology
Laxma Reddy Avenue, Dundigal, Quthbullapur (M), Hyderabad – 500 043

Faculty Name	:	N.SANDHYA RANI
Subject	:	PPS
Topic	:	**Problem Solving and algorithmic thinking & introduction to c language**

UNIT –I CLASSNOTES

What is algorithm?

An algorithm is a procedure or step-by-step instruction for solving a problem. They form the foundation of writing a program.

For writing any programs, the following has to be known:

- Input
- Tasks to be preformed
- Output expected

Write an algorithm to add two numbers and display the result.

Input –Take two numbers

Processing – adding of two numbers

Output – display result

Algorithm

Step-1 Start

Step-2 Input first numbers say A

Step-3 Input second number say B

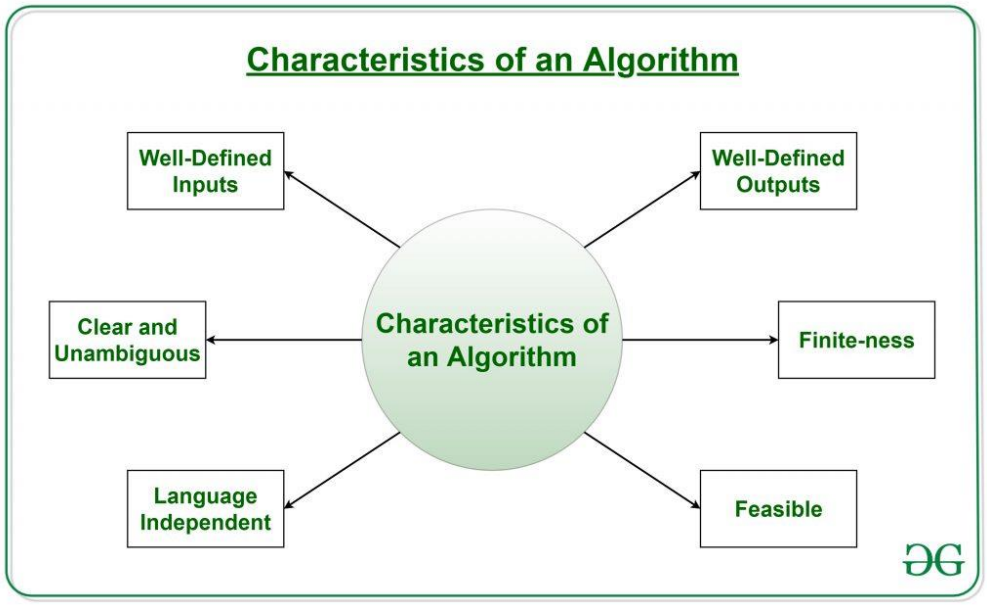
Step-4 SUM = A + B

Step-5 Display SUM

Step-6 Stop

Algorithm

Properties of an Algorithm



Not all procedures can be called an algorithm. An algorithm should have the below mentioned characteristics −

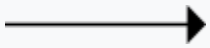
- **Unambiguous** − Algorithm should be clear and unambiguous. Each of its steps (or phases), and their input/outputs should be clear and must lead to only one meaning.
- **Input** − An algorithm should have 0 or more well defined inputs.
- **Output** − An algorithm should have 1 or more well defined outputs, and should match the desired output.
- **Finiteness** − Algorithms must terminate after a finite number of steps.
- **Feasibility** − Should be feasible with the available resources.
- **Independent** − An algorithm should have step-by-step directions which should be independent of any programming code.

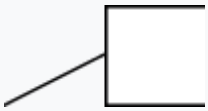
Flowchart

Definition:

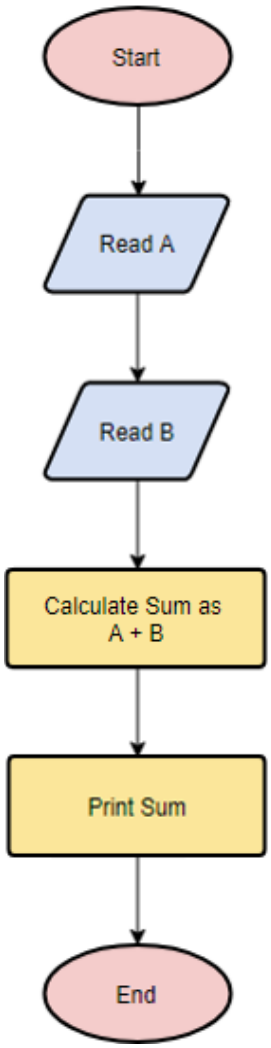
A **flowchart** is a type of diagram that represents a workflow or process. A flowchart can also be defined as a diagrammatic representation of an algorithm, a step-by-step approach to solving a task.

Common Symbols used in flowchart

ANSI/ISO Shape	Name	Description
	Flowline (Arrowhead)	Shows the process's order of operation. A line coming from one symbol and pointing at another. Arrowheads are added if the flow is not the standard top-to-bottom, left-to right
	Terminal	Indicates the beginning and ending of a program or sub-process. Represented as a stadium, oval or rounded (fillet) rectangle. They usually contain the word "Start" or "End", or another phrase signaling the start or end of a process, such as "submit inquiry" or "receive product".
	Process	Represents a set of operations that changes value, form, or location of data. Represented as a rectangle.
	Decision	Shows a conditional operation that determines which one of the two paths the program will take. The operation is commonly a yes/no question or true/false test. Represented as a diamond (rhombus).
	Input/Output	Indicates the process of inputting and outputting data as in entering data or displaying results. Represented as a rhomboid.

	Annotation (Comment)	Indicating additional information about a step in the program. Represented as an open rectangle with a dashed or solid line connecting it to the corresponding symbol in the flowchart.
	Predefined Process	Shows named process which is defined elsewhere. Represented as a rectangle with double-struck vertical edges.
	On-page Connector	Pairs of labeled connectors replace long or confusing lines on a flowchart page. Represented by a small circle with a letter inside.
	Off-page Connector	A labeled connector for use when the target is on another page. Represented as a home plate-shaped pentagon.

Example 1: Calculate sum of two number.



Flowgorithm

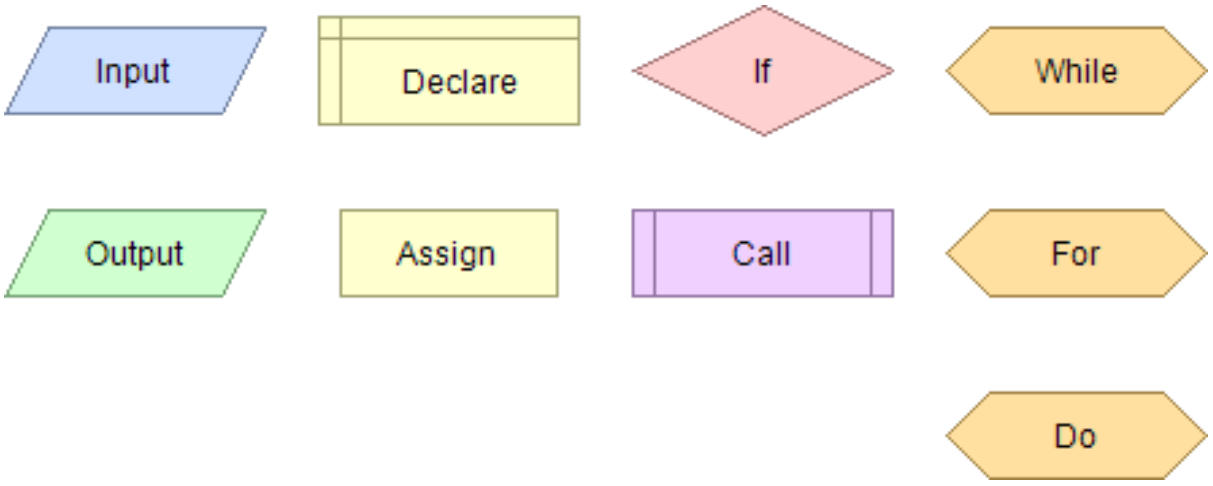
Introduction

Flowgorithm is a graphical authoring tool which allows users to write and execute programs using flow charts.

Flowgorithm is a beginner-friendly programming language in which a user would have to create a flowchart for carrying forward a task, which means in a programming language you would have to code it thoroughly to create and start a task.

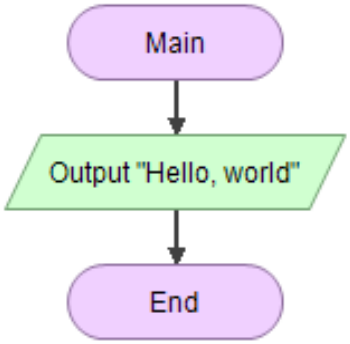
Graphical Shapes of Flowgorithm

- **Flowgorithm combines the classic flowchart symbols**
- **Flowgorithm supports a wide-variable of color schemes**
- **The application comes with a selection of built-in schemes.**

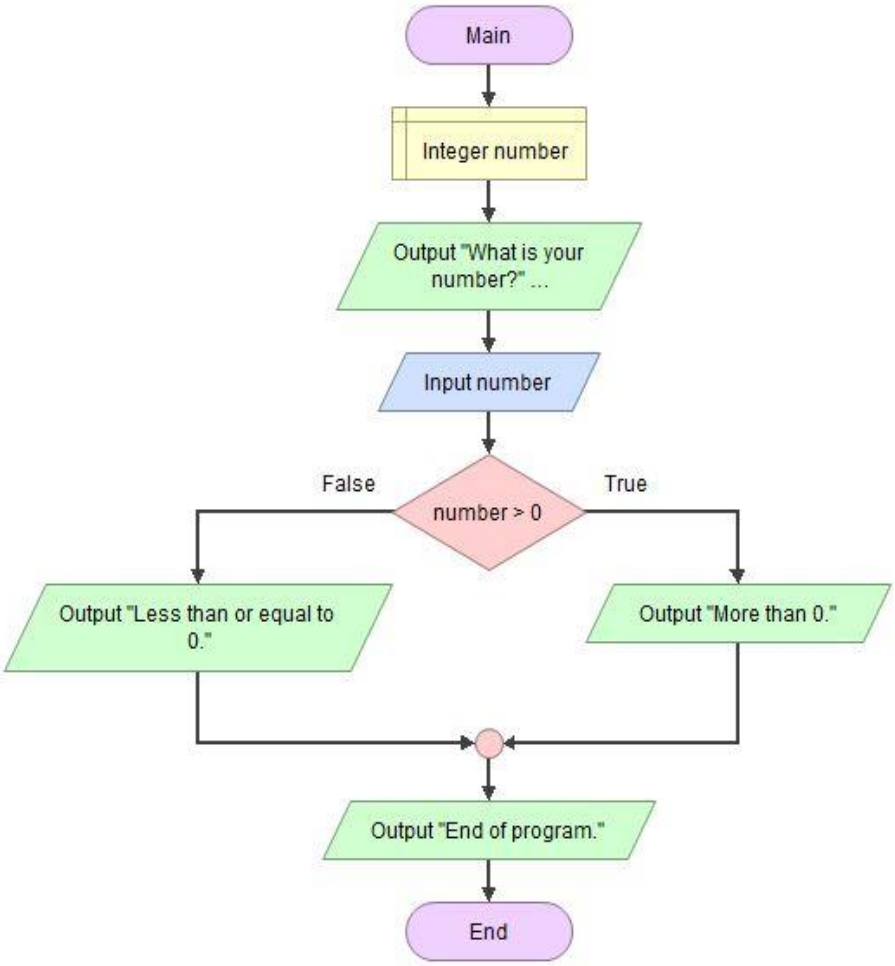


Examples of Flowgorithm

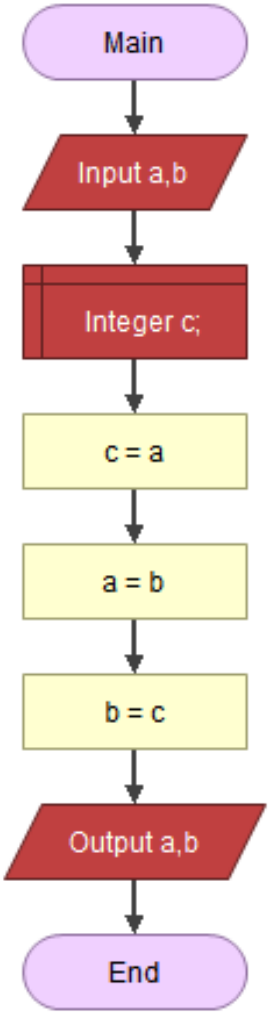
Example 1: Simple example



Example 2: Flowgorithm of a number which is greater or less than zero



Example 3: Flowagorithm of swapping of two numbers



Constituents of algorithms – (Sequence, Selection and Repetition)

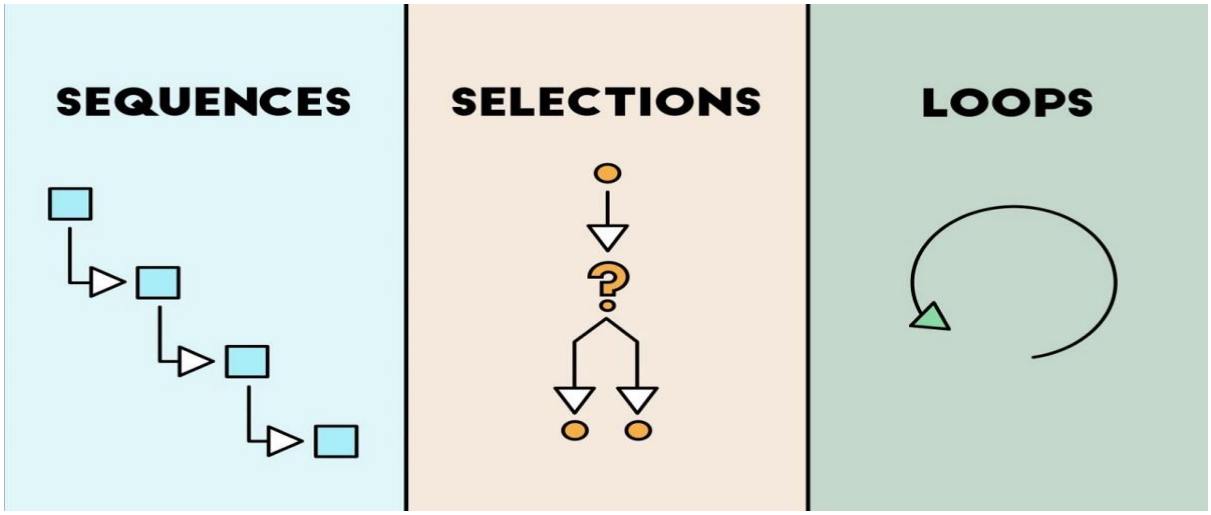
- The algorithm and flowchart, classification to the three types of control structures.
  They are:

  1. Sequence

  2. Branching (Selection)

  3. Loop (Repetition /iteration)

- These are three basic building blocks (constructs) to use when designing algorithms.
- These three control structures are sufficient for all purposes



SEQUENCE:-

- The sequence is exemplified by sequence of statements place one after the other – the one above or before another gets executed first. In flowcharts, sequence of statements is usually contained in the rectangular process box.

EX: FIND THE AREA OF CIRCLE

Problem 1: Find the area of a Circle of radius r.

Inputs to the algorithm: Radius r of the Circle.

Task :calculate radius of circle

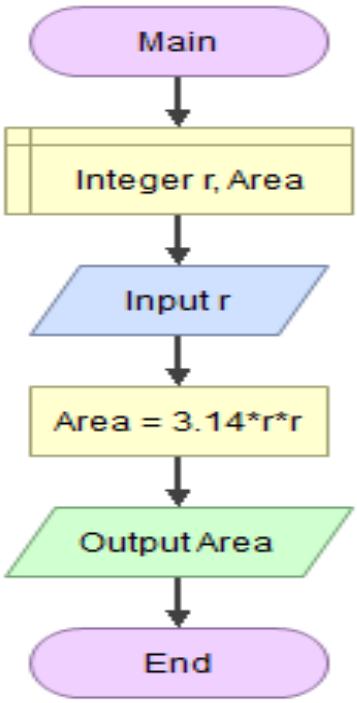
Expected output: Area of the Circle

Algorithm:

- Step1:Start
- Step 2: Read\input the Radius r of the Circle
- Step 3: Area PI*r*r

// calculation of area

- Step 4: Print Area
- Step 5:Stop



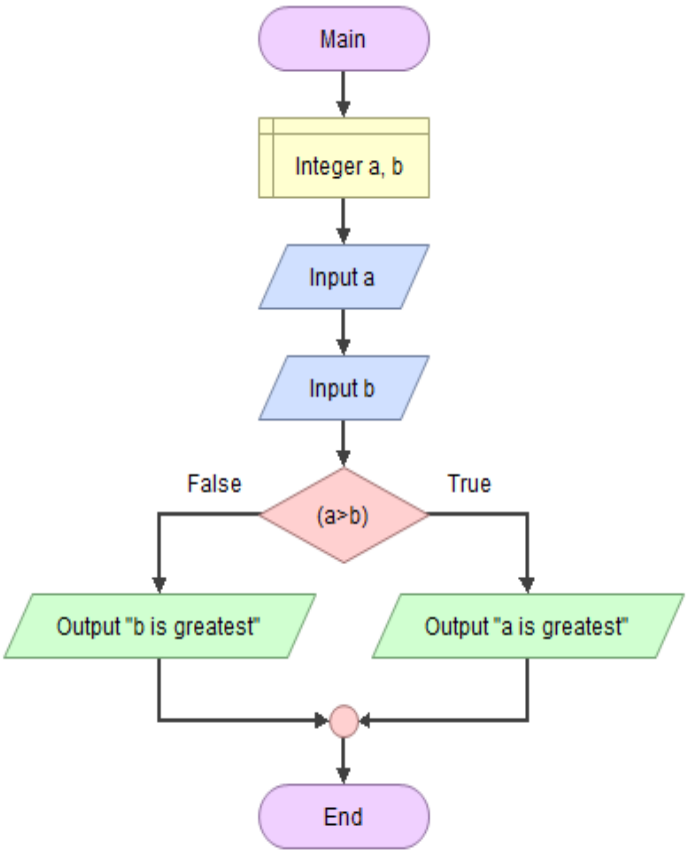
Selection:-

- When designing algorithms, there are many steps where decisions must be made.
- Selection is a decision or question.
- At some point in an algorithm there may need to be a question because the algorithm has reached a step where one or more options are available.
- Depending on the answer given, the algorithm will follow certain steps and ignore others.
- The branch refers to a binary decision based on some condition.
- If the condition is true, one of the two branches is explored;
- if the condition is false, the other alternative is taken.
- This is usually represented by the 'if-then' construct in pseudo-codes and programs.
- In flowcharts, this is represented by the diamond-shaped decision box.
- This structure is also known as the Selection/Branching structure.

- **Problem1: write algorithm to find the greater number between two numbers**

  - Inputs to the algorithm: First num1. Second num2.
  - Task: find greatest of two numbers
  - Expected output: print the greatest number.

  Algorithm:
  Step1: Start
  Step2: Read/input A and B
  Step3: if A greater than B then
  display "A is greatest"
  Step4: otherwise
  display "B is greatest"
  Step6: End



REPETITION:-

When designing algorithms, there may be some steps that need repeating. This is known as iteration and can be displayed with algorithms and flowcharts.

Iteration in programming means repeating steps, or instructions, over and over again. This is often called a ‘loop’.

Algorithms consist of instructions that are carried out (performed) one after another. Sometimes an algorithm needs to repeat certain steps until told to stop or until a particular condition has been met.

Iteration is the process of repeating steps.

The loop allows a statement or a sequence of statements to be repeatedly executed based on some loop condition.

It is represented by the ‘while’ and ‘for’ constructs in most programming languages.

In the flowcharts, a back arrow hints the presence of a loop.

A trip around the loop is known as iteration.

You must ensure that the condition for the termination of the looping must be satisfied after some finite number of iterations, otherwise it ends up as an infinite loop, a common mistake made by inexperienced programmers.

The loop is also known as the repetition structure.

Problem 1: A algorithm to find sum of individual digits of a given number

- Inputs to the algorithm: number
- Task: calculate sum of individual digits
- Expected output: print result

Algorithm:

Step 1:Start

Step 2: input num

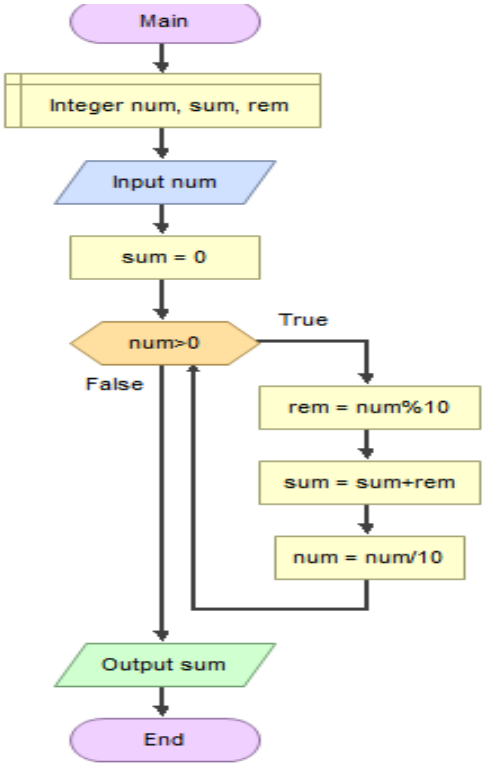
Step 3: assign sum=0

Step 4:Repeat the steps a to c until the condition n>0 is false

- step a:rem=n%10
- step b:sum=sum + rem
- step c:n=n/10

Step 5:print sum

Step 6:stop


Main
Integer num, sum, rem
Input num
sum = 0
num>0
True
False
rem = num%10
sum = sum+rem
num = num/10
Output sum
End

Faculty Name: N.SANDHYA RANI **Subject** :PPS
Topic: Introduction to C language: Structure of C programs

Unit No :2
Lecture No :L17
Book Reference:T1

Basic structure of C programs:

C is a structured programming language. Every c program and its statements must be in a particular structure. Every c program has the following general structure...

```
/* comments */          // It is opional. Generally used to provice description about the program
preprocessing commands   // It is opional. Generally used to include header files, define constants and enum
gloabal declarations;    // It is opional. Used to declare the variables that ae common for multiple functions
int main()               // main is a user defined function and it is compulsory statement.
{                        // It indicates the starting point of program execution.
   local declarations;   // Without main compiler does not understand from which statement execusion starts
   executable statements;
   .                     // Local declaration and executable statements are written according to our requirment
   .
   return 0;
}
userdefined function()   // It is opional. Used to provide implementation for user defined functions that already
{                        // declared either at global or local declaration part.
   function definition;
}
.
```

Line 1: Comments - They are ignored by the compiler

This section is used to provide small description of the program. The comment lines are simply ignored by the compiler, that means they are not executed. In C, there are two types of comments.

1. **Single Line Comments:** Single line comment begins with // symbol. We can write any number of single line comments.
2. **Multiple Lines Comments:** Multiple lines comment begins with /* symbol and ends with */. We can write any number of multiple lines comments in a program.

In a C program, the comment lines are optional. Based on the requirement, we write the comments. All the comment lines in a C program just provide the guidelines to understand the program and its code.

Line 2: Preprocessing Commands

Pre-processing commands are used to include header files and to define constants. We use **#include** statement to include header file into our program. We use **#define** statement to define a constant. The pre-processing statements are used according to the requirment. If we don't need any header file, then no need to write #include statement. If we don't need any constant, then no need to write #define statement.

Line 3: Global Declaration

Global declaration is used to define the global variables, which are common for all the functions after its declaration. We also use the global declaration to declare functions. This global declaration is used based on the requirement.

Line 4: int main()

Every C program must write this statement. This statement (main) specifies the starting point of the C program execution. Here, main is a user defined method which tells the compiler that this is the starting point of the program execution. Here, **int** is a datatype of a value that is going to return to the Operating System after completing the main method execution. If we don't want to return any value, we can use it as **void**.

Line 5: Open Brase ({)

The open brase indicates the begining of the block which belongs to the main method. In C program, every block begins with '{' symbol.

Line 6: Local Declaration

In this section, we declare the variables and functions that are local to the function or block in which they are declared. The variables which are declared in this section are valid only within the function or block in which they are declared.

Line 7: Executable statements

In this section, we write the statements which perform tasks like reading data, displaying result, calculations etc., All the statements in this section are written according to the requirment.

Line 9: Closing Brase (})

The close brase indicates the end of the block which belongs to the main method. In C program every block ends with '}' symbol.

Line 10, 11, 12, ...: Userdefined function()

This is the place where we implement the userdefined functions. The userdefined function implementation can also be performed before the main method. In this case, the user defined function need not to be declared. Directly it can be implemented, but it must be before the main method. In a program, we can define as many userdefined functions as we want. Every user defined function needs a function call to execute its statements.

1.Every executable statement must end with semicolon symbol (;).

2.Every C program must contain exactly one main method (Starting point of the program execution).

3.All the system defined words (keywords) must be used in lowercase letters.

4.Keywords can not be used as user defined names(identifiers).

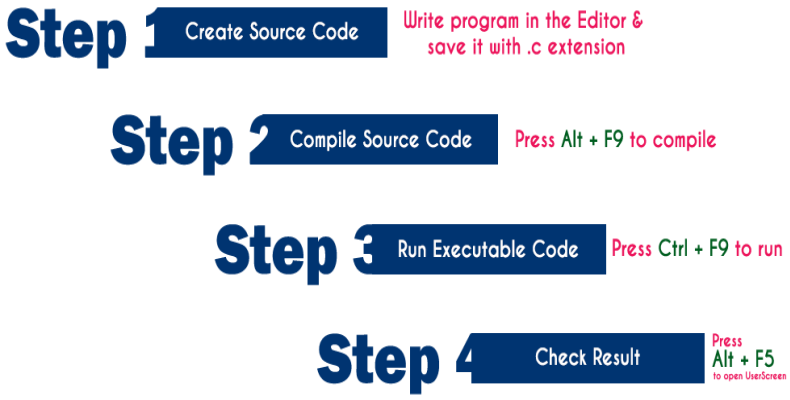
5.For every open brase ({), there must be respective closing brase (}).
6.Every variable must be declared before it is used.

Process of compiling and running a C program:

Generally, the programs created using programming languages like C, C++, Java etc., are written using high level language like English. But, computer cannot understand the high level language. It can understand only low level language. So, the program written in high level language needs to be converted into low level language to make it understandable for the computer. This conversion is performed using either Interpreter or Compiler.

Popular programming languages like C, C++, Java etc., use compiler to convert high level language instructions into low level language instructions. Compiler is a program that converts high level language instructions into low level language instructions. Generally, compiler performs two things, first it verifies the program errors, if errors are found, it returns list of errors otherwise it converts the complete code into low level language.

To create and execute C programs in Windows Operating System, we need to install Turbo C software. We use the following steps to create and execute C programs in Windows OS…



Step 1: Creating Source Code

Source code is a file with C programming instructions in high level language. To create source code, we use any text editor to write the program instructions. The instructions written in the source code must follow the C programming language rules. The following steps are used to create source code file in Windows OS…

- Click on Start button
- Select Run
- Type cmd and press Enter
- Type cd c:\TC\bin in the command prompt and press Enter
- Type TC press Enter
- Click on File -> New in C Editor window
- Type the program
- Save it as FileName.c (Use shortcut key F2 to save)

Step 2: Compile Source Code (Alt + F9)
Compilation is the process of converting high level language instructions into low level language instructions. We use the shortcut key Alt + F9 to compile a C program in Turbo C.
Compilation is the process of converting high level language instructions into low level language instructions.
Whenever we press Alt + F9, the source file is going to be submitted to the Compiler. On receiving a source file, the compiler first checks for the Errors. If there are any Errors then compiler returns List of Errors, if there are no errors then the source code is converted into object code and stores it as file with .obj extension. Then the object code is given to the Linker. The Linker combines both the object code and specified header file code and generates an Executable file with .exe extension.
Step 3: Executing / Running Executable File (Ctrl + F9)
After completing compilation successfully, an executable file is created with .exe extension. The processor can understand this .exe file content so that it can perform the task specified in the source file.

We use a shortcut key Ctrl + F9 to run a C program. Whenever we press Ctrl + F9, the .exe file is submitted to the CPU. On receiving .exefile, CPU performs the task according to the instruction written in the file. The result generated from the execution is placed in a window called User Screen.
Step 4: Check Result (Alt + F5)
After running the program, the result is placed into User Screen. Just we need to open the User Screen to check the result of the program execution. We use the shortcut key Alt + F5 to open the User Screen and check the result.

Faculty Name: N.SANDHYA RANI **Subject** :PPS
Topic: Data types

Unit No :2
Lecture No :L18
Book Reference:T1

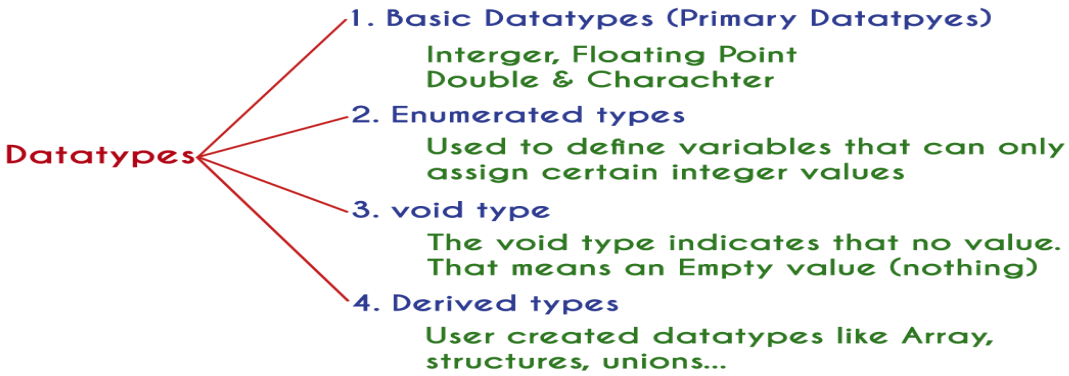
DataTypes:
Data used in c program is classified into different types based on its properties. In c programming langauge, datatype can be defined as a set of values with similar characteristics. All the values in a datatype have the same properties.

Datatypes in c programming language are used to specify what kind of value can be stored in a variable. The memory size and type of value of a variable are determined by varible datatype. In a c program, each variable or constant or array must have a datatype and this datatype specifies how much memory is to be allocated and what type of values are to be stored in that variable or constant or array. The formal definition of datatype is as follows...

Datatype is a set of value with predefined characteristics. Datatypes are used to declare variable, constants, arrays, pointers and functions.

In c programming language, datatypes are classified as follows...

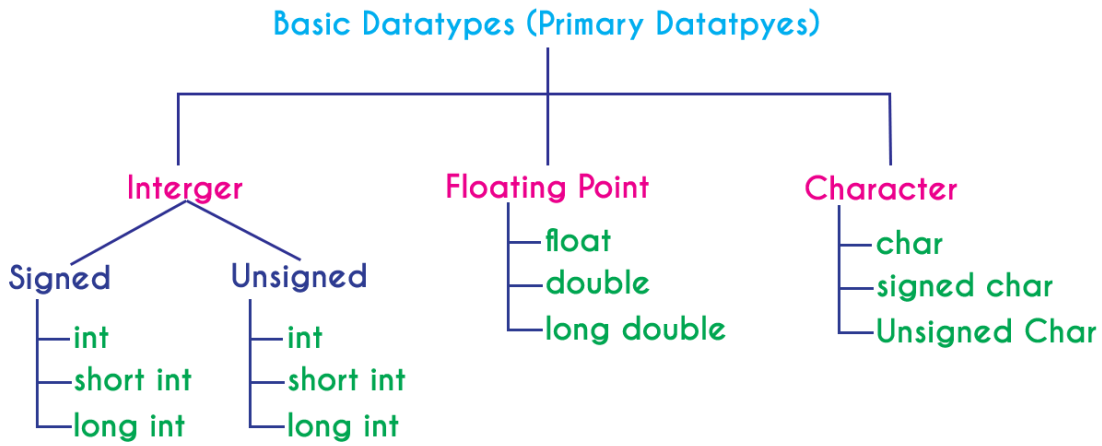
1.Primary Datatypes (Basic Datatypes OR Predefined Datatypes)
2.Derived Datatypes (Secondary Datatypes OR Userdefined Datatypes)
3.Enumeration Datatypes
4.Void Datatype



Primary Datatypes

The primary datatypes in C programming language are the basic datatypes. All the primary datatypes are already defined in the system. Primary datatypes are also called as Built-In datatypes. The following are the primary datatypes in c programming lanuage...

1.Integer Datatype
2.Floating Point Datatype
3.Double Datatype
4.Character Datatype



Integer Datatype

Integer datatype is a set of whole numbers. Every integer value does not have the decimal value. We use the keyword "**int**" to represent integer datatype in c. We use the keyword int to declare the variables and to specify return type of a function. The integer datatype is used with different type modifiers like short, long, signed and unsigned. The following table provides complete details about integer datatype.

Type	Size (Bytes)	Range	Specifier
int (signed short int)	2	-32768 to +32767	%d
short int (signed short int)	2	-32768 to +32767	%d
long int (signed long int)	4	-2,147,483,648 to +2,147,483,647	%d
unsigned int (unsigned short int)	2	0 to 65535	%u
unsigned long int	4	0 to 4,294,967,295	%u

Floating Point Datatypes

Floating point datatypes are set of numbers with decimal value. Every floating point value must contain the decimal value. The floating point datatype has two variants...

- float
- double

We use the keyword "**float**" to represent floating point datatype and "**double**" to represent double datatype in c. Both float and double are similar but they differ in number of decimal places. The float value contains 6 decimal places whereas double value contains 15 or 19 decimal places. The following table provides complete details about floating point datatypes.

Type	Size (Bytes)	Range	Specifier
float	4	1.2E - 38 to 3.4E + 38	%f
double	8	2.3E-308 to 1.7E+308	%lf
long double	10	3.4E-4932 to 1.1E+4932	%lf

Character Datatype

Character datatype is a set of characters enclosed in single quotations. The following table provides complete details about character datatype.

Type	Size (Bytes)	Range	Specifier
char (signed char)	1	-128 to +127	%c
unsigned char	1	0 to 255	%c

The following table provides complete information about all the datatypes in c programming language...

	Integer	Floating Point	Double	Character
What is it?	Numbers without decimal value	Numbers with decimal value	Numbers with decimal value	Any symbol enclosed in single quotation
Keyword	int	float	double	char
Memory Size	2 or 4 Bytes	4 Bytes	8 or 10 Bytes	1 Byte
Range	-32768 to +32767 (or) 0 to 65535 (Incase of 2 bytes only)	1.2E - 38 to 3.4E + 38	2.3E-308 to 1.7E+308	-128 to + 127 (or) 0 to 255
Type Specifier	%d or %i or %u	%f	%ld	%c or %s
Type Modifier	short, long signed, unsigned	No modifiers	long	signed, unsigned
Type Qualifier	const, volatile	const, volatile	const, volatil	const, volatile

void Datatype

The void datatype means nothing or no value. Generally, void is used to specify a function which does not return any value. We also use the void datatype to specify empty parameters of a function.

Enumerated Datatype

An enumerated datatype is a user-defined data type that consists of integer constants and each integer constant is given a name. The keyword "**enum**" is used to define enumerated datatype.

Derived Datatypes

Derived datatypes are user-defined data types. The derived datatypes are also called as user defined datatypes or secondary datatypes. In c programming language, the derived datatypes are created using the following concepts...

- Arrays
- Structures
- Unions
- Enumeration

Faculty Name: N.SANDHYA RANI **Subject** :PPS
Topic: data inputs, output statements

Unit No :2
Lecture No :L19
Book Reference:T1

Input Functions in C:

C programming language provides built-in functions to perform input operations. The input opearations are used to read user values (input) from keyboard. C programming language provides the following built-in input functions...

1. **scanf()**
2. **getchar()**
3. **getch()**

4. **gets()**
5. **fscanf()**

scanf() function:

The scanf() function is used to read multiple data values of different data types from the keyboard. The scanf() function is built-in function defined in a header file called "**stdio.h**". When we want to use scanf() function in our program, we need to include the respective header file (stdio.h) using **#include** statement. The scanf() function has the following syntax...

Syntax:

```
scanf("format strings",&variableNames);
```

Example Program

```
#include <stdio.h>

void main(){

 int i;

  printf("\nEnter any integer value: ");

scanf("%d",&i);

 printf("\nYou have entered %d number",i);

}
```

Output:

```
Enter any integer value: 55
You have entered 55 number
```

In the above example program, we used the scanf() function to read an integer value from the keyboard and store it into variable 'i'.

The scanf function also used to read multiple data values of different or same data types. Consider the following example program...

```
#include <stdio.h>

void main(){

int i;

float x;
```

```
printf("\nEnter one integer followed by one float value : ");

scanf("%d%f",&i, &x);

 printf("\ninteger = %d, float = %f",i, x);   }
```

Output:

```
Enter one integer followed by one float value : 20 30.5
integer = 20, float = 30.5
```

In the above example program, we used the scanf() function to read one integer value and one float value from the keyboard. Here **'i'** is an integer variable so we have used format string %d, and **'x'** is a float variable so we have used format string %f.

The scanf() function returns an integer value equal to the total number of input values read using scanf function.

Example Program

```
#include <stdio.h>

void main(){

  int i,a,b;

  float x;

  printf("\nEnter two integers and one float : ");

  i = scanf("%d%d%f",&a, &b, &x);

  printf("\nTotal inputs read : %d",i);

}
```

Output:

```
Enter two integers and one float : 10 20 55.5
Total inputs read : 3
```

getchar() function

The getchar() function is used to read a character from the keyboard and return it to the program. This function is used to read only single character. To read multiple characters we need to write multiple times or use a looping statement. Consider the following example program...

```
#include <stdio.h>

void main(){

  char ch;

  printf("\nEnter any character : ");

  ch = getchar();

  printf("\nYou have entered : %c",ch);

}
```

Output:

```
Enter any character : A
You have entered : A
```

getch() function

The getch() function is similar to getchar function. The getch() function is used to read a character from the keyboard and return it to the program. This function is used to read only single character. To read multiple characters we need to write multiple times or use a looping statement. Consider the following example program...

```
#include <stdio.h>

void main(){

  char ch;

  printf("\nEnter any character : ");

  ch = getch();

  printf("\nYou have entered : %c",ch);

}
```

Output:

```
Enter any character :
You have entered : A
```

gets() function

The gets() function is used to read a line of string and stores it into character array. The gets() function reads a line of string or sequence of characters till a newline symbol enters. Consider the following example program...

```
#include <stdio.h>

void main(){

  char name[30];

  printf("\nEnter your favourite website: ");

  gets(name);

  printf("%s",name);

}
```

Output:

Enter your favourite website: www.btechsmartclass.com

fscanf() function

The fscanf() function is used with the concept of files. The fscanf() function is used to read data values from a file. When you want to use fscanf() function the file must be opened in reading mode.

Output Functions in C:

C programming language provides built-in functions to perform output operation. The output opearations are used to display data on user screen (output screen) or printer or any file. C programming language provides the following built-in output functions...

printf()

putchar()

puts()

fprintf()

printf() function

The printf() function is used to print string or data values or combination of string and data values on the output screen (User screen). The printf() function is built-in function defined in a header file called "**stdio.h**". When we want to use printf() function in our program we need to include the respective header file (stdio.h) using **#include** statement. The printf() function has the following syntax...

Syntax:

```
printf("message to be display!!!");
```

Example Program

```
#include <stdio.h>

void main(){

   printf("Hello! Welcome to btechsmartclass!!!");

}
```

Output:

```
Hello! Welcome to btechsmartclass!!!
```

In the above example program, we used the printf() function to print a string on to the output screen.

The printf() function is also used to display data values. When we want to display data values we use **format string** of the data value to be display.

Syntax:

```
printf("format string",variableName);
```

Example Program

```
#include <stdio.h>

void main(){

   int i = 10;

   float x = 5.5;
```

```
  printf("%d %f",i, x);   }
```

Output:

10 5.5

In the above example program, we used the printf() function to print data values of variables i and x on to the output screen. Here i is a integer variable so we have used format string %d and x is a float variable so we have used format string %f.

The printf() function can also used to display string along with data values.

Syntax:

printf("String format string",variableName);

Example Program

```
#include <stdio.h>

void main(){

  int i = 10;

  float x = 5.5;

  printf("Integer value = %d, float value = %f",i, x);


}
```

Output:

Integer value = 10, float value = 5.5

In the above program we are displaying string along with data values.

Every function in C programming language must have a return value. The printf() function also have integer as return value. The printf() function returns an integer value equalent to the total number of characters it has printed.

Example Program

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

```
void main(){
  int i;
  i = printf("btechsmartclass");
  printf(" is %d number of characters.",i);   }
```

Output:

btechsmartclass is 15 number of characters.

In the above program, first printf() function printing "btechsmartclass" which is of 15 characters. So it returns integer value 15 to variable "i". The value of "i" is printed in the second printf() function.

Formatted printf() function

Generally, when we write multiple printf() statements the result is displayed in single line because the printf() function displays the output in a single line. Consider the following example program...

```
#include <stdio.h>
void main(){
  printf("Welcome to ");
  printf("btechsmartclass ");
  printf("the perfect website for learning");
}
```

Output:

Welcome to btechsmartclass the perfect website for learning

In the above program, there are 3 printf() statements written in different lines but the output is displayed in single line only.

To display the output in different lines or as we wish, we use some special characters called escape sequences. Escape sequences are special characters with special functionality used in printf() function to format the output according to the user requirement. In C programming language, we have the following escape sequences...

Escape Sequence	Meaning
\n	New line
\t	Horizontal Tab
\v	Vertical Tab
\a	Beep sound
\b	Backspace
\\	Backward slash
\?	Question mark
\'	Single quotation mark
\"	Double quotation mark

Consider the following example program...

```
#include <stdio.h>
void main(){
  printf("Welcome to\n");
  printf("btechsmartclass\n");
  printf("the perfect website for learning");
}
```

Output:

```
Welcome to
btechsmartclass
the perfect website for learning
```

putchar() function

The putchar() function is used to display single character on the output screen. The putchar() functions prints the character which is passed as parameter to it and returns the same character as return value. This function is used to print only single charater. To print multiple characters we need to write multiple times or use a looping statement. Consiider the following example program...

```
#include <stdio.h>

void main(){

  char ch = 'A';

  putchar(ch);
```

```
}
```

Output:

A

puts() function

The puts() function is used to display string on the output screen. The puts() functions prints a string or sequence of characters till the newline. Consiider the following example program...

```
#include <stdio.h>
void main(){
  char name[30];
  printf("\nEnter your favourite website: ");
  gets(name);
  puts(name);
}
```

Output:

```
Enter your favourite website: www.btechsmartclass.com
www.btechsmartclass.com
```

fprintf() function

The fprintf() function is used with the concept of files. The fprintf() function is used to print a line into the file. When you want to use fprintf() function the file must be opened in writting mode.

Faculty Name: N.SANDHYA RANI **Subject** :PPS
Topic: Operators

Unit No :2
Lecture No :L20
Book Reference:T1

Operators:

An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in programs to manipulate data and variables.

C operators can be classified into a number of categories, they are
1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Assignment Operators
5. Increment and decrement operators
6. Conditional operators
7. Bitwise Operators
8. Special operators

1. Arithmetic Operators:

The arithmetic operators are the symbols that are used to perform basic mathematical operations like addition, subtraction, multiplication, division and percentage modulo. The following table provides information about arithmetic operators.

Operator	Meaning	Example
+	Addition	10 + 5 = 15
-	Subtraction	10 - 5 = 5
*	Multiplication	5 * 10 = 50
/	Division	10 / 5 = 2
%	Remainder of Division	5 % 2 = 1

The addition operator can be used with numerical data types and character data type. When it is used with numerical values, it performs mathematical addition and when it is used with character data type values, it performs concatination (appending).

The remainder of division operator is used with integer data type only.

2.Relational Operators :

The relational operators are the symbols that are used to compare two values. That means, the relational operators are used to check the relationship between two values. Every relational operator has two results TRUE or FALSE. In simple words, the relational operators are used to define conditions in a program. The following table provides information about relational operators.

Operator	Meaning	Example
<	Returns TRUE if first value is smaller than second value otherwise returns FALSE	10 < 5 is FALSE
>	Returns TRUE if first value is larger than second value otherwise returns FALSE	10 > 5 is TRUE
<=	Returns TRUE if first value is smaller than or equal to second value otherwise returns FALSE	10 <= 5 is FALSE
>=	Returns TRUE if first value is larger than or equal to second value otherwise returns FALSE	10 >= 5 is TRUE
==	Returns TRUE if both values are equal otherwise returns FALSE	10 == 5 is FALSE
!=	Returns TRUE if both values are not equal otherwise returns FALSE	10 != 5 is TRUE

3. Logical Operators:

The logical operators are the symbols that are used to combine multiple conditions into one condition. The following table provides information about logical operators.

Operator	Meaning	Example
&&	**Logical AND** - Returns TRUE if all conditions are TRUE otherwise returns FALSE	10 < 5 && 12 > 10 is FALSE
||	**Logical OR** - Returns FALSE if all conditions are FALSE otherwise returns TRUE	10 < 5 || 12 > 10 is TRUE
!	**Logical NOT** - Returns TRUE if condition is FLASE and returns FALSE if it is TRUE	!(10 < 5 && 12 > 10) is TRUE

Logical AND - Returns TRUE only if all conditions are TRUE, if any of the conditions is FALSE then complete condition becomes FALSE.
Logical OR - Returns FALSE only if all conditions are FALSE, if any of the conditions is TRUE then complete condition becomes TRUE.

4. Assignment Operators :

The assignment operators are used to assign right hand side value (Rvalue) to the left hand side variable (Lvalue). The assignment operator is used in different variants along with arithmetic operators. The following table describes all the assignment operators in C programming language.

Operator	Meaning	Example
=	Assign the right hand side value to left hand side variable	A = 15
+=	Add both left and right hand side values and store the result into left hand side variable	A += 10 ⇒ A=A+10
-=	Subtract right hand side value from left hand side variable value and store the result into left hand side variable	A -= B ⇒ A=A-B
*=	Multiply right hand side value with left hand side variable value and store the result into left hand side variable	A *= B ⇒ A=A*B
/=	Divide left hand side variable value with right hand side variable value and store the result into left hand side variable	A /= B ⇒ A=A/B
%=	Divide left hand side variable value with right hand side variable value and store the remainder into left hand side variable	A %= B ⇒ A=A%B

5.Increment & Decrement Operators:

The increment and decrement operators are called as unary operators because, both needs only one operand. The increment operators adds one to the existing value of the operand and the decrement operator subtracts one from the existing value of the operand. The following table provides information about increment and decrement operators.

Operator	Meaning	Example
++	**Increment** - Adds one to existing value	int a = 5; a++; ⇒ a = 6
--	**Decrement** - Subtracts one from existing value	int a = 5; a--; ⇒ a = 4

The increment and decrement operators are used infront of the operand (++a) or after the operand (a++). If it is used infront of the operand, we call it as pre-increment or pre-decrement and if it is used after the operand, we call it as post-increment or post-decrement.

Pre-Increment or Pre-Decrement

In case of pre-increment, the value of the variable is increased by one before the expression evaluation. In case of pre-decrement, the value of the variable is decreased by one before the expression evaluation. That means, when we use pre increment or pre decrement, first

the value of the variable is incremented or decremented by one, then modified value is used in the expression evaluation.

Example:

```
#include <stdio.h>
void main()
{
    int i = 5,j;
    j = ++i; // Pre-Increment
    printf("i = %d, j = %d",i,j);
}
```

Output:

```
i = 6, j = 6
```

Post-Increment or Post-Decrement

In case of post-increment, the value of the variable is increased by one after the expression evaluation. In case of post-decrement, the value of the variable is decreased by one after the expression evaluation. That means, when we use post-increment or post-decrement, first the expression is evaluated with existing value, then the value of the variable is incremented or decremented by one.

Example

```
#include <stdio.h>
void main()
{
   int i = 5,j;
    j = i++;  // Post-Increment
    printf("i = %d, j = %d",i,j);
}
```

Output:

```
i = 6, j = 5
```

***6.*Conditional Operator (?:)**

The conditional operator is also called as ternary operator because it requires three operands. This operator is used for decision making. In this operator, first we verify a condition, then we perform one operation out of the two operations based on the condition result. If the condition is TRUE the first option is performed, if the condition is FALSE the second option is performed. The conditional operator is used with the following syntax...

Condition ? TRUE Part : FALSE Part ;

Example:
A = (10<15) ? 100 : 200 ; ⇒ A value is 100

7.Bitwise Operators:

The bitwise operators are used to perform bit level operations in c programming language. When we use the bitwise operators, the operations are performed based on the binary values. The following table describes all the bitwise operators in C programming language.

Let us consider two variables A and B as A = 25 (11001) and B = 20 (10100)

Operator	Meaning	Example
&	the result of **Bitwise AND** is 1 if all the bits are 1 otherwise it is 0	A & B ⇒ 16 (10000)
|	the result of **Bitwise OR** is 0 if all the bits are 0 otherwise it is 1	A | B ⇒ 29 (11101)
^	the result of **Bitwise XOR** is 0 if all the bits are same otherwise it is 1	A ^ B ⇒ 13 (01101)
~	the result of **Bitwise once complement** is nagation of the bit (Flipping)	~A ⇒ 6 (00110)
<<	the **Bitwise left shift** operator shifts all the bits to the left by specified number of positions	A<<2 ⇒ 100 (1100100)
>>	the **Bitwise right shift** operator shifts all the bits to the right by specified number of positions	A>>2 ⇒ 6 (00110)

8.Special Operators (sizeof, pointer, comma, dot etc.)

The following are the special operators in c programming language.

sizeof operator

This operator is used to find the size of the memory (in bytes) allocated for a variable. This operator is used with the following syntax...

sizeof(variableName);

Example

sizeof(A); ⇒ result is 2 if A is an integer

Pointer operator (*)

This operator is used to define pointer variables in c programming language.

Comma operator (,)

This operator is used to separate variables while they are declaring, separate the expressions in function calls etc..

Dot operator (.)

This operator is used to access members of structure or union.

Faculty Name: N.SANDHYA RANI **Subject** :PPS
Topic: precedence and associativity

Unit No :2
Lecture No :L21
Book Reference:T1

Operator Precedence and Associativity

Explain Operator Precedence

Operator precedence is used to determine the order of operators evaluated in an expression. In c programming language every operator has precedence (priority). When there is more than one operator in an expression the operator with higher precedence is evaluated first and the operator with least precedence is evaluated last.

Discuss about Operator Associativity

Operator associativity is used to determine the order of operators with equal precedence evaluated in an expression. In c programming language, when an expression contains multiple operators with equal precedence, we use associativity to determine the order of evaluation of those operators.
In c programming language the operator precedence and associativity is as shown in the following table...

Precedence	Operator	Operator Meaning	Associativity
1	() [] -> .	function call array reference structure member access structure member access	Left to Right
2	! ~ + - ++ -- & * sizeof (type)	negation 1's complement Unary plus Unary minus increment operator decrement operator address of operator pointer returns size of a variable type conversion	Right to Left

3	* / %	multiplication division remainder	Left to Right
4	+ -	addition subtraction	Left to Right
5	<< >>	left shift right shift	Left to Right
6	< <= > >=	less than less than or equal to greater than greater than or equal to	Left to Right
7	== !=	equal to not equal to	Left to Right
8	&	bitwise AND	Left to Right
9	^	bitwise EXCLUSIVE OR	Left to Right
10	|	bitwise OR	Left to Right
11	&&	logical AND	Left to Right
12	||	logical OR	Left to Right
13	?:	conditional operator	Left to Right
14	= *= /= %= += -= &= ^= |= <<= >>=	assignment assign multiplication assign division assign remainder assign additon assign subtraction assign bitwise AND assign bitwise XOR assign bitwise OR assign left shift assign right shift	Right to Left
15	,	separator	Left to Right

In the above table, the operator precedence decrease from top to bottom and increase from bottom to top.

Expressions:

Define an Expression

In any programming language, if we want to perform any calculation or to frame any condition etc., we use a set of symbols to perform the task. These set of symbols makes an expression.

In C programming language, an expression is defined as follows...

An expression is a collection of operators and operands that represents a specific value.

In the above definition, operator is a symbol which performs tasks like arithmetic operations, logical operations and conditional operations etc.,
Operands are the values on which the operators perform the task. Here operand can be a direct value or variable or address of memory location.

Expression Types in C

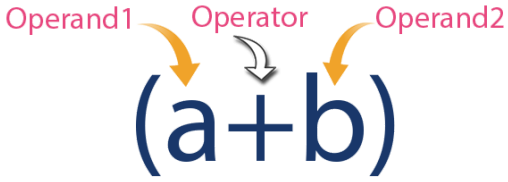
In C programming language, expressions are divided into THREE types. They are as follows...

1.Infix Expression
2.Postfix Expression
3.Prefix Expression

The above classification is based on the operator position in the expression.
1.Infix Expression

The expression in which operator is used between operands is called as infix expression.
The infix expression has the following general structure...

Operand1 Operator Operand2
Example



2.Postfix Expression

The expression in which operator is used after operands is called as postfix expression.
The postfix expression has the following general structure...

Operand1 Operand2 Operator

Example:

3.Prefix Expression

The expression in which operator is used before operands is called as prefix expression.
The prefix expression has the following general structure...

Operator Operand1 Operand2

Example:

Expression Evaluation in C

In C programming language, expression is evaluated based on the operator precedence and associativity. When there are multiple operators in an expression, they are evaluated according to their pr
ecedence and associativity. The operator with higher precedence is evaluated first and the operator with least precedence is evaluated last.

An expression is evaluated based on the precedence and associativity of the operators in that expression.

To understand expression evaluation in c, let us consider the following simple example expression...

10 + 4 * 3 / 2

In the above expression there are three operators +, * and /. Among these three operators, both multiplication and division have same higher precedence and addition has lower precedence. So, according to the operator precedence both multiplication and division are evaluated first and then addition is evaluated. As multiplication and division have same precedence they are evaluated based on the associativity. Here, the associativity of multiplication and division is left to right. So, multiplication is performed first, then division and finally addition. So, the above expression is evaluated in the order of * / and +. It is evaluated as follows...

4 * 3 ====> 12
12 / 2 ===> 6
10 + 6 ===> 16
The expression is evaluated to **16**.

Type Conversions

In a programming language, the expression contains data values of same datatype or different datatypes. When the expression contains similar datatype values then it is evaluated without any problem. But if the expression contains two or more different datatype values then they must be converted to single datatype of destination datatype. Here, destination is the location where the final result of that expression is stored. For example,the multiplication of an integer data value with float data value and storing the result into a float variable. In this case, the integer value must be converted to float value so that the final result is a float datatype value.

In c programming language, the data conversion is performed in two different methods as follows.
Type Conversion
Type Casting

Type Conversion

The type conversion is the process of converting a data value from one datatype to another datatype automatically by the compiler. Sometimes type conversion is also called as **implicit type conversion**. The implicit type conversion is automatically performed by the compiler.
For example, in c programming language, when we assign an integer value to a float variable the integer value automically gets converted to float value by adding decimal value 0. And when a float value is assigned to an integer variable the float value automatically gets converted to integer value by removing the decimal value. To understand more about type conversion observe the following...

```
int i = 10 ;
float x = 15.5 ;
char ch = 'A' ;
```

i = x ; ========> x value 15.5 is converted as 15 and assigned to variable i

x = i ; ========> Here i value 10 is converted as 10.000000 and assigned to variable x

i = ch ; ========> Here the ASCII value of A (65) is assigned to i

Example Program

```
#include <stdio.h>
void main()
{
  int i = 95 ;
  float x = 90.99 ;
```

```
   char ch = 'A' ;
    i = x ;
   printf("i value is %d\n",i);
   x = i ;
   printf("x value is %f\n",x);
   i = ch ;
   printf("i value is %d\n",i);
}
```

Output:

```
i value is 90
x value is 90.000000
i value is 65
```

In the above program, we assign **i = x**, i.e., float variable value is assigned to integer variable. Here, the compiler automatically converts the float value (90.99) into integer value (90) by removing the decimal part of the float value (90.99) and then it is assigned to variable **i**. Similarly when we assign **x = i**, the integer value (90) gets converted to float value (90.000000) by adding zero as decimal part.

Type Casting

Type casting is also called as **explicit type conversion**. Compiler converts data from one datatype to another datatype implicitly. When compiler converts implicitly, there may be a data loss.In such case, we convert the data from one datatype to another datatype using explicit type conversion. To perform this we use the **unary cast operator**. To convert data from one type to another type we specify the target datatype in paranthesis as a prefix to the data value that has to be converted. The general syntax of type casting is as follows...

(TargetDatatype) DataValue

Example

```
int totalMarks = 450, maxMarks = 600 ;
float average ;

average = (float) totalMarks / maxMarks * 100 ;
```

In the above example code, both totalMarks and maxMarks are integer data values. When we perform totalMarks / maxMarks the result is a float value, but the destination (average) datatype is float. So we use type casting to convert totalMarks and maxMarks into float datatype.

Example Program

```
#include <stdio.h>
void main(){
   int a, b, c ;
   float avg ;
   printf("Enter any three integer values : ") ;
```

```
    scanf("%d%d%d", &a, &b, &c) ;
    avg = (a + b + c) / 3 ;
    printf("avg before casting = %f\n",avg);
    avg = (float)(a + b + c) / 3 ;
    printf("avg after casting = %f\n",avg);
}
```

Output:

```
Enter any three integer values : 5 3 2
avg before casting = 3
avg after casting = 3.333333
```

Control structures: Decision statements; if and switch statement; Loop control statements: while, for and do while loops, jump statements, break, continue, goto statements.

Decision Making statements:

In c programming language, the program execution flow is, line by line from top to bottom. That means the c program is executed line by line from the main method. But this type of execution flow may not be suitable for all the program solutions. Sometimes, we make some decisions or we may skip the execution of one or more lines of code. Consider a situation, where we write a program to check whether a student has passed or failed in a particular subject. *Here,* we need to check whether the marks are greater than the pass marks or not. If marks are greater, then we take the decision that the student has passed otherwise failed. To solve such kind of problems in c we use the statements called decision making statements.

Decision making statements are the statements that are used to verify a given condition and decides whether a block of statements gets executed or not based on the condition result.

In c programming language, there are two decision making statements they are as follows...
1. if statement
2. switch statement

1. if statement in c

In c, if statement is used to make decisions based on a condition. The if statement verifies the given condition and decides whether a block of statements are executed or not based on the condition result. In c, if statement is classified into four types as follows...

1. Simple if statement
2. if - else statement
3. Nested if statement
4. if-else-if statement (if-else ladder)

1. Simple if statement

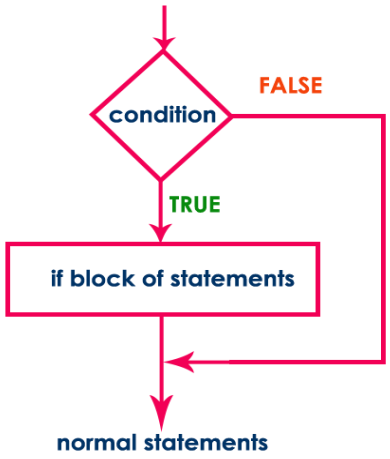
Simple if statement is used to verify the given condition and executes the block of statements based on the condition result. The simple if statement evaluates specified condition. If it is TRUE, it executes the next statement or block of statements. If the condition is FALSE, it

skips the execution of the next statement or block of statements. The general syntax and execution flow of the simple if statement is as follows...

Syntax

```
if ( condition )
{
   ....
   block of statements;
   ....
}
```

Execution flow diagram



Simple if statement is used when we have only one option that is executed or skipped based on a condition.

Example Program | Test whether given number is divisible by 5.

```
#include <stdio.h>
#include<conio.h>
void main(){
  int n ;
  clrscr() ;
  printf("Enter any integer number: ") ;
  scanf("%d", &n) ;
  if ( n%5 == 0 )
    printf("Given number is divisible by 5\n") ;
  printf("statement does not belong to if!!!") ;
}
```

Output 1:

Enter any integer number: 100
Given number is divisible by 5
statement does not belong to if!!!

Output 2:

Enter any integer number: 99
statement does not belong to if!!!

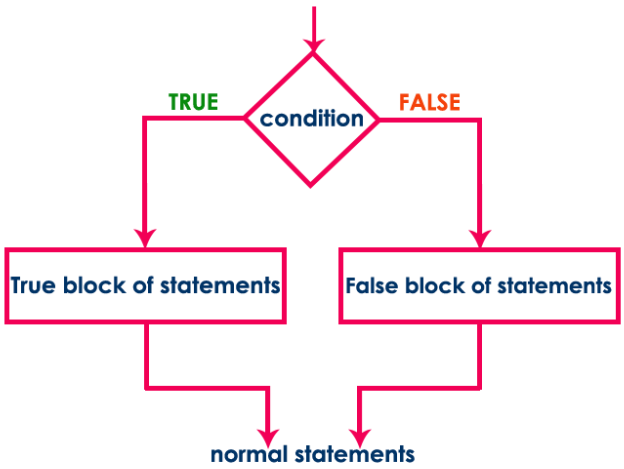
2. if - else statement

The if - else statement is used to verify the given condition and executes only one out of the two blocks of statements based on the condition result. The if-else statement evaluates the specified condition. If it is TRUE, it executes a block of statements (True block). If the condition is FALSE, it executes another block of statements (False block). The general syntax and execution flow of the if-else statement is as follows...

Syntax

```
if ( condition )
{
   ....
   True block of statements;
   ....
}
else
{
   ....
   False block of statements;
   ....
}
```

Execution flow diagram



The if-else statement is used when we have two options and only one option has to be executed based on a condition result (TRUE or FALSE).

Example Program | Test whether given number is even or odd.

```
#include <stdio.h>
#include<conio.h>
void main(){
  int n ;
  clrscr() ;
  printf("Enter any integer number: ") ;
  scanf("%d", &n) ;
  if ( n%2 == 0 )
    printf("Given number is EVEN\n") ;
  else
    printf("Given number is ODD\n") ;
}
```

Output 1:

Enter any integer number: 100
Given number is EVEN

Output 2:

Enter any integer number: 99
Given number is ODD

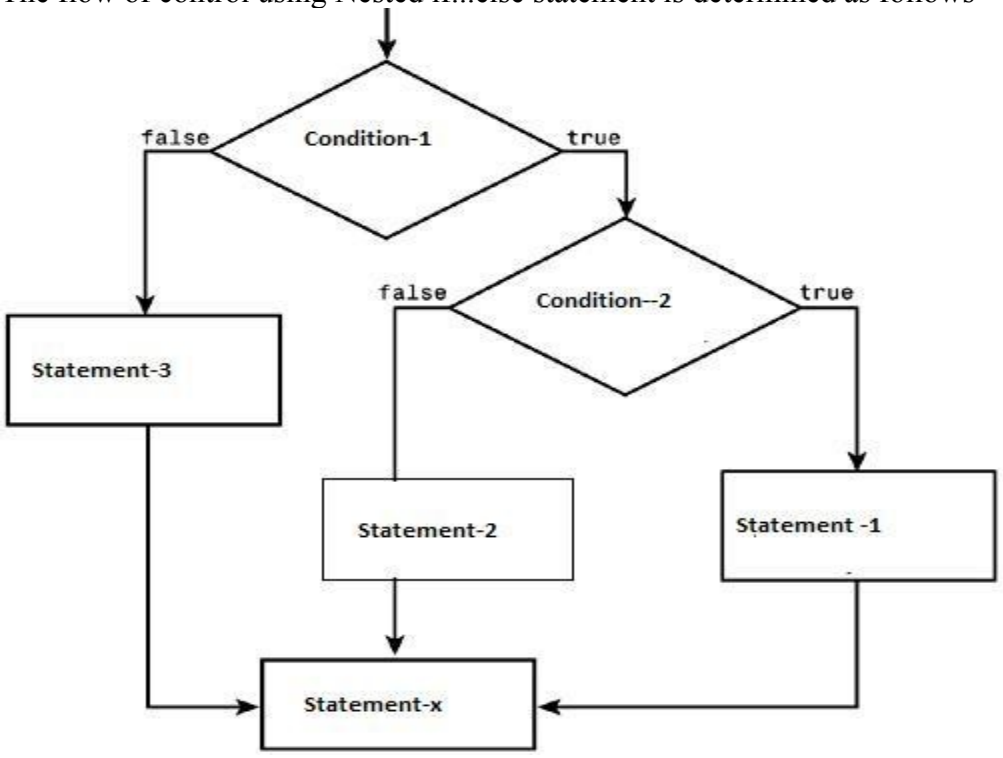
3. Nested if statement

Writing a if statement inside another if statement is called nested if statement. The general syntax of the nested if statement is as follows...

Syntax

```
if ( condition1 )
{
   if ( condition2 )
   {
      ....
      True block of statements 1;
   }
   ....
}
else
{
    False block of condition1;
}
```

The nested if statement can be defined using any combination of simple if & if-else statements. The flow of control using Nested if...else statement is determined as follows



Whenever nested if…else statement is encountered, first <condition1> is tested. It returns either true or false.

If condition1 (or outer condition) is false, then the control transfers to else-body (if exists) by skipping if-body.

If condition1 (or outer condition) is true, then condition2 (or inner condition) is tested. If the condition2 is true, if-body gets executed. Otherwise, the else-body that is inside of if statement gets executed.

Example Program | Test whether given number is even or odd if it is below 100.

```
#include <stdio.h>
#include<conio.h>
void main(){
  int n ;
  clrscr() ;
  printf("Enter any integer number: ") ;
  scanf("%d", &n) ;
  if ( n < 100 )
  {
    printf("Given number is below 100\n") ;
    if( n%2 == 0)
       printf("And it is EVEN") ;
    else
       printf("And it is ODD") ;
  }
  else
     printf("Given number is not below 100") ;
}
```

Output 1:

Enter any integer number: 55
Given number is below 100
And it is ODD

Output 2:

Enter any integer number: 999
Given number is not below 100

4. if - else - if statement (if-else ladder)

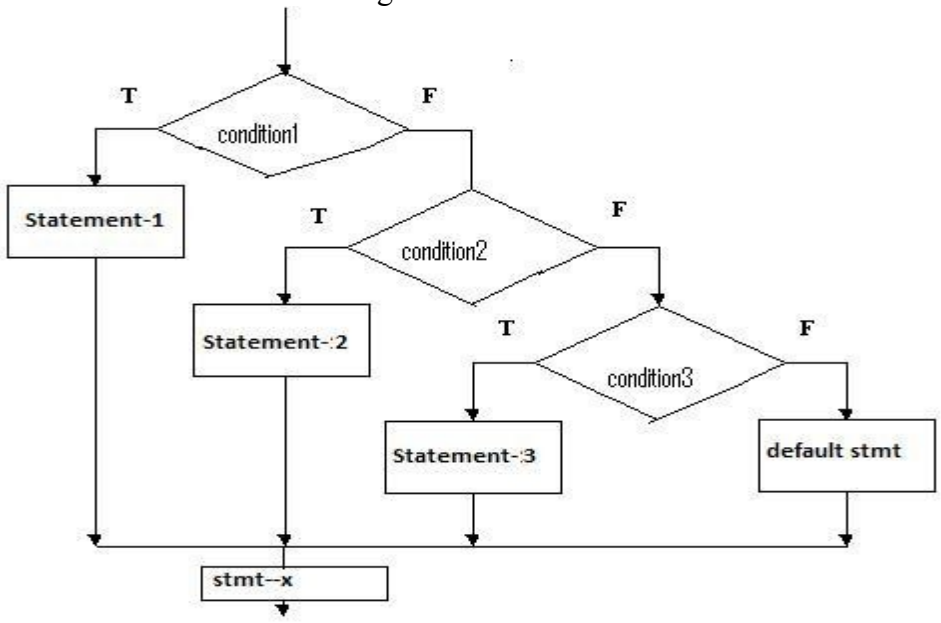
Writing a if statement inside else of a if statement is called if - else - if statement. The general syntax of the if-else-if statement is as follows...

Syntax

```
if ( condition1 )
{
    ....
    True block of statements1;
    ....
}
else if ( condition2 )
{
    False block of condition1;
    &
    True block of condition2
}
```

The if-else-if statement can be defined using any combination of simple if & if-else statements.

The flow of control using else--- if ladder statement is determined as follows:



Whenever else if ladder is encountered, condition1 is tested first. If it is true, the statement 1 gets executed. After then the control transfers to *stmt-x*.

If *condition1* is false, then *condition2* is tested. If *condition2* is false, the other conditions are tested. If all are false, the default stmt at the end gets executed. After then the control transfers to *stmt-x*.

If any one of all conditions is true, then the body associated with it gets executed. After then the control transfers to *stmt-x*.

Example Program | Find the largest of three numbers.

```
#include <stdio.h>
#include<conio.h>
void main(){
  int a, b, c ;
  clrscr() ;

  printf("Enter any three integer numbers: ") ;
  scanf("%d%d%d", &a, &b, &c) ;

  if( a>=b && a>=c)
     printf("%d is the largest number", a) ;

   else if (b>=a && b>=c)
     printf("%d is the largest number", b) ;

   else
     printf("%d is the largest number", c) ;
}
```

Output 1:

Enter any three integer numbers: 55 60 20
60 is the largest number

MOST IMPORTANT POINTS TO BE REMEMBERED

When we use conditional control statement like if statement, condition might be an expression evaluated to a numerical value, a variable or a direct numerical value. If the expression value or direct value is zero the conditon becomes FALSE otherwise becomes TRUE.

To understand more consider the following statements...
if(10) - is TRUE
if(x) - is FALSE if x value is zero otherwise TRUE
if(a+b) - is FALSE if a+b value is zero otherwise TRUE
if(a = 99) - is TRUE because a value is non-zero
if(10, 5, 0) - is FALSE because it considers last value
if(0) - is FALSE
if(a=10, b=15, c=0) - is FALSE because last value is zero

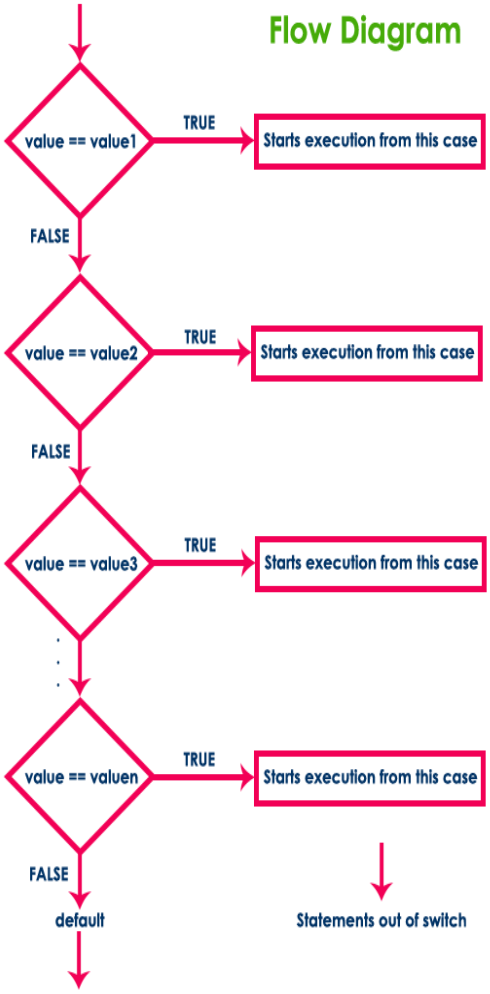
Switch statement:

Consider a situation in which we have more number of options out of which we need to select only one option that is to be executed. Such kind of problems can be solved using **nested if** statement. But as the number of options increases, the complexity of the program also gets increased. This type of problems can be solved very easily using **switch** statement. Using switch statement, one can select only one option from more number of options very easily. In switch statement, we provide a value that is to be compared with a value associated with each option. Whenever the given value matches with the value associated with an option, the execution starts from that option. In switch statement every option is defined as a **case**.

The switch statement has the following syntax and execution flow diagram...

Syntax

```
switch ( expression or value )
{
      case  value1:  set of statements;
                     ....
      case  value2:  set of statements;
                     ....
      case  value3:  set of statements;
                     ....
      case  value4:  set of statements;
                     ....
      case  value5:  set of statements;
                     ....
      .
      .
      default: set of statements;
}
```



The switch statement contains one or more number of cases and each case has a value associated with it. At first switch statement compares the first case value with the switchValue, if it gets matched the execution starts from the first case. If it doesn't match the switch statement compares the second case value with the switchValue and if it is matched the execution starts from the second case. This process continues until it finds a match. If no case value matches with the switchValue specified in the switch statement, then a special case called **default** is executed. When a case value matches with the switchValue, the execution starts from that particular case. This execution flow continues with next case statements also. To avoid this, we use "**break**" statement at the end of each case. That means the **break** statement is used to terminate the switch statement. However it is optional.

Example Program | Display pressed digit in words.

```
#include <stdio.h>
#include<conio.h>
void main(){
  int n ;
  clrscr() ;

  printf("Enter any digit: ") ;
  scanf("%d", &n) ;

  switch( n )
  {
              case 0: printf("ZERO") ;
              break ;
              case 1: printf("ONE") ;
              break ;
              case 2: printf("TWO") ;
              break ;
              case 3: printf("THREE") ;
              break ;
              case 4: printf("FOUR") ;
              break ;
              case 5: printf("FIVE") ;
              break ;
              case 6: printf("SIX") ;
              break ;
              case 7: printf("SEVEN") ;
              break ;
              case 8: printf("EIGHT") ;
              break ;
              case 9: printf("NINE") ;
              break ;
              default: printf("Not a Digit") ;
  }
  getch() ;
}
```

Output 1:

Enter any digit: 5

FIVE

Output 2:

Enter any digit: 15
Not a Digit

MOST IMPORTANT POINTS TO BE REMEMBERED

When we use switch statement, we must follow the following...
Both switch and case are keywords so they must be used only in lower case letters.
The data type of case value and the value specified in switch statement must be same.
switch and case values must be either integer or character but not float or string.
A switch statement can contain any number of cases.
The keyword case and its value must be separated with a white space.
The case values need not be defined in sequence, they can be in any order.
The default case is optional and it can be defined anywhere inside the switch statement.
The switch value might be a direct value, a variable or an expression.

Faculty Name: N.SANDHYA RANI **Subject** :PPS
Topic: Loop Control Statements :While

Unit No :2
Lecture No :L26
Book Reference: T1

Loop control statements:

Consider a situation in which we execute a single statement or block of statements repeatedly for required number of times. Such kind of problems can be solved using **looping** statements in C. For example, assume a situation where we print a message for 100 times. If we want to perform that task without using looping statements, we have to either write 100 printf statements or we have to write the same message for 100 times in a single printf statement. Both are complex methods. The same task can be performed very easily using looping statements.
The looping statements are used to execute a single statement or block of statements repeatedly until the given condition is FALSE.

C language provides three looping statements...
1.while statement
2.do-while statement
3.for statement

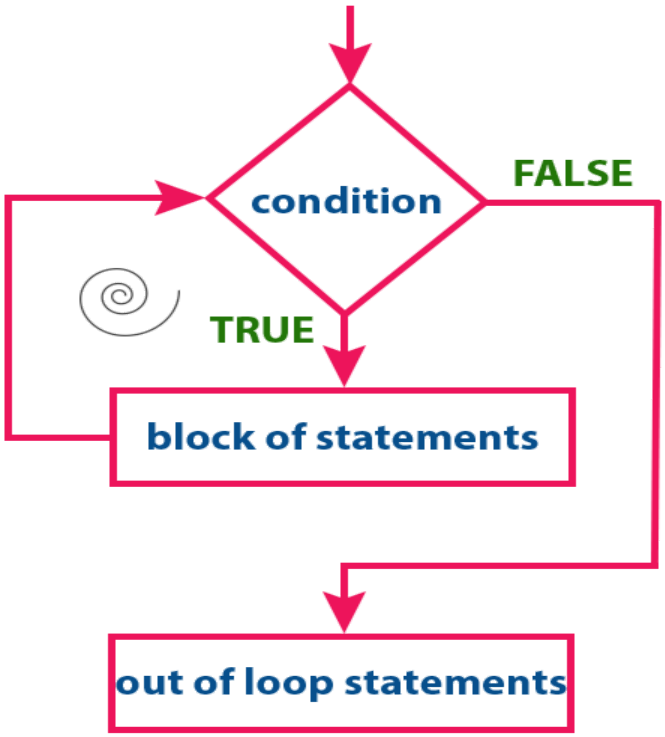
1. While statement:

The while statement is used to execute a single statement or block of statements repeatedly as long as the given condition is TRUE. The while statement is also known as **Entry control looping statement**. The while statement has the following syntax...

Syntax:

```
while( condition )
{
    ...
    block of statements;
    ...
}
```

The while statement has the following execution flow diagram...



At first, the given condition is evaluated. If the condition is TRUE, the single statement or block of statements gets executed. Once the execution gets completed the condition is evaluated again. If it is TRUE, again the same statements gets executed. The same process is repeated until the condition is evaluated to FALSE. Whenever the condition is evaluated to FALSE, the execution control moves out of the while block.

Example Program | Program to display even numbers upto 10.

```
#include <stdio.h>
#include<conio.h>
void main(){
  int n = 0;
  clrscr() ;
  printf("Even numbers upto 10\n");

  while( n <= 10 )
  {
    if( n%2 == 0)
      printf("%d\t", n) ;
    n++ ;
  }

  getch() ;
}
```

Output 1:

Even numbers upto 10
0 2 4 6 8 10

MOST IMPORTANT POINTS TO BE REMEMBERED

When we use while statement, we must follow the following...
while is a keyword so it must be used only in lower case letters.
If the condition contains variable, it must be assigned a value before it is used.
The value of the variable used in condition must be modified according to the requirement inside the while block.
In while statement, the condition may be a direct integer value, a variable or a condition.
A while statement can be an empty statement.

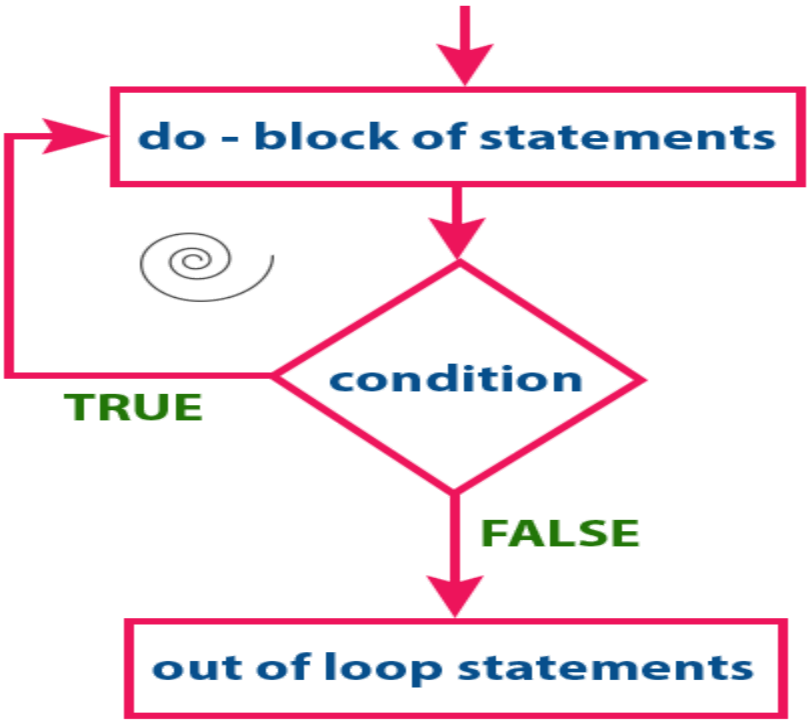
Faculty Name: N.SANDHYA RANI **Subject** :PPS **Topic:** Loop Control Statements : Do-While ,for	**Unit No** :2 **Lecture No** :L27 **Book Reference**: T1

2. do-While statement:

The do-while statement is used to execute a single statement or block of statements repeatedly as long as given the condition is TRUE. The do-while statement is also known as **Exit control looping statement**. The do-while statement has the following syntax...

```
Syntax:

do
{
    ...
    block of statements;
    ...
} while( condition ) ;
```

The do-while statement has the following execution flow diagram.



At first, the single statement or block of statements which are defined in **do** block are executed. After execution of do block, the given condition gets evaluated. If the condition is evaluated to TRUE, the single statement or block of statements of do block are executed again. Once the execution gets completed again the condition is evaluated. If it is TRUE, again the same statements are executed. The same process is repeated until the condition is evaluated to FALSE. Whenever the condition is evaluated to FALSE, the execution control moves out of the while block.

Example Program | Program to display even numbers upto 10.

```
#include <stdio.h>
#include<conio.h>
void main(){
   int n = 0;
   clrscr() ;
   printf("Even numbers upto 10\n");

   do
   {
      if( n%2 == 0)
         printf("%d\t", n) ;
      n++ ;
   }while( n <= 10 ) ;

   getch() ;
}
```

Output 1:

Even numbers upto 10
0 2 4 6 8 10

MOST IMPORTANT POINTS TO BE REMEMBERED

When we use do-while statement, we must follow the following...
Both do and while are keywords so they must be used only in lower case letters.
If the condition contains variable, it must be assigned a value before it is used.
The value of the variable used in condition must be modified according to the requirement inside the do block.
In do-while statement the condition may be, a direct integer value, a variable or a condition.
A do-while statement can be an empty statement.
In do-while, the block of statements are executed at least once.

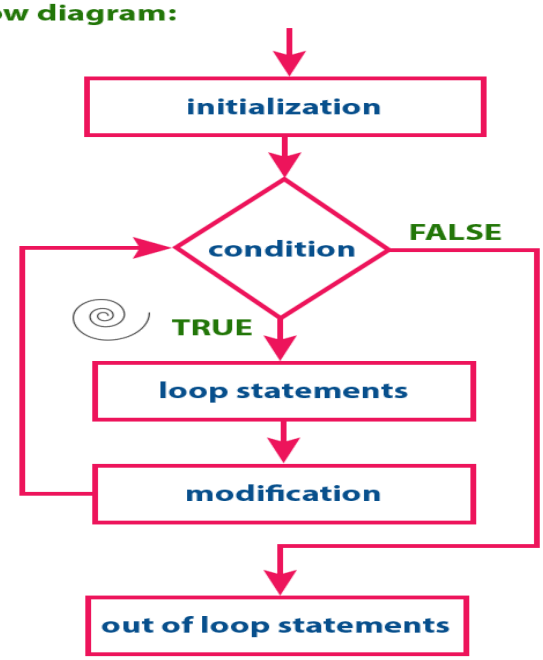
3.for statement:

The for statement is used to execute a single statement or a block of statements repeatedly as long as the given condition is TRUE. The for statement has the following syntax and execution flow diagram...

Syntax:

```
for( initialization ; condition ; modification )
{
    ...
    block of statements;
    ...
}
```

Execution flow diagram:



At first, the for statement executes initialization followed by condition evaluation. If the condition is evaluated to TRUE, the single statement or block of statements of for statement are executed. Once the execution gets completed, the modification statement is executed and again the condition is evaluated. If it is TRUE, again the same statements are executed. The same process is repeated until the condition is evaluated to FALSE. Whenever the condition is evaluated to FALSE, the execution control moves out of the for block.

Example Program | Program to display even numbers upto 10.

```
#include <stdio.h>
#include<conio.h>
void main(){
   int n ;
   clrscr() ;
   printf("Even numbers upto 10\n");

   for( n = 0 ; n <= 10 ; n++ )
   {
      if( n%2 == 0)
         printf("%d\t", n) ;
   }

   getch() ;
}
```

Output 1:

Even numbers upto 10
0 2 4 6 8 10

MOST IMPORTANT POINTS TO BE REMEMBERED

When we use for statement, we must follow the following.
for is a keyword so it must be used only in lower case letters.
Every for statement must be provided with initialization, condition and modification (They can be empty but must be separated with ";")
Ex: for (; ;) or for (; condition ; modification) or for (; condition ;)
In for statement, the condition may be a direct integer value, a variable or a condition.
The for statement can be an empty statement.

Faculty Name: N.SANDHYA RANI **Subject** : PPS
Topic: Braeak, Jump , Continue , goto statement

Unit No :2
Lecture No :L28
Book Reference: T1

Break,continue and goto statements:

In c, there are control statements which does not need any condition to control the program execution flow. These control statements are called as **unconditional control statements**. C programming language provides the following unconditional control statements...

1.break
2.continue
3.goto

The above three statements does not need any condition to control the program execution flow.

1.break statement:

In C, the **break statement** is used to perform the following two things.

break statement is used to terminate switch case statement
break statement is also used to terminate looping statements like while, do-while and for.

When a **break** statement is encountered inside the switch case statement, the execution control moves out of the switch statement directly. For example consider the following program.

Example Program | Program to perform all arithmetic operations using switch statement.

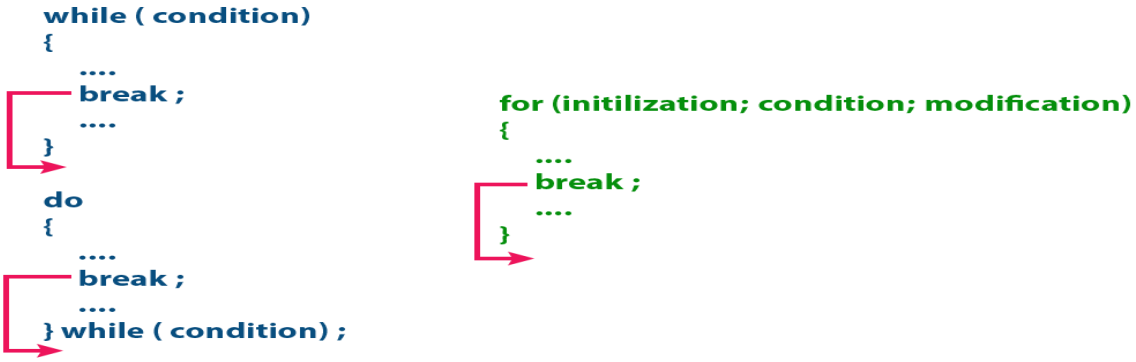
```
#include <stdio.h>
#include<conio.h>
void main(){
  int number1, number2, result ;
  char operator;
  clrscr() ;
  printf("Enter any two integer numbers: ") ;
  scanf("%d%d", &number1, &number2) ;
  printf("Please enter any arithmetic operator: ");
  operator = getchar();
  switch(operator)
  {
    case '+': result = number1 + number2 ;
              printf("Addition = %d", result) ;
              break;
    case '-': result = number1 - number2 ;
              printf("Subtraction = %d", result) ;
              break;
    case '*': result = number1 * number2 ;
              printf("Multiplication = %d", result) ;
              break;
```

```
    case '/': result = number1 / number2 ;
              printf("Division = %d", result) ;
              break;
    case '%': result = number1 % number2 ;
              printf("Remainder = %d", result) ;
              break;
    default: printf("\nWrong selection!!!") ;
  }
  getch() ;
}
```

Output

```
Enter any two integer numbers: 50 30
Please enter any arithmetic operator: *
Multiplication = 1500
```

When the **break** statement is encountered inside the looping statement, the execution control moves out of the looping statements. The **break** statement execution is as shown in the following figure.



For example, consider the following example program...

Example Program for *break statement*.

```
#include <stdio.h>
#include<conio.h>
void main(){
  char ch ;
  clrscr() ;
  do
  {
    printf("Enter Y / N : ") ;
    scanf("%c", &ch) ;
    if(ch == 'Y')
    {
      printf("Okay!!! Repeat again !!!\n") ;
    }
    else
```

```
    {
      printf("Okay !!! Breaking the loop !!!") ;
      break ;
    }
  } while( 1 ) ;
  getch() ;
}
```

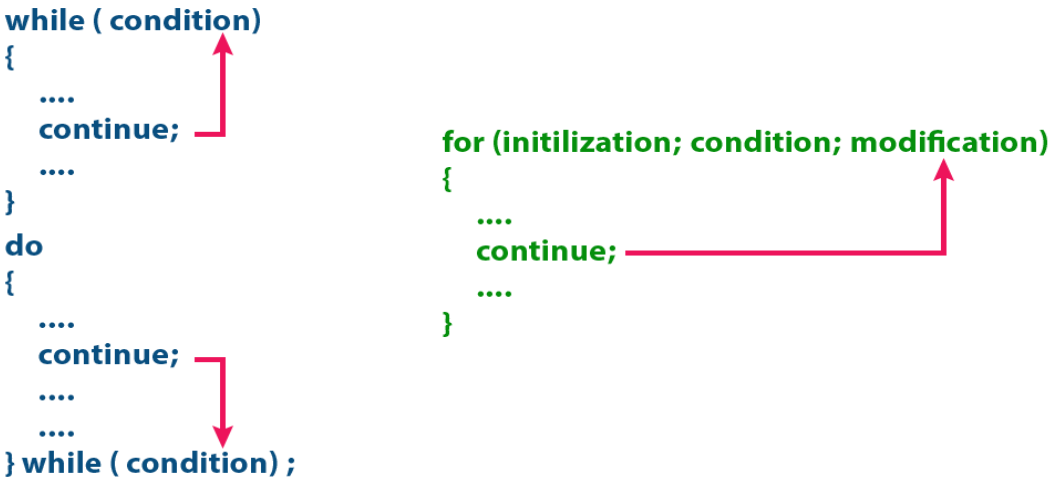
Output:

Enter Y / N : Y
Okay!!! Repeat again !!!
Enter Y / N : Y
Okay!!! Repeat again !!!
Enter Y / N : N
Okay !!! Breaking the loop !!!

2.Continue statement:

The **continue** statement is used to move the program execution control to the beginning of looping statement. When **continue** statement is encountered in a looping statement, the execution control skips the rest of the statements in the looping block and directly jumps to the beginning of the loop. The **continue** statement can be used with looping statements like while, do-while and for.

When we use **continue** statement with **while** and **do-while** statements the execution control directly jumps to the condition. When we use **continue** statement with **for** statement the execution control directly jumps to the modification portion (increment / decrement / any modification) of the for loop. The **continue** statement execution is as shown in the following figure.



Example Program | Program to illustrate continue statement.

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

```
#include<conio.h>
void main(){
   int number ;
   clrscr() ;
   while( 1 )
   {
       printf("Enter any integer number: ") ;
       scanf("%d", &number) ;
       if(number%2 == 0)
       {
           printf("Entered number is EVEN!!! Try another number!!!\n") ;
           continue ;
       }
       else
       {
           printf("You have entered ODD number!!! Bye!!!") ;
           exit(0) ;
       }
   }
   getch() ;
}
```

Output

Enter any integer numbers: 50
Entered number is EVEN!!! Try another number!!!
Enter any integer numbers: 100
Entered number is EVEN!!! Try another number!!!
Enter any integer numbers: 10
Entered number is EVEN!!! Try another number!!!
Enter any integer number: 15
You have entered ODD number!!! Bye!!!

3.goto statement:

The **goto** statement is used to jump from one line to another line in the program.
Using **goto** statement we can jump from top to bottom or bottom to top. To jump from one line to another line, the goto statement requires a **lable**. Lable is a name given to the instruction or line in the program. When we use **goto** statement in the program, the execution control directly jumps to the line with specified lable.

Example Program for *goto statement*.

```
#include <stdio.h>
#include<conio.h>
void main(){
   clrscr() ;
   printf("We are at first printf statement!!!\n") ;
   goto last ;
   printf("We are at second printf statement!!!\n") ;
   printf("We are at third printf statement!!!\n") ;
   last: printf("We are at last printf statement!!!\n") ;
   getch() ;
}
```

Output

We are at first printf statement!!!
We are at last printf statement!!!

MOST IMPORTANT POINTS TO BE REMEMBERED

When we use break, continue and goto statements, we must follow the following...

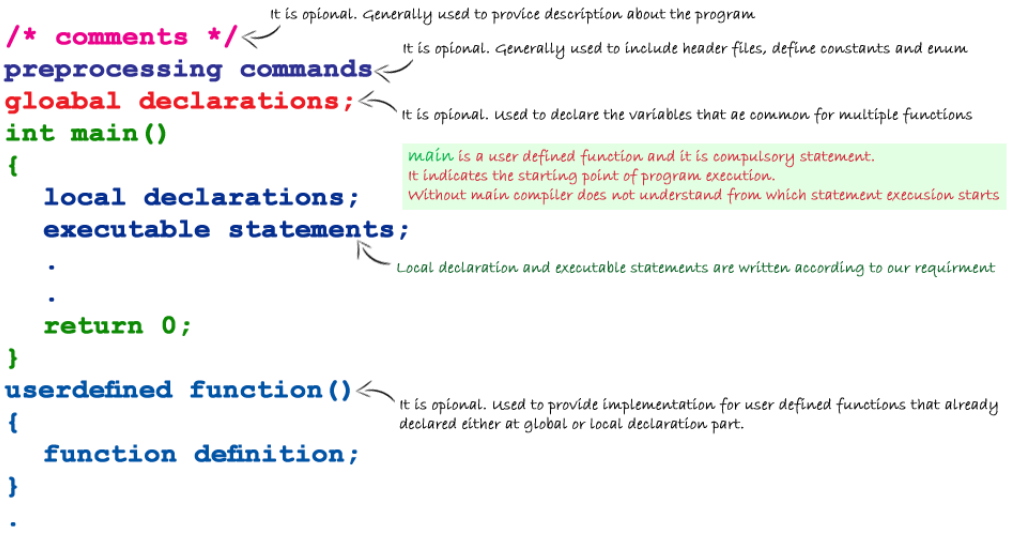
- The **break** is a keyword so it must be used only in lower case letters.
- The **break** statement can not be used with **if** statement.
- The **break** statement can be used only in switch case and looping statements.
- The **break** statement can be used with **if** statement, only if that **if statement** is written inside the switch case or looping statements.
- The **continue** is a keyword so it must be used only in lower case letters.
- The **continue** statement is used only within **looping statements**.
- The **continue** statement can be used with **if** statement, only if that **if statement** is written inside the looping statements.
- The **goto** is a keyword so it must be used only in lower case letters.
- The **goto** statement must requires a **lable**.
- The **goto** statement can be used with any statement like if, switch, while, do-while and for etc,.

Faculty Name: N.Thulasi Chitra
Subject :PPS
Topic: Introduction to C language: Structure of C programs

Unit No :2
Lecture No :L17
Book Reference:T1

Basic structure of C programs:

C is a structured programming language. Every c program and its statements must be in a particular structure. Every c program has the following general structure...



Line 1: Comments - They are ignored by the compiler

This section is used to provide small description of the program. The comment lines are simply ignored by the compiler, that means they are not executed. In C, there are two types of comments.

1.**Single Line Comments:** Single line comment begins with // symbol. We can write any number of single line comments.

2.**Multiple Lines Comments:** Multiple lines comment begins with /* symbol and ends with */. We can write any number of multiple lines comments in a program.

In a C program, the comment lines are optional. Based on the requirement, we write the comments. All the comment lines in a C program just provide the guidelines to understand the program and its code.

Line 2: Preprocessing Commands

Pre-processing commands are used to include header files and to define constants. We use **#include** statement to include header file into our program. We use **#define** statement to define a constant. The pre-processing statements are used according to the requirment. If we don't need

any header file, then no need to write #include statement. If we don't need any constant, then no need to write #define statement.

Line 3: Global Declaration

Global declaration is used to define the global variables, which are common for all the functions after its declaration. We also use the global declaration to declare functions. This global declaration is used based on the requirement.

Line 4: int main()

Every C program must write this statement. This statement (main) specifies the starting point of the C program execution. Here, main is a user defined method which tells the compiler that this is the starting point of the program execution. Here, **int** is a datatype of a value that is going to return to the Operating System after completing the main method execution. If we don't want to return any value, we can use it as **void**.

Line 5: Open Brase ({)

The open brase indicates the begining of the block which belongs to the main method. In C program, every block begins with '{' symbol.

Line 6: Local Declaration

In this section, we declare the variables and functions that are local to the function or block in which they are declared. The variables which are declared in this section are valid only within the function or block in which they are declared.

Line 7: Executable statements

In this section, we write the statements which perform tasks like reading data, displaying result, calculations etc., All the statements in this section are written according to the requirment.

Line 9: Closing Brase (})

The close brase indicates the end of the block which belongs to the main method. In C program every block ends with '}' symbol.

Line 10, 11, 12, ...: Userdefined function()

This is the place where we implement the userdefined functions. The userdefined function implementation can also be performed before the main method. In this case, the user defined function need not to be declared. Directly it can be implemented, but it must be before the main method. In a program, we can define as many userdefined functions as we want. Every user defined function needs a function call to execute its statements.

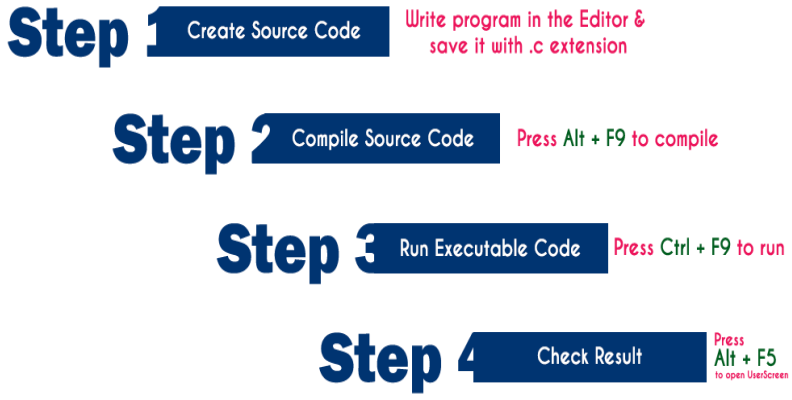
1.Every executable statement must end with semicolon symbol (;).
2.Every C program must contain exactly one main method (Starting point of the program execution).
3.All the system defined words (keywords) must be used in lowercase letters.
4.Keywords can not be used as user defined names(identifiers).
5.For every open brase ({), there must be respective closing brase (}).
6.Every variable must be declared before it is used.

Process of compiling and running a C program:

Generally, the programs created using programming languages like C, C++, Java etc., are written using high level language like English. But, computer cannot understand the high level language. It can understand only low level language. So, the program written in high level language needs to be converted into low level language to make it understandable for the computer. This conversion is performed using either Interpreter or Compiler.

Popular programming languages like C, C++, Java etc., use compiler to convert high level language instructions into low level language instructions. Compiler is a program that converts high level language instructions into low level language instructions. Generally, compiler performs two things, first it verifies the program errors, if errors are found, it returns list of errors otherwise it converts the complete code into low level language.

To create and execute C programs in Windows Operating System, we need to install Turbo C software. We use the following steps to create and execute C programs in Windows OS…



Step 1: Creating Source Code

Source code is a file with C programming instructions in high level language. To create source code, we use any text editor to write the program instructions. The instructions written in the source code must follow the C programming language rules. The following steps are used to create source code file in Windows OS…

- Click on Start button

- Select Run
- Type cmd and press Enter
- Type cd c:\TC\bin in the command prompt and press Enter
- Type TC press Enter
- Click on File -> New in C Editor window
- Type the program
- Save it as FileName.c (Use shortcut key F2 to save)

Step 2: Compile Source Code (Alt + F9)
Compilation is the process of converting high level language instructions into low level language instructions. We use the shortcut key Alt + F9 to compile a C program in Turbo C.
Compilation is the process of converting high level language instructions into low level language instructions.
Whenever we press Alt + F9, the source file is going to be submitted to the Compiler. On receiving a source file, the compiler first checks for the Errors. If there are any Errors then compiler returns List of Errors, if there are no errors then the source code is converted into object code and stores it as file with .obj extension. Then the object code is given to the Linker. The Linker combines both the object code and specified header file code and generates an Executable file with .exe extension.
Step 3: Executing / Running Executable File (Ctrl + F9)
After completing compilation successfully, an executable file is created with .exe extension. The processor can understand this .exe file content so that it can perform the task specified in the source file.

We use a shortcut key Ctrl + F9 to run a C program. Whenever we press Ctrl + F9, the .exe file is submitted to the CPU. On receiving .exefile, CPU performs the task according to the instruction written in the file. The result generated from the execution is placed in a window called User Screen.
Step 4: Check Result (Alt + F5)
After running the program, the result is placed into User Screen. Just we need to open the User Screen to check the result of the program execution. We use the shortcut key Alt + F5 to open the User Screen and check the result.

Faculty Name: N.Thulasi Chitra
Subject :PPS
Topic: Data types

Unit No :2
Lecture No :L18
Book Reference:T1

DataTypes:

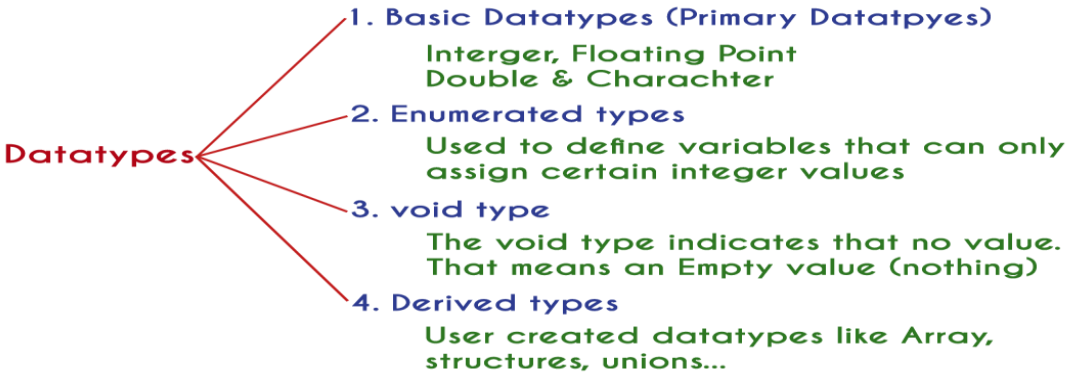
Data used in c program is classified into different types based on its properties. In c programming langauge, datatype can be defined as a set of values with similar characteristics. All the values in a datatype have the same properties.

Datatypes in c programming language are used to specify what kind of value can be stored in a variable. The memory size and type of value of a variable are determined by varible datatype. In a c program, each variable or constant or array must have a datatype and this datatype specifies how much memory is to be allocated and what type of values are to be stored in that variable or constant or array. The formal definition of datatype is as follows...

Datatype is a set of value with predefined characteristics. Datatypes are used to declare variable, constants, arrays, pointers and functions.

In c programming language, datatypes are classified as follows...

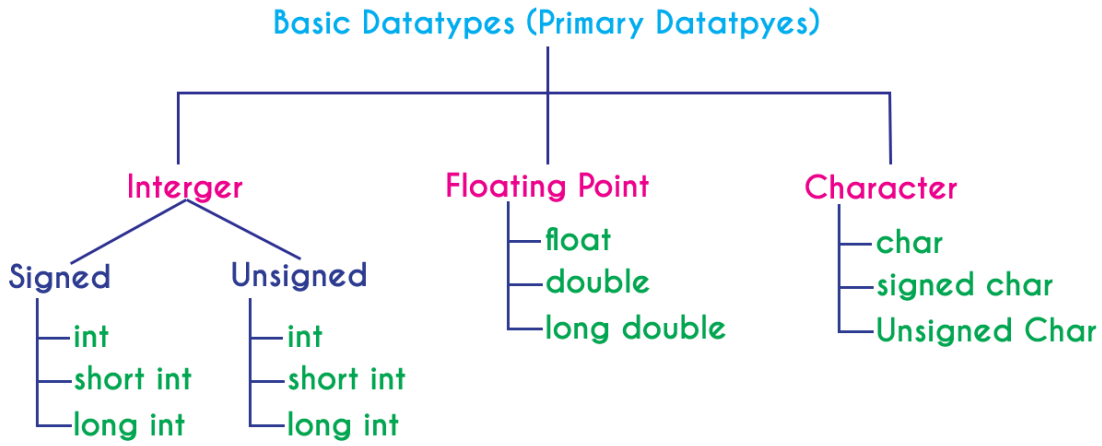
1.Primary Datatypes (Basic Datatypes OR Predefined Datatypes)
2.Derived Datatypes (Secondary Datatypes OR Userdefined Datatypes)
3.Enumeration Datatypes
4.Void Datatype



Primary Datatypes

The primary datatypes in C programming language are the basic datatypes. All the primary datatypes are already defined in the system. Primary datatypes are also called as Built-In datatypes. The following are the primary datatypes in c programming lanuage...

1.Integer Datatype
2.Floating Point Datatype
3.Double Datatype
4.Character Datatype



Integer Datatype

Integer datatype is a set of whole numbers. Every integer value does not have the decimal value. We use the keyword "**int**" to represent integer datatype in c. We use the keyword int to declare the variables and to specify return type of a function. The integer datatype is used with different type modifiers like short, long, signed and unsigned. The following table provides complete details about integer datatype.

Type	Size (Bytes)	Range	Specifier
int (signed short int)	2	-32768 to +32767	%d
short int (signed short int)	2	-32768 to +32767	%d
long int (signed long int)	4	-2,147,483,648 to +2,147,483,647	%d
unsigned int (unsigned short int)	2	0 to 65535	%u
unsigned long int	4	0 to 4,294,967,295	%u

Floating Point Datatypes

Floating point datatypes are set of numbers with decimal value. Every floating point value must contain the decimal value. The floating point datatype has two variants...

- float
- double

We use the keyword "**float**" to represent floating point datatype and "**double**" to represent double datatype in c. Both float and double are similar but they differ in number of decimal places. The float value contains 6 decimal places whereas double value contains 15 or 19 decimal places. The following table provides complete details about floating point datatypes.

Type	Size (Bytes)	Range	Specifier
float	4	1.2E - 38 to 3.4E + 38	%f
double	8	2.3E-308 to 1.7E+308	%lf
long double	10	3.4E-4932 to 1.1E+4932	%lf

Character Datatype

Character datatype is a set of characters enclosed in single quotations. The following table provides complete details about character datatype.

Type	Size (Bytes)	Range	Specifier
char (signed char)	1	-128 to +127	%c
unsigned char	1	0 to 255	%c

The following table provides complete information about all the datatypes in c programming language...

	Integer	Floating Point	Double	Character
What is it?	Numbers without decimal value	Numbers with decimal value	Numbers with decimal value	Any symbol enclosed in single quotation
Keyword	int	float	double	char
Memory Size	2 or 4 Bytes	4 Bytes	8 or 10 Bytes	1 Byte
Range	-32768 to +32767 (or) 0 to 65535 (Incase of 2 bytes only)	1.2E - 38 to 3.4E + 38	2.3E-308 to 1.7E+308	-128 to + 127 (or) 0 to 255
Type Specifier	%d or %i or %u	%f	%ld	%c or %s
Type Modifier	short, long signed, unsigned	No modifiers	long	signed, unsigned
Type Qualifier	const, volatile	const, volatile	const, volatil	const, volatile

void Datatype

The void datatype means nothing or no value. Generally, void is used to specify a function which does not return any value. We also use the void datatype to specify empty parameters of a function.

Enumerated Datatype

An enumerated datatype is a user-defined data type that consists of integer constants and each integer constant is given a name. The keyword "**enum**" is used to define enumerated datatype.

Derived Datatypes

Derived datatypes are user-defined data types. The derived datatypes are also called as user defined datatypes or secondary datatypes. In c programming language, the derived datatypes are created using the following concepts...

- Arrays
- Structures
- Unions
- Enumeration

Faculty Name: N.Thulasi Chitra
Subject :PPS
Topic: data inputs, output statements

Unit No :2
Lecture No :L19
Book Reference:T1

Input Functions in C:

C programming language provides built-in functions to perform input operations. The input opearations are used to read user values (input) from keyboard. C programming language provides the following built-in input functions...

1. **scanf()**
2. **getchar()**
3. **getch()**
4. **gets()**
5. **fscanf()**

scanf() function:

The scanf() function is used to read multiple data values of different data types from the keyboard. The scanf() function is built-in function defined in a header file called "**stdio.h**". When we want to use scanf() function in our program, we need to include the respective header file (stdio.h) using **#include** statement. The scanf() function has the following syntax...

Syntax:

```
scanf("format strings",&variableNames);
```

Example Program

```
#include <stdio.h>

void main(){

 int i;

  printf("\nEnter any integer value: ");

scanf("%d",&i);

 printf("\nYou have entered %d number",i);

}
```

Output:

Enter any integer value: 55
You have entered 55 number

In the above example program, we used the scanf() function to read an integer value from the keyboard and store it into variable 'i'.

The scanf function also used to read multiple data values of different or same data types. Consider the following example program...

```
#include <stdio.h>

void main(){

int i;

float x;

printf("\nEnter one integer followed by one float value : ");

scanf("%d%f",&i, &x);

 printf("\ninteger = %d, float = %f",i, x);    }
```

Output:

Enter one integer followed by one float value : 20 30.5
integer = 20, float = 30.5

In the above example program, we used the scanf() function to read one integer value and one float value from the keyboard. Here '**i**' is an integer variable so we have used format string %d, and '**x**' is a float variable so we have used format string %f.

The scanf() function returns an integer value equal to the total number of input values read using scanf function.

Example Program

```
#include <stdio.h>

void main(){

   int i,a,b;

   float x;

   printf("\nEnter two integers and one float : ");

   i = scanf("%d%d%f",&a, &b, &x);

   printf("\nTotal inputs read : %d",i);

}
```

Output:

Enter two integers and one float : 10 20 55.5
Total inputs read : 3

getchar() function

The getchar() function is used to read a character from the keyboard and return it to the program. This function is used to read only single character. To read multiple characters we need to write multiple times or use a looping statement. Consider the following example program...

```
#include <stdio.h>

void main(){

   char ch;
```

```
    printf("\nEnter any character : ");

    ch = getchar();

    printf("\nYou have entered : %c",ch);

}
```

Output:

```
Enter any character : A
You have entered : A
```

getch() function

The getch() function is similar to getchar function. The getch() function is used to read a character from the keyboard and return it to the program. This function is used to read only single character. To read multiple characters we need to write multiple times or use a looping statement. Consider the following example program...

```
#include <stdio.h>

void main(){

    char ch;

    printf("\nEnter any character : ");

    ch = getch();

    printf("\nYou have entered : %c",ch);

}
```

Output:

```
Enter any character :
You have entered : A
```

gets() function

The gets() function is used to read a line of string and stores it into character array. The gets() function reads a line of string or sequence of characters till a newline symbol enters. Consider the following example program...

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

```
void main(){
  char name[30];
  printf("\nEnter your favourite website: ");
  gets(name);
  printf("%s",name);
}
```

Output:

```
Enter your favourite website: www.btechsmartclass.com
```

fscanf() function

The fscanf() function is used with the concept of files. The fscanf() function is used to read data values from a file. When you want to use fscanf() function the file must be opened in reading mode.

Output Functions in C:

C programming language provides built-in functions to perform output operation. The output opearations are used to display data on user screen (output screen) or printer or any file. C programming language provides the following built-in output functions...

printf()

putchar()

puts()

fprintf()

printf() function

The printf() function is used to print string or data values or combination of string and data values on the output screen (User screen). The printf() function is built-in function defined in a header file called "**stdio.h**". When we want to use printf() function in our program we need to include the respective header file (stdio.h) using **#include** statement. The printf() function has the following syntax...

Syntax:

```
printf("message to be display!!!");
```

Example Program

```
#include <stdio.h>

void main(){

   printf("Hello! Welcome to btechsmartclass!!!");

}
```

Output:

```
Hello! Welcome to btechsmartclass!!!
```

In the above example program, we used the printf() function to print a string on to the output screen.

The printf() function is also used to display data values. When we want to display data values we use **format string** of the data value to be display.

Syntax:

```
printf("format string",variableName);
```

Example Program

```
#include <stdio.h>

void main(){

   int i = 10;

   float x = 5.5;

   printf("%d %f",i, x);   }
```

Output:

```
10 5.5
```

In the above example program, we used the printf() function to print data values of variables i and x on to the output screen. Here i is a integer variable so we have used format string %d and x

is a float variable so we have used format string %f.

The printf() function can also used to display string along with data values.

Syntax:

printf("String format string",variableName);

Example Program

```
#include <stdio.h>

void main(){

  int i = 10;

  float x = 5.5;

  printf("Integer value = %d, float value = %f",i, x);


}
```

Output:

Integer value = 10, float value = 5.5

In the above program we are displaying string along with data values.

Every function in C programming language must have a return value. The printf() function also have integer as return value. The printf() function returns an integer value equalent to the total number of characters it has printed.

Example Program

```
#include <stdio.h>

void main(){

  int i;

  i = printf("btechsmartclass");

  printf(" is %d number of characters.",i);   }
```

Output:

btechsmartclass is 15 number of characters.

In the above program, first printf() function printing "btechsmartclass" which is of 15 characters. So it returns integer value 15 to variable "i". The value of "i" is printed in the second printf() function.

Formatted printf() function

Generally, when we write multiple printf() statements the result is displayed in single line because the printf() function displays the output in a single line. Consider the following example program...

```
#include <stdio.h>

void main(){

   printf("Welcome to ");

   printf("btechsmartclass ");

   printf("the perfect website for learning");

}
```

Output:

Welcome to btechsmartclass the perfect website for learning

In the above program, there are 3 printf() statements written in different lines but the output is displayed in single line only.

To display the output in different lines or as we wish, we use some special characters called escape sequences. Escape sequences are special characters with special functionality used in printf() function to format the output according to the user requirement. In C programming language, we have the following escape sequences...

Escape Sequence	Meaning
\n	New line
\t	Horizontal Tab
\v	Vertical Tab
\a	Beep sound
\b	Backspace
\\	Backward slash
\?	Question mark
\'	Single quotation mark
\"	Double quotation mark

Consider the following example program...

```
#include <stdio.h>
void main(){
  printf("Welcome to\n");
  printf("btechsmartclass\n");
  printf("the perfect website for learning");
}
```

Output:

```
Welcome to
btechsmartclass
the perfect website for learning
```

putchar() function

The putchar() function is used to display single character on the output screen. The putchar() functions prints the character which is passed as parameter to it and returns the same character as return value. This function is used to print only single charater. To print multiple characters we need to write multiple times or use a looping statement. Consiider the following example program...

```
#include <stdio.h>

void main(){

   char ch = 'A';

   putchar(ch);

}
```

Output:

```
A
```

puts() function

The puts() function is used to display string on the output screen. The puts() functions prints a string or sequence of characters till the newline. Consiider the following example program...

```
#include <stdio.h>

void main(){
   char name[30];
   printf("\nEnter your favourite website: ");
   gets(name);
   puts(name);
}
```

Output:

```
Enter your favourite website: www.btechsmartclass.com
www.btechsmartclass.com
```

fprintf() function

The fprintf() function is used with the concept of files. The fprintf() function is used to print a line into the file. When you want to use fprintf() function the file must be opened in writting mode.

Faculty Name: N.Thulasi Chitra
Subject :PPS
Topic: Operators

Unit No :2
Lecture No :L20
Book Reference:T1

Operators:

An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in programs to manipulate data and variables.

C operators can be classified into a number of categories, they are
1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Assignment Operators
5. Increment and decrement operators
6. Conditional operators
7. Bitwise Operators
8. Special operators

1. Arithmetic Operators:

The arithmetic operators are the symbols that are used to perform basic mathematical operations like addition, subtraction, multiplication, division and percentage modulo. The following table provides information about arithmetic operators.

Operator	Meaning	Example
+	Addition	10 + 5 = 15
-	Subtraction	10 - 5 = 5
*	Multiplication	5 * 10 = 50
/	Division	10 / 5 = 2
%	Remainder of Division	5 % 2 = 1

The addition operator can be used with numerical data types and character data type. When it is used with numerical values, it performs mathematical addition and when it is used with character data type values, it performs concatination (appending).

The remainder of division operator is used with integer data type only.

2.Relational Operators :

The relational operators are the symbols that are used to compare two values. That means, the relational operators are used to check the relationship between two values. Every relational operator has two results TRUE or FALSE. In simple words, the relational operators are used to define conditions in a program. The following table provides information about relational operators.

Operator	Meaning	Example
<	Returns TRUE if first value is smaller than second value otherwise returns FALSE	10 < 5 is FALSE
>	Returns TRUE if first value is larger than second value otherwise returns FALSE	10 > 5 is TRUE
<=	Returns TRUE if first value is smaller than or equal to second value otherwise returns FALSE	10 <= 5 is FALSE
>=	Returns TRUE if first value is larger than or equal to second value otherwise returns FALSE	10 >= 5 is TRUE
==	Returns TRUE if both values are equal otherwise returns FALSE	10 == 5 is FALSE
!=	Returns TRUE if both values are not equal otherwise returns FALSE	10 != 5 is TRUE

3. Logical Operators:

The logical operators are the symbols that are used to combine multiple conditions into one condition. The following table provides information about logical operators.

Operator	Meaning	Example
&&	**Logical AND** - Returns TRUE if all conditions are TRUE otherwise returns FALSE	10 < 5 && 12 > 10 is FALSE
||	**Logical OR** - Returns FALSE if all conditions are FALSE otherwise returns TRUE	10 < 5 || 12 > 10 is TRUE
!	**Logical NOT** - Returns TRUE if condition is FLASE and returns FALSE if it is TRUE	!(10 < 5 && 12 > 10) is TRUE

Logical AND - Returns TRUE only if all conditions are TRUE, if any of the conditions is FALSE then complete condition becomes FALSE.
Logical OR - Returns FALSE only if all conditions are FALSE, if any of the conditions is TRUE then complete condition becomes TRUE.

4. Assignment Operators :

The assignment operators are used to assign right hand side value (Rvalue) to the left hand side variable (Lvalue). The assignment operator is used in different variants along with arithmetic operators. The following table describes all the assignment operators in C programming language.

Operator	Meaning	Example
=	Assign the right hand side value to left hand side variable	A = 15
+=	Add both left and right hand side values and store the result into left hand side variable	A += 10 ⇒ A=A+10
-=	Subtract right hand side value from left hand side variable value and store the result into left hand side variable	A -= B ⇒ A=A-B
*=	Multiply right hand side value with left hand side variable value and store the result into left hand side variable	A *= B ⇒ A=A*B
/=	Divide left hand side variable value with right hand side variable value and store the result into left hand side variable	A /= B ⇒ A=A/B
%=	Divide left hand side variable value with right hand side variable value and store the remainder into left hand side variable	A %= B ⇒ A=A%B

5.Increment & Decrement Operators:

The increment and decrement operators are called as unary operators because, both needs only one operand. The increment operators adds one to the existing value of the operand and the decrement operator subtracts one from the existing value of the operand. The following table provides information about increment and decrement operators.

Operator	Meaning	Example
++	**Increment** - Adds one to existing value	int a = 5; a++; ⇒ a = 6
--	**Decrement** - Subtracts one from existing value	int a = 5; a--; ⇒ a = 4

The increment and decrement operators are used infront of the operand (++a) or after the operand (a++). If it is used infront of the operand, we call it as pre-increment or pre-decrement and if it is used after the operand, we call it as post-increment or post-decrement.

Pre-Increment or Pre-Decrement

In case of pre-increment, the value of the variable is increased by one before the expression evaluation. In case of pre-decrement, the value of the variable is decreased by one before the expression evaluation. That means, when we use pre increment or pre decrement, first

the value of the variable is incremented or decremented by one, then modified value is used in the expression evaluation.

Example:

```
#include <stdio.h>
void main()
{
    int i = 5,j;
    j = ++i; // Pre-Increment
    printf("i = %d, j = %d",i,j);
}
```

Output:

```
i = 6, j = 6
```

Post-Increment or Post-Decrement

In case of post-increment, the value of the variable is increased by one after the expression evaluation. In case of post-decrement, the value of the variable is decreased by one after the expression evaluation. That means, when we use post-increment or post-decrement, first the expression is evaluated with existing value, then the value of the variable is incremented or decremented by one.

Example

```
#include <stdio.h>
void main()
{
   int i = 5,j;
    j = i++;  // Post-Increment
    printf("i = %d, j = %d",i,j);
}
```

Output:

```
i = 6, j = 5
```

***6.*Conditional Operator (?:)**

The conditional operator is also called as ternary operator because it requires three operands. This operator is used for decision making. In this operator, first we verify a condition, then we perform one operation out of the two operations based on the condition result. If the condition is TRUE the first option is performed, if the condition is FALSE the second option is performed. The conditional operator is used with the following syntax...

Condition ? TRUE Part : FALSE Part ;
Example:
A = (10<15) ? 100 : 200 ; ⇒ A value is 100

7.Bitwise Operators:

The bitwise operators are used to perform bit level operations in c programming language. When we use the bitwise operators, the operations are performed based on the binary values. The following table describes all the bitwise operators in C programming language.

Let us consider two variables A and B as A = 25 (11001) and B = 20 (10100)

Operator	Meaning	Example
&	the result of **Bitwise AND** is 1 if all the bits are 1 otherwise it is 0	A & B ⇒ 16 (10000)
|	the result of **Bitwise OR** is 0 if all the bits are 0 otherwise it is 1	A | B ⇒ 29 (11101)
^	the result of **Bitwise XOR** is 0 if all the bits are same otherwise it is 1	A ^ B ⇒ 13 (01101)
~	the result of **Bitwise once complement** is nagation of the bit (Flipping)	~A ⇒ 6 (00110)
<<	the **Bitwise left shift** operator shifts all the bits to the left by specified number of positions	A<<2 ⇒ 100 (1100100)
>>	the **Bitwise right shift** operator shifts all the bits to the right by specified number of positions	A>>2 ⇒ 6 (00110)

8.Special Operators (sizeof, pointer, comma, dot etc.)

The following are the special operators in c programming language.

sizeof operator

This operator is used to find the size of the memory (in bytes) allocated for a variable. This operator is used with the following syntax...

sizeof(variableName);

Example

sizeof(A); ⇒ result is 2 if A is an integer

Pointer operator (*)

This operator is used to define pointer variables in c programming language.

Comma operator (,)

This operator is used to separate variables while they are declaring, separate the expressions in function calls etc..

Dot operator (.)

This operator is used to access members of structure or union.

Faculty Name: N.Thulasi Chitra
Subject :PPS
Topic: precedence and associativity

Unit No :2
Lecture No :L21
Book Reference:T1

Operator Precedence and Associativity

Explain Operator Precedence

Operator precedence is used to determine the order of operators evaluated in an expression. In c programming language every operator has precedence (priority). When there is more than one operator in an expression the operator with higher precedence is evaluated first and the operator with least precedence is evaluated last.

Discuss about Operator Associativity

Operator associativity is used to determine the order of operators with equal precedence evaluated in an expression. In c programming language, when an expression contains multiple operators with equal precedence, we use associativity to determine the order of evaluation of those operators.
In c programming language the operator precedence and associativity is as shown in the following table...

Precedence	Operator	Operator Meaning	Associativity
1	() [] -> .	function call array reference structure member access structure member access	Left to Right
2	! ~ + - ++ -- & * sizeof	negation 1's complement Unary plus Unary minus increment operator decrement operator address of operator pointer returns size of a variable	Right to Left

	(type)	type conversion	
3	* / %	multiplication division remainder	Left to Right
4	+ -	addition subtraction	Left to Right
5	<< >>	left shift right shift	Left to Right
6	< <= > >=	less than less than or equal to greater than greater than or equal to	Left to Right
7	== !=	equal to not equal to	Left to Right
8	&	bitwise AND	Left to Right
9	^	bitwise EXCLUSIVE OR	Left to Right
10	|	bitwise OR	Left to Right
11	&&	logical AND	Left to Right
12	||	logical OR	Left to Right
13	?:	conditional operator	Left to Right
14	= *= /= %= += -= &= ^= |= <<= >>=	assignment assign multiplication assign division assign remainder assign additon assign subtraction assign bitwise AND assign bitwise XOR assign bitwise OR assign left shift assign right shift	Right to Left
15	,	separator	Left to Right

In the above table, the operator precedence decrease from top to bottom and increase from bottom to top.

Faculty Name: N.Thulasi Chitra
Subject :PPS
Topic: evaluation of expressions

Unit No :2
Lecture No :L22
Book Reference:T1

Expressions:

Define an Expression

In any programming language, if we want to perform any calculation or to frame any condition etc., we use a set of symbols to perform the task. These set of symbols makes an expression.

In C programming language, an expression is defined as follows...

An expression is a collection of operators and operands that represents a specific value.

In the above definition, operator is a symbol which performs tasks like arithmetic operations, logical operations and conditional operations etc.,
Operands are the values on which the operators perform the task. Here operand can be a direct value or variable or address of memory location.

Expression Types in C

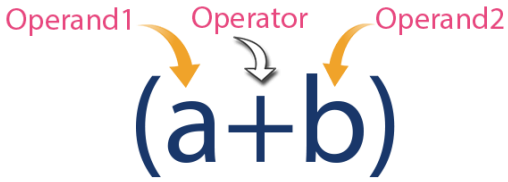
In C programming language, expressions are divided into THREE types. They are as follows...

1.Infix Expression
2.Postfix Expression
3.Prefix Expression

The above classification is based on the operator position in the expression.
1.Infix Expression

The expression in which operator is used between operands is called as infix expression.
The infix expression has the following general structure...

Operand1 Operator Operand2
Example



2.Postfix Expression

The expression in which operator is used after operands is called as postfix expression.
The postfix expression has the following general structure...

Operand1 Operand2 Operator

Example:

3.Prefix Expression

The expression in which operator is used before operands is called as prefix expression.
The prefix expression has the following general structure...

Operator Operand1 Operand2

Example:

Expression Evaluation in C

In C programming language, expression is evaluated based on the operator precedence and associativity. When there are multiple operators in an expression, they are evaluated according to their pr
ecedence and associativity. The operator with higher precedence is evaluated first and the operator with least precedence is evaluated last.

An expression is evaluated based on the precedence and associativity of the operators in that expression.

To understand expression evaluation in c, let us consider the following simple example expression...

10 + 4 * 3 / 2

In the above expression there are three operators +, * and /. Among these three operators, both multiplication and division have same higher precedence and addition has lower precedence. So, according to the operator precedence both multiplication and division are evaluated first and then addition is evaluated. As multiplication and division have same precedence they are evaluated based on the associativity. Here, the associativity of multiplication and division is left to right. So, multiplication is performed first, then division and finally

addition. So, the above expression is evaluated in the order of * / and +. It is evaluated as follows...

4 * 3 ====> 12
12 / 2 ===> 6
10 + 6 ===> 16

The expression is evaluated to **16**.

Faculty Name: N.Thulasi Chitra
Subject :PPS
Topic: Type Conversions In Expresssions

Unit No :2
Lecture No :L23
Book Reference:T1

In a programming language, the expression contains data values of same datatype or different datatypes. When the expression contains similar datatype values then it is evaluated without any problem. But if the expression contains two or more different datatype values then they must be converted to single datatype of destination datatype. Here, destination is the location where the final result of that expression is stored. For example,the multiplication of an integer data value with float data value and storing the result into a float variable. In this case, the integer value must be converted to float value so that the final result is a float datatype value.

In c programming language, the data conversion is performed in two different methods as follows.
Type Conversion
Type Casting

Type Conversion

The type conversion is the process of converting a data value from one datatype to another datatype automatically by the compiler. Sometimes type conversion is also called as **implicit type conversion**. The implicit type conversion is automatically performed by the compiler.
For example, in c programming language, when we assign an integer value to a float variable the integer value automically gets converted to float value by adding decimal value 0. And when a float value is assigned to an integer variable the float value automatically gets converted to integer value by removing the decimal value. To understand more about type conversion observe the following...

```
int i = 10 ;
float x = 15.5 ;
char ch = 'A' ;
```

i = x ; =======> x value 15.5 is converted as 15 and assigned to variable i

x = i ; =======> Here i value 10 is converted as 10.000000 and assigned to variable x

i = ch ; =======> Here the ASCII value of A (65) is assigned to i

Example Program

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

```
void main()
{
   int i = 95 ;
   float x = 90.99 ;
   char ch = 'A' ;
    i = x ;
   printf("i value is %d\n",i);
   x = i ;
   printf("x value is %f\n",x);
   i = ch ;
   printf("i value is %d\n",i);
}
```

Output:

```
i value is 90
x value is 90.000000
i value is 65
```

In the above program, we assign **i = x**, i.e., float variable value is assigned to integer variable. Here, the compiler automatically converts the float value (90.99) into integer value (90) by removing the decimal part of the float value (90.99) and then it is assigned to variable **i**. Similarly when we assign **x = i**, the integer value (90) gets converted to float value (90.000000) by adding zero as decimal part.

Type Casting

Type casting is also called as **explicit type conversion**. Compiler converts data from one datatype to another datatype implicitly. When compiler converts implicitly, there may be a data loss.In such case, we convert the data from one datatype to another datatype using explicit type conversion. To perform this we use the **unary cast operator**. To convert data from one type to another type we specify the target datatype in paranthesis as a prefix to the data value that has to be converted. The general syntax of type casting is as follows...

(TargetDatatype) DataValue

Example

```
int totalMarks = 450, maxMarks = 600 ;
float average ;

average = (float) totalMarks / maxMarks * 100 ;
```

In the above example code, both totalMarks and maxMarks are integer data values. When we perform totalMarks / maxMarks the result is a float value, but the destination (average) datatype is float. So we use type casting to convert totalMarks and maxMarks into float datatype.

Example Program

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

```
void main(){
   int a, b, c ;
   float avg ;
   printf("Enter any three integer values : ") ;
   scanf("%d%d%d", &a, &b, &c) ;
   avg = (a + b + c) / 3 ;
   printf("avg before casting = %f\n",avg);
   avg = (float)(a + b + c) / 3 ;
   printf("avg after casting = %f\n",avg);
}
```

Output:

```
Enter any three integer values : 5 3 2
avg before casting = 3
avg after casting = 3.333333
```

MARRI LAXMAN REDDY GROUP OF INSTITUTIONS

MLR Institute of Technology

Laxma Reddy Avenue, Dundigal, Quthbullapur (M), Hyderabad – 500 043

Faculty Name: N.Thulasi Chitra
Subject :PPS
Topic: Control Structures: if

Unit No :2
Lecture No :L24
Book Reference:T1

Control structures: Decision statements; if and switch statement; Loop control statements: while, for and do while loops, jump statements, break, continue, goto statements.

Decision Making statements:

In c programming language, the program execution flow is, line by line from top to bottom. That means the c program is executed line by line from the main method. But this type of execution flow may not be suitable for all the program solutions. Sometimes, we make some decisions or we may skip the execution of one or more lines of code. Consider a situation, where we write a program to check whether a student has passed or failed in a particular subject. *Here*, we need to check whether the marks are greater than the pass marks or not. If marks are greater, then we take the decision that the student has passed otherwise failed. To solve such kind of problems in c we use the statements called decision making statements.

Decision making statements are the statements that are used to verify a given condition and decides whether a block of statements gets executed or not based on the condition result.

In c programming language, there are two decision making statements they are as follows...
1. if statement
2. switch statement

1. if statement in c

In c, if statement is used to make decisions based on a condition. The if statement verifies the given condition and decides whether a block of statements are executed or not based on the condition result. In c, if statement is classified into four types as follows...

1. Simple if statement
2. if - else statement
3. Nested if statement
4. if-else-if statement (if-else ladder)

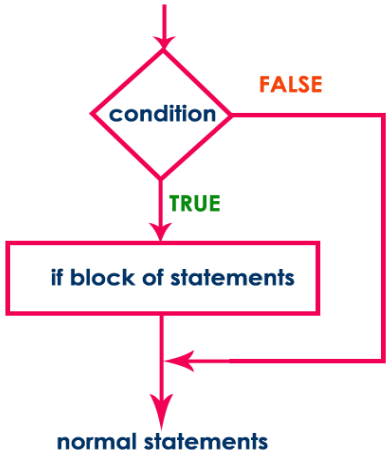
1. Simple if statement

Simple if statement is used to verify the given condition and executes the block of statements based on the condition result. The simple if statement evaluates specified condition. If it is TRUE, it executes the next statement or block of statements. If the condition is FALSE, it skips the execution of the next statement or block of statements. The general syntax and execution flow of the simple if statement is as follows...

Syntax

```
if ( condition )
{
   ....
   block of statements;
   ....
}
```

Execution flow diagram



Simple if statement is used when we have only one option that is executed or skipped based on a condition.

Example Program | Test whether given number is divisible by 5.

```
#include <stdio.h>
#include<conio.h>
void main(){
  int n ;
  clrscr() ;
  printf("Enter any integer number: ") ;
  scanf("%d", &n) ;
  if ( n%5 == 0 )
     printf("Given number is divisible by 5\n") ;
  printf("statement does not belong to if!!!") ;
}
```

Output 1:

Enter any integer number: 100
Given number is divisible by 5
statement does not belong to if!!!

Output 2:

Enter any integer number: 99
statement does not belong to if!!!

2. if - else statement

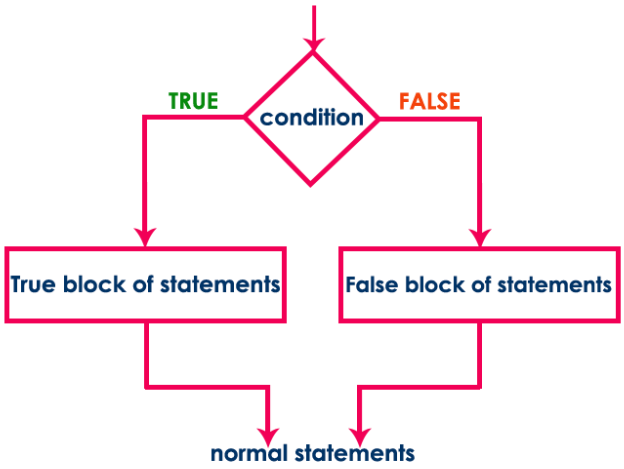
The if - else statement is used to verify the given condition and executes only one out of the two blocks of statements based on the condition result. The if-else statement evaluates the specified

condition. If it is TRUE, it executes a block of statements (True block). If the condition is FALSE, it executes another block of statements (False block). The general syntax and execution flow of the if-else statement is as follows...

Syntax

```
if ( condition )
{
   ....
   True block of statements;
   ....
}
else
{
   ....
   False block of statements;
   ....
}
```

Execution flow diagram



The if-else statement is used when we have two options and only one option has to be executed based on a condition result (TRUE or FALSE).

Example Program | Test whether given number is even or odd.

```
#include <stdio.h>
#include<conio.h>
void main(){
  int n ;
  clrscr() ;
  printf("Enter any integer number: ") ;
  scanf("%d", &n) ;
  if ( n%2 == 0 )
    printf("Given number is EVEN\n") ;
  else
    printf("Given number is ODD\n") ;
}
```

Output 1:

Enter any integer number: 100
Given number is EVEN

Output 2:

Enter any integer number: 99
Given number is ODD

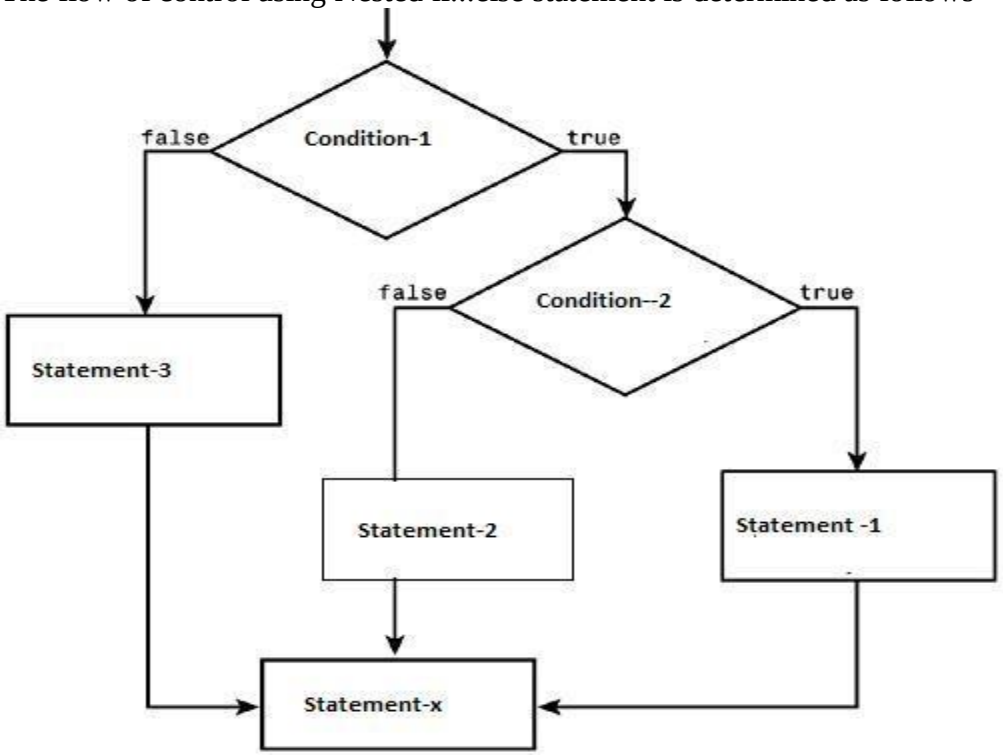
3. Nested if statement

Writing a if statement inside another if statement is called nested if statement. The general syntax of the nested if statement is as follows...

Syntax

```
if ( condition1 )
{
   if ( condition2 )
   {
      ....
      True block of statements 1;
   }
   ....
}
else
{
    False block of condition1;
}
```

The nested if statement can be defined using any combination of simple if & if-else statements. The flow of control using Nested if...else statement is determined as follows



Whenever nested if…else statement is encountered, first <condition1> is tested. It returns either true or false.

If condition1 (or outer condition) is false, then the control transfers to else-body (if exists) by skipping if-body.

If condition1 (or outer condition) is true, then condition2 (or inner condition) is tested. If the condition2 is true, if-body gets executed. Otherwise, the else-body that is inside of if statement gets executed.

Example Program | Test whether given number is even or odd if it is below 100.

```
#include <stdio.h>
#include<conio.h>
void main(){
  int n ;
  clrscr() ;
  printf("Enter any integer number: ") ;
  scanf("%d", &n) ;
  if ( n < 100 )
  {
    printf("Given number is below 100\n") ;
    if( n%2 == 0)
       printf("And it is EVEN") ;
    else
       printf("And it is ODD") ;
  }
  else
     printf("Given number is not below 100") ;
}
```

Output 1:

Enter any integer number: 55
Given number is below 100
And it is ODD

Output 2:

Enter any integer number: 999
Given number is not below 100

4. if - else - if statement (if-else ladder)

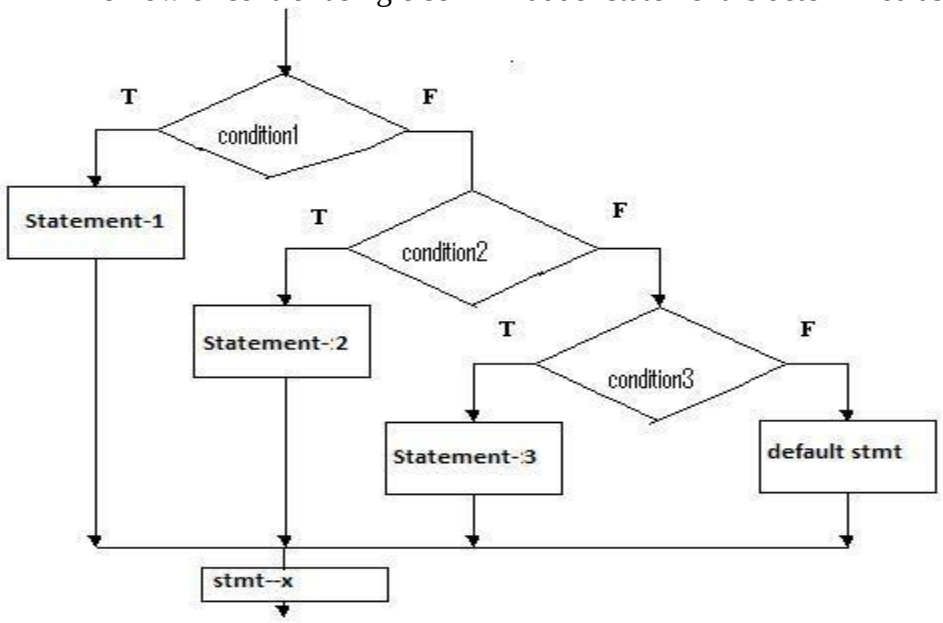
Writing a if statement inside else of a if statement is called if - else - if statement. The general syntax of the if-else-if statement is as follows...

Syntax

```
if ( condition1 )
{
    ....
    True block of statements1;
    ....
}
else if ( condition2 )
{
    False block of condition1;
    &
    True block of condition2
}
```

The if-else-if statement can be defined using any combination of simple if & if-else statements.

The flow of control using else--- if ladder statement is determined as follows:



Whenever else if ladder is encountered, condition1 is tested first. If it is true, the statement 1 gets executed. After then the control transfers to *stmt-x*.

If *condition1* is false, then *condition2* is tested. If *condition2* is false, the other conditions are tested. If all are false, the default stmt at the end gets executed. After then the control transfers to *stmt-x*.

If any one of all conditions is true, then the body associated with it gets executed. After then the control transfers to *stmt-x*.

Example Program | Find the largest of three numbers.

```
#include <stdio.h>
#include<conio.h>
void main(){
  int a, b, c ;
  clrscr() ;

  printf("Enter any three integer numbers: ") ;
  scanf("%d%d%d", &a, &b, &c) ;

  if( a>=b && a>=c)
     printf("%d is the largest number", a) ;

   else if (b>=a && b>=c)
     printf("%d is the largest number", b) ;

   else
     printf("%d is the largest number", c) ;
}
```

Output 1:

Enter any three integer numbers: 55 60 20
60 is the largest number

MOST IMPORTANT POINTS TO BE REMEMBERED

When we use conditional control statement like if statement, condition might be an expression evaluated to a numerical value, a variable or a direct numerical value. If the expression value or direct value is zero the conditon becomes FALSE otherwise becomes TRUE.

To understand more consider the following statements...
if(10) - is TRUE
if(x) - is FALSE if x value is zero otherwise TRUE
if(a+b) - is FALSE if a+b value is zero otherwise TRUE
if(a = 99) - is TRUE because a value is non-zero
if(10, 5, 0) - is FALSE because it considers last value
if(0) - is FALSE
if(a=10, b=15, c=0) - is FALSE because last value is zero

Faculty Name: N.Thulasi Chitra
Subject :PPS
Topic: Switch Statement

Unit No :2
Lecture No :L25
Book Reference:T1

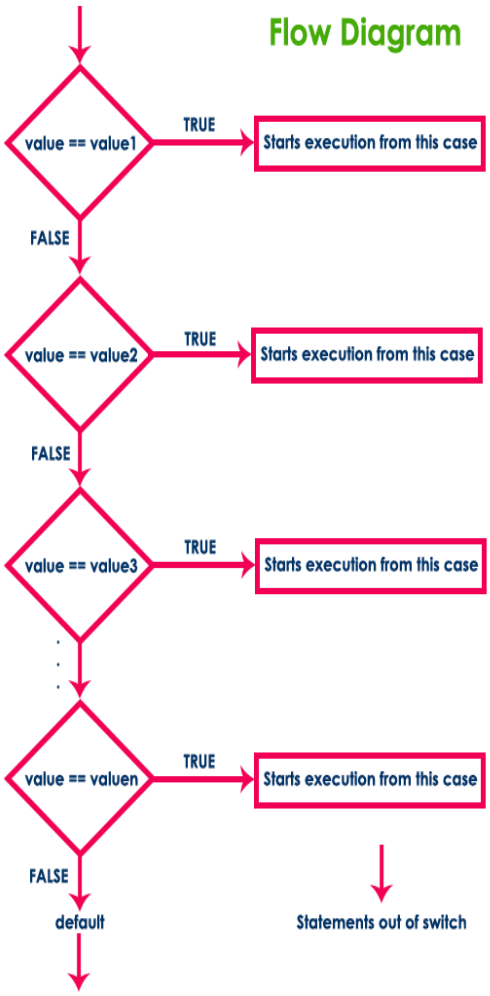
Switch statement:

Consider a situation in which we have more number of options out of which we need to select only one option that is to be executed. Such kind of problems can be solved using **nested if** statement. But as the number of options increases, the complexity of the program also gets increased. This type of problems can be solved very easily using **switch** statement. Using switch statement, one can select only one option from more number of options very easily. In switch statement, we provide a value that is to be compared with a value associated with each option. Whenever the given value matches with the value associated with an option, the execution starts from that option. In switch statement every option is defined as a **case**.

The switch statement has the following syntax and execution flow diagram...

Syntax

```
switch ( expression or value )
{
      case  value1:  set of statements;
                     ....
      case  value2:  set of statements;
                     ....
      case  value3:  set of statements;
                     ....
      case  value4:  set of statements;
                     ....
      case  value5:  set of statements;
                     ....
      .
      .
      default: set of statements;
}
```



The switch statement contains one or more number of cases and each case has a value associated with it. At first switch statement compares the first case value with the switchValue, if it gets matched the execution starts from the first case. If it doesn't match the switch statement compares the second case value with the switchValue and if it is matched the execution starts from the second case. This process continues until it finds a match. If no case value matches with the switchValue specified in the switch statement, then a special case called **default** is executed. When a case value matches with the switchValue, the execution starts from that particular case. This execution flow continues with next case statements also. To avoid this, we use "**break**" statement at the end of each case. That means the **break** statement is used to terminate the switch statement. However it is optional.

Example Program | Display pressed digit in words.

```
#include <stdio.h>
#include<conio.h>
void main(){
  int n ;
  clrscr() ;
```

```
  printf("Enter any digit: ") ;
  scanf("%d", &n) ;

  switch( n )
  {
            case 0: printf("ZERO") ;
            break ;
            case 1: printf("ONE") ;
            break ;
            case 2: printf("TWO") ;
            break ;
            case 3: printf("THREE") ;
            break ;
            case 4: printf("FOUR") ;
            break ;
            case 5: printf("FIVE") ;
            break ;
            case 6: printf("SIX") ;
            break ;
            case 7: printf("SEVEN") ;
            break ;
            case 8: printf("EIGHT") ;
            break ;
            case 9: printf("NINE") ;
            break ;
            default: printf("Not a Digit") ;
  }
  getch() ;
}
```

Output 1:

Enter any digit: 5
FIVE

Output 2:

Enter any digit: 15
Not a Digit

MOST IMPORTANT POINTS TO BE REMEMBERED

When we use switch statement, we must follow the following...
Both switch and case are keywords so they must be used only in lower case letters.
The data type of case value and the value specified in switch statement must be same.
switch and case values must be either integer or character but not float or string.
A switch statement can contain any number of cases.
The keyword case and its value must be separated with a white space.
The case values need not be defined in sequence, they can be in any order.
The default case is optional and it can be defined anywhere inside the switch statement.
The switch value might be a direct value, a variable or an expression.

Faculty Name: N.Thulasi Chitra
Subject :PPS
Topic: Loop Control Statements :While

Unit No :2
Lecture No :L26
Book Reference: T1

Loop control statements:

Consider a situation in which we execute a single statement or block of statements repeatedly for required number of times. Such kind of problems can be solved using **looping** statements in C. For example, assume a situation where we print a message for 100 times. If we want to perform that task without using looping statements, we have to either write 100 printf statements or we have to write the same message for 100 times in a single printf statement. Both are complex methods. The same task can be performed very easily using looping statements.

The looping statements are used to execute a single statement or block of statements repeatedly until the given condition is FALSE.

C language provides three looping statements...

1.while statement
2.do-while statement
3.for statement

1. While statement:

The while statement is used to execute a single statement or block of statements repeatedly as long as the given condition is TRUE. The while statement is also known as **Entry control looping statement**. The while statement has the following syntax...

Syntax:

```
while( condition )
{
    ...
    block of statements;
    ...
}
```

The while statement has the following execution flow

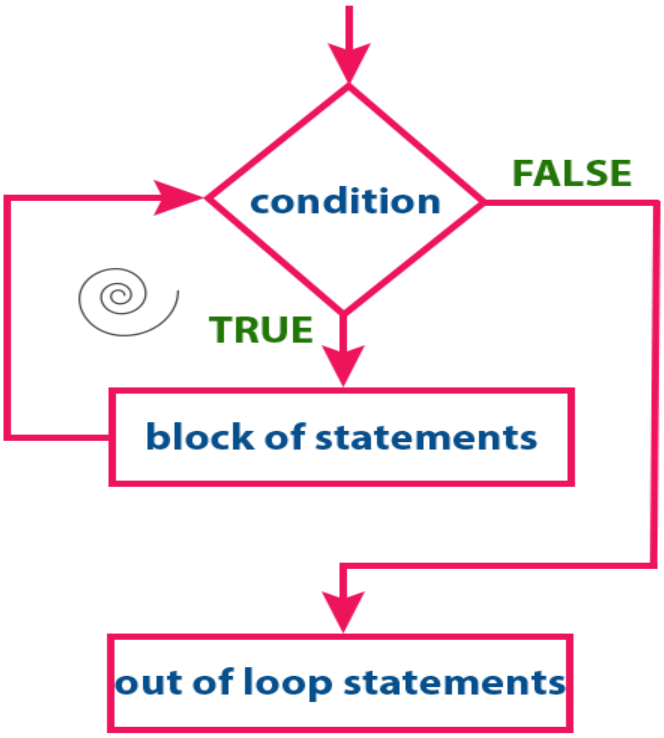


diagram...

At first, the given condition is evaluated. If the condition is TRUE, the single statement or block of statements gets executed. Once the execution gets completed the condition is evaluated again. If it is TRUE, again the same statements gets executed. The same process is repeated until the condition is evaluated to FALSE. Whenever the condition is evaluated to FALSE, the execution control moves out of the while block.

Example Program | Program to display even numbers upto 10.

```
#include <stdio.h>
#include<conio.h>
void main(){
  int n = 0;
  clrscr() ;
  printf("Even numbers upto 10\n");

  while( n <= 10 )
  {
    if( n%2 == 0)
      printf("%d\t", n) ;
    n++ ;
  }

  getch() ;
}
```

Output 1:
Even numbers upto 10
0 2 4 6 8 10

MOST IMPORTANT POINTS TO BE REMEMBERED

When we use while statement, we must follow the following...
while is a keyword so it must be used only in lower case letters.
If the condition contains variable, it must be assigned a value before it is used.
The value of the variable used in condition must be modified according to the requirement inside the while block.
In while statement, the condition may be a direct integer value, a variable or a condition.
A while statement can be an empty statement.

Faculty Name: N.Thulasi Chitra
Subject :PPS
Topic: Loop Control Statements : Do-While ,for

Unit No :2
Lecture No :L27
Book Reference: T1

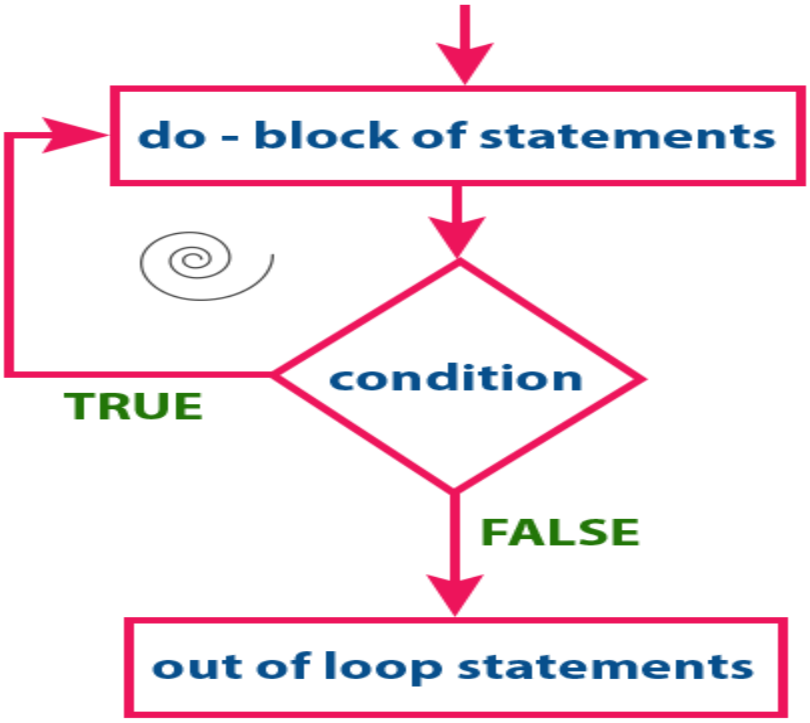
2. do-While statement:

The do-while statement is used to execute a single statement or block of statements repeatedly as long as given the condition is TRUE. The do-while statement is also known as **Exit control looping statement**. The do-while statement has the following syntax...

Syntax:

```
do
{
    ...
    block of statements;
    ...
} while( condition ) ;
```

The do-while statement has the following execution flow diagram.



At first, the single statement or block of statements which are defined in **do** block are executed. After execution of do block, the given condition gets evaluated. If the condition is evaluated to TRUE, the single statement or block of statements of do block are executed again. Once the execution gets completed again the condition is evaluated. If it is TRUE, again the same statements are executed. The same process is repeated until the condition is evaluated to FALSE. Whenever the condition is evaluated to FALSE, the execution control moves out of the while block.

Example Program | Program to display even numbers upto 10.

```
#include <stdio.h>
#include<conio.h>
void main(){
   int n = 0;
   clrscr() ;
   printf("Even numbers upto 10\n");

   do
   {
      if( n%2 == 0)
         printf("%d\t", n) ;
      n++ ;
   }while( n <= 10 ) ;

   getch() ;
}
```

Output 1:

Even numbers upto 10
0 2 4 6 8 10

MOST IMPORTANT POINTS TO BE REMEMBERED

When we use do-while statement, we must follow the following...
Both do and while are keywords so they must be used only in lower case letters.
If the condition contains variable, it must be assigned a value before it is used.
The value of the variable used in condition must be modified according to the requirement inside the do block.
In do-while statement the condition may be, a direct integer value, a variable or a condition.
A do-while statement can be an empty statement.
In do-while, the block of statements are executed at least once.

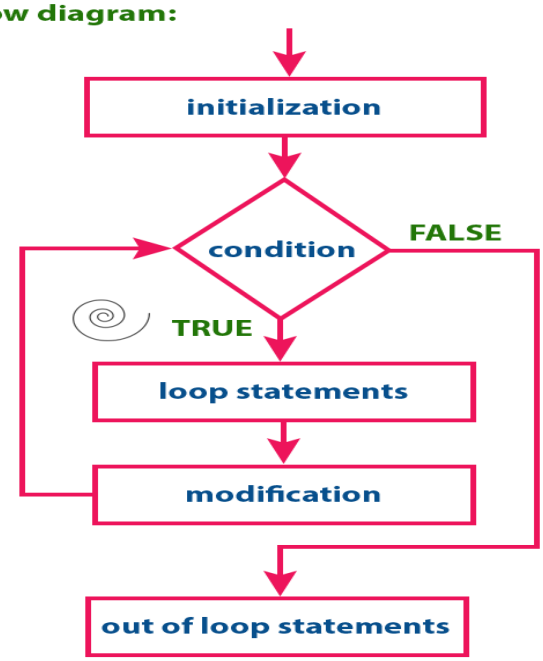
3.for statement:

The for statement is used to execute a single statement or a block of statements repeatedly as long as the given condition is TRUE. The for statement has the following syntax and execution flow diagram...

Syntax:

```
for( initialization ; condition ; modification )
{
    ...
    block of statements;
    ...
}
```

Execution flow diagram:



At first, the for statement executes initialization followed by condition evaluation. If the condition is evaluated to TRUE, the single statement or block of statements of for statement are executed. Once the execution gets completed, the modification statement is executed and again the condition is evaluated. If it is TRUE, again the same statements are executed. The same process is repeated until the condition is evaluated to FALSE. Whenever the condition is evaluated to FALSE, the execution control moves out of the for block.

Example Program | Program to display even numbers upto 10.

```
#include <stdio.h>
#include<conio.h>
void main(){
   int n ;
   clrscr() ;
   printf("Even numbers upto 10\n");

   for( n = 0 ; n <= 10 ; n++ )
   {
      if( n%2 == 0)
         printf("%d\t", n) ;
   }

   getch() ;
}
```

Output 1:

Even numbers upto 10
0 2 4 6 8 10

MOST IMPORTANT POINTS TO BE REMEMBERED

When we use for statement, we must follow the following.
for is a keyword so it must be used only in lower case letters.
Every for statement must be provided with initialization, condition and modification (They can be empty but must be separated with ";")
Ex: for (; ;) or for (; condition ; modification) or for (; condition ;)
In for statement, the condition may be a direct integer value, a variable or a condition.
The for statement can be an empty statement.

Faculty Name: N.Thulasi Chitra
Subject : PPS
Topic: Braeak, Jump , Continue , goto statement

Unit No :2
Lecture No :L28
Book Reference: T1

Break,continue and goto statements:

In c, there are control statements which does not need any condition to control the program execution flow. These control statements are called as **unconditional control statements**. C programming language provides the following unconditional control statements...

1.break
2.continue
3.goto

The above three statements does not need any condition to control the program execution flow.

1.break statement:

In C, the **break statement** is used to perform the following two things.

break statement is used to terminate switch case statement
break statement is also used to terminate looping statements like while, do-while and for.

When a **break** statement is encountered inside the switch case statement, the execution control moves out of the switch statement directly. For example consider the following program.

Example Program | Program to perform all arithmetic operations using switch statement.

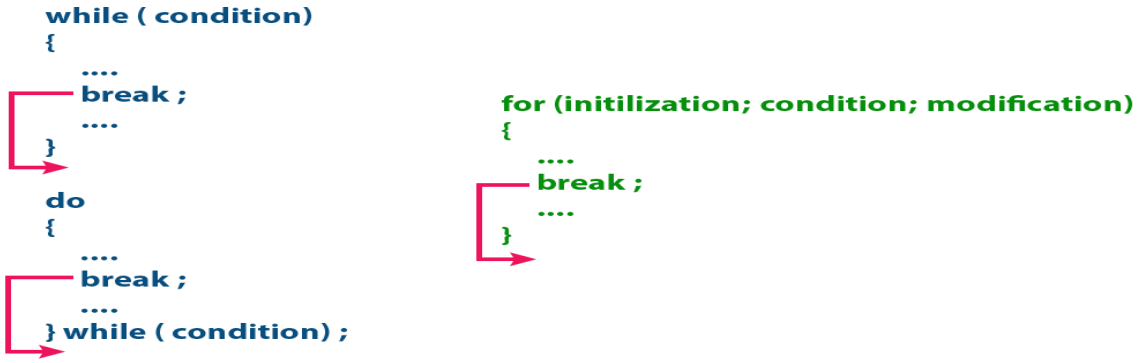
```
#include <stdio.h>
#include<conio.h>
void main(){
   int number1, number2, result ;
   char operator;
   clrscr() ;
   printf("Enter any two integer numbers: ") ;
   scanf("%d%d", &number1, &number2) ;
   printf("Please enter any arithmetic operator: ");
   operator = getchar();
   switch(operator)
   {
      case '+': result = number1 + number2 ;
                printf("Addition = %d", result) ;
                break;
      case '-': result = number1 - number2 ;
                printf("Subtraction = %d", result) ;
                break;
      case '*': result = number1 * number2 ;
```

```
            printf("Multiplication = %d", result) ;
            break;
      case '/': result = number1 / number2 ;
            printf("Division = %d", result) ;
            break;
      case '%': result = number1 % number2 ;
            printf("Remainder = %d", result) ;
            break;
      default: printf("\nWrong selection!!!") ;
   }
   getch() ;
}
```

Output

```
Enter any two integer numbers: 50 30
Please enter any arithmetic operator: *
Multiplication = 1500
```

When the **break** statement is encountered inside the looping statement, the execution control moves out of the looping statements. The **break** statement execution is as shown in the following figure.



For example, consider the following example program...

Example Program for *break statement*.

```
#include <stdio.h>
#include<conio.h>
void main(){
   char ch ;
   clrscr() ;
   do
   {
      printf("Enter Y / N : ") ;
      scanf("%c", &ch) ;
      if(ch == 'Y')
      {
```

```
        printf("Okay!!! Repeat again !!!\n") ;
      }
      else
      {
        printf("Okay !!! Breaking the loop !!!") ;
        break ;
      }
   } while( 1 ) ;
   getch() ;
}
```

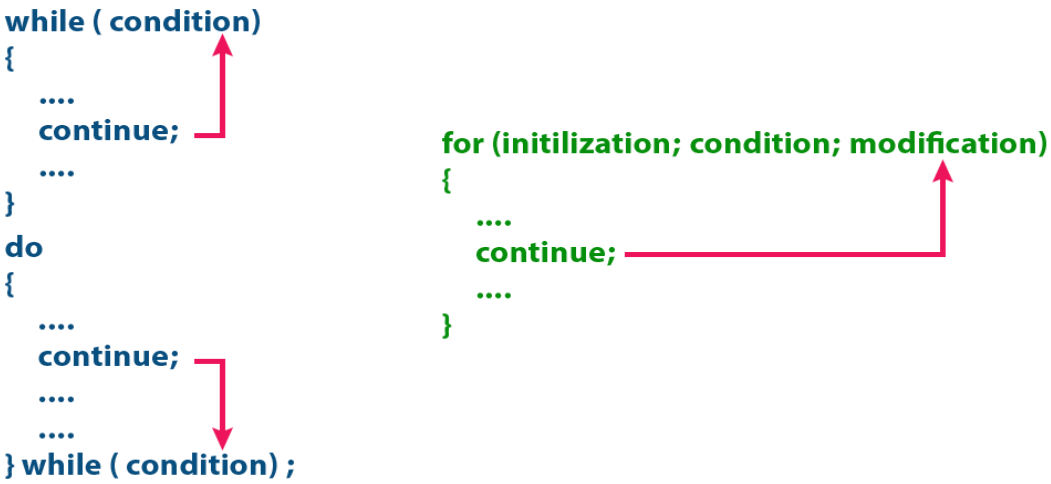
Output:

Enter Y / N : Y
Okay!!! Repeat again !!!
Enter Y / N : Y
Okay!!! Repeat again !!!
Enter Y / N : N
Okay !!! Breaking the loop !!!

2.Continue statement:

The **continue** statement is used to move the program execution control to the beginning of looping statement. When **continue** statement is encountered in a looping statement, the execution control skips the rest of the statements in the looping block and directly jumps to the beginning of the loop. The **continue** statement can be used with looping statements like while, do-while and for.

When we use **continue** statement with **while** and **do-while** statements the execution control directly jumps to the condition. When we use **continue** statement with **for** statement the execution control directly jumps to the modification portion (increment / decrement / any modification) of the for loop. The **continue** statement execution is as shown in the following figure.



Example Program | Program to illustrate continue statement.

```
#include <stdio.h>
#include<conio.h>
void main(){
   int number ;
   clrscr() ;
   while( 1 )
   {
       printf("Enter any integer number: ") ;
       scanf("%d", &number) ;
       if(number%2 == 0)
       {
           printf("Entered number is EVEN!!! Try another number!!!\n") ;
           continue ;
       }
       else
       {
           printf("You have entered ODD number!!! Bye!!!") ;
           exit(0) ;
       }
   }
   getch() ;
}
```

Output

Enter any integer numbers: 50
Entered number is EVEN!!! Try another number!!!
Enter any integer numbers: 100
Entered number is EVEN!!! Try another number!!!
Enter any integer numbers: 10
Entered number is EVEN!!! Try another number!!!
Enter any integer number: 15
You have entered ODD number!!! Bye!!!

3.goto statement:

The **goto** statement is used to jump from one line to another line in the program.
Using **goto** statement we can jump from top to bottom or bottom to top. To jump from one line to another line, the goto statement requires a **lable**. Lable is a name given to the instruction or line in the program. When we use **goto** statement in the program, the execution control directly jumps to the line with specified lable.

Example Program for *goto statement*.

```
#include <stdio.h>
#include<conio.h>
void main(){
   clrscr() ;
   printf("We are at first printf statement!!!\n") ;
   goto last ;
   printf("We are at second printf statement!!!\n") ;
   printf("We are at third printf statement!!!\n") ;
   last: printf("We are at last printf statement!!!\n") ;
   getch() ;
}
```

Output

We are at first printf statement!!!
We are at last printf statement!!!

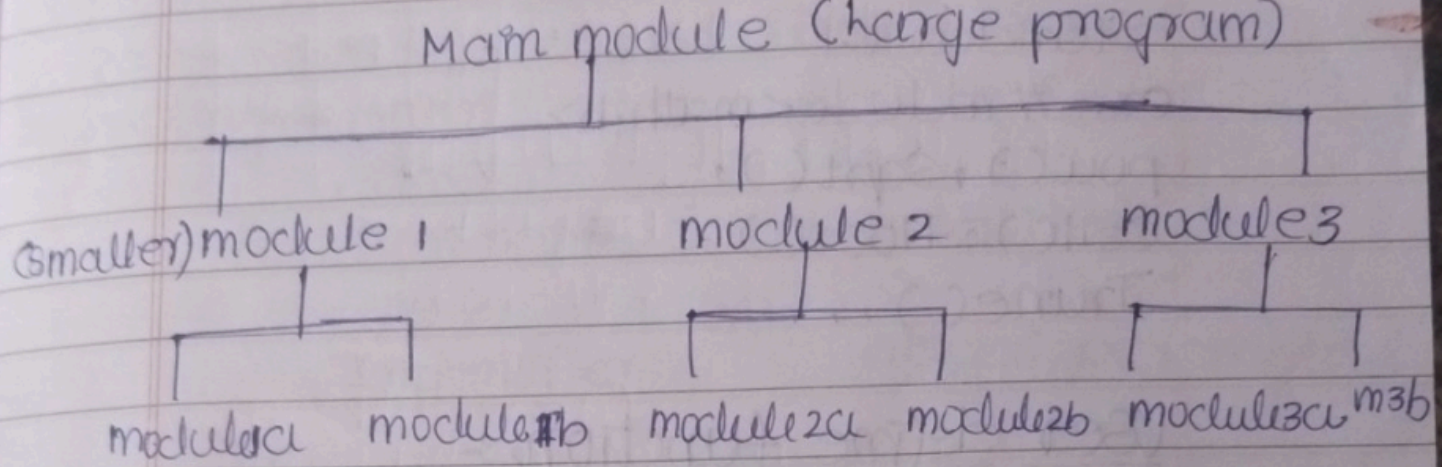
MOST IMPORTANT POINTS TO BE REMEMBERED

When we use break, continue and goto statements, we must follow the following...

- The **break** is a keyword so it must be used only in lower case letters.
- The **break** statement can not be used with **if** statement.
- The **break** statement can be used only in switch case and looping statements.
- The **break** statement can be used with **if** statement, only if that **if statement** is written inside the switch case or looping statements.
- The **continue** is a keyword so it must be used only in lower case letters.
- The **continue** statement is used only within **looping statements**.
- The **continue** statement can be used with **if** statement, only if that **if statement** is written inside the looping statements.
- The **goto** is a keyword so it must be used only in lower case letters.
- The **goto** statement must requires a **lable**.
- The **goto** statement can be used with any statement like if, switch, while, do-while and for etc,.

3. Functions

→ Function is a sub part of a program used to profrom sepecific task at is excuted individually.



Advantages of function :-

1. code reusability
2. once a function is created it can be be used for many times.
2. using functions we can impliment modular programming.
3. larger programs are divide into smaller programs.
4. Improve the readability of code.
5. debuging (error identification) of the code is easier.
6. using functions program implimentation become easy.

Types of functions :-

1. Built-in-Functions (or) predefined (or) Standard.
ex:- #include<math.h>
pow(), sqrt(),
ceil(), floor(),
True()....

2. User defined functions are one for which the compiler generates machine code at compile time.

User define function :-

→ User define functions are written by user
→ Every function has three parts
1. Function Declaration / Function prototype
2. Function call / calling Function
3. Function Defination / called function

1. function deblaration :-

→ This is declared before the main function

Syntax :-

returntype functionname(parameterlist);

- returntype → void, int, char, float
- functionname → Indentifier
- parameterlist → formal parameters

ex:- void add();
int add(int a, int b);
int add();

2. Function Call :-
→ This is declared in the main function
Syntax :-

function name (parameters list);
↓ ↓
Indentifier actual parameters

ex:-
add();
add(a,b);

3. Function definition :-
→ Function definition should be After the main function.
Syntax :-

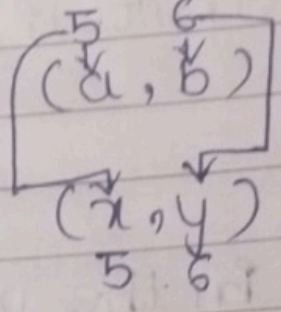
returntype functionname (parameterlist)
↓
formal Parameter
{
//actual logic
}

ex:- void add()
{
logic
}

ex:- int add()
{
//logic
return c;
}

→ No need of using same names

ex:-



→ That values copied into another

Actual parameter:-

→ The parameters specified in calling function are said to be actual parameter.

Formal parameter:-

→ The parameter specified in called function are said to be formal parameter

→ The value of actual parameter is always copied to formal parameter

→ Based on parameter and return type functions are classified into four types.

1. without parameters and without return value.
2. With parameters and without return value
3. Without parameters and with return value
4. With parameters and with return value.

Date
Page

Without parameter
- with return
- without return

With parameter
- with return
- without return

→ It terminate only in main function()
→ After function defmition it will go to function call.

1. Without parameters & without return value

```
#include<stdio.h>
void add(); // Function Declaration
int main()
{
   add(); // Function call
}
void add()  // Function definition
{
   int a, b, c;
   printf("Enter a, b values");
   Scanf("%d %d", &a, &b);
   c=a+b;
   printf("c=%d", c);
}
```

2. without paramaters & with return value

```
# include <stdio.h>
  int add();
int main()
{
  int res;
  res = add();
  printf("res = %d", res);
}
int add()
{
  int a, b, c;
  printf("enter a, b values");
  scanf("%d %d", &a, &b);
  c = a+b;
  return c;
}
```

3. With parameters & withaitreturn valu.

```
# include <stdio.h>
void add(int a, int b); // Function declarat
int main()
{
int a, b;
printf("Enter a, b values");
scanf("%d %d", &a, &b); // 10 20
```

```
add(a,b);  // Function call
   └→ actual parameters
}
void add(int a, int b)  // Function definition
          └→ Formal parameters
{
    int c;
    c = a+b;
    printf("c = %d\n", c);
}
```

4. with parameters and with return value

```
#include<stdio.h>
int add(int a, int b);
int main()
{
  int a,b, result;
  printf("enter a&b values");
  scanf("%d %d", &a, &b);
  res = add(a,b);
  printf("res = %d\n", res);
}
int add(int a, int b)
{
return(a+b);
}
```

Q. write a c program to find the factorial of a given number.

```
#include<stdio.h>
void fact();
int main()
{
    fact();
}
void fact()
{
    int n, i, fact=1;
    printf("enter n value");
    scanf("%d", &n);
    for(i=1; i<=n; i++)
    {
        fact = fact * i;
        printf("%d", fact);
    }
```

Q.

```
#include <stdio.h>
int fact();
int main()
{
    int fact;
    fact=fact();
    printf("fact = %d", fact);
}
```

```
int fact()
{
    int n, i, fact=1;
    printf("enter n value");
    scanf("%d", &n);
    for(i=1; i<n; i++)
    {
        fact=fact*i;
        return fact;
}
```

9.

```
#include<stdio.h>
void fact(int n);
int main()
{
    int n,
    printf("enter n value");
    scanf("%d", &n)
    fact(n);
}
void fact(int n)
{
    int n, fact=1, i;
    for(i=1; i<n; i++)
    {
        fact=fact*i;
        printf("fact=%d\n", fact);
}
```

Q.

```
#include<stdio.h>
int fact(int n);
int main()
{
    int n, i, fact=1;
    printf("enter n value");
    scanf("%d",&n);
    fact=fact(n);
    printf("fact=%d", fact);
}
int fac(int n)
{
    int i, fact=1;
    for(i=1;i<n;i++)
    fact=fact*i;
    return fact;
}
```

Q. write a c program to print Fibonosics series upto n.

```
#include<stdio.h>
void fib(int,n);
int main()
{
    int n;
    printf("enter n value");
```

```
scanf("%d",&n);
fib(n);
}
void fib(int n)
{
 int a=0,b=1,c;
 printf("%d %d",a,b)
    c=a+b;
 while(c<=n)
 {
  printf("%d",c);
   a=b;
   b=c;
   c=a+b;
 }
}
```

III-Unit

1. defination function and Advantages
2. Built in function with examples.
3. types of user defme function based on parameters and return type.
4. factorial and fibanosis programs.
5. C program diagram.
6. operators with examples.
7. primitive datatype, bytes, range in terms of 2 powers.
8. Array - defnations, types defnations, Syntax, memory list.

Q. Write a C program to print Transpose of a given matrice.

```
#include<stdio.h>
int main()
{
   int a[20][20], n, i, j;
printf("enter size of array");
Scanf ("%d", &n);
printf("enter array elements");
for(i=0; i<n; i++)
 {
   for(j=0; j<n; j++)
    {
      Scanf("%d", &a[i][j]);
    }
}
for(i=0; i<n; i++)
  {
    for(j=0; j<n; j++)
     {
       printf("%d", a[j][i]);
     }
  }
return 0;
}
```

eg:- $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$

output:- $A = A^T$ $\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$

Q. write a program to print opposite diagonal elements in a given matrice.

```
#include<stdio.h>
int main()
{
 int a[20][20], n, i, j;
printf("enter size of array");
Scanf("%d", &n);
printf("enter array elements");
for(i=0; i<n; i++)
{
 for(j=0; j<n; j++)
 {
 Scanf("%d", &a[i][j]);
 }
}
for(i=0; i<n; i++)
{
 printf("%d", a[i][n-i-1]);
}
return 0;
}
```

~~for principle diagonal~~ elements

for principle diagonal elements:-

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

Q. Sum of individual
write a program to find the sum of individual digit of a given number using non-Recursion function.

```
#include<stdio.h>
void sum( );
int main( )
{
      sum( );
}
void sum( )
{
   int n, sum=0, rem;
   printf("enter n value");
   scanf("%d", &n);
   while(n!=0)
   {Rem = n%10;
    sum1 = sum1 + Rem;
    n = n/10;
   }
   printf("sum1 = %d", sum1);
}
```

```
#include<stdio.h>
int sum( );
int main( )
{
      int [illegible];
      [illegible] = sum( );
```

```
printf("[illegible] = %d", [illegible]);
}
int sum()
{
    int n, sum1=0, Rem;
    printf("Enter n value");
    Scanf("%d",&n);
    [illegible]
    while(n!=0)
    { Rem=n%10;
      sum1=sum1+Rem;
      n=n/10;
    }
    [illegible]
  return sum1;
}
```

```
#include<stdio.h>
void sum(int n);
int main()
{
    int n
  printf("enter n value");
  Scanf("%d",&n);
  sum(n);
}
void sum(int n)
{
    int rem,sum1=0;
    while(n!=0)
```

```
    {   Rem = n%10;
        Sum = Sum + Rem;
        n = n/10
    }
    printf("Sum1 = %d", Sum1);
}
```

```
#include<stdio.h>
int sum(int n);
int main()
{
    int n, res;
    printf("enter n value:")
    scanf("%d", &n);
    res = sum(n);
    printf("Sum = %d\n", res);
}
int sum(int n)
{
    int rem, sum1=0;
    while(n!=0)
    {
        rem = n%10;
        sum1 = sum1 + rem;
        n = n/10;
    }
    return sum1;
}
```

Q. Write a c program to find the Reverse of a given digit.

```
#include<stdio.h>
void sum();
int main()
{
   sum();
}
void sum()
{
   int n, sum1=0, rem;
printf("enter n value");
scanf("%d",&n);
while(n!=0)
{  Rem = n%10;
   sum1 = sum1*10 + Rem;
    n = n/10;
}
   printf("sum1 = %d", sum1);
}
```

```
#include<stdio.h>
int sum();
int main()
{
   int res;
    res = sum();
    printf("res = %d", res);
}
```

Q. Write a c program to find sum of n natural numbers using non-recursive function.

```
#include<stdio.h>
int sum_natural(int n);
int main()
{
    int n, res;
printf("enter n value");
scanf("%d", &n);
 res = sum_natural(n);
printf("sum of n natural numbers=
                %d\n", res);
}
int sum_natural(int n)
{
    int i, sum=0;
    for(i=1; i<=n; i++)
    {
        sum = sum + i;
    }
  return sum;
}
```

i=1 i<=5 sum=0
i=1 1<=5
sum= 0+1=1
i=2 2<=5
sum = 1+2=3
i=3 3<=5
sum = 3+3=6

Date
Page

(8.) write a c program to check wheather the given number is Armstrong or not
153, 370, 371 → Armstrong no.

```
#include<stdio.h>
int armstrong(int n);
int main()
{
    int n, res;
printf("%d ",&n);
scanf("%d",&n);
 res=armstrong(n);
 if(res==n)
 printf("Amstrong number");
 else
 printf("Not Armstrong");
}
int armstrong(int n)
{
  int rem, S=0;
  while(n!=0)
  {
    rem = n%10;
    sum = sum+(rem*rem*rem);
    n = n/10;
  }
  return S;
}
```

(9.) write a c program to check whether the given number is palindrome or not

```
#include<stdio.h>
int palindrome( )
int main( )
{
    int n, res;
printf("enter n value");
Scanf("%d",&n);
res = palindrome(n);
If (res==n)
printf(" palindrome number");
else
printf (" Not palindrome number");
}
int palindrome(int n)
{
   int rem, sum=0;
   while (n!=0)
   {rem = n%10;
    sum= sum*10+rem;
     n=n/10;
   }
 return sum;
}
```

Q. write a c program to check toheather the given number is prime or not.

```
# include <stdio.h>
int prime()
int main()
{
  int n, res;
printf("enter n value");
Scanf("%d", &n);
res = prime(n);
if(res == 2)
printf("prime no");
else
prnf("not prime no");
}
int prime(int n)
{
 int c=0, i;
 for(i=0; i<=n; i++)
 {
   if (n%i==0)
      c++;
 }
 return c;
}
```

Q. write a c program to display factors of a given numbers.

```
#include<stdio.h>
void factors(int n)
int main()
{
    int n,i;
printf("enter n value");
scanf("%d",&n);
factors(n);
}
void factors(int n)
{
    int i;
   for(i=1;i<=n;i++)
    {
      if(n%i==0)
       printf("%d",i)
    }
}
```

Q. write a c program to find power of a given number.

```
#include<stdio.h>
int power(int b, int e);
int main()
{
    int b,e,res;
printf("enter base, exponent:\n");
```

```
Scanf("%d%d",&b,&e);
  res=power(b,e);
printf("power of %d^%d=%d",b,e,res);
}
int power(int b,int e)
{
 int i,x=1;
 for(i=1;i<=e;i++)
 {
   x=x*b;
 }
return x;
}
```

(9.) write a c program to find gcd of three numbers.

```
#include<stdio.h>
int gcd(int a, int b);
int main()
{
    int a,b,c,res;
 printf("enter a,b,c values");
 Scanf("%d%d%d",&a,&b,&c);
res=gcd(gcd(a,b),c);
print("gcd of %d and %d=%d\n",a,b,res);
}
int gcd(int a, int b)
{
   int i,g;
```

```
for(i=1;i<=a&& i<=b; i++)
{
   if (a%i==0 && b%i==0)
   {
      g=i;
   }
}
return g;
}
```

(9.) Write a cprogram to find LCM of three number.

```
#include<stdio.h>
int gcd(int a, int b);
int main()
{
   int a,b,c,res;
   printf("enter a,b,c values");
   scanf("%d %d %d",&a,&b,&c);
   res = gcd(gcd(a,b),c) ;
printf("LCM of 3 numbers = %d\n"
                  res);
}
int gcd(int a, int b)
{
   int i,g,l;
   for(i=1;i<=a&& i<=b;i++)
   {
   if (a%i==0 && b%i==0)
```

```
{
    q = r;
}
l = (a*b)/q;
return l;
}
}
```

Q. write a c program to find sum of all even or odd numbers in given range using non recursion function. from 1 to n.

```
#include<stdio.h>
int sum_odd_even(int s, int e);
int main()
{
    int e, res1;
printf("enter end value");
scanf("%d", &e);
printf("sum of even and odd number
    with in the range");
res1 = odd_sum(1,e);
printf("odd sum = %d\n", res1);
res2 = even_sum(2,e);
printf("even sum = %d\n", res2);
}
int sum_odd_even(int s, int e)
{
    int i, sum = 0;
```

```
for(i=s; i<=e ; i=i+2)
{
    sum = sum+i;
}
return sum;
}
```

1, 2, 3, 4, 5

odd = 1+3+5 = 9

even = 2+4 = 6

Date
Page

Built-in functions :-

Library / prefined / standard.

→ These are the functions who's defination is already known to the complier.
→ The user should known the function declaration or usage of function.
→ All this standard function declarations are present in header files.
ex:- printf(), scanf(), fprintf(), fscanf(). → stdio.h

pow(), sqrt(), ceil(), round()
→ math.h

malloc(), calloc(), realloc(), exit()
→ stdlib.h
standard library header file.

is lower(), isupper(), isdigit() → ctype.h

Under 'Math.h' Functions :-

1. Absolute function :- It returns the absault value of a given integer or real number.
Syntax :- int abs(float);
float abs(float);

Eg:- abs(-3) = 3
abs(-3.5) = 3.5
abs(3) = 3

2. ceil():- It returns the highest integer value which is near to the given number.
Syntax:- int ceil(float);
Ex:- ceil(1.2) = 2
ceil(-3.4) = -3

3. floor():- It returns the least integer value which is near to the given number.
Syntax:- int floor(float);
Ex:- floor(1.2) = 1
floor(-3.4) = -4

4. Truncate function : trunc()
It returns the integer value by truncating / removing the decimal part.
Syntax:- int trunc(float);
Ex:- trunc(3.4) = 3
trunc(-1.2) = -1

5. round():- It returns the nearest integer value depends on the decimal part.

Syntax: int round(float);
ex: round(2.1)=2
round(2.5)=3
round(2.9)=3

power Function: The power(x,y) function returns the value of the x raised to the power of y i.e., x^y
ex: pow(2,3)=8 $2^3=8$

Sqrt Function: sqrt() returns non-negative square root of a number. error occurs if the number is negative.
ex: sqrt(25)=5
sqrt(256)=16.

parameter passing in function:

(callby value and call by reference)

There are two ways of passing parameters from calling function to called function.
1. call by value (direct value)
2. call by Reference (Address of valu

1. call by value :-

→ In call by value method, the value of the actual parameters is copied into the formal parameters
→ A copy of the value is passed into the function.
→ changes made inside the function is limited to the function only. The values of the actual parameters do not change by changing the formal parameters.
→ Actual and formal are created at the different memory location.

```
#include<stdio.h>
void swap(int a, int b);
int main()
{
  int a=10;
  int b=20;
printf("Before swapping the values
      a= %d, b=%d\n", a, b);
swap(a, b);
printf("After swapping values a=%d
    b= %d\n", a, b);
}
void swap(int a, int b)
{
  int temp;
```

```
temp = a;
a = b;
b = temp;
printf("After swapping values a=%d,
        b=%d\n", a, b);
}
```

Before swap = a=10 ; b=20
After swap = a=10 ; b=20

2. call by Reference :-

→ In call by reference, the address of the variable is passed into the function call as the actual parameter.
→ An Address of value is passed into the function.
→ changes made inside the function validate outside the function also. The values of the actual parameters do change by changing the formal parameters.
→ Actual and formal arguments are created at the same memory location

Note :-

- we make use pointers(*) in call by reference
- pointer

```
temp = a;
a = b;
b = temp;
printf("After swapping values a=%d,
        b= %d\n", a, b);
}
```

Before swap = a=10 ; b=20
After swap = a=10 ; b=20

2. call by Reference :-

→ In call by reference, the address of the variable is passed into the function call as the actual parameter.
→ An Address of value is passed into the function.
→ changes made inside the function validate outside the function also. The values of the actual parameters do change by changing the formal parameters.
→ Actual and formal arguments are created at the same memory location.

Note :-

- we make use pointers(*) in call by reference
- pointer

```
#include <stdio.h>
void swap(int *a, int *b);
int main()
{
  int a=10;
  int b=20;
  printf("befor swapping the value
          a=%d, b=%d\n",a,b);
  swap(&a,&b);
  printf("after swapping values
          *a=%d, b=%d\n",*a,*b);
}
void swap(int *a, int *b)
{
   int temp;
    temp=*a;
    *a=*b
    *b=temp;
} // printf("after swapping values
        a=%d, b=%d\n",*a,*b);
```

Date
Page

Passing Arrays to Function :-

1. passing individual elements
2. passing entire array.

1. passing individual elements

```
#include<stdio.h>
void fun(int x);
int main()
{
  int a[3]={10,20,30};
             0  1  2

  fun(a[1]); //20
}
void fun(int x)
 {
  x=x+20  // 20+20=40
  printf("x=%d\n",x); //40
}
```

2. entire array

```
#include<stdio.h>
void fun(int x[3]);
int main()
{
  int a[3]={10,20,30};
  fun(a);
}
void fun(int x[3])
{
```

```
int i;
for(i=0; i<3; i++)
printf("%d\n", x[i]);
}
```

Non - Recursion :-

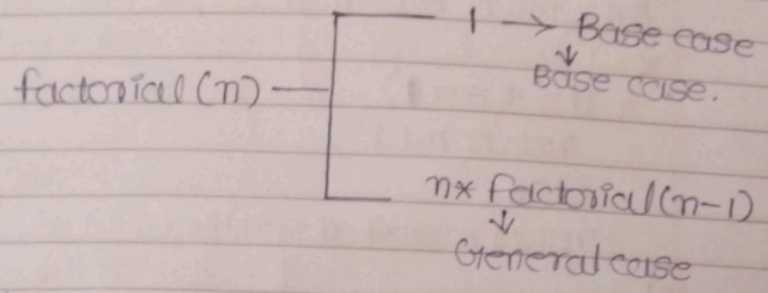
→ A function containing any loop statement in its implemenation is called non-Recursion.

Recursion :-

→ A function calling itself.
(If and else statement)
→ It contain
→ to write a recursion function we need two types.
1. Base case
2. General case.

1. Base case :- every recursion function must have base case based on that it stops the recursion.
2. General case :- The solution of function is return in terms of n is called General case.

Base case :-



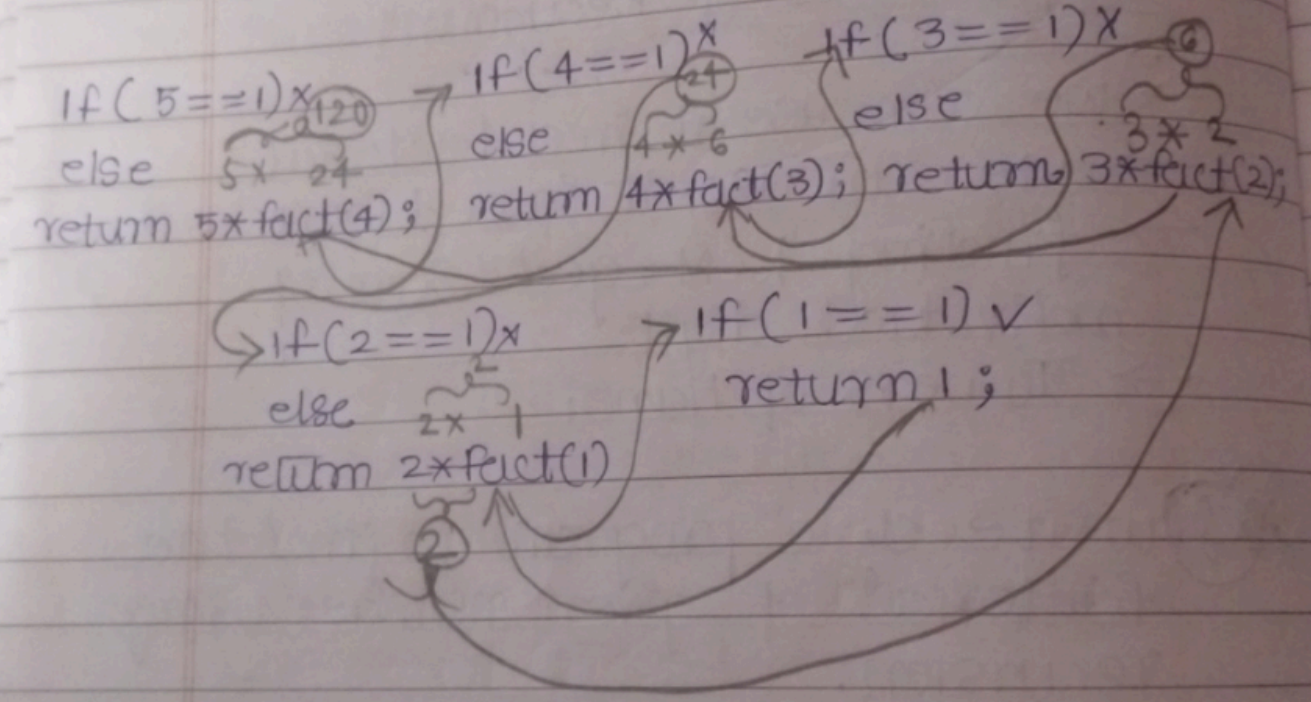
Applications of Recursion :-

→ we can use it in a factorial
→ we can use it in a fibonocciss series
→ finding LCM, gcd, sum of individual digit
→ Towers of Hanoi

(Q.) write a c program to find the factorial of given number using recursion.

```
#include<stdio.h>
int fact(int n);
int main()
{
    int n, res;
    printf("enter n value");
    Scanf("%d", &n);
    res = fact(n);
    printf("factorial = %d", res);
}
```

```
int fact(int n)
{
    if(n==1)
    return 1;
    else
    return n*fact(n-1);
}
```



8. C program to print fibonacci series

fib(n) —
- 0; 1; } Base case
- fib(n-2)+fib(n-1) General case

```
#include<stdio.h>
int fib(int n);
int main()
{
 int res,n;
printf("enter n value");
Scanf("%d",&n);
printf("Fibonacci series is;");
for(i=1;i<=n;i++)
{
 res=fib(i);
 printf("%d",res);
}
}
int fib(int i)
{
 if(i==1)
 return 0;
 else if(i==2)
  return 1;
else
return fib(i-2)+fib(i-1);
}
```

n=1 i=1
i=1 1<=1
res=fib(1)

if(1==1)
return 0;

printf → 0

i=2 2<=1 X

n=2
i=1 1<=2
res = fib(1)

if (1==1)
return 0;

printf – 0
i=2 2<=2
res = fib(2);

if (2==1) x
else if (2==2);
return 1;

printf – 1

printf – 0, 1

n=3
i=1 1<3
res = fib(1);

fib(1) = 0
fib(2) = 1

i=2 2<=3
res=fib(2)
i=3 3<=3
if (3==1) x
else if (3==2) x
else
return fib(i-2) + fib(i-1);

fib(1) fib(2)

1 2
0 1
0+1 = 1

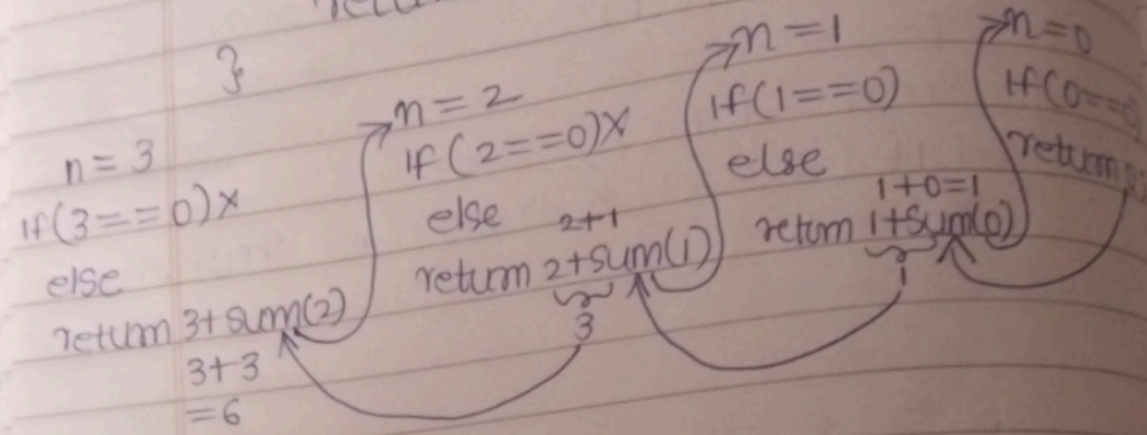
printf → 0, 1, 1

n = 2
i = 1 1<=2
res = fib(1)
i = 2 2<=2
res = fib(2)
if (2==1) *

Q. write a c program to find sum of n natural numbers using recursion.

```
#include<stdio.h>
int sum(int n);
int main()
{
   int n, res;
printf("enter n value");
scanf("%d", &n);
res = sum(n);
printf("sum of n natural numbers
      = %d \n", res);
}
int sum(int n)
{
   if (n==0)
     return 0;
   else
```

return n+sum(n-1);
}



(9) c program to find sum of individual digits of a given number.

```
#include<stdio.h>
int sum(int n);
 int s=0;
int main()
{
 int n, res;
 printf("enter n value");
 scanf("%d",&n);
  res = sum(n);
printf("sum of individual digits=
        %d\n",res);
}
int sum(int n)
{
 if(n==0)
  return 0;
 else
```

```
{ rem = n%10;
  s = s+rem;
  sum(n/10);
}
return s;
}
```

n = 456 ==0x

456==0x
else
{ rem=6
s=0+6=6
sum(456/10)

45==0x
else
rem=5
s=5+5=11
sum(45/10)

4==0x
else
rem=4
s=15
sum(4/10)

Q. C program to find reverse of a given number.

```
#include<stdio.h>
int sum(int n);
int s=0;
int main()
{
int n, res;
printf("enter n value");
scanf("%d",&n);
 res = sum(n);
printf("sum of reverse of a given no
        = %d\n", res);
}
 int sum(int n)
{
  if(n==0)
  return 0;
  else
{  rem = n%10;
     s = s*10+rem;
```

gcd(a, b) — [b=0 return a; / gcd(b, a%b)]

```
        sum = (n/10);
    }
    return s;
}
```

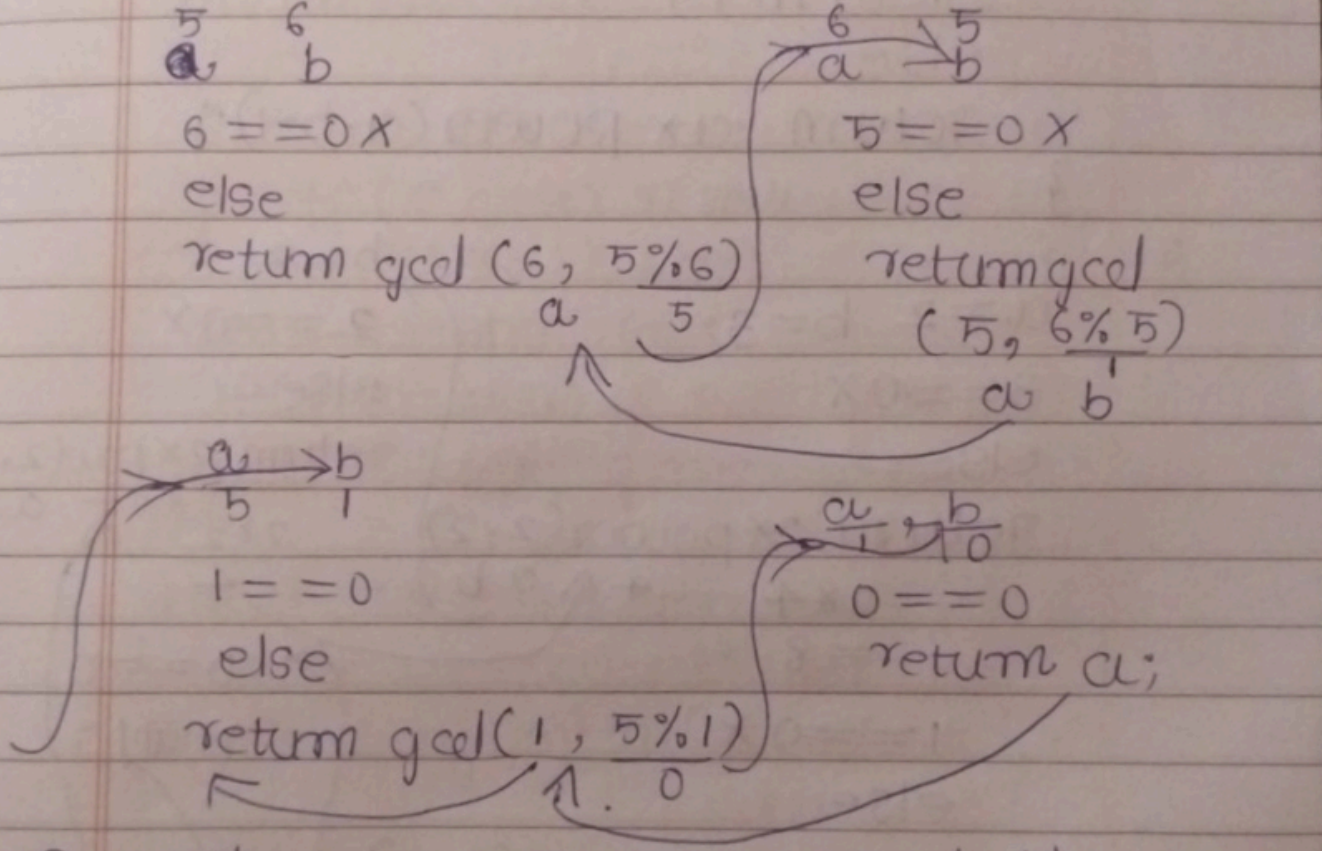
H/w Q. write a c program to find the number given no is Armstrong or not
Q. palinchrone or not.

Q. write a c program to find gcd, lcm of given number.

```
#include<stdio.h>
int gcd(int a, int b);
int main()
{
    int a, b, res;
    printf("enter a,b values");
    scanf("%d%d", &a, &b);
    res = gcd(a,b);
    printf("gcd of two numbers = %d\n",
                res);
    lcm = (a*b)/res;
    printf("lcm of 2 number = %d\n",
                lcm);
}
int gcd(int a, int b)
{
    if (b==0)
    return a;
```

```
else
 return gcd(b, a%b);
}
```

power(a,b) — b==0 return 1; / return a*pow(a,b-1);



8. write a c program to find the power of given number.

```
#include<stdio.h>
int power(int a, int b);
int main()
{
 int a, b, res, item;
 printf("Enter a, b, values");
 scanf("%d %d", &a, &b);
```

```
    res = power(a,b);
    printf("power of a,b = %d\n",res);
}
int power(int a, int b)
{
    if (b==0)
     return 1;
    else
    return a* power(a,b-1);
}
```

a=2 b=3

3==0 X

else

return 2* power(2,2)

2*4

=8

1==0 X

else

return 2* power(2,0)

2*1

=2

b=2

2==0 X

else

return 2* pow(2,1)

2*2 ab

2*2

4

4 ab

0==0

return 1;

Q. write a c program to find the given number is Armstrong or not

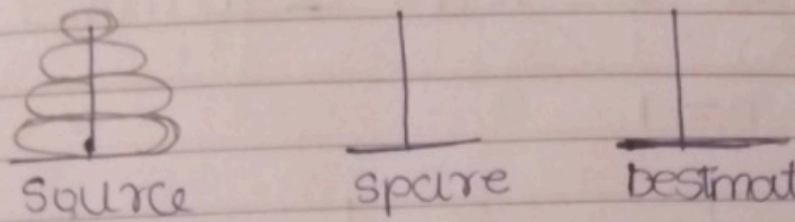
```
# include<stdio.h>
int Armstrong(int n);
int s=0;
int main()
{
 int n, res ;
 printf("enter n value");
 scanf("%d",&n);
 res= Armstrong(n);
 if(res==n)
printf("Amstrong number");
else
printf("Not Amstrong number");
}
int Amstrong(int n)
{
 int rem, s=0;
 if(n==0)
  return 0;
  else
 { rem=n%10;
     s=s+(rem*rem*rem);
      Amstrong(n/10);
 }
 return s;
}
```

Date
Page

(9.) palindrome or not.

```
#include<stdio.h>
int palindrome(int n);
int main()
{
    int n, rem;
printf("enter n value");
scanf("%d",&n);
res = palindrome(n);
if (res==n)
printf("palindrome");
 else
printf("Not palindrome");
}
int palindrome (int n)
{
  int rem, s=0;
   if(n==0)
    return 0;
   else
  {
     rem = n%10;
      s = s*10+Rem;
      palindrome(n);
  }
   return s;
}
```

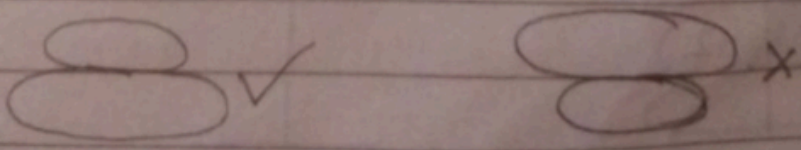
Q. Write a c program to find Towers of Hanoi.



→ It is a mathematical puzzle consists of three rods and n disks initially the source rod includes n disks in different sizes where the small disk is on the top of large disk.
→ The objective is to arrange the stack of disk of the source rod should be moved to the destination rod with the help of spare rod and must be arranged in the same fashion as in the source rod.

Rules:-

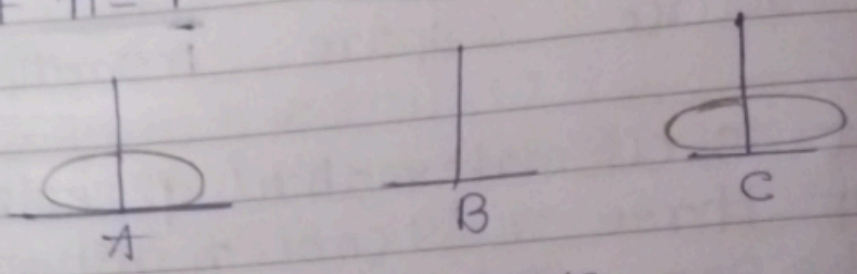
→ only one disk must be moved during each iteration.
→ Always it is allowed to fit the top most disk from the rod.
→ It is allowed to place the small disk on top of large disk



Note:-

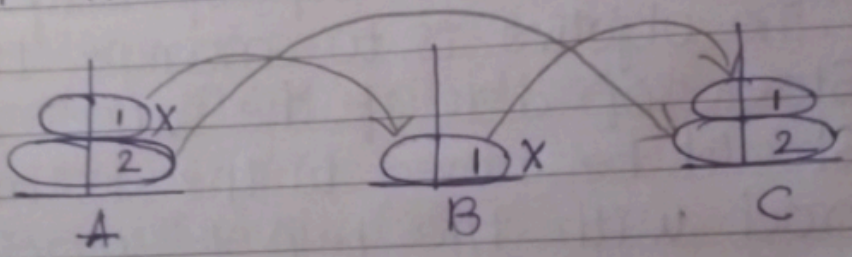
→ If we have a n number of disks then we have $\boxed{2^n - 1}$ moves are done.

① If $n = 1$



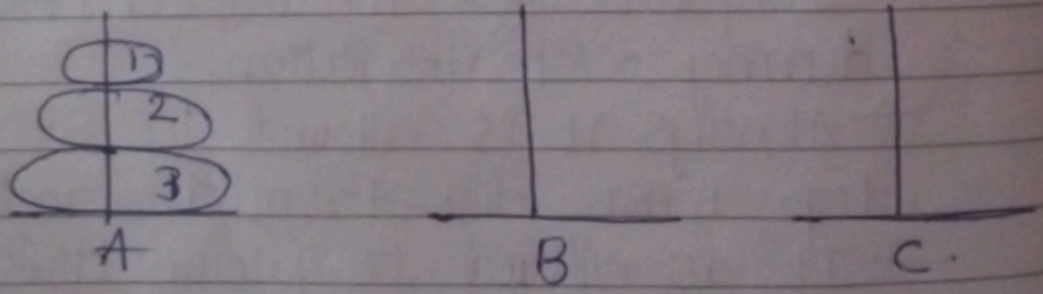
A to C move

② If $n = 2$

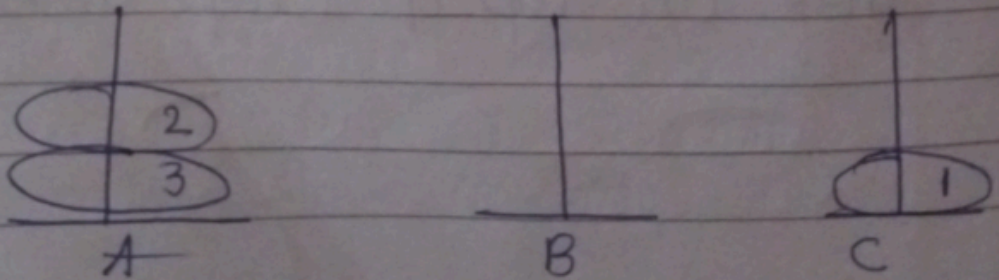


A to B
A to C
B to C

③ If $n = 3$

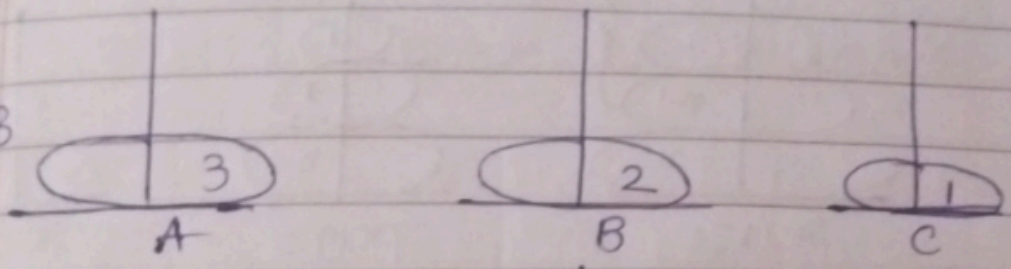


A to C

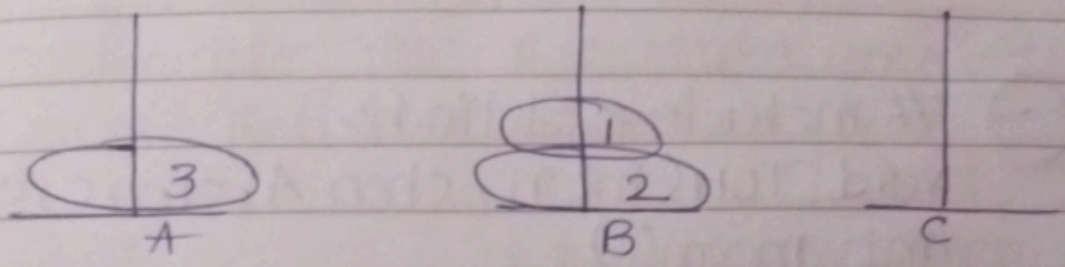


Date
Page

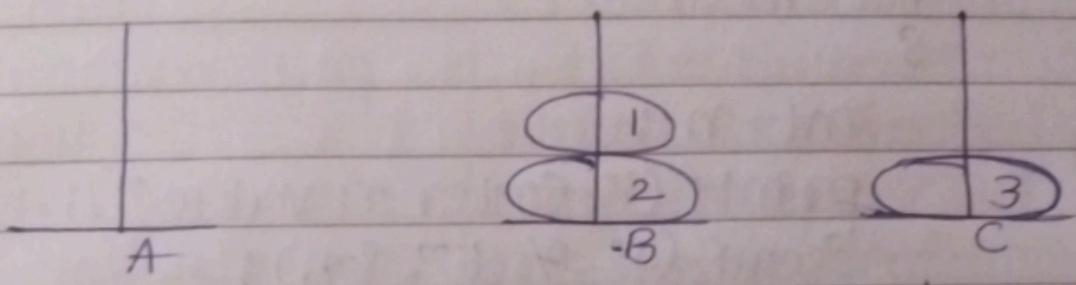
A to B



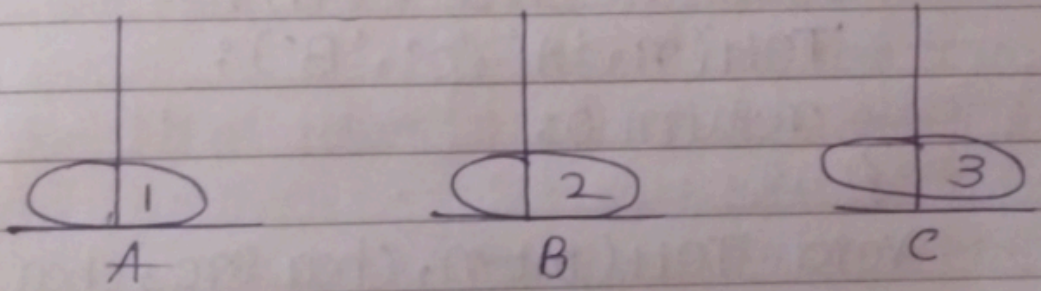
C to B



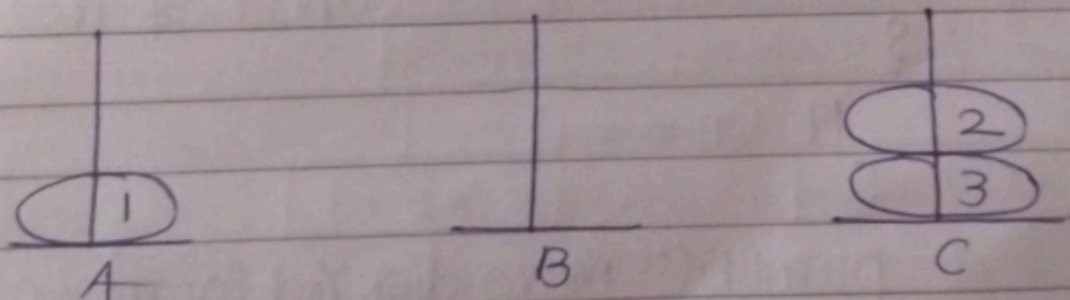
A to C



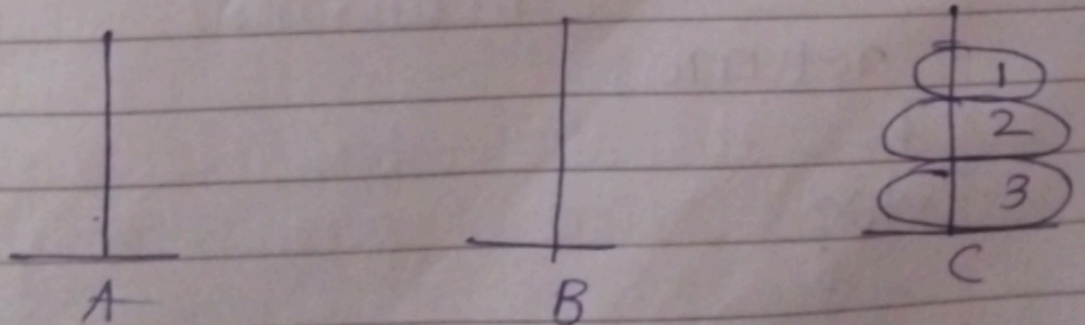
B to A

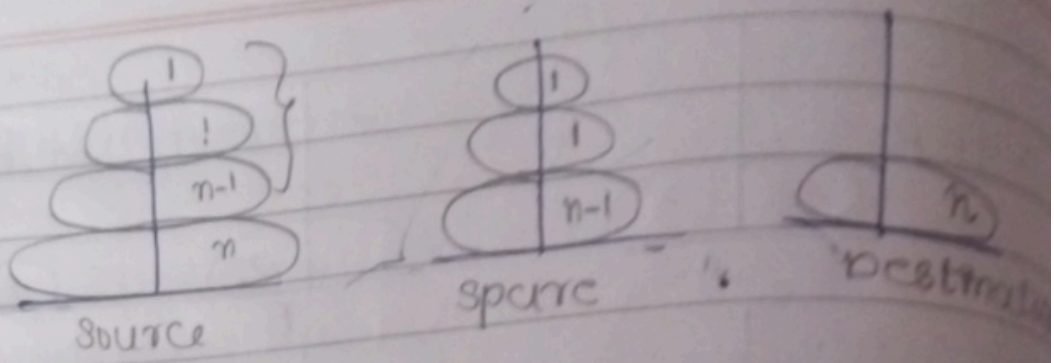


B to C



A to C






```
#include<stdio.h>
void TOH(int n, char A, char C, char B);
int main()
{
  int n;
  printf("enter n value");
  scanf("%d", &n);
  TOH(n, 'A', 'C', 'B');
  return 0;
}
void TOH(int n, char src, char dest,
                    char spare)
{
  if (n==1)
  {
printf("Move disk%d from %c to %c\n",
              n, src, des);
  return;
}
else
   {
     TOH(n-1, source, spare, pest);
printf("Move disk %d from %c to %c\n",
                    n, dest);
                      dest spare
```

Date
Page

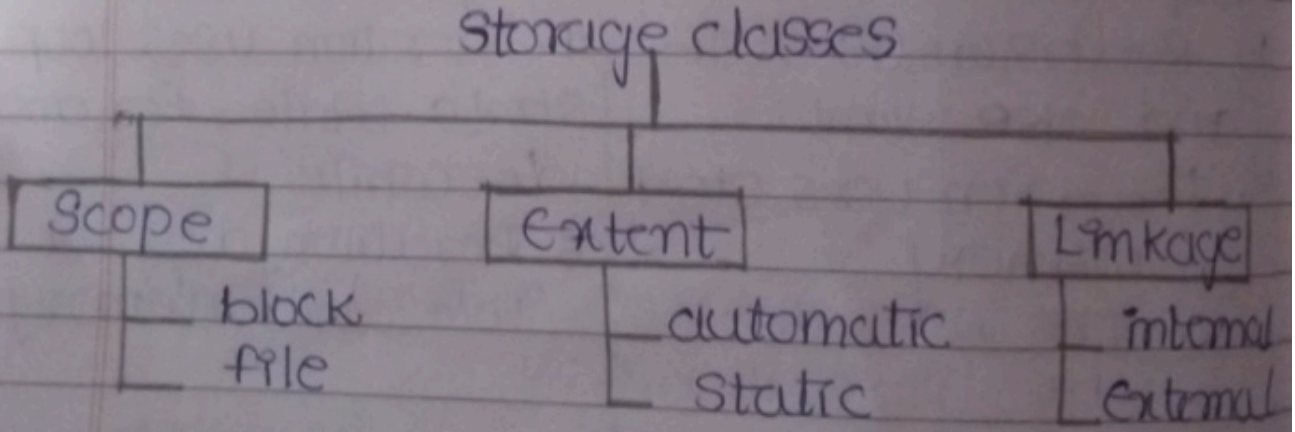
```
    TOH(n-1, spare, dest, source);
}
}
```

Difference b/w Recursion and non-Recursion.

Recursion	Iteration.
1. A function calling itself	1. A set of statements which are repeatedly executing.
2. when the base case satisfies recursions terminates.	2. when the condition fails, iteration terminates
3. when the base case fails it goes to infinite loops	3. when the condition is true all the time, it goes to infinite.
4. Recursion uses if and else stmts.	4. Iteration uses loop stmts while, for and do-while.
5. Recursion uses stack memory	5. Iteration does not use stack memory.

STORAGE CLASSES :-

→ Each and every variable is stored in computer memory.
→ Every variable in C language is associated with datatype and storage class
→ The Syntax :-
storage class datatype variable name;
ex :- auto int a;
int a;
→ Each storage class is associated with 3 features.
→ Storage class specifies the scope, extent, linkage.



block, automatic, internal — local variable for declaration of particular block.

file, static, external - Global variable declaration of variables for entire program code.

Scope :- The visibility of a varaible in the program is called a scope.

It is two types :-

1. block or function scope (Local)
2. file scope (Global)

The statements which are written with in opening and closing braces is block scope.

ex :-

```
int main()
{
    int a=0;
        {
            int a=20;
            printf("%d", a); //20
        }
    printf("%d", a); //10
}
```

2. File scope (Global)

The variables which are declared for the entire program.

```
#include<stdio.h>
int x=10;
void add();
int main()
{
    x=x+10;
    printf("%d", x); //20
```

```
add();
}
void add()
{
x = x + 30;
printf("%d", x); // 50
}
```

EXTENT :-

→ It is also called storage duration or lifetime of a variable.

→ extend defines how much time the variable is alive in the program.

These are 2 types :-

1. automatic
2. static.

1. Automatic extent :- The life time of variable within the block or function

→ The variable is assigned with memory when it enters into the block and gets released when it exit from the block.

2. Static extent :- The life time of variable is throughout the file (program)

→ All static variables are initialised to zero.

** Re-initialization of variable is not possible

LINKAGE :-

A large application program may be broken into several modules, with each module potentially written by a different programmer. Each module is a separate source file with its own objects.
We can define two types of linkage :-
1. Internal
2. External.

1. Internal linkage :- An object with an internal linkage is declared and visible in one module. Other modules cannot refer to this object.

2. External linkage :- An object with an external linkage is declared in one module but is visible in all other modules with a special keyword, extern.

Storage classes

1. Automatic Storage class
2. Register Storage class
3. Static Storage class
4. External Storage class

1. Automatic

The keyword is auto
The default storage class is auto
all automatic variables are stored in Ram.
Automatic variables are declared but it is not initialized, then it contains garbage value.

Scope : block (local)
extent : automatic
Linkage : Internal
default : Garbage value

Syntax :- auto datatype variable name;
Ex :- auto int a;

Ex :- #include <stdio.h>
int main()
{
auto int j=1;
{
auto int j=2;
{printf("%d
auto int j=3;
{
printf("%d ", j); //3
}

```
printf("%d", j); //2
}
printf("%d", j); //1
}
```

output :- 3, 2, 1

2. Register Storage class :-

→ Register variables are similar to auto variables. the only difference is register variables stored in cpu register instead of memory (RAM)
→ Register variables are faster than normal variables
→ mostly programmer use register to store frequently used variables.
→ programmers can store only few variables in the cpu register because size of register is less.

Syntax :- register datatype variable;
ex :- register int a;

```
#include<stdio.h>
int main()
{
  int m1 = 5;
  register int m2 = 10;
  printf("the value of m1 : %d", m1);
  printf("\n the value of m2 : %d", m2);
  return 0;
}
```

The value of m1 : 5
The value of m2 : 10.

Key word	Register.
Declaration	Inside a function/block.
Storage	CPU registers
Default initial value	Garbage value
Scope (visibility)	Within the function
Lifetime (Alive)	Till the end of function / block

Static

Syntax :- static datatype variable;

→ It is a variable which stores its value thorough out the program.
→ The variable can be declared as local or global.
→ A local static variable will be created and initialized only once when the function is executed for the first time.

Auto INT

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

output:- a=1
a=1
a=1

Static int

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

output:- a=1
a=2
a=3

keyword	static(local)	static(global)
Declaration	inside a function/ block	outside all functions
Storage	RAM	RAM
default initial value	zero	zero
Scope(visibility)	with in the function / block	outside all functions

Lifetime (Alive)	Till the end of program	Till the end of program.

Extern

→ The keyword is extern.
→ These are the variables which are declared outside above all the functions.
→ These are also known as global variables
→ These variables can be accessed anywhere in the program and in other programs also

Syntax:- extern int n;
extern datatype variable name;

ex:-
```
# include <stdio.h>
extern int i;
int main()
{
    printf("the value of extern variable is = %d", i);
return 0;
}
# include <stdio.h>
int i = 48;
```

Date.....
Page.....

keyword	extern
Declaration	· outside of all variable
Storage	RAM
Default initial value	zero
Scope	Global multiple file
Life time	Till the end of the program.

Storage class	Declaration	Default initial value	Storage	Scope (visibity)	Lifetime (Alive)
auto	Inside the block/function	Garbage value	RAM	Within block/ function	Till the end of block/Function
Register	Inside the block	Garbage value	CPU register	within block	Till the end of block
static (Local)	Inside the block	Zero	RAM	with in block	Till the end of block
Static (Global)	outside all the function	Zero	RAM	Global (outside all the functn)	Till the end of the program
Extern	Outside all the variable	Zero	RAM	Global (multiple files)	Till the end of program.

Date
Page

pointers:-

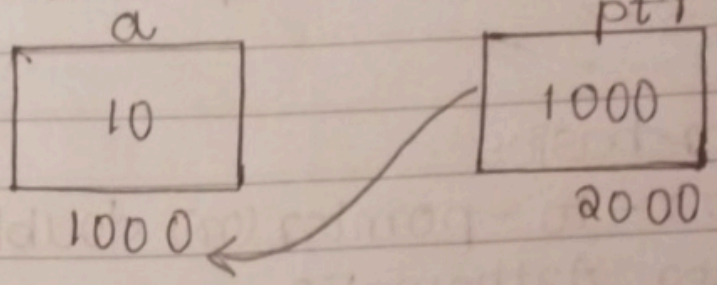
1 pointer basics
pointer-to-pointer (or) double pointer
pointer Arthemetic
Generic pointer
Array of pointer
function returning pointers
Dynamic memory allocation.

→ pointer is a derived datatype.

→ A pointer variable holds the Address of another.
* → indirection operator
Declaration of a pointer:-
Syntax:- datatype *pointer variable;
Ex:- int *ptr;
float *ptr;

Access variable using pointer:-
→ once a pointer variable is declared and initialized we can access the value of variable using pointer variable that is * operator.
→ where * operator is also known as indirection operator.
Syntax:- *pointer variable = & variable;
Ex:- *ptr = &a;
(or)

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




```
#include<stdio.h>
int main()
{
  int a=10;
  int *ptr;
  ptr= &a;
printf("value of a=%d\n",a); //10
printf("Address of a=%u\n",&a);//1000
printf("Address of ptr=%u\n",ptr);//1000
printf("value of ptr=%d\n",*ptr);//10
}
```

ptr =&a
ptr = 1000
&ptr = 2000

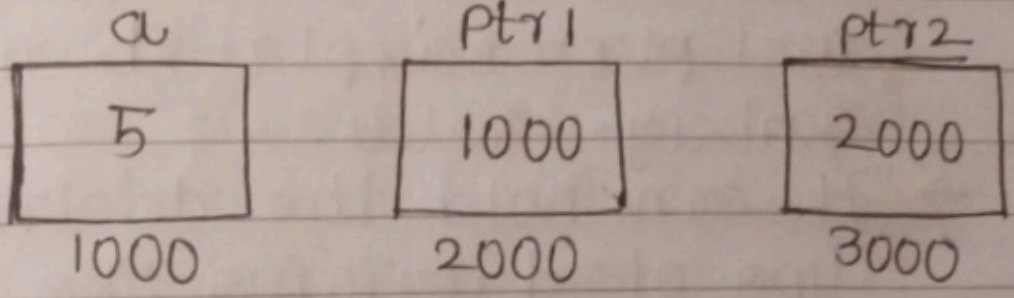
Applications :-

→ It will allocate size at runtime

- Dynamic memory allocation.
- Array & string
- File handling system
- To access and manipulate the address.
- More efficient
- Save memory space

pointer to pointer (or) Double pointer.

→ The pointer variable which holds the address of another pointer variable.



✓
```
#include<stdio.h>
int main()
{
  int a=5;
  int *ptr1, **ptr2;
  ptr1 = &a;
  ptr2 = &ptr1;
printf("value of a = %d\n", a);
pf("Address                    &ptr2
                               &ptr1
                               , &a);
                               , ptr1);
                               , ptr2);
                               , *ptr1);
                               , **ptr2);
```

type conversion = *(data type *)ptr;

Generic pointer (or) void pointer;

ptr = &a
ptr = &b
ptr = &z
it will only check and print ptr=&z and ptr=&a, ptr=&b it will print garbage value.

→ It can hold the address of any type of datatitems like int, float, char, double etc.

→ To read or print the data we must use type conversion.
((datatype *))

ex:

```
# include <stdio.h>
{
    void *ptr;
    int a = 10;
    float b = 20.5;
    char z = 'A';
    ptr = &a;
printf("value of a=%d", *(int *)ptr);
 ptr = &b;
printf("value of b=%f", *(float *)ptr);
   ptr = &z;
printf("value of z=%c", *(char *)ptr);
}
```

pointer variable = ptr1, ptr2
pointer values = *ptr1, *ptr2

pointer Arthematic :-

→ pointer variables are used to store address of variables.
→ Address of any variable is an unsigned integer value that is a numerical value. so, we can perform Arthematic operations on pointer values.
+, -, *, /, %
✓ a=5, b=7
- ptr1 = &a
- ptr2 = &b

(*ptr1) + (*ptr2) = 5+7
(*ptr1) - (*ptr2) = 5-7
(*ptr1) * (*ptr2) = 5*7
(*ptr1) / (*ptr2) = 5/7
(*ptr1) % (*ptr2) = 5%7

→ But when we perform arthematic operation on pointer variable the result depend on amount of memory required by the variable to which the pointer is pointing.
→ In c programming language we can perform the following arthematic operations on pointers.
1. Addition 2. Substraction
3. Increment 4. Decrement

int a = 5
int ptr = &a

① ptr = ptr + 3
= 1000 + 3 x (size of datatype)
= 1000 + 3 * 2
= 1006

② ptr = ptr - 3
= 1000 - 3 * 2
= 1000 - 6
= 994

③ ptr++
ptr = ptr + 1
= 1000 + (1 * datatype)
= 1000 + 2
= 1002

→ In c programming language the addition or subtraction operation on pointer variables is represented by using the formula.
current address of ptr ± (value to be added / subtracted * size of datatype)

```
#include<stdio.h>
int main()
{
int a, *int ptr;
float b, *float ptr;
char c, *char ptr;
int ptr = &a; // Assume address of a is 1000
```

```
float ptr=&b; // Assume address b is 2000
char ptr=&c; // " " c is 3000
printf("Address of int ptr=%u\n",
    int ptr); // 1000
printf("Address of float ptr=%u\n",
        float ptr); // 2000
printf("Address of char ptr=%u\n",
        char ptr); // 3000;
int ptr = int ptr+3 // 1000+(3*4)=1012
float ptr = float ptr+4 // 2000+(4*4)=2016
char ptr = char ptr+5 // 3000+(5*1)=3005
```

→ Increment & Decrement operations on pointer variables can be calculated as current address ± (1 * size of (datatype)) value of ptr.

```
#include<stdio.h>
int main()
{
  int a, *int ptr; float b, *float ptr.
  int ptr++; // 1000+(1*4)=1004
  float ptr++; // 2000+(1*4)=2004
  char ptr++; // 3000+(1*1)=3001
printf("Address of int ptr=%u\n",
        int ptr); // 1004
printf("Address of float ptr=%u\n",
        float ptr); // 2004
printf("Address of char ptr=%u\n",
        char ptr); // 3000)
```

→ passing pointers to functions :-
(refer call by reference both theory & program)

→ pointers & Arrays :-

→ The elements of array can be accessed by using pointers

→ continuous memory locations are allocated for all the elements of the array by the compiler.

→ The base address is the location of the first element with index zero of the array.

ex :- int a[5] = {10, 20, 30, 40, 50}

Index	a[0]	a[1]	a[2]	a[3]	a[4]
value	10	20	30	40	50
Address	1000	1004	1008	1012	1016

↓

base address

a = a[0] = 1000

↔ Normal representation for reading 10 elements in array.

```
for (i=0; i<=n; i++)
scanf ("%d", &a[i]);
```

a[0] = (a+0) = *(a+i) = 10

(1000+0) = *(1000+0) = 10

a[1] = (a+1)

(1000+1*4) = *(1004) = 20

a[i] = (a+i) = *(a+i);

Read & print 1D array elements using pointers.

Read

```
for(i=0; i<n; i++)
scanf("%d", (a+i));
```

print

```
for(i=0; i<n; i++)
printf("%d", *(a+i)); & print
```

Q) write a c program to read 1D array elements using pointers.

```
#include<stdio.h>
int main()
{
  int a[5], i, *ptr;
ptr = a or ptr = &a[0];
printf("enter array elements:");
for(i=0; i<5; i++)
scanf("%d", (a+i));
printf("Array elements:");
for(i=0; i<5; i++)
print("%d", *(ptr+i));
}
```

→ Representation of 2D array elements:

a[i] — *(a+i) — 1D

a[i][j] — *(*(a+i)+j) — 2D

Q) write a c program to print 2D array elements using pointers

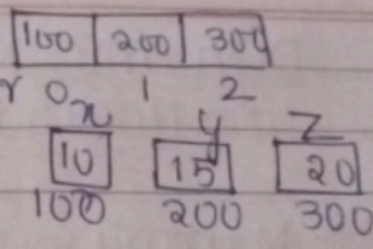
```
#include<stdio.h>
int main()
{
    int a[3][3]={10,20,30,40,50,60,70,80,90};
    int *ptr,i;
    ptr = &a[0][0];
    for(i=0; i<3; i++)
    {
     for(j=0; j<3; j++)
      {
       printf("%d", *(*(a+i)+j));
      }
     printf("\n");
    }
}
```

✓ Arrays of pointers :-

→ Array is a collection of variables of same datatype.

→ Smilliary Array of pointer is a collection of Address this contains address of more than one variable of same datatype in it.

Ex:
```
#include<stdio.h>
int main()
{
 int *ptr[3];
 int x=10, y=15, z=20;
 ptr[0]=&x;
```




```
ptr[1] = &y;
ptr[2] = &z;
for (i=0; i<3; i++)
printf ("Address of ptr[%d] = %d\n",
        (ptr+i), *(ptr+i));
}
```

→ Function returning pointers:-

→ A pointer which keep a Address of a function. is known as Function pointer.

✓
```
#include <stdio.h> int *abc();
int main()
{
    int *ptr;
    ptr = abc();
printf("Address of ptr = %u\n", ptr);
printf("Value of ptr = %d\n", *ptr);
}
int *abc()
{
   int *p, a = 100;
      p = &a;
     return p;
}
```

Dynamic Memory allocation / Memory allocation functions :-

→ Memory allocation is mainly of two types.
1. Static memory allocation
2. Dynamic memory allocation

→ Static memory allocation means memory is allocated to variables at complie.

→ Dynamic memory allocation means memory is allocated to variables at run time or Execution Time.

→ The memory used by the program is mainly categories into three types:
1. The program instructions and global variables are stored in a region known as permanent storage area.
2. The local variables are stored in Stack memory.
3. memory space b/w two areas is available for Dynamic allocation during execution of program this free region is called heap.
4. the size of heap is keeps on changing

local variables	} Stack
Free memory	} Heap
Global variables	} Permanent storage area
Instructions.	

→ there are four allocation functions which are used to allocate the memory for variables at run time.

1. malloc ( )
2. calloc ( )
3. realloc ( )

used for memory allocation.

→ free ( ) is used for releasing the memory when not required.

#include <stdlib.h> — header file.

1. Malloc ( )

→ Malloc ( ) is a block memory allocation function

→ The return type of malloc ( ) function is void

→ It contains the address of first byte of memory allocated from heap.

→ If the memory is successfully allocated then it returns the address otherwise it returns null.

→ The default value stored in the memory is garbage

Syntax:-

```
void * malloc (sizeof (datatype));
```

Ex:- `int * x;`

`x = (int *) malloc (sizeof (int));`

```
void * malloc (n * sizeof (datatype));
```

→ for Arrays

5 * 4 = 20

9. write a c program to read elements dynamically using malloc() function and interchange them using call by reference.

```
#include <stdio.h>
#include <stdlib.h>
int swap (int *x, int *y);
int main()
{
    int *a, *b;
  a = (int *) malloc (sizeof (int));
  b = (int *) malloc (sizeof (int));
printf ("enter a,b values:");
Scanf ("%d %d", a, b);
 Swap (a,b);
printf("After swapping a=%d b=%d\n",
                *a, *b);
}
int swap(int *x, int *y)
{
```

```
int temp;
temp = *x;
*x = *y;
*y = temp;
}
```

o/p :- Enter a, b values
10 20
After swapping
a = 20, b = 10

Q. write a c program to find sum of n given number using malloc() function.

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

int main()
{
  int *a, sum=0, n;
  printf("enter n value");
  scanf("%d", &n);
  a = (int *) malloc(n*sizeof(int))
printf("enter array elements");
for(i=0; i<n; i++)
{
  scanf("%d", (a+i));
  sum = sum + *(a+i);
}
```

```
    printf("sum=%d\n", sum);
}
```

(8.) write a program to read the element of 1D array using pointers and print them in reverse order.

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int *a, n, i;
    printf("enter n value");
    scanf("%d", &n);
    a = (int *) malloc(n * sizeof(int));
    printf("enter array elements");
    for(i=0; i<n; i++)
    scanf("%d", (a+i));
    printf("Array elements are");
    for(i=n-1; i>=0; i--)
    printf("%d", *(a+i));
}
```

8. calloc()
→ calloc() is a continuous memory allocation function
→ It is generally used for Arrays
→ the return type of calloc() is void
→ It returns address of first byte of memory allocated from heap.
→ If the memory is succesfully allocated then it return the address otherwise it return null.
→ The default value for all the variables is zero.
Syntax:- void *calloc(int size, size of (datatype));

Ex:- int *x;
x = (int *) calloc(2, size of (datatype))
x = (int *) calloc(2, sizeof (int));

✓ Q. Write a program to read and print elements of 1D Array using calloc() function.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int *a, n, i;
printf("enter n value");
scanf("%d", &n);
```

```
a=(int *) calloc (n, sizeof (int));
printf("enter array elements");
for(i=0; i<n; i++)
  scanf("%d", a+i);
printf("Array elements are:");
// for(i=n-1; i>=0; i--)
  printf("%d", *(a+i)); //
}
```

realloc ():- reallocation

→ reallocation is used to change the size of memory block without losing the old data.
→ The return type of realloc is void
→ The default values are null values

Syntax:-

```
void * realloc(void * old pointer,
                newsize * (Size of (datatype
```

Ex:- `int *x;`

```
x=(int *)calloc(2, Size of (int));
x=(int *)realloc(x, 10* Size of (int)
```

```
# include <stdio.h>
# include <stdlib.h>
int main()
{
  int *a, n, i, n1;
```

```
printf("enter n value");
scanf("%d",&n);
a=(int*)calloc(n,sizeof(int));
printf("enter array elements:");
for(i=0;i<n;i++)
scanf("%d",(a+i));
printf("Array elements are:\n");
for(i=0;i<n;i++)
scanf("%d",*(a+i));
printf("enter new size:");
scanf("%d",&n1);
a=(int*)realloc(a,n1*sizeof(int));
printf("enter array elements:");
for(i=0;i<n1;i++)
scanf("%d",(a+i));
printf("Array elements are:\n");
for(i=0;i<n1;i++)
printf("%d",*(a+i));
}
```

free() :-

→ free() is used to release the memory which is allocated by using any one of the memory allocation functions.

Syntax: void free(void * pointer);

```
int *a;
free(a);
```

→ It is always better to assign NULL for a pointer variable after using free() function.

```
free(a);
a = NULL;
```

String

String :- A string is a sequence of character which is terminated by delimiter ('\0') and is enclosed in dobble qotation marks(" ") string is also known as character of Array.

Declaration of string :-

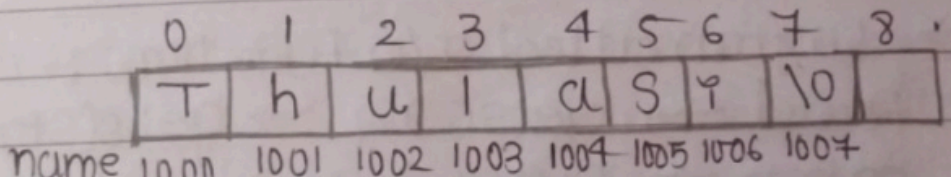
Syntax :- char stringname[size];

ex :- char name[20];

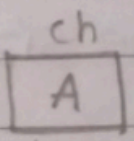
Initialization :-

char name[20] = "Thulasi";

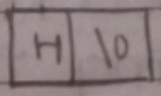
chare name[20] = {'T','h','u','l','a','s','i','\0'};



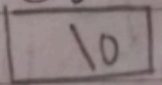
char ch = 'A'



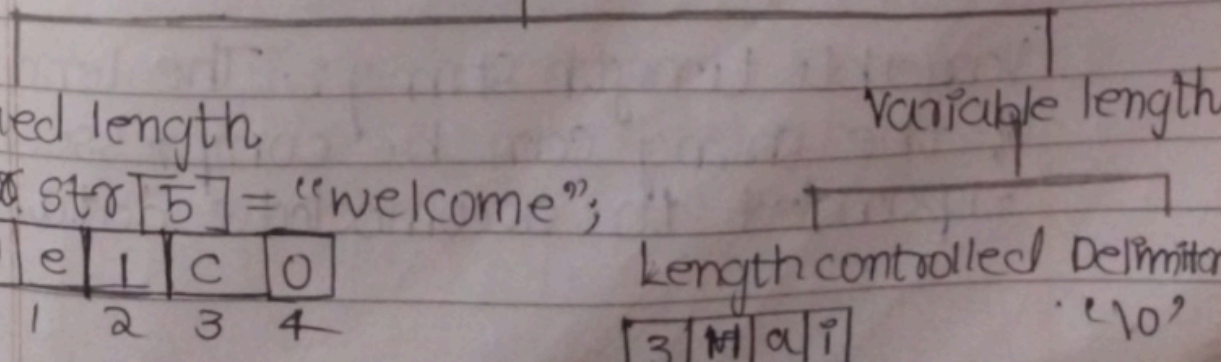
char ch = "H"



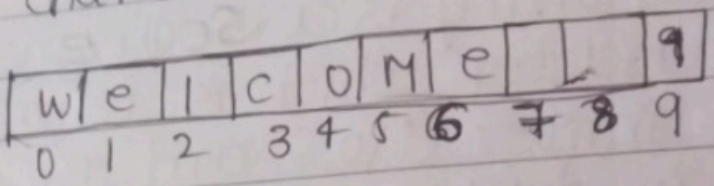
char ch = "";



String classifications



char str[10] = "welcome";



Variable length
1. length controlled :- By giving string the size memory will be wasted

String Input/output Functions :-

➡ Format specifier for string is %s
→ Formatted I/O functions for string
scanf("%s", str);
printf("%s", str);

→ unformatted I/O functions :-
Read → gets() → gets(str);
print → puts() → puts(str);

Fixed length string :- The size of string is fixed
disadvantages :- If we take to small size string we can't store it. If we take it as two big than wastage of memory.

Variable length string :- The length of the string can be compressed or expanded to accomodate data.

Length stored string :- These strings added a count which specifies the number of characters in the string at the starting itself.

Delimitter string :- This string end with delimitter at the end of the string the most common delimitter is null character '\0'.
The ascii value of null character is zero.

→ String is also known as character of Array
→ No need of for using loop because of %s
→ No need to place & in scanf it will automally takes base Address.
→ There is a disadvantage of scanf if we write space like " %s " it will take a null character '\0'.
→ gets(str) takes more and more variables until we enter press the enter button.

Q. write a c program to read and print string using formatted Input / output function.

Formatted :-

```
#include<stdio.h>
int main()
{
    char name[20];
    printf("enter string:\n");
    scanf("%s", name);
    printf("The string is: %s", name);
}
```

ouput :-

```
enter string:
Hello world
The string is:
Hello
```

unformatted :-

```
#include<stdio.h>
int main()
{
  char name[20];
  printf("enter string:\n");
  gets(name);
  puts(name);
}
```

output :-

```
Enter string:
   Hello world
Hello world
```

getchar(); I/p
putchar(); O/P

Character of Strings :-

```
char str[5][10] = {"Delhi", "Mumbai",
                   "Hyderabad", "Jaipur",
                   "Kolkata"};
```

	0	1	2	3	4	5	6	7	8	9
0	D	e	l	h	i	\0				
1	M	u	M	b	a	i	\0			
2	H	y	d	e	r	a	b	a	d	\0
3	J	a	i	p	u	r	\0			
4	K	o	l	K	a	t	a	\0		

A - 65
a - 97

Date
Page

character library function/ character input/output functions:-

Input - getchar()
output - putchar()
Header file is → #include<ctype.h>

Syntax:-

isalpha()	int isalpha(char c);
is digit()	int isdigit(char c);
isupper()	int isupper(char c);
is lower()	int islower(char c);
tolower()	char tolower(char c);
toupper()	char toupper(char c);

Description:-

1. If the given character is alphabetic, it returns true(1) else returns false(0)
2. If the given character is digit, it returns true(1) else returns false(0)
3. If the given character is upper, it returns true(1) else returns false(0)
4. If the given character is lower, it returns true (1) else (0)
5. It converts the given upper case character to lower case
6. It converts the given lower case character to upper case.

Example:-
isalpha(e) → returns 1
isalpha(9) → returns 0

isdigit(2) → returns 1
isdigit(a) → returns 0

isupper(G) → returns 1
isupper(g) → returns 0

islower(i) → returns 1
islower(I) → returns 0

tolower(J) → j
tolower(j) → j

toupper(H) → H
toupper(H) → (H)

* String handling functions (or) string manipulation function :-

→ To perform some operations on string we can use string handling functions.
→ To use string handling function we include the header file is
#include<string.h>

1. string length :- strlen()

→ This function is used to find the length of string.
→ It excludes the null character.
→ syntax :- int strlen(char s[20]);
size

ex :- char s[20] = "Hello world";
int l = strlen(s);
length = 11

Q. write a c program to find the length of string using strlen().

```
#include<stdio.h>
#include<string.h>
int main()
{
    char s[20]; int l;
    printf("enter string:");
    gets(s);
    l = strlen(s);
    printf("length of string is: %d", l);
}
```

Q. write a c program to find the length of string without using strlen().

```
#include<stdio.h>
int main()
{
  char s[20], int i, c=0;
  printf("enter string:");
   gets(s);
  for(i=0; s[i]!='\0'; i++)
     c++;
  printf("length of string=%d\n",c);
}
```

H	e	l	l	o	\0
0	1	2	3	4	5

Length = 5

String comparison function :-

→ returns integer value.

→ string comparison function is used

→ to compare two strings

→ The two string comparison functions are

1. strcmp()
2. strncmp()

1. strcmp()

Syntax:-

```
int strcmp(char s1[20], char s2[20]);
```

2. strncmp()

```
int strncmp(char s1[20], char s2[20], int
```

n → Represents specified no. of characters

→ If both strings are equal it returns '0'.
→ If string1 is less than string 2 it returns -ve
→ If string1 is greater than string 2 it returns +ve

Str1 = Str2 → 0
Str1 < Str2 → -ve
Str1 > Str2 → +ve

Ex: Str1 = "Hello"; } returns 0.
Str2 = "Hello";

Str1 = "Abcd" A = 65 65-97 = -
Str2 = "abcd" a = 97
65 < 97 → returns -ve

Str1 = "abcd"; A = 65 a = 97
Str2 = "Abcd" 97 - 65 = +
97 > 65 → returns +ve

(8.) write a c program to check wheather two strings are equal or not using strcmp function.

```
# include<stdio.h
# include<string.h>
int main()
{
    char s1[20], s2[20];
```

```
int d;
printf("Enter string 1:");
gets(s1);
printf("Enter string 2:");
gets(s2);
d = strcmp(s1,s2)
if(d==0)
printf("Two strings are equal");
else if(d<0)
printf("string 2 is greater");
else (d>0)
printf("string 1 is greater:");
}
```

Q. Write a c program to check wheather the two strings are equal or not without using strcmp()

```
#include<stdio.h>
int main()
{
    char s1[20], s2[20];
     int l1, l2, i, flag=0;
    printf("Enter string1:");
    gets(s1);
    printf("enter string 2:");
    gets(s2);
    l1 = strlen(s1);
```

```
for(i=0; i<l1 || i<l2, i++)
{
    if(s1[i] != s2[i])
    {
        flag=1;
        break;
    }
}
if (flag==0)
printf("Two strings are equal");
else
printf("Two strings are not equal");
}
```

String concatenation:-

→ String concatenation function is used to combine one string to the end of another string.
→ The function returns the Address pointer to the destination string.
→ These are two types
1. Strcat()
2. Strncat()

1. Strcat()

Syntax:- strcat(char s1[20], char s2[20]);
Ex: str1 = "concate";
str2 = "nation";

Str1 = "concatenation";

2. strncat()

Strncat(char s1[20], char s2[20], int n);

Ex:-

```
Str1 = " wel";
str2 = " come";
Strncat(Str1, Str2, 3);
Str1 = " welcom";
```

(9) write a c program to concatenate two strings using strcat().

```
#include<stdio.h>
#include<string.h>
int main()
{
    char s1[20], s2[20];
    printf("enter string1:");
    gets(s1);
    printf("enter string2:");
    gets(s2);
    strcat(s1, s2);
    puts(s1);
}
```

(10) write a c program to concatenate two strings without using strcat()

```
#include<stdio.h>
#include<string.h>
int main()
{
  char s1[20], s2[20]; int i, l1, l2;
  printf("Enter string 1:");
  gets(s1);
  printf("Enter string 2:");
  gets(s2);
  l1 = strlen(s1);
  l2 = strlen(s2);
  for(i=0; i<l2; i++)
  {
    s1[l1++] = s2[i];
  }
  s1[l1] = '\0';
  puts(s1);
}
```

s1 | w | o | r | l | d | \0 |

s2 | H | e | l | l | o | \0 |

s1 | w | o | r | l | d | h | e | l | l | o | \0 |

String copy function :-

→ String copy function is used to copy one string content to another string

→ these are of two types

1. strcpy()
2. strncpy()

1. strcpy()
Syntax:-
strcpy(char destination string[20], char sourstring[20]);
*destingstring
*Source string

ex:-
str1="Hai"
strcpy(str2, str1);
str2="Hai";

2. strncpy()
syntax:-
strncpy(char destinationstring[20], char sourstring[20], int n);

ex:-
str1="Hai"
strcpy(str2, str1, 2);
str2="Ha";

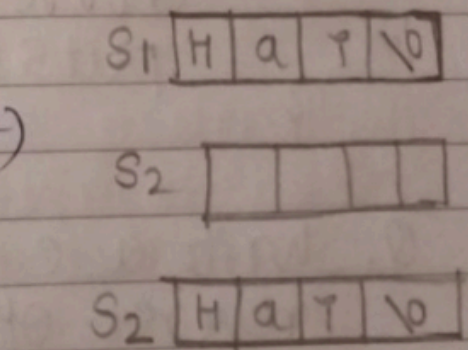
Q. Write a c program to copy the string one to another string using strcpy()

```
#include<stdio.h>
#include<string.h>
int main()
{
    char s1[20], s2[20];
    printf("Enter string1:");
    gets(s1);
    strcpy(s2, s1);
    puts(s2);
}
```

Date
Page

(9) write a c program to copy the string one to another without using strcpy.

```
#include<stdio.h>
#include<string.h>
int main()
{
  char s1[20], s2[20]; int l1, i;
 printf("enter string:");
  gets(s1);
 l1 = strlen(s1);
 for(i=0; i<l1; i++)
  {
    s2[i] = s1[i];
  }
  s2[i] = '\0';
  puts(s2);
}
```



String reverse :- strrev()

→ Strrev() function is used to reverse the given string.

Syntax :- strrev(char stringname[size]);

ex :- str1 = "Hello";
strrev(str1);
str1 = "olleH";

Q. write a c program to print the given string in reverse order using strrev().

```
#include<stdio.h>
#include<string.h>
int main()
{
    char s[20];
    printf("enter string:");
    gets(s);
    strrev(s);
    puts(s);
}
```

Q. Write a c program to find the reverse of string without using strrev().

```
#include<stdio.h>
int main()
{
    char s[20];
    int i, l;
    printf("enter string:");
    gets(s);
    l = strlen(s);
    for(i=l-1; i>=0; i--)
    printf("%c", s[i]);
        (or)
// putchar(s[i]);//
}
```

9. write a c program to check whether the given string is palindrome or not.

```
# include<stdio.h>
# include<string.h>
int main()
{
   char s1[20], s2[20];
  printf("enter string:");
   gets(s1);
   strcpy(s2,s1);
   strrev(s1);
   if (strcmp(s1,s2)==0)
     printf("palindrome");
   else
     printf("not a palindrome");
}
```

8. write a program to replace lower case character to upper case & upper to lower characters.

```
#include<stdio.h>

int main()
{
   char s[20];
    int i;
```

A=65 A to a add +32
a=97 a to A sub -32

```
printf("enter string?");
gets(S);
for(i=0; S[i]!='\0'; i++)
{
    if(S[i]>='A' && S[i]<='Z')
      S[i]=S[i]+32;
    else if(S[i]>='a' && S[i]>='z')
      S[i]=S[i]-32;
}   else S[i]=' ';
puts(S);
}
```

Q. Write a C program delet n character in a given string based on specified position.

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

n=3

p	r	ø	g	r	a	m	\0
0	1	2	3	4	5	6	7

strlen=7 pos=2
n=3
i=2, j=2+3, 5<7
S[2]=S[5]
i=3, j=6, 6<7
S[3]=S[6]
i=4, 7<7 X

p	r	a	m	\0

```
#include<stdio.h>
#include<string.h>
int main()
{
    char s[20];
    int i, j, l, n, pos;
    printf("enter string:");
    gets(s);
    l = strlen(s);
    printf("enter position");
    scanf("%d", &pos);
    printf("enter number of characters
            to be deleted:");
    scanf("%d", &n);
    for(i=pos, j=pos+n; j<l; i++, j++)
        s[i] = s[j];
        s[j] = '\0';
        puts(s);
}
```

Substring → strstr()

→ This function is used to check sub string is present in mainstring or not

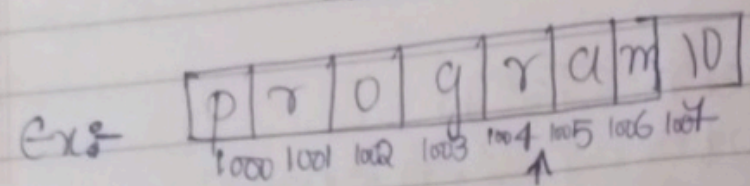
→ This return the position of sub string in the given mainstring

→ It returns the pointer to the substring.

Syntax:-

```
char *strstr(char mainstring[],
             char substring[]);
```

Ex:-




```
char *pos
pos = 1004      ms = 1000
= pos - ms + 1
= 1004 - 1000 + 1
= 4 + 1
= 5   (%d)
```

Q. write a c program to check substring is present in mainstring or not if substring present return the position otherwise return -1.

```
#include<stdio.h>
#include<string.h>
int main()
{
   char ms[20], ss[20], *pos;
   printf("enter main string:");
   gets(ms);
   printf("enter substring:");
   gets(ss);
   pos = strstr(ms, ss);
   if(pos)
```

printf("%s is present in %s at %d position:", ss, ms, pos-ms+1);
else
return -1;
}

strchr():-
→ It returns the position of given character in the given string from the beginning of the string.
Syntax:- char *strchr(char stringname[], char ch);
ex:- str = "program"
strchr(str, 'r');

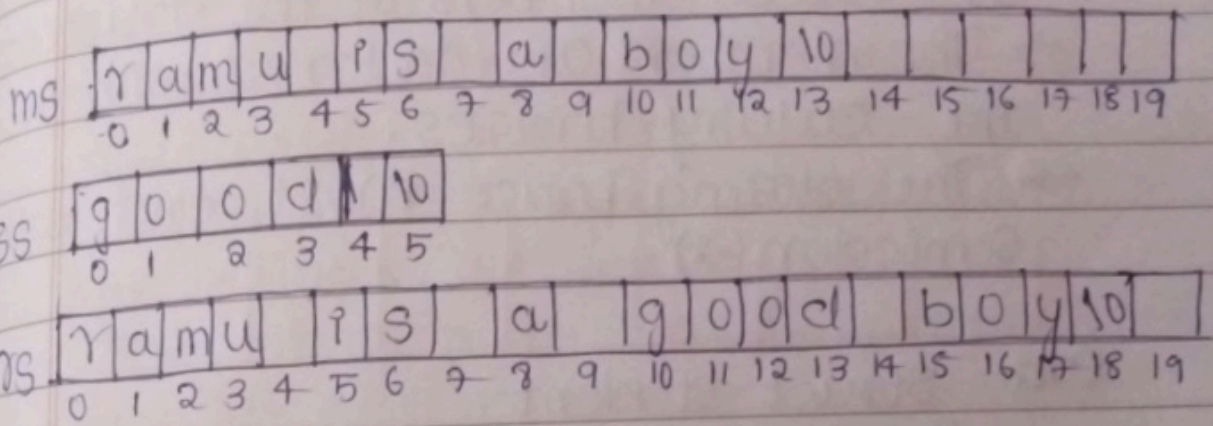
strrchr():-
syntax:-
char *strrchr(char stringname[], char ch);
ex:- str = "program"

Strrchr(str, 'r');
→ It returns the position of given character in the given string from the ending of the string.

write a program to insert sub-string into main string.

```
#include<stdio.h>
#include<string.h>
int main()
{
  int pos, ml, sl;
  char ms[30], ss[30], rs[30];
  printf("enter main string:");
  gets(ms);
  printf("enter substring:");
  gets(ss);
  printf("enter position:");
  scanf("%d", &pos);
  ml = strlen(ms);
  sl = strlen(ss);
  if(pos>ml)
  {
    for(i=0; i<sl; i++)
    ms[ml++]=ss[i];
    ms[ml]='\0';
    puts(ms);
  }
  else
  {
   for(i=0; i<pos; i++)
    rs[i]=ms[i];
    for(j=0; j<sl; j++)
    rs[i++]=ss[j];
   for(j=pos; j<ml; j++)
    rs[i++]=ms[j];
```

```
    rs[i] = '\0';
    puts(rs);
  }
}
```



Structure :- Structure is a collection of dissimilar (heterogenous) datatems stored in a continuous memory locations under a single name.

Structure can be declared in two ways:-
1. Tagged structure
2. Type definition structure.

1. Tagged structure

Syntax :-

```
struct Tag_name
{
    datatype member1;
    datatype member2;      > structure member.
    :
};
```

→ where struct is the keyword which tells the compiler that a structure is being defined.
→ Tag_name is a name of the structure
→ member1, member2,.... membern are called members of the structure.
→ The members are declared with in the culary braces.
→ The closing brace must end with semicolon(;)

ex:- struct student
{
char sname[20];
char rno[15];
int m1, m2, m3;
float avg;
char grade;
};

2. Type definition structure.
Syntax:-

typedef struct
{
datatype member1;
datatype member2;
:
:
} Type name;

→ where typedef is a keyword added to the beginning of defmition.
→ Struct is the keyword which tells the complier that a structure is being defined.
→ member1, member2, .. membern are called members of the structure.
→ The members are declared with in the curuly braces.
→ The closing bracee must end with type defmition name which mturn ends with semicolon (;)

Ex:-
```
typedef struct
{
    char sname[20];
    char rno[15];
    int m1, m2, m3;
    float avg;
    char grade;
} student;
```

Note:- All variables (members) of the structure are declared with in the structure.
→ During the declaration of structure the memory is not allocated to the structure.

Structure variable declaration:-

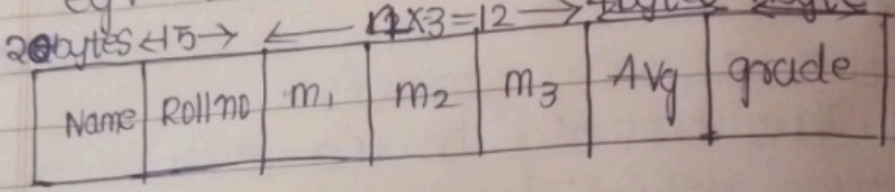
→ when we declare a variable to the

structure then only the memory will be allocated to the structure.

Tagged Structure :-

Syntax :- struct tagname var1, var2;

Eg :- struct student s1, s2;



$S_1 = 20+15+12+4+1$

$S_1 = 52$ bytes.

Type defmition Structure :-

type name var1, var2, ... ;

student s1, s2;

same as tagged = 2 bytes.

Initialization :-

```
struct student
{
    char name[20];
    char rNO[15];
    int m1, m2, m3;
    float avg;
    char grade;
};
```

s={'xxx',"22R1A0549",25,26,27,28,'A'};

Accessing the memory of Structure:-

2 operators.

1. Dot operator: Structure variable is a normal variable
2. Arrow operator: Structure variable is a pointer variable.

Normal variable → Structure variable → Structure variable.

pointer variable → Structure variables → Structure number.

Ex: struct sample
{ int x;
float y;
char c;
};
struct sample s, *s1;
s.x (s1→x)
s.y (s1→y)
s.c (s1→c)

(10) write a c program to read and print Student details as Student name.

#include<stdio.h>
Struct student

```
{
  char sname[20];
  char rno[15];
  int m1, m2, m3, total;
  float avg;
  char grade;
} s;
int main()
{
printf("enter student name, roll no,
      3 subject marks);
scanf("%s %s %d %d %d", s.sname,
s.r.no, &s.m1, &sm2, &sm3);
s.total = s.m1 + s.m2 + s.m3;
s.avg = s.total/3.0;
if (avg >= 75)
   s.grade = 'A';
else
   s.grade = 'B';
printf("Student name = %s\n", s.name);
printf("Roll no = %s\n", s.Rno);
printf("Total = %d\n", s.total);
printf("Average = %f\n", s.avg);
printf("Grade = %c\n", s.grade);
}
```

Q. Write C program to read employee details as employee name, employee Id, employee basic salary and

print name, Id, and gross salary.
BS, HRA, DA, PF.
S = employee.details.

```
# include <Stdio.h>
struct employee
{
   char ename[20];
   char eId[20];
(X)  int BC, HRA, DA, PF, gross salaray;
} S;
int main()
{
printf("enter employee name, Id, Basic
   Salary, HRA, DA, PF");
Scanf("%s%s%d%d%d", ename,
 eId, & eB.C, & eHRA, & ePF, & eDA);
e.gross salary = BC+DA+HRA-PF;
printf("employee name = %s\n", ename
printf("employee Id = %s\n", eId);
printf("gross salary = %d\n", egross
                salary);
}

# include < stdio.h >
struct employee
{
   char ename[20];
   char eId[20];
```

```
float DA, HRA, gross salary;
int PF, BC;
} e;
int main()
{
  printf("enter employee name, Id, BC, PF");
  Scanf("%s%s%d%d", e.name, e.Id,
        &e.BC &e.PF);
  DA = 0.1 * BC;
  HRA = 5/100 * BC;

  gross salary = BC+DA+HRA-PF;
  printf("employee name = %s\n",
    employee Id = %s\n, gross salary = %f\n;
  e.name, e.Id, e.gross salary);
}
```

Nested structure :-

→ One structure present inside the another structure is called as nested structure.

→ Any number of structure can be nested but not more than three.

Syntax :-

```
struct Tag-name1
{

};
```

```
float DA, HRA, gross salary;
int PF, BC;
} E;
int main()
{
 printf("enter employee name, Id, BC, PF");
 Scanf("%s%s%d%d", e.name, e.Id,
      &e.BC &e.PF);
 DA = 0.1 * BC;
 HRA = 5/100 * BC;

 gross salary = BC+DA+HRA-PF;
 printf("employee name = %s\n",
  employee Id = %s\n, gross salary = %f\n",
 e.name, e.Id, e.gross salary);
}
```

Nested structure:-

→ one structure present inside the another structure is called as nested structure.

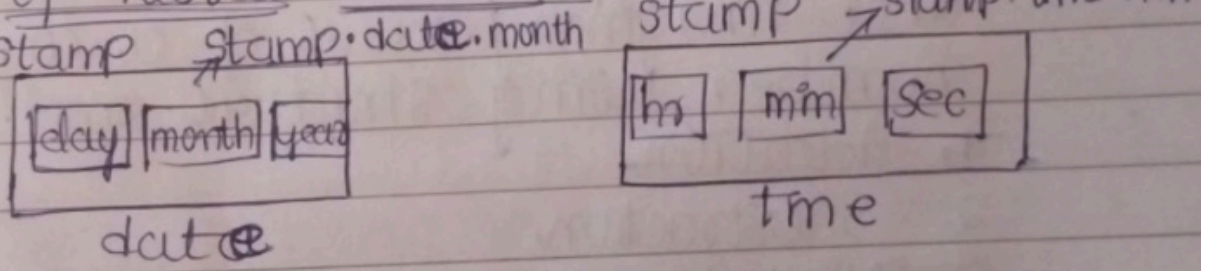
→ Any number of structure can be nested but not more than three.

Syntax:-

```
Struct Tag-name1
{

};
```

```
struct tag-name2
{

};
struct tag-name3
{
  Struct Tag-name1;
  Struct Tag-name2;
};
Struct Tag-name 3 variable;
```

Declaration, initialization and accessing of Nested Structures:-




```
# include<stdio.h>
struct date
{
   int day, month, year;
};
struct time
{
  int hr, min, sec;
};
struct stamp
{
```

```
    struct date d;
    struct time t;
} s;
int main()
{
    struct stamps s = {{7, 11, 2023}, {10, 25, 30}};
printf("Date = %d-%d-%d", s.d.day,
        s.d.month, s.d.year);
printf("time = %d:%d:%d", s.t.hr,
        s.t.min, s.t.sec);
}
```

(Q) write a c program to read real and imaginary values of a complex number using structure and find
1. Addition
2. Substraction
3. Multiplication.

```
struct complex
{
    int real, img;
} c1, c2, c3;
int main()
{
printf("enter complex number 1:");
scanf("%d%d", &c1.real, &c1.img);
printf("enter complex number 2:");
scanf("%d%d", &c2.real, &c2.img);
```

```
c3.real = c1.real + c2.real;
c3.img = c1.img + c2.img;
printf("Addition of 2 complex numbers=
%d+i %d\n", c3.real, c3.img);
printf("Substraction of 2 complex numbers=
%d-i%d\n", c3.real, c3.img);

c3.real = (c1.real * c2.real) - (c1.img *
          c2.img);
c3.img = (c1.real * c2.img) + (c1.img * c2.real);
```

Array of structures :-

If a structure variable is declared as an Array then it is known as array of structures

Syntax for Array of structure declaration.

```
struct tag_name array of structure [size];
                (or)
Structure variables.
```

Ex:- Let us take an example to store the information of 3 students we can have the following structure defination and declaration,

```
Struct Student
{
  char name[10];
   int rno;
   float avg;
};
Struct Student S[3];
```

or

```
} S[3];
```

Initialization of Array of Structures:-

```
Struct Student S[3] = {{"ABC", 1, 56.7},
{"xyz", 2, 65.8}, {"pqr", 3, 82.4}};
```

(9) write a c program to store name, roll number, year and marks of three subjects of n students and print the student name roll number grade average, and grade based on the marks of average of the student using structures.

```
#include <stdio.h>
Struct Student
{
   char name[100];
   char rno[100];
    int y, m1, m2, m3;
} S1[100];
int main()
{  int n, i;
   float avg;
   char g;
printf("Enter n value:");
Scanf("%d", &n);
printf(" Enter roll no, name, year, m1, m2,
                    m3 ;");
```

```
for(i=0; i<n; i++)
scanf("%s%s%d%d%d%d%d", s1[i].rno,
s1[i].name, &s1[i].y, &s1[i].m1, &s1[i].m2,
&s1[i].m3);
for(i=0; i<n; i++)
{ avg=(s1[i].m1+s1[i].m2+s1[i].m3)/3;
  if(avg>=75)
    g='A';
  else if(avg>=50)
    g='B';
  else
    g='C';
printf("\n Roll no=%s\n name=%s\n Average
=%f\n Grade=%c\n", s1[i].rno, s1[i].name,
avg, g);
}
};
```

8. Write a c programe to read employee details as employee no, elpoyname, Basis salary, HRA, DA of n employee using structures and print employee no, employee name and gross salary of n employee.

```
#include<stdio.h>
struct employee
{   char name[100];
    char no[100];
    int BS, HRA, DA;
} e[100];
int main()
```

```
{ int n, i;
int Gross salary;
printf("enter n value");
Scanf("%d", &n);
printf("enter name, no, BS, HRA, DA");
for(i=0; i<n; i++)
scanf("%s%s%d%d%d", &E[i].name,
E[i].no, &E[i].BS, &E[i].HRA, &E[i].DA);
for(i=0; i<n; i++)
{ Gross salary = BS+HRA+DA;
printf("\n Eno=%os, E.name=%os,
Gross salary = %d",
E[i].no; E.[i].name,
Gross salary);
gross salary= e[i].bs+ e[i].hra+e[i].da;
```

Structures and functions :-

→ Structures are more useful if we are able to pass them to the functions and return them.

→ The structures members can be passed to the function in various ways as shown below.

Call by value
1. By passing individual members of structure
2. By passing the whole structure

Call by reference
3. By passing structure through pointers

passing individual members of structure:

actual parameters - members of the structure
formal parameters - normal variables like int, float.

Q. write a c program to read numerator and denominator for a fraction using structures and find multiplication of two fractions by passing individual members of structure.

```
# include<stdio.h>
int multiply(int x, int y);
struct fraction
{
  int n,d;
} f1,f2,f3;
void main()
{ printf("enter the fraction\n");
scanf("%d%d",&f1.n,&f1.d);
printf("enter the 2 fraction 2\n");
scanf("%d%d",&f2.n,&f2.d);
f3.n=multiply(f1.n,f2.n);
f3.d=multiply(f1.d,f2.d);
printf("result=%d%d\n",f3.n,f3.d);
}
int multiply(int x, int y)
{
 return(x*y);
}
```

2. PASS THE WHOLE STRUCTURE:

→ Actual Arguments are the names of the structure variables.
→ The formal Arguments are structure variables with specification of structure type.

```
#include<stdio.h>
Struct fraction multiply(Struct fraction f1,
        Struct fraction f2);
Struct fraction
{
 int n,d;
} f1,f2,f3;
void main()
{ printf("Enter the fraction\n");
 Scanf("%d%d",&f1.n,&f1.d);
printf("Enter the 2 fraction\n");
 Scanf("%d%d",&f2.n,&f2.d);
f3 = multiply (f1,f2);
printf("Result = %d/%d \n", f3.n,f3.d);
}
Struct fraction multiply (Struct fraction f1,
  Struct fraction f2)
{
 Struct fraction res;
 res.n = f1.n * f2.n
 res.d = f1.d * f2.d;
  return res;
}
```

3. passing structure through pointers:-

→ Actual parameters → & structure variable
Address of structure variables

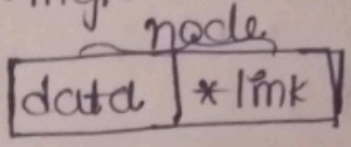
→ Formal parameters → structure variables are declared as pointer variables.

```
#include<stdio.h>
struct fraction multiply(struct fraction *f1,
  struct fraction * f2);
{
  int n,d;
}
void main()
{
  struct fraction *f1,*f2,*f3,a,b,c;
  f1 = &a;
  f2 = &b;
  f3 = &c;
printf("enter the fraction numerator\n");
Scanf("%d", &f1->n);
printf("enter the fraction denomenator\n");
scanf("%d", &f1->d);
printf("enter the 2nd fraction numerator\n");
scanf("%d", &f2->n);
printf("enter the 2nd fraction denomenator");
Scanf("%d", &f2->d);
*f3 = multiply(f1,f2);
printf("result = %d/%d\n", f3->n, f3->d);
}
```

```
struct fraction multiply (struct fraction
* f1, struct fraction * f2)
{
  struct fraction * res, d;
  res = &d;
  res->n = f1->n * f2->n;
  res->d = f1->d * f2->d;
  return * res;
}
```

Self referential functions :- Structures

→ A structure which refers to itself is known as self referential structure.
→ It is mainly used in implementation of data structures.
→ Example : to create a single linked list.

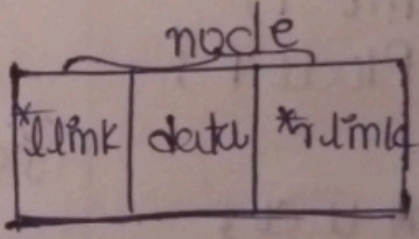



```
struct node
{ int data;
  struct node *link; → self-referential
}
```

→ In the above example node is pointing to node again until link assigned with null.

example :- to create a node in double linked list

```
struct node
{ struct node *llink;
  int data;
  struct node *rlink;
}
```



In the above example node is pointing to node again until link assigned with null.

UNION :-

→ Union is a derived datatype.
→ Union is a collection of dissimilar dataitems stored in continuous memory locations under a single name.

Note :- The syntax declaration and use of Union is similar to the structure but

The functionality is different.

Syntax:-

```
union tag-name
{ type1 member;
  type2 member;
  ....
};
```

Variable declaration is similary like structure.

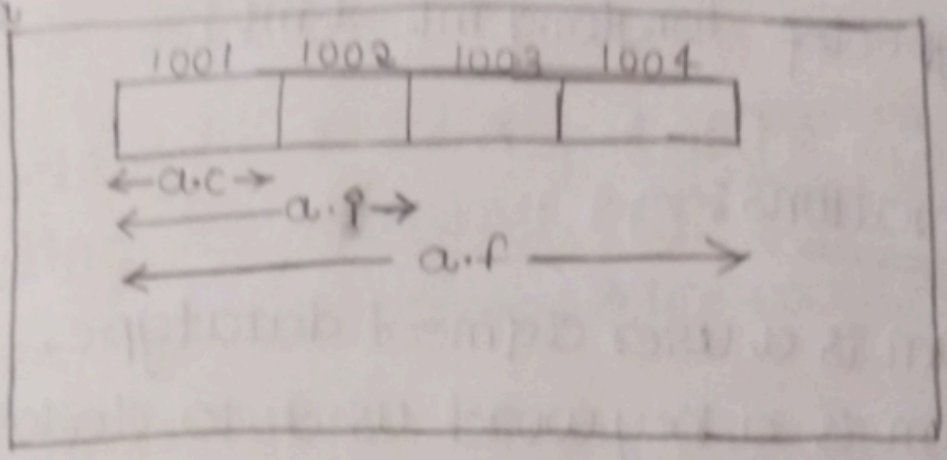
Syntax:- union tag-name var1, var2...;

union u { char c; int i; float f; }; union u a; tagged union	union u { char c; int i; float f; }a; variable union	typedef union { char c; int i; float f; }U; Ua; typeclef union.

→ we can access various members of the union as mentioned below.

a.c → 1 byte
a.i → 2 byte
a.f → 4 byte

for string (s) → 20 byte.




```
#include <stdio.h>
union student
{ char name[20];
  int marks;
}u;
void main()
{ printf("enter name\n");
  scanf("%s", u.name);
  printf("name=%s", u.name);
  printf("enter marks\n");
  scanf("%d", &u.marks);
  printf("Marks=%d\n", u.marks);
}
```

Typedef :-

→ Type def is used to redefine the name of an existing variable type

Syntax :- typedef data-type identifier;

(or)

typedef existing datatype newdatatype

Ex:- typedef longlong int dint;

Enumerations:-

→ Enum is a user defined datatype.
→ Enum is a keyword used to declare new enumeration types.
→ It is mainly used to assign names to the integral constants (sequence of digits without a decimal point).

Syntax:- enum identifier{const1, const2, ...};

enum - is the keyword
identifier - name of enumeration.

By default, the values of the constants are as follows :-

const1 = 0, const2 = 0

```
#include<stdio.h>
enum weekday{sun, mon, tue, wed, thr,
             fri, sat};
             0   1   2   3   4   5   6
void main()
{
  enum weekday today;
  today = sat;
  printf("today = %d\n", today);
}
```

Output :- 6

Structures,Unions and Files

STRUCTURES:

It is a derived data type. It is a collection of elements of heterogenous data type(different).

Definition: It is heterogeneous collection of elements of different data elements under a single name.

Declaration:

Two types

We can declare structure in two ways:

- **Tagged structure**
- **Typedef structure**

Tagged structure:

It starts with a keyword 'struct'

Syntax:

```
struct structurename
{
datatype  var1;
datatype  var2;//structure mem
…….;
};
```

Eg: struct student

```
{
char name[50]; int rollno;
char gender; char grade; float m1,m2,m3; int age;
};
```

Declaration of structure Variable:

The Variables of the structure can be declared inside the main function or after structure declaration.

Syntax:

```
struct structurename
{
datatype Variable1;
 datatype Variable2;
…..
……
};
int main()
{
struct structurename Variables;//Variable
}
```

Example:

```
struct student
{
char name[50]; int age;
float per;
};
int main()
{
struct student s1;
}
```

Accessing the members of structures:

The members of structures can be accessed by the help of dot operator.
This dot operator is also called as member operator.

Note:

If the structure variables are pointers then the structure members can be accessed by arrow operator.(->) It is also called as indirect selection operator.

Ex:

s1.name; s1.age; s1.per;

```
//write a c program to read and print the details of a single student using
structures.#include<stdio.h>
 #include<stdlib.h>
struct student
{
char name[30];
 int rollno;
int m1,m2,m3; float avg;
};
int main()
{
struct student s1; printf("Enter details:\n");
scanf("%s%d%d%d%d",s1.name,&s1.rollno,&s1.m1,&s1.m2,&s1.m3);
s1.avg=(s1.m1+s1.m2+s1.m3)/3;
printf("%s\n%d\n%f\n",s1.name,s1.rollno,s1.avg);
}Output:
Bidya 104
95
96
98
95.00000
Bidya
104
95.00000
```

```
//write a c program to read and print the details of two students using structures.
#include<stdio.h>
#include<stdlib.h>
struct student
{
char name[50]; int age;
int rollno;
}s1,s2;
int main()
{
printf("enter student1 details:");
scanf("%s%d%d",s1.name,&s1.age,&s1.rollno);
printf("enter student2 details:");
scanf("%s%d%d",s2.name,&s2.age,&s2.rollno);
printf("%s %d %d\n",s1.name,s1.age,s1.rollno);
printf("%s %d %d\n",s2.name,s2.age,s2.rollno);
}
Output:
Enter student1 details:
Bidya 20
104
Pinky 20
110
Bidya 20 104
Pinky 20 110
```

```
//write a c program to read the details of n students(name,rollno,three sub marks) and calculate avg and grade of each student#include<stdio.h>
#include<stdlib.h>
struct student
{
char name[50]; int rollno;
int m1,m2,m3; float avg; char grade;
};
struct student s[100]; int main()
{
int n,i;
printf("enter no of students:"); scanf("%d",&n);
for(i=1;i<n;i++)
{
scanf("%s%d%d%d%d",s[i].name,&s[i].rollno,&s[i].m1,&s[i].m2,&s[i].m3);
s[i].avg=(float)(s[i].m1+s[i].m2+s[i].m3)/3;
if(s[i].avg>=75)
s[i].grade='A';
else if(s[i].avg>=50)
 s[i].grade='B';
else s[i].grade='C';
}
for(i=1;i<n;i++)
{
printf("name=%s\n",s[i].name);
printf("rollno=%d",s[i].rollno);
printf("average=%f\n",s[i].avg);
printf("grade=%c\n",s[i].grade);
}
}
```

Intialization of structure:

The structure can be initialized in two ways:

- **static**
- **Dynamic**

i) static:

```
struct structurename
{
//member declaration
};
struct structurename variable={value};

example:
struct emp
{
char name[50]; int id;
float salary;
};
struct emp e1;
int main()
{
emp e1={“raju”,”104”,54000};
printf(“%s%d%f”,e1.name,e1.id,e1.salary);
}
```

2) type def structure:

We can also declare a structure by using typedef keyword. It is two ways different from tagged structure.

We use type def as keyword as the beginning of the structure. The structure is given at the end of closing bracket.

Syntax:

```
typedef   struct
{
datatype  var1;
datatype var2;
datatype var3;
…..
} structurename;
```

Eg:

```
typedef struct
{
char name[80]; float m1,m2,m3; int age;
char gender; char grade;
}student;
```

Variable declaration in type def:

Syntax:

typedef struct structurename structure Variables;

example:

typedef struct emp e1,e2,e3;

Structure Variable declaration:

1.Tagged structure

Syntax:

```
struct structurename
{
datatype  Var1;
datatype  Var2;//structure mem
…….;
} structure Varibles;
```

(or)

struct structurename Varaibles;

Eg:

```
struct student
{
char a[50]; int rollno; char gender; char grade;
float m1,m2,m3; int age;
} s1,s2,s3; Sizeof(s1)=50+4+1+1+12+4=72
Sizeof(s2)=72 Sizeof(s3)=72
```

Typedef structure Syntax:

```
typedef     struct
{
datatype  var1;
datatype var2;
 datatype var3;
…..

} structurename; structure name variables;
```

Eg:

```
typedef struct
{
char a[80]; flaot m1,m2,m3; int age;
char gender; char grade;
}student; student s1,s2,s3;
```

àDuring structure declaration memory is not allocated to the structure

-->the memory allocated to structure after structure declaration

Accessing the members of the structure

Two operators are used to access the members of the structure

.(dot) operator(or) direct selection

->(arrow)operator (or) indirect selection operator

Syntax:

structureVaraible. sturucturemember

StructureVaraible-> structuremember

Eg:

```
#include<stdio.h>
struct student
{
char name[90];
int age;
int rollno;
};struct student s1,s2;
int main()
{
printf("enter student1 details");
scanf("%s%d%d",s1.name,&s1.age,&s1.rollno);
 printf("enter student2 details");
scanf("%s%d%d",s2.name,&s2.age,&s2.rollno);
printf("%s %d %d\n",s1.name,s1.age,s1.rollno);
 printf("%s %d %d\n",s2.name,s2.age,s2.rollno);
}
```

Output:

Important points on structures:

- **structure is a collection of heterogeneous collection of elements under a single name.**
- **It holds related information about entity structure name.**
- **structure is a user defined datatype.**
- **structure members are also called as attributes or data fields.**
- **The main difference between array and structure is array holds the data of similar data types.**
- **All the members of the structure are declared under a single name called structure name or entity.**
- **The members of the structure accessed by using dot operator.**
- **The name of the structure and structure member name should not be same.**
- **memory is allocated for structures when we declare the structure Variables.**

Structure initialization usind dot() operator:

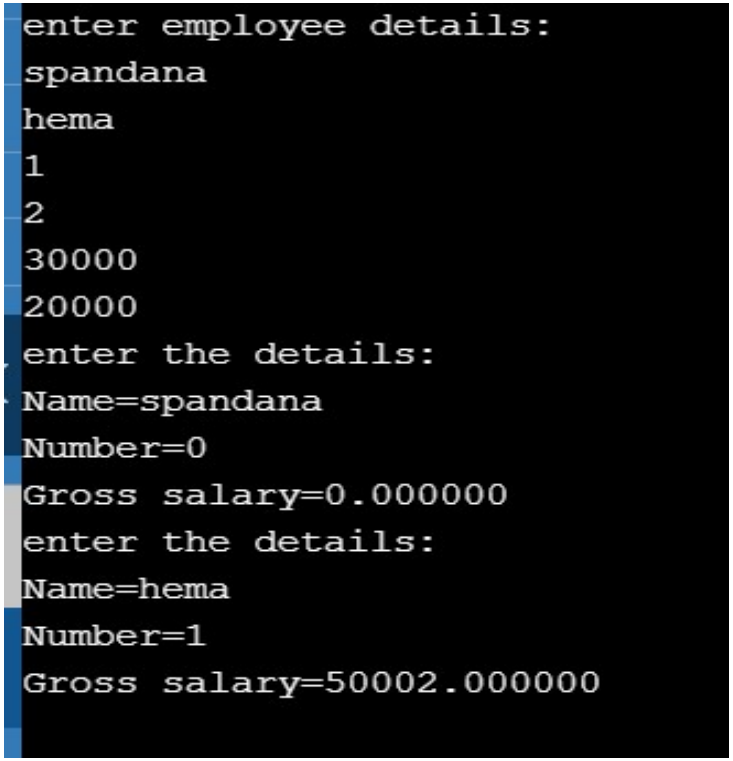
Example:

```
#include<stdio.h>
#include<stdlib.h>
 struct EMP
{
char name[50];
int emid;
float salary;
float exp;
};
struct EMP e1,e2,e3;//global declaration
 int main()
{
struct EMP e1={"aaaa",123,55.456,6.7};
struct EMP e2={"bbbb",124,78000,8.9};
printf("%s %d %f %f\n",e1.name,e1.emid,e1.salary,e1.exp);
printf("%s %d %f %f",e2.name,e2.emid,e2.salary,e1.exp);
 }
```

```
aaaa 123 55.456001 6.700000
bbbb 124 78000.000000 6.700000
```

//Write a c program to read the details of employee(emp no,name,basic salary,HRA,DA) of ‘n’ employees. Calculate the gross salary of each employee and print emp name,no,gross salary.(codetantra)

```
#include<stdio.h>
#include<stdlib.h>
struct employee
{
char name[100];
int no;
float BS,HRA,DA,GS;
};struct employee E[100];
int main()
{
int n,i;
printf("enter no of employees:");
scanf("%d",&n);
printf("enter employee details:\n");
for(i=1;i<=n;i++)
{
scanf("%s%d%f%f%f",E[i].name,&E[i].no,&E[i].BS,&E[i].HRA,&E[i].DA);
E[i].GS=E[i].BS+E[i].HRA+E[i].DA;
}
for(i=1;i<=n;i++)
{
printf(“enter the details:\n”);
printf("Name=%s\n",E[i].name); printf("Number=%d\n",E[i].no);
 printf("Gross salary=%f\n",E[i].GS);
}}
```



```
//program on structures using -> arrow operator
#include <stdio.h>
#include<stdlib.h>
struct student
{
char name[20];
int number;
int rank;
};
int main()
{
struct student s1;
struct student *s2;
s2=&s1;
scanf("%s",s2->name);
 scanf("%d",&s2->number);
scanf("%d",&s2->rank);
printf("NAME:%s\n",s2->name);
printf("NUMBER:%d\n",s2->number);
printf("RANK:%d\n",s2->rank);
return 0;
}
```

Output:

```
S 1620
1
NAME=spa NUMBER=1620 RANK=1
```

```
//program on structures using -> arrow operator
#include <stdio.h>
#include<stdlib.h>
struct student
{
char name[20];
int number;
int rank;
};
int main()
{
struct student s1;
struct student *s2;
s2=&s1;
scanf("%s",s2->name);
 scanf("%d",&s2->number);
scanf("%d",&s2->rank);
printf("NAME:%s\n",s2->name);
printf("NUMBER:%d\n",s2->number);
printf("RANK:%d\n",s2->rank);
return 0;
}
```

Output:

```
S 1620
1
NAME=spa NUMBER=1620 RANK=1
```

Complex structures:

arrays,pointers,functions are called complex structures

1.nested structure

structure within another structure is called nested.

Declaration:

tagged nested structure

```
struct outer_structurename
{
datatype outer_mem1;
 datatype outer_mem2;
 struct inner_structurename
{
datatype inner_mem1;
datatype inner_mem2;
}inner structure_varibles;
}outer_structreVaraible;
```

Accesssing:

```
outer_structureVarible.outer_mem1;
outer_structureVarible.outer_mem2;
outer_structureVariable.inner_structureVariable.inner_mem1;
outer_structureVariable.inner_structureVariable.inner_mem2;
```

eg:

```
#include<stdio.h>
 #include<stdlib.h>
struct ABC//outer structure
{
int x;
 int y;
struct PQR//inner structure
{
int w;
 int v;
}E;
}S;
int main()
{
printf("enter outer");
 scanf("%d%d",&S.x,&S.y);
printf("enter inner");
 scanf("%d%d",&S.E.w,&S.E.v);
printf("%d %d\n",S.x,S.y);
printf("%d %d\n",S.E.w,S.E.v);
}
```

```
enter outer2
3
enter inner4
5
2 3
4 5
```

typedef nested structure

```
typedef struct
{
datatype outer_mem1;
datatype outer_mem2;
typrdef struct inner_structurename
{
datatype inner_mem1;
datatype inner_mem2;
} inner_structurename ;
} outer_structurename;
```

```
outer_structurename outer_structreVaraible;
 inner_structurename  inner structure_Varibles;
```

Accesssing:

```
outer_structureVarible.outer_mem1;
outer_structureVarible.outer_mem2;
outer_structureVariable.inner_structureVariable.inner_mem1;
outer_structureVariable.inner_structureVariable.inner_mem2;
```

Array within a structure(or)structure containg arrays

*structure members are of arrays(example)

```
struct student
{
char name[50];//structure member is array
 int marks[5];
char subjects[4][50];
float avg;
};
struct student S={"AAA",{12,13,14,15},{Ch,Phy,Math,PPS},11};
S.marks[0],S.marks[1]
```

Array of structure

structure varible is of array type

eg:

```
struct student
{
char name[50]; int marks[5]; char rollno[4]; float avg;
}s[50];//structure varible is  array
struct student s={{"AAA",{12,13,14,15},{Ch,Phy,Math,PPS},11},
{"BBB",{10,11,12,13},{Ch,Phy,Math,PPS},12,
{"CCC",{10,10,11,14},{Ch,Phy,Math,PPS},10}};
/**student details**/
/**employee details**/
```

Structures and functions

call by Value

call by reference

A structure can be passed to a function in three ways

1. Passing indiVidual members of structure as parameter

Passing address indiVidual members of structure as parameter

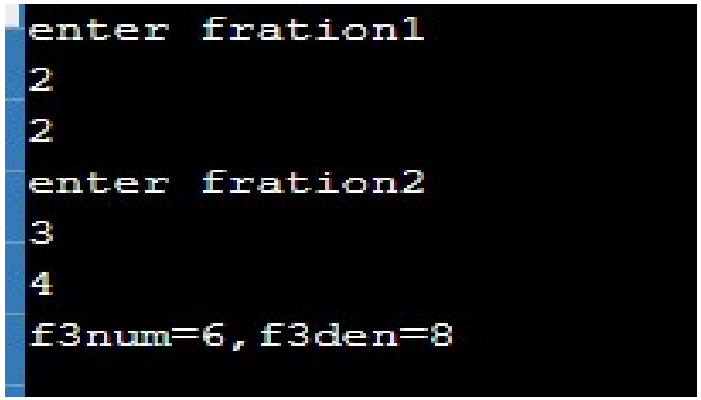
2.Passing whole structure as parameter

Passing address of structure as parameter

1)Passing indiVidual members of structure as parameter

/*multiply two fractional numbers*/

```
#include<stdio.h>
#include<stdlib.h>
 struct fraction
{
int num,den;
};struct fraction f1,f2,f3;
int mul(int ,int);//function declaration
int main()
{
printf("enter fration1");
scanf("%d%d",&f1.num,&f1.den);
printf("enter fration2");
scanf("%d%d",&f2.num,&f2.den);
f3.num=mul(f1.num,f2.num);//actual
f3.den=mul(f1.den,f2.den);//function call
printf("f3num=%d,f3den=%d",f3.num,f3.den);
}
int mul(int x,int y)//function definition
{
return (x*y);
}
```



Passing address individual members of structure as parameter

```
#include<stdio.h>
#include<stdlib.h>
struct fraction
{
int num,den;
};
struct fraction f1,f2,f3;
int mul(int *x,int *y);//function declaration
 int main()
{
printf("enter fration1");
scanf("%d%d",&f1.num,&f1.den);
printf("enter fration2");
 scanf("%d%d",&f2.num,&f2.den);
f3.num=mul(&f1.num,&f2.num);//actual
 f3.den=mul(&f1.den,&f2.den);//function call
printf("f3num=%d,f3den=%d",f3.num,f3.den);
}
int mul(int *x,int *y)//function definition
{
return (*x)*(*y);
}
```

Write a C program to read real and imaginary parts of a complex number using structures and perform the following operations on complex numbers.

Add two complex numbers.
Multiply two complex numbers.
Subtract two complex numbers.

```
#include <stdio.h>
#include<stdlib.h>
 struct complex
{
int real, img;
};
struct complex a, b, c;

Void add(struct complex a,struct complex b)
{
c.real = a.real + b.real;
c.img = a.img + b.img;
printf("Addition = %d + i %d\n",c.real,c.img);
}

Void sub(struct complex a,struct complex b)
{
c.real = a.real - b.real; c.img = a.img - b.img;
printf("Subtraction = %d + i %d\n",c.real,c.img);
}
```

```
void mul(struct complex a,struct complex b)
{
c.real = a.real*b.real - a.img*b.img;
c.img = a.img*b.real + a.real*b.img;
printf("Multiplication = %d + i %d\n",c.real,c.img);
}

int main()
{
printf("Enter complex1 : ");
scanf("%d%d", &a.real,&a.img);
printf("Enter complex2 : ");
scanf("%d%d", &b.real,&b.img);
add(a,b);
sub(a,b);
mul(a,b);
 return 0;
}
```

Write a C program to read time in hours, minutes, seconds using structures and perform the following operations on time.
Addition of two time periods.
Subtraction of two time periods.

```
#include<stdio.h>
 #include<stdlib.h>
 struct time
{
int hr,min,sec;
};
struct time t1,t2,t3;

Void add()
{
t3.min = t1.min+t2.min;
t3.sec = t1.sec+t2.sec;
t3.hr = t1.hr+t2.hr;
while(t3.sec>=60)
{
t3.min++;
t3.sec = t3.sec-60;
}
while(t3.min>=60)
{
t3.hr++;
t3.min = t3.min-60;
}
printf("Addition = %d:%d:%d\n",t3.hr,t3.min,t3.sec);
}
```

```
void sub()
{
if(t1.sec<t2.sec)
{
t1.min--;
t1.sec = t1.sec+60;
}
t3.sec = t1.sec-t2.sec;
if(t1.min<t2.min)
{
t1.hr--;
t1.min = t1.min+60;
}
t3.min = t1.min-t2.min;
t3.hr = t1.hr-t2.hr;
printf("Subtraction = %d:%d:%d\n",t3.hr,t3.min,t3.sec);
}
int main()
{
printf("Enter time1 : ");
scanf("%d%d%d",&t1.hr,&t1.min,&t1.sec);
printf("Enter time2 : ");
scanf("%d%d%d",&t2.hr,&t2.min,&t2.sec);
add();
sub();
 return 0;
}
```

Passing whole structure as parameter

```
#include<stdio.h>
#include<stdlib.h>//header file for stryctures
struct fraction
{
int num,den;
};
struct fraction f1,f2,f3;
struct fraction mul(struct fraction f1,struct fraction f2);//function declaration
int main()
{
printf("enter fration1");
scanf("%d%d",&f1.num,&f1.den);//(1,2)
printf("enter fration2");
scanf("%d%d",&f2.num,&f2.den);//(3,4)
f3=mul(f1,f2);(1,2,3,4);//function call
printf("F3num=%d,F3den=%d",f3.num,f3.den);
}
struct fraction mul(struct fraction f1,struct fraction f2)//function definition
{
struct fraction res;
res.num=f1.num * f2.num;
res.den=f1.den * f2.den;
return res;
}
```

Passing address of whole structure as parameter

```
#include<stdio.h>
struct fraction
{
int num,den;
};
struct fraction *f1,*f2,*f3;
struct fraction mul(struct fraction *f1,struct fraction *f2);
 int main()
{
printf("enter fration1");
scanf("%d%d",f1->num,f1->den);
printf("enter fration2");
scanf("%d%d",f2->num,f2->den);
*f3=mul(f1,f2);
printf("F3num=%d,F3den=%d",f3->num,f3->den);
}
struct fraction mul(struct fraction *f1,struct fraction *f2)
{
struct fraction *res;
 res.num=f1->num * f2->num;
 res.den=f1->den * f2->den;
 return res;
 }
```

UNIONS:

it is also heterogeneous collection of data elements declaration:

```
union union_name
{
//union members;
datatype mem1;
 datatype mem2;
 datatype mem3;
.....
};
union union_name Varaible;//union Varaible declaration
```

*only difference b/w structure and union is in terms of memory allocation

*memory associated with structure is sum of datatypes of all datamembers

*memory associated with union is highest data member memory

*members of union can accessed by using dot operator

eg:

```
 #include<stdio.h>
 #include<stdlib.h>
union student
{
char name[70];
int age;
char gender;
char grade;
 float m1,m2,m3;
};union student s1;//70+4+1+1+12=88(structure)//union =70
int main(){
printf("enter name"); scanf("%s",s1.name);
printf("%s",s1.name);
 printf("enter age"); scanf("%d",&s1.age);
printf("%d",s1.age);
 printf("enter gender"); scanf("%c",&s1.gender);
printf("%c",s1.gender);
}
```

self referential structure

A structure referring itself is called as self referential structure. it is mainly used in Data structure

syntax:

```
struct structure_name
{
struct structure_name *pointer
 };
```

eg:

```
struct node
{
int data;
struct node *link;
};
```

Representation of node single linked list

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

Representation of node double linked list

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

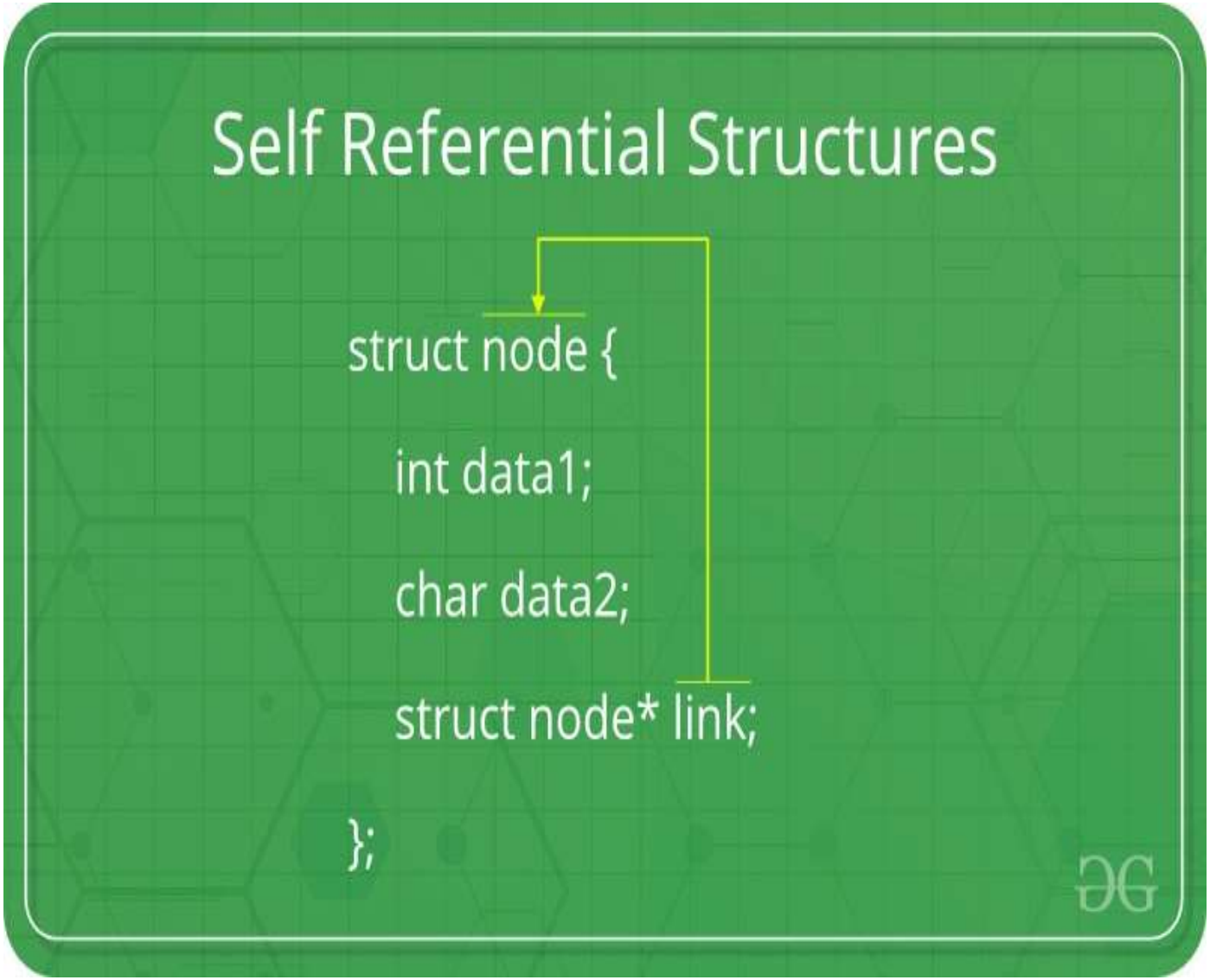
Self Referential Structures

Self Referential structures are those structures that have one or more pointers which point to the same type of structure, as their member.
In other words, structures pointing to the same type of structures are self- referential in nature.

Example:

```
filter_none brightness_4
struct node
{
int data1;
char data2;
struct node* link;
};
 int main()
{
struct node ob;
return 0;
}
```

In the above example ‘link’ is a pointer to a structure of type ‘node’. Hence, the structure ‘node’ is a self-referential structure with ‘link’ as the referencing pointer.
An important point to consider is that the pointer should be initialized properly before accessing, as by default it contains garbage value.

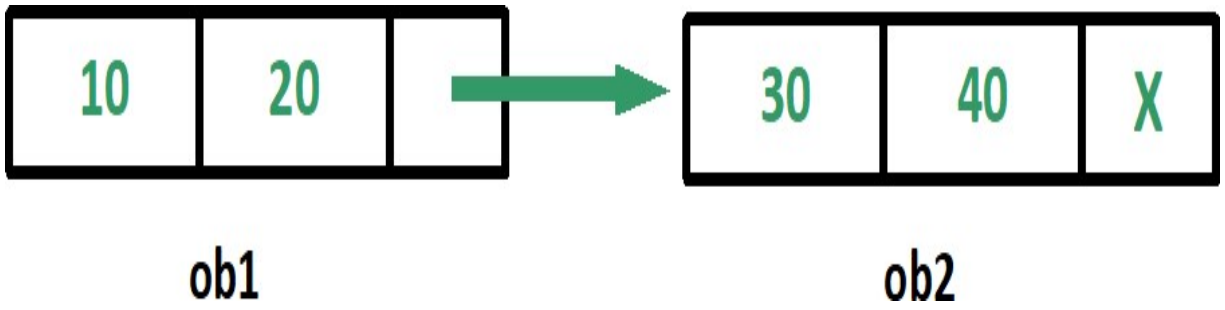

Self Referential Structures
struct node {
int data1;
char data2;
struct node* link;
};

Types of Self Referential Structures:

- **Self Referential Structure with Single Link**
- **Self Referential Structure with Multiple Links**

Self Referential Structure with Single Link:

These structures can have only one self-pointer as their member. The following example will show us how to connect the objects of a self-referential structure with the single link and access the corresponding data members. The connection formed is shown in the following figure.

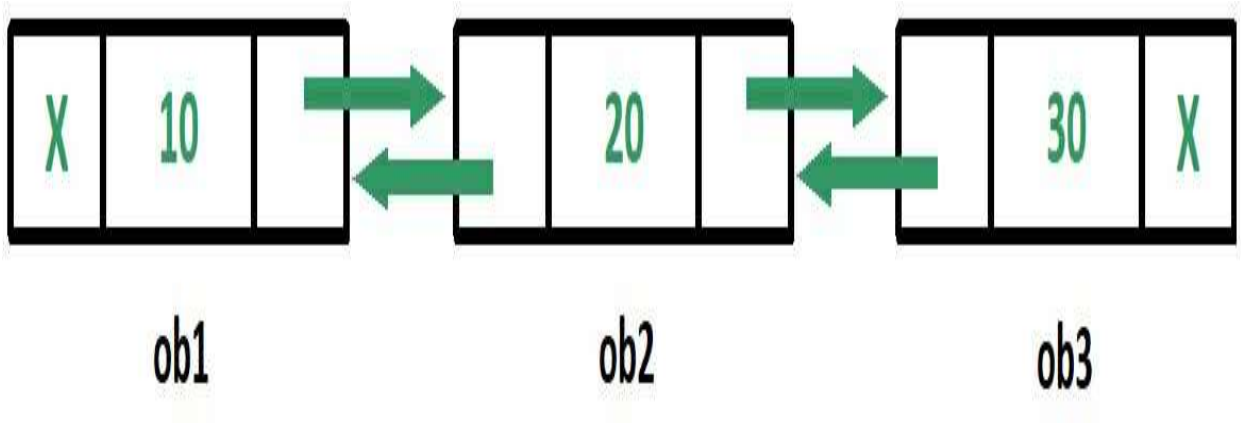


```
filter_none edit play_arrow brightness_4
#include <stdio.h>

struct node { int data1; char data2;
struct node* link;
};

int main()
{

struct node ob1; // Node1

// Initialization ob1.link = NULL; ob1.data1 = 10;
ob1.data2 = 20;

struct node ob2; // Node2

// Initialization ob2.link = NULL; ob2.data1 = 30;
ob2.data2 = 40;

// Linking ob1 and ob2 ob1.link = &ob2;

// Accessing data members of ob2 using ob1 printf("%d", ob1.link->data1);
printf("\n%d", ob1.link->data2); return 0;
}
```

Output:

Self Referential Structure with Multiple Links: Self referential structures with multiple links can have more than one self-pointers. Many complicated data structures can be easily constructed using these structures. Such structures can easily connect to more than one nodes at a time.

The following example shows one such structure with more than one links.
The connections made in the above example can be understood using the following figure.



```
#include <stdio.h>
 struct node { int data;
struct node* prev_link;
struct node* next_link;
};
 int main()
{
struct node ob1; // Node1
 // Initialization
ob1.prev_link = NULL; o
b1.next_link = NULL;
ob1.data = 10;
 struct node ob2; // Node2
 // Initialization
ob2.prev_link = NULL;
ob2.next_link = NULL;
ob2.data = 20;
 struct node ob3; // Node3
 // Initialization
ob3.prev_link = NULL;
ob3.next_link = NULL;
ob3.data = 30;
 // Forward links
 ob1.next_link = &ob2;
ob2.next_link = &ob3;
 // Backward links
ob2.prev_link = &ob1;
ob3.prev_link = &ob2;
```

```
 // Accessing data of ob1, ob2 and ob3 by ob1
printf("%d\t", ob1.data);
printf("%d\t", ob1.next_link->data);
printf("%d\n", ob1.next_link->next_link->data);
 // Accessing data of ob1, ob2 and ob3 by ob2
printf("%d\t", ob2.prev_link->data);
printf("%d\t", ob2.data);
printf("%d\n", ob2.next_link->data);
 // Accessing data of ob1, ob2 and ob3 by ob3
printf("%d\t", ob3.prev_link->prev_link->data);
printf("%d\t", ob3.prev_link->data);
printf("%d", ob3.data);
return 0;
}
```

```
10   20   30
10   20   30
10   20   30
```

Output:

In the above example we can see that ‘ob1’, ‘ob2’ and ‘ob3’ are three objects of the self referential structure ‘node’. And they are connected

using their links in such a way that any of them can easily access each other’s data. This is the beauty of the self referential structures. The connections can be manipulated according to the requirements of the programmer.

typedef:
stands for type definition.
It is used for representing existing data types or rename the existing variables.
Syntax:
typedef datatype identifier;
ex:

```
typedef int marks;
marks sub1,sub2,sub3;
typedef float a;
a s1;
#include<stdio.h> int main()
{
typedef int marks;
marks s1,s2,s3;
printf("enter marks:");
scanf("%d%d%d",&s1,&s2,&s3);
printf("the marks are:");
printf("%d %d %d",s1,s2,s3);
}
```

Enumerated data type:
It is represented by “enum”
Syntax:

```
enum typename
{
mem1,mem2,mem3;
};
```

```
#include <stdio.h>
 enum week
 {
Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
};
 int main()
 { // creating today variable of enum week type
 enum week today;
today = Wednesday;
printf("Day %d",today+1);
return 0;
 }
```

Output

```
Day 4
```

Write a C program to store name, roll number, year and marks of three subjects of n students and print the student the name, roll number, average, and grade based on average marks of the student using structures.

```
#include<stdio.h>
struct student {
int rollno;
char name[30]; int year;
int m1,m2,m3;
};struct student s[20];
 int main()
{
int n,i; float avg; char grade;
printf("Enter no of students(<=3) : ");
scanf("%d",&n);
printf("Enter rollno,name,year,m1,m2,m3 : ");
 for(i=0; i<n; i++)
 scanf("%d%s%d%d%d%d",&s[i].rollno,s[i].name,&s[i].year,&s[i].m1,&s[i].m2, &s[i].m3);
for(i=0; i<n; i++)
{
avg = (s[i].m1+s[i].m2+s[i].m3)/3.0; if(avg>=75)
grade = 'A';
else if(avg>=50) grade = 'B';
else
grade = 'C';
printf("Rollno = %d\nName = %s\nAverage = %f\nGrade =
%c\n",s[i].rollno,s[i].name,avg,grade);
}
```

Write a C program to read real and imaginary parts of a complex number using structures and perform the following operations on complex numbers.

Add two complex numbers.
Multiply two complex numbers.
Subtract two complex numbers.

```
#include <stdio.h>
struct complex
{
int real, img;
};
struct complex a, b, c;
 Void add(struct complex a,struct complex b)
{
c.real = a.real + b.real; c.img = a.img + b.img;
printf("Addition = %d + i %d\n",c.real,c.img);
}
 Void sub(struct complex a,struct complex b)
{
c.real = a.real - b.real; c.img = a.img - b.img;
printf("Subtraction = %d + i %d\n",c.real,c.img);
}
 Void mul(struct complex a,struct complex b)
{
c.real = a.real*b.real - a.img*b.img; c.img = a.img*b.real + a.real*b.img;
printf("Multiplication = %d + i %d\n",c.real,c.img);
}
 int main()
{
printf("Enter complex1 : ");
scanf("%d%d", &a.real,&a.img); printf("Enter complex2 : ");
scanf("%d%d", &b.real,&b.img); add(a,b);
sub(a,b);
mul(a,b);
 return 0;
}
```

Structures in C

In C programming language, a structure is a collection of elements of the different data type. The structure is used to create user-defined data type in the C programming language. As the structure used to create a user-defined data type, the structure is also said to be “user-defined data type in C”.

In other words, a structure is a collection of non-homogeneous elements. Using structure we can define new data types called user-defined data types that holds multiple values of the different data type.

The formal definition of structure is as follows...

Structure is a colloction of different type of elements under a single name that acts as user defined data type in C.

Generally, structures are used to define a record in the c programming language. Structures allow us to combine elements of a different data type into a group. The elements that are defined in a structure are called members of structure.

How to create structure?

To create structure in c, we use the keyword called "***struct***".

We use the following syntax to create structures in c programming language.

```
struct <structure_name>
{
    data_type member1;
    data_type member2, member3;
    .
    .
} ;
```

Following is the example of creating a structure called Student which is used to hold student record.

Creating structure in C

```
struct Student
{
char stud_name[30];
int roll_number;
float percentage;
} ;
```

Important Points to be Remembered

Every structure must terminated with semicolon symbol (;).

"struct" is a keyword, it must be used in lowercase letters only.

Creating and Using structure variables

In a c programming language, there are two ways to create structure variables. We can create structure variable while defining the structure and we can also create after terminating structure using struct keyword.

To access members of a structure using structure variable, we use dot (.) operator.

Consider the following example code...

Creating and Using structure variables in C

```
struct Student
 {
 char stud_name[30];
int roll_number;
float percentage;
} stud_1 ; // while defining structure
void main()
{ struct Student stud_2; // using struct keyword
 printf("Enter details of stud_1 : \n");
 printf("Name : ");
scanf("%s", stud_1.stud_name);
printf("Roll Number : ");
scanf("%d", &stud_1.roll_number);
printf("Percentage : ");
scanf("%f", &stud_1.percentage);
 printf("***** Student 1 Details *****\n);
printf("Name of the Student : %s\n", stud_1.stud_name);
printf("Roll Number of the Student : %i\n", stud_1.roll_number); printf("Percentage of the Student : %f\n",
stud_1.percentage);
}
```

In the above example program, the stucture variable "**stud_1** is created while defining the structure and the variable "**stud_2** is careted using struct keyword. Whenever we access the members of a structure we use the dot (.) operator.

Memory allocation of Structure

When the structures are used in the c programming language, the memory does not allocate on defining a structure. The memory is allocated when we create the variable of a particular structure. As long as the variable of a structure is created no memory is allocated. The size of memory allocated is equal to the sum of memory required by individual members of that structure.

In the above example program, the variables stud_1 and stud_2 are allocated with 36 bytes of memory each.

```
struct Student{
   char stud_name[30];———— 30 bytes
   int roll_number;———————— 02 bytes
   float percentage;———————— 04 bytes
};                          sum = 36 bytes
```

Here the variable of **Student** structure is allocated with 36 bytes of memory.

Important Points to be Remembered

All the members of a structure can be used simultaneously.

Until variable of a structure is created no memory is allocated.

The memory required by a structure variable is sum of the memory required by individual members of that structure.

Unions in C

In C programming language, the union is a collection of elements of the different data type. The union is used to create user-defined data type in the C programming language. As the union used to create a user-defined data type, the union is also said to be “user-defined data type in C”.

In other words, the union is a collection of non-homogeneous elements. Using union we can define new data types called user-defined data types that holds multiple values of the different data type.

The formal definition of a union is as follows...

Union is a colloction of different type of elements under a single name that acts as user defined data type in C.

Generally, unions are used to define a record in the c programming language. Unions allow us to combine elements of a different data type into a group. The elements that are defined in a union are called members of union.

How to create union?

To create union in c, we use the keyword called "***union***".

We use the following syntax to create unions in c programming language.

```
union <structure_name>
{
    data_type member1;
    data_type member2, member3;
    .
    .
} ;
```

Following is the example of creating a union called Student which is used to hold student record.

Creating union in C

union Student { char stud_name[30]; int roll_number; float percentage; } ;

Important Points to be Remembered

Every union must terminated with semicolon symbol (;).

"union" is a keyword, it must be used in lowercase letters only.

Creating and Using union variables

In a c programming language, there are two ways to create union variables. We can create union variable while the union is defined and we can also create after terminating union using union keyword.

TO access members of a union using union variable, we use dot (.) operator.

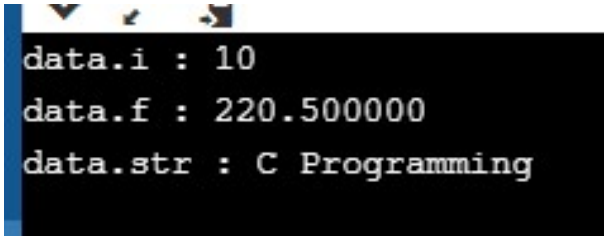
Consider the following example code...

Creating and Using union variables in C

```
union Student {
char stud_name[30];
int roll_number;
float percentage;
} stud_1 ; // while defining union
void main()
{
union Student stud_2; // using union keyword
printf("Enter details of stud_1 : \n");
printf("Name : ");
scanf("%s", stud_1.stud_name);
printf("Roll Number : ");
scanf("%d", &stud_1.roll_number);
printf("Percentage : ");
scanf("%f", &stud_1.percentage);
printf("***** Student 1 Details *****\n);
printf("Name of the Student : %s\n", stud_1.stud_name);
printf("Roll Number of the Student : %i\n", stud_1.roll_number);
printf("Percentage of the Student : %f\n", stud_1.percentage);
}
```

In the above example program, the union variable "**stud_1** is created while defining the union and the variable "**stud_2** is careted using union keyword. Whenever we access the members of a union we use the dot (.) operator.

```
#include <stdio.h>
#include <string.h>
 union Data
 {
 int i;
float f;
 char str[20];
 };
 int main( )
{
union Data data;
 data.i = 10
; printf( "data.i : %d\n", data.i);
 data.f = 220.5;
printf( "data.f : %f\n", data.f);
strcpy( data.str, "C Programming");
printf( "data.str : %s\n", data.str);
return 0;
}
```



When the above code is compiled and executed, it produces the following result −

```
data.i : 10
data.f : 220.500000
data.str : C Programming
```

Memory allocation of Union

When the unions are used in the c programming language, the memory does not allocate on defining union. The memory is allocated when we create the variable of a particular union. As long as the variable of a union is created no memory is allocated. The size of memory allocated is equal to the maximum memory required by an individual member among all members of that union.

In the above example program, the variables stud_1 and stud_2 are allocated with 30 bytes of memory each.

```
union Student{
   char stud_name[30];  ——— 30 bytes
   int roll_number;     ——— 02 bytes
   float percentage;    ——— 04 bytes
};                          max = 30 bytes
```

Here the variable of **Student** union is allocated with 30 bytes of memory and it is shared by all the members of that union.

Files in C

Generally, a file is used to store user data in a computer. In other words, computer stores the data using files.

we can define a file as follows...

File is a collection of data that stored on secondary memory like harddisk of a computer.

C programming language supports two types of files and they are as follows...

- **Text Files (or) ASCII Files**
- **Binary Files**

Text File (or) ASCII File - The file that contains ASCII codes of data like digits, alphabets and symbols is called text file (or) ASCII file.

Binary File - The file that contains data in the form of bytes (0's and 1's) is called as binary file. Generally, the binary files are compiled version of text files.

File Operations in C

The following are the operations performed on files in c programming langauge...

- **Creating (or) Opening a file**
- **Reading data from a file**
- **Writing data into a file**
- **Closing a file**

All the above operations are performed using file handling functions available in C. We discuss file handling functions in the next topic.

File Handling Functions in C

File is a collection of data that stored on secondary memory like hard disk of a computer.

The following are the operations performed on files in the c programming language...

Creating (or) Opening a file
Reading data from a file
Writing data into a file
Closing a file

All the above operations are performed using file handling functions available in C.

Creating (or) Opening a file

To create a new file or open an existing file, we need to create a file pointer of FILE type.

Following is the sample code for creating file pointer.

File *f_ptr ;

We use the pre-defined method **fopen()** to create a new file or to open an existing file. There are different modes in which a file can be opened.

Consider the following code...

File *f_ptr ; *f_ptr = fopen("abc.txt", "w") ;

The above example code creates a new file called **abc.txt** if it does not exists otherwise it is opened in writing mode.

In C programming language, there different modes are available to open a file and they are shown in the following table.

S. No.	Mode	Description
1	r	Opens a text file in **reading** mode.
2	w	Opens a text file in **wirting** mode.
3	a	Opens a text file in **append** mode.
4	r+	Opens a text file in both **reading and writing** mode.
5	w+	Opens a text file in both **reading and writing** mode. It set the cursor position to the begining of the file if it exists.
6	a+	Opens a text file in both **reading and writing** mode. The reading operation is performed from begining and writing operation is performed at the end of the file.

Note - The above modes are used with text files only. If we want to work with binary files we use

rb, wb, ab, rb+, wb+ and ab+.

Reading from a file

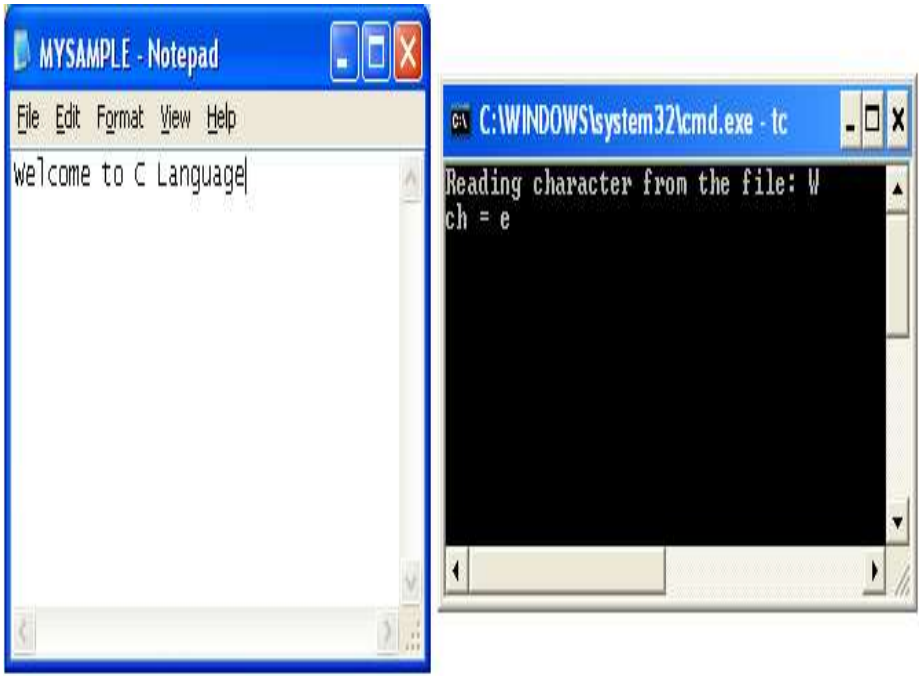
The reading from a file operation is performed using the following pre-defined file handling methods.

getc()
getw()
fscanf()
fgets()
fread()

getc(*file_pointer) - This function is used to read a character from specified file which is opened in reading mode. It reads from the current position of the cursor. After reading the character the cursor will be at next character.

Example Program to illustrate getc() in C.

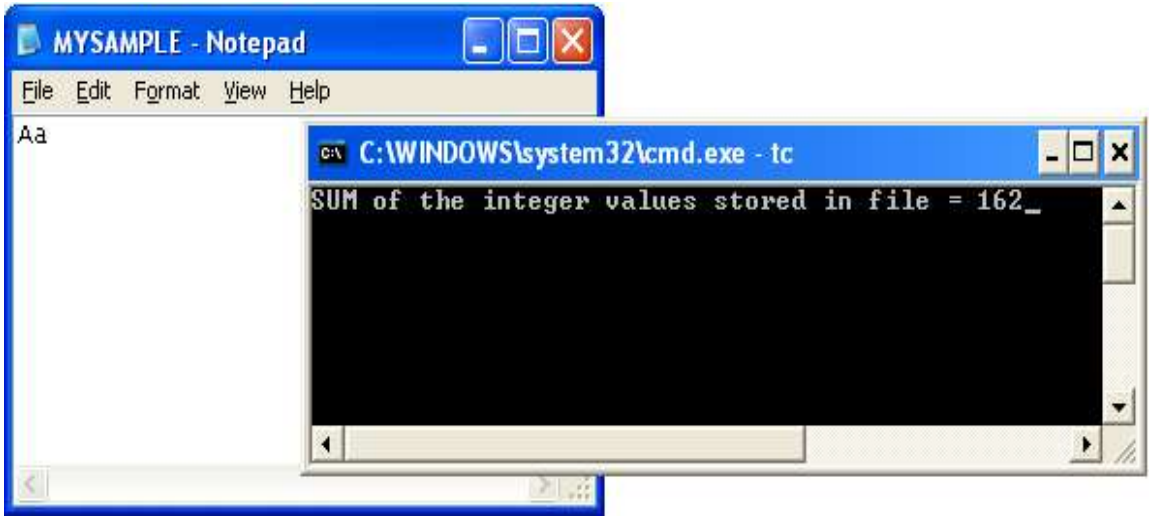
```
#include<stdio.h>
#include<conio.h>
int main()
{
FILE *fp;
 char ch;
clrscr();
fp = fopen("MySample.txt","r");
printf("Reading character from the file: %c\n",getc(fp));
ch = getc(fp);
printf("ch = %c", ch);
fclose(fp);
getch();
return 0;
}
```



getw(*file_pointer) - This function is used to read an integer value form the specified file which is opened in reading mode. If the data in file is set of characters then it reads ASCII values of those characters.

Example Program to illustrate <u>getw()</u> in C.

```
#include<stdio.h>
 #include<conio.h>
int main()
{
FILE *fp;
int i,j;
 clrscr();
fp = fopen("MySample.txt","w");
putw(65,fp); // inserts A
 putw(97,fp); // inserts a
fclose(fp); fp = fopen("MySample.txt","r");
i = getw(fp); // reads 65 - ASCII value of A
 j = getw(fp); // reads 97 - ASCII value of a
printf("SUM of the integer values stored in file = %d", i+j); // 65 + 97 = 162
fclose(fp);
getch();
 return 0;
 }
```

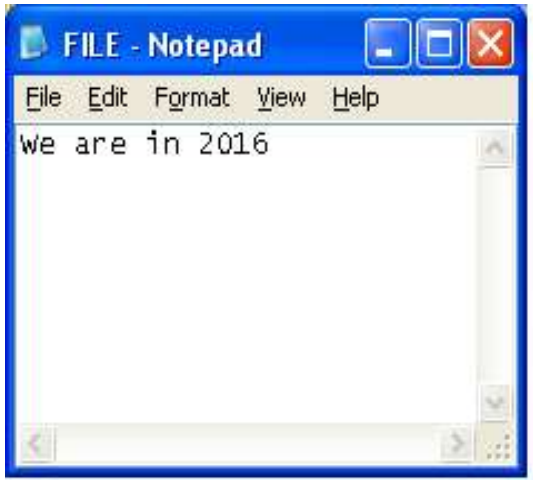


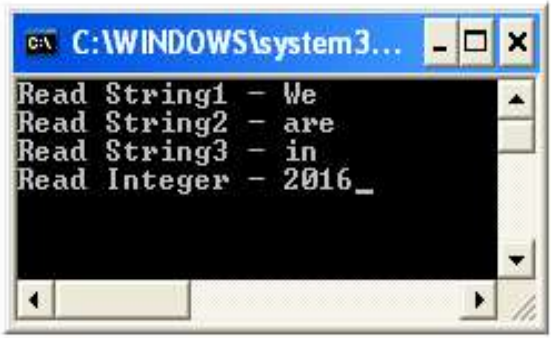
Output

fscanf(*file_pointer, typeSpecifier, &variableName) - This function is used to read multiple datatype values from specified file which is opened in reading mode.

Example Program to illustrate fscanf() in C.

```
#include<stdio.h>
#include<conio.h>
int main()
{
 char str1[10], str2[10], str3[10];
 int year;
 FILE * fp;
clrscr();
fp = fopen ("file.txt", "w+");
fputs("We are in 2016", fp);
rewind(fp); // moves the cursor to begining of the file
fscanf(fp, "%s %s %s %d", str1, str2, str3, &year);
printf("Read String1 - %s\n", str1 );
 printf("Read String2 - %s\n", str2 );
 printf("Read String3 - %s\n", str3 );
printf("Read Integer - %d", year );
 fclose(fp);
 getch();
 return 0;
}
```

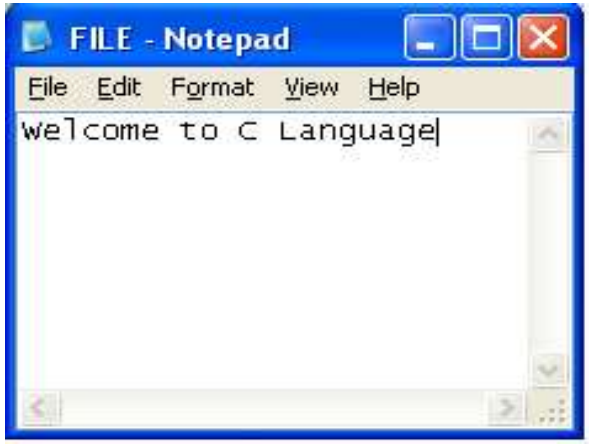




fgets(variableName, numberOfCharacters, *file_pointer) - This method is used for reading a set of characters from a file which is opened in reading mode starting from the current cursor position. The fgets() function reading terminates with reading NULL character.

Example Program to illustrate fgets() in C.

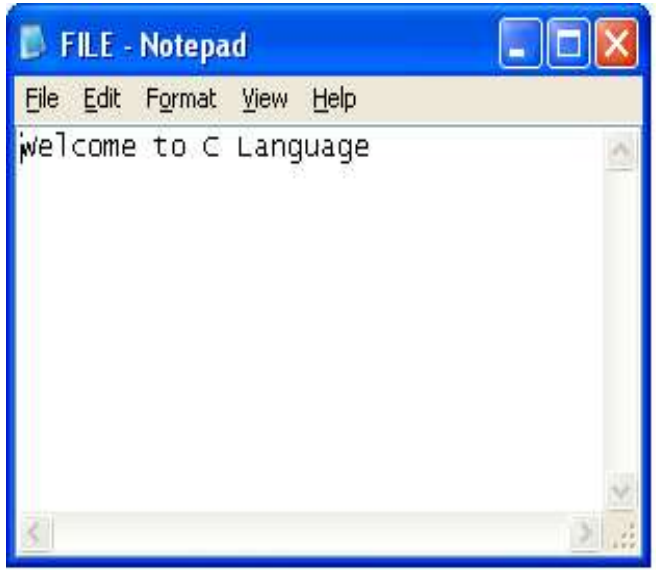
```
#include<stdio.h>
 #include<conio.h>
int main()
{
FILE *fp;
 char *str;
clrscr();
fp = fopen ("file.txt", "r");
 fgets(str,6,fp);
printf("str = %s", str);
fclose(fp);
getch();
return 0;
}
```

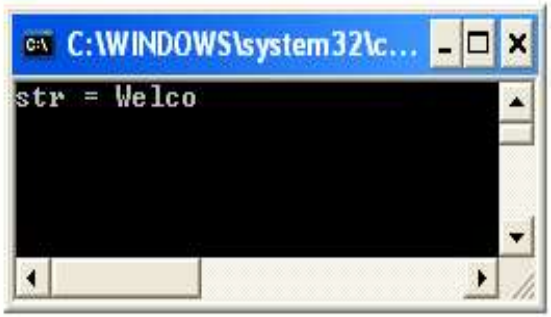


*fread(source, sizeofReadingElement, numberOfCharacters, FILE *pointer) –*
This function is used to read specific number of sequence of characters from the specified file which is opened in reading mode.

Example Program to illustrate fgets() in C.

```
#include<stdio.h>
 #include<conio.h>
int main()
{
 FILE *fp;
char *str;
clrscr();
fp = fopen ("file.txt", "r");
fread(str,sizeof(char),5,fp);
str[strlen(str)+1] = 0;
printf("str = %s", str);
 fclose(fp);
getch();
 return 0;
}
```





Writing into a file

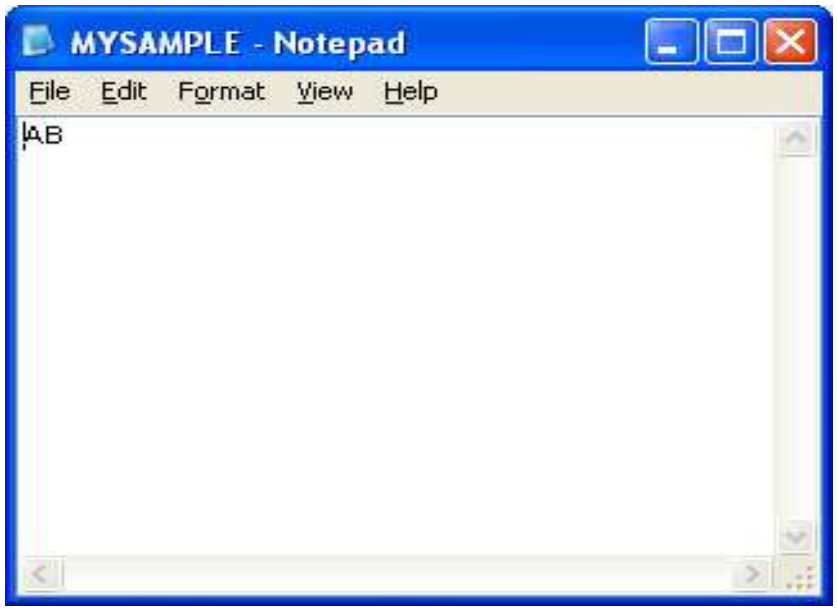
The writing into a file operation is performed using the following pre-defined file handling methods.

putc()
putw()
fprintf()
fputs()
fwrite()

putc(char, *file_pointer) - This function is used to write/insert a character to the specified file when the file is opened in writing mode.

Example Program to illustrate <u>putc()</u> in C.

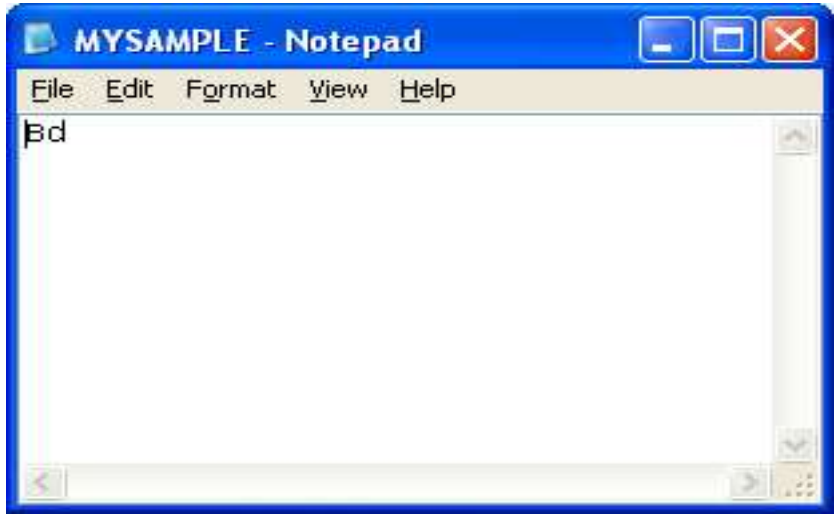
```
#include<stdio.h>
#include<conio.h>
int main()
{
 FILE *fp;
char ch;
 clrscr();
fp = fopen("C:/TC/EXAMPLES/MySample.txt","w");
putc('A',fp);
ch = 'B';
putc(ch,fp);
fclose(fp);
getch();
return 0;
}
```



putw(int, *file_pointer) - This function is used to writes/inserts an integer value to the specified file when the file is opened in writing mode.

Example Program to illustrate <u>putw()</u> in C.

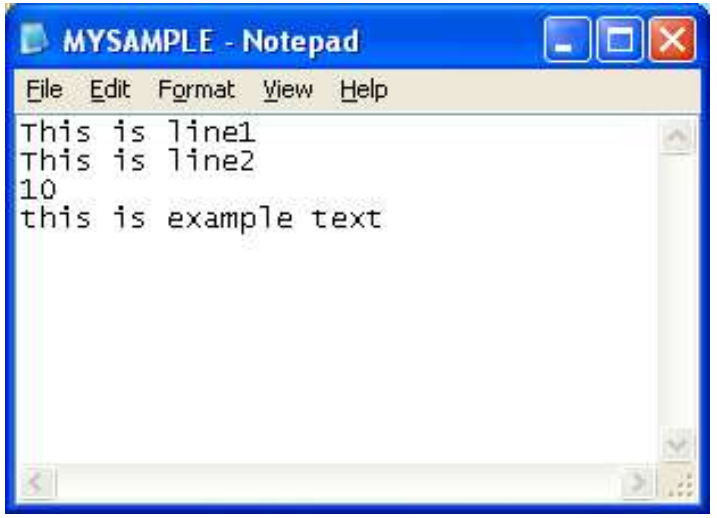
```
#include<stdio.h>
#include<conio.h>
int main()
{
 FILE *fp;
 int i;
 clrscr();
 fp = fopen("MySample.txt","w");
putw(66,fp);
i = 100;
putw(i,fp);
fclose(fp);
getch();
return 0;
 }
```



fprintf(*file_pointer, "text") - This function is used to writes/inserts multiple lines of text with mixed data types (char, int, float, double) into specified file which is opened in writing mode.

Example Program to illustrate "fprintf()" in C.

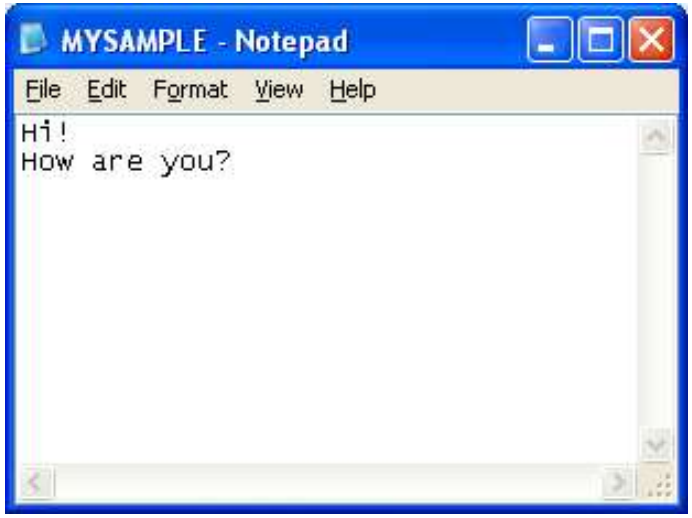
```
#include<stdio.h>
 #include<conio.h>
 int main()
{
 FILE *fp;
char *text = "\nthis is example text"; int i = 10;
clrscr();
 fp = fopen("MySample.txt","w");
fprintf(fp,"This is line1\nThis is line2\n%d", i);
fprintf(fp,text);
fclose(fp);
getch();
 return 0;
}
```



fputs("string", *file_pointer) - TThis method is used to insert string data into specified file which is opened in writing mode.

Example Program to illustrate fputs() in C.

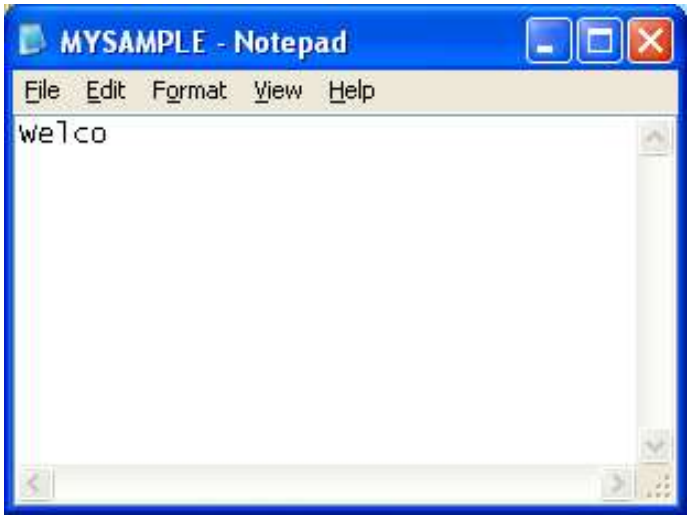
```
#include<stdio.h>
#include<conio.h>
int main()
{
FILE *fp;
char *text = "\nthis is example text";
clrscr();
fp = fopen("MySample.txt","w");
 fputs("Hi!\nHow are you?",fp);
fclose(fp);
getch();
return 0;
}
```



fwrite("StringData", sizeof(char), numberOfCharacters, FILE *pointer) -
This function is used to insert specified number of characters into a binary file which is opened in writing mode.

Example Program to illustrate fwrite() in C.

```
#include<stdio.h>
 #include<conio.h>
 int main()
{
 FILE *fp;
char *text = "Welcome to C Language";
clrscr();
fp = fopen("MySample.txt","wb");
fwrite(text,sizeof(char),5,fp);
fclose(fp);
 getch();
return 0;
}
```



Closing a file

Closing a file is performed using a pre-defined method fclose().

fclose(*f_ptr)

The method fclose() returns '0'on success of file close otherwise it returns EOF (End Of File).

Cursor Positioning Functions in Files

C programming language provides various pre-defined methods to set the cursor position in files.

The following are the methods available in c, to position cursor in a file.

ftell()

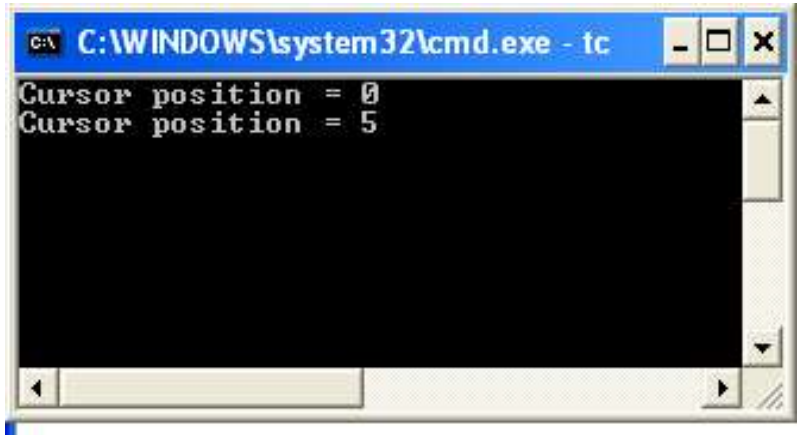
rewind()

fseek()

ftell(*file_pointer) - This function returns the current position of the cursor in the file.

Example Program to illustrate ftell() in C.

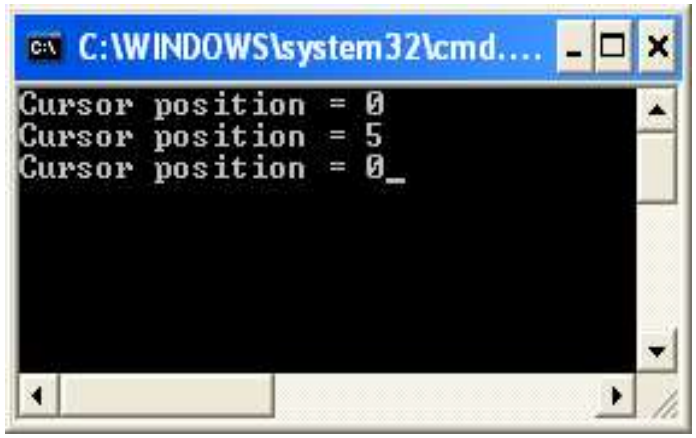
```
#include<stdio.h>
#include<conio.h>
 int main()
{
 FILE *fp;
 int position;
 clrscr();
fp = fopen ("file.txt", "r");
 position = ftell(fp);
 printf("Cursor position = %d\n",position);
 fseek(fp,5,0);
position = ftell(fp);
printf("Cursor position = %d", position);
fclose(fp);
getch();
 return 0;
}
```



rewind(*file_pointer) - This function is used reset the cursor position to the beginning of the file.

Example Program to illustrate rewind() in C.

```
#include<stdio.h>
#include<conio.h>
int main()
{
 FILE *fp;
int position;
clrscr();
fp = fopen ("file.txt", "r");
 position = ftell(fp);
printf("Cursor position = %d\n",position);
fseek(fp,5,0);
position = ftell(fp);
printf("Cursor position = %d\n", position);
 rewind(fp);
position = ftell(fp);
printf("Cursor position = %d", position);
fclose(fp);
getch();
return 0;
 }
```

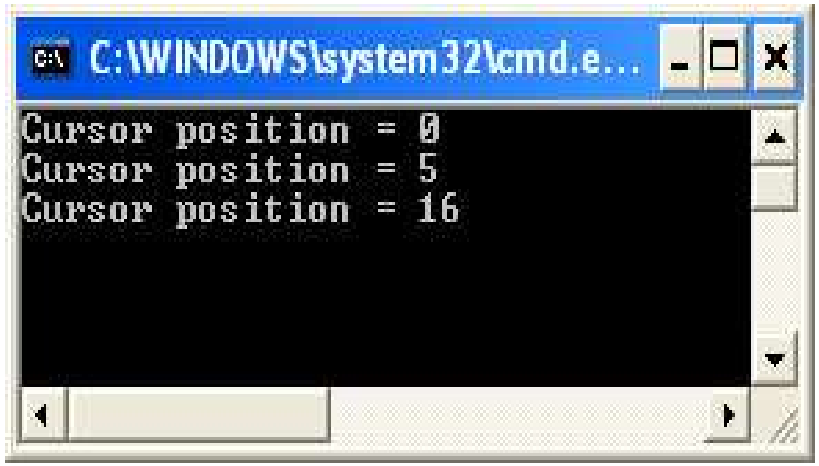


fseek(*file_pointer, numberOfCharacters, fromPosition) - This function is used to set the cursor position to the specific position.

Using this function we can set the cursor position from three different position they are as follows.
from beginning of the file (indicated with 0)
from current cursor position (indicated with 1)
from ending of the file (indicated with 2)

Example Program to illustrate fseek() in C.

```
#include<stdio.h>
#include<conio.h>
 int main()
{
FILE *fp;
 int position;
clrscr();
fp = fopen ("file.txt", "r");
position = ftell(fp);
printf("Cursor position = %d\n",position);
fseek(fp,5,0);
position = ftell(fp);
 printf("Cursor position = %d\n", position);
 fseek(fp, -5, 2);
position = ftell(fp);
printf("Cursor position = %d", position);
fclose(fp);
 getch();
return 0;
}
```




Thank
you