

UNIT I

Computer Programming Languages:**Programming language:**

Definition: A programming language is the communication bridge between a programmer and computer, which is designed **to communicate instructions to a machine, particularly a computer.**

Or

Definition: A programming language is the communication bridge between a programmer and computer, which is developed **for passing our data (or) instructions to the computer to do specific task (or) job.**

We have three types of programming languages

1. Low level language(Machine level language)
2. Mid-level language(Assembly level language)
3. High level language

1. Machine level language (low level)-1940's:

1. It is the **basic (or) fundamental language** of computer's processor. Machine level language is the language that **is directly understood by the computer**
2. Machine level language is the **combination of 0's and 1's**, which **represents off and on.**
3. All programs (high level) are converted into machine level language before they can be executed.

Advantages:

1. The advantage of Machine level language **is it runs very fast** because CPU directly understands machine language instructions.
2. It takes **less time to execute** the instructions.

Disadvantages:

1. It **is difficult to understand and develop programs** using machine level language.
2. **Machine dependent** (machine language varies from one computer to another computer).
3. **Writing the error free code is also difficult** (hard to understand and remember the various combinations of 0's and 1's representing data and instructions).
4. **Difficult to debug and modify:** Finding the errors and modifying the task is highly problematic.

2. Assembly level language (mid-level)-1950's:

1. All the instructions are in the forms of **mnemonics** (combination of symbols, letters and digits).
The **symbolic instruction language** called as **Assembly Language.**

2. A translator program is needed to translate the Assembly level language to machine level language(ASCII -American Standard Code For Information Interchange)
3. This translator program is called **Assembler**.
4. Here we can use meaningful words like ADD,SUB,MUL

Example: LD R1 7; LOAD 1st VALUE in r1
 LD R2 10; LOAD 2nd VALUE in r2
 ADD R1, R2; R1=R1+R2
 Halt Halt the process

Advantages:

1. Writing a program in assembly language is more convenient than writing one in machine language because the assembly level program written in the form of symbolic instructions, which improves readability.
2. **Execution is very fast** like machine level language.
3. Easier to correct the errors and modify program instructions

Disadvantages:

1. **Machine dependent** (Assembly Language programs written for one processor will not work on a different processor).
2. Programming is different and **time consuming**.
3. The programmer must know the logical structure of a computer.

3.High level language-1960's:

1. High level languages have the instructions that are very similar to human languages
2. High level languages are simple that uses English words and it has syntax and semantics that makes it easy for a programmer to write programs and identify and correct the errors.
3. The high level language is converted to machine level language for the computer to understand. for this we need **compiler** and this process is called **compilation**.
4. Examples are **FORTRAN**(formula translation language), **COBOL** (common business oriented language), **BASIC** (beginner's all-purpose symbolic instruction code) and C.

Example: int a, b, c;
 c=a+b;

Advantages:

1. Easy to learn (we are using native languages).
2. **Portable** (programs can run on different machines with little (or) no change).
3. Errors can be easily detected and removed.
4. Developing programs is very easy because the instructions are very closer to the native language

Translators:

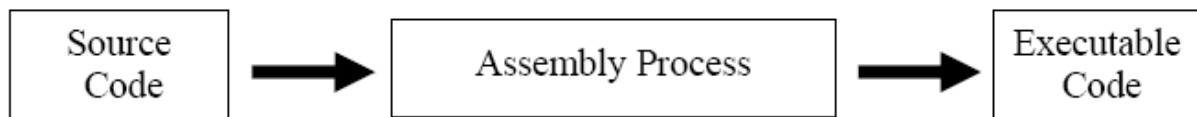
High level and Mid-level languages require translators for communicating the instructions to the computer. We have three types of translators.

1. Assembler
2. Interpreter
3. Compiler

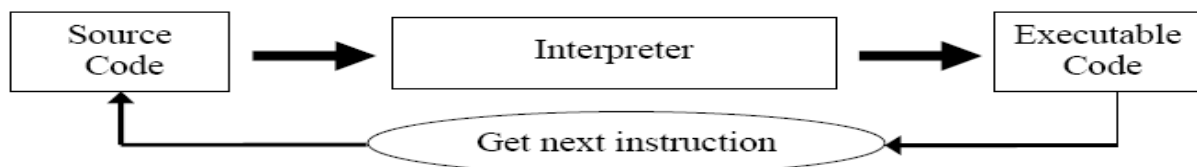
🚦 **Source Code**: Source code is the code **that is input to a translator**.

🚦 **Executable code**: Executable code is the code **that is output from the translator**.

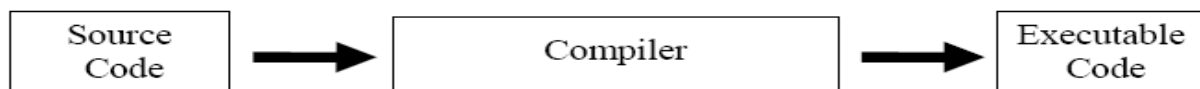
Assembler: An Assembler converts an assembly program into machine code.



Interpreter: An Interpreter is also a program that translates high-level source code into Executable code. However the difference between a compiler and an interpreter is that an **interpreter translates one line at a time and then executes it** and then the next instruction is translated and this goes on until end of the program.



Compiler: A Compiler is a program that translates a high level language into machine code.

**Algorithm:**

Algorithm is nothing but **step-by-step solution of a problem** which is written in a normal language.

Or

Algorithm is an **effective procedure for solving a problem in a finite number of steps**.

a) One problem may have many numbers of algorithms.

An algorithm must possess the following properties:

Input: An Algorithm must take zero or more inputs.

Output: An Algorithm should provide at least one output.

Finiteness: An Algorithm should give results in a finite number of steps.

Definiteness: An Algorithm should not contain ambiguous statements (or) instructions (Each instruction must be clear and unambiguous).

Effective: Each statement in an algorithm is easily converted into an instruction and it can be executed in a finite amount of time.

Advantages:

1. It provides core solution to the given problem.
2. Reduces the task into a series of smaller steps of more manageable size.
3. Algorithm can be implemented on a computer by using any programming language.
4. Problems can be approached as a series of small, solvable sub-problems, which facilitates program development.
5. It facilitates finding the efficient solution.

Disadvantages:

1. For the complex problems tracing of algorithms and flow of control will be difficult.
2. Algorithms have no visual representation like flow charts.
3. Understanding the logic is difficult when compared to flow chart.

Ex 1: Algorithm to read the name and print the name

Algorithm: Step 1 : Start
 Step 2 : Read input name
 Step 3 : Print name
 Step 4 : Stop

Ex 2: Algorithm to calculate area of square

Algorithm: Step 1 : Start
 Step 2 : Read value for a (side)
 Step 3 : [Compute] $\text{Area} = A * A$
 Step 4 : Output Area
 Step 5 : Stop

Ex 3: Algorithm to find the average of three numbers

Algorithm: Step1 : Start
 Step 2 : Read three Numbers A, B and C
 Step 3 : Compute $\text{Average} = (A+B+C)/3$
 Step 4 : Print Average
 Step 5 : Stop

Ex4: - Algorithm for finding the maximum of two numbers:

Algorithm: Step1: start

Step2: Read three numbers (A,B)
Step3: If $A < B$, then goto step5
Step4: print 'A' is maximum goto step6
Step5: Print 'B' is maximum.
Step6: Stop.

Ex5:- Algorithm to find whether the given number is even (or) odd:

Algorithm: Step1: start
Step2: Read number num.
Step3: Compute the modulus operation i.e. $chk = num \% 2$.
Step4: If $chk = 0$ then
 Display the given number is even
Step5: If $chk = 1$ then
 Display the given number is odd
Step6: stop

Ex6: Write an algorithm to find the sum of digits of a number.

Algorithm: Step1: start
Step2: Read n
Step3: Initialize $sum = 0$
Step4: if $n \leq 0$ then goto step6
Step5: $sum \leftarrow sum + n \bmod 10$
 $n \leftarrow n / 10$ then goto step4
Step6: print sum
Step7: Stop.

Ex7 : Write an algorithm to find sum of n natural numbers.

Algorithm: step1: start
Step2: Read n
Step3: $sum \leftarrow 0, i \leftarrow 1$
Step4: if $i > n$ then goto step6
Step5: $sum \leftarrow sum + i$
 $i \leftarrow i + 1$ then goto step4
step6: print sum
step7: stop.

Flow chart:

Graphical or symbolic representation of an algorithm is “Flow chart”

(Or)

Diagrammatic representation of the step-by-step solution to a given problem

Flow charts are useful to analyze working procedure of an algorithm and identify any pitfalls in the algorithm. The following graphic symbols are used while preparing a flowchart.

Advantages of Flowchart:

1. **Conveys Better meaning:** Since a flowchart is a pictorial representation of an algorithm, it is easier for a programmer to understand and explain the logic of the program to other programmers.
2. **Analyses the problem effectively:** Flow chart helps the programmers to analyze the problem in detail by breaking the flow chart into detailed parts.
3. **Effective Coding:** It is very easy for the programmers to write the program, because flow chart will give a clear idea of the steps involved.
4. **Systematic Debugging:** It enables easy detection of errors.

Disadvantages:

1. For a large program flowchart becomes very complex and confusing.
2. Modification of a flowchart is difficult.
3. Excessive use of connectors may confuse programmers.

Start and End (oval):



The start and end symbols indicate both the beginning and end of the flowchart.

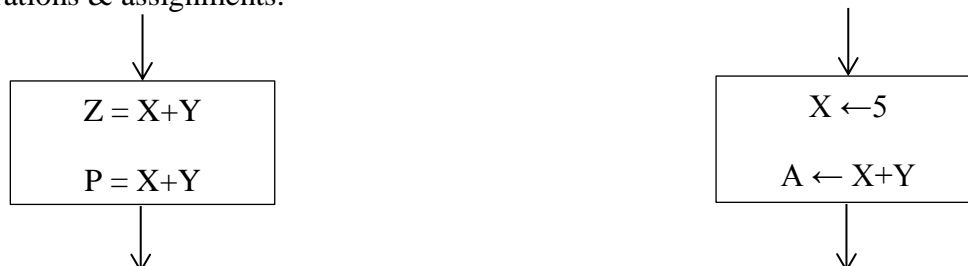
Input/output Symbol (parallelogram):

The input/output symbol looks like a parallelogram and is used for input and output of data.



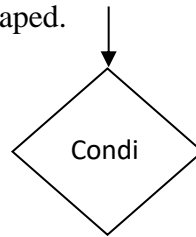
Process Block (rectangle):

The symbol for process block is indicated by a rectangle. The symbol is used to indicate operations & assignments.



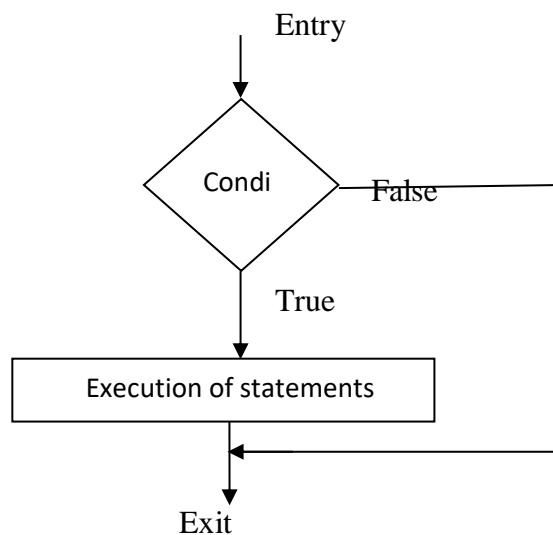
Decision (or) Test Symbols (diamond): This is a diamond shaped box, which is used to indicate logical checking and gives decision after comparing between two or more objects (Eg. Yes or No; True or False, =, >, <, etc.).

The decision symbol is diamond shaped.

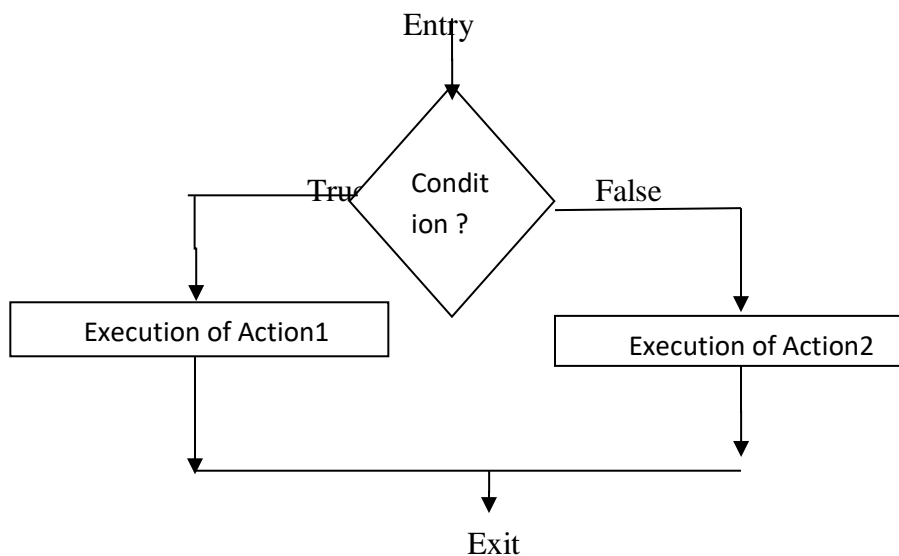


Depending upon condition the decision block selects one of the alternatives.

Ex: - Single alternative decision

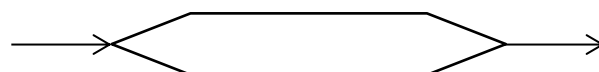


Ex :- Two alternative decision

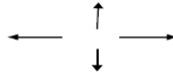


Loop Symbol (hexagon):

The Symbol looks like a hexagon. This symbol is used for implementation of loops only.



Flow Lines: These are arrow mark symbols used to connect two boxes and to indicate the direction of data or processing flow.



Connector Symbol (circle):

This is a Circular-shaped symbol used to connect different parts of flowchart. When the flow chart is lengthy, it is split into different pages. Then these connectors are used to connect between the pages at the beginning and at the end of each page.



Connector for connecting to the next box



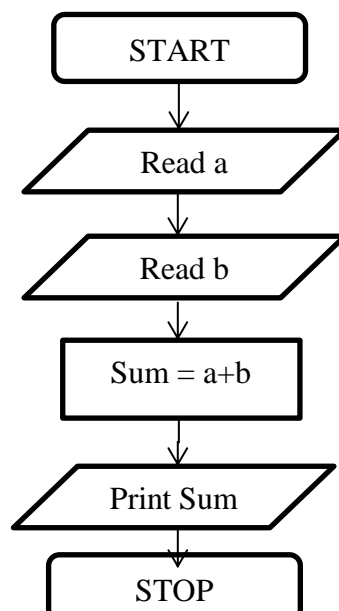
Connector that comes from the previous block

Design rules to draw effective flowchart:

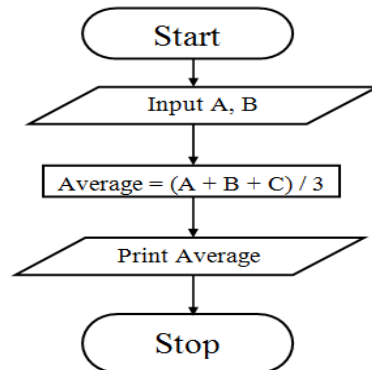
1. Flow chart should begin with start symbol and should end with stop symbol.
2. All boxes in the flowchart are connected with arrows.
3. The process flow should be from top to bottom (or) from left to right.
4. The decision symbol has two exit points; these can be sides (or) one bottom and one side.
5. The action flow chart symbol must have only one input arrow and one output arrow.
6. Connectors are used to connect breaks
 - I. From one page to another page
 - II. From the bottom of the page to the top of the page
7. Flowchart should be neat, clear and unambiguous

Examples of Flow Charts:

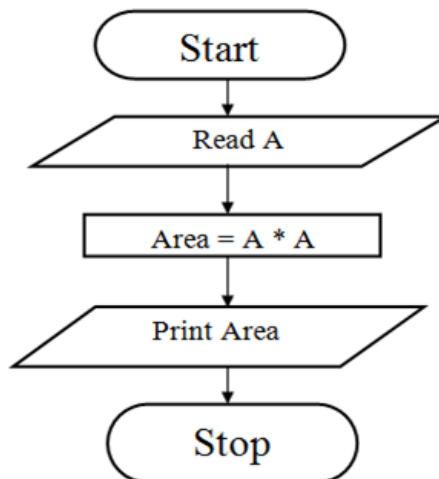
Problem statement: Read two numbers and produce the sum of those numbers.



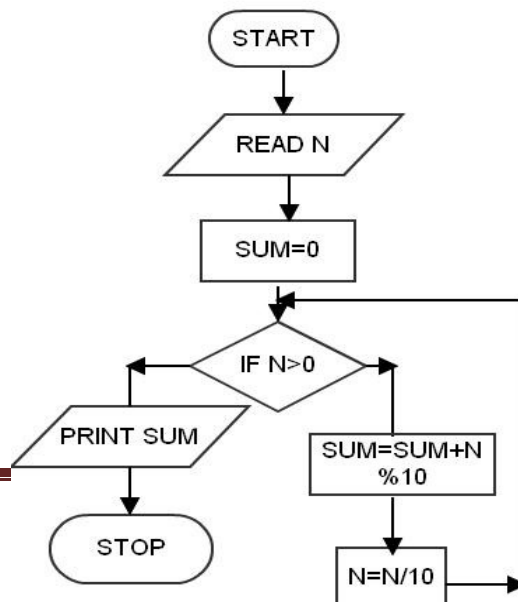
Problem Statement: Flowchart to find the average of three numbers



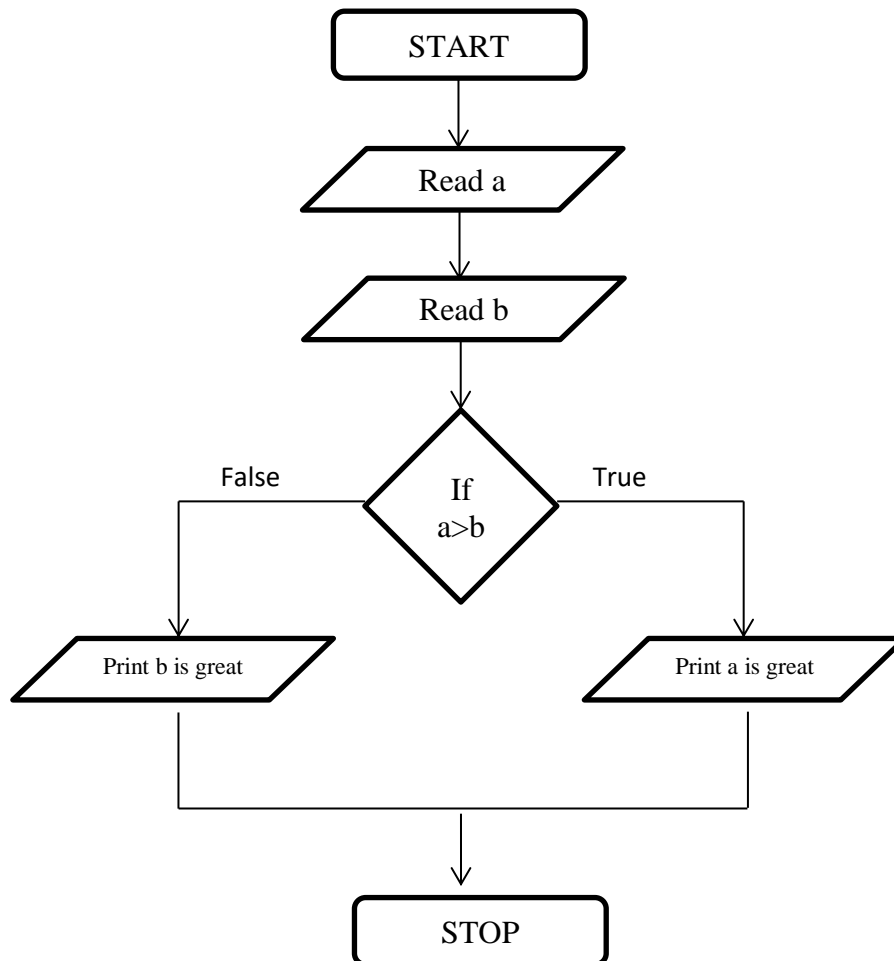
Problem Statement: Flowchart to calculate area of square



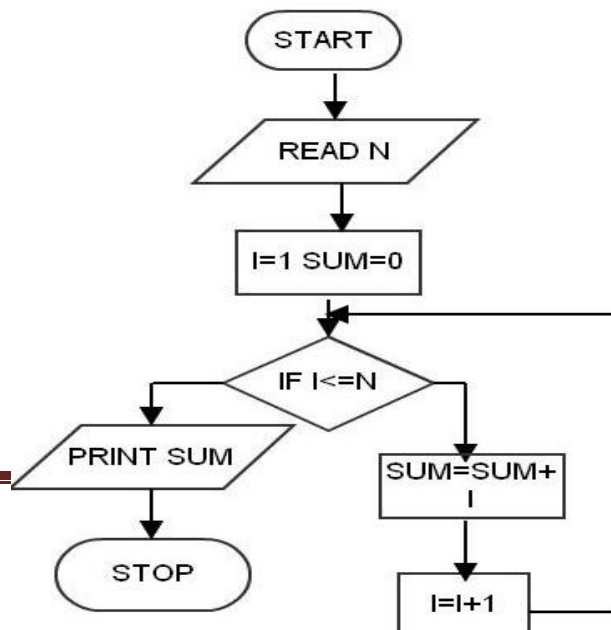
Problem Statement: Flowchart to find the sum of individual digits of a given number.



Problem Statement: Read two numbers & print the greater number.



Problem Statement: Flowchart to find the sum of n natural numbers.



Program Control Structures:

The structure of an algorithm (or) Flowchart (or) program decides the order of execution of the steps. The steps in an algorithm can be divided into three categories

- 1. Sequence**
- 2. Selection**
- 3. Iteration**

1. Sequence: The steps described in an algorithm are performed successively one by one without skipping any step.

Ex: Algorithm:

- Step 1 : Start
- Step 2 : Read input name
- Step 3 : Print name
- Step 4 : Stop

2. Selection:

If the failure occurs in the sequence, there must be some alternative to handle. The selection statement is

```
if (Condition)
    Statement1;
else
    Statement 2;
```

Example:Algorithm: step1:start

- Step2:Read two numbers (A,B)
- Step3:If $A < B$, then goto step5
- Step4:print 'A' as maximum. goto step6
- Step5:Print 'B' is maximum.
- Step6:Stop.

3. Iteration:

In a program it is sometimes necessary to perform the same action for number of times. If the statement is written repetitively, it will increase the program code. To avoid this problem, iteration mechanism is applied. The statements used are for, while and do while.

1. for loop:

```
    for (expression1; expression2; expression3)
    {
        Statement;
    }
```

2. While loop:

```
    expression1;
    while (expression2)
    {
        Statement;
        expression3;
    }
```

3. Do-while loop:

```
    expression1
do
    {
        Statement;
        expression3;
    } while (expression2);
```

Ex: Algorithm:

```
    step1: start
    step2: Read n
    step3: sum ← 0, i ← 1
    step4: if i > n then goto step6
    step5: sum ← sum + i
           i ← i + 1 then goto step4
    step6: print sum
    step7: stop.
```

Software Development Method:

Programmers use the Software Development Method for solving a problem.

The Following are the steps required for developing the software.

- 1. Specify the Problem requirements**
- 2. Analyze the Problem**
- 3. Design the algorithm to solve the problem**
- 4. Implement the algorithm**
- 5. Test and verify the program**

6. Maintain and update the program

Let us discuss each method in detail.

1. Specify the problem requirements: In this find out the problem clearly and unambiguously and understand what is required for its solution

2. Analyze the problem: Analyzing the problem clearly or identifying the problems

- a) Input- This is the data you have to work with
- b) Output-The desired results
- c) Any additional requirement or constraints on the solution.

Here we should also determine the required format, in which results should be displayed, what are the variables that are required for the problem, what are the hardware and software requirements.

Eg: compute and display total cost of books purchased given the number of books purchased and cost of each book.

Problem input: Number of books purchased and Cost of Each Book.

Output: total cost of the books.

Once we know the problem inputs and output develop a list of formulas that shows the relation between them.

$$\text{Total cost} = \text{unit cost} * \text{number of items}$$

So, total cost of books = cost of each book * Number of Books

3. Design the algorithm to solve the problem:

Once we have fully understand and analyze the problem, we can design an algorithm and flowchart to solve the problem. For designing the algorithm we can follow top-down approach, in this you first list out the sub problems, that needs to be solved, then you solve the original problem.

Algorithm for programming problem:

- a) Get Data (input)
- b) Perform Computations
- c) Display results(output)

4. Implement the algorithm: Implementing the algorithm means writing it as a program. Each algorithm step is converted into one or more statements in a programming language. It can be in either top down or bottom up approach.

5. Test and verify the completed program: After we have written our program we must test ,it verifies that it works properly or not. Run the program several times using different sets of input data, to make sure that our program must work for every legal input.

6. Maintain and update the program: Maintaining and updating the program involves modifying the program to remove the previously detected errors keeping updating the program.

Example: Applying the software development method

Let us take a case study and apply the software development method steps

Case study: converting miles to kilometers

Problem:our problem is to convert miles to kilometers.

Analysis: The problem must be analyzed clearly by taking problem inputs, problem output and deriving the relevant formula for that problem.

Data Requirements: Problem input: miles

Problem output: kilometers

Relevant formula: 1 mile = 1.609 kilometers.

Designing algorithm: The algorithm must be designed as follows:

Algorithm

- 1) Start
- 2) Get the distance in miles
- 3) Convert the distance into kilometers
- 4) Display the distance in kilometers
- 5) stop

Implementation:

The algorithm must be implemented in as a 'C' Program. The following is the program for this case study.

```
#include<stdio.h>
#include<conio.h>
int main ()
{
    double miles, kms;
    printf("Enter the distance in miles");
    scanf ("%f", &miles);
    kms=1.609*miles;
    printf ("Distance in kilometers is: %f",kms);
    return 0;
}
```

Testing: After writing the program, we check our program whether it gives the correct results for all legal inputs.

Maintenance: To remove all the detected errors and keep our program up-to-date.

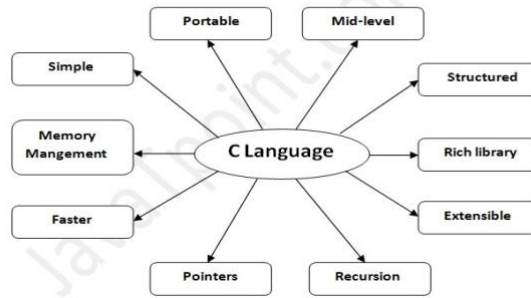
UNIT II**Introduction to C Programming****History & Evaluation of C**

C programming language was developed in 1972 by Dennis Ritchie at bell laboratories of AT&T (American Telephone & Telegraph), located in U.S.A. It was developed to overcome the problems of previous languages such as B, BCPL etc. Initially, C language was developed to be used in **UNIX operating system**. It inherits many features of previous languages such as B and BCPL and added the concept of data types and other powerful features.

Language	Year	Developed By
ALGOL	1960	International Group
BCPL	1967	Martin Richard
B	1970	Ken Thompson
Traditional C	1972	Dennis Ritchie
K & R C	1978	Kernighan & Dennis Ritchie
ANSI C	1989	ANSI Committee
C89	1990	ISO
C99	1999	Standardization committee

The famous book *The C Programming Language* was written by Kernighan and Ritchie in 1978. In 1983, ANSI (American national standard institute) established a committee, whose goal was to produce a machine independent definition of the c language. The committee approved a version of c language known as ANSI C in 1989, then it also approved by ISO the result is C89. In 1999 the standardization committee released a version of C99 by adding some features of C++ / JAVA.

Features of C Language:



C is the widely used language. It provides a lot of **features** that are given below.

1).Simple: C is a simple language in the sense that it provides **structured approach** (to break the problem into parts), **rich set of library functions, data types** etc.

2).Machine Independent or Portable: C programs **can be executed in many machines** with little bit or no change. But it is not platform-independent.

3).Mid-level programming language: C is **also used to do low level programming**. It is used to develop system applications such as kernel, driver etc. It **also supports the features of high level languages**. That is why it is known as mid-level language.

4).Structured programming language: C is a structured programming language in the sense that **we can break the program into parts using functions**. So, it is easy to understand and modify.

5).Rich Library: C **provides a lot of inbuilt functions** that makes the development fast.

6).Memory Management: It supports the feature of **dynamic memory allocation**. In C language, we can free the allocated memory at any time by calling the **free()** function.

7).Speed: The compilation and execution time of C language is fast.

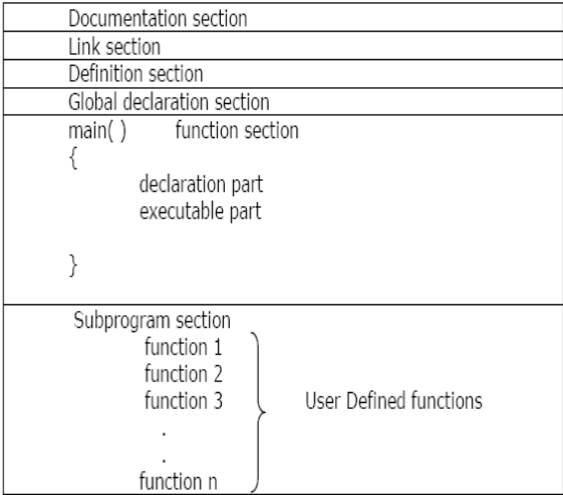
8).Pointer: C provides the feature of pointers. We can directly interact with the memory by using the pointers. We **can use pointers for memory, structures, functions, array** etc.

9).Recursion: In C, we **can call the function within the function**. It provides code reusability for every function.

10).Extensible: C language is extensible because it **can easily adopt new features**.

11).Case sensitive: Upper case and Lowercase letters are treated differently.

Basic Structure of a C Program: C program can be viewed as a group of building blocks called functions. A function (or) subroutine that may include one (or) more statements to perform a specific task.



The Documentation Section: This section contains comment lines giving the name of the program, the author, the purpose of the program and other details that the programmer would like to have. These comments begin and ends with the two characters `/*` and `*/` respectively. It specifies a brief description of what the program does.

The Link Section: This section provides instructions to the compiler to link functions from system library. To perform the task it uses the statement `#include <headerfile.h>`

Example: `#include<stdio.h>` here
 `# ---->` preprocessor directive.
 `include ----->` Includes the content of the external file into the program.
 `stdio.h ----->` Standard header file that provides input output functions like `printf()`
 which displays information on the screen.

The Definition Section: Definition section is used to define all symbolic constants. The following statement does definition - `#define symbolic_name constant_name.`

Ex: #define PI 3.14

Global declaration section: There are some variables that are used in one or more functions. Such variables are called **global** variables and are declared in the global declaration section that is outside of all the functions.

main() function: Every C program must have one **main() function** section. This section contains two parts. These are:

1. Declaration part
2. Executable part

The declaration part declares all variables used in program execution.

The executable part that contains at least one executable statement. These two parts can appear between the opening and closing brace. The program execution begins at the opening brace and ends at the

closing brace. The closing brace of the main function section is the logical end of the program. All the statements in the declaration and executable parts ends with a semicolon (;)

The sub function section: contains all the user defined functions that are called in the main() function. User defined functions are generally placed immediately after the main function.

All sections, except the main function section, may be absent when they are not required

A sample C Program:

```
// This is a program to show the structure of C
#include<stdio.h>
#include<conio.h>
#define PI 3.14
int a;
main()
{
    scanf("%d",&a);
    area();
}
area()
{
    float area;
    area = PI*a*a;
    printf("Area is %f\n",area);
}
```

Annotations in the diagram:

- Single line comment: points to the first line of the program.
- Link section: points to the preprocessor directives (`#include`).
- Definition section: points to the macro definition (`#define`).
- main section: points to the `main()` function.
- User defined functions: points to the `area()` function.

Steps To Execute C program:

Executing a C program involves a series of steps. These are

Creating the program:

1. First Write C Program using C Editor, such as Turboc2.
2. Save Program by using **.C Extension**.
3. File Saved with .C extension is called “**Source Program**“.

Compiling the program:

1. C Source code with (.C) extension is given as input to compiler and compiler converts it into **Equivalent Machine Instruction**.
2. **Compiler Checks for errors**. If source code is error-free then Code is converted into Object File (.obj).
3. During **Compilation Compiler will check for error**, If compiler finds any error then it will report it and Users have to **re-edit the program**.
4. After **re-editing program, Compiler again check for any error**.
5. If program is error-free then program is linked with appropriate libraries and it produces executable program (prog.exe).

Linking:

1. Program is linked with **included header files**.
2. Program is linked with other libraries.

Executing the program: Run the program and test it whether we get the desired output, if not we go to Edit the program and make modifications and additions to our source program. Here the run time Errors are called logical errors.

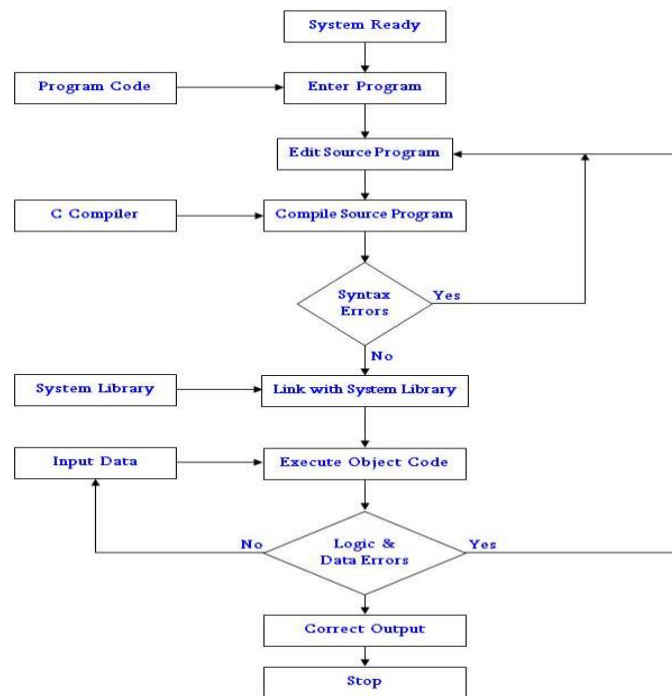


Figure 1: Process of compiling and running a C program.

Procedure:

1. Use an editor to write or prepare the “C” program called source code. The source program file has the (dot extension) .c (i.e. prog.c).
2. Compile the program using a C compiler. If the compiler does not find any errors (compilation errors) in the program it produces an object file (prog.obj). If the compiler finds errors we return to step 1 and correct errors in the source program. Link the program using a linker. If no error occurs the linker produces an executable program.(prog.exe)
3. Linker-loader is a tool that takes all the object files and links them together to form one executable file in the format that the operating system supports.
4. Execute or run the program and test it whether we get the desired output, if not we go to step 1 and make modifications and additions to our source program (Here we find out run time Errors, called logical errors).

Steps to execute c program in turboc2 c – compiler

- F2 -> to save the current file
- Alt+F9 -> to compile the current C program
- Ctrl+F9 -> after compilation to run the current C program
- Alt+F5 -> To see the result screen
- Alt+x -> To exit the turboc2 IDE

Main functions: The main is a part of every c program. C permits different forms of main statements.

```
* main()
* int main()
* void main()
* main(void)
* int main(void)
* void main(void)
```

1. The empty pair of parenthesis indicates that the function has no arguments. This may be explicitly indicated by using the key word **void** inside the parentheses.
2. We may also specify the keyword **int** or **void** before the word main.
3. The keyword **void** means that the function does not return any information to the operating system and **int** means that the function returns an integer value to the OS.
4. When **int** is specified the last statement in the main program must be `return 0`.

The syntax for main function is

```
main()
{
// function body
}
```

ERROR: While writing c programs, errors may occur unwillingly which may leads to incorrect compilation and incorrect execution of the program. There are three types of errors that may occur while developing (or) writing C program.

❖ **Compile Errors**

❖ **Logical Errors**

❖ **Runtime Errors**

Compile Errors: Compile errors are the errors that occur at the time of compilation of the program.

In C compile errors may be further classified as two types

❖ **Syntax errors**

❖ **Semantic errors**

Syntax Errors: The set of rules (grammatical rules) of a programming language for writing statements of the computer program is known as syntax of the language. The program statements are written strictly according to these rules. Syntax error occur when syntax of a programming language are not followed in writing the source code. The compiler detects these errors at compiling time of source code. The compiler reports a proper error message about the error. The compiler does not compile a program if the program contains syntax errors. The syntax errors are easy to detect and remove.

Some examples are given below:-

- ❖ Missing semicolon (;) at the end of statement.
- ❖ Missing any of delimiters i.e { or }
- ❖ Incorrect spelling of any keyword.
- ❖ Using variable without declaration etc.

Semantic Errors: Semantic errors are reported by the compiler when the statements written in the c program are not meaningful to the compiler.

For example, consider the statement,

b+c=a;

In the above statement we are trying to assign value of a in the value obtained by summation of b and c. The correct statement will be

a=b+c;

Logical Errors: The errors in the logic of the program are called **logical error**. The compiler cannot detect logical errors. A program with logical errors is compiled (translated) and run successfully but it does not give correct result.

Some examples are given below:

- ✓ The sequence of instructions used in a program may be incorrect.
- ✓ The mathematical formulas used in program instructions may be incorrect etc.

The logical errors are difficult to detect. Logical errors can only be detected by examining all the units of the program one by one. It is a very time consuming and lengthy process.

Runtime Errors: The errors that occur during the execution of program are called the **runtime errors**.

These types of errors may occur due to the following reasons.

- ✓ When the program attempts to perform an illegal operation such as dividing a number by zero.
- ✓ If input data given to the program is not in a correct format or input data file is not found in the specified path.
- ✓ If hardware problem occurs such as hard disk error, or disk full or printer error etc.

When a runtime error occurs, the computer stops the execution of program and displays an error message.

Character Set: A character denotes any alphabet, digit or special symbol, used to form words, numbers and expressions (or) represent information.

1. Letters - upper case A...Z, lower case a..z

2. Digits - all decimal digits 0..9

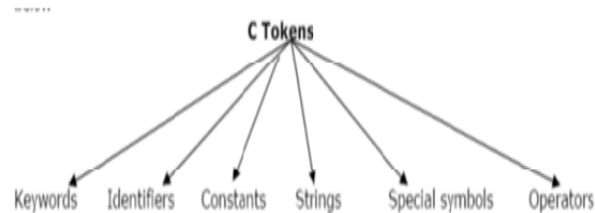
3. Special characters –

.	- Period	\$	- dollar sign
,	- comma	:	- colon
;	- semi colon	'	- apostrophe
”	- quotation mark	#	- number sign
+	- plus	(- left parenthesis
-	- minus)	- right parenthesis
*	- asterisk	{	- left brace
}	- right brace	[- left bracket
]	-right bracket	<	-left angle bracket
>	- right angle bracket		

4. White spaces - a) Blank space b) Horizontal tab c) Carriage return
d) New line e) Form feed etc.

Tokens: The smallest individual units in C program are called tokens. programs can be written by

using tokens and syntax of a language.



Keywords: Keywords are reserved words in C language. They have fixed meanings and these meanings cannot be changed. Keywords serve as basic building blocks for program statements. All keywords must be written in lowercase. Eg:- auto, long, char, short etc. C language contains 32 keywords

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

Identifiers: - Identifiers refer to the names of variables, functions and arrays. These are user-defined names. An identifier in C can be made up of letters, digits and underscore. Identifier supports both uppercase and lowercase letters, but we prefer lowercase letters in c language

The rules to be followed to frame an identifier:

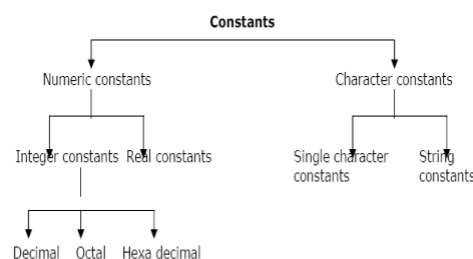
- The first character in the identifier should be a letter or _ and can be followed by any number of letters or digits or under score.
- No extra symbols are allowed other than letters, digits and _.
- The length of an identifier can be up to a maximum of 31 characters, but most of the compilers accept 8 characters only.
- Cannot use a keyword
- must not contain white space

Examples: **valid:** principle_amount
area_number_a_b

Invalid: sum of digits
sum-of-digits.

Constants: Constants in C refer to fixed values that do not change during the execution of a program.

C support several types of constants.



Integer constants:

An integer constant refers to a sequence of digits (or) an integer constant is a signed or unsigned whole number. C has three types of integer constants.

1. **Decimal integer constants**: set of digits from 0 to 9 with an optional + or – sign.

Example:-

Valid examples

Invalid examples

1. 123
2. -76565
3. 0
4. +736

1. 153 43
2. 20,000
3. \$667
4. 9-898

2. **Octal integer constants**: consists of any combination of digits from 0 to 7 with leading zero (0) (In C the first digit of an octal number must be a zero (0)).

Examples:

Valid examples

Invalid examples

1. 034
2. 07
3. 07655

1. 0X2
2. 0283
3. 0987

3. **Hexa decimal integer constants**: **Hexadecimal** integer constants can be any combination of digits 0 through 9 and alphabets from a to f or A to F. In C, a hexadecimal constant must begin with 0x or 0X (zero x) so as to differentiate it from a decimal number.

Examples:

Valid examples

Invalid examples

1. 0x2
2. 0x87665
3. 0xabc
4. 0x53647

1. 0xnmjk
2. 0x1286m
3. 08972
4. 0khjdi

Floating Constants (Real): The numbers containing fractional part is called real or floating-point constants. We have two notations

1. Decimal notation
2. Scientific notation

Decimal notation: It represents numbers having a whole number followed by decimal point and fractional part.

Example: 215
.95
-.75
+.5

Scientific notation: A real number may also be expressed in exponential (or) scientific notation.

Example: The number 215.65 may be written as 2.1565e2 in exponential notation. e2 means multiply by 10^2 .

Syntax: mantissa e exponent

Mantissa: mantissa is any real number expressed in decimal notation.

Exponent: It is an integer number with an optional sign.

E (or) e: It separates the mantissa and the exponent, that can be written in either uppercase (or) lower

case.

Example: 0.65e4
 12e-2
 1.5e+5
 3.18e3

Exponential notation is useful for representing the numbers that are either very large (or) very small in magnitude.

Example: 7500000000 is equals to 7.5e9 is also equals to 75e8
 -0.000000368 is equals to -3.68e-7

Single Character Constants:

A single character constant contains a single character enclosed by single quote marks is called single character constants.

Examples: 'a'
 '#'
 '2'
 ','

Points to remember:

1. The character constant '2' is not same as the number 2.
2. Blank space enclosed by single quote is also called as character constant.
3. Character constants have integer values known as ASCII values.

Example:

- 1) printf("%d",'a'); - output will be 97 which is the ASCII value of 'a'.
- 2) printf("%c",98); - output will be 'b'.

String Constants: A string constant is a sequence of characters enclosed by double quotes. The characters may be any letter, digits, special characters and blank space .

Example: "Welcome to c programming"
 "2001"
 "Well done"
 "34+23"
 "d"

Backslash character constants: Backslash character constants are used in output functions. Each backslash character constant contains two characters to represent a single character. These combinations are called escape sequences.

1. \a – Bell (beep)
2. \b – Backspace
3. \f – Form feed
4. \n – New line
5. \r – Carriage return
6. \t – Horizontal tab
7. \v – Vertical tab
8. \\ – Backslash
9. \' – Single quotation mark

10. \ " – Double quotation mark

11. \0 – A Null character

Data Types: The type of the value that a variable can store in the memory can be defined using data type. ANSI C supports four types of data types

- a) Primary (or) Primitive (or) basic (or) fundamental data type
- b) User-defined data type
- c) **void** data type
- d) Derived data type

Primitive Data Types: The primitive data types are also called primary data types (or) basic data type's. The Primitive data types of C Language are classified into following categories integer, character, floating point and double.

Integer: Integers are whole numbers with a range of values, range of values are machine dependent. Generally an integer occupies 2 bytes memory space and its value range limited to -32768 to +32767. A signed integer use one bit for storing sign and rest 15 bits for number. To control the range of numbers and storage space, C has three classes of integer storage namely short int, int and long int in both signed and unsigned forms.

Signed : signed short int
signed int
signed long int
Unsigned : unsigned short int
unsigned int
unsigned long int

A short int requires half the amount of storage than normal integer. Unsigned integers are always positive and use all the bits for the magnitude of the number. Therefore, the range of an unsigned integer will be from 0 to 65535. The long integers are used to declare a longer range of values and it occupies 4 bytes of storage space..

- For signed data type (-2^{n-1}) to $(+2^{n-1}-1)$
- For unsigned data type 0 to (2^n-1)

Size and range of integer data type:

Type	Size (bits)	Range	Place holder
Signed short int	8	-128 to 127	%d(or)%i
int or signed int	16	-2^{15} to $2^{15}-1$	%d(or)%i
long int (or) signed long int	32	-2^{31} to $2^{31}-1$	%ld
unsigned short int	8	0 to 255	%u
unsigned int	16	0 to 65535	%u
unsigned long int	32	0 to $2^{32}-1$	%lu

Character: Character type variable can hold a single character and are declared by using the keyword **char**. As there are signed and unsigned int (either short or long), in the same way there are signed and unsigned chars; both occupy 1 byte each, but having different ranges. Unsigned characters have values between 0 and 255, signed characters have values from -128 to 127

Size and range of character data type

Type	size (bits)	Range	Place holder
char (or) signed char	8	-128 to 127	%c
unsigned char	8	0 to 255	%c

Floating point: The float data type is used to store fractional numbers (real numbers) with 6 digits of precision. Floating point numbers are denoted by the keyword **float**. When the accuracy of the floating point number is insufficient, we can use the double to define the number.

Double: The double is same as float but with longer precision (14 digits of precision) and takes double space (8 bytes) than float.

Long double: long double is used for extended precisions. The size of the long double is 10bytes(80bits).

Size and range of integer data type

Type	Size (bits)	Range	Place holder
float	32	3.4E-38 to 3.4E+38	%f
double	64	1.7E-308 to 1.7 E+308	%f
long double	80	3.4E-4932 to 1.1 E+4932	%lf

User defined Data Types: we have two types

1. typedef
2. enumerator

typedef: C supports a feature known as "**type definition**" that allows user to define an identifier that would represent an existing data type. The user defined data type identifier can later be used to declare variables.

The general syntax of the typedef is shown below

Syntax: **typedef type identifier;**

or

typedef existing_data_type new_user_define_data_type;

typedef: typedef is a key word that allows the programmer to create a new name for an existing data type.

Type->Refers to an existing data type

Identifier->The new name given to the data type

Remember using typedef we are not creating new data types, we are creating only new names for the existing data type. These new data type's names are called user defined data types.

Example: typedef int units;
 typedef float marks;

Here units symbolizes int and marks symbolizes float, they can be later used to declare variables as follows.

```
units batch1, batch2;
marks name1[20], name2[20];
```

here batch1, batch2 are declared as int variable and name1[20], name2[20] declared as float array variables.

Advantages of typedef data type: The main advantage of typedef is that we can create meaningful

data type names for increasing the readability of the program.

Example :

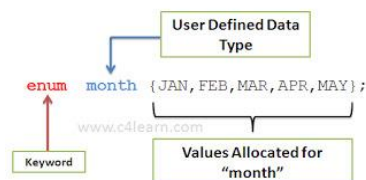
```
#include<stdio.h>
int main()
{
    typedef int number;
    number num1=40,num2=20;
    number answer;
    clrscr();
    answer=num1+num2;
    printf("answer %d",answer);
    getch();
    return 0;
}
```

Output: answer 60

Enumerated data Type: Enumerated Types allow us to create our own symbolic names for a list of related items. It is defined as

Syntax: `enum identifier {value1,value2,.....valuen};`

Example:



enum: It is the keyword which allows the user to create our own symbolic names for a list of ideas.

Identifier: It is the name of enumerated data type ,which can be used to declare variables that can have one of the variables enclosed within the braces(known as enumeration constants).

After definition we can declare the variables to be of this new type

enum identifier v1,v2...vn;

The enumerated variables v1,v2...vn can only have one of the value i.e value1,value2...valuen.

v1=value3;

v2=value1;

Example:

```
enum day{mon,tue,wed....sun};
enum day week_st, week_end;
week_st=mon;
week_end=fri;
```

The compiler automatically assign integer digits beginning with 0 to all enumeration constants, that is value1=0,value2=1.....however the programmer can change the default values.

```
enum day{ mon=0,tue=10,wed=20...sun=15};
```

The definition and declaration of enumerated variables can be combined in one statement.

Syntax: `enum identifier{value1,value2...valuen}v1,v2;`

Example:

```
#include <stdio.h>
enum week{ sunday, monday, tuesday, wednesday, thursday, friday, saturday};
int main()
```

```
{
    enum week today;
    clrscr();
    today=wednesday;
    printf("%d day",today+1);
    getch();
    return 0;
}
```

Output: 4 day

void data type:

- void data type is an empty data type
- It is normally used in functions to indicate that the function does not return any value
- No value is associated with this data type it does not occupy any space in the memory.

Derived Data Type:

Definition: Those data types which are derived from fundamental data types are called derived data types. The derived data types are shown below:

1. Arrays
2. Pointers
3. Structure
4. Union
5. function

Type Conversion: Converting the one data type into another data type is known as type casting or type conversion. Type conversions are divided into two types.

- 1) **Implicit (or) up casting (or) widening (or) automatic type conversion.**
- 2) **Explicit (or) down casting (or) narrowing type conversion.**

Implicit type conversion (Automatic type conversion): In this one data type is automatically converted into another type as per the rules described in c language, which means the lower level data type is converted automatically into higher level data type before the operation proceeds. The result of the data type having higher level data type.

Example: int x = 123;
 double y = x;

In the above statement, the conversion of data from int to double is done implicitly, in other words programmer don't need to specify any type operators. Widening is safe because there will not be any loss of data. This is the reason even though the programmer does not use the cast operator the compiler does not complaint because of lower data type is converting into higher data type. Here higher data type having the much more space to store the lower data type.

Explicit type conversion: Converting higher data type into lower data type is called narrowing.

Here we can place intended data type in front of the variable to be cast.

Example: double d=12.67853;
 int n = (int) d;

Here we are converting higher level data type into lower level data type that means double type is

converted into int type, the fractional part of the number is lost and only 12 is stored in n. Here we are losing some digits this is the reason the compiler forces the programmer to use the cast operator when going for explicit casting.

Type conversion makes both operands belongs to similar data type. The following rules that are applied while evaluating the arithmetic expression

1. if one of the operands is long double, double (or) float, the other will be converted to long double, double(or)float and the result will be long double, double (or) float respectively.
2. Otherwise, if either operand is unsigned long int, the other is converted to unsigned long int and the result will be unsigned long int.
3. Otherwise, if one operand is long int and other is unsigned int, then
 - a) If unsigned int can be converted to long int, the unsigned int operand will be converted as such and the result will be long int;
 - b) Else, both operands will be converted to unsigned long int and the result will be unsigned long int.
4. Else, if the one of the operands is long int, the other will be converted to long int and the result will be long int.
5. Else, if one of the operands is unsigned int, the other will be converted to unsigned int and the result will be unsigned int.

Variables:

Definition: A variable is a data name that may be used to hold a data value. A variable may take different values at different times during the program execution.

(or)

Definition: A Variable is defined as the name given to a memory location where the data can be stored, accessed or manipulated.

(or)

Definition: Variable is an identifier which is used to store a value. The value stored in the variable may change during execution of the program.

Rules for forming a variable:

1. The starting character should be a letter (or) underscore. The first character may be followed by a sequence of letters or digits.
2. ANSI Standard recognizes a length of 31 characters. However the length should not be normally more than 8 characters because only the first 8 characters are significant by many compilers
3. Upper and lower case are significant. Example: TOTAL is not same as total or Total.
4. The variable should not be a keyword.
5. White space is not allowed.
6. Special characters except _ (underscore) symbol are not allowed.

Examples:**Valid**

john
value
tot_amt
a1

Invalid

1868 - (The first character should be a letter)
(area) - (Special characters are not allowed)
27th - (The first character should be a letter)
% - (Special characters are not allowed)

Declaration of variables:

1. After designing suitable variable names, we must declare them to the compiler. Declaration does two things.
 - i) It tells the compiler what the variable name is
 - ii) It specifies what type of data the variable will hold.
2. All the variables should be declared at the beginning of the program. This way of declaring all the variables at the beginning of the program is called type declarations or declaration of variables.
3. The syntax to declare a variable is shown below

data_type v₁, v₂ ,----, v_n;

Where data_type is any of the basic data type such as int, float, char, double etc

v₁, v₂ ,----, v_n are the variables. all the variables should be separated by commas and finally should end with semicolon;

Example:

1. int count;
2. int a,b;
3. float area,volume;

Assignment statements:

Values can be assigned to variables using the assignment operator = as follows

datatype variable _name =constant or expression or variable;

Examples: int final=100;
 char yes='x';
 double balance=39.98;

Scope of a variable: The scope of a variable determines over what part(s) of the program a variable is actually available for use (active).

The scope of any variable can be broadly categorized into two categories

1. Local scope
2. Global scope

Local scope: When variable is defined inside a function (or) a block, then it is locally accessible within the block, which means we can access the variable with in the function only, hence it is a local variable.

Example: #include<stdio.h>
 int main()
 {
 int m = 22, n = 44; // m, n are local variables of main function
 /*m and n variables are having scope within this main function only.
 These are not visible to test funtion.*/

```

        /* If you try to access a and b in this function, you will get 'a' undeclared
           and 'b' undeclared error */
        clrscr();
        printf("\nvalues : m = %d and n = %d", m, n);
        test();
        getch();
    }
    test()
    {
        int a = 50, b = 80;    // a, b are local variables of test function
        /*a and b variables are having scope within this test function only.
           These are not visible to main function.*/
        /* If you try to access m and n in this function, you will get 'm'
           undeclared and 'n' undeclared error */
        printf("\n values : a = %d and b = %d", a, b);
    }

```

Output: values: m = 22 and n = 44
values: a = 50 and b = 80

Global scope: When variable is defined outside all functions, then it is available to all the functions of the program, means we can access the variable anywhere in the program (or) the variable is visible to main function and all other sub functions (variable is available for all the blocks of the program), hence it is a global variable.

Example1: `#include<stdio.h>`

```

int m = 22, n = 44; // global variables declared
int a = 50, b = 80; // global variables declared
int main()
{
    clrscr();
    printf(" All variables are accessed from main function");
    printf("\n values: m=%d: n=%d: a=%d: b=%d", m,n,a,b);
    test();
    getch();
}
test()
{
    printf("\n\n All variables are accessed from test function");
    printf("\n values: m=%d: n=%d: a=%d: b=%d", m,n,a,b);
}

```

Output: All variables are accessed from main function
Values: m = 22: n = 44: a = 50: b = 80
All variables are accessed from test function
Values: m = 22: n = 44: a = 50: b = 80

Example2: `#include<stdio.h>`

```

int global = 100;    // global variable declared
void main()
{

```

```

        int local = 10;      // local variable declared
        printf("Global variable is %d",global);
        printf("Local variable is %d",local);
        func1();
    }
    void func1()
    {
        printf("Global inside func1 is %d",global); /*Would print the global
                                                    variable successfully.*/
        printf("Local inside func1 is %d", local); /*It would produce an error,
                                                    because local is a local variable and can
                                                    be accessed only inside main function. */
    }

```

Output: Global variable is 100
 Local variable is 10
 Global inside func1 is 100

Operators: An operator is a symbol, which performs operation on operands.
 (Or)

An operator is a symbol that tells the computer to perform certain mathematical (or) logical operations on operands.

The operators in C language can be classified based on the number of operands and the type of operation being performed. The operators are classified into three major categories based on the number of operands as shown below

1. Unary operators
2. Binary operators
3. Ternary operators

Unary Operators: An operator which performs operation on single operand is called unary operator.

Example: a++,a--,--a,++a etc

Binary Operator: An operator which performs operation on two operands is called a binary operator.

In binary operation the operator is in between two operands.

Example: a+b, a*b, a/b, a-b etc

Ternary Operator: An Operator which performs operation on three operands is called a ternary operator. Ternary operator is also called conditional operator.

Example: a > b ? c : d

C operators can be classified into eight categories as shown below

1. **Arithmetic Operators**
2. **Assignment Operators**
3. **Increment / Decrement operators**
4. **Relational Operators**
5. **Logical Operators**
6. **Conditional Operators**
7. **Bitwise Operators**
8. **Special Operators**

Arithmetic Operators:

The operators that are used to perform **Arithmetic** operations such as addition, subtraction,

multiplication, ----etc are called **Arithmetic** operators. The modulus operation is denoted by % is used only for integer values and not for the floating point numbers

Operator	Description	Example
+	addition or unary plus	a+b
-	subtraction or unary minus	a-b
*	multiplication	a*b
/	division	a/b
%	remainder after division (modulo division)	a%b

Example:

```
#include <stdio.h>
int main()
{
    int a=9,b=4,c;
    clrscr();
    c=a+b;
    printf("a+b=%d\n",c);
    c=a-b;
    printf("a-b=%d\n",c);
    c=a*b;
    printf("a*b=%d\n",c);
    c=a/b;
    printf("a/b=%d\n",c);
    c=a%b;
    printf("Remainder when a divided by b=%d\n",c);
    getch();
    return 0;
}
```

Output:

```
a+b=13
a-b=5
a*b=36
a/b=2
Remainder when a divided by b=1
```

Integer Arithmetic: When both the operands in a single arithmetic expression such as a+b are integers, the expression is called as integer expression and the operation is called integer arithmetic

if a=4, b=2, then

```
a+b = 6
a-b = 2
a*b = 8
a/b = 2
a%b = 0
```

Real Arithmetic: An arithmetic operation involving only real operands is called real arithmetic. A real operand may assume values either in decimal or exponential notation.

If a, b and c are floats then, $a = 60/70 = 0.857143$

$$b = 1.0/3.0 = 0.33333$$

$$c = -2.0/3.0 = -0.66667$$

Mixed-mode Arithmetic: When one of the operands is real and the other is integer the expression is called a mixed-mode arithmetic expression.

Example: $15/10.0=1.5$ & $15.0/1.0$

Assignment operators:

These are used to assign the values for the variables in C programs. The most common assignment operator is =. The syntax is shown below: data type variable_name = expression;

Example: 1. $a = b$; 2. $\text{Area} = \text{length} * \text{breadth}$; 3. $A = 14$;

Operator	Description	Example	Same as
=	Assignment	$a=b$	$a=b$
+=	Addition assignment	$a+=b$	$a=a+b$
-=	Subtraction assignment	$a-=b$	$a=a-b$
=	Multiplication assignment	$a=b$	$a=a*b$
/=	Division assignment	$a/=b$	$a=a/b$
%=	Modula division assignment	$a\%=b$	$a=a\%b$

Example1:

```
#include <stdio.h>
main()
{
    int a = 21;
    int c ;
    clrscr();
    c = a;
    printf("Value of c = %d\n", c );
    c += a;
    printf("Value of c = %d\n", c );
    c -= a;
    printf("Value of c = %d\n", c );
    c *= a;
    printf("Value of c = %d\n", c );
    c /= a;
    printf("Value of c = %d\n", c );
    c=22;
    c %= a;
    printf("Value of c = %d\n", c );
    getch();
}
```

Output: Value of c = 21
Value of c = 42
Value of c = 21
Value of c = 441
Value of c = 21
Value of c = 1

Example2: Write a C program to swap two numbers using third variable.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a,b,c;
    clrscr();
    printf("enter values of a,b");
    scanf("%d%d",&a, &b);
    c=a;
    a=b;
    b=c;
    printf("Swapped values are %d,%d",a,b);
    getch();
}
```

Example3: Write a C program to swap two numbers without using third variable.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a,b;
    printf("enter values of a,b");
    scanf("%d%d",&a,&b);
    a=a-b;
    b=a+b;
    a=b-a;
    printf("Swapped values are %d,%d",a,b);
    getch();
}
```

Increment/Decrement Operators: Increment operators are used to increase the value of the variable by one and decrement operators are used to decrease the value of the variable by one in C programs.

Syntax: Increment operator: ++var_name; (or) var_name++;
Decrement operator: --var_name; (or) var_name --;

Increment Operators:

1. ++ is an increment operator. This is an unary operator.
2. It increments the value of the operand by one.

Post Increment (Eg: a++)

Pre Increment (Eg: ++a)

If the increment operator ++ is placed after the operand, then operator is called Post Increment. As the name indicates Post indicates increment after the operand value is used.

Example: void main()

```
{
    int a=20;
    a++;
    printf("a value is=%d",a);
}
```

Output: a=21

If the increment operator ++ is placed before the operand, then operator is called Pre Increment.

As the name indicates Pre indicates , increment before the operand value is used.

Example: void main()

```
{
    int a=20;
    int b=++a;

    printf("%d %d",a,b);
}
```

Output: 21 21

a=20, b=++a=21

But output is : a=21, b=21

Decrement Operators:

- ✓ -- is a decrement operator. This is an unary operator.
- ✓ It decrements the value of the operand by one.
- ✓ The decrement operator is classified into two categories:
 1. Post Decrement ==> a- -
 2. Pre Decrement ==> - -a

If the decrement operator -- is placed after the operand, then the operator is called post decrement. So the operand value is used first and then the operand value is decremented by 1.

Example: void main()

```
{
    int a=20;
    a--;
    printf("%d",a);
}
```

Output: 19

If the decrement operator -- is placed before the operand then the operator is called Pre decrement. So the operand value is decrement by 1 first and this decremented value is used.

Example: void main()

```
{
    int a = 20;
    int b = --a;
    printf("%d%d", a, b);
}
```

Output: 19 19

Example: #include<stdio.h>

#include<conio.h>

void main()

```
{
    int p,q,x,y;
    clrscr();
    printf("enter the value of x\n");
    scanf("%d",&x);
    printf("enter the value of y\n");
    scanf("%d",&y);
    printf("x=%d\n y=%d\n",x,y);
    p=x++;
    q=y++;
    printf("x=%d\t y=%d\n",x,y);
    printf("p=%d\t q=%d\n",p,q);
    p=++x;
```

```

    q=++y;
    printf("x=%d\t y=%d\n",x,y);
    printf("p=%d\t q=%d\n",p,q);
    p--x;
    q--y;
    printf("x=%d\t y=%d\n",x,y);
    printf("p=%d\t q=%d\n",p,q);
    p=x--;
    q=y--;
    printf("x=%d\t y=%d\n",x,y);
    printf("p=%d\t q=%d\n",p,q);
    getch();
}

```

Output:

```

enter the value of x    10
Enter the value of y    20
X=10
Y=20
X=11 Y=21
P=10 Q=20
X=12 y=22
P=12 q=22
X=11 Y=21
P=11 Q=21
X=10 y=20
P=11 q=21

```

Relational Operators: These operators are used to compare the value of two variables. Relational operator's checks relationship between two operands. If the relation is true, it returns value 1 and if the relation is false, it returns value 0.

Example: $a > b$. Here, $>$ is a relational operator. If a is greater than b , $a > b$ returns 1 if not then, it returns 0.

Operator	Description	Example
==	Equal to	$5 == 3$ returns false (0)
>	Greater than	$5 > 3$ returns true (1)
<	Less than	$5 < 3$ returns false (0)
!=	Not equal to	$5 != 3$ returns true(1)
>=	Greater than or equal to	$5 >= 3$ returns true (1)
<=	Less than or equal to	$5 <= 3$ return false (0)

Example:

```

#include <stdio.h>
#include <conio.h>
void main()
{
    int a = 21, b = 10, c ;
    clrscr();
    if( a == b )
    {
        printf("a is equal to b\n" );
    }
}

```

```
    }
    else
    {
        printf("a is not equal to b\n" );
    }
    if ( a < b )
    {
        printf("a is less than b\n" );
    }
    else
    {
        printf(" a is not less than b\n" );
    }
    if ( a > b )
    {
        printf("a is greater than b\n" );
    }
    else
    {
        printf("a is not greater than b\n" );
    }
    /* Lets change value of a and b */
    a = 5;
    b = 20;
    if ( a <= b )
    {
        printf("a is either less than or equal to b\n" );
    }
    else
    {
        printf("a is not either less than or equal to b\n" );
    }
    if ( b >= a )
    {
        printf("b is either greater than or equal to b\n" );
    }
    else
    {
        printf("b is not either greater than or equal to b\n" );
    }
    getch();
}
```

Output: a is not equal to b
a is not less than b
a is greater than b
a is either less than or equal to b
b is either greater than or equal to b

Logical Operators: These operators are used to perform logical operations on the given two variables.

Logical operators are used to combine expressions containing relation operators. In C, there are 3 logical operators.

Example: if (a==b && b==c)

Operator	Description	Example
&&	Logical AND	If c=5 and d=2 then, ((c==5) && (d>5)) returns false.
	Logical OR	If c=5 and d=2 then, ((c==5) (d>5)) returns true.
!	Logical NOT	If c=5 then, !(c==5) returns false.

Example: #include <stdio.h>

```
int main()
{
    int m=40,n=20;
    int o=20,p=30;
    if (m>n && m!=0)
    {
        printf("true\n");
    }
    else
    {
        printf("false\n");
    }
    if (o>p || p!=20)
    {
        printf(" true\n");
    }
    else
    {
        printf(" false\n");
    }
    if (!(m>n && m !=0))
    {
        printf("true\n");
    }
    else
    {
        printf("false");
    }
}
```

Output: true true false

Conditional Operators:

1. The conditional operator is also called a ternary operator. As the name indicates an operator that operates on three operands is called ternary operator.
2. The ternary operators consists of two symbols i.e ? and :

Syntax: (condition ? true_value : false_value);

Where condition is an expression evaluated True or False. If condition is evaluated to True, the true_value is executed. If condition is evaluated to False, the false_value is executed.

Example: (a>b) ? a : b;

Example1: #include<stdio.h>

```
void main()
{
    int a,b,result,choice;
    clrscr();
    printf("enter first number \n");
    scanf("%d",&a);
    printf("enter second number \n");
    scanf("%d",&b);
    printf("enter 1 for addition or 2 for multiplication");
    scanf("%d",&choice);
    result=(choice==1)?a+b:(choice==2)?a*b:printf("invalid input");
    if(choice==1||choice==2)
    {
        printf("the result is %d",result);
        getch();
    }
}
```

Output: Enter first number=10
Enter second number=10
Enter 1 for addition or 2 for multiplication
1
The result is 20

Example2: #include<stdio.h>

#include<conio.h>

void main()

```
{
    int a = 10, b = 11;
    int c;
    c = (a < b)? a : b;
    printf("%d", c);
}
```

Output:10

Example3: Write a C program to print biggest from three numbers using conditional operator.

#include<stdio.h>

#include<conio.h>

void main ()

```
{
    int a,b,c,max;
    clrscr();
    printf("enter the a,b,c values");
    scanf("%d%d%d",&a,&b,&c);
    max=(a>b&&a>c?a:b>c?b:c);
    printf("biggest no is%d",max);
    getch();
}
```


Example4: Write a C program to check whether a given number is even or odd using conditional operator.

```
#include<stdio.h>
#include<conio.h>
void main ()
{
    int n;
    clrscr();
    printf("enter a number");
    scanf("%d",&n);
    (n%2==0)?printf("even"):printf("odd");
    getch();
}
```

Bitwise Operator: A bitwise operator works on each bit of data. Bitwise operators are used in bit level programming

x	y	x y	x & y	x ^ y	Operator	Description
0	0	0	0	0	&	Bitwise AND
0	1	1	0	1		Bitwise OR
1	0	1	0	1	~	Bitwise NOT
1	1	1	1	0	^	XOR
					<<	Left Shift
					>>	Right Shift

Example:

```
#include <stdio.h>
main()
{
    int a = 60;           /* 60 = 0011 1100 */
    int b = 13;           /* 13 = 0000 1101 */
    int c = 0;
    c = a & b;             /* 12 = 0000 1100 */
    printf("Value of c is %d\n", c );
    c = a | b;             /* 61 = 0011 1101 */
    printf("Value of c is %d\n", c );
    c = a ^ b;             /* 49 = 0011 0001 */
    printf("Value of c is %d\n", c );
    c = ~a;                /* -61 = 1100 0011 */
    printf("Value of c is %d\n", c );
    c = a << 2;            /* 240 = 1111 0000 */
    printf("Value of c is %d\n", c );
    c = a >> 2;            /* 15 = 0000 1111 */
    printf("Value of c is %d\n", c );
}
```

Output: Value of c is 12 Value of c is 61 Value of c is 49
Value of c is -61 Value of c is 240 Value of c is 15

Special Operators:

S.no	Operators	Description
------	-----------	-------------

1	&	This is used to get the address of the variable. Example : &a will give address of a.
2	*	This is used as pointer to a variable. Example : * a where, * is pointer to the variable a.
3	Sizeof ()	This gives the size of the variable. Example : size of (char) will give us 1.
4	,	Comma Operator

Example program for & and * operators in C:

In this program, “&” symbol is used to get the address of the variable and “*” symbol is used to get the value of the variable that the pointer is pointing to.

```
#include <stdio.h>
int main()
{
    int *ptr, q;
    q = 50;
    /* address of q is assigned to ptr */
    ptr = &q;
    /* display q's value using ptr variable */
    printf("%d", *ptr);
    return 0;
}
```

Output:50

Example program for sizeof() operator in C:

sizeof() operator is used to find the memory space allocated for each C data types.

```
#include <stdio.h>
#include <limits.h>
int main()
{
    int a;
    char b;
    float c;
    double d;
    printf("Storage size for int data type:%d \n",sizeof(a));
    printf("Storage size for int data type:%d \n",sizeof(int));
    printf("Storage size for char data type:%d \n",sizeof(b));
    printf("Storage size for char data type:%d \n",sizeof(float));
    printf("Storage size for float data type:%d \n",sizeof(c));
    printf("Storage size for float data type:%d \n",sizeof(char));
    printf("Storage size for double data type:%d\n",sizeof(d));
    printf("Storage size for double data type:%d \n",sizeof(double));
    return 0;
}
```

Output: Storage size for int data type:2
 Storage size for int data type:2
 Storage size for char data type:1
 Storage size for char data type:1
 Storage size for float data type:4
 Storage size for float data type:4
 Storage size for double data type:8

Storage size for double data type:8

Comma operator: It is special kind of operator which is widely used in programming to separate the declaration of multiple variables. Comma Operator has Lowest Precedence i.e it is having lowest priority so it is evaluated at last. Comma operator returns the value of the rightmost operand when multiple comma operators are used inside an expression.

Comma Operator Can acts as –**Operator:** In the Expression

Separator : Declaring Variable , In Function Call Parameter List

Example:

```
#include<stdio.h>
void main()
{
    int num1 = 1, num2 = 2;
    int res;
    res = (num1, num2);
    printf("%d", res);
}
```

Usage of Comma Operator: Consider above example

Comma as Separator: int num1 = 1, num2 = 2;
It can acts as Seperator in – Function calls
Function definitions
Variable declarations
Enum declarations
Comma as Operator

res = (num1, num2);

In this case value of rightmost operator will be assigned to the variable. In this case value of num2 will be assigned to variable res.

Examples of comma operator:

Comma Operator have lowest priority in C Programming Operators. All possible operations that can be performed on comma operator are summarized below –

Verity1:

```
#include<stdio.h>
int main()
{
    int i;
    i = 1,2,3;
    printf("i:%d\n",i);
    return 0;
}
```

Output: i:1

Explanation: i = 1,2,3;

1. Above Expression contain 3 comma operator and 1 assignment operator.
2. If we check precedence table then we can say that “Comma” operator has lowest precedence than assignment operator
3. So Assignment statement will be executed first .
4. 1 is assigned to variable “i”

Verity2: Using Comma Operator with Round Braces

```
#include<stdio.h>
int main()
{
    int i;
    i = (1,2,3);
    printf("i:%d\n",i);
    return 0;
}
```

Output: i:3**Explanation:** i = (1,2,3);

- Bracket has highest priority than any operator.
- Inside bracket we have 2 comma operators.
- Comma operator has associativity from Left to Right.
- Comma Operator will return Rightmost operand

i = (1,2,3)

==> 1,2 will return 2

==> 2,3 will return 3

==> means (1,2,3) will return 3

==> Assign 3 to variable i.

Verity3: Using Comma Operator inside printf statement

```
#include<stdio.h>
#include< conio.h>
void main()
{
    clrscr();
    printf("Computer","Programming");
    getch();
}
```

Output: Computer

You might feel that answer of this statement should be “Programming” because comma operator always returns rightmost operator, in case of printf statement once comma is read then it will consider preceding things as variable or values for format specifier.

Verity4: Using Comma Operator inside Switch cases.

```
#include<stdio.h>
#include< conio.h>
void main()
{
    int choice = 2 ;
    switch(choice)
    {
        case 1,2,1:
            printf("\nAllas");
            break;
        case 1,3,2:
            printf("\nBabo");
            break;
        case 4,5,3:
            printf("\nHurray");
    }
```

```

        break;
    }
}
Output: Babo
Verity5: Using Comma Operator inside For Loop
#include<stdio.h>
int main()
{
    int i,j;
    for(i=0,j=0;i<5;i++)
    {
        printf("\nValue of J : %d",j);
        j++;
    }
    return (0);
}

```

Output: Value of j : 0
Value of j : 1
Value of j : 2
Value of j : 3
Value of j : 4

Example: #include <stdio.h>
int main ()
{
 int i = 10, b = 20, c = 30;
 i = b, c;
 printf("%i\n", i);
 i = (b, c);
 printf("%i\n", i);
 return 0;
}

Output: 20
30

The precedence of operators: (Hierarchy of Operator)

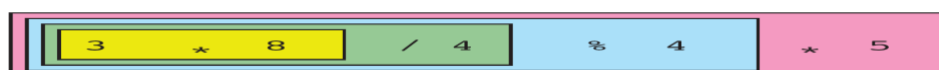
In c language each operator is associated with priority value based on the priority, the expressions are evaluated. The priority of each operator is Pre-defined.

Operator Precedence: When two operators share an operand the operator with the higher *precedence* goes first. For example, $1 + 2 * 3$ is treated as $1 + (2 * 3)$, whereas $1 * 2 + 3$ is treated as $(1 * 2) + 3$ since multiplication has a higher precedence than addition.

Associativity: When an expression has two operators with the same precedence, the expression is evaluated according to its associativity. It is divided into two types

Left-To-Right: Here we can evaluate the expression from left to right.

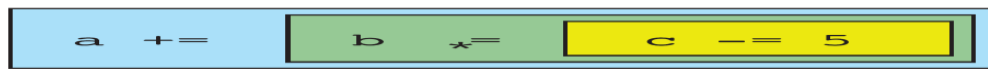
Example: $3 * 8 / 4 \% 4 * 5$



Hear all the operators having the same precedence so the associativity is Left to Right

Right-To-Left: Here we can evaluate the expression from right to left.

Example: $a += b * c -= 5$.



OPERATOR	DESCRIPTION	ASSOCIATIVITY	RANK
()	Inter most brackets/Function calls	L→R	1
[]	Array element reference	L→R	2
++, --, !, sizeof, ~, +, -, &, *	Unary Operator	R→L	3
∀ or →	Member Access	L→R	4
*, /, %, +, -	Arithmetic	L→R	5
<<, >>	Shift Operator	L→R	6
<, <=, >, >=	Relational	L→R	7
==, !=	Equality	L→R	8
~, <<, >>, &, ^,	Bitwise	L→R	9
&&,	Logical	L→R	10
?:	Conditional	R→L	11
=, +=, -=, *=, /=, &=, <<=, ^=, =	Assignment	R→L	12
,	Comma	L→R	13

Expressions: An expression in C consists of a syntactically valid combination of operators and operands that computes to a value. The constants and variables in an expression are called operands. An expression by itself is not a statement. Remember, a statement is terminated by a semicolon; an expression is not.

❖ An expression is a simple or complex.

A simple expression contains only one operator.

- Ex: $3+6$, $-a$

A complex expression contains more than one operator.

- Ex: $2*5+7-8$

❖ An Expression always reduces to a single value

Example: $3 * 4 \% 5 \rightarrow$ is an expression with value 2.

$x = 3 * 4 \rightarrow$ This statement is an example of an assignment expression. (Note the absence of the semicolons in the assignment above. The terminating semicolon would have converted the expression into a statement.)

- To evaluate an expression we use BODMAS Rule
- During Evolution:
 - B = Brackets
 - O = Of operator
 - D = Division
 - M = Multiplication
 - A = Addition
 - S = Subtraction

Arithmetic expression: An expression that contains arithmetic operators is called arithmetic expression

Example:

- ❖ $(6 + 4) \times 5$
First solve inside 'brackets' $6 + 4 = 10$, then $10 \times 5 = 50$
- ❖ $15 \div 3 \times 1 \div 5$
'Multiplication' and 'Division' perform equally, so calculate from left to right side. First solve $15 \div 3 = 5$, then $5 \times 1 = 5$, then $5 \div 5 = 1$
- ❖ $7 + 19 - 11 + 13$
'Addition' and 'Subtraction' perform equally, so calculate from left to right side. First solve $7 + 19 = 26$, then $26 - 11 = 15$ and then $15 + 13 = 28$.

Relational Expression: An expression that contains relational operators is called Relational Expression.

Example: $a + b > b * b - 4 * a * c == 0$

Step1 $\rightarrow b * b$, step2 $\rightarrow 4 * a$, step3 \rightarrow step2 * c, step4 $\rightarrow a + b$, step5 \rightarrow step2 - step3, step6 \rightarrow step4 > step5, step7 \rightarrow step6 == 0.

Logical Expression: The logical operators are used to combine two or more relational expressions.

Example: $a > b \ \&\& \ a + b > c$

Step1 $\rightarrow a + b$, step2 $\rightarrow a > b$, step3 \rightarrow step1 > c, step4 \rightarrow step2 && step3

Rules for Evolution of Expressions:

1. First parenthesized sub expressions from left to right are evaluated.
2. If parentheses are nested, the evaluation begins with the inner most sub-expression.
3. The precedence rule is applied in determining the order of application of operators in evaluating sub expression.
4. The association rule is applied when two or more operators of the same precedence level appear in a sub-expression
5. Arithmetic expressions evaluated from left to right using the rules of precedence.
6. When parentheses are used, the expressions within parenthesis assume highest priority.

Examples:

1. $X = 5 - 2 * 7 - 9$

Answer: step1: $2 * 7 = 14$

Step2: $5 - 14 = -9$

Step3: $-9 - 9 = -18$

Step4: $x = -18$

2. $X = 2 * 3 + 4 * 5$

Answer: step1: $2 * 3 = 6$

Step2: $4 * 5 = 20$

Step3: $6 + 20 = 26$

Step4: $x = 26$

3. $X=2*(3+4)*5$

Answer: step1:3+4=7
 Step2:7*2=14
 Step3:14*5=70
 Step4:x=70

4. $X=7*6\%15/9$

Answer: Sep1:7*6=42
 Step2:42%15=12
 Step3:12/9=1
 Step4:x=1

5. $X=7*(6\%15)/9$

Answer: Step1:6%15=6
 Step2:6*7=42
 Step3:42/9=4
 Step4:x=4

6. $7*6\%(15/9)$

Answer: Step1:15/9=1
 Step2:7*6=42
 Step3:42/1=0
 Step4:x=0

7. $7*((6\%15)/9)$

Answer: Step1:6%15=6
 Step2:6/9=0
 Step3:7*0=0
 Step4:x=0

Program: Source code to find ASCII value of a character entered by user

```
#include <stdio.h>
#include <conio.h>
void main()
{
    char c;
    printf("Enter a character: ");
    scanf("%c",&c);    /* Takes a character from user */
    printf("ASCII value of %c = %d",c,c);
    getch();
}
```

Output: Enter a character: G
 ASCII value of G = 71

Program: C Program to compute remainder and quotient


```
#include <stdio.h>
#include<conio.h>
void main()
{
    int dividend, divisor, quotient, remainder;
    printf("Enter dividend: ");
    scanf("%d",&dividend);
    printf("Enter divisor: ");
    scanf("%d",&divisor);
    quotient=dividend/divisor;
    remainder=dividend%divisor;
    printf("Quotient = %d\n",quotient);
    printf("Remainder = %d",remainder);
    getch();
}
```

Output: Enter dividend: 25
Enter divisor: 4
Quotient = 6
Remainder = 1

program: C program to convert given number of days to a measure of time given in years, weeks and days. For example 375 days is equal to 1 year 1 week and 3 days (ignore leap year)

```
#include <stdio.h>
#include<conio.h>
#define DAYSINWEEK 7
void main()
{
    int ndays, year, week, days;
    printf("Enter the number of daysn");
    scanf("%d", &ndays);
    year = ndays / 365;
    week =(ndays % 365) / DAYSINWEEK;
    days =(ndays % 365) % DAYSINWEEK;
    printf ("%d is equivalent to %d years, %d weeks and %d
            daysn",ndays, year, week, days);

    getch();
}
```

Output1: Enter the number of days 29
29 is equivalent to 0 years, 4 weeks and 1 days

Output2: Enter the number of days 1000
1000 is equivalent to 2 years, 38 weeks and 4 days

Program: C Program to Extract Last two Digits of a given Year.

```
#include <stdio.h>
#include<conio.h>
void main()
{
    int year, yr;
    printf("Enter the year ");
    scanf("%d", &year);
    yr = year % 100;
```

```
printf("Last two digits of year is: %02d", yr);  
getch();  
}
```

Output: Enter the year 2012
Last two digits of year is: 12

Input & Output Operations: Reading, processing and writing of data are the three essential functions of a computer program. Most programs take some data as input and display the processed data, known as results. We have two methods for providing data to the program variables.

1. One method is to assign the values to variables through the assignment statement such as `x=5;a=0;` and
2. Another method is to use the input functions like `scanf`, `getchar`, `gets` which can read data from a keyboard. For outputting the results we have used the function `printf`, `putchar`, `puts` which sends results out to a terminal.

C language provides a collection of functions and can be used in writing our programs. Among these, a set of functions, which support reading data from the input devices and writing data on to the user screen. These functions are known as standard I/O library. the statement is as follows

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

It tells the compiler to search for the file called `stdio.h` and place its contents at the required place in the program. An input and output function comes under two categories namely formatted functions and unformatted functions.

Formatted functions:	input:	<code>scanf()</code>
	output:	<code>printf()</code>
Unformatted functions:	input:	<code>getchar()</code>
		<code>getch()</code>
		<code>gets()</code>
	output:	<code>putchar()</code>
		<code>putch()</code>
		<code>puts()</code>

Formatted functions: It refers to an input data that has been arranged in a particular format

Input functions:

Scanf():

1. Formatted input refers to an input that has been arranged in a particular format.
2. We use `scanf` function for formatted input.
3. `scanf()` is included in header file “`stdio.h`”.
4. The syntax is: `scanf (“control string”, arg1, arg2,.....,argn);`
5. Here the **control string** specifies what type of data values need to be read from the user (or) keyboard and the arguments. `arg1`, `arg2` ...`argn` specifies the address of the locations where the data is stored. Control string and arguments are separated by commas.
6. Control string is also known as **format string**, it contains field specifications (place holders)

which direct the interpretation of input data.

7. Field specifications consisting of **conversions characters %**, a **data type character** and an **optional number** specifying the field width.
8. The field specification for reading an integer number is %d. Here % indicates that conversion specification follows d is known as data type character.
9. The field specification for reading a floating number is %f
10. The field specification for reading character string: %s or %c

Rules for scanf function:

1. Each variable to be read must have a format specification
2. For each format specification, there must be a variable address of proper type
3. Never end the format string with white spaces, it is a data error
4. Any non-white space character used in the format string must have a matching character in the user input
5. The scanf reads until:
 - A white space character is found in a numeric specification
 - The max number of characters have been read (or) an error is detected (or) the end of file is reached.

Inputting integer numbers: The field specification for reading integer number is

Syntax: %wd

The % sign indicates that a conversion character, w is an integer number that specifies the field width of the number to be read and d, known as data type character, indicates that the number to be read is in integer mode.

Example: scanf(“%2d %5d”,&num1,&num2);

Input: 50 31465

The value 50 is assigned to num1 and 31465 is assigned to num2.

Input: 31465 50

The value 31 will be assigned to num1 and 426 will be assigned to num2. The value 50 is unread, will be assigned to the first variable in the next scanf call.

Inputting Real numbers: The field width of real numbers is not to be specified and therefore scanf reads real numbers using the simple specification %f for both the notations, namely decimal point notation and exponential notation

Example: scanf(“%f %f %f”,&x,&y,&z);

Input: 475.89 43.21e-1 678

The 475.89 will be assigned to x, 4.321 to y and 678.0 to z. The input field specifications may be separated by any arbitrary blank spaces.

Inputting Character Strings numbers: The notation for reading character strings is

 %ws (or) %wc

Reading mixed datatypes: It is possible to use one scanf statement to input a data line containing mixed mode data. In such cases to ensure that the input data items match the control specifications in order and type.

Example: scanf(“%d%c%f%s”,&count,&code,&ratio,&name);

Input: 15 p 1.575 coffee

For example if you give input like 15 p coffee 1.575 The scanf function would encounter a string when it was expecting a floating point value ,and would therefore terminate its scanf after reading the first value.

Formatted output:

printf():

- ✓ For formatted output we will use printf statement.
- ✓ printf() is included in header file “stdio.h”
- ✓ The syntax is: **printf (“Control string “, arg1,arg2,.....argn);**
- ✓ The control string consist of three types of items:
 - Characters that will be printed on the screen as they appear.
 - Format specifications that define the output format for display of each item.
 - Escape square characters such as in \n, \t and \b
- ✓ The control string indicates how many arguments follow and what their types.
- ✓ The arguments arg1, arg2argN are the variables and whose values are formatted and printed according to the specifications of the control string.

The printf() function can be more viewable by giving proper spacing(or)width.

Output Integer Numbers:The format specification for printing an integer number is

Syntax: %wd

Where w specifies minimum field width for the output and d specifies that the value to be printed is an integer.

Example: **input:** int a=143;
 Printf(“%d”,a);

Output:

1	4	3
---	---	---

Example: **input:** int a=143;
 Printf(“%2d”,a);

Output:

1	4	3
---	---	---

❖ If a number is greater than the specified field width,it will be printed in full.

Right justified:

Example: **input:** int a=143;
 Printf(“%5d”,a);

Output:

		1	4	3
--	--	---	---	---

Left justified:

Example: **input:** int a=143;
 Printf(“%-5d”,a);

Output:

Example: **input:** int a=143;
 Printf("%05d",a);

Output:

0	0	1	4	3
---	---	---	---	---

Output of Floating point numbers: The format specification for printing an floating point number is

Syntax: %w.pf

Where w=field width that are useful to display the value and P=precession indicates number of digits to be display after the decimal point.

The format specification for printing an floating point number in exponential notation is

Syntax: %w.pe

Example: **input** float b=14.378413;
 printf("%.5f",b);

Output:

1	4	.	3	7	8	4	1	3
---	---	---	---	---	---	---	---	---

Right justified:

Example: **input:** float b=14.526325;
 Printf("%.2f",b);(or)

Output:

	1	4	.	5	2
--	---	---	---	---	---

Example: **input:** float b=14.526325;
 Printf("%.02f",b);

Output:

1	4	.	5	2
---	---	---	---	---

Left justified:

Example: **input:** float b=14.526325;
 Printf("%.6.2f",b);(or)

Output:

1	4	.	5	2	
---	---	---	---	---	--

Example: **input** float b=14.378413;
 printf("%.07.2f",b);

Output:

0	0	1	4	.	3	7
---	---	---	---	---	---	---

Example: **input:** float b=14.526325;
 Printf("%.10.2e",b);(or)

Output:

		1	.	4	5	E	+	0	1
--	--	---	---	---	---	---	---	---	---

Example: **input:** float b=14.526325;
 Printf("%.10.2e",b);(or)

Output:

1	.	4	5	E	+	0	1		
---	---	---	---	---	---	---	---	--	--

Output of a single character: A single character can be displayed in a desired position using the format

Syntax: %wc

Where w=field width and P=only 1st p characters of the string are to be displayed. The default value for w is 1.

Output of Strings: The format specification for outputting strings is

Syntax: %w.ps

Where w specifies the field width for display and p instructs that only the first p characters of the string are to be displayed.

Example: **input:** `char mycity[]="DELHI";`
 `Printf("%s",mycity);`

Output:

D	E	L	H	I
----------	----------	----------	----------	----------

Example: **input:** `Char mycity[]="DELHI";`
 `Printf("%3s",mycity);`

Output:

D	E	L	H	I
----------	----------	----------	----------	----------

Right justified:

Example: **input:** `char mycity[]="DELHI";`
 `printf("%10s",mycity);`

Output:

					D	E	L	H	I
--	--	--	--	--	----------	----------	----------	----------	----------

Left justified:

Example: **input:** `char mycity[]="DELHI";`
 `Printf("%-10s",mycity);`

Output:

D	E	L	H	I					
----------	----------	----------	----------	----------	--	--	--	--	--

Example: **input:** `char mycity[]="DELHI";`
 `Printf("%10.3s",mycity);`

Output:

							D	E	L
--	--	--	--	--	--	--	----------	----------	----------

Example: **input:** `char mycity[]="DELHI";`
 `Printf("%-10.3s",mycity);`

Output:

D	E	L							
----------	----------	----------	--	--	--	--	--	--	--

Unformatted functions:-Unformatted Input functions:

getchar(): `getchar()` is used for reading a single character from the keyboard. It is a buffered input function this means that when a character is typed on the keyboard, it is immediately not passed on to the memory; It remains in buffer area. The typed character is stored in a memory when enter key is pressed after typing the character.so it gives a chance to correct, if you mistype the character. `getchar()` is included in header file "stdio.h".

Syntax: `int getchar(void);`

Example: `#include <stdio.h>`
 `int main()`
 `{`
 `char a;`
 `clrscr();`
 `Printf("enter a character");`
 `a=getchar();`
 `printf("given character is %c", a);`
 `getch();`
 `return 0;`
 `}`

}

Input: enter character g(press enter key)**Output:** given character is g

getch(): This function will read a single character. The inputted character does not return on to the output screen. whenever you press enter the control goes into program. getch() is included in header file "conio.h"

Syntax: `int getch();`

Example:

```
#include <stdio.h>
#include <conio.h>
int main()
{
    printf("%c", getch());
    return 0;
}
```

Input: g

getche(): It reads a single character from the keyboard and displays immediately on output screen without waiting for enter key. getche() is included in header file "conio.h"

Syntax: `int getche(void);`

Example:

```
#include <stdio.h>
#include <conio.h>
int main()
{
    printf("%c", getche());
    return 0;
}
```

Input: g**Output:** gg

gets(): It is used to read a string of characters from the keyboard until enter key is pressed (or) a newline character occurred (or) the end-of-file is reached. gets() is included in header file "stdio.h"

Syntax: `char gets (string_variable_name);`

Example:

```
#include <stdio.h>
int main()
{
    char str[50];
    printf("Enter a string : ");
    gets(str);
    printf("You entered: %s", str);
    return(0);
}
```

Output: Enter a string: tutorialspoint.com
You entered: tutorialspoint.com

Unformatted Output functions:

putchar(): This function is used to display one character at a time on to the user screen. putchar() is included in header file "stdio.h".

Syntax: **putchar(char_variable_name);**

Example: `#include <stdio.h>
int main ()
{
 char ch;
 for(ch = 'A' ; ch <= 'Z' ; ch++)
 {
 putchar(ch);
 }
 return(0);
}`

Output: ABCDEFGHIJKLMNOPQRSTUVWXYZ

putch():putch displays any alphanumeric characters to the standard output device. It displays only one character at a time. putch() is included in header file “conio.h”

Syntax: **putch (alphanumeric_variable_name);**

Example: `#include<stdio.h>
#include<conio.h>
int main()
{
 char ch;
 printf("enter value of ch ");
 ch =getch();
 printf("\n value of ch is \t ");
 putch(ch);
}`

Input: enter value of ch a(then don't need to hit enter)

Output: value of ch is a

puts():It displays a single (or) paragraph of text to the standard output device. puts is included in header file “stdio.h”

syntax: **puts(variable_name);**

Example: `#include<stdio.h>
#include<conio.h>
void main()
{
 char a[20];
 gets(a);
 puts(a);
 getch();
}`

Output: Hi hello

clrscr():This function is used to erase all the previously displayed text (or) values on the screen, that is clear the user screen. Generally we can write the clrscr() function after the variables declaration (or) declaration of variables. clrscr() is included in header file “conio.h”

Syntax: **clrscr();**

exit(): The exit function is used to stop the program execution at the statement where it is written in the program. The statements written after this exit statement will not be executed as the program terminates its execution when it sees exit() function. exit() is included in header file "dos.h".

Syntax: **exit();**

UNIT-III

Control Statements

The normal flow of execution in a high level language is sequential, i.e., each statement is executed in the order of its appearance in the program. However, depending on the requirements of a problem it might be required to alter the normal sequence of execution in a program. The statements which specify the order of execution of statements are called **control flow statements**.

Statement: Statement is an instruction given to the computer to perform any kind of action. Action may be in the form of data movement, decision making etc. Statement is always terminated by semicolon.

(Or)

Statement is an expression followed by semicolon, which performs some kind of action (or) task.

Example: x=a+b;

Compound Statement: A compound statement is a grouping of statements in which each individual statement ends with a semi-colon. The group of statements is called statement_block. Compound statements are enclosed between the pair of braces ({ }). The opening brace ({) signifies the beginning and closing brace (}) signifies the end of the block.

Syntax: {
 Single statement (or) multiple statements;
 }

Example1: {
 X=a+b;
 Y=c+d;
 }

Example2: {
 X=a+b;
 }

Null Statement: Writing only a semicolon indicates a null statement. Thus ';' is a null or empty statement.

Syntax: ;

Control statements are classified into three types:

1. **Conditional control statements (or) Non iterative statements.**
2. **Looping control statements (or) Iterative statements (or) Repetition.**
3. **Unconditional control statements**

1.Non iterative statements (or)Conditional control statements: Conditional statements are used to execute a statement or a group of statements based on certain conditions are called conditional control statements. The conditional control statements are divided into two types:

1. **Decision→simple-if, if-else, nested-if-else, else-if-ladder.**
2. **Selection→switch**

Simple-if: The if statement in C programming language is used to execute a block of code only if condition is true. The general form (or) syntax of simple if is

Syntax: **if (test_expression)**
 {
 true_statement_block;
 }

The true_statement_block may be a single statement (or) a group of statements. If the condition is true, the true_statement_block will be executed; otherwise the true_statement_block will be skipped and the execution will jump to the statements after if statement.

Example1: Write a C program to check a person is eligible for voting using simple if.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int age;
    clrscr();
    printf("Enter the age : ");
    scanf("%d",&age);
    if(age>=18)
    {
        printf("Eligible for voting...\n");
    }
    printf("thank you");
    getch();
}
```

Example2: Write a C program to check whether the given number is greater than 7 using simple if.

```
#include <stdio.h>
#include<conio.h>
void main()
{
    int number = 0;
```

```
        clrscr();
        printf("\nEnter an integer between 1 and 10: ");
        scanf("%d",&number);
        if (number > 7)
            printf("You entered %d which is greater than 7\n", number);
        getch();
    }
```

Example3: Write a C program to check if two numbers are equal using simple if.

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int m=40,n=40;
    clrscr();
    if (m == n)
    {
        printf("m and n are equal");
    }
    getch();
}
```

if-else: The if-else statement is the extension of simple-if statement. The if else statement in C programming language is used to execute a set of statements if condition is true and execute another set of statements when condition is false. The syntax is

Syntax:

```
    if(test_expression)
    {
        True_block_statements;
    }
    else
    {
        False_block_statements;
    }
```

In above syntax, if condition is true, then the True_block_statements are executed; otherwise, the false_block_statements are executed. In either case, either True_block_statements(or) false_block_statements will be executed, not both.

Example1: Write a C program to check the given number is even (or) odd using if-else.

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

```
    {  
        printf("%d is odd number.",number);  
    }  
    printf("thank you");  
    getch();  
}
```

Example2: Write a C program to check the student is pass (or) fail using if-else.

```
#include<stdio.h>  
#include<conio.h>  
void main()  
{  
    int marks=50;  
    clrscr();  
    if(marks>=40)  
    {  
        printf("Student is Pass");  
    }  
    else  
    {  
        printf("Student is Fail");  
    }  
    getch();  
}
```

Example3: Write a C program to check the two numbers are equal (or) not using if-else.

```
#include <stdio.h>  
#include<conio.h>  
void main()  
{  
    int m=40,n=20;  
    clrscr();  
    if (m == n)  
    {  
        printf("m and n are equal");  
    }  
    else  
    {  
        printf("m and n are not equal");  
    }  
    getch();  
}
```

Example4: Write a C program to find maximum of two numbers using if-else.

```
#include<stdio.h>  
#include<conio.h>  
void main()  
{  
    int x,y;  
    printf("Enter x,y values :");  
    scanf("%d%d",&x,&y);  
    if ( x > y )  
    {  
        printf("X is large number - %d\n",x);  
    }  
}
```

```
    }  
    else  
    {  
        printf("Y is large number - %d\n",y);  
    }  
}
```

Example5: Write a C program to find the given year is leap year or not using if-else.

```
#include<stdio.h>  
#include<conio.h>  
void main()  
{  
    int year;  
    clrscr();  
    printf("Enter the year");  
    scanf("%d",&year);  
    if(year%4==0)  
    {  
        printf("It is a leap year");  
    }  
    else  
    {  
        printf("It is not a leap year");  
    }  
    getch();  
}
```

Example6: Write a C Program to check Perfect Square using if-else.

```
#include <stdio.h>  
#include<conio.h>  
#include <math.h>  
void main()  
{  
    int n, temp;  
    printf("Enter a number: ");  
    scanf("%d",&n);  
    temp = sqrt(n);  
    if(temp*temp == n)  
    {  
        printf("YES.");  
    }  
    else  
    {  
        printf("NO.");  
    }  
    getch();  
}
```

Nested-if-else: The statements within the if statement can contain another if statement and which in turn may contain another if and so on is called nested if-else statement. The nested if statement in C programming language is used when multiple conditions needs to be tested. The inner statement will execute only when outer if statement is true otherwise control won't even reach inner if statement.

Syntax:

```
if(test_expression1)
{
    if(test_expression2)
    {
        True_block_statements2;
    }
    else
    {
        False_block_statements2;
    }
}
else
{
    False_block_statements1;
}
```

In above syntax, the if test_expression evaluates to 0 the false_block_statements1 will be executed; otherwise it checks test_expression2. if the test_expression2 is true, the True_block_statements2 will be executed; otherwise False_block_statements2 will be executed.

Example1: Write a C program to find out maximum of three numbers using nested-if-else.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    float a, b, c;
    printf("enter a,b,c values");
    scanf("%f%f%f", &a, &b, &c);
    printf("largest value is\n");
    if(a>b)
    {
        if(a>c)
        {
            printf("%f\n", a);
        }
        else
        {
            printf("%f\n", c);
        }
    }
    else
    {
        if(b>c)
        {
            printf("%f\n", b);
        }
        else
        {
            printf("%f\n", c);
        }
    }
}
```

```
    getch();
```

```
}
```

Example2: Write a C program to find the Two numbers are equal or one number is greater than other or one number is less than another using nested-if-else.

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

```
#include<conio.h>
```

```
void main()
```

```
{
```

```
    int numb1, numb2;
```

```
    clrscr();
```

```
    printf("Enter two integers to check\n");
```

```
    scanf("%d %d",&numb1,&numb2);
```

```
    if(numb1==numb2)
```

```
    {
```

```
        printf("Result: %d = %d",numb1,numb2);
```

```
    }
```

```
    else
```

```
    {
```

```
        if(numb1>numb2)
```

```
        {
```

```
            printf("Result: %d > %d",numb1,numb2);
```

```
        }
```

```
        else
```

```
        {
```

```
            printf("Result: %d > %d",numb2,numb1);
```

```
        }
```

```
    }
```

```
}
```

Example3: Write a C program to find the Two No's are equal or one no. is greater than other Or one no. is less than another using nested-if-else.

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

```
#include<conio.h>
```

```
void main()
```

```
{
```

```
    int m=40,n=20;
```

```
    if (m>n)
```

```
    {
```

```
        printf("m is greater than n");
```

```
    }
```

```
    else
```

```
    {
```

```
        if(m<n)
```

```
        {
```

```
            printf("m is less than n");
```

```
        }
```

```
        else
```

```
        {
```

```
            printf("m is equal to n");
```

```
        }
```

```
    }
```

```
}
```

Example4: Write a C program to find the login by using nested if else.

```
#include <stdio.h>
#include <conio.h>
void main()
{
    char username;
    int password;
    clrscr();
    printf("Username:");
    scanf("%c",&username);
    printf("Password:");
    scanf("%d",&password);
    if(username=='a')
    {
        if(password==12345)
        {
            printf("Login successful");
        }
        else
        {
            printf("Password is incorrect, Try again.");
        }
    }
    else
    {
        printf("Username is incorrect, Try again.");
    }
    getch();
}
```

Example5: Write a C program to find biggest among three nos by using nested-if-else.

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int a,b,c;
    int biggest;
    clrscr();
    printf("Enter 1st Number: ");
    scanf("%d", &a);
    printf("Enter 2nd Number: ");
    scanf("%d", &b);
    printf("Enter 3rd Number: ");
    scanf("%d", &c);
    if(a > b)
    {
        if(a > c)
            biggest = a;
        else
            biggest = c;
    }
    else
```



```
        {
            if(b > c)
                biggest = b;
            else
                biggest = c;
        }
        printf("Biggest of 3 numbers is: %d\n", biggest);
    }
```

Example6: Write a C program to find the roots of Quadratic Equation using nested-if-else.

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
void main()
{
    float r1,r2,d,a,b,c;
    clrscr();
    printf("enter a,b,c values");
    scanf("%f%f%f",&a,&b,&c);
    d=b*b-4*a*c;
    if(d==0)
    {
        printf("roots are equal");
        r1=-b/(2*a);
        r2=-b/(2*a);
        printf("the roots are r1=%f \n r2=%f \n",r1,r2);
    }
    else
    {
        if(d>0)
        {
            printf("roots are real and distinct");
            r1=(-b+sqrt(d))/(2*a);
            r2=(-b-sqrt(d))/(2*a);
            printf("the roots are r1=%f\nr2=%f\n",r1,r2);
        }
        else
        {
            printf("roots are imaginary");
        }
    }
    getch();
}
```

Program7: C program to check whether a year is leap year or not using if else statement.

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int year;
    printf("Enter a year: ");
    scanf("%d",&year);
```

```
if(year%4 == 0)
{
    if( year%100 == 0)
    {
        if ( year%400 == 0)
        {
            printf("%d is a leap year.", year);
        }
        else
        {
            printf("%d is not a leap year.", year);
        }
    }
    else
    {
        printf("%d is a leap year.", year );
    }
}
else
{
    printf("%d is not a leap year.", year);
}
getch();
}
```

Output 2: Enter year: 1900
1900 is not a leap year.

Output 3: Enter year: 2012
2012 is a leap year.

Program12: C program To check given number is negative or positive or zero

```
#include <stdio.h>
#include <conio.h>
void main()
{
    float num;
    printf("Enter a number: ");
    scanf("%f",&num);
    if (num<=0)
    {
        if (num==0)
        {
            printf("You entered zero.");
        }
        else
        {
            printf("%.2f is negative.",num);
        }
    }
    else
        printf("%.2f is positive.",num);
}
```

```
    getch();
```

```
}
```

Output 1: Enter a number: 12.3
12.30 is positive.

Dangling else problem: This occurs when a matching else is not available for every if. The answer for this is very simple. Always match an else to the most recent unmatched if in the current block. In some cases, it is possible that the false condition is not required. In such situations, else statement may be omitted.

else-if-ladder (or) if-else-if: It is also called as multi-way decision statement. The else–if-ladder is a form of nested-if statement. Here nesting is allowed only in the else part. The if-else-ladder statement in C programming language is used to test set of conditions in sequence. An if condition is tested only when all previous if conditions in if-else ladder is false. If any of the conditional expression evaluates to true, then it will execute the corresponding code block and exits whole if-else ladder. The syntax of the else-if ladder is:

Syntax:

```
if (condition1)
{
    true_block_statements1;
}
else if(condition2)
{
    true_block_statements2;
}
else if(condition3)
{
    true_block_statements3;
}
else if(condition4)
{
    true_block_statements4;
}
else
{
    default statement;
}
```

Example1: Write a C program which takes percentage as input and provides grades as output.
A university provides grades for the percentages as follows:

Percentage	Grade
>=70	Distinction
>=60&<70	First class
>=50&<60	second class
>=40&<50	Third class
<40	fail

```
#include<stdio.h>
#include<conio.h>
void main()
{
```

```
float per;
clrscr();
printf("enter the percentage");
scanf("%f",&per);
if(per>=70)
{
    printf("the grade is distinction");
}
else if(per>=60 && per<70)
{
    printf("the grade is first class");
}
else if(per>=50 && per<60)
{
    printf("the grade is second class");
}
else if(per>=40 && per<50)
{
    printf("the grade is third class");
}
else
{
    printf("the grade is fail");
}
getch();
}
```

Example2: Write a C program to select random color using else-if-ladder.

```
#include<stdio.h>
#include<string.h>
void main()
{
    int n;
    printf(" Enter 1 to 4 to select random color");
    scanf("%d",&n);
    if(n==1)
    {
        printf("You selected Red color");
    }
    else if(n==2)
    {
        printf("You selected Green color");
    }
    else if(n==3)
    {
        printf("You selected yellow color");
    }
    else if(n==4)
    {
        printf("You selected Blue color");
    }
}
```

```
        else
        {
            printf("No color selected");
        }
        getch();
    }
}
```

Example3: Write a C program to find the Roots of Quadratic Equation using else-if-ladder.

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
void main()
{
    float r1,r2,d,a,b,c;
    clrscr();
    printf("enter a,b,c values");
    scanf("%f%f%f",&a,&b,&c);
    d=b*b-4*a*c;
    if(d==0)
    {
        printf("roots are equal");
        r1=-b/(2*a);
        r2=-b/(2*a);
        printf("the roots are r1=%f\nr2=%f\n",r1,r2);
    }
    else if(d>0)
    {
        printf("roots are real and distinct");
        r1=(-b+sqrt(d))/(2*a);
        r2=(-b-sqrt(d))/(2*a);
        printf("the roots are r1=%f\n r2=%f\n", r1,r2);
    }
    else
    {
        printf("roots are imaginary");
    }
    getch();
}
```

Example4: Write a program to read three numbers and find the largest one by using else-if-ladder

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int a, b, c;
    clrscr();
    printf("Enter 1st number:");
    scanf("%d", &a);
    printf("Enter 2nd number:");
    scanf("%d", &b);
    printf("Enter 3rd number:");
    scanf("%d", &c);
```

```
        if ((a>b) && (a>c))
        {
            printf("Highest Number is: %d", a);
        }
        else if ((b>a) && (b>c))
        {
            printf("Highest Number is: %d", b);
        }
        else
        {
            printf("Highest Numbers is: %d", c);
        }
        getch( );
    }
```

Example5: Write a C program to find the Current bill of a Customer using else-if-ladder.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int units, custnum;
    float charges;
    printf("Enter CUSTOMER NO. and UNITS consumed\n");
    scanf("%d%d", &custnum, &units);
    if (units <= 200)
    {
        charges = 0.5 * units;
    }
    else if (units <= 400)
    {
        charges = 100 + 0.65 * (units - 200);
    }
    else if (units <= 600)
    {
        charges = 230 + 0.8 * (units - 400);
    }
    else
    {
        charges = 390 + (units - 600);
    }
    printf("\n\nCustomer No: %d: Charges = %.2f\n", custnum, charges);
}
```

Selection:

Switch statement: A switch statement allows a variable or value of an expression to be tested for equality against a list of possible case values and when match is found, the block of code associated with that case is executed. The syntax of switch statement is:

Syntax: **switch (expression)**
 {
 case constant1: statements_block1;
 break;

```

    case constant2: Statements_lock2;
                break;
    ...

    default: default_block
}

```

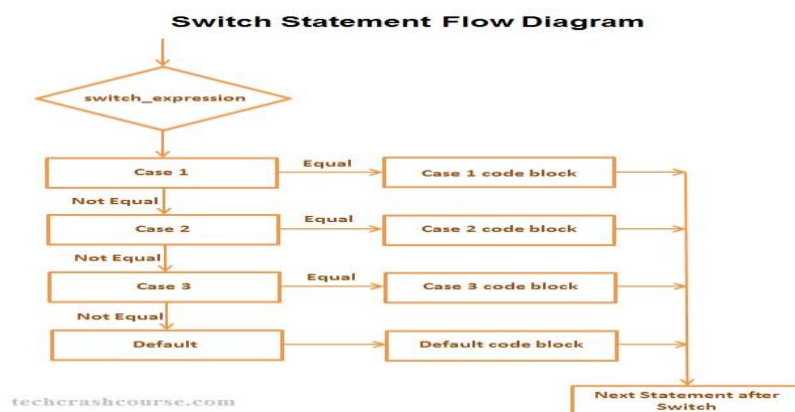
switch: It is the keyword indicating the start of the switch statement.

expression: It is an expression (or) variable (or) constant that results in an integer value (or) character value with which the comparisons will be done for equality with the case constants inside the switch statement.

case: It is the keyword indicating each of the alternatives for the value of the expression in the switch statement. Each case will have a label, a value to compare, a block of statements associated with it for execution when the comparison is satisfied. It will have a break as the last statement for each case.

break: It is the statement that transfers the execution control to go outside the switch statement when it is executed. If we don't write the break statement as the last statement of each case block, then the execution control will not break and will also execute the next cases below, even though they are not matching to the expression's resulting value.

default: It is the keyword used as the last case option and it will get executed when none of the above case alternatives are not equal to the value given by the switch expression.



Rules for switch statement:-

1. The switch expression must be an integral Type (**integer, character**)
2. Case labels must be consonants or consonant expression.
3. Case label must be of integral Type (Integer, Character).
4. Case labels must be unique. No two labels can have the same values.
5. Case Labels must ends with Colon.
6. The break statements transfer the control out of switch statement.
7. The break statement is optional. i.e; two or more case labels may belong to the same statements.
8. The default label is optional if present it will be executed when the expression does not find a

matching case label.

9. There can be at most one default label.
10. The default may be placed anywhere but usually placed at the end.
11. Nesting (switch within switch) is allowed

Example1:

```
#include<stdio.h>
#include<conio.h>
int main()
{
    int i;
    clrscr();
    printf("enter i value");
    scanf("%d", &i);
    switch (i)
    {
        case 1:
            printf("Case1 ");
            break;
        case 2:
            printf("Case2 ");
            break;
        case 3:
            printf("Case3 ");
            break;
        case 4:
            printf("Case4 ");
            break;
        default:
            printf("Default ");
    }
    return 0;
}
```

Example2:

```
#include <stdio.h>
#include<conio.h>
void main()
{
    char grade ;
    printf("enter your grade");
    scanf("%c", &grade);
    switch(grade)
    {
        case 'A' :
            printf("Excellent!\n" );
            break;
        case 'B' :
            printf("good!\n" );
            break;
        case 'C' :
            printf("Well done\n" );
            break;
    }
}
```



```
        case 'D' :
            printf("You passed\n" );
            break;
        case 'F' :
            printf("Better try again\n" );
            break;
        default :
            printf("Invalid grade\n" );
    }
    printf("Your grade is  %c\n", grade );
}
```

Example3:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    char ch;
    printf("Enter the Vowel (In Capital Letter):");
    scanf("%c", &ch);
    switch( ch )
    {
        case 'A' :
        case 'a': printf( "Your Character Is A or a\n" );
                break;
        case 'E' :
        case 'e': printf( "Your Character Is E or e\n" );
                break;
        case 'I' :
        case 'i': printf( "Your Character Is I or i\n" );
                break;
        case 'O':
        case 'o': printf( "Your Character Is O or o\n" );
                break;
        case 'U':
        case 'u': printf( "Your Character Is U or u\n" );
                break;
        default : printf( "Your Character is Not Vowel.Otherwise Not a Capital
                        Letter\n" );
                break;
    }
    getch();
}
```

Example4: Write a C program to check whether a given character is vowel or not using switch.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    char ch;
    clrscr();
    printf("enter a character");
    scanf("%c",&ch);
    switch(ch)
```

```
{
    case 'a': printf("vowel");
                break;
    case 'A': printf("vowel");
                break;
    case 'e': printf("vowel");
                break;
    case 'E': printf("vowel");
                break;
    case 'i': printf("vowel");
                break;
    case 'I': printf("vowel");
                break;
    case 'o': printf("vowel");
                break;
    case 'O': printf("vowel");
                break;
    case 'u': printf("vowel");
                break;
    case 'U': printf("vowel");
                break;
    default: printf(" not vowel");
}
getch();
}
```

Example5:

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int a, b, c;
    char ch;
    clrscr();
    printf("Enter your operator(+, -, /, *, %)\n");
    scanf("%c", &ch);
    printf("Enter the values of a and b\n");
    scanf("%d%d", &a, &b);
    switch(ch)
    {
        case '+': c = a + b;
                    printf("addition of two numbers is %d", c);
                    break;
        case '-': c = a - b;
                    printf("subtraction of two numbers is %d", c);
                    break;
        case '*': c = a * b;
                    printf("multiplication of two numbers is %d", c);
                    break;
        case '/': c = a / b;
                    printf("quotient of two numbers is %d", c);
```

```
        break;
    case '%': c = a % b;
        printf("remainder of two numbers is %d", c);
        break;
    default: printf("Invalid operator");
}
getch();
```

Example6:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int day;
    printf("\nEnter the number of the day:");
    scanf("%d",&day);
    switch(day)
    {
        case 1:printf("Sunday");
            break;
        case 2:printf("Monday");
            break;
        case 3: printf("Tuesday");
            break;
        case 4:printf("Wednesday");
            break;
        case 5:printf("Thursday");
            break;
        case 6: printf("Friday");
            break;
        case 7:printf("Saturday");
            break;
        default: printf("Invalid choice");
    }
}
```

Example7: Write a C program to perform addition, subtraction, multiplication, division and modulus using switch.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a, b, op, result;
    printf("enter the values of a,b");
    scanf("%d%d",&a,&b);
    printf("1.addition\n2.substraction\n3.multiplication\n4.division\n5.Modulous");
    printf("enter choice");
    scanf("%d",&op);
    switch(op)
    {
        case 1:
            printf("addition");
            result=a+b;
```

```
        printf("%d",result);
        break;
    case 2:
        printf("subtraction");
        result=a-b;
        printf("%d", result);
        break;
    case 3:
        printf("multiplication");
        result=a*b;
        printf("%d",result);
        break;
    case 4:
        printf("Division");
        result=a/b;
        printf("%d",result);
        break;
    case 5:
        printf("Modulous");
        result=a%b;
        printf("%d",result);
        break;
    default: printf("invalid choice");
    }
    getch();
}
```

Looping Control Statements (or) Repetition (or) Iterative statements: Loop control statements in C are used to execute a block of code several times until the given condition is true. Whenever we need to execute some statements multiple times, we need to use a loop statement.

(or)

The looping control statements that enable the programmer to execute a set of statements repeatedly till the required activity is completed are called looping control statements.

1. The loop is defined as a block of statements that are repeatedly executed for certain number of times.
2. The statements within a loop may be executed for a fixed number of times or until a certain condition is reached.
3. The various types of loop control statements are: **for, while and do while.**

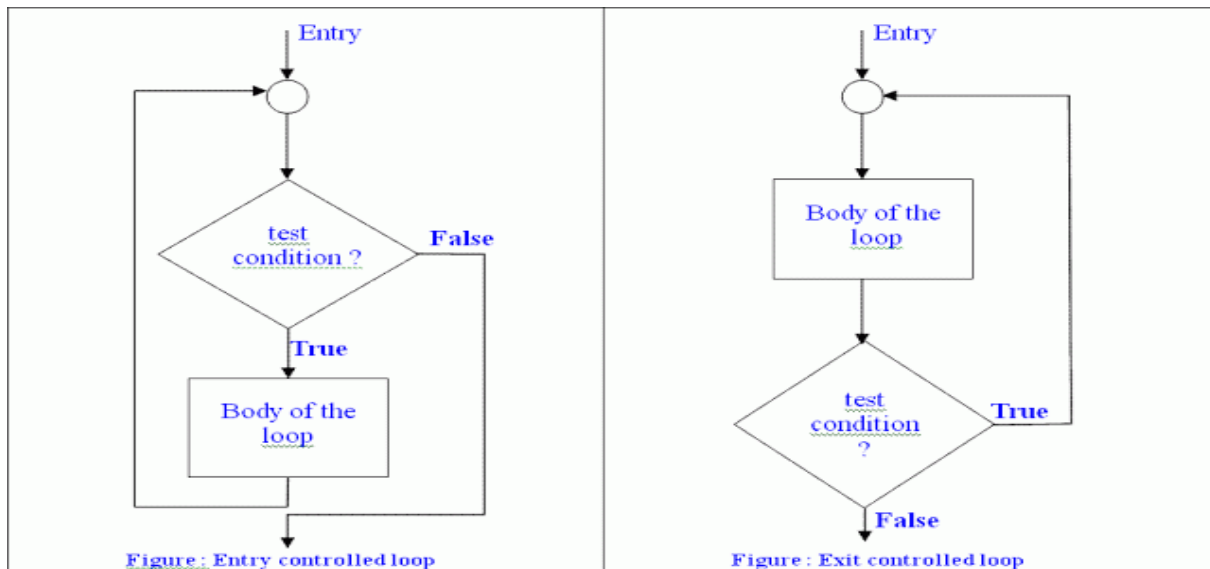
TYPES OF LOOPING CONTROL STATEMENTS:

Basically, the type of looping control statements depends on the condition checking mode. Condition checking can be made in two ways as: Before loop and after loop. So, there are 2(two) types of looping control statements.

- ❖ Entry controlled loop
- ❖ Exit controlled loop

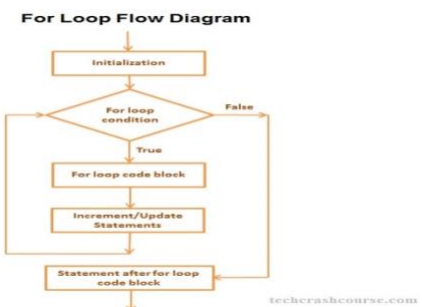
Entry controlled loop: In such type of loop, the test condition is checked first before the loop is executed. Examples of Entry controlled loop are: **for** and **while**.

Exit controlled loop: In such type of loop, the loop is executed first, then condition is checked, the loop executed at least one time. Examples of exit controlled loop is **do-while**.



for: Repeats a block of code multiple times until a given condition is true. Initialization, looping condition and update expression (increment/decrement) is part of for loop. The syntax is

Syntax: `for(initialization;condition;inc/dec)`
 {
 Body of the loop;
 }



Where

- ✓ **for** is a keyword (reserved word)
- ✓ **initialization** :The initialization is usually an assignment statement that sets the loop control variable. For Example: `i=1` and `count=0` Here `i`, `count` are loop control variables.
- ✓ **condition** :The condition is a relational expression that determines when the loop exits
 Example: `i>10` that determines when the loop will exit. if the condition is true, the body of the loop is executed; otherwise the loop is terminated.
- ✓ **increment/decrement**: The increment/decrement defines how the loop control variable changes each time the loop is repeated.

Example: `i=i+1`

These three major sections must be separated by semicolons. The for loop continues to execute as long as the condition is true.

Example1: Write a C program to print n natural numbers using for loop.

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

Example2: Write a C program to find out the sum of n natural numbers using for loop.

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int n, i, sum=0;
    printf("Enter the value of n.\n");
    scanf("%d",&n);
    for(i=1;i<=n;i++)    /*for loop terminates if i>n*/
    {
        sum=sum+i;
    }
    printf("Sum=%d",sum);
    getch();
}
```

Example3: Write a C program to find factors of a integer using for loop.

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int no, x;
    clrscr();
    printf("Enter the required number:");
    scanf("%d",&no);
    printf("\nThe factors are:");
    for(x=1; x<=no; x++)
    {
        if(no%x==0)
            printf("\n%d",x);
    }
    getch();
}
```

Example4: Write a C Program to check whether the given number is Perfect Square (or) not

using for loop.

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int i, n;
    printf("Enter a number: ");
    scanf("%d", &n);
    for(i=0; i<=n; i++)
    {
        if (n==i*i)
        {
            printf("YES");
            exit(0);
        }
    }
    printf("NO");
    getch();
}
```

Program5: C program to find multiplication table up to 10.

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int n, i;
    printf("Enter an integer to find multiplication table:");
    scanf("%d", &n);
    for(i=1; i<=10; ++i)
    {
        printf("%d * %d = %d\n", n, i, n*i);
    }
    getch();
}
```

Output: Enter an integer to find multiplication table: 9

```
9 * 1 = 9
9 * 2 = 18
9 * 3 = 27
9 * 4 = 36
9 * 5 = 45
9 * 6 = 54
9 * 7 = 63
9 * 8 = 72
9 * 9 = 81
9 * 10 = 90
```

Example6: Write a C program to print Fibonacci series using for loop.

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int a=0, b=1, n, f, i;
    printf("enter the value of n");
```

```
scanf("%d",&n);
printf("%d %d",a,b);
for(i=1;i<=n-2;i++)
{
    f=a+b;
    printf("%4d",f);
    a=b;
    b=f;
}
getch();
}
```

Example7: Write a C program to print sum of even no.s in first n natural no.s using for loop.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int n,i,sum=0;
    clrscr();
    printf("\n\tenter the value of n");
    scanf("%d", &n);
    for(i=1;i<=n;i++)
    {
        if ( i%2 == 0)    // to add even no.s only
        {
            sum=sum+i;
        }
    }
    printf("\n\tsum of even nos is %d", sum);
    getch();
}
```

Example8: Write a C program to find out biggest and smallest numbers in a list of integers.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int n,i,big,small,i,totalnumber;
    clrscr();
    printf("\n\tenter the value of total number");
    scanf("%d", &totalnumber);
    for(i=1;i<=totalnumber;i++)
    {
        scanf("%d",&n);
        if(i==1)
        {
            big=small=n;
        }
        if(big<n)
            big=n;
        if(small>n)
```



```

        small=n;
    }
    printf("\n The biggest number of all is %d", big);
    printf("\n The smallest number of all is %d", small);
    getch();
}

```

Example9: Write a C program to print the prime numbers from 1 to n using for loop.

```

#include<stdio.h>
#include<conio.h>
void main()
{
    int n, c, j, i;
    printf("enter the value of n");
    scanf("%d",&n);
    for(i=1;i<=n;i++)
    {
        c=0;
        for(j=1;j<=i;j++)
        {
            if(i%j==0)
            {
                c++;
            }
        }
        if(c==2)
        {
            printf("%5d", i);
        }
    }
    getch();
}

```

Example10: Write a C program to print sum of individual digits of a positive integer.

```

#include<stdio.h>
#include<conio.h>
void main()
{
    int n, k, sum=0;
    printf("enter the value of n");
    scanf("%d",&n);
    for( ; n!=0; n=n/10)
    {
        k=n%10;
        sum=sum+k;
    }
    printf("value of sum is %d",sum);
    getch();
}

```

Program11: C program to display all prime numbers between Two interval entered by user.

```

#include <stdio.h>
#include<conio.h>

```

```
void main()
{
    int n1, n2, i, j, flag;
    printf("Enter two numbers(intervals): ");
    scanf("%d %d", &n1, &n2);
    printf("Prime numbers between %d and %d are: ", n1, n2);
    for(i=n1+1; i<n2; ++i)
    {
        flag=0;
        for(j=2; j<=i/2; ++j)
        {
            if(i%j==0)
            {
                flag=1;
                break;
            }
        }
        if(flag==0)
            printf("%d ",i);
    }

    getch();
}
```

Output: Enter two numbers(intervals): 20 50
Prime numbers between 20 and 50 are: 23 29 31 37 41 43 47

Program12: Simple program of c find the largest number

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int n,num,i;
    int big;
    printf("Enter the values of n: ");
    scanf("%d",&n);
    printf("Number %d",1);
    scanf("%d",&big);
    for(i=2;i<=n;i++)
    {
        printf("Number %d: ",i);
        scanf("%d",&num);
        if(big<num)
            big=num;
    }
    printf("Largest number is: %d",big);
    getch();
}
```

Output: Enter the values of n:
Number 1: 12
Number 2: 32
Number 3: 35
Largest number is: 35

Program13: Write a c program for finding gcd (greatest common divisor) of two given numbers

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int x,y,m,i;
    printf("Insert any two number: ");
    scanf("%d%d",&x,&y);
    if(x>y)
        m=y;
    else
        m=x;
    for(i=m;i>=1;i--)
    {
        if(x%i==0&&y%i==0)
        {
            printf("\nGCD of two number is : %d",i) ;
            break;
        }
    }
    getch();
}
```

Program14: C program to find the number of integers divisible by 5 Between the given range num1 and num2, where num1 < num2. Also find the sum of all these integer numbers which are divisible by 5 and display the total.

```
#include <stdio.h>
#include<conio.h>
void main()
{
    int i, num1, num2, count = 0, sum = 0;
    printf("Enter the value of num1 and num2 \n");
    scanf("%d %d", &num1, &num2);
    /* Count the number and compute their sum*/
    printf("Integers divisible by 5 are \n");
    for(i = num1; i < num2; i++)
    {
        if (i % 5 == 0)
        {
            printf("%3d,", i);
            count++;
            sum = sum + i;
        }
    }
    printf("\n Number of integers divisible by 5 between %d
        and %d= %d\n", num1, num2, count);
    printf("Sum of all integers that are divisible by 5 =
        %d\n",sum);
    getch();
}
```

Output: Enter the value of num1 and num2 12 17
Integers divisible by 5 are 15,
Number of integers divisible by 5 between 12 and 17 = 1
Sum of all integers that are divisible by 5 = 15

Example15: Write a C program to print pyramid of numbers using for loop.

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
#include<stdio.h>
#include<conio.h>
void main()
{
    int i,j,rows;
    clrscr();
    printf("Enter the number of rows: ");
    scanf("%d",&rows);
    for(i=1;i<=rows;++i)
    {
        for(j=1;j<=i;++j)
        {
            printf("%d ",j);
        }
        printf("\n");
    }
    getch();
}
```

Example16: Write a C program to print Floyd's triangle using for loop.

```
1
2 3
4 5 6
7 8 9 10
#include<stdio.h>
#include<conio.h>
void main()
{
    int rows,i,j,k=1;
    clrscr();
    printf("Enter number of rows: ");
    scanf("%d",&rows);
    for(i=1;i<=rows;i++)
    {
        for(j=1;j<=i;++j)
        {
            printf("%5d ",k);
            ++k;
        }
        printf("\n");
    }
    getch();
}
```

}

Example 17. Write a c program to print following pattern using for loop.

```

A B C D E F G F E D C B A
A B C D E F   F E D C B A
A B C D E     E D C B A
A B C D       D C B A
A B C         C B A
A B           B A
A             A

```

```

#include<stdio.h>
#include<conio.h>
void main()
{
    int i,j;
    clrscr();
    for(i=0;i<=6;i++)
    {
        for(j=65;j<=71-i;j++)
        {
            printf("%c",j);
        }
        for(j=1;j<=2*i-1;j++)
        {
            printf(" ");
        }
        for(j=71-i;j>=65;j--)
        {
            if(j!=71)
                printf("%c",j);
        }

        printf("\n");
    }
    getch();
}

```

Program18: C Program to Print Diamond Pattern.

```

*
***
*****
*****
*****
*****
***
*
#include <stdio.h>
#include<conio.h>
void main()
{
    int number, i, k, count = 1;

```

```

printf("Enter number of rows\n");
scanf("%d", &number);
count = number - 1;
for (k = 1; k <= number; k++)
{
    for (i = 1; i <= count; i++)
        printf(" ");
    count--;
    for (i = 1; i <= 2 * k - 1; i++)
        printf("*");
    printf("\n");
}
count = 1;
for(k = 1; k <= number - 1; k++)
{
    for (i = 1; i <= count; i++)
        printf(" ");
    count++;
    for (i = 1 ; i <= 2 *(number - k)- 1; i++)
        printf("*");
    printf("\n");
}
getch();
}

```

Output: Enter number of rows 3

```

*
***
*****
***
*

```

Example 19: Write a c program to print following pattern using for loop.

```

0
1 1 1
2 2 2 2 2
3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4
#include<stdio.h>
#include<conio.h>
void main()
{
    int i,j;
    for(i=1;i<=5;i++)
    {
        for(j=1;j<=5-i;j++)

```

```

        {
            printf(" ");
        }
        for(j=1;j<=2*i-1;j++)
        {
            printf("%2d",i-1);
        }
        printf("\n");
    }
    getch();
}

```

Example 20: Write a program to print pyramid of numbers using for loop

```

1
1 2 1
1 2 3 2 1
1 2 3 4 3 2 1
#include<stdio.h>
void main()
{
    int i,j,n;
    printf("Enter number of rows");
    scanf("%d",&n);
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n-i;j++)
        {
            printf(" ");
        }
        for(j=1;j<=i;j++)
        {
            printf("%2d",j);
        }
        for(j=i-1;j>=1;j--)
        {
            printf("%2d",j);
        }
        printf("\n");
    }
    getch()
}

```

Program21: Write a C Program to print half pyramid as using * as shown in figure below.

```

*
* *
* * *
* * * *
* * * * *
#include <stdio.h>
#include<conio.h>
void main()
{
    int i,j,rows;

```

```
        printf("Enter the number of rows: ");
        scanf("%d",&rows);
        for(i=1;i<=rows;++i)
        {
            for(j=1;j<=i;++j)
            {
                printf("* ");
            }
            printf("\n");
        }
        getch();
    }
```

Program22: Write a C Program to print triangle of characters as below.

```
    A
   B B
  C C C
 D D D D
E E E E E
#include<stdio.h>
#include<conio.h>
void main()
{
    int i,j;
    char input,temp='A';
    printf("Enter uppercase character you want in triangle at last row: ");
    scanf("%c",&input);
    for(i=1;i<=(input-'A'+1);++i)
    {
        for(j=1;j<=i;++j)
            printf("%c",temp);
        ++temp;
        printf("\n");
    }
    getch();
}
```

Program23: Write a C Program to print inverted half pyramid using * as shown below.

```
*****
****
***
**
*
#include <stdio.h>
#include<conio.h>
void main()
{
    int i,j,rows;
    printf("Enter the number of rows: ");
```



```

        scanf("%d",&rows);
        for(i=rows;i>=1;--i)
        {
            for(j=1;j<=i;++j)
            {
                printf("* ");
            }
            printf("\n");
        }
        getch();
    }

```

Program24: Write a C Program to print inverted half pyramid as using numbers as shown below.

```

1 2 3 4 5
1 2 3 4
1 2 3
1 2
1

```

```

#include <stdio.h>
#include<conio.h>
void main()
{
    int i,j,rows;
    printf("Enter the number of rows: ");
    scanf("%d",&rows);
    for(i=rows;i>=1;--i)
    {
        for(j=1;j<=i;++j)
        {
            printf("%d ",j);
        }
        printf("\n");
    }
    getch();
}

```

Program25: Write a C program to print pyramid using *.

```

      *
    * * *
  * * * * *
* * * * * *

```

```

#include <stdio.h>
#include<conio.h>
void main()
{
    int i,space,rows,k=0;

```

```

printf("Enter the number of rows: ");
scanf("%d",&rows);
for(i=1;i<=rows;++i)
{
    for(space=1;space<=rows-i;++space)
    {
        printf(" ");
    }
    while(k!=2*i-1)
    {
        printf("* ");
        ++k;
    }
    k=0;
    printf("\n");
}
getch();
}

```

Program26: Write a C program to print the pyramid of digits in pattern as below.

```

1
2 3 2
3 4 5 4 3
4 5 6 7 6 5 4
5 6 7 8 9 8 7 6 5
#include <stdio.h>
#include<conio.h>
void main()
{
    int i,space,rows,k=0,count=0,count1=0;
    printf("Enter the number of rows: ");
    scanf("%d",&rows);
    for(i=1;i<=rows;++i)
    {
        for(space=1;space<=rows-i;++space)
        {
            printf(" ");
            ++count;
        }
        while(k!=2*i-1)
        {
            if (count<=rows-1)
            {
                printf("%d ",(i+k));
                ++count;
            }
            else
            {
                ++count1;
                printf("%d ", (i+k-2*count1));
            }
            ++k;
        }
    }
}

```

```

    }
    count1=count=k=0;
    printf("\n");
}
getch();
}

```

Program27: Write a C program to display reverse pyramid.

```

* * * * *
* * * * *
* * * *
* * *
*

#include<stdio.h>
#include<conio.h>
void main()
{
    int rows,i,j,space;
    printf("Enter number of rows: ");
    scanf("%d",&rows);
    for(i=rows;i>=1;--i)
    {
        for(space=0;space<rows-i;++space)
        {
            printf(" ");
        }
        for(j=i;j<=2*i-1;++j)
        {
            printf("* ");
        }
        printf("\n");
    }
    getch();
}

```

Program28: C Program to Draw Pascal's triangle

```

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
#include<stdio.h>
#include<conio.h>
void main()
{
    int rows,coef=1,space,i,j;
    printf("Enter number of rows: ");
    scanf("%d",&rows);
    for(i=0;i<rows;i++)
    {
        for(space=1;space<=rows-i;space++)
            printf(" ");

```

```

        for(j=0;j<=i;j++)
        {
            if (j==0||i==0)
                coef=1;
            else
                coef=coef*(i-j+1)/j;
            printf("%4d",coef);
        }
        printf("\n");
    }
    getch();
}

```

Program29: C Program to Draw following pattern.

```

1
2   4
3   6   9
4   8   12  16
5   10  15  20  25
6   12  18  24  30  36
7   14  21  28  35  42  49
8   16  24  32  40  48  56  64
9   18  27  36  45  54  63  72  81
10  20  30  40  50  60  70  80  90  100

```

```

#include<stdio.h>
#include<conio.h>
void main()
{
    int i,j;
    for(i=1;i<=10;i++)
    {
        for(j=1;j<=10;j++)
        {
            printf("%5d ",i*j);
            if(i==j)
                break;
        }
        printf("\n");
    }
    getch();
}

```

Program30: Write a c program to Print square pattern

```

#####
#####
#####
#####
#####
#####
#####
#####
#####
#include<stdio.h>
#include<conio.h>

```

```

int main()
{
    int size = 8, row, col;
    for (row = 1; row <= size; ++row)
    {
        for (col = 1; col <= size; ++col)
        {
            printf("# ");
        }
        printf("\n");
    }

    return 0;
}

```

Program31: write C pattern program of stars and alphabets:

```

*
*A*
*A*A*
*A*A*A*
#include<stdio.h>
#include<conio.h>
void main()
{
    int n, c, k, space, count = 1;
    printf("Enter number of rows\n");
    scanf("%d",&n);
    space = n;
    for ( c = 1 ; c <= n ; c++)
    {
        for( k = 1 ; k < space ; k++)
            printf(" ");
        for ( k = 1 ; k <= c ; k++)
        {
            printf("*");
            if ( c > 1 && count < c)
            {
                printf("A");
                count++;
            }
        }
        printf("\n");
        space--;
        count = 1;
    }
    getch();
}

```

Program32: C Program to print the following Pattern.

```

*
* *
* * *
* * * *

```

```

* * * * *
#include <stdio.h>
#include <conio.h>
void main()
{
    int i,j,k;
    clrscr();
    for (i=1; i<=5; i++)
    {
        for (j=5; j>=i; j--)
        {
            printf(" ");
        }
        for (k=1; k<=i; k++)
        {
            printf("*");
        }
        printf("\n");
    }
    getch();
}

```

Program33: C Program to print the following Pattern.

```

* * * * *
* * * *
* * *
* *
*
#include <stdio.h>
#include <conio.h>
void main()
{
    int i,j,k,samp=1;
    clrscr();
    for (i=5; i>=1; i--)
    {
        for (k=samp; k>=0; k--)
        {
            printf(" ");
        }
        for (j=i; j>=1; j--)
        {
            printf("*");
        }
        samp = samp + 1;
        printf("\n");
    }
    getch();
}

```

Program34: Write a C Program to Print the following Pattern.

```

*
* *

```

```

    * * *
  * * * *
* * * * *
  * * * *
    * * *
      * *
        *

#include <stdio.h>
#include <conio.h>
void main()
{
    int i,j,k,samp=1;
    clrscr();
    for (i=1; i<=5; i++)
    {
        for (k=samp; k<=5; k++)
        {
            printf(" ");
        }
        for (j=0; j< i; j++)
        {
            printf("*");
        }
        samp = samp + 1;
        printf("\n");
    }
    samp = 1;
    for (i=4; i>=1; i--)
    {
        for (k=samp; k>=0; k--)
        {
            printf(" ");
        }
        for (j=i; j>=1; j--)
        {
            printf("*");
        }
        samp = samp + 1;
        printf("\n");
    }
    getch();
}

```

Program35: C Program to Print the following Pattern.

```

0
1 0 1
2 1 0 1 2
3 2 1 0 1 2 3
4 3 2 1 0 1 2 3 4
5 4 3 2 1 0 1 2 3 4 5

```

```

#include <stdio.h>
#include <conio.h>

```

```

void main()
{
    int no,i,y,x=35;
    clrscr();
    printf("Enter number of rows: ");
    scanf("%d", &no);
    for (y=0;y<=no;y++)
    {
        goto(x,y+1);
        for (i=0-y; i<=y; i++)
        {
            printf(" %3d ", abs(i));
            x=x-3;
        }
    }
    getch();
}

```

Program36: C Program to Print the following Pattern.

```

1
123
12345
1234567
123456789
1234567
12345
123
1
#include<stdio.h>
#include<conio.h>
int main()
{
    int i, j, k;
    for(i=1;i<=5;i++)
    {
        for(j=i;j<=5;j++)
        {
            printf(" ");
        }
        for(k=1;k<=(i*2);k++)
        {
            printf("%d",k);
        }
        printf("\n");
    }
    for(i=4;i>=1;i--)
    {
        for(j=5;j>i;j--)
        {
            printf(" ");
        }
        for(k=1;k<=(i*2);k++)
    }
}

```



```

        {
            printf("%d",k);
        }
        printf("\n");
    }
    return 0;
}

```

Program37: C Program to print the following Pattern.

```

1      1
12     21
123    321
1234   4321
1234554321

```

```

#include<stdio.h>
#include<conio.h>
void main()
{
    int i,j,k;
    for(i=1;i<=5;i++)
    {
        for(j=1;j<=5;j++)
        {
            if(j<=i)
                printf("%d",j);
            else
                printf(" ");
        }
        for(j=5;j>=1;j--)
        {
            if(j<=i)
                printf("%d",j);
            else
                printf(" ");
        }
        printf("\n");
    }
    getch();
}

```

Program38: C Program to Print the following Pattern.

```

1
2 2
3 3
4 4
3 3
2 2
1

```

```

#include<stdio.h>
#include<conio.h>
int main()
{

```

```
int i,j,k;
for(i=1;i<=4;i++)
{
    for(j=4;j>=(i-1)*2-1;j--)
        printf(" ");
    printf("%d",i);
    for(j=2;j<=(i-1)*4;j++)
        printf(" ");
    if(i>1)
        printf("%d",i);
    printf("\n");
}
for(i=3;i>=1;i--)
{
    for(j=4;j>=(i-1)*2-1;j--)
        printf(" ");
    printf("%d",i);
    for(j=2;j<=(i-1)*4;j++)
        printf(" ");
    if(i>1)
        printf("%d",i);
    printf("\n");
}
return 0;
}
```

Program39: C Program to print the following Pattern.

```
1  2  3  4  5
16           6
15           7
14           8
13 12 11 10 9
```

```
#include<stdio.h>
#include<conio.h>
int main()
{
    int i,j,k=6,l=13,m=16;
    for(i=1;i<=5;i++)
    {
        for(j=1;j<=5;j++)
        {
            if(i==1)
                printf("%-3d",j);
            else if(j==5)
                printf("%-3d",k++);
            else if(i==5)
                printf("%-3d",l--);
            else if(j==1)
                printf("%-3d",m--);
            else
                printf(" ");
        }
    }
```

```

        printf("\n");
    }
    return 0;
}

```

Program40: C Program to print the following Pattern.

```

*
* *
* *
* *
* *
* *
* *
*
#include<stdio.h>
#include<conio.h>
int main()
{
    int i, j;
    for(i=1; i<=5; i++)
    {
        for(j=5; j>i; j--)
        {
            printf(" ");
        }
        printf("*");
        for(j=1; j<(i-1)*2; j++)
        {
            printf(" ");
        }
        if(i==1)
            printf("\n");
        else
            printf("*\n");
    }
    for(i=4; i>=1; i--)
    {
        for(j=5; j>i; j--)
        {
            printf(" ");
        }
        printf("*");
        for(j=1; j<(i-1)*2; j++)
        {
            printf(" ");
        }
        if(i==1)
            printf("\n");
        else
            printf("*\n");
    }
}

```

```
return 0;
```

```
}
```

Program41: C Program to print the following Pattern.

```
*
 *
*  *
 *  *
*   *
 *   *
*    *
 *    *
*     *
 *     *
*      *
```

```
#include<stdio.h>
#include<conio.h>
int main()
{
    char prnt = '*';
    int i, j, k, s, p, r, nos = 7;
    for (i = 1; i <= 5; i++)
    {
        for (j = 1; j <= i; j++)
        {
            if ((i % 2) != 0 && (j % 2) != 0)
            {
                printf("%3c", prnt);
            }
            else if ((i % 2) == 0 && (j % 2) == 0)
            {
                printf("%3c", prnt);
            }
            else
            {
                printf(" ");
            }
        }
        for (s = nos; s >= 1; s--)
        {
            printf(" ");
        }
        for (k = 1; k <= i; k++)
        {
            if (i == 5 && k == 1)
            {
                continue;
            }
            if ((k % 2) != 0)
            {
                printf("%3c", prnt);
            }
            else
            {
                printf(" ");
            }
        }
    }
}
```

```
        }
        printf("\n");
        nos = nos - 2;
    }
    nos = 1;
    for (p = 4; p >= 1; p--)
    {
        for (r = 1; r <= p; r++)
        {
            if ((p % 2) != 0 && (r % 2) != 0)
            {
                printf("%3c", prnt);
            }
            else if ((p % 2) == 0 && (r % 2) == 0)
            {
                printf("%3c", prnt);
            }
            else
            {
                printf(" ");
            }
        }
        for (s = nos; s >= 1; s--)
        {
            printf(" ");
        }
        for (k = 1; k <= p; k++)
        {
            if ((k % 2) != 0)
            {
                printf("%3c", prnt);
            }
            else
            {
                printf(" ");
            }
        }
        nos = nos + 2;
        printf("\n");
    }
    return 0;
}
```

Program42: Write a C program to print the following pattern

```
  *
 * * *
* * * * *
* * * * * * *
* * * * * * * *
 * * * * *
  * * * * *
```

```
    * * *
      *
    * * *
* * * * *

#include<stdio.h>
#include<conio.h>
int main()
{
    char prnt = '*';
    int i, j, k, s, nos = 4;
    for (i = 1; i <= 5; i++)
    {
        for (s = nos; s >= 1; s--)
        {
            printf(" ");
        }
        for (j = 1; j <= i; j++)
        {
            printf("%2c", prnt);
        }
        for (k = 1; k <= (i - 1); k++)
        {
            if (i == 1)
            {
                continue;
            }
            printf("%2c", prnt);
        }
        printf("\n");
        nos--;
    }
    nos = 1;
    for (i = 4; i >= 1; i--)
    {
        for (s = nos; s >= 1; s--)
        {
            printf(" ");
        }
        for (j = 1; j <= i; j++)
        {
            printf("%2c", prnt);
        }
        for (k = 1; k <= (i - 1); k++)
        {
            printf("%2c", prnt);
        }
        nos++;
        printf("\n");
    }
    nos = 3;
    for (i = 2; i <= 5; i++)
```

```

    {
        if ((i % 2) != 0)
        {
            for (s = nos; s >= 1; s--)
            {
                printf(" ");
            }
            for (j = 1; j <= i; j++)
            {
                printf("%2c", prnt);
            }
        }
        if ((i % 2) != 0)
        {
            printf("\n");
            nos--;
        }
    }
    return 0;
}

```

Program43: Write a C program to print the following pattern

```

*****
*****
***
**
*
**
***
*****
*****

```

```

#include<stdio.h>
#include<conio.h>
int main()
{
    char prnt = '*';
    int i, j, k, s, nos = -1;
    for (i = 5; i >= 1; i--)
    {
        for (j = 1; j <= i; j++)
        {
            printf("%2c", prnt);
        }
        for (s = nos; s >= 1; s--)
        {
            printf(" ");
        }
        for (k = 1; k <= i; k++)
        {
            if (i == 5 && k == 5)
            {
                continue;
            }
        }
    }
}

```

```

    }
    printf("%2c", prnt);
}
nos = nos + 2;
printf("\n");
}
nos = 5;
for (i = 2; i <= 5; i++)
{
    for (j = 1; j <= i; j++)
    {
        printf("%2c", prnt);
    }
    for (s = nos; s >= 1; s--)
    {
        printf(" ");
    }
    for (k = 1; k <= i; k++)
    {
        if (i == 5 && k == 5)
        {
            break;
        }
        printf("%2c", prnt);
    }
    nos = nos - 2;
    printf("\n");
}
return 0;
}

```

Program44: Write a C program to print the following pattern.

```

* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
*

```

```

#include<stdio.h>
#include<conio.h>
int main()
{
    char prnt = '*';
    int i, j, k, s, sp, nos = 0, nosp = -1;
    for (i = 9; i >= 3; (i = i - 2))
    {
        for (s = nos; s >= 1; s--)
            printf(" ");
        for (j = 1; j <= i; j++)

```



```

        printf("%2c", prnt);
    for (sp = nosp; sp >= 1; sp--)
        printf(" ");
    for (k = 1; k <= i; k++)
    {
        if (i == 9 && k == 1)
            continue;
        printf("%2c", prnt);
    }
    nos++;
    nosp = nosp + 2;
    printf("\n");
}
nos = 4;
for (i = 9; i >= 1; (i = i - 2))
{
    for (s = nos; s >= 1; s--)
        printf(" ");
    for (j = 1; j <= i; j++)
    {
        printf("%2c", prnt);
    }
    nos++;
    printf("\n");
}
return 0;
}

```

Program45: Write a C program to print a character rhombus.

```

    A
  ABA
ABCBA
ABCD CBA
ABCBA
  ABA
    A

```

```

#include<stdio.h>
#include<conio.h>
int main()
{
    char ch,r,c,chw;
    int sp;
    printf("\nEnter any character : ");
    scanf("%c",&ch);
    if(ch>='a' && ch<='z')
        chw=ch-32;
    printf("\n");
    for(r='A'; r<=chw; r++)
    {
        for(sp=chw-r; sp>=1; sp--)
            printf(" ");
        for(c='A'; c<=r; c++)

```

```

        printf("%c",c);
        for(c=r-1; c>='A'; c--)
            printf("%c",c);
        printf("\n");
    }
    for(r='A'; 'A'<=ch; ch--,r++)
    {
        for(sp=r; sp>='A'; sp--)
            printf(" ");
        for(c='A'; c<=ch-1; c++)
            printf("%c",c);
        for(c=ch-2; c>='A'; c--)
            printf("%c",c);
        printf("\n");
    }
    getch();
    return 0;
}

```

Program46: C program to print a pattern like below

```

*
* *
*  *
*   *
*    *
#include<stdio.h>
#include<conio.h>
void main()
{
    int i,j,k;
    for (i=1; i<=10; i+=2)
    {
        for(k=10; k > i; k-=2) /* small inner loop for the spaces */
        {
            printf(" ");
        }
        for (j=0; j < i; j++)
        {
            if(j==0)
                printf("*");
            else if(j==i-1)
                printf("*");
            printf(" ");
        }
        printf("\n");
    }
}

```

Program47: write a C program to display the character in following fashion

A
AB

```

      ABC
     ABCD
    ABCDE
   ABCD
  ABC
 AB
 A
#include<stdio.h>
#include<conio.h>
int main()
{
    char ch,r,c;
    int sp;
    printf("\nEnter last character of triangle : ");
    scanf("%c",&ch);
    if(ch>='a' && ch<='z')
        ch=ch-32;
    printf("\n");
    for(r='A'; r<=ch; r++)
    {
        for(sp=ch-r; sp>=1; sp--)
            printf(" ");
        for(c='A'; c<=r; c++)
            printf("%c",c);
        printf("\n");
    }
    for(r='A'; 'A'<=ch-1; ch--,r++)
    {
        for(sp=r; sp>='A'; sp--)
            printf(" ");
        for(c='A'; c<ch; c++)
            printf("%c",c);
        printf("\n");
    }
    getch();
    return 0;
}

```

Program48: C Program to print the following Pattern.

```

A B C D E F G H
A B C D E F G
A B C D E F
A B C D E
A B C D
A B C
A B
A

```

```

#include<stdio.h>
void main()
{
    char ch = 'A';
    int n, c, k, space = 0;

```

```
scanf("%d", &n);
for ( k = n ; k >= 1 ; k-- )
{
    for ( c = 1 ; c <= space ; c++)
        printf(" ");
    space++;
    for ( c = 1 ; c <= k ; c++ )
    {
        printf("%c ", ch);
        ch++;
    }
    printf("\n");
    ch = 'A';
}
}
```

Program49: write a C program to display the character in following fashion

```
A
BA
CBA
DCBA
EDCBA
DCBA
CBA
BA
A
#include<stdio.h>
#include<conio.h>
void main()
{
    char ch,r,c;
    int sp;
    printf("\nEnter last character of triangle : ");
    scanf("%c",&ch);
    if(ch>='a' && ch<='z')
        ch=ch-32;
    printf("\n");
    for(r='A'; r<=ch; r++)
    {
        for(c=r; c>='A'; c--)
            printf("%c",c);
        printf("\n");
    }
    for(r='A'; 'A'<=ch; ch--,r++)
    {
        for(c=ch-1; c>='A'; c--)
            printf("%c",c);
        printf("\n");
    }
    getch();
}
```

Program50: To find the GCD and LCD of given two integer numbers

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int n1, n2, prod, gcd, lcm ;
    clrscr() ;
    printf("Enter the two numbers : ") ;
    scanf("%d %d", &n1, &n2) ;
    prod = n1 * n2 ;
    while(n1 != n2)
    {
        if(n1 > n2)
            n1 = n1 - n2 ;
        if(n2 > n1)
            n2 = n2 - n1 ;
    }
    gcd = n1 ;
    lcm = prod / gcd ;
    printf("\nThe GCD is : %d", gcd) ;
    printf("\n\nThe LCM is : %d", lcm);
    getch() ;
}
```

Output: Enter the two numbers : 10 8
The GCD is : 2 The LCM is : 40

LCM Calculation - LCM can be calculated using a simple formula -

$\text{LCM} = \text{Product of numbers} / \text{GCD of numbers} = n1 * n2 / \text{gcd}.$

Program51: Write a C program to compute the sum of first n terms ($n \geq 1$) of the following series using 'for' loop. $1 - 3 + 5 - 7 + 9 - \dots$

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int start=1,i=1,no=0,sum=0;
    clrscr();
    printf ("\nEnter number of terms to be added:-> ");
    scanf("%d",&no);
    for (i=1;i<=no;i++)
    {
        if (i%2!=0)
        {
            sum+=start;
            if (i==1)
                printf ("%d",start);
            else
                printf ("+%d",start);
        }
        else
        {
            sum-=start;
            printf ("- %d",start);
        }
    }
}
```

```

        start+=2;
    }
    printf ("%d",sum);
    getch();
}

```

Program52: Write a C Program to print 15 terms of 1 , 2 , 4 , 7 , 11 , 16....(1 + 2 [1 + 1] + 4 [2 + 2] + 7 [4 + 3] + 11 [7 + 4] + 16 [11 + 5] + 22[16 + 6] + ...)

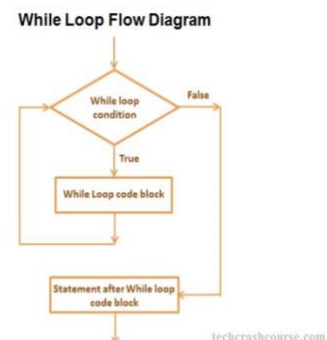
```

#include<stdio.h>
#include<conio.h>
void main()
{
    clrscr();
    int i,j,k=0;
    int m=1;
    printf("Enter The Number of Terms For The Series:\n");
    scanf("%d",&i);
    for(j=0;j<i;j++)
    {
        m=m+k;
        printf("%d+",m);
        k++;
    }
    getch();
}

```

while: Repeats a block of code multiple times until a given condition is true. Unlike for loop, while loop doesn't contain initialization and update expression in it's syntax. The syntax of the while loop is

Syntax: **while (condition)**
 {
 Body of the loop
 }



Where

1. while is keyword
2. **condition:** The Test condition is evaluated and if the condition is true, then the body of the loop is executed. After execution of the body, the test condition is once again evaluated and if it is true, the body is executed once again. This process of repeated execution of the body continues until the test condition is true. The test condition must be enclosed with in parentheses.

3. **Body of the loop:** The body of the loop may have one or more statements. The braces are needed only if the body contains two or more statements.

Example1: Write a C program to print sum of n numbers using while loop.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int i = 1,sum = 0,n;
    printf("enter the value of n");
    scanf("%d",&n);
    while(i<=n)
    {
        sum = sum + i;
        i = i + 1;
    }
    printf("Total : %d ",sum);
}
```

Example2: Write a C program to print sum of individual digits of a positive integer using while loop.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int n, k, sum=0;
    printf("enter the value of n");
    scanf("%d",&n);
    while(n!=0)
    {
        k=n%10;
        sum=sum+k;
        n=n/10;
    }
    printf("value of sum is %d",sum);
    getch();
}
```

Example3: Write a C program to reverse a given number using while loop.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int n, k, rev=0;
    printf("enter the value of n");
    scanf("%d",&n);
    while(n!=0)
    {
        k=n%10;
        rev=rev*10+k;
        n=n/10;
    }
}
```

```

        printf("value of reverse is %d", rev);
        getch();
    }

```

Example4: Write a C program to find whether a number is palindrome or not using while loop.

```

#include<stdio.h>
#include<conio.h>
void main()
{
    int v, n, k, rev=0;
    printf("enter the value of n");
    scanf("%d",&n);
    v=n;
    while(n!=0)
    {
        k=n%10;
        rev= rev*10+k;
        n=n/10;
    }
    if(rev==v)
    {
        printf("number is palindrome ");
    }
    else
    {
        printf("number is not palindrome");
    }
    getch();
}

```

Example5: Write a C program to find whether a no is Armstrong or not using while loop.

```

#include<stdio.h>
#include<conio.h>
void main()
{
    int n, k,v,sum=0;
    printf("enter the value of n");
    scanf("%d",&n);
    v=n;
    while(n!=0)
    {
        k=n%10;
        sum=sum+(k*k*k);
        n=n/10;
    }
    if(sum==v)
    {
        printf("number is Armstrong");
    }
    else
    {

```



```

        printf("number is not Armstrong");
    }
    getch();
}

```

Example6: Write a C program to find the factorial of a number using while loop.

```

#include<stdio.h>
#include<conio.h>
void main()
{
    int n, f=1;
    printf("enter the value of n");
    scanf("%d",&n);
    while(n!=0)
    {
        f=f*n;
        n--;
    }
    printf("result is %d",f);
    getch();
}

```

Example7: Write a C program to convert binary number into decimal number using while loop.

```

#include<stdio.h>
#include<conio.h>
void main()
{
    int bnum, digit, decimal=0, bin, base=0;
    clrscr();
    printf("enter binary number");
    scanf("%d",&bnum);
    printf("%d",bnum);
    bin=bnum;
    while(bnum!=0)
    {
        digit=bnum%10;
        decimal=decimal+(digit<<base);
        base=base+1;
        bnum=bnum/10;
    }
    printf("The binary equivalent is %d in decimal is %d",bin,decimal);
    getch();
}

```

Example8: Write a C program to print Fibonacci series using while loop.

```

#include<stdio.h>
#include<conio.h>
void main()
{
    int a=0, b=1, n, f, i=1;
    printf("enter the value of n");
    scanf("%d",&n);

```

```

        printf("%d %d",a,b);
        while(i<=n-2)
        {
            f=a+b;
            printf("%4d",f);
            a=b;
            b=f;
            i++;
        }
        getch();
    }

```

Example9: Write a C program to print the prime numbers from 1 to n using while loop.

```

#include<stdio.h>
#include<conio.h>
void main()
{
    int n, c, j, i=1;
    printf("enter the value of n");
    scanf("%d",&n);
    while(i<=n)
    {
        c=0;
        j=1;
        while(j<=i)
        {
            if(i%j==0)
            {
                c++;
            }
            j++;
        }
        if(c==2)
        {
            printf("%5d", i);
        }
        i++;
    }
    getch();
}

```

Example10: Write a C program to mask most significant digit of a given number using while loop.

```

#include<stdio.h>
#include<conio.h>
#include<math.h>
void main()
{
    int n,x,c=0,num;
    printf("Enter a number");
    scanf("%d",&n);
    num=n;

```

```

        while(n>0)
        {
            c++;
            n=n/10;
        }
        x = num % (int)pow(10,c-1);
        printf("\nResult=%d",x);
        getch();
    }

```

Program11: C Program to Find Number of Digits in a Number

```

#include <stdio.h>
#include<conio.h>
void main()
{
    int n,count=0;
    printf("Enter an integer: ");
    scanf("%d", &n);
    while(n!=0)
    {
        n=n/10;
        ++count;
    }
    printf("Number of digits: %d",count);
    getch();
}

```

Output: Enter an integer: 34523
Number of digits: 5

Program12: C program to calculate the power of an integer

```

#include <stdio.h>
#include<conio.h>
void main()
{
    int base, exp;
    long long int value=1;
    printf("Enter base number and exponent respectively: ");
    scanf("%d%d", &base, &exp);
    while (exp!=0)
    {
        value = value*base;
        --exp;
    }
    printf("Answer = %d", value);

    getch();
}

```

Output: Enter base number and exponent respectively: 3
4
Answer = 81

Program13: Extract digits from integer in c language

```

#include<stdio.h>
#include<conio.h>
void main()
{
    int num,temp,factor=1;
    printf("Enter a number: ");
    scanf("%d",&num);

    temp=num;
    while(temp)
    {
        temp=temp/10;
        factor = factor*10;
    }

    printf("Each digits of given number are: ");
    while(factor>1)
    {
        factor = factor/10;
        printf("%d ",num/factor);
        num = num % factor;
    }
    getch();
}

```

Output: Enter a number: 123
Each digits of given number are: 1 2 3

Program14: C program to find Prime factor of a number.

```

#include<stdio.h>
#include<conio.h>
void main()
{
    int num,i=1,j,k;
    printf("\nEnter a number:");
    scanf("%d",&num);
    while(i<=num)
    {
        k=0;
        if(num%i==0)
        {
            j=1;
            while(j<=i)
            {
                if(i%j==0)
                k++;
                j++;
            }
            if(k==2)
                printf("\n%d is a prime factor",i);
        }
    }
}

```

```

        i++;
    }
    getch();
}

```

Program15: HCF (Highest common factor) program with two numbers in c

```

#include<stdio.h>
#include<conio.h>
void main()
{
    int n1,n2;
    printf("\nEnter two numbers:");
    scanf("%d %d",&n1,&n2);
    while(n1!=n2)
    {
        if(n1>=n2-1)
            n1=n1-n2;
        else
            n2=n2-n1;
    }
    printf("\nHCF=%d",n1);
    getch();
}

```

Program16: C program to accept a decimal number and convert it to binary and count the number of 1's in the binary number.

```

#include <stdio.h>
#include<conio.h>
void main()
{
    long int num, decimal_num, remainder, base = 1, binary = 0, no_of_1s = 0;
    printf("Enter a decimal integer \n");
    scanf("%ld", &num);
    decimal_num = num;
    while (num > 0)
    {
        remainder = num % 2;
        /* To count no.of 1s */
        if (remainder == 1)
        {
            no_of_1s++;
        }
        binary = binary + remainder * base;
        num = num / 2;
        base = base * 10;
    }
    printf("Input number is = %ld\n", decimal_num);
    printf("Its binary equivalent is = %ld\n", binary);
    printf("No.of 1's in the binary number is = %ld\n",no_of_1s);
    getch();
}

```

Output: Enter a decimal integer 134

Input number is = 134
 Its binary equivalent is = 10000110
 No. of 1's in the binary number is = 3

Program17: C Program to Print Armstrong Number from 1 to 1000.

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int number, temp, digit1, digit2, digit3;
    printf("Print all Armstrong numbers between 1 and
           1000:\n");
    number = 001;
    while (number <= 900)
    {
        digit1 = number - ((number / 10) * 10);
        digit2 = (number / 10) - ((number / 100) * 10);
        digit3 = (number / 100) - ((number / 1000) * 10);
        temp = (digit1 * digit1 * digit1) + (digit2 * digit2
            * digit2) + (digit3 * digit3 * digit3);
        if (temp == number)
        {
            printf("\n Armstrong no is:%d", temp);
        }
        number++;
    }
    getch();
}
```

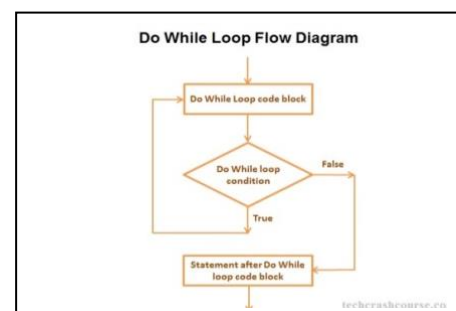
Output: Print all Armstrong numbers between 1 and 1000:

Armstrong no is:1
 Armstrong no is:153
 Armstrong no is:370
 Armstrong no is:371

Armstrong no is:407

do-while: Similar to while loop, but it tests the condition at the end of the loop body. The block of code inside do while loop will execute at least once. The syntax of the do-while loop is :

syntax: **do**
 {
 Body of the loop;
 } while (condition);



1. Where **do** and **while** are two keywords.

2. Here the body of the loop is executed first. At the end of the loop, the condition in the while statement is evaluated. If the condition is true, the program continues to execute the body of the loop once again. This process continues as long as the condition is true. When the condition becomes false, the loop will be terminated and the control goes to the statements that appears immediately after the while statement. Remember the body of the loop is always executed at least once.
3. There is semicolon at the end of while(condition); in do-while loop

Example1:

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int i=3;
    clrscr();
    do
    {
        printf("Hello %d\n", i);
        i = i -1;
    }while ( i > 0 );
    getch();
}
```

Output:
Hello 3
Hello 2
Hello 1

Example2:

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int a = 10;
    do
    {
        printf("value of a: %d\n", a);
        a = a + 1;
    }while( a < 15 );
}
```

Output:
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14

Example3: Write a C program to add all the numbers entered by a user until user enters 0 using do-while loop.

```
#include <stdio.h>
#include <conio.h>
void main()
{
```

```

int sum=0, num;
do
{
    printf("Enter a number\n");
    scanf("%d",&num);
    sum=sum+num;
}while(num!=0);
printf("sum=%d",sum);
}

```

Output: Enter a number 3
Enter a number -2
Enter a number 0
sum=1

Example4: Write a C program to print sum of individual digits of a positive integer using do-while loop.

```

#include<stdio.h>
#include<conio.h>
void main()
{
    int n, k, sum=0;
    printf("enter the value of n");
    scanf("%d", &n);
    do
    {
        k=n%10;
        sum=sum+k;
        n=n/10;
    } while(n!=0);
    printf("value of sum is %d",sum);
    getch();
}

```

Example5: Write a C program to print the prime numbers from 1 to n using do-while loop.

```

#include<stdio.h>
#include<conio.h>
void main()
{
    int n, c, j, i=1;
    printf("enter the value of n");
    scanf("%d",&n);
    do
    {
        c=0;
        for(j=1;j<=i;j++)
        {
            if(i%j==0)
            {
                c++;
            }
        }
        if(c==2)
    }
}

```



```

        {
            printf("%5d", i);
        }
        i++;
    } while(i<=n);
    getch();
}

```

Example6: Write a C program to print Fibonacci series using do-while loop.

```

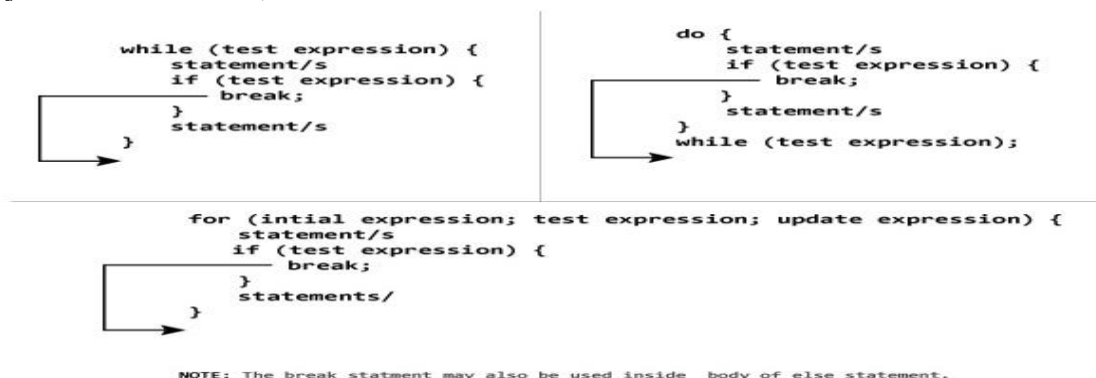
#include<stdio.h>
#include<conio.h>
void main()
{
    int a=0, b=1, n, f, i=1;
    printf("enter the value of n");
    scanf("%d", &n);
    printf("%d %d", a, b);
    do
    {
        f=a+b;
        printf("%4d", f);
        a=b;
        b=f;
        i++;
    } while(i<=n-2);
    getch();
}

```

Unconditional Control Statements (or) Loop Control and Jump Statements: Loop control statements alter the normal execution path of a program. Loop control statements are used when we want to skip some statements inside loop or terminate the loop immediately when some condition becomes true.

break: The break statement is used to stop the execution of loop and switch case statements. It means we can use break statement inside any loop(for, while and do-while). It terminates the loop immediately and program control resumes at the next statement following the loop. We can use break statement to terminate a case in the switch statement.

Syntax : **break;**



Example1: `#include<stdio.h>`
`#include<conio.h>`
`void main()`
`{`
`int i;`
`clrscr();`
`for(i=1;i<10 ; i++)`
`{`
`if(i>5)`
`break;`
`printf("%d\t",i); // 5 times only`
`}`
`getch();`
`}`

Output: 1 2 3 4 5

Example2: `#include<stdio.h>`
`#include<conio.h>`
`void main()`
`{`
`int i;`
`int j = 10;`
`clrscr();`
`for(i = 0; i <= j; i ++)`
`{`
`if(i == 5)`
`{`
`break;`
`}`
`printf("Hello %d\n", i);`
`}`
`getch();`
`}`

Example3: `#include <stdio.h>`
`#include<conio.h>`
`void main()`
`{`
`int i = 10;`
`clrscr();`
`do`
`{`
`printf("Hello %d\n", i);`
`i = i -1;`
`if(i == 6)`
`{`
`break;`
`}`
`}while (i > 0);`
`getch();`
`}`

Output: Hello 10
Hello 9
Hello 8
Hello 7

Example4:

```
#include <stdio.h>
#include<conio.h>
void main()
{
    /* local variable definition */
    int a = 10;
    /* while loop execution */
    while( a < 20 )
    {
        printf("value of a: %d\n", a);
        a++;
        if( a > 15)
        {
            /* terminate the loop using break statement */
            break;
        }
    }
    getch();
}
```

Output: value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15

Example5:

```
#include <stdio.h>
int main()
{
    int num =0;
    while(num<=100)
    {
        printf("variable value is: %d", num);
        if (num==2)
        {
            break;
        }
        num++;
    }
    printf("Out of while-loop");
    return 0;
}
```

Output: variable value is: 0
variable value is: 1
variable value is: 2
Out of while-loop

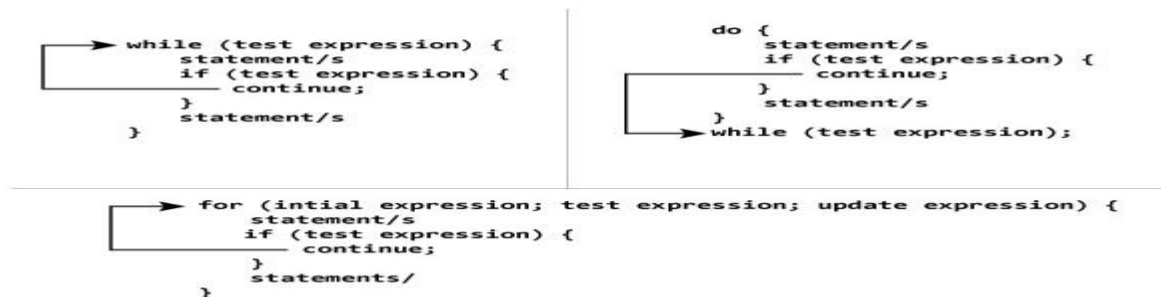
Example6:

```
#include <stdio.h>
int main()
{
    int var;
    for (var=100;var>=10;var--)
    {
        printf("var: %d", num);
        if (var==99)
        {
            break;
        }
    }
    printf("Out of for-loop");
    return 0;
}
```

Output: var: 100 var: 99 Out of for-loop

continue: The continue statement is used for skipping part of loop's body. It means We can use continue statement inside any loop(for, while and do-while). It skips the remaining statements of loop's body and starts next iteration.

Syntax: **continue;**



NOTE: The continue statement may also be used inside body of else statement.

Example1:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int i;
    int j = 8;

    for( i = 0; i <= j; i ++ )
    {
        if( i == 5 )
        {
            continue;
        }
        printf("Hello %d\n", i );
    }
    getch();
}
```

Output: Hello 0
Hello 1
Hello 2
Hello 3
Hello 4
Hello 6
Hello 7
Hello 8

Example2: Write a C program to find the product of 4 integers entered by a user. If user enters 0 skip it.

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int i,num,product;
    for(i=1,product=1;i<=4;++i)
    {
        printf("Enter num%d:",i);
        scanf("%d",&num);
        if(num==0)
            continue; /*In this program, when num equals to zero, it skips the
                        statement product*=num and continue the loop. */
        product*=num;
    }
    printf("product=%d",product);
    getch();
}
```

Output: Enter num1:3
Enter num2:0
Enter num3:-5
Enter num4:2
product=-30

Example3:

```
#include <stdio.h>
#include <conio.h>
void main()
{
    /* local variable definition */
    int a = 12;
    clrscr();
    /* do loop execution */
    do
    {
        if( a == 15)
        {
            /* skip the iteration */
            a = a + 1;
            continue;
        }
        printf("value of a: %d\n", a);
    }
```

```

        a++;
    }while( a < 18 );
    getch();
}

```

Output: value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17

Example4: continue statement inside for loop

```

#include<stdio.h>
#include<conio.h>
void main()
{
    int j;
    clrscr();
    for ( j=0; j<=8; j++)
    {
        if (j==4)
        {
            continue;
        }
        printf("%d ", j);
    }
    getch();
}

```

Output: 0 1 2 3 5 6 7 8

Example5: #include <stdio.h>

```

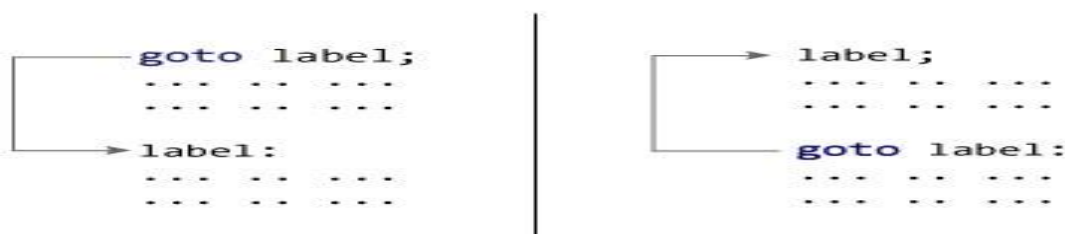
#include<conio.h>
void main()
{
    int j=3;
    do
    {
        if (j==7)
        {
            j++;
            continue;
        }
        printf("\nvalue of j: %d", j);
        j++;
    }while(j<10);
    getch();
}

```

Output: value of j: 3
value of j: 4
value of j: 5
value of j: 6
value of j: 8
value of j: 9

goto:

1. C supports the “goto” statement to jump unconditionally from one point to another in the program.
2. The goto requires a label in order to identify the place where the jump is to be made.
3. A label is any valid variable name and must be followed by a colon (:).
4. The label is placed immediately before the statement where the control is to be transferred.
5. The label can be anywhere in the program either before or after the goto label statement.
6. During running of a program, when a statement like “goto begin;” is met, the flow of control will jump to the statement immediately following the label “begin:” this happens unconditionally.
7. goto breaks the normal sequential execution of the program.
8. If the “label:” is before the statement “goto label;” a loop will be formed and some statements will be executed repeatedly. Such a jump is known as a „backward jump“.
9. If the “label:” is placed after the “goto label;” some statements will be skipped and the jump is known as a “forward jump”.

**Example1:**

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int x;
    clrscr( );
    printf("Enter a Number:");
    scanf("%d", &x);
    if(x%2==0)
    {
        goto even;
    }
    else
    {
        goto odd;
    }
    even: printf("\n %d is Even Number");
        return;
    odd: printf("\n %d is Odd Number");
        getch();
}
```

Output: Enter a Number: 55

55 is Odd Number.

Example2:

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int a = 12;
    LOOP:do
    {
        if( a == 15)
        {
            /* skip the iteration */
            a = a + 1;
            goto LOOP;
        }
        printf("value of a: %d\n", a);
        a++;
    }while( a < 18);
    getch();
}
```

Output:

```
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
```

Example3:

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int age;
    clrscr();
    Vote: printf("you are eligible for voting");
    NoVote: printf("you are not eligible to vote");
    printf("Enter you age:");
    scanf("%d", &age);
    if(age>=18)
        goto Vote;
    else
        goto NoVote;

    getch();
}
```

return: The return statement forces a return from a function and can be used to transfer a value back to the calling routine. It has these two forms:

syntax:

```
return;
Or
return value;
Or
return expression;
```


Example1:

```
#include <stdio.h>
#include<conio.h>
void main()
{
    int number;
    clrscr();
    printf("Enter an integer\n");
    scanf("%d",&number);
    printf("Integer entered by you is %d\n", number);
    return;
}
```

Output: Enter a number 5
Number entered by you is 5

Example2:

```
#include <stdio.h>
#include<conio.h>
int main()
{
    int number;
    clrscr();
    printf("Enter an integer\n");
    scanf("%d",&number);
    printf("Integer entered by you is %d\n", number);
    return 0;
}
```

Output: Enter a number 5
Number entered by you is 5

Example3:

```
#include <stdio.h>
#include<conio.h>
int sum();
void main()
{
    int addition;
    addition = sum();
    printf("\nSum of two given values = %d", addition);
    return;
}
int sum()
{
    int a = 50, b = 80, sum;
    sum = a + b;
    return sum;
}
```

Output: Sum of two given values = 130

UNIT IV ARRAYS

Importance of an array in c language:

So far we have used only the fundamental data types, namely char, int, float, double and variations of int and double. Although these types are very useful, they can be used only to handle limited amount of data. In many applications, however, we need to handle a large volume of data in terms of reading, processing and printing. To process such large amount of data, we need a powerful data type that would facilitate efficient storing, accessing and manipulation of data items. C Language supports a derived data type known as array that can be used for such applications

Need of arrays while writing c programs:

Consider a problem that is a program that can print its input in reverse order. If there are two values, this is easy and the program is

```
main()
{
    int v1,v2;
    printf("enter two values");
    scanf("%d%d",&v1,&v2);
    printf("%d%d",v2,v1);
}
```

If there are three values, this is still relatively easy. But what if there are ten Or twenty Or one hundred values? Then it is difficult.

Consider another problem, the average of n integer numbers given by the user can easily computed as follows.

```
main()
{
    int count=0,s=0,n,num;
    float avg;
    printf("how many numbers")
    scanf("%d",&n);
    for(count=1;count<=n;count++)
    {
```

```

        printf("enter the number\n");
        scanf("%d",&num);
        s=s+num;
    }
    avg=(float)s/n;
    printf("avarge is%f:",avg);
}

```

Now the problem is given as print the numbers that are greater than the average, then one solution is to read the numbers twice. That is,

1. Read in all the numbers and calculate the average
2. Read in all the numbers again, this time checking each as it is read against a previously calculated average

If input is from the keyboard, then the user has to enter each number twice and accurately, with no mistakes. This is not a viable solution. Because, for 25 numbers entered, the user has to remember all the numbers. But what if there are 50 Or 100 numbers? Then, it is not easy. To solve this problem, an array is required. It is a collection of numbered elements.

Definition: An array is a fixed-size sequenced collection of elements of the same data type that shares a common name.

Or

An array is a collection of individual data elements that is ordered, fixed in size, and homogeneous.

Or

Array is a group of Homogeneous data type elements which shares a common name.

Each individual data item is known as an element of the array. The elements of the array are stored in subsequent memory locations starting from the memory location given by the array name. Each individual data item in the Array is referenced by a subscript (or index) enclosed in a pair of square brackets i.e. []. This subscript indicates the position of an individual data item in an Array. It is simply a grouping of like-type data. Generally array can be used to represent a list of numbers, **or** a list of names.

Some examples where an array can be used:

1. List of temperatures recorded every hour in a day, Or a month, Or a year.
2. List of employees in an organization.
3. List of products and their cost sold by a store.
4. Test scores of class students.
5. List of customers and their telephone numbers.

As we mentioned above, an array is sequenced collection of similar data items that shares a common name. For instance, we can use array name as salary to represent a set of salaries of group of employees in an organization. We can refer individual salaries by writing a number called index Or subscript in

brackets after the array name.

Example: salary [10]

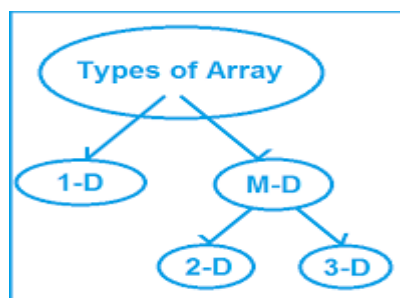
Represents the salary of the 10th employee. While the complete set of values is referred to as an array, the individual values are called elements. Array can be of any variable type.

The ability to use a single name to represent a collection of items and refer to an item by specifying the item number enables us to develop concise and efficient programs.

Array Properties:

1. The type of an array is the data type of its elements
2. The location of an array is location of its first element
3. The length of an array is the number of data elements in the array.
4. The size of an array is the length of the array times the size of an element.

Types of arrays: An array may be one-dimensional or multi-dimensional



One-dimensional array: A list of items can be given by one variable name using only one subscript and such a variable is called a single – subscripted variable (or) a one-dimensional array. It is also known as vector.

Declaration: Every array must be declared before use like other variables

Syntax: data_type array_name[size];

- **data_type:** It is the type of elements that an array stores, Array may have any of the data types like int, float, char, double.... If array stores character elements then type of array is 'char'. If array stores integer elements then type of array is 'int'.
- **array_name:** It is an identifier represents the array name.
- **size:** The array Size must be an integer constant greater than zero, which represents the total number of elements in the array.

The base address of an array is the address of zeroth element(starting element) of the array .When an array is declared, the compiler allocates a base address and reserves enough space in memory for all the elements of an array.In c, the array name represents this base address

Example1: int num[5];

It declares integer array num having maximum of 5 num of int data type

Example2: float x[5];

It is the declaration of one dimensional array. It defines float array of x of size 5 that represents a block of 5 consecutive storage regions. Here each element in the array is referred to by an array variables x[0],x[1],x[2],x[3],x[4] where 0,1,2,3,4 represent subscripts or indices(index) of an array.

- An array variable is also known as subscripted variable.
- By default the array subscripts start from zero in c and they are integers.
- X[0]→ starting element
- X[4]→last element

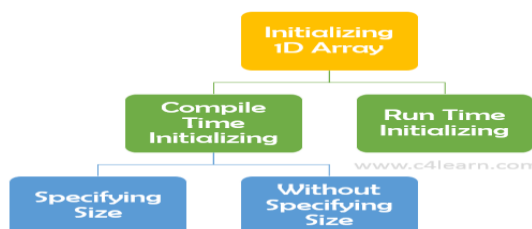
Example3: double list[10],name[15];

It declares two arrays, named list and name having 10 and 15 elements respectively of double data type.

Example4: char str[10];

It declares character array str having maximum of 20charactersnof char data type.

Initialization: Assigning the required information to a variable before processing is called initialization.



Initialization of one-dimensional arrays: Assigning the required information to a variable before processing is called initialization. We can initialize the individual elements of an array. Array elements can be initialized at the time of declaration.

Array elements are enclosed in braces and separated by comma.

Example: int a[5] = { 10,15,20,25,30};

Ways Of Array Initializing 1-D Array:

1. **Compile time initialization:**
 - Size is Specified Directly
 - Size is Specified Indirectly

2. **Runtime initialization**

Compile time initialization:**Size is Specified Directly:**

- ✓ Arrays can be initialized at the time of declaration when their initial values are known in advance.
- ✓ Array elements can be initialized with data items of type integer, character, float etc.

Integer array initialization:

Syntax: `int num[5] = {2,8,7,6,0};`

In the above example we have specified the size of array as 5 directly in the initialization statement. Compiler will assign the set of values to particular element of the array.

```
num[0] = 2
num[1] = 8
num[2] = 7
num[3] = 6
num[4] = 0
```

As at the time of compilation all the elements are at Specified Position So This Initialization Scheme is Called as “Compile Time Initialization”.

Graphical Representation:



If the size of integer is 2 bytes, 10 bytes will be allocated for the variable num

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int i;
    int arr[3]={2,3,4}; //Compile time array initialization
    for(i=0 ; i<3 ; i++)
    {
        printf("%d\t",arr[i]);
    }
    getch();
}
```

Output: 2 3 4

Partial array initialization:

Syntax: `int a[10] = {0, 1, 2, 3, 4, 5, 6};`

In the above example we have specified the size of array as 10 directly in the initialization statement.

Compiler will assign the set of values to particular element of the array from left to right, and the remaining elements are automatically initialized to 0(would initialize a[7], a[8], and a[9] to 0).

```
num[0] = 0
num[1] = 1
num[2] = 2
num[3] = 3
num[4] = 4
num[5] = 5
num[6] = 6
num[7] = 0
num[8] = 0
num[9] = 0
```

character initialization:

`char b[9] = { 'C', 'O', 'M', 'P', 'U', 'T', 'E', 'R' };`

b[0] b[1] b[2] b[3] b[4] b[5] b[6] b[7] b[8]

C	O	M	P	U	T	E	R	\0
---	---	---	---	---	---	---	---	----

If character occupies 1 byte, 9 bytes will be allocated for variable b.in these 8 for characters and one for terminating null.

String array initialization:

```
char b[9] ="COMPUTER";
```

b[0] b[1] b[2] b[3] b[4] b[5] b[6] b[7] b[8]

C	O	M	P	U	T	E	R	\0
---	---	---	---	---	---	---	---	----

If character occupies 1 byte, 9 bytes will be allocated for variable b.in these 8 for characters and one for terminating null.

Points on Array initialization:

1. If the number of values to be initialized is less than the size of the array, then the elements are initialized in the order from 0th location.
2. The remaining locations will be initialized to zero automatically.

Ex: int a[5] = { 10, 15 };

a[0]	a[1]	a[2]	a[3]	a[4]
10	15	0	0	0

Size is Specified Indirectly: In this scheme of compile time Initialization, We does not provide size to an array but instead we provide set of values to the array.

Integer array initialization:

Syntax: int num[]={2,8,7,6,0};

- ❖ Compiler Counts the Number Of Elements Written Inside Pair of Braces and Determines the Size of An Array.
- ❖ After counting the number of elements inside the braces, The size of array is considered as 5 during complete execution.
- ❖ This type of Initialization Scheme is also Called as “Compile Time Initialization”

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int i;
    int arr[]={2,3,4}; //Compile time array initialization
    for(i=0 ; i<3 ; i++)
```

```

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

```

Output: 2 3 4

Character array initialization:

Syntax: `char b[] = { 'C','O','M','P','U','T','E','R' };`

❖ In this declaration, even though we have not specified exact number of elements to be used in array b, the array size will be set to the total number of initial values specified plus one.

❖ So, the array size will be set to 9 automatically. The array b is initialized as shown below.

b[0] b[1] b[2] b[3] b[4] b[5] b[6] b[7] b[8]

C	O	M	P	U	T	E	R	\0
---	---	---	---	---	---	---	---	----

String array initialization:

Syntax: `char b[]="COMPUTER";`

Even though the string "computer" contains 8 characters, because it is a string it always ends with null character. So, the array size is 9 bytes.

b[0] b[1] b[2] b[3] b[4] b[5] b[6] b[7] b[8]

C	O	M	P	U	T	E	R	\0
---	---	---	---	---	---	---	---	----

Run time Initialization: An array can also be initialized at runtime using scanf() function. This approach is usually used for initializing large array, or to initialize array with user specified values.

Consider the declaration shown below:

```
int a[5];
```

Here, 5 memory locations are reserved and each item in the memory location can be accessed by specifying the index as shown below:

Using a[0] through a[4] we can access 5 data items.

By reading the n data items from keyboard using scanf function

```

scanf ("%d", &a[0]);
scanf ("%d", &a[1]);
scanf ("%d", &a[2]);
scanf ("%d", &a[3]);
scanf ("%d", &a[4]);

```

In general, scanf ("%d", &a[i]); where i = 0,1,2,3,-----,n-1

If we want to read n data items from the keyboard the following statement can be used:

```
for (i=0; i<=n-1;i++)
```



```

    {
        scanf ("%d", &a[i]);
    }

```

Example:

```

#include <stdio.h>
void main()
{
    int n, a[10], i;
    printf (" Enter the no of items");
    scanf ("%d", &n);
    printf ("Enter n elements");
    for (i = 0; i<n; i++)
    {
        scanf ("%d", &a[i]);
    }
    printf (" the n elements are");
    for (i=0; i<n; i++)
    {
        printf ("%d\n", a[i]);
    }
}

```

Reading (storing) and writing (accessing) from a one dimensional array:

Reading and writing of integers arrays:

In this section, let us concentrate on how to read the data from the keyboard and how to display data items stored in the array. Consider the declaration shown below:

```
int a[5];
```

Here, 5 memory locations are reserved and each item in the memory location can be accessed by specifying the index as shown below:

Using a[0] through a[4] we can access 5 data items.

By reading the n data items from keyboard using scanf function

```

scanf ("%d", &a[0]);
scanf ("%d", &a[1]);
scanf ("%d", &a[2]);
scanf ("%d", &a[3]);
scanf ("%d", &a[4]);

```

In general, scanf ("%d", &a[i]); where i = 0,1,2,3,-----,n-1

If we want to read n data items from the keyboard the following statement can be used:

```

for (i=0; i<=n-1;i++)
{
    scanf ("%d", &a[i]);
}

```

Example:

```

#include <stdio.h>

```

```

void main()
{
    int n, a[10], i;
    printf (" Enter the no of items");
    scanf ("%d", &n);
    printf ("Enter n elements");
    for (i = 0; i<n; i++)
    {
        scanf ("%d", &a[i]);
    }
    printf (" the n elements are");
    for (i=0; i<n; i++)
    {
        printf ("%d\n", a[i]);
    }
}

```

Example:

```

#include <stdio.h>
int main()
{
    int n[ 10 ]; /* n is an array of 10 integers */
    int i,j; /* initialize elements of array n to 0 */
    for ( i = 0; i < 10; i++ )
    {
        n[ i ] = i + 100; /* set element at location i to i + 100 */
    } /* output each array element's value */
    for (j = 0; j < 10; j++ )
    {
        printf("Element[%d] = %d\n", j, n[j] );
    }
    return 0;
}

```

Reading and writing of character arrays: A string is a one dimensional array of characters terminated by a null character. It can be read as a single entity, unlike other types of arrays. The library function as used for reading and writing a string as a whole

Reading from keyboard	Writing to the screen	Remark
scanf() with %s as conversion specification	printf()	scanf() reads a string until a while space is encountered
gets()	puts()	gets() reads a string until a until a new line is encountered

When a string is read using above functions, the null character (\0) is automatically inserted at the end of the string. Hence, the size of the array must be equal to the number of characters pus one. Even though null is not a part of normal text, it is included to mark the end of the string.

Example: Write a program to display character array with their addresses

```

#include<stdio.h>
void main()
{

```

```

char name [10] = { "A", "R", "R", "A", "Y"};
int i=0;
printf (" character memory location \n");
while (name[i]!='\0')
{
    printf ("\n [%c] \t \t [%u]", name[i], &name[i]);
    i++;
}

```

Output:

character memory location	
[A]	4054
[R]	4055
[R]	4056
[A]	4057
[Y]	4058

Example: C Program to Read Array Elements

```

#include<stdio.h>
main()
{
    int i, arr[50], num;
    printf("\nEnter no of elements :");
    scanf("%d", &num);
    //Reading values into Array
    printf("\nEnter the values :");
    for (i=0;i<num;i++)
    {
        scanf("%d",&arr[i]);
    }
    //Printing of all elements of array
    for (i=0;i<num;i++)
    {
        printf("\narr[%d]=%d",i,arr[i]);
    }
}

```

Example: C Program to Delete an element from the specified location from Array

```

#include<stdio.h>
main()
{
    int arr[30], num, i, loc;
    printf("\nEnter no of elements :");
    scanf("%d", &num);
    //Read elements in an array
    printf("\nEnter %d elements :", num);
    for (i=0;i<num;i++)
    {
        scanf("%d",&arr[i]);
    }
}

```

```

    }
    //Read the location
    printf("\n location of the element to be deleted :");
    scanf("%d",&loc);
    /* loop for the deletion */
    while (loc<num)
    {
        arr[loc-1]=arr[loc];
        loc++;
    }
    num--; // No of elements reduced by 1
    //Print Array
    for (i=0;i<num;i++)
    printf("\n %d",arr[i]);
}

```

Example: C Program to Insert an element in an Array

```

#include<stdio.h>
main()
{
    int arr[30], element, num, i, location;
    printf("\nEnter no of elements :");
    scanf("%d", &num);
    for (i=0;i<num;i++)
    {
        scanf("%d",&arr[i]);
    }
    printf("\nEnter the element to be inserted :");
    scanf("%d", &element);
    printf("\nEnter the location");
    scanf("%d", &location);
    //Create space at the specified location
    for(i=num;i>=location;i--)
    {
        arr[i]=arr[i-1];
    }
    num++;
    arr[location-1] = element;
    //Print out the result of insertion
    for(i=0;i<num;i++)
        printf("n%d", arr[i]);
}

```

Output:

```

Enter no of elements: 5
1 2 3 4 5
Enter the element to be inserted: 6
Enter the location: 2
1 6 2 3 4 5

```

Example: C Program to Copy all elements of an array into Another array

```

#include<stdio.h>
main()
{

```

```

int arr1[30], arr2[30], i, num;
printf("\nEnter no of elements :");
scanf("%d", &num);
//Accepting values into Array
printf("\nEnter the values :");
for(i=0;i<num;i++)
{
    scanf("%d",&arr1[i]);
}
/* Copying data from array 'a' to array 'b */
for(i=0;i<num;i++)
{
    arr2[i] = arr1[i];
}
//Printing of all elements of array
printf("The copied array is :");
for(i=0;i<num;i++)
    printf("\narr2[%d] = %d",i,arr2[i]);
}

```

Output: Enter no of elements: 5
Enter the values: 11 22 33 44 55
The copied array is: 11 22 33 44 55

Example: C Program to Merge Two arrays in C Programming

```

#include<stdio.h>
main()
{
    int arr1[30], arr2[30], res[60];
    int i, j, k, n1, n2;
    printf("\nEnter no of elements in 1st array :");
    scanf("%d", &n1);
    for(i=0;i<n1;i++)
    {
        scanf("%d",&arr1[i]);
    }
    printf("\nEnter no of elements in 2nd array :");
    scanf("%d", &n2);
    for(i=0;i<n2;i++)
    {
        scanf("%d",&arr2[i]);
    }
    i = 0;
    j = 0;
    k = 0;
    while(i<n1&& j<n2)
    {
        if(arr1[i]<=arr2[j])
        {
            res[k]=arr1[i];
            i++;
            k++;
        }
    }
}

```

```

    }
    else
    {
        res[k]=arr2[j];
        k++;
        j++;
    }
}
/* Some elements in array 'arr1' are still remaining
where as the array 'arr2' is exhausted */
while(i < n1)
{
    res[k]=arr1[i];
    i++;
    k++;
}
/* Some elements in array 'arr2' are still remaining
where as the array 'arr1' is exhausted */
while(j<n2)
{
    res[k]=arr2[j];
    k++;
    j++;
}
//Displaying elements of array 'res'
printf("\nMerged array is :");
for(i=0;i<n1+n2;i++)
    printf("%d ",res[i]);
}

```

Output:

```

Enter no of elements in 1st array: 4
11 22 33 44
Enter no of elements in 2nd array: 3
10 40 80
Merged array is : 10 11 22 33 40 44 80

```

Example: C Program to Reversing an Array Elements in C Programming

```

#include<stdio.h>
main()
{
    int arr[30], i, j, num, temp;
    printf("\nEnter no of elements : ");
    scanf("%d", &num);
    //Read elements in an array
    for(i=0;i<num;i++)
    {
        scanf("%d",&arr[i]);
    }
    j=i-1;
    i=0;
    while(i < j)
    {

```

```

        temp=arr[i];
        arr[i]=arr[j];
        arr[j]=temp;
        i++;          // increment i
        j--;          // decrement j
    }
    printf("\nResult after reversal : ");
    for(i=0;i<num;i++)
    {
        printf("%d \t",arr[i]);
    }
;    }

```

Output: Enter no of elements: 5
 11 22 33 44 55
 Result after reversal: 55 44 33 22 11

Example: C Program to Reversing an Array Elements in C Programming

```

#include<stdio.h>
int main()
{
    int arr[30], i, j, num, temp;
    clrscr();
    printf("\nEnter no of elements : ");
    scanf("%d", &num);
    for(i=0;i<=num;i++)
    {
        scanf("%d",&arr[i]);
    }
    printf("\nResult after reversal : ");
    for(i=num;i>=0;i--)
    {
        printf("%d \t",arr[i]);
    }
    return (0);
}

```

Output: Enter no of elements: 5
 11 22 33 44 55
 Result after reversal: 55 44 33 22 11

Example: C Program to find greatest number from one dimensional array

```

#include<stdio.h>
#include<conio.h>
main()
{
    int a[20],n,i,max=0; // declared the array a with size 20
    clrscr();
    printf("\n Enter the number of elements for 1-D array : ");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("\n Enter element [%d] : ",i+1);
        scanf("%d",&a[i]);
    }
}

```

```

    }
    for(i=0;i<n;i++)
    {
        if(max<a[i])
            max=a[i];
    }
    printf("\n Greatest element is : %d",max);
    getch();
}

```

Output: Enter the number of elements for 1-D array : 5

Enter element [1] : 1

Enter element [2] : 8

Enter element [3] : 2

Enter element [4] : 5

Enter element [5] : 9

Greatest element from above array inserted is: 9

Example: C Program to Find Smallest Element in Array in C Programming

```

#include<stdio.h>
main()
{
    int a[30], i, num, smallest;
    printf("\nEnter no of elements :");
    scanf("%d", &num);
    //Read n elements in an array
    for(i=0;i<num;i++)
        scanf("%d",&a[i]);

    //Consider first element as smallest
    smallest=a[0];

    for(i=0;i<num;i++)
    {
        if(a[i]<smallest)
        {
            smallest=a[i];
        }
    }
    // Print out the Result
    printf("\nSmallest Element : %d", smallest);
}

```

Output: Enter no of elements : 5

11 44 22 55 99

Smallest Element : 11

Example: C code to find largest and smallest number in an array

```

#include<stdio.h>
main()
{
    int a[50],size,i,big,small;
    printf("\nEnter the size of the array: ");
    scanf("%d",&size);
}

```



```

printf("\nEnter %d elements in to the array: ", size);
for(i=0;i<size;i++)
    scanf("%d",&a[i]);
big=a[0];
for(i=1;i<size;i++)
{
    if(big<a[i])
        big=a[i];
}
printf("Largest element: %d",big);
small=a[0];
for(i=1;i<size;i++)
{
    if(small>a[i])
        small=a[i];
}
printf("Smallest element: %d",small);
}

```

Output: Enter the size of the array: 4
Enter 4 elements in to the array: 2 7 8 1
Largest element: 8
Smallest element: 1

Example: C Program to Calculate Addition of All Elements in Array

```

#include<stdio.h>
main()
{
    int i, arr[50], sum, num;
    printf("\nEnter no of elements :");
    scanf("%d", &num);
    printf("\nEnter the values :");
    for(i=0;i<num;i++)
        scanf("%d", &arr[i]);
    sum=0;
    for(i=0;i<num;i++)
        sum=sum+arr[i];
    for(i=0;i<num;i++)
        printf("\na[%d]=%d",i,arr[i]);
    printf("\nSum=%d",sum);
}

```

Output: Enter no of elements : 3
Enter the values : 11 22 33
a[0]=11
a[1]=22
a[2]=33
Sum=66

Example: C Program to Delete duplicate elements from an array

```

#include<stdio.h>
main()
{
    int arr[20], i, j, k, size;

```

```

printf("\nEnter array size : ");
scanf("%d", &size);
printf("\nAccept Numbers : ");
for(i = 0; i < size; i++)
    scanf("%d", &arr[i]);
printf("\nArray with Unique list : ");
for(i=0;i<size;i++)
{
    for(j=i+1;j<size;)
    {
        if(arr[j]==arr[i])
        {
            for(k=j;k<size;k++)
            {
                arr[k]=arr[k + 1];
            }
            size--;
        }
        else
        {
            j++;
        }
    }
}
for(i=0;i<size;i++)
{
    printf("%d ",arr[i]);
}

```

Output: Enter array size: 5
 Accept Numbers: 1 3 4 5 3
 Array with Unique list: 1 3 4 5

Example: C Program to Print the Number of Odd & Even Numbers in an Array

```

#include <stdio.h>
void main()
{
    int array[100], i, num;

    printf("Enter the size of an array \n");
    scanf("%d", &num);
    printf("Enter the elements of the array \n");
    for (i = 0; i < num; i++)
    {
        scanf("%d", &array[i]);
    }
    printf("Even numbers in the array are - ");
    for (i=0;i<num;i++)
    {
        if (array[i]%2==0)
        {
            printf("%d \t", array[i]);

```

```

    }
}
printf("\n Odd numbers in the array are -");
for (i=0; i<num;i++)
{
    if (array[i]%2!=0)
    {
        printf("%d \t", array[i]);
    }
}
}

```

Example: C Program to Generate Pascal Triangle 1 D Array

```

#include<stdio.h>
void main()
{
    int array[30], temp[30], i, j, k, l, num;
    printf("Enter the number of lines to be printed: ");
    scanf("%d", &num);
    temp[0] = 1;
    array[0] = 1;
    for (j = 0; j < num; j++)
    {
        printf(" ");
    }
    printf(" 1\n");
    for (i = 1; i < num; i++)
    {
        for (j = 0; j < i; j++)
            printf(" ");
        for (k = 1; k < num; k++)
        {
            array[k] = temp[k - 1] + temp[k];
        }
        array[i] = 1;
        for (l = 0; l <= i; l++)
        {
            printf("%3d", array[l]);
            temp[l] = array[l];
        }
        printf("\n");
    }
}

```

Example: C Program to Print the Number of Odd & Even Numbers in an Array

```

#include <stdio.h>
void main()
{
    int array[100], i, num;
    printf("Enter the size of an array \n");
    scanf("%d", &num);
    printf("Enter the elements of the array \n");
}

```

```

    for (i = 0; i < num; i++)
    {
        scanf("%d", &array[i]);
    }
    printf("Even numbers in the array are - ");
    for (i = 0; i < num; i++)
    {
        if (array[i] % 2 == 0)
        {
            printf("%d \t", array[i]);
        }
    }
    printf("\n Odd numbers in the array are -");
    for (i = 0; i < num; i++)
    {
        if (array[i] % 2 != 0)
        {
            printf("%d \t", array[i]);
        }
    }
}

```

Example: C program to accept N numbers and arrange them in an ascending order

```

#include <stdio.h>
void main()
{
    int i, j, a, n, number[30];
    printf("Enter the value of N \n");
    scanf("%d", &n);
    printf("Enter the numbers \n");
    for (i = 0; i < n; ++i)
        scanf("%d", &number[i]);
    for (i = 0; i < n; ++i)
    {
        for (j = i + 1; j < n; ++j)
        {
            if (number[i] > number[j])
            {
                a = number[i];
                number[i] = number[j];
                number[j] = a;
            }
        }
    }
    printf("The numbers arranged in ascending order are \n");
    for (i = 0; i < n; ++i)
        printf("%d\n", number[i]);
}

```

Example: C program to accept N numbers and arrange them in an descending order

```

#include <stdio.h>
void main()
{
    int i, j, a, n, number[30];
    printf("Enter the value of N \n");
    scanf("%d", &n);
    printf("Enter the numbers \n");
    for (i = 0; i < n; ++i)
        scanf("%d", &number[i]);
    for (i = 0; i < n; ++i)
    {
        for (j = i + 1; j < n; ++j)
        {
            if (number[i] < number[j])
            {
                a = number[i];
                number[i] = number[j];
                number[j] = a;
            }
        }
    }
    printf("The numbers arranged in descending order are \n");
    for (i = 0; i < n; ++i)
        printf("%d\n", number[i]);
}

```

Multidimensional array: Array having more than one subscript variable is called multidimensional array.

Types of multidimensional array: we have two types

1. Two dimensional array
2. Three dimensional array

Two dimensional array: Two Dimensional Array requires Two Subscript Variables. Two Dimensional Array stores the values in the form of matrix. One Subscript Variable denotes the “Row” of a matrix. Another Subscript Variable denotes the “Column” of a matrix. It is also known as matrix.

Declaration of two dimensional arrays: The syntax for two-dimensional arrays

Syntax: data-type array-name [row-size] [column-size];

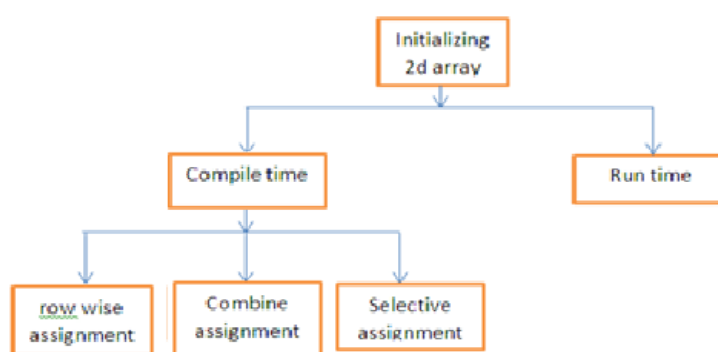
Example: **int a[3] [4]**

As with the single dimensional arrays, each dimension of the array is indexed from zero to its maximum size minus one. The first index selects the row and the second index selects the column within that row

	Column 0	Column	Column	Column
Row	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Array name Row subscript Column subscript

Initialization of two-dimensional arrays: Assigning the required information to a variable before processing is called initialization.



Initializing all Elements row wise:

Syntax: `int a[3][4] = { {1,2,3,4}, {5,6,7,8}, {9,10,11,12} };`

	C O L U M N S			
	a00	a01	a02	a03
	1	2	3	4
R	a10	a11	a12	a13
O	5	6	7	8
W	a20	a21	a22	a23
S	9	10	11	12

Example:

```

#include<stdio.h>
main()
{
    int i, j;
    int a[3][2] = { { 1, 4 }, { 5, 2 }, { 6, 5 } };
    for (i = 0; i < 3; i++)
    {
        for (j = 0; j < 2; j++)
  
```

```

        {
            printf("%d ", a[i][j]);
        }
        printf("\n");
    }
}

```

Output: 1 4
5 2
6 5

Combine and Initializing 2D Array:

Syntax: `int a[3][4] = {1,2,3,4,5,6,7,8,9,10,11,12};`

The initial values are assigned by row wise:

C O L U M N S				
R O W S	a00	a01	a02	a03
	1	2	3	4
	a10	a11	a12	a13
	5	6	7	8
S	a20	a21	a22	a23
	9	10	11	12

Example:

```

#include <stdio.h>
main()
{
    int i, j, a[3][2] = { 1, 4, 5, 2, 6, 5 };
    for (i = 0; i < 3; i++)
    {
        for (j = 0; j < 2; j++)
        {
            printf("%d ", a[i][j]);
        }
        printf("\n");
    }
}

```

Some Elements could be initialized:

Syntax: `int a[3][2] = {{ 1 }, { 5, 2 }, { 6 }};`

Now we have again going with the way 1 but we are removing some of the elements from the array.

In this case we have declared and initialized 2-D array like this

C O L U M N S		
R	a00	a01
O	1	0

W	a10	a11
S	5	2
	a20	a21
	6	0

Example: `#include <stdio.h>`
`main()`
`{`
`int i, j;`
`int a[3][2] = { { 1 }, { 5, 2 }, { 6 } };`
`for (i = 0; i < 3; i++)`
`{`
`for (j = 0; j < 2; j++)`
`{`
`printf("%d ", a[i][j]);`
`}`
`printf("\n");`
`}`
`}`

Output: 1 0
5 2
6 0

syntax: `int a[3] [4] = { 1,2,3,4,5,6,7,8,9}`

C O L U M N S				
	a00	a01	a02	a03
	1	2	3	4
R	a10	a11	a12	a13
O	5	6	7	8
W	a20	a21	a22	a23
S	9	0	0	0

Syntax: `int a[3] [4] = { {1,2,3}, {4,5,6}, {7,8,9} };`

C O L U M N S				
	a00	a01	a02	a03
	1	2	3	0
R	a10	a11	a12	a13
O	4	5	6	0
W	a20	a21	a22	a23
S	7	8	9	0

Example: `#include <stdio.h>`
`main()`


```

{
    int a[4][4], i, j;
    clrscr();
    for (i=0; i<4; i++)
    {
        for(j=0; j<4; j++)
        {
            if(i==j)
                a[i][i] = 1;
            else
                a[i][j]=0;
        }
        printf("\n");
    }
    for(i=0; i<4; i++)
    {
        for(j=0; j<4; j++)
        {
            printf("%d\t", a[i][j]);
        }
        printf("\n");
    }
    getch();
}

```

Output:

1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

Storing and Accessing elements from two dimensional arrays:

Write a program to read elements of two dimensional matrix and display its elements on the monitor

Example:

```

#include<stdio.h>
main()
{
    int a[10][10], m, n, i, j;
    printf("Enter number of rows and columns");
    scanf("%d%d", &m, &n);
    printf("Enter elements:\n");
    for(i=0; i<m; i++)
    {
        for(j=0; j<n; j++)
        {
            scanf("%d", &a[i][j]);
        }
    }
    printf("The elements of the matrix are:\n");
    for(i=0; i<m; i++)
    {
        for(j=0; j<n; j++)
        {
            printf("%d\t", a[i][j]);
        }
    }
}

```

```

    }
    printf("\n");
}

```

Example: Addition of All Elements in Matrix

```

#include<stdio.h>
main()
{
    int i, j, mat[10][10], row, col;
    int sum = 0;
    printf("\nEnter the number of Rows and columns : ");
    scanf("%d%d", &row,&col);
    printf("Enter the elements");
    for (i = 0; i < row; i++)
    {
        for (j = 0; j < col; j++)
        {
            scanf("%d", &mat[i][j]);
        }
    }
    for (i = 0; i < row; i++)
    {
        for (j = 0; j < col; j++)
        {
            sum = sum + mat[i][j];
        }
    }
    printf("\nSum of All Elements in Matrix : %d", sum);
}

```

Example: Write a C program to find sum of two matrix of order 2*2 using multidimensional arrays where, elements of matrix are entered by user.

```

#include <stdio.h>
main()
{
    int a[2][2], b[2][2], c[2][2];
    int i,j;
    printf("Enter the elements of 1st matrix\n");
    for(i=0;i<2;++i)
    {
        for(j=0;j<2;++j)
        {
            printf("Enter a%d%d: ",i+1,j+1);
            scanf("%d",&a[i][j]);
        }
    }
    printf("Enter the elements of 2nd matrix\n");
    for(i=0;i<2;++i)
    {
        for(j=0;j<2;++j)
        {

```

```

        printf("Enter b%d%d: ",i+1,j+1);
        scanf("%d",&b[i][j]);
    }
}
for(i=0;i<2;++i)
{
    for(j=0;j<2;++j)
    {
        c[i][j]=a[i][j]+b[i][j];
    }
}
printf("\nSum Of Matrix:");
for(i=0;i<2;++i)
{
    for(j=0;j<2;++j)
    {
        printf("%d\t",c[i][j]);
        if(j==1)        /* To display matrix sum in order. */
            printf("\n");
    }
}
}

```

Example: C program for addition of two matrices using arrays source code.

```

#include<stdio.h>
main()
{
    int a[3][3],b[3][3],c[3][3],i,j;
    printf("Enter the First matrix->");
    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
        {
            scanf("%d",&a[i][j]);
        }
    }
    printf("\nEnter the Second matrix->");
    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
        {
            scanf("%d",&b[i][j]);
        }
    }
    printf("\nThe First matrix is\n");
    for(i=0;i<3;i++)
    {
        printf("\n");
        for(j=0;j<3;j++)
        {

```

```

        printf("%d\t",a[i][j]);
    }
}
printf("\nThe Second matrix is\n");
for(i=0;i<3;i++)
{
    printf("\n");
    for(j=0;j<3;j++)
    {
        printf("%d\t",b[i][j]);
    }
}
for(i=0;i<3;i++)
{
    for(j=0;j<3;j++)
    {
        c[i][j]=a[i][j]+b[i][j];
    }
}
printf("\nThe Addition of two matrix is\n");
for(i=0;i<3;i++)
{
    printf("\n");
    for(j=0;j<3;j++)
    {
        printf("%d\t",c[i][j]);
    }
}
}

```

Example: subtraction of two matrices using c program

```

#include<stdio.h>
main()
{
    int a[3][3],b[3][3],c[3][3],i,j;
    printf("Enter the First matrix->");
    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
        {
            scanf("%d",&a[i][j]);
        }
    }
    printf("\nEnter the Second matrix->");
    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
        {
            scanf("%d",&b[i][j]);
        }
    }
}

```

```

    }
}
printf("\nThe First matrix is\n");
for(i=0;i<3;i++)
{
    printf("\n");
    for(j=0;j<3;j++)
    {
        printf("%d\t",a[i][j]);
    }
}
printf("\nThe Second matrix is\n");
for(i=0;i<3;i++)
{
    printf("\n");
    for(j=0;j<3;j++)
    {
        printf("%d\t",b[i][j]);
    }
}
for(i=0;i<3;i++)
{
    for(j=0;j<3;j++)
    {
        c[i][j]=a[i][j]-b[i][j];
    }
}
printf("\nThe Subtraction of two matrix is\n");
for(i=0;i<3;i++)
{
    printf("\n");
    for(j=0;j<3;j++)
    {
        printf("%d\t",c[i][j]);
    }
}
}

```

Example: Write a program for matrix multiplication in c

```

#include<stdio.h>
main()
{
    int a[5][5],b[5][5],c[5][5],i,j,k,sum=0,m,n,o,p;
    printf("\nEnter the row and column of first matrix");
    scanf("%d %d",&m,&n);
    printf("\nEnter the row and column of second matrix");
    scanf("%d %d",&o,&p);
    if(n!=o)
    {
        printf("Matrix mutiplication is not possible");
    }
}

```

```

printf("\n rows of first matrix must be same as Column of
second matrix");
}
else
{
printf("\nEnter the First matrix->");
for(i=0;i<m;i++)
{
    for(j=0;j<n;j++)
    {
        scanf("%d",&a[i][j]);
    }
}
printf("\nEnter the Second matrix->");
for(i=0;i<o;i++)
{
    for(j=0;j<p;j++)
    {
        scanf("%d",&b[i][j]);
    }
}
printf("\nThe First matrix is\n");
for(i=0;i<m;i++)
{
    printf("\n");
    for(j=0;j<n;j++)
    {
        printf("%d\t",a[i][j]);
    }
}
printf("\nThe Second matrix is\n");
for(i=0;i<o;i++)
{
    printf("\n");
    for(j=0;j<p;j++)
    {
        printf("%d\t",b[i][j]);
    }
}
for(i=0;i<m;i++)
{ //row of first matrix
    for(j=0;j<p;j++)
    {
        /*column of second matrix*/
        sum=0;
        for(k=0;k<n;k++)
        {
            sum=sum+a[i][k]*b[k][j];
            c[i][j]=sum;
        }
    }
}

```

```

    }
    printf("\nThe multiplication of two matrix is\n");
    for(i=0;i<m;i++)
    {
        printf("\n");
        for(j=0;j<p;j++)
        {
            printf("%d\t",c[i][j]);
        }
    }
}
}

```

Example: Sum of diagonal elements of a matrix in c

```

#include<stdio.h>
main()
{
    int a[10][10],i,j,sum=0,m,n;
    printf("\nEnter the row and column of matrix: ");
    scanf("%d %d",&m,&n);
    printf("\nEnter the elements of matrix: ");
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
        {
            scanf("%d",&a[i][j]);
        }
    }
    printf("\nThe matrix is\n");
    for(i=0;i<m;i++)
    {
        printf("\n");
        for(j=0;j<m;j++)
        {
            printf("%d\t",a[i][j]);
        }
    }
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
        {
            if(i==j)
                sum=sum+a[i][j];
        }
    }
    printf("\n\nSum of the diagonal elements of a matrix is: %d",sum);
}

```

Output: Enter the row and column of matrix: 3 3

Enter the elements of matrix:

2
3
5
6
7
9
2
6
7

The matrix is

2	3	5
6	7	9
2	6	7

Sum of the diagonal elements of a matrix is: 16

Example: C program to accept a matrix of order MxN and find its transpose

```
#include <stdio.h>
void main()
{
    int array[10][10];
    int i, j, m, n;
    printf("Enter the order of the matrix \n");
    scanf("%d %d", &m, &n);
    printf("Enter the coefficients of the matrix\n");
    for (i = 0; i < m; ++i)
    {
        for (j = 0; j < n; ++j)
        {
            scanf("%d", &array[i][j]);
        }
    }
    printf("The given matrix is \n");
    for (i = 0; i < m; ++i)
    {
        for (j = 0; j < n; ++j)
        {
            printf(" %d", array[i][j]);
        }
        printf("\n");
    }
    printf("Transpose of matrix is \n");
    for (j = 0; j < n; ++j)
    {
        for (i = 0; i < m; ++i)
        {
            printf(" %d", array[i][j]);
        }
        printf("\n");
    }
}
```

Example: C program to accept a matrix of order MxN and find its transpose


```

#include <stdio.h>
#include <conio.h>
void main()
{
    int i,j,M,N;
    int A[10][10], B[10][10];
    int transpose(int A[][10], int r, int c); /*Function prototype*/
    clrscr();
    printf("Enter the order of matrix A\n");
    scanf("%d %d", &M, &N);
    printf("Enter the elements of matrix\n");
    for(i=0;i<M;i++)
    {
        for(j=0;j<N;j++)
        {
            scanf("%d",&A[i][j]);
        }
    }
    printf("Matrix A is\n");
    for(i=0;i<M;i++)
    {
        for(j=0;j<N;j++)
        {
            printf("%3d",A[i][j]);
        }
        printf("\n");
    }
    /* Finding Transpose of matrix*/
    for(i=0;i<M;i++)
    {
        for(j=0;j<N;j++)
        {
            B[i][j] = A[j][i];
        }
    }
    printf("Its Transpose is\n");
    for(i=0;i<M;i++)
    {
        for(j=0;j<N;j++)
        {
            printf("%3d",B[i][j]);
        }
        printf("\n");
    }
}

```

Example: C code for Determinant of 2X2 matrix

```

#include<stdio.h>
main()
{

```

```

int a[2][2],i,j;
long determinant;
printf("Enter the 4 elements of matrix: ");
for(i=0;i<2;i++)
{
    for(j=0;j<2;j++)
    {
        scanf("%d",&a[i][j]);
    }
}
printf("\nThe matrix is\n");
for(i=0;i<2;i++)
{
    printf("\n");
    for(j=0;j<2;j++)
    {
        printf("%d\t",a[i][j]);
    }
}
determinant = a[0][0]*a[1][1] - a[1][0]*a[0][1];
printf("\nDeterminant of 2X2 matrix: %ld",determinant);
}

```

Output: Enter the 4 elements of matrix: 4
 8
 3
 9

The matrix is 4 8
 3 9

Determinant of 2X2 matrix: 12

Example: C code for Determinant of 3X3 matrix

```

#include<stdio.h>
main()
{
    int a[3][3],i,j;
    long determinant;
    printf("Enter the 9 elements of matrix: ");
    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
        {
            scanf("%d",&a[i][j]);
        }
    }
    printf("\nThe matrix is\n");
    for(i=0;i<3;i++)
    {
        printf("\n");
        for(j=0;j<3;j++)
        {

```

```

        printf("%d\t",a[i][j]);
    }
}
determinant = a[0][0]*((a[1][1]*a[2][2]) - (a[2][1]*a[1][2])) -
              a[0][1]*(a[1][0]*a[2][2] - a[2][0]*a[1][2]) +
              a[0][2]*(a[1][0]*a[2][1] - a[2][0]*a[1][1]);
printf("\nDeterminant of 3X3 matrix: %ld",determinant);
}

```

Output: Enter the 9 elements of matrix:

```

1 2 3
4 5 6
7 8 9

```

The matrix is

1	2	3
4	5	6
7	8	9

Determinant of 3X3 matrix: 0

Example: C Program to find addition of Lower Triangular Elements in C Programming

```

#include<stdio.h>
#include<conio.h>
main()
{
    int i, j, a[10][10], sum, rows, columns;
    printf("\nEnter the number of Rows and Columns : ");
    scanf("%d%d ", &rows, &columns);
    /*Accept the Elements in Matrix*/
    for (i = 0; i < rows; i++)
    {
        for (j = 0; j < columns; j++)
        {
            scanf("%d", &a[i][j]);
        }
    }
    /*Addition of all Diagonal Elements*/
    sum = 0;
    for (i = 0; i < rows; i++)
    {
        for (j = 0; j < columns; j++)
        {
            if (i > j)
            {
                sum = sum + a[i][j];
            }
        }
    }
    printf("\nSum of Lower Triangle Elements : %d", sum);
}

```

Example: C program to calculate sum of Upper Triangular Elements in C

```

#include<stdio.h>
#include<conio.h>
main()
{

```

```

int i, j, a[10][10], sum, rows, columns;
printf("\nEnter the number of Rows and columns : ");
scanf("%d%d", &rows,&columns);
printf("Enter the elements");
for (i = 0; i < rows; i++)
{
    for (j = 0; j < columns; j++)
    {
        scanf("%d", &a[i][j]);
    }
}
sum = 0;
for (i = 0; i < rows; i++)
{
    for (j = 0; j < columns; j++)
    {
        if (i < j)
        {
            sum = sum + a[i][j];
        }
    }
}
printf("\nSum of Upper Triangle Elements : %d", sum);
}

```

A **three dimensional array** can be used to represent a collection of tables. It can be thought of as a book where each page is equivalent to a table and the page number represents the 3rd dimension.

Advantages:

- ❖ It is used to represent multiple data items of same type by using only single name.
- ❖ It can be used to implement other data structures like linked lists, stacks, queues, trees, graphs etc.
- ❖ It is capable of storing many elements at a time
- ❖ It allows random accessing of elements i.e. any element of the array can be randomly accessed using indexes.
- ❖ Arrays are simple to understand and use.
- ❖ 2D arrays are used to represent matrices.

Limitations of an Array:

Static Data:

- ❖ Array is Static data Structure
- ❖ Memory Allocated during Compile time.
- ❖ Once Memory is allocated at Compile Time it Cannot be Changed during Run-time

Can hold data belonging to same Data types:

- ❖ Elements belonging to different data types cannot be stored in array because array data structure can hold data belonging to same data type.

- ❖ Example : Character and Integer values can be stored inside separate array but cannot be stored in single array

Inserting data in Array is Difficult:

- ❖ Inserting element is very difficult because before inserting element in an array we have to create empty space by shifting other elements one position ahead.
- ❖ This operation is faster if the array size is smaller, but same operation will be more and more time consuming and non-efficient in case of array with large size.

Deletion Operation is difficult:

- ❖ Deletion is not easy because the elements are stored in contiguous memory location.
- ❖ Like insertion operation , we have to delete element from the array and after deletion empty space will be created and thus we need to fill the space by moving elements up in the array.

Bound Checking:

- ❖ If we specify the size of array as 'N' then we can access elements upto 'N-1' but in C if we try to access elements after 'N-1' i.e Nth element or N+1th element then we does not get any error message.
- ❖ Process of Checking the extreme limit of array is called Bound Checking and C does not perform Bound Checking.
- ❖ If the array range exceeds then we will get garbage value as result.

Shortage of Memory:

- ❖ Array is Static data structure. Memory can be allocated at compile time only Thus if after executing program we need more space for storing additional information then we cannot allocate additional space at run time.
- ❖ Shortage of Memory , if we don't know the size of memory in advance

Wastage of Memory: Wastage of Memory , if array of large size is defined

Strings

String: a string is a sequence of characters i.e treated as a single data item

(Or)

Group of characters between the double quotes is called as string constant

(Or)

A string is one dimensional array of characters terminated by a null character ('\\0') and it can be read as a single entity.

String is useful whenever we accept name of the person, Address of the person, some descriptive information. We cannot declare string using String Data Type, instead of we use array of type character to create String.

Declaration of String Variables: As a variable has to be declared before it is used, a string variable

also has to be declared before it is used.

syntax: **char array_name [size];**

Examples: char city[30];
 char name[20];
 char message[50];

These are some sample declarations of the String. In the first example we have defined string to store name of city. Maximum Size to store City is 30 which must be specified inside the Square brackets.

During declaration, we must provide enough storage for the data and delimiter. The storage space must be 1 byte larger than the maximum data size so to accommodate '\0' at the end of the string.

Size=maximum no of characters+1.

Length=no of characters in a string.

Initialization of strings:

Initialization is the process of assigning values to a variable before doing manipulation.

The initialization can be done in ways:

1. compile time initialization
2. runtime initialization

Compile time initialization:

1. Initializing locations character by character
2. initialization with a string
3. Strings and assignment operator

Initializing locations character by character:

Initializing locations character by character with specifying enough size:

Syntax: **char A[9] = { 'C', 'O', 'M', 'P', 'U', 'T', 'E', 'R' };**

Or

char A[9] = { 'C', 'O', 'M', 'P', 'U', 'T', 'E', 'R', '\0' };

The computer allocates 9 memory locations ranging from 0 to 8 and in these 8 locations are initialized with the characters and the 9th location filled with terminating null(string must be terminated with null character '\0').

C	O	M	P	U	T	E	R	\0
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]

The responsibility of the programmer is to allocate sufficient memory so as to accommodate NULL character at the end.

Initializing locations character by character with specifying exact size:

Syntax: `char A[8] = { 'C', 'O', 'M', 'P', 'U', 'T', 'E', 'R' };`

The computer allocates 8 memory locations ranging from 0 to 7 and in these 8 locations are initialized with the characters, and this string is not terminated with nullcharacter('\0').

C	O	M	P	U	T	E	R
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]

The responsibility of the programmer is to allocate sufficient memory so as to accommodate NULL character at the end.

Initializing locations character by character with specifying less size:

Syntax: `char A[4] = { 'C', 'O', 'M', 'P', 'U', 'T', 'E', 'R' };`

In the above initialization only the first four characters of the string are assigned to the array a, and null is not included. The output may contain unexpected characters after the four characters (or) we will get error message like too many initializers in function main.

Partial initialization: If the number of characters to be initialized is less than the size of the array, then characters are stored sequentially from left to right. The remaining locations will be initialized to NULL Characters automatically.

syntax: `char A[10] = { 'R', 'A', 'M', 'A' };`

The above statement allocates 10 bytes for the variable a ranging from 0 to 9 and initializes first four locations with the ASCII characters of 'R', 'A', 'M', 'A' and The remaining locations will be initialized to NULL Characters automatically

R	A	M	A	\0	\0	\0	\0	\0	\0
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[5]	A[6]	A[7]	A[8]

Initializing locations character by character without specifying size:

Syntax: `char A[] = { 'C', 'O', 'M', 'P', 'U', 'T', 'E', 'R' };`

For this declaration, the compiler will set the array size is 9 bytes, 8 for letters and one for the terminating null.

C	O	M	P	U	T	E	R	\0
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]

Initializing locations by string:

Initializing locations by string with specifying enough size:

Syntax: `char A[9] = "COMPUTER";`

The computer allocates 9 memory locations ranging from 0 to 8 and in these 8 locations are initialized with the characters and the 9th location filled with terminating null (string must be terminated with null character '\0').

C	O	M	P	U	T	E	R	\0
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]

Initializing locations by string with specifying exact size:

Syntax: `char A[8] = "COMPUTER";`

Here, the array holds only the 8 specified characters. The string is not terminated by \0 by the compiler.

C	O	M	P	U	T	E	R
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]

Initializing locations by string with specifying less size:

syntax: `char a[4]="program";`

In the above initialization, we will get error message like too many initializers in function main.

Partial initialization:

Syntax: `char A[10] = "RAMA";`

The above statement allocates 10 bytes for the variable a ranging from 0 to 9 and initializes first four locations with the ASCII characters of 'R', 'A', 'M', 'A' and The remaining locations will be initialized to NULL Characters automatically

R	A	M	A	\0	\0	\0	\0	\0	\0
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[5]	A[6]	A[7]	A[8]

Initializing locations by string without specifying size:

Syntax: `char A[] = "COMPUTER";`

Here, the string length is 8 bytes. But, String size is 9 bytes. So, the compiler reserves 8+1(8 for string and 1 for terminating null).

C	O	M	P	U	T	E	R	\0
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]

The string "COMPUTER" contains 8 characters, because it is a string, it always ends with null character. So, the array size is 9 bytes.i.e. string length + 1 byte for NULL Character.

Strings and assignment operator:

1. The string is an array of characters; the name of the string is pointer constant. The constants cannot be used on the left hand side of assignment operator.

Example: "Krishna" = "Rama"; → Invalid

2. We cannot separate the initialization from declaration.

Example: `char s1[10];`
`S1="good";` } invalid

3. An array name cannot be used as the left operand of an assignment operator.

Example: `char s1[10]="abc";`
`Char s2[4];`
`S2=s1; /* error*/` } Invalid

Reading strings from terminal (Or) Runtime initialization: we have many functions to initialize the string at run time. These are:

scanf(): The familiar scanf input function scanf can be used with %s format specification to read a string from the keyboard.

Syntax: `scanf("%s",variable_name);`

Example: `char s1[20];`
`scanf("%s",s1);`

Example: Write a C program to illustrate how to read string from terminal.

```
#include <stdio.h>
main()
{
    char name[20];
    printf("Enter name: ");
    scanf("%s",name);
    printf("Your name is %s.",name);
}
```

Output: Enter name: Dennis Ritchie
 Your name is Dennis.

- It reads a string until white space character occurs and the null is automatically appended at the end of the string.
- Here we must observe that address list of scanf contains name of the array only (i.e string name) without '&' preceding each.
- The problem with scanf is that it terminates its input on the first white space it finds (white space includes blanks, tabs and newlines).
- Generally scanf is used to read a word from the keyboard.

gets(): another and more convenient method of reading a string of text containing white spaces is to use the library function gets() which is available in stdio.h header file.

Syntax: `gets(string_name);`

Example: `char s1[10];`

```
gets(s1);
```

- This function reads characters into s1 from the keyboard until a new line character is encountered and the the compiler automatically adds the null at the end of the string.
- This function is used to read a line of text from the keyboard.

Example:

```
#include <stdio.h>
main()
{
    char name[30];
    printf("Enter name: ");
    gets(name); /*Function to read string from user.*/
    printf("Name: ");
    puts(name); /*Function to display string.*/
}
```

getchar(): getchar() function is used for read only one character.

Syntax: **variable=getchar();**

Example: char a;
 a=getchar();

Example:

```
#include <stdio.h>
main()
{
    char name[30],ch;
    int i=0;
    printf("Enter name: ");
    while(ch!='\n') /*terminates if user hit enter */
    {
        ch=getchar();
        name[i]=ch;
        i++;
    }
    name[i]='\0'; /* inserting null character at end */
    printf("Name: %s",name);
}
```

Writing strings to screen:

printf(): we have used extensively the printf function with %s format specification to print strings to the screen.

Example: char s1[]="hai";
 printf("%s",s1);

output: hai

puts(): another and more convenient way of printing string values on to the screen is puts, which is available in the header file stdio.h.

syntax: **puts(stringvariable_name);**

Example: char s1[20]="wel come";
 puts(s1);

output: wel come

Example:

```
#include <stdio.h>
main()
{
    char name[30];
    printf("Enter name: ");
    gets(name); /*Function to read string from user.*/
    printf("Name: ");
    puts(name); /*Function to display string.*/
}
```

putchar(): This function is used for print only one character

syntax: **putchar(stringvariable_name);**

Example:

```
#include <stdio.h>
#include <conio.h>
main()
{
    char inputString[100],c;
    int i=0;
    printf("Enter a string\n");
    /* Read string from user using getchar inside while loop */
    while((c=getchar())!='\n')
    {
        inputString[i]=c;
        i++;
    }
    inputString[i]='\0';
    /* Print string stored in inputString using putchar */
    i=0;
    while(inputString[i]!='\0')
    {
        putchar(inputString[i]);
        index++;
    }
    getch();
}
```

Output:

```
Enter a string
C Programming
C Programming
```

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    char string[]="This is an example string";
    int i=0;
    clrscr();
    while(string[i]!='\0')
    {
        putchar(string[i]);
        i++;
    }
}
```

```

    }
    getch();
}

```

String Handling functions: String is not a standard data type, but it can be treated as derived data type. In a C language, we cannot manipulate the strings directly using 'C' operators. C provides a rich set of string handling functions. The various string handling functions that are supported in C language are:

strlen(): This function finds the length of given string i.e it counts all the character upto '\0', except '\0'. So, an empty string has length zero.

Syntax: **strlen(string_name);**

Example: `char s1[]="good"`
 `n=strlen(s1);`

Output: 4

Example: `n=strlen("good");`

Output: 4

Example: `#include <stdio.h>`
 `void main()`
 `{`
 `char str[]:"COMPUTER";`
 `printf ("length=%d\n",strlen(str));`
 `}`

strcpy() – string copy:- The function, strcpy copies the contents of source string(src) to destination string(dest) including \0. So, the size of destination string (dest) should be greater or equal to the size of source string (src). After copying, the original contents of destination string (dest) are lost. The syntax of this function is

syntax: **strcpy(dest,src);**

Example: `strcpy(s1,s2);`
 `strcpy(s1,"good");`

Example: `#include<stdio.h>`
 `#include<string.h>`
 `void main()`
 `{`
 `char src[] = "DATA";`
 `char dest[6];`
 `strcpy(dest, src)`
 `printf ("Destination string = %s", dest);`
 `}`

Output: Destination string=DATA

strncpy(); this function copies first n characters from string2 to string1. this is a three parameter function.

Syntax: `strncpy(string1,string2,n);`

Example: `char s1[10]="hello";
char s2[10]="howareyou";
strncat(s1,s2,3);`

Output: `s1=how`

strcat(): the strcat function join two strings together that is appends one string at the end of another string. the syntax is

syntax: `strcat(string1,string2)`

string1 and string2 are character arrays, when strcat function is executed string2 is appended to string1.

Example: `#include<stdio.h>
#include<string.h>
void main()
{
 char s1[12] = "RAMA";
 char s2[10] = "KRISHNA";
 strcat (s1, s2);
 printf ("concatenated string is=%s\n",s1);
}`

OUTPUT: Concatenated string is: RAMAKRISHNA

1. We must ensure that the size of the string1 is large enough to accommodate the final string.
2. Strcat() may also append a string constant to a string variable.

Syntax: `strcat(string1,"good");`

3. C language also permits nesting of strcat functions.

Example: `strcat(strcat(string1,string2),string3);`

In the above example concatenates all the 3 strings together and the resultant string is stored in string1.

strncat(): This function will concatenates first n characters of string2 to the end of string1, and it is a three parameters function. The syntax is

Syntax: `strncat(string1,string2,n);`

Example: `char s1[10]="hello";
char s2[10]="howareyou";
strncat(s1,s2,3);`

output: `s1=hellohow`

strcmp(): This function is used to compare two strings. it returns a value 0 if they are equal. If they are not equal. it returns the numeric difference between the first non-matching character in the string. The syntax for this function is

Syntax: `strcmp(string1,string2);`

The following values are returned after comparison:

*if the two strings are equal, the function returns 0.

*if s₁ is greater than s₂, a positive value is returned

*if s₁ is less than s₂, the function returns a negative value

Example: char s₁[]= "Hai";
 char s₂[]= "Hai";
 strcmp(s₁,s₂);

Output: it returns zero(0).

Example: char s₁[]= "Hai";
 char s₂[]= "HAi";
 strcmp(s₁,s₂);

Output: it returns positive value(32)(i.e ASCII value of a-ASCII value of A→97-65=32).

Example: char s₁[]= "their";
 char s₂[]= "there";
 strcmp(s₁,s₂);

Output: it returns negative value(-9)(i.e ASCII value of i-ASCII value of r→106-115=-9).

Example: #include <stdio.h>
 #include <string.h>
 void main()
 {
 char s₁[]= "DATA";
 char s₂[]= "DATA";
 int i;
 i = strcmp (s₁,s₂);
 if (i==0)
 printf("s₁= s₂");
 else if (i>0)
 printf ("s₁>s₂");
 else
 printf ("s₁<s₂");
 }

Example: #include <stdio.h>
 #include <string.h>
 main()
 {
 char str1[] = "fresh" ;
 char str2[] = "refresh" ;
 int i, j, k ;
 i = strcmp (str1, "fresh") ;
 j = strcmp (str1, str2) ;
 k = strcmp (str1, "f") ;
 printf ("\n%d %d %d", i, j, k) ;
 }

Output: 0 -12 114

strncmp(): This function compares first n characters of string1 and string2. it returns a value 0 if they are equal. If they are not equal.it returns the numeric difference between the first non-matching

character in the string

Syntax: **strncmp(s1,s2,n);**

The following values are returned after comparison:

- *if the two strings are equal, the function returns 0.
- *if s₁ is greater than s₂, a positive value is returned
- *if s₁ is less than s₂, the function returns a negative value

Example: `#include <stdio.h>`
 `main()`
 `{`
 `char string1[20];`
 `char string2[20];`
 `strcpy(string1, "Hello");`
 `strcpy(string2, "Hellooo");`
 `printf("Return Value is : %d\n", strncmp(string1, string2, 4));`
 `strcpy(string1, "Helloooo");`
 `strcpy(string2, "Hellooo");`
 `printf("Return Value is : %d\n", strncmp(string1, string2, 10));`
 `strcpy(string1, "Hellooo");`
 `strcpy(string2, "Hellooo");`
 `printf("Return Value is : %d\n", strncmp(string1, string2, 20));`
 `}`

strncmpi(): This function compare two strings with case insensitively(i.e, “A” and “a” are treated as same characters). it returns a value 0 f they are equal. If they are not equal.it returns the numeric difference between the first non-matching character in the string. The syntax is.

Syntax: **int strncmpi(string1,string2);**

It returns

- ❖ zero if they are same.
- ❖ Negative value If length of string1 < string2.
- ❖ Positive value If length of string1 > string2.

Example: `#include <stdio.h>`
 `#include <string.h>`
 `main()`
 `{`
 `char str1[] = "fresh" ;`
 `char str2[] = "refresh" ;`
 `int i, j, k ;`
 `i=strncmpi (str1,"FRESH") ;`
 `j=strncmpi (str1,str2) ;`
 `k=strncmpi (str1,"f") ;`
 `printf ("\n%d%d%d",i,j,k) ;`
 `}`

Output: 0 -12 114

strncmpi(): This function compares first n characters of two strings with case insensitively(i.e, “A”

and “a” are treated as same characters). it returns a value 0 if they are equal. If they are not equal, it returns the numeric difference between the first non-matching character in the string. the syntax is

syntax: **strncmpi(string1,string2,n);**

It returns

- ❖ zero if they are same.
- ❖ Negative value If length of string1 < string2.
- ❖ Positive value If length of string1 > string2.

Example: `#include <stdio.h>`
 `#include <string.h>`
 `main()`
 `{`
 `char str1[] = "fresh" ,str2[] = "freshvegetables" ;`
 `int i, j, k ;`
 `i=strncmpi(str1,str2,5) ;`
 `j=strncmpi(str1,str2,7) ;`
 `k=strncmpi (str1,g,5) ;`
 `printf ("\n%d%d%d",i,j,k) ;`
 `}`

Output: 0 -118 -1

strstr(): This function searches the substring in string for the first occurrence from the beginning. This function is called string in string. This function returns

1. On success, a pointer to the character is returned
2. On failure, NULL is returned.

The syntax of this function is

Syntax: **strstr(s₁, s₂);**

Here *s₁ is the string

 *s₂ is the substring be searched in s₁.

Example: `#include <stdio.h>`
 `#include <string.h>`
 `main()`
 `{`
 `char s1[20] = "TutorialsPoint";`
 `char s2[10] = "Point";`
 `printf("The substring is: %s\n", strstr(s1,s2));`
 `}`

Output: The substring is: Point

Example: `#include <stdio.h>`
 `#include <string.h>`
 `main()`
 `{`
 `char s1[20] = "TutorialsPoint";`
 `char s2[10] = "hi";`


```
printf("The substring is: %s\n", strstr(s1,s2));
```

```
}
```

Output: The substring is:(null)

Example: WAP to displays the position or index in the string S where the string T begins, or -1 if S

doesn't contain T.

```
#include<stdio.h>
#include<string.h>
#include<conio.h>
void main()
{
    char s[30], t[20];
    char *found;
    clrscr();
    puts("Enter the first string: ");
    gets(s);
    puts("Enter the string to be searched: ");
    gets(t);
    found = strstr(s, t);
    if(found)
    {
        printf("Second String is found in the First String at %d position.\n",
            found - s);
    }
    else
    {
        printf("-1");
    }
    getch();
}
```

strrev(): This function reverses all characters in the string str except the terminating null character '\0'. The syntax of this function is

syntax **strrev(str);**

Here str is the string to be reversed. The function returns pointer to the reversed string.

Example: #include <string.h>
void main()
{
 char str[] = " DATA" ;
 strrev(str);
 printf ("string : %s \n", str);
}

Output: ATAD

strlwr(): This function converts a string to lowercase.

Syntax: **strlwr(string);**

Example: #include<stdio.h>
#include<string.h>
main()

```

    {
        char str[ ] = "MODIFY This String To Lower";
        printf("%s\n",strlwr (str));
    }

```

Output: modify this string to lower

strupr(): This function converts a string to uppercase.

Syntax: **strupr(string);**

Example:

```

#include<stdio.h>
#include<string.h>
main()
{
    char str[ ] = "MODIFY This String To Lower";
    printf("%s\n",strupr(str));
}

```

Output: MODIFY THIS STRING TO LOWER

strdup(): This function duplicates the given string. The syntax is

syntax: **strdup(string);**

Example:

```

#include <stdio.h>
#include <string.h>
main()
{
    char p1[] = "Raja";
    printf("Duplicated string is : %s", strdup(p1));
}

```

Output: Duplicated string is : Raja

strchr (): This function finds first occurrence of a given character in a string. The syntax is.

syntax: **strchr (s1,ch);**

where s₁ is the string
ch is character to be searched.

It returns

1. On success, a pointer to the character is returned
2. On failure, NULL is returned.

Example:

```

#include <stdio.h>
#include <string.h>
main ()
{
    const char str[] = "http://www.tutorialspoint.com";
    const char ch = '.';
    printf("String after %c is %s\n", ch, strchr(str, ch));
}

```

Output: String after. is .tutorialspoint.com

Example:

```

#include <stdio.h>
#include <string.h>
main ()
{
    const char str[] = "http://www.tutorialspoint.com";

```

```

        const char ch = '.';
        printf("String after %c is %s\n", ch, strchr(str, ch));
    }

```

Output: String after . is (null)

strrchr(): This function returns pointer to the last occurrence of the character in a given string. The

Syntax for `strrchr()` function is.

Syntax: `strrchr(str, int character);`

Example:

```

#include <stdio.h>
#include <string.h>
main ()
{
    char string[55]="This is a string for testing";
    printf ("Character i is found at position %d\n",p-string+1);
    printf ("Last occurrence of character i in %s is %s",string,
                                                    strrchr (string,'i'));
}

```

Output: Character i is found at position 26

Last occurrence of character i in This is a string for testing is ing

strrstr(): This function returns pointer to the last occurrence of the string in a given string. The

Syntax for `strrstr()` function is.

Syntax: `strrstr(str1,str2);`

Example:

```

#include <stdio.h>
#include <string.h>
main ()
{
    char string[55]="This is a test string for testing";
    printf ("Last occurrence of string \"test\" in \"%s\" is \"\
    \"%s\"",string, strrstr (string,"test"));
}

```

Output: Last occurrence of string “test” in “This is a test string for testing” is “testing”

strset(): This function sets all the characters in a string to given character. The Syntax for `strset()`

function is.

Syntax: `strset(string,ch);`

Example:

```

#include<stdio.h>
#include<string.h>
main()
{
    char str[20] = "Test String";
    printf("Original string is : %s", str);
    printf("Test string after strset() : %s",strset(str,'#'));
    printf("After string set: %s",str);
}

```

Output: Original string is : Test String

Test string after strset() : #####

strnset(): This function sets portion of characters in a string to given character. The Syntax for strnset() function is.

Syntax: `strnset(string,ch,n);`

Example:

```
#include<stdio.h>
#include<string.h>
main()
{
    char str[20] = "Test String";
    printf("Original string is : %s", str);
    printf("Test string after string set \" : %s",strnset(str,'#',4));
}
```

Output: Original string is : Test String
Test string after string set : ##### String

Example: Find the Position of the Given Character from a String in C

```
#include<stdio.h>
#include<conio.h>
void main()
{
    char x[30];
    char c;
    int i;
    clrscr();
    printf("enter the string:");
    scanf("%s",x);
    printf("enter the char to which its position is to be found:");
    scanf(" %c",&c);
    for(i=0;x[i]!='\0';i++)
    {
        if(x[i]==c)
        {
            printf("the position is:%d",i+1);
        }
    }
    getch();
}
```

Example: Source Code to Find the Frequency of Characters

```
#include <stdio.h>
main()
{
    char c[1000],ch;
    int i,count=0;
    printf("Enter a string: ");
    gets(c);
    printf("Enter a character to find frequency: ");
    scanf("%c",&ch);
    for(i=0;c[i]!='\0';++i)
    {
```

```

        if(ch==c[i])
            ++count;
    }
    printf("Frequency of %c = %d", ch, count);
}

```

Output: Enter a string: This website is awesome.
Enter a character to find frequency: e
Frequency of e = 4

Example: Source Code to Calculated Length without Using strlen() Function

```

#include <stdio.h>
main()
{
    char s[1000],i;
    printf("Enter a string: ");
    scanf("%s",s);
    for(i=0; s[i]!='\0'; ++i);
    printf("Length of string: %d",i);
}

```

Output: Enter a string: Programiz
Length of string: 9

Example: Source Code to Concatenate Two Strings Manually

```

#include <stdio.h>
main()
{
    char s1[100], s2[100], i, j;
    printf("Enter first string: ");
    scanf("%s",s1);
    printf("Enter second string: ");
    scanf("%s",s2);
    for(i=0;s1[i]!='\0';++i);
    { /* i contains length of string s1. */
        for(j=0;s2[j]!='\0';++j,++i)
        {
            s1[i]=s2[j];
        }
    }
    s1[i]='\0';
    printf("After concatenation: %s",s1);
}

```

Output: Enter first string: raja
Enter second string: sekhar
After concatenation: rajasekhar

Example: Source Code to Copy String Manually

```

#include <stdio.h>
main()
{
    char s1[100], s2[100], i;
    printf("Enter string s1: ");
    scanf("%s",s1);
    for(i=0; s1[i]!='\0'; ++i)

```

```

        {
            s2[i]=s1[i];
        }
        s2[i]='\0';
        printf("String s2: %s",s2);
    }

```

Output: Enter String s1: programiz
String s2: programiz

Example: Source Code to Remove Characters in String Except Alphabets

```

#include<stdio.h>
int main()
{
    char line[150];
    int i,j;
    printf("Enter a string: ");
    gets(line);
    for(i=0; line[i]!='\0'; ++i)
    {
        while (!((line[i]>='a'&&line[i]<='z')||(line[i]>='A'&&line[i]<='Z')||
            line[i]=='\0'))
        {
            for(j=i;line[j]!='\0';++j)
            {
                line[j]=line[j+1];
            }
            line[j]='\0';
        }
    }
    printf("Output String: ");
    puts(line);
}

```

Output: Enter a string: p2'r"o@gram84iz./
Output String: programiz

Example: Reverse String Without Using Library Function [Strrev].

```

#include<stdio.h>
#include<string.h>
main()
{
    char str[100], temp;
    int i, j = 0;
    printf("\nEnter the string :");
    gets(str);
    i=0;
    j=strlen(str)-1;
    while (i < j)
    {
        temp = str[i];
        str[i] = str[j];
        str[j] = temp;
        i++;
    }
}

```

```

        j--;
    }
    printf("\nReverse string is :%s", str);
}

```

Output: Enter the string : Pritesh
Reverse string is : hsetirP

Example: C Program to Compare Two Strings Without Using Library Function

```

#include<stdio.h>
main()
{
    char str1[30], str2[30];
    int i;
    printf("\nEnter two strings :");
    gets(str1);
    gets(str2);
    i = 0;
    while (str1[i]==str2[i]&&str1[i]!='\0')
        i++;
    if (str1[i] > str2[i])
        printf("str1 > str2");
    else if (str1[i] < str2[i])
        printf("str1 < str2");
    else
        printf("str1 = str2");
}

```

Example: Write a C Program to Reverse Letter in Each Word of the Entered String

```

#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
    char msg[] = "Welcome to Programming World";
    char str[10];
    int i = 0, j = 0;
    clrscr();
    while (msg[i] != '\0')
    {
        if (msg[i] != ' ')
        {
            str[j] = msg[i];
            j++;
        }
        else
        {
            str[j] = '\0';
            printf("%s", strrev(str));
            printf(" ");
            j = 0;
        }
    }
}

```

```

        i++;
    }
    str[j] = '\0';
    printf("%s", strev(str));
    getch();
}

```

Example: Program to sort set of strings in alphabetical order.

```

#include<stdio.h>
#include<string.h>
void main()
{
    char s[5][20], t[20];
    int i, j;
    clrscr();
    printf("\nEnter any five strings : ");
    for (i = 0; i < 5; i++)
        scanf("%s", s[i]);

    for (i = 1; i < 5; i++)
    {
        for (j = 1; j < 5; j++)
        {
            if (strcmp(s[j - 1], s[j]) > 0)
            {
                strcpy(t, s[j - 1]);
                strcpy(s[j - 1], s[j]);
                strcpy(s[j], t);
            }
        }
    }

    printf("\nStrings in order are : ");
    for (i = 0; i < 5; i++)
        printf("\n%s", s[i]);
    getch();
}

```

Example: C Program to check the given string is palindrome or not

```

#include <stdio.h>
#include <conio.h>
void main()
{
    char a[10];
    int i, len, flag=0;
    clrscr();
    printf("\nENTER A STRING: ");
    gets(a);
    len=strlen(a);
    for (i=0; i<len; i++)
    {
        if(a[i]==a[len-i-1])

```



```

        flag=flag+1;
    }
    if(flag==len)
        printf("\nTHE STRING IS PALINDROM");
    else
        printf("\nTHE STRING IS NOT PALINDROM");
    getch();
}

```

Example: C program to read a string and check if it's a palindrome, without using library functions. Display the result.

```

#include <stdio.h>
#include <string.h>
void main()
{
    char string[25], reverse_string[25] = {'\0'};
    int i, length = 0, flag = 0;
    printf("Enter a string \n");
    gets(string);
    /* keep going through each character of the string till its end */
    for (i = 0; string[i] != '\0'; i++)
    {
        length++;
    }
    for (i = length - 1; i >= 0; i--)
    {
        reverse_string[length - i - 1] = string[i];
    }
    /*Compare the input string and its reverse. If both are equal then
    the input string is palindrome.*/
    for (i = 0; i < length; i++)
    {
        if (reverse_string[i] == string[i])
            flag = 1;
        else
            flag = 0;
    }
    if (flag == 1)
        printf("%s is a palindrome \n", string);
    else
        printf("%s is not a palindrome \n", string);
    getch();
}

```

Example: C program that counts words, characters, lines.

```

#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
    char line[81], ctr;
    int i,c,end = 0,characters = 0,words = 0,lines = 0;

```

```

printf("ENTER THE TEXT.\n");
printf("GIVE ONE SPACE AFTER EACH WORD.\n");
printf("WHEN COMPLETED, PRESS 2 TIMES ENTER BUTTON.\n\n");
while(end == 0)
{
    /* Reading a line of text */
    c = 0;
    while((ctr=getchar()) != '\n')
        line[c++] = ctr;
    line[c] = '\0';
    /* counting the words in a line */
    if(line[0] == '\0')
        break ;
    else
    {
        words++;
        for(i=0; line[i] != '\0';i++)
            if(line[i] == ' ' || line[i] == '\t')
                words++;
    }
    /* counting lines and characters */
    lines = lines + 1;
    characters = characters + strlen(line);
}
printf ("\n");
printf("Number of lines = %d\n", lines);
printf("Number of words = %d\n", words);
printf("Number of characters = %d\n", characters);
getch();
}

```

Example: Write a c program for swapping of two string.

```

#include<stdio.h>
main()
{
    int i=0,j=0,k=0;
    char str1[20],str2[20],temp[20];
    puts("Enter first string");
    gets(str1);
    puts("Enter second string");
    gets(str2);
    printf("Before swaping the strings are\n");
    puts(str1);
    puts(str2);
    while(str1[i]!='\0')
    {
        temp[j++]=str1[i++];
    }
    temp[j]='\0';
    i=0,j=0;
    while(str2[i]!='\0')

```

```

    {
        str1[j++]=str2[i++];
    }
    str1[j]='\0';
    i=0,j=0;
    while(temp[i]!='\0')
    {
        str2[j++]=temp[i++];
    }
    str2[j]='\0';
    printf("After swaping the strings are\n");
    puts(str1);
    puts(str2);
}

```

Example: Write a C program to insert a sub-string in to given main string from a given position.

```

#include <stdio.h>
#include <conio.h>
#include <string.h>
main()
{
    char a[10];
    char b[10];
    char c[10];
    int p=0,r=0,i=0;
    int t=0;
    int x,g,s,n,o;
    clrscr();
    puts("Enter First String:");
    gets(a);
    puts("Enter Second String:");
    gets(b);
    printf("Enter the position where the item has to be inserted: ");
    scanf("%d",&p);
    r = strlen(a);
    n = strlen(b);
    i=0;
    /*Copying the input string into another array*/
    while(i <= r)
    {
        c[i]=a[i];
        i++;
    }
    s = n+r;
    o = p+n;
    /* Adding the sub-string*/
    for(i=p;i<s;i++)
    {
        x = c[i];
        if(t<n)

```

```

        {
            a[i] = b[t];
            t=t+1;
        }
        a[o]=x;
        o=o+1;
    }
    printf("%s", a);
    getch();
}

```

Character operations: c language provides character I/O functions as a part of stdio library and an extensive collection of character oriented functions available in the library ctype.h.

The stdio library includes a function named getchar() that is used to get the single character from the standard input source.

Unlike scanf(), getchar() does not expect the calling module to pass as an argument the address of a variable in which to store the input character. rather, getchar() takes no arguments and returns the character as its result

Example: scanf("%c",&ch);
 Ch=getchar();

To get a single character from a file, use the function getc

Example: getc(inp);

Here the character is obtained from the file accessed by file pointer inp

The standard library's single character output functions are

1. putchar()-It displays character on output device.
2. putc-It displays character on file.

Character Oriented functions:(Required header: #include <ctype.h>)

Category	ASCII Characters
Uppercase letters	'A' through 'Z'
Lowercase letters	'a' through 'z'
Digits (0 through 9)	'0' through '9'
Whitespace	Space, tab, line feed(newline), and carriage return
Punctuation	!"#\$%&'()*+,-./:;<=>?@[\\]^_`{ }~
Blank space	The blank space character

isalnum(): It returns a TRUE (nonzero) if the argument is digit 0-9, or an alphabetic character (alphanumeric). Otherwise returns FALSE.

Syntax: **isalnum(ch);**

Example: #include <stdio.h>
 main()

```

{
    if( isalnum( ';' ) )
    {
        printf( "Character ; is not alphanumeric\n" );
    }
    if( isalnum( 'A' ) )
    {
        printf( "Character A is alphanumeric\n" );
    }
}

```

Output: Character A is alphanumeric

isalpha(): It returns TRUE if the argument is an upper or lower case letter.

Syntax: **isalpha(ch);**

Example: #include <stdio.h>

```

main()
{
    if( isalpha( '9' ) )
    {
        printf( "Character 9 is not alpha\n" );
    }
    if( isalpha( 'A' ) )
    {
        printf( "Character A is alpha\n" );
    }
}

```

Output: Character A is alpha

isdigit(): It returns TRUE if the argument is a digit 0 – 9, otherwise it returns FALSE.

Syntax: **isdigit(ch);**

Example: if(isdigit(ch))
 Printf(“character is digit”);
 else
 printf(“not a digit”);

islower(): It returns TRUE if the argument is a lowercase letter.

Syntax: **islower(ch);**

Example: if(islower(ch))
 Printf(“character is lowercase alphabet”);
 else
 printf(“not lowercase alphabet”);

isupper(): It returns TRUE if the argument is an uppercase letter.

Syntax: **isupper(ch);**

Example: if(isupper(ch))
 Printf(“character is uppercase alphabet”);
 else
 printf(“not uppercase alphabet”);

ispunct(): It returns TRUE if the argument is any punctuation character .

Syntax: **ispunct(ch);**

Example: if(ispunct(ch))
 Printf("character is punctuation character");
 else
 printf("not punctuation character");

isspace():It returns TRUE if the argument is a whitespace (see chart above).

Syntax: **isspace(ch);**

Example: if(isspace(ch))
 Printf("character is whitespace character");
 else
 printf("not whitespace character");

toascii():It converts the argument (an arbitrary integer) to a valid ASCII character number 0-127.

Syntax: **toascii(ch);**

Example: c = toascii(500);
 gets number 116(modulus 500%128)

Example: c = toascii('d');
 c gets number 100

tolower():It converts the argument (an uppercase ASCII character) to lowercase.

Syntax: **tolower(ch);**

Example: c = tolower('Q');
 c becomes 'q'

toupper():It converts the argument (a lowercase ASCII character) to uppercase.

Syntax: **toupper(ch);**

Example: c = toupper('q');
 c becomes 'Q'

UNIT V

Functions

Top down approach:

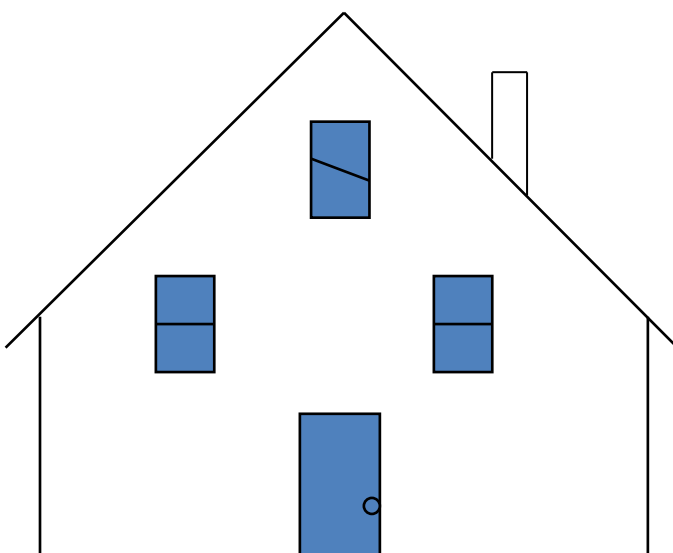
- ✚ C programmer follows Top-down approach. On other words procedural programming follows this approach. Top-down approach also called as step-wise approach.
- ✚ Top-down approach starts with high level system or design then it goes to low level system or design or development.
- ✚ Top-down approach first focus on abstract of overall system or project. At last it focuses on detail design or development.
- ✚ In this approach first programmer has to write code for main function. In main function they will call other sub function. At last they will write code for each sub function.
- ✚ Top-down approach expects good planning and good understanding of the system or project.

Advantages:

- ✚ Breaking the problem into parts helps us to clarify what needs to be done.
- ✚ At each step of refinement, the new parts become less complicated and, therefore, easier to figure out.
- ✚ Parts of the solution may turn out to be reusable.
- ✚ Breaking the problem into parts allows more than one person to work on the solution.

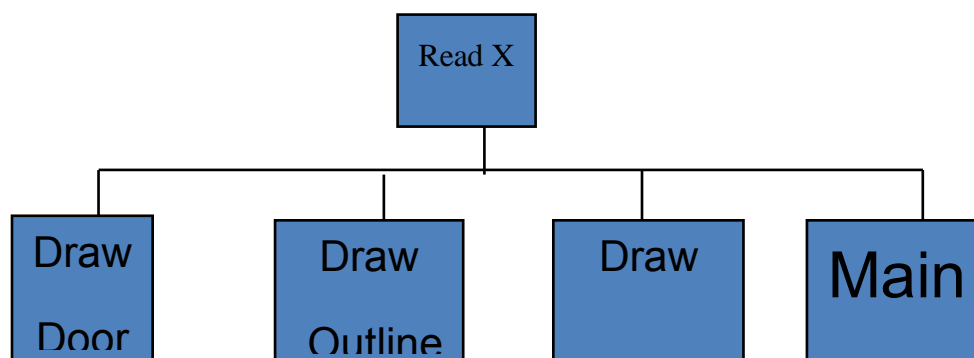
Example:

Problem: Write a program that draws this picture of a house.



The Top Level:

- ✚ Draw the outline of the house
- ✚ Draw the chimney
- ✚ Draw the door
- ✚ Draw the windows

**Function:**

A program can be used solving a simple problem (or) a large and complex problem. Programs solving simple problems are easier to understand and identify mistakes, if any in them. If problems are large, it is difficult to understand the steps involved in it. Hence it is subdivided into a number of smaller programs called subprograms (or) modules.

Each subprogram specifies one (or) more actions to be performed for the larger program. Such sub programs are called as functions.

Definition: A Function is a self-contained block of code that performs a particular task.

C functions can be classified into two categories,

- ✚ Library functions
- ✚ User-defined functions

Library functions: Library functions are part of the C compiler that has been written for general purposes are called library functions. They are also called built in functions. Library functions are defined by C library, the standard C library is a collection of various types of functions which perform some standard and pre-defined tasks. For example, printf(), scanf(), strcat() etc. You just need to include appropriate header files to use these functions. These are already declared and defined in C libraries.

Example: sqrt() – Finds the Square root of a given number.

Example:

```

#include<stdio.h>
#include<conio.h>
#include<math.h>
main()
{

```



```

float n,s;
scanf("%f",&n);
s=sqrt(n);
printf("Square root of %f =%f",n,s);
getch();
}

/*called function*/
float sqrt(int n)
{
    stmt1;
    stmt2;
    return result;
}

```

Output: Square root of 36 = 6.00000 { $\sqrt{36} = 6.00000$ }

In c language, main() is the first calling function in any program. It is a special function which tells the compiler to start the execution of c program from the beginning of the function main().it is not possible to have more than one main() function because the compiler will not know where to start execution in such situation.

Execution:

- ✚ Execution starts from the function main().
- ✚ The main() function is calling the function sqrt(). So, the function main() is called “Calling Function”.
- ✚ When a function is called the control is transferred from main() to sqrt() is shown in above program.
- ✚ Each statement in the function sqrt() is executed and square root of a given number is computed.
- ✚ After executing the last statement i.e., *return result;* the control will be transferred to the function main() along with result.
- ✚ The result obtained from sqrt() is copied into variable.
- ✚ The result is displayed on the screen.
- ❖ **A function that invokes (calls) another function is known as calling function**
- ❖ **A function which is invoked (called) by another function is known as called function.**

User-defined functions: User-defined functions are those functions which are defined by the user at the time of writing program to do some specific tasks are called user-defined functions. However, a user defined function can later become a part of the c program library. Functions are made for code reusability and for saving time and space. User defined functions are subordinate to main() and also to one another.

Example: consider a program for addition of two numbers.

```

#include<stdio.h>
#include<conio.h>
main()
{
    int a,b,sum;

```

```

        printf("Enter the values of a and b :");
        scanf("%d%d",&a,&b);
        sum=a+b;
        printf("%d",sum);
    }

```

The above program is very simple. It accepts two numbers adds these numbers and displays the result.

Can we write a user defined function to add two numbers?

- ✚ The previous program itself is a function. But the name of the function is main().
- ✚ If we want to write our own function, then just change the name of the function main to add i.e.,

Example:

```

#include<stdio.h>
void add()
{
    int a,b,sum;
    printf("Enter the values of a and b :"); scanf("%d%d",&a,&b);
    sum=a+b;
    printf("%d",sum);
}

```

So, we have defined a function whose name is add() this is called Function Definition. This is also called as user defined function.

But, it is not complete. we know that execution always starts from function main(). But there is no main(). so, we have to write one function main() which calls the function add.

Example: consider a program for addition of two numbers.

```

#include<stdio.h>
#include<conio.h>
void main()
{
    add();
}
void add()
{
    int a,b,sum;
    printf("Enter two values A and B:");
    scanf("%d%d",&a,&b);
    sum=a+b;
    printf("Result : %d", sum);
}

```

Advantages of functions:

- ✚ C functions are used to avoid rewriting same logic/code again and again in a program.
- ✚ There is no limit in calling C functions to make use of same functionality wherever required.
- ✚ We can call functions any number of times in a program and from any place in a program.
- ✚ A large C program can easily be tracked when it is divided into functions.

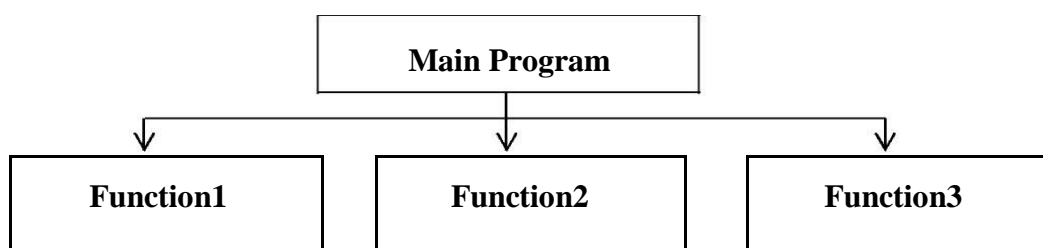
- ✚ The core concept of C functions are, re-usability, dividing a big task into small pieces to achieve the functionality and to improve understandability of very large C programs.

Need for user defined functions: As pointed out earlier, main is specially recognized function in c language. Every program must have a main function to indicate where the program has to begin its execution. While it is possible to code any program utilizing only main function, it leads to a number of problems. The program may become too large and complex and as a result of task debugging, testing, and maintaining becomes difficult. If a program is divided into functional parts, then each part may be independently coded and later combined into a single unit. These independently coded programs are called sub programs that are much easier to understand, debug, and test. In c such subprograms are referred to as functions.

There are times when certain types of operations (or) calculations are repeated at many points throughout a program. For, instance, we might use the factorial of a number at several times in the program. In such situations, we may repeat the program statements where ever they are needed. Another approach is to design a function that can be called and used whenever required. This saves both time and space.

This division approach clearly results in a number of advantages

- ✚ It facilitates top-down modular programming. The high logic of the overall problem is solved first while the details of each lower level function are addressed later.
- ✚ The length of source program can be reduced by using functions at appropriate places.
- ✚ It is easy to locate & isolate a faulty function for further investigations.
- ✚ A function may be used by many other programs. This means that a programmer can build on what others have already done, instead of staring all over again from scratch.



User defined functions: In order to make use of a user defined function, we need to establish 3 elements that are related to functions.

- ✚ Function definition
- ✚ Function call
- ✚ Function declaration

Function Definition: A function is a block of code that performs a particular task Or The program module that is written to achieve a specific task is called Function Definition.



Example:

function header: The function header consists of three parts:the function type(also known as return type),the function name and the formal parameter list.Note that a semicolon is not used at the end of the function header.

type: The function type specifies the type of value (can be int, float, double, void), that the function is expected to return to the program calling the function. If the function is not returning anything then we need to specify the return type as void. If the return type is not explicitly specified, c will assume that it is an integer type. The value returned is the output produced by the function.

name: The function name is any valid identifier, and therefore, the rules that are followed to frame variable are followed to frame function name. Normally the name of the function itself should indicate the activity being performed by the function. In the example, add is the name of the function and it performs addition operation.

parameters_list: Parameters_list declares list of variables. All the variables should be separately declared and each declaration should be separated by a comma. All the parameters should be enclosed within (and). These variables accept the values which are sent by the calling program. They serve as

input data to the function to carry out the specified task. These variables are called formal parameters (or) dummy parameters. **In the example variables m & n are called formal parameters.**

function-body: The function body contains the declarations and the statements necessary for performing the required task and return statement. The body is enclosed within curly braces { } and consists of three parts

function body=declaration part + execution part + return statement

declaration part: Apart from the parameters, whatever variables that are used in the function should be declared. These variables are called local variables. In the above example the variable sum is declared as local variable.

Execution part: This part contains the statements that perform the actual job of the function.

Return Statement: Using this statement, the function returns the evaluated result. If a function does not return any value, the return statement can be omitted. In the example, the function returns sum of two numbers. If the function does not return any value, we can omit the return statement. However, note that its return type should be specified as void.

return values and their type: As pointed earlier, a function may Or may not send back any value to the calling function. If does, it is done through the return statement. While it is possible to pass to the called function any number of values, the called function can only return one value per call, at the most. The return statement can take one of the following forms:

return; This statement is used when the function is not sending any value to the calling function.

Example: if(error)
 return;

return expression; This statement is used when the function is sending a value to the calling function.

Example: int mul(int x,int y)
 {
 int p;
 p=x*y;
 return(p);
 }

In the above example it returns the value of p which is the product of the values x and y .The last two statements can be combined into one statement as follows:

 return(x*y);

Function call: After writing the functions, appropriate functions have to be invoked to get the job done from those functions. This invocation of a function is called as function call. After writing the functions, appropriate function has to be invoked to get the job done from the functions. The invocation of a function is called function call.

Execution starts from the function `main()`. After reading the values of `m` and `n`, the function is invoked using the statement:

The variables that are used while calling the function are called actual parameters.

S.No	Actual Parameters	Formal Parameters
1	<p>The arguments that are passed in a function call are called actual arguments. These arguments are defined in the calling function</p> <p>Example: c=sum(a,b); Here a & b are actual parameters</p>	<p>Formal Parameters are used in function header of a called function.</p> <p>Example: int sum(int m,int n) <div style="text-align: center;">{ } }</div> Here m & n are formal parameters</p>
2	<p>Actual parameters can be constants,variables(or) expressions.</p> <p>Example: c=sum(a+4,b)</p>	<p>Formal parameters should be only variables. Expressions and constants are not allowed.</p> <p>Example: int sum(int m,int n) <i>Correct</i> int sum(int m+n,int n) <i>Wrong</i></p>
3	<p>Actual parameters send values to the formal parameters.</p>	<p>Formal parameters receives values from the actual parameters</p>
4	<p>Address of actual parameters can be sent to formal parameters.</p>	<p>If formal parameters contain addresses, they should be declared as pointers.</p>

Function declaration: Like all the variables, all functions in c program must be declared, before they are invoked. A function declaration consists of four parts:

- ✚ **Function type**
- ✚ **Function name**
- ✚ **Parameter list**
- ✚ **Terminating semicolon**

Syntax: **function_type function_name(parameter_list);**

Example: **int mul(int m,int n);**

type: The function type specifies the type of value (can be int, float, double, void), that the function is expected to return to the program calling the function. If the function is not returning anything then we need to specify the return type as void. If the return type is not explicitly specified, c will assume that it is an integer type. The value returned is the output produced by the function.

name: The function name is any valid identifier, and therefore, the rules that are followed to frame variable are followed to frame function name. Normally the name of the function itself should indicate the activity being performed by the function. In the example, add is the name of the function and it performs addition operation.

parameters_list: Parameters_list declares list of variables. All the variables should be separately declared and each declaration should be separated by a comma. All the parameters should be enclosed within (and).

Function declaration is very similar to the function header line except the terminating semicolon. If the value returned by a function is int type, the declaration of function is optional. For all other types of functions the declaration is mandatory.

In function declaration, parameter names in the parameter_list are optional. Hence it is possible to have datatype of each parameter without mentioning the parameter name that is

Syntax: **function_type function_name(data_type,data_type.....data_type);**

Example: **int sum(int,int,int...int);**

Parameter less function is declared by using void inside the parentheses that is

Syntax: **function_type function_name(void);**

Categories of Functions with respect to parameters and return values: Based on the parameters and return values, the functions can be categorized as shown below,

- ✚ Function with no parameters and no return values
- ✚ Function with no parameters and return values
- ✚ Function with parameters and no return values

- ✚ Function with parameters and return values
- ✚ Function that returns multiple values

Function with no parameters and no return values:

- ✚ In this category, there is no data transfer between the calling function and the called function. But there is a flow of control from calling function to the called function.
- ✚ Function without any arguments means no data is passed (values like int, char, etc..) to the called function.
- ✚ Similarly, function with no return type does not pass back data to the calling function.

Example:

```
#include<stdio.h>
void sum();
void main()
{
    sum();
}
void sum()
{
    int a,b,c;
    printf("Enter a and b values:");
    scanf("%d%d",&a,&b);
    c=a+b;
    printf("Sum = %d",c);
}
```

- ✚ In the previous program, the function sum do not receive any values from the function main() and it does not return any values to the function main().
- ✚ But we still input two numbers from the keyboard and perform addition of two numbers.

Example1:

```
#include<stdio.h>
#include<conio.h>
void printline();
void main()
{
    clrscr();
    printf("Welcome to function in C");
    printline();
    printf("Function easy to learn.");
    printline();
    getch();
}
void printline()
{
    int i;
    printf("\n");
    for(i=0;i<30;i++)
    {
        printf("-");
    }
}
```



```

        printf("\n");
    }

```




Example: C program to check whether a number entered by user is prime or not using function with no arguments and no return value

```

#include <stdio.h>
void prime();
main()
{
    prime();    /*No argument is passed to prime()*/
}
void prime()
{
    /* There is no return value to calling function main().
       Hence, return type of prime() is void */
    int num,i,flag=0;
    printf("Enter positive integer enter to check:\n");
    scanf("%d",&num);
    for(i=2;i<=num/2;++i)
    {
        if(num%i==0)
        {
            flag=1;
        }
    }
    if (flag==1)
        printf("%d is not prime",num);
    else
        printf("%d is prime",num);
}

```

Function with no parameters and return values

-  In this category there is no data transfer from the calling function to the called function. But there is data transfer from called function to the calling function.
-  When no parameters are there, the function does not receive any value from the calling function.
-  When function returns a value, the calling function receives a value from the called function.

Example:

```

#include<stdio.h>
int sum();
void main()
{
    int c;
    c=sum();
    printf("%d",c);
}
int sum()
{
    int a,b,c;
    printf("Enter a and b values");
    scanf("%d%d",&a,&b);
}

```

```

        c=a+b;
        return c;
    }

```

✚ In the above program, the function does not receive any value from the calling function main().

✚ But it accepts data from the keyboard, finds the sum of those numbers and returns the result to the calling function.

Example1:

```

#include<stdio.h>
#include<conio.h>
int send()
{
    int no1;
    printf("Enter a no : ");
    scanf("%d",&no1);
    return(no1);
}
void main()
{
    int z;
    clrscr();
    z = send();
    printf("\nYou entered : %d.", z);
    getch();
}

```

Example: C program to check whether a number entered by user is prime or not using function with no arguments but having return value

```

#include <stdio.h>
int input();
main()
{
    int num,i,flag = 0;
    num=input();
    for(i=2; i<=num/2; ++i)
    {
        if(num%i==0)
        {
            flag = 1;
            break;
        }
    }
    if(flag == 1)
        printf("%d is not prime",num);
    else
        printf("%d is prime", num);
}
int input()
{ /* Integer value is returned from input() to calling function */
    int n;

```

```

        printf("Enter positive integer to check:\n");
        scanf("%d",&n);
        return n;
    }

```

Functions with parameters and no return values:-

- ✚ In this category there is transfer of data from calling function to the called function using parameters. But there is no data transfer from called function to the calling function.
- ✚ When parameters are passed, the function can receive values from the calling function.
- ✚ When the function does not return any value, the calling function cannot receive any value from the called function.

Example:

```

#include<stdio.h>
void sum(int a,int b);
void main()
{
    int m,n;
    printf("Enter m and n values");
    scanf("%d%d",&m,&n);
    sum(m,n);
}
void sum(int x, int y)
{
    int a;
    a=x+y;
    printf("Result = %d",a);
}

```

- ✚ In the above program, the function sum() receives two values from the calling function main(), find the sum of these numbers and displays the result there itself.
- ✚ The result is not passed to the calling function, but only control is transferred.

Example1:

```

#include<stdio.h>
int factorial(int m);           /*function declaration*/
main( )
{
    int n;
    scanf("%d",&n);
    factorial(n);              /*Function calling*/
}
int factorial(int m)           /*function definition*/
{
    int i,p=1;
    for(i=1;i<=m;++i)
        p=p*i;
    printf("\n factorial of %d is %d ",m,p);
}

```

Example2: #include<stdio.h>

```

#include<conio.h>
void add(int , int );
void main()
{
    clrscr();
    add(30,15);
    add(63,49);
    add(952,321);
    getch();
}
void add(int x, int y)
{
    int result;
    result = x+y;
    printf("Sum of %d and %d is %d.\n\n",x,y,result);
}

```

Example: Program to check whether a number entered by user is prime or not using function with arguments and no return value

```

#include <stdio.h>
void check_display(int n);
main()
{
    int num;
    printf("Enter positive enter to check:\n");
    scanf("%d",&num);
    check_display(num); /* Argument num is passed to function. */
}
void check_display(int n)
{ /* no return value to calling function. Hence, return type of function is void. */
    int i, flag = 0;
    for(i=2; i<=n/2; ++i)
    {
        if(n%i==0)
        {
            flag = 1;
            break;
        }
    }
    if(flag == 1)
        printf("%d is not prime",n);
    else
        printf("%d is prime", n);
}

```

Function with parameters and return values:

- ✚ In this category, there is data transfer between the calling function and the called function.
- ✚ When parameters are passed, the called function can receive values from the calling function.
- ✚ When the function returns a value, the calling function can receive a value from the called

function.

Example:

```
#include <stdio.h>
int sum(int a, int b)
void main()
{
    int m,n,c;
    printf("Enter m and n values:");
    scanf("%d%d",&m,&n);
    c=sum(m,n);
}
int sum( int x,int y)
{
    int d;
    d=x+y;
    return d;
}
```

In the above program the function receives two values from the function main(), finds the sum of these numbers and sends the result back to the calling function.

Example1:

```
#include<stdio.h>
int maximum(int x,int y);
main( )
{
    int a,b,c;
    scanf("%d%d",&a,&b);
    c=maximum(a,b);
    printf("\n maximum number is : %d",c);
}
int maximum(int x, int y)
{
    int z;
    z=(x>=y)?x:y;
    return(z);
}
```

Example2:

```
#include<stdio.h>
#include<conio.h>
int add(int , int );
void main()
{
    int z;
    clrscr();
    z = add(952,321);
    printf("Result %d.\n\n",add(30,55));
    printf("Result %d.\n\n",z);
}
int add(int x, int y)
{
    int result;
    result = x+y;
```

```

        return(result);
    }

```

Example: Program to check whether a number entered by user is prime or not using function with argument and return value

```

#include <stdio.h>
int check(int n);
main()
{
    int num,num_check=0;
    printf("Enter positive enter to check:\n");
    scanf("%d",&num);
    num_check=check(num); /* Argument num is passed to check() function. */
    if(num_check==1)
        printf("%d is not prime",num);
    else
        printf("%d is prime",num);
}

int check(int n)
{
    /* Integer value is returned from function check() */
    int i;
    for(i=2;i<=n/2;++i)
    {
        if(n%i==0)
            return 1;
    }
    return 0;
}

```

Example: Factorial without using recursion

```

#include <stdio.h>
#include <conio.h>
void main()
{
    int n, a, b;
    clrscr();
    printf("Enter any number\n");
    scanf("%d", &n);
    b = nonrecfactorial(n);
    printf("The factorial of a given number using nonrecursion is %d ", b);
    getch();
}

int nonrecfactorial(int x)
{
    int i, f = 1;
    for(i = 1; i <= x; i++)
    {
        f = f * i;
    }
    return(f);
}

```

Example: To calculate GCD using recursive function.

```

#include <stdio.h>
int gcd(int n1, int n2);
main()
{
    int n1, n2;
    printf("Enter two positive integers: ");
    scanf("%d%d", &n1, &n2);
    printf("GCD of %d and %d = %d", n1, n2, gcd(n1,n2));
}
int gcd(int n1, int n2)
{
    if (n2!=0)
        return hcf(n2, n1%n2);
    else
        return n1;
}

```

Example: To find the GCD (greatest common divisor) of two given integers without using recursion

```

#include <stdio.h>
int gcd(int, int);
main()
{
    int a, b, result;
    printf("Enter the two numbers to find their HCF: ");
    scanf("%d%d", &a, &b);
    result = gcd(a, b);
    printf("The GCD of %d and %d is %d.\n", a, b, result);
}
int gcd(int a, int b)
{
    while (a != b)
    {
        if (a > b)
        {
            a = a - b;
        }
        else
        {
            b = b - a;
        }
    }
    return a;
}

```

Example: To find the GCD (greatest common divisor) of two given integers without using recursion

```

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

```

```

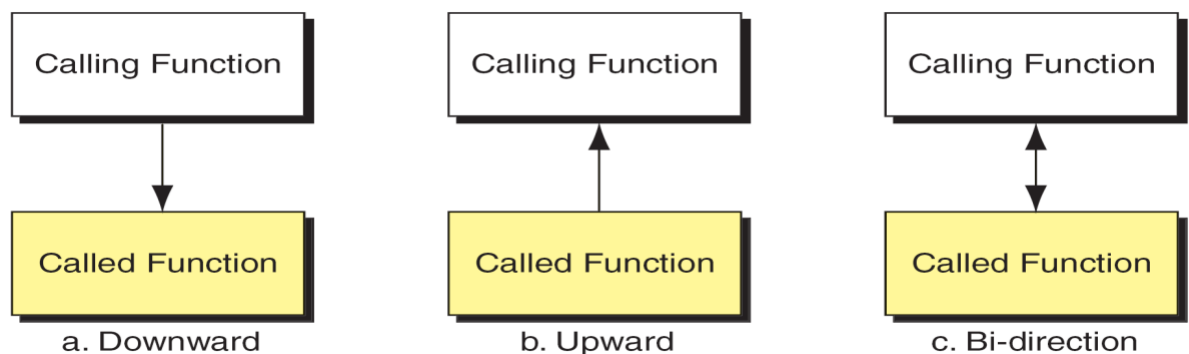
        clrscr();
        printf("Enter two numbers a, b\n");
        scanf("%d%d", &a, &b);
        d = nonrecgcd(a, b);
        printf("The gcd of two numbers using nonrecursion is %d", d);
        getch();
    }
    int nonrecgcd(int x, int y)
    {
        int z;
        while(x % y != 0)
        {
            z = x % y;
            x = y;
            y = z;
        }
        return(y);
    }
}

```

Inter-function Communication: Functions have to communicate between them to exchange data

The data flow between the calling and called functions can be divided into three types

- ✚ A downward flow – from the calling to the called function
- ✚ An upward flow- from the called to the calling function
- ✚ A bi-directional flow -in both directions



Downward flow

- ✚ In downward communication, the calling function sends data to the called function
- No data flows in the opposite direction
- ✚ Copies of data items are passed from the calling function to the called function.
- ✚ The called function may change the values passed, but the original values in the calling function remain untouched

Example:

```

#include<stdio.h>
int downflow( int x, int y);
int main(void)
{
    int a=15;
    downflow(a,5);
}

```



```

        printf("%d",a);
    }
    int downflow(int x,int y)
    {
        x=x+y
    }

```

Upward flow: Upward communication occurs when the called function sends data back to the calling function without receiving any data from it.

Bi-directional flow:

- ✚ Bi-directional communication occurs when the calling function sends data down to the called function.
- ✚ During or at the end of its processing, the called function then sends data up to the calling function

Passing parameters to a function (Or) parameter passing mechanism:

- ✚ The term parameter-passing refers to the different ways in which parameters (or arguments) are passed to or from a function.
- ✚ There are two basic models on how data transfer takes place between functions:

call_by_value: In this type value of actual arguments are passed to the formal arguments of the called function. Any changes made in the formal arguments does not effect the actual arguments because formal arguments are photocopy of actual arguments. Hence when the function is called by the call by value method, it does not effect the actual contents of the actual arguments. Changes made in the formal arguments are local to the block of called function. Once control returns back to the calling function the changes made vanish.

- ✚ In call by value method, the value of the variable is passed to the function as parameter.
- ✚ The value of the actual parameter cannot be modified by formal parameter.
- ✚ Different Memory is allocated for both actual and formal parameters. Because, value of actual parameter is copied to formal parameter.

Example:

```

#include<stdio.h>
void exchange(int m, int n)
{
    int temp;
    temp=m;
    m=n;
    n=temp;
}
void main()
{
    int a=10,b=20;
    exchange(a,b);
}

```

```

        printf("%d %d",a,b);
    }
Output:      10 20
Example1:   #include<stdio.h>
                void add( int n);
                main()
                {
                    int num = 2;
                    printf("\n The value of num before calling the function = %d", num);
                    add(num);
                    printf("\n The value of num after calling the function = %d", num);
                }
                void add(int n)
                {
                    n = n + 10;
                    printf("\n The value of num in the called function = %d", n);
                }

```

Output: The value of num before calling the function = 2
The value of num in the called function = 12
The value of num after calling the function = 2

call_by_reference: In *call by reference*, the address of actual arguments is passed to formal arguments of the called function and any change made to the formal arguments in the called function have effect on the values of actual arguments in the calling function

- 🚦 In call by reference method, the address of the variable is passed to the function as parameter.
- 🚦 The value of the actual parameter can be modified by formal parameter.
- 🚦 Same memory is used for both actual and formal parameters since only address is used by both parameters

```

Example:   #include<stdio.h>
                void exchange(int *m, int *n);
                void exchange (int *m, int *n)
                {
                    int temp;
                    temp=*m;
                    *m=*n;
                    *n=temp;
                }
                void main()
                {
                    int a=10,b=20;
                    exchange(&a,&b);
                    printf("%d%d",a,b);
                }

```

Output: a=20, b=10

```

Example1: #include<stdio.h>
                void add( int *n);
                main()

```

```

{
    int num = 2;
    printf("\nThe value of num before calling the function = %d", num);
    add(&num);
    printf("\n The value of num after calling the function = %d", num);
}
void add( int *n)
{
    *n = *n + 10;
    printf("\n The value of num in the called function = %d", *n);
}

```



Output: The value of num before calling the function = 2
The value of num in the called function = 12
The value of num after calling the function = 12

Difference between call by value and call by reference:

S.No	Call by value	Call by reference
1	When a function is called, the values of variables are passed	When a function is called the address of variables are passed
2	The type of formal parameters should be same as type of actual parameters	The type of formal parameters should be same as type of actual parameters, but they are to be declared as pointers
3	Formal parameters contain the values of actual parameters	Formal parameters contain the address of actual parameters.
4	Any update made inside the called function will not affect the original value of variable in calling function.(actual parameters)	Any update made on formal parameters in the called function will affect the original value of variable in calling function.(actual parameters)
5	Execution is slower since all the values have to be copied into formal parameters.	Execution is faster since only addresses are copied.





Recursion: A function calling itself again and again to compute a value is known as recursive function or recursion function or recursion. Normally a function is called by the main program or by some other function but in recursion the same function is called by itself repeatedly.

Every recursive solution has two major cases, they are:

-  Base Case
-  Recursive Case

Base case: The problem is simple enough to be solved directly without making any further calls to the same function.

Recursive case:

-  first, the problem is divided into simpler sub parts.
-  Second, the function calls itself but with sub parts of the problem obtained in the first step.
-  Third, the result is obtained by combining the solutions of simpler sub-parts.
-  Therefore, recursion is defining large and complex problems in terms of a smaller and more

easily solvable problem.

Factorial of A Number Using Recursion

PROBLEM	SOLUTION
5!	5 X 4 X 3 X 2 X 1!
= 5 X 4!	= 5 X 4 X 3 X 2 X 1
= 5 X 4 X 3!	= 5 X 4 X 3 X 2
= 5 X 4 X 3 X 2!	= 5 X 4 X 6
= 5 X 4 X 3 X 2 X 1!	= 5 X 24
	= 120

Base case is when $n=1$, because if $n = 1$, the result is known to be 1

Recursive case of the factorial function will call itself but with a smaller value of n , this case can be given as

$$\text{factorial}(n) = n \times \text{factorial}(n-1)$$

We have two types of recursion **1. Direct recursion**

2. Indirect recursion

Direct recursion: If a function calls itself in the function body of its function definition, it is known as direct recursion.

Example:

```
int Func( int n)
{
    if(n==0)
        return n;
    return (Func(n-1));
}
```

Indirect recursion: If a function calls another function, which in turn calls the first function, then it is called as indirect recursion.

Example:

```
int Func1(int n)
{
    if(n==0)
        return n;
    return Func2(n);
}
int Func2(int x)
{
    return Func1(x-1);
}
```

Example: Source Code to Calculated Sum of n natural numbers using Recursion

```
#include<stdio.h>
int add(int n);
```

```

main()
{
    int n;
    printf("Enter an positive integer: ");
    scanf("%d",&n);
    printf("Sum = %d",add(n));
}
int add(int n)
{
    if(n!=0)
        return n+add(n-1); /* recursive call */
}

```

Example: Factorial of A Number Using Recursion

```

#include<stdio.h>
int fact(int n)
{
    if(n==1)
        return 1;
    else
        return (n * fact(n-1));
}
main()
{
    int num,f;
    printf("enter num value ");
    scanf("%d", &num);
    f=fact(num);
    printf("\n Factorial of %d = %d", num,f);
}

```

Example: c program to print Fibonacci series using recursion.

```

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

```

Example: To find the GCD (greatest common divisor) of two given integers using recursion.

```

#include <stdio.h>
#include <conio.h>
void main()
{
    int a, b, c, d;
    clrscr();
    printf("Enter two numbers a, b\n");
    scanf("%d%d", &a, &b);
    c = recgcd(a, b);
    printf("The gcd of two numbers using recursion is %d\n", c);
    getch();
}
int recgcd(int x, int y)
{
    if(y==0)
    {
        return(x);
    }
    else
    {
        return(recgcd(y,x%y));
    }
}

```

Example: To find the GCD (greatest common divisor) of two given integers using recursion.

```

#include <stdio.h>
int gcd(int, int);
main()
{
    int a, b, result;
    printf("Enter the two numbers to find their GCD: ");
    scanf("%d%d", &a, &b);
    result = gcd(a, b);
    printf("The GCD of %d and %d is %d.\n", a, b, result);
}
int gcd(int a, int b)
{
    while(a!= b)
    {
        if(a > b)
        {
            return gcd(a-b, b);
        }
        else
        {
            return gcd(a,b - a);
        }
    }
    return a;
}

```

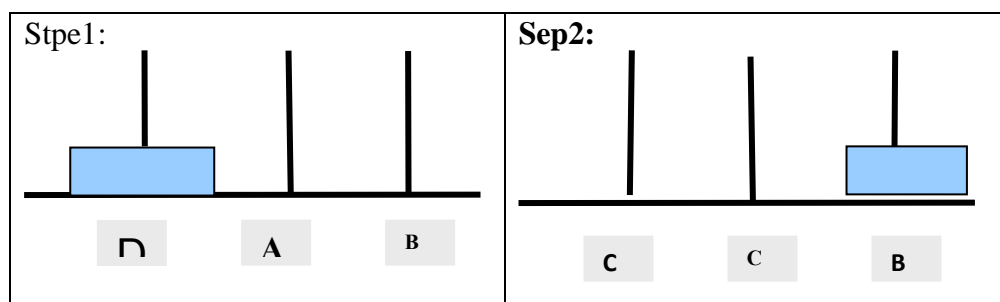
Towers Of Hanoi: The Tower of Hanoi puzzle was invented by the French mathematician Edouard

Lucas in 1883. He was inspired by a legend that tells of a Hindu temple where the puzzle was presented to young priests. At the beginning of time, the priests were given three poles and a stack of 64 gold disks, each disk a little smaller than the one beneath it. Their assignment was to transfer all 64 disks from one of the three poles to another, with following constraints.

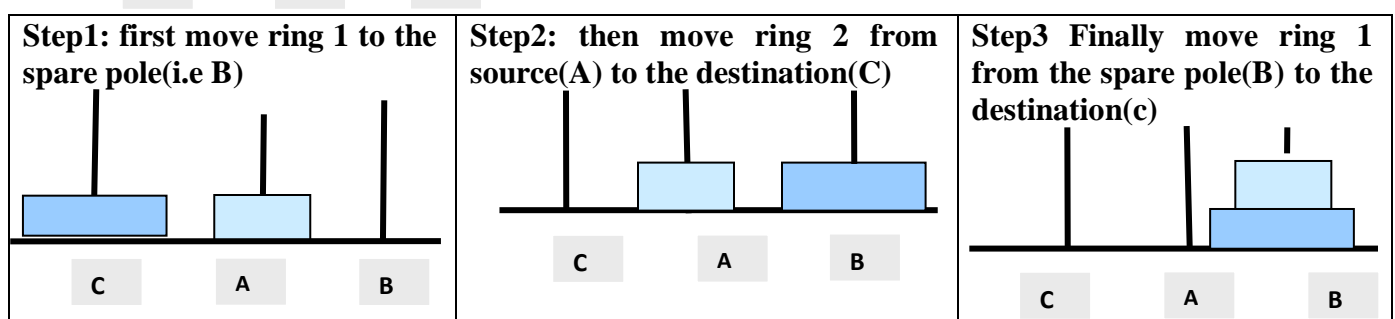
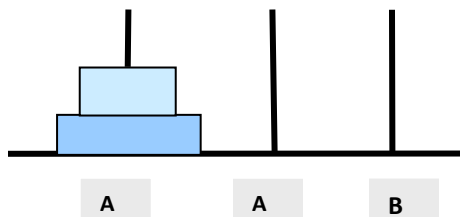
- 🚦 There are n disks (1, 2, 3,... n) and three towers (poles). The towers are labeled 'A', 'B', and 'C'.
- 🚦 All the disks are initially placed on the first source tower(the 'A' pole).
- 🚦 They could only move one disk at a time.
- 🚦 They could never place a larger disk on top of a smaller one.

Tower of Hanoi is one of the main applications of a recursion. It says, "if you can solve $n-1$ cases, then you can easily solve the n th case?"

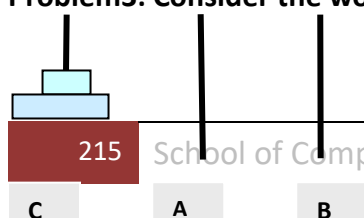
1. If there is only one ring, then move the ring from source to the Destination

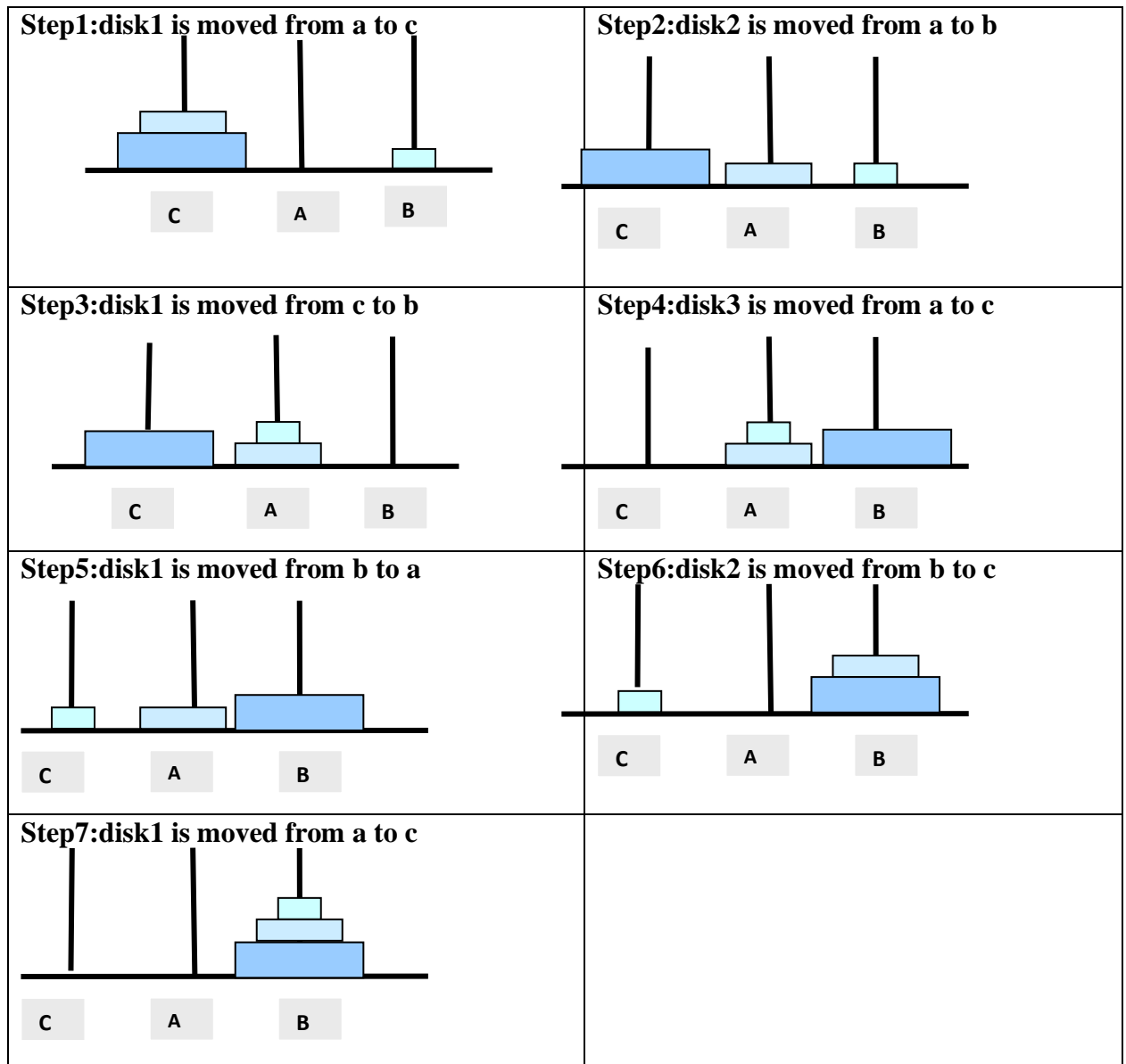
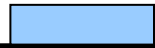


Problem: If there are two rings, then.



Problem3: Consider the working with three rings





Example: C Program to Solve Towers of Hanoi Problem

```
#include<stdio.h>
#include<conio.h>
void hanoi(int, char, char, char);
void main()
{
    int disks;
    char source, destination, aux;
    clrscr();
    printf("\nEnter No. of Disks:");
    scanf("%d",&disks);
    printf("\nEnter Source, Destination and Auxillary Disk Names:");
    scanf(" %c %c %c",&source, &destination, &aux);
```



```

        hanoi(disks,source,destination,aux);
        getch();
    }
    void hanoi(int n, char s, char d, char a)
    {
        if(n>0)
        {
            hanoi(n-1,s,a,d);
            printf("\n%d is moved from %c to %c",n,s,d);
            hanoi(n-1,a,d,s);
        }
    }
}

```

Scope of a variable: Scope relates to any object that is declared, such as variable declaration and function declaration. It determines the part of the program where the object is visible. The scope of local variables is limited to the functions in which they are declared, or in other words these variables are inaccessible outside of the function. Likewise the scope of the block variables is limited to the block in which they are declared. Global variables have a scope that spans the entire source program, which is why they can be used in any function.

Classification of the variables based on the place of declaration:

Local variables (have Local Scope): A variable declared inside a function is called a local variable. This name derives from the fact that a variable declared inside a function can be used only inside of that function. For example, the variable *i* declared within the block of the following main function has block scope:

Example:

```

main()
{
    int i;    /* block scope */
    .
    .
    .
}

```

Usually, a variable with block scope is called a local variable.

Global variables (have program scope): The variables you declare in the global declaration section are called global variables or external variables. While the local variable can be used inside the function in which it is declared, a global variable can be used anywhere in the program.

Example:

```

int x = 0;          /* program scope */
float y = 0.0;      /* program scope */
main()
{
    int i; /* block scope */
    .
    .
}

```

```

}
```

Here the int variable x and the float variable y have program (or) global scope.

Example:

```

#include <stdio.h>
int x = 1234;          /*program scope */
double y = 1.234567;   /*program scope */
void function_1()
{
    printf("From function_1:\n x=%d, y=%f\n", x, y);
}
main()
{
    int x = 4321;       /* block scope 1 */
    function_1();
    printf("Within the main block:\nx=%d, y=%f\n", x, y); /*a nested block */
    {
        double y = 7.654321;    /* block scope 2 */
        function_1();
        printf("Within the nested block:\n x=%d, y=%f\n", x, y);
    }
}
```

Output:

```

From function_1:    x=1234,y=1.234567
Within the main block: x=4321,y=1.234567
From function_1:    x=1234,y=1.234567
Within the nested block:x=4321,y=7.654321
```

Block variables (have Block Scope): The variable declared inside any block such variables are called block variables.

Example:

```

#include <stdio.h>
main()
{
    int i = 32; /* block scope 1 */
    printf("Within the outer block: i=%d\n", i);
    {
        /* the beginning of the inner block */
        int i, j;    /* block scope 2, int i hides the outer int i */
        printf("Within the inner block:\n");
        for (i=0, j=5; i<=5; i++, j--)
            printf("i=%2d, j=%2d\n", i, j);
    }
    /* the end of the inner block */
    printf("Within the outer block: i=%d\n", i);
}
```

Output:

```

Within the outer block: i=32
i=0,j=5
i=1,j=4
i=2,j=3
i=3,j=2
i=4,j=1
i=5,j=0
Within the outer block: i=32
```

Differentiate between Global & Local variables with examples

S.No	Local Variable	Global Variable
1	Local variables are declared inside a function.	Global Variables are declared before the main function.
2	Local Variables cannot be accessed outside the function.	Global Variables can be accessed in any function.
3	Local Variables are alive only for a function.	Global Variables are alive till the end of the program

Example (local variable): program to add any two integers

```

void main()
{
    int a,b,sum;
    clrscr();
    printf("Enter any two integer value");
    scanf("%d%d",&a,&b);
    sum=a+b;
    printf("\nSum of two integers %d and %d is %d",a,b,sum);
}

```

Here, a,b, and sum are local variables which are declared in main function.

Example(global variable): program to find the sum and difference between two numbers

```

#include<stdio.h>
int a,b,result;
void main()
{
    clrscr();
    sum();
    sub();
    getch();
}
sum()
{
    printf("Enter two numbers to find their sum");
    scanf("%d%d",&a,&b);
    result=a+b;
    printf("\n the sum of two numbers is %d",result);
}
sub()
{
    printf("Enter two numbers to find their difference");
    scanf("%d%d",&a,&b);
    result=a-b;
    printf("\n the difference between two numbers is %d",result);
}

```

Here, a, b and result are global variables which are declared before the main function.

STORAGE CLASSES: To fully define a variable in C, it is required to define not only its data type but also its storage class. If we don't specify the storage class of a variable in its declaration, the

compiler will assume default storage class dependent on the context in which the variable is used. The default storage class in C is auto.

The storage class of a variable in C determines the life time of the variable if this is 'global' or 'local'. Along with the life time of a variable, storage class also determines variable's storage location (memory or registers), the scope (visibility level) of the variable, and the initial value of the variable.

Types of Storage Classes in C:

- ✚ Automatic storage class.
- ✚ External storage class.
- ✚ Register storage class.
- ✚ Static storage class.

Automatic Storage Class: when the variables are declared as automatic storage class variable (i.e., Automatic variables), it is stored in the memory. Default value of automatic variable will be some garbage value. Scope of the variable is within the block where it is defined and the life time of the variable is until the control remains within the block. Because of this property, automatic variables are also referred to as local or internal variables.

Example:

```
main()
{
    int n;
    _____
    _____
}
```

We may also use the key word auto to declare automatic variables explicitly.

Syntax: auto data-type var_name;

Example:

```
main()
{
    auto int n;
    _____
    _____
}
```

In the above example, the variable n is declared as integer type and auto. The keyword auto is not mandatory because the default storage class in C is auto.

Example: Program to illustrate automatic storage class.

```
#include<stdio.h>
main()
{
    auto int m=1000;
    function2();
    printf("%d\n",m);
}
function1()
{
```

```

        auto int m=10;
        printf(" %d\n",m);
    }
    function2()
    {
        auto int m=100;
        function1();
        printf("%d\n",m);
    }
}

```

Output: 10 100 1000

Example: #include <stdio.h>

```

main( )
{
    auto int i = 1;
    {
        auto int i = 2;
        {
            auto int i = 3;
            printf ( "\n%d ", i);
        }
        printf ( "%d ", i);
    }
    printf( "%d\n", i);
}

```

Output: 3 2 1

One important feature of automatic variables is that their value cannot be changed accidentally by what happens in some other functions in the program. This assures that we may declare and use the same name for variable name in different functions in the same program without causing any confusion to the compiler.

External Storage Class: When the variables are declared as an external storage class variable, it is stored in memory. External variable's default value is zero. The scope of the variable is global and life of the variable is throughout the entire program. To define a variable as external storage class, the keyword `extern` is used, and variables are declared outside a function. These variables are also called as global variable.

Syntax: `extern int x;`

Example: Program to illustrate the properties of global variables.

```

#include<stdio.h>
extern int x;
main()
{
    x=25;
    printf("x=%d \n ",x);
    printf("x=%d \n",function1());
    printf("x= %d \n",function2());
    printf("x=%d \n", function3());
}

```

```

    }
    function1()
    {
        x=x+10;
        return(x);
    }
    function2()
    {
        int x; x=10;
        return(x);
    }
    function3()
    {
        x=x+10;
        return(x);
    }

```

Output: x=25 x=35 x=10 x=45

In the above program x is declare and we can use anywhere in the program.

Register Storage class: When a variable is declared as register storage class, it is stored in the CPU registers. The default value of the variable will be garbage value.scope of the variable is within the block where it is defined and life of the variable is until the control remains within the block.To define a variable as register storage class, the keyword register is used.

Syntax: **register int i;**

Scope of the register variable is within the block where it is defined and the life time of the variable is until the control remains within the block.

If CPU cannot store the variables in CPU registers, then the variables are assumed as auto and stored in the memory. Most compilers allow only int or char variables to be placed in the register.

Example: Program to declare a register variable

```

#include<stdio.h>
void main()
{
    register int i=1;
    do
    {
        printf("%d",i);
        i++;
    }while(j<=10);
}

```

Output: 1 2 3 4 5 6 7 8 9 10

Static Storage class: when a variable is declared as static storage class variable, it is stored in the memory. The default value of the static variable is zero. scope of the variable is within the block where it is defined and life of the variable persists between different function calls. To define a variable as

static storage class, the keyword `static` is used. A static variable can be initialized only once, it cannot be reinitialized.

Example: program to illustrate the properties of static variables

```
main()
{
    int i;
    for(i=1;i<=3;i++)
        fun();
}
fun()
{
    static int x=0;
    x=x+1;
    printf("This function is called %d times \n", x);
}
```


Output: This function is called 1 times
This function is called 2 times
This function is called 3 times

A static variable is initialized only once, when the program is compiled, it is never initialized again. During the first call to `fun()`, `x` is incremented to 1. Because `x` is static, this value persists and therefore, the next call adds another 1 to `x` giving it a value of 2. The value of `x` becomes 3 when the third call is made.

Storage Class	Keyword	Declaration	Memory Allocation	Default value	Scope	Lifetime
Automatic	<code>auto</code>	<code>auto int a;</code>	memory(RAM)	Garbage value	Within the Block	Within the block
Register	<code>register</code>	<code>register int a;</code>	CPU registers	Garbage value	Within The block	Within The block
Static	<code>static</code>	<code>static int a;</code>	memory(RAM)	0,0.0,NULL Depending on Data type	Within the block	Entire program
Extern	<code>extern</code>	<code>extern int a;</code>	memory(RAM)	0,0.0,NULL Depending on Data type	Entire program	Entire program

Preprocessor commands: The preprocessor is unique feature in c. The preprocessor is a program that processes the source code before it passes through the compiler. The following facilities are provided by the preprocessor

 Macro Substitution

 File inclusion

Conditional compilation




These facilities are controlled in a program by preprocessor control lines or directives. Generally preprocessor directives appear before the function `main()`. But they may appear anywhere in the program. Any preprocessor directive must begin with the character `#`. There is no semicolon to end any preprocessor.

Macro Substitution: macro substitution is a process where an identifier in a program is replaced by a predefined string composed of one (or) more tokens. The preprocessor accomplishes this task under the definition of `#define` statement. This statement, usually known as a macro definition.

Syntax: `#define identifier string`

In this statement is included in the program at the beginning, then the preprocessor replaces every occurrence of the identifier in the source code by the string.

We have three types of macro substitution facilities are possible

-  simple macro substitution
-  argumented macro substitution
-  nested macro substitution

Simple macro substitution: simple string replacement is commonly used to define constants.

Syntax: `#define constant_name constant`

Example:

```
#define PI 3.1428
#define COUNT 100
#define MAXROW 10
#define CAPITAL "DELHI"
```

A macro definition can include more than a simple constant value. It can include expression as well.

Syntax: `#define constant_name expression`

Example:

```
#define AREA 5*12.46
#include SIZE sizeof(int)*4
```

Example:

```
#define D 45-22
#define A 78+32
```

Input: `Ratio=D/A`

Output: `Ratio=(45-22)/(78+32)`

A macro definition can include literal text substitution. For example,

Syntax: `#define constant_name literal text`

Example:

```
#define TEST if(x>y)
#define AND
#define PRINT printf("very good");
```

Input: `TEST AND PRINT`

Output: The preprocessor would translate this line to
`If(x>y)printf("very good");`

Example: `#defineEQUALS ==`


```
#defineAND      &&
#defineOR       ||
#defineNOT_EQUAL!=
#defineSTART    main(){
#defineEND      }
#defineMOD      %
#defineBLANK_LINE printf("\n");
#defineINCREMENT ++
```

An example of the use of syntactic replacement is:

```
START
.....
.....
if(total EQUALS 240 AND average EQUALS 60)
INCRMENT count;
.....
.....
END;
```

argumented macro substitution: The preprocessor permits us to define more complex and more useful form of replacements.

Syntax: `#define macroname(arg1,arg2....argn) string`

Rules for defining macros:

- ✚ At least one white space between define and macro name, additional white spaces are discarded.
- ✚ No white space between macro name and the opening parenthesis (.
- ✚ Arguments are placed with in parenthesis separated by commas.
- ✚ At least one white space must be present between the closing parenthesis')' and string. Additional white spaces are discarded.
- ✚ Comments may be used in the place of white space.

Macro with argument is called macro call. When a macro is called, the preprocessor substitutes the string, replacing the formal parameters with the actual parameters

Example: `#define CUBE(x) (x*x*x)`

Suppose if the following statement appears in the program
`volume=CUBE(side);`

then the preprocessor would expand this statement to:

`Volume=(side*side*side);`

Example: `#define CUBE(x) ((x)*(x)*(x))`

Suppose if the following statement appears in the program
`volume=CUBE(side);`

Then the preprocessor would expand this statement to:

`Volume=((side)*(side)*(side));`

Examples: `#define MAX(a,b) (((a)>(b))?(a):(b))`
`#define MIN(a,b) (((a)<(b))?(a):(b))`
`#define STREQ(s1,s2) (strcmp((s1),(s2))==0)`

Nested macro substitution: we can also use one macro in the definition of another macro. That is ,macro definitions may nested.

Example: `#define M 5`
 `#define N M+1`
Example: `#define SQUARE(x) ((x)*(x))`
 `#define CUBE(x) (SQUARE(x)*(x))`
 `#define SIXTH(x) (CUBE(x)*CUBE(x))`

For example , `A=SIXTH(x)` It is expanded to

`A=(CUBE(x)*CUBE(x))`
`A=((SQUARE(x)*(x))*(SQUARE(x)*(x)))`
`A=(((x)*(x))*(x))*(((x)*(x))*(x))`

Undefining a macro: A defined macro can be undefined, using the `#undef` statement

Syntax: `#undef identifier`

Where , identifier=macro name

This is useful when we want to restrict the definition only to a particular part of the program.

Example: program for undef in c (This directive undefines existing macro in the program).

```
#include <stdio.h>
#define height 100
void main()
{
    printf("First defined value for height : %dn",height);
    #undef height      /* undefining variable*/
    #define height 600 /* redefining the same for new value*/
    printf("value of height after undef & redefine:%d",height);
}
```

Output: First defined value for height : 100
 value of height after undef & redefine : 600

File inclusion: An external file containing functions (or) macro definitions can be included as a part of a program so that we need not rewrite those functions(or) macro definitions. This is achieved by the preprocessor directive.

Syntax: `#include "filename"`
 (or)
 `#include<filename>`

Where filename is the name of the file containing the required definitions (or) functions. At this point, the preprocessor inserts the entire contents of filename into the source code of the program. When the filename included within the double quotation marks, the search for the file is made first in the current directory and then in the standard directories. When the filename is within the anchor(<>) tags the file is searched only in the standard directories.

Conditional compilation: A section of source code may be compiled conditionally using conditional compilation provided by the preprocessor. The following directives may be used in the conditional compilation.

 **#if**
 **#else**
 **#ifdef**
 **#elif**
 **#endif**
 **#ifndef**

Each **#ifdef** directive appears alone as a control line. These directives much like the if-else(or)if-else-if control structure.

#if directive is followed by optional else/elif directives. finally, if ends with **#endif** directive.

#if: **#if** is used to check whether the result of given expression is zero (or) not.If the result is not zero, then the statements after **#if** are compiled else not.

Syntax: **#if const_exp**
 statements;
 #endif

Example: **#include <stdio.h>**
 #define a 100
 main()
 {
 #if (a==100)
 printf("This line will be added in this C file since ""a = 100n");
 #endif
 }

#if,#else: **#if** ,**#else** is used to check whether the result of given expression is zero (or) not.If the result is not zero, then the statements after **#if** are compiled else the statements after **#else** are compiled.

Syntax: **#if const_exp**
 statements;
 #else
 statements;
 #endif

Example: program for #if, #else and #endif in c

```
#include <stdio.h>
#define a 100
main()
{
    #if (a==100)
```

```

        printf("This line will be added in this C file since ""a = 100n");
    #else
        printf("This line will be added in this C file since ""a is not equal to
        100n");
    #endif
}

```

Output: This line will be added in this C file since a = 100

#elif: The #elif enables us to establish an if..else..if sequence for testing multiple conditions.

Syntax:

```

#if const_exp
    Statements1;
#elif const_exp
    Statemetns2;
#else
    Statements3;
#endif

```

Example:

```

#include<stdio.h>
#define NUM 10
void main()
{
    #if(NUM == 0)
        printf("\nNumber is Zero");
    #elif(NUM > 0)
        printf("\nNumber is Positive");
    #else
        printf("\nNumber is Negative");
    #endif
}

```

#ifdef: “#ifdef” directive checks whether particular macro is defined or not. If it is defined, “If” clause statements are included in source file. Otherwise, “else” clause statements are included in source file for compilation and execution.

Syntax: **#ifdef identifier**

Example: program for #ifdef, #else and #endif in c:

```

#include <stdio.h>
#define RAJU 100
main()
{
    #ifdef RAJU
        printf("RAJU is defined. So, this line will be added in ""this C file");
    #else
        printf("RAJU is not definedn");
    #endif
}

```

Output: RAJU is defined. So, this line will be added in this C file

#ifndef: #ifndef exactly acts as reverse as #ifdef directive. If particular macro is not defined, “If” clause statements are included in source file. Otherwise, else clause statements are included in source file for compilation and execution.

Syntax: **#ifndef identifier**

Example: program for #ifndef and #endif in c

```
#include <stdio.h>
#define RAJU 100
main()
{
    #ifndef SELVA
    {
        printf("SELVA is not defined. So now we are going to ""define here");
        #define SELVA 300
    }
    #else
    printf("SELVA is already defined in the program");
    #endif
}
```

Output: SELVA is not defined. So, now we are going to define here

Arrays to Functions: like the values of simple variables, it is also possible to pass the values of an array to a function.

Passing 1d-array as an argument to a function: To pass “one-dimensional array” to a called function, we need to follow three rules

1. The function must be called by passing only the name of the array and the size of the array.

Syntax: **max(a,n);**

2. In the function definition, the formal parameter must be an array type; the size of the array does not need to specified.
3. The function header might look like:

Syntax: **float max(float array [], int size)**

The function max is defined to take two arguments, the array name and the size of the array to specify the number of elements in the array.

4. The declaration of the formal argument array is made as follows:

Syntax: **float array [];**

In the above statement the pair of brackets informs the compiler that the argument array is an array of numbers. It is not necessary to specify the size of the array here.

Example: c program which demonstrates , how to pass two dimensional array as an argument to a function.

```
#include<stdio.h>
#define N 10
```

```

main ()
{
    int sum (int a[ ], int);
    int a[N] = { 10,20,30,40,50,15,25,35,45,55};
    printf ("%d\n", sum(a, N) );
}
int sum(int a[],int n)
{
    int total=0,i;
    for(i=0;i<n;i++)
    {
        total=total+a[i];
    }
    return total;
}

```

Example: c program for finding maximum value in a given array, which demonstrates, how to pass two dimensional array as an argument to a function.

```

#include<stdio.h>
main( )
{
    float largest (float a[ ], int n);
    float value [4] = { 2.5, -4.75, 1.2, 3.67}
    printf ("%f\n", largest (value,4));
}
float largest (float a[ ], int n)
{
    int i;
    float max;
    max = a[0];
    for (i=1; i<n; i++)
    {
        if (max <a[i])
        {
            max = a[i];
        }
    }
    return (max);
}

```

Example: c program to sort the array elements in ascending order, which demonstrates , how to pass two dimensional array as an argument to a function.

```

#include<stdio.h>
void sort(int m,int x[]);
main()
{
    int i;
    int marks[5]={ 40,90,73,81,35};
    printf("marks before sorting");
    for(i=0;i<5;i++)
    {
        printf("%d",marks[i]);
    }
}

```

```

    }
    printf("\n\n");
    sort(5,marks);
    printf("marks after sorting");
    for(i=0;i<5;i++)
    {
        printf("%4d",marks[i]);
    }
    printf("\n");
}
void sort(int m,int x[])
{
    int i,j,t;
    for(i=1;i<=m-1;i++)
    {
        for(j=1;j<=m-i;j++)
        {
            if(x[j-1]>=x[j])
            {
                t=x[j-1];
                x[j-1]=x[j];
                x[j]=t;
            }
        }
    }
}

```

Output:

```

marks before sorting
40   90   73   81   35
Marks after sorting
35   40   73   81   90

```

Passing 2d-array as an argument to a function: we can also pass multi-dimensional arrays to functions. The approach is similar to the one we did with one dimensional arrays. the rules are simple

1. The function must be called by passing only the array name

Syntax: **function_name(c,n);**

2. In the function definition, we must indicate that the array has two dimensions by including two sets of brackets.
3. The size of the second dimension must be specified.

Syntax: **void function_name(int c[][2],int n)**

4. The prototype declaration should be similar to the function header.

Example: **double average(int x[][n],int m,int n)**
{

```

    int i,j;
    double sum=0.0;
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)

```

```

        {
            Sum=sum+x[i][j];
        }
    }
    return (sum/(m*n));
}

```

Example: c program which demonstrates , how to pass two dimensional array as an argument to a function.

```

#include <stdio.h>
void Function(int c[][2],int n);
main()
{
    int c[5][2],i,j,n;
    printf("enter row size");
    scanf("%d",&n);
    printf("Enter 4 numbers:\n");
    for(i=0;i<n;++i)
    {
        for(j=0;j<2;++j)
        {
            scanf("%d",&c[i][j]);
        }
    }
    Function(c,n);      /* passing multi-dimensional array to function */
}
void Function(int c[][2],int n)
{
    int i,j;
    printf("Displaying:\n");
    for(i=0;i<n;++i)
    {
        for(j=0;j<2;++j)
        {
            printf("%d\n",c[i][j]);
        }
    }
}

```

Example: c program which demonstrates , how to pass two dimensional array as an argument to a function.

```

#include <stdio.h>
void Function(int c[2][2]);
main()
{
    int c[2][2],i,j;
    printf("Enter 4 numbers:\n");
    for(i=0;i<2;++i)
    {
        for(j=0;j<2;++j)
        {
            scanf("%d",&c[i][j]);
        }
    }
}

```



```

    }
}
Function(c);
}
void Function(int c[2][2])
{
    int i,j;
    printf("Displaying:\n");
    for(i=0;i<2;++i)
    {
        for(j=0;j<2;++j)
        {
            printf("%d\n",c[i][j]);
        }
    }
}

```

Output: Enter 4 numbers: 2 3
 4 5
 Displaying: 2 3
 4 5

Example: Write a C program to add two matrices using functions.

```

#include <stdio.h>
int rows, columns;
void matrixAddition(int mat1[][10],int mat2[][10],int mat3[][10]);
main()
{
    int matrix1[10][10], matrix2[10][10];
    int matrix3[10][10], i, j;
    printf("Enter the no of rows and columns(<=10):");
    scanf("%d%d", &rows, &columns);
    printf("Enter the input for first matrix:");
    for (i = 0; i < rows; i++)
    {
        for (j = 0; j < columns; j++)
        {
            scanf("%d", &matrix1[i][j]);
        }
    }
    printf("Enter the input for second matrix:");
    for (i = 0; i < rows; i++)
    {
        for (j = 0; j < columns; j++)
        {
            scanf("%d", &matrix2[i][j]);
        }
    }
    matrixAddition(matrix1, matrix2, matrix3);
    printf("\nResult of Matrix Addition:\n");
    for (i = 0; i < rows; i++)
    {

```

```

        for (j = 0; j < columns; j++)
        {
            printf("%5d", matrix3[i][j]);
        }
        printf("\n");
    }
}

void matrixAddition(int mat1[][10], int mat2[][10], int mat3[][10])
{
    int i, j;
    for (i = 0; i < rows; i++)
    {
        for (j = 0; j < columns; j++)
        {
            mat3[i][j] = mat1[i][j] + mat2[i][j];
        }
    }
}

```

Output:

Enter the no of rows and columns:	3	3	
Enter the input for first matrix:	10	20	30
	40	54	60
	70	80	90
Enter the input for second matrix:	100	110	120
	130	140	150
	160	170	180
Result of Matrix Addition:	110	130	150
	170	194	210
	230	250	270

Example: Write a C program to multiply two matrices using functions.

```

#include <stdio.h>
void takedata(int a[][10], int b[][10], int r1,int c1, int r2, int c2);
void multiplication(int a[][10],int b[][10],int mult[][10],int r1,int c1,int r2,int c2);
void display(int mult[][10], int r1, int c2);
main()
{
    int a[10][10], b[10][10], mult[10][10], r1, c1, r2, c2, i, j, k;
    clrscr();
    printf("Enter rows and column for first matrix: ");
    scanf("%d%d", &r1, &c1);
    printf("Enter rows and column for second matrix: ");
    scanf("%d%d",&r2, &c2);
    if(c1!=r2)
    {
        printf("matrix multiplication is not possible");
    }
    else
    {
        takedata(a,b,r1,c1,r2,c2);
        multiplication(a,b,mult,r1,c1,r2,c2);
        display(mult,r1,c2);
    }
}

```

```

    }
    getch();
}
void takedata(int a[][10],int b[][10],int r1,int c1,int r2, int c2)
{
    int i,j;
    printf("\nEnter elements of matrix 1:\n");
    for(i=0; i<r1; ++i)
    {
        for(j=0; j<c1; ++j)
        {
            scanf("%d",&a[i][j]);
        }
    }
    printf("\nEnter elements of matrix 2:\n");
    for(i=0; i<r2; ++i)
    {
        for(j=0; j<c2; ++j)
        {
            scanf("%d",&b[i][j]);
        }
    }
}
void multiplication(int a[][10],int b[][10],int mult[][10],int r1,int c1,int r2,int c2)
{
    int i,j,k;
    for(i=0; i<r1; ++i)
    {
        for(j=0; j<c2; ++j)
        {
            mult[i][j]=0;
            for(k=0; k<c1; ++k)
            {
                mult[i][j]+=a[i][k]*b[k][j];
            }
        }
    }
}
void display(int mult[][10], int r1, int c2)
{
    int i, j;
    printf("\nOutput Matrix:\n");
    for(i=0; i<r1; ++i)
    {
        for(j=0; j<c2; ++j)
        {
            printf("%d ",mult[i][j]);
        }
        printf("\n\n");
    }
}

```

```

}
Output: Enter rows and column for first matrix: 2 3
Enter rows and column for second matrix: 3 2
Enter elements of matrix 1: 2 2 2
                          2 2 2
Enter elements of matrix 2: 2 2
                          2 2
                          2 2
Output Matrix: 12 12
              12 12

```

Strings and functions: The strings are treated as character arrays in c and therefore the rules for passing strings to functions are very similar to passing arrays to functions. The rules are

- 1) The string to be passed must be declared as a formal argument of the function when it is defined.

Syntax: **void display (char str[])**
{

}

- 2) The function declaration must show that the argument is a string, for the above function definition the declaration is written as

Syntax: **void display (char str[]);**

- 3) A call to the function must have a string array name without subscripts as its actual argument.

Example: **display (str);**

Example: Write a C program to Delete a string in to given main string from a Given position.

```

#include<stdio.h>
#include<conio.h>
#include<string.h>
void string_deletion(char[100]);
main()
{
    char ms[100];
    int i;
    clrscr();
    printf("Enter Main String:");
    scanf("%s",ms);
    printf("Main string and its position:\n");
    for(i=0;ms[i]!='\0';i++)
        printf("%c ",ms[i]);
    printf("\n");
    for(i=0;ms[i]!='\0';i++)
        printf("%d ",i);
    string_deletion(ms);
    getch();
}
void string_deletion(char ms[100])

```

```

{
    int position,s1,i,n;
    printf("\n Enter number of charaters your want to remove:");
    scanf("%d",&n);
    printf("Enter the position:");
    scanf("%d",&position);
    s1=strlen(ms);
    if(position<0||position>s1)
        printf("\nInvalid position.....!");
    else
    {
        for(i=position+n; i<=s1;i++)
            ms[i-n]=ms[i];
        printf("After Deletion %s",ms);
    }
}

```

Output: Enter Main String:
MAHESHBABU
Main string and its position:
M A H E S H B A B U
0 1 2 3 4 5 6 7 8 9
Enter number of characters you want to remove: 4
Enter the position:6
After Deletion MAHESH

Example: write a c program To insert a sub-string in to a given main string from the specified position.

```

#include<stdio.h>
#include<conio.h>
#include<string.h>
void insert(char [], char [], char [], int);
main()
{
    char a[30],b[20],c[20];
    int n,pos;
    clrscr();
    printf(" enter string\n");
    gets(a);
    printf("enter sub string to insert\n");
    gets(b);
    printf("enter insertion position\n");
    scanf("%d",&pos);
    insert(a,b,c,pos);
    getch();
}

void insert( char a[],char b[], char c[], int p)
{
    int i,j,k;
    for(i=0;i<p;i++)
        c[i]=a[i];
    for(j=0;b[j]!='\0';j++)

```

```
        c[i+j]=b[j];  
    for(k=p;a[k]!='\0';k++)  
        c[i+j++]=a[k];  
    c[i+j]='\0';  
    puts(c);  
}
```

POINTERS

When variables are declared memory is allocated to each variable. C provides data manipulation with addresses of variables therefore execution time is reduced. Such concept is possible with special data type called pointer. The basic data types in c language are int, float, char, double and void and so on. Pointer is a special data type which is derived from these basic datatypes. so pointers are called derived data type. Pointer is a data object that refers to a memory location, which is an address. Thus, a pointer variable may contain address of another variable (or) any valid address in the computer memory.

Pointer: A Variable which holds address of another variable is called a pointer variable. In other words, pointer variable which holds pointer value (i.e., address of a variable).

Features of pointers:

- ✚ Pointers save the memory space.
- ✚ Execution time with pointers is faster because data is manipulated with the addresses ,that is direct access to memory location.
- ✚ The memory is accessed efficiently with the pointers. the pointer assign the memory space and it also releases .dynamically memory is allocated
- ✚ Pointers are useful for representing two and multi-dimensional arrays

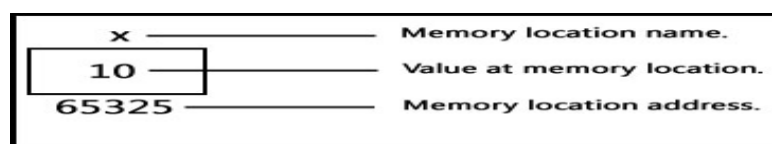
Consider the Declaration:

Syntax: **int x=3;**

The above declaration tells the compiler to

- ✚ Reserve the space in memory to hold the integer value
- ✚ Associate the name x with this memory location
- ✚ Store the value 3 at this location.

We may represent x's location in the memory by the following memory map



Accessing the address of a variable: we can access the address of a variable by using &.The & immediately preceding a variable associated with it.

Syntax: **p=&x;**

Example: We can print the address through the following program

```
#include<stdio.h>
main()
{
    int x=3;
    Printf("\n address of x is %u",&x);
}
```

```
Printf("\n value of I is %d",x);
```

```
}
```

Output: address of x is 64825
value of x is 3

The '&' operator used in the first printf statement, which is called **address of** operator. The '&x' returns the address of variable x.

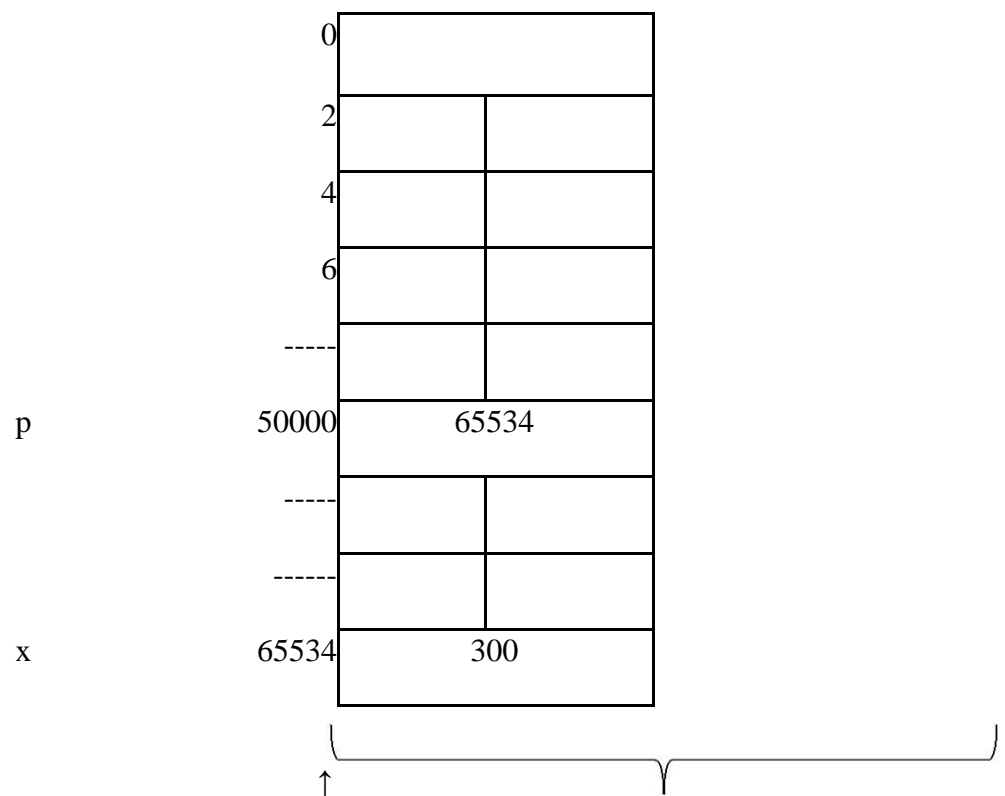
For example, if x is a variable and address of x is stored in a variable p as shown below.

```
p=&x;
```

Here the variable p is called a pointer variable. The memory organization after executing the above statement is shown below

Physical representation:

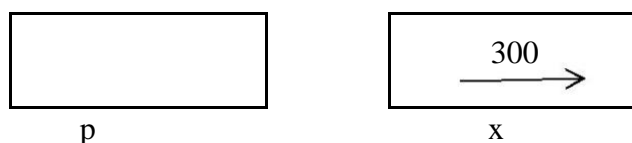
Variable Address Memory Locations



Physical representation

The address of a variable x which is 65534 is assigned to variable p. so, even though address of p is 50,000 (pointer value), the value stored in that location is 65534.

Since p is a variable which contains address of a variable x, so the variable p is called pointer variable.

Logical representation:

In the logical representation, the variable p points to the variable x. So, we say that the variable p points to x and hence the name pointer.

If we use the statement: `p = &x;`

The variable p contains the address of variable x.

A Pointer variable should contain only the address (i.e, pointer value)

The difference between pointer value and value stored in memory location is pointer value is 50,000 and value stored in memory location is 65534.

pointer declaration:

In a C language all the variables should be declared before they are used. Pointer variables also should be declared before they are used. The syntax to declare a pointer variable is

Syntax: *type * identifier*

Here, **identifier** –it is name given to the pointer variable.

* -tells that the identifier is a pointer variable

Type- it can be any data type such as int, float, char etc., it can be derived (user defined data type also)

Example: `int *p;`

The above declaration can be read as “p is a pointer to integer variable” that is p is a pointer variable and can hold the address of another variable of type int. In the declaration the position of * is immaterial, For example the following declarations are same;

Example: `int *pa;`
`int * pa;`
`int* pa;`

consider the multiple declarations

Example: `int* pa, pb, pc;`

Only pa is a pointer variable, whereas the variable pb and pc are ordinary integer variables. For better readability, the above declaration can be written as shown below:

```

int *pa;
int pb;
int pc;
  
```

Initializing a pointer variable: Initialization of a pointer variable is the process of assigning the

address of a variable to a pointer variable. The initialization of a pointer variable can be done using following three steps.

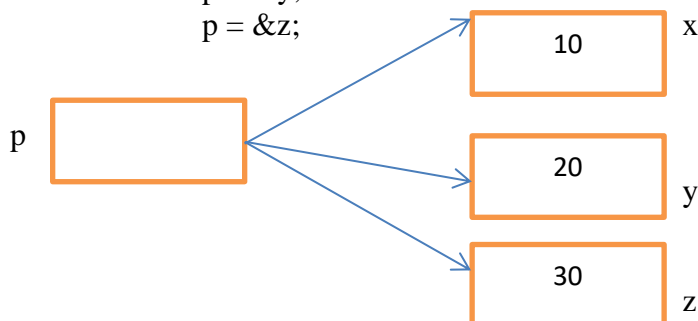
1. Declare a data variable
2. Declare a pointer variable
3. Assign address of a data variable to a pointer variable using & operator and assignment operator.

Example: `int x;` (1) step
 `int *p;` (2) step
 `p = &x;` (3) step

Here, the variable x is declared as integer variable. Since p is a pointer variable of type integer, it should contain address of integer variable. So, using the below statement.

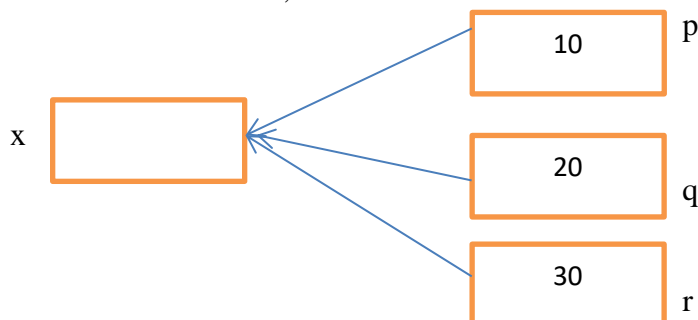
`p = &x`

Example: `int x = 10, y = 20, z = 30;`
 `int *p;`
 `p = &x;`
 `p = &y;`
 `p = &z;`



Here, the pointer variable p points to different memory locations by storing the address of different variables. Thus, the same pointer can be pointed to different data variables.

Example: `int x=10;`
 `Int *p,*q,*r;`
 `P=&x;`
 `Q=&x;`
 `R=&x;`



accessing variables through its pointers: The following steps to be followed while accessing the

variables using pointers.

🚦	Declare a data variable	<code>int a,n;</code>
🚦	Declare a pointer variable	<code>int *p;</code>
🚦	Initialize a pointer variable	<code>p=&a;</code>

How to access the value of variable using pointer: once a pointer has been assigned the address of a variable, the question remains as to how to access the value of the variable using the pointer. This can be done by using pointer variable and unary operator *. This operator is called indirection (or) value at address (or) dereferencing operator.

`n=*p;`

Example: `int x=100,y;`

`int *p;`

`P=&x; /*address of x is stored in pointer variable p */`

`Y=*p; /*p has address of x, the value of variable x is, i.e 100 is copied into y*/`

Note: By specifying *p, the value of the variable whose address is stored in p can be accessed. Here p should be a pointer variable.

Step 1:

Declaration	Variables	Address	
<code>int *p;</code>	p	5000	
<code>int *q;</code>	q	5002	
<code>int *r;</code>	r	5004	
<code>int x = 10;</code>	x	5006	

After declaration, the pointer variables p, q & r does not contain valid addresses and hence are called **dangling pointers**.

Step 2:

Initialization	Variables	Address	
<code>p = &x;</code>	p	5000	
<code>q = &x;</code>	q	5002	
<code>r = &x;</code>	r	5004	
	x	5006	

After executing these statements, the pointer variables p, q and r contains the address of integer variable x.

Step 3: Accessing the item 10:

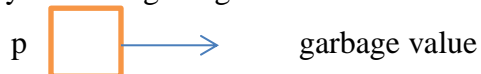
```
printf("&p = %u, p = %u, *p = %d \n", &p, p, *p);
printf("&q = %u, q = %u, *q = %d\n", &q, q, *q);
printf("&r = %u, r = %u, *r = %d\n", &r, r, *r);
```

output: &p = 5000, p = 5006, *p = 10
 &q = 5002, q = 5006, *q = 10
 &r = 5004, r = 5006, *r = 10

Dangling Pointer: A pointer variable which does not contain a valid address is called dangling pointer. Consider the following declaration.

```
int *p;
```

This indicates that p is a pointer variable and the corresponding memory location should contain address of an integer variable. But the declaration will not initialize the memory location and memory contains garbage value as shown below:

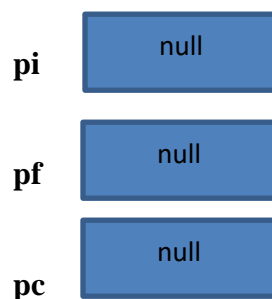


Here, the pointer variable p does not contain a valid address and we say that it is a dangling pointer.

Consider the following declarations assume that all are global variables

```
int *pi;
float *pf;
char *pc;
```

All global variables are initialized by the compiler during compilation. The pointer variables are initialized to null indicating that they do not point to any memory locations as shown below.

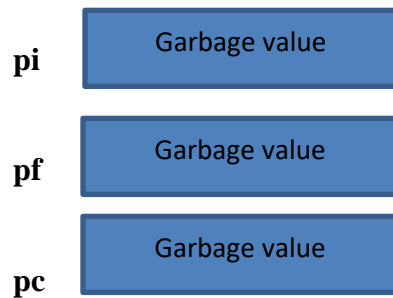


Consider the following declarations assume that all are local variables

```
int *pi;
float *pf;
char *pc;
```

The global variables are not initialized by the compiler during compilation. This is because; the local variables are created and used only during runtime. The pointer variables are also will not be initialized

hence the memory contains some garbage values



The data can be accessed using pointers after declaration and initialization as follows:

Example:

```
#include<stdio.h>
main()
{
    int i=3;
    int *j;
    j=&i;
    printf("address of I is %d",&i);
    printf("address of I is %d",j);
    printf("address of j is %d",&j);
    printf("value of j is %d",j);
    printf("value of I is %d",i);
    printf("value of I is %d",*(&i));
    printf("value of I is %d",*j);
}
```

The concept of pointer can be further extended. We know that pointer is a variable which holds the address of another variable. Now this variable itself contains the address of another pointer. Thus we can have a pointer, which contains another pointers address.

Example:

```
#include<stdio.h>
main()
{
    int i=3;
    int *j;
    int **k;
    j=&i;
    k=&j;
    printf("address of I is %d",&i);
    printf("address of I is %d",j);
    printf("address of j is %d",*k);
    printf("value of j is %d",&j);
    printf("value of j is %d",k);
    printf("value of k is %d",&k);
    printf("value of I is %d",i);
    printf("value of I is %d",*(&i));
    printf("value of I is %d",*j);
    printf("value of I is %d",**k);
    printf("value of j is %d",j);
}
```

```
        printf("value of k is %d",k);
    }
```

Null Pointer: A Null pointer is defined as a special pointer value that points to nowhere in the memory. If it is too early in the code to assign a value to the pointer, then it is better to assign NULL (i.e, \0 or 0) to the pointer.

Example: #include <stdio.h>
 int *p = NULL;

Here the pointer variable p is a NULL pointer. This indicates that the pointer variable p does not point to any part of the memory. The value for NULL is defined in the header file "stdio.h".

The programmer can access the data using the pointer variable p if and only it does not contain NULL. The error condition can be checked using the following statement:

```
    if ( p == NULL)
        printf(", p does not point to any memory");
    else
        printf("Access the value of p \n");
```

Example: int *x; int y;
 x = y // error.

The value of data variable cannot be assigned to a pointer variable. So the **statement x = y; results an error**. The **correct statement is x = &y;**

Example: Write a program to read two numbers and then add two numbers using pointers.

```
#include <stdio.h>
void main ( )
{
    int a , b, sum; int *p, *q;
    p = &a;
    q = &b;
    scanf ("%d %d", &a, &b);
    sum = *p + *q;
    printf (",sum = %d\n", sum);
}
```

Input: 10 20

Output: 30

Pointers and expressions: pointer variables can be used for accessing values in arithmetic expressions.

Example: (*p1)*(*p2)
 Sum=sum+(*p1);

Arithmetic operations on pointer variables: Arithmetic operations on pointer variables are also possible.

Increment and decrement of pointers: It is possible to increment and decrement pointer variables using ++ and – operators respectively.

A pointer expression may be preceded by and followed by ++ (or) --.in such cases the operators &,*and,++(or)—are involved

While evaluating these expressions, it is very important to remember the operator precedence and associativity of the operators. The operators &,*and,++ and,-- have same precedence level and the evaluation is carried out from right to left.

Example: int *p,x=20;

 P=&x;

 ++ *p;

 *p ++;

Example: #include<stdio.h>

 main()

 {

 int *p,*q,*r;

 int a=10,b=20,c=30;

 p=&a;

 q=&b;

 r=&c;

 printf(“%u%u%u”,p,q,r);

 printf(“%d%d%d”,*p,*q,*r);

 printf(“%u%u%u”,++p,++q,++r);

 }

Output: 100 200 300

 10 20 30

 102 202 302

Scale factor length: when we increment a pointer its value is incremented by the length of the data type it is pointing is called scale factor.

Example: Assume p1 is an integer pointer with initial value of 2800, then after executing the p1++ statement, the value of p1 will be 2802, and not 2801.

Example: Assume p1 is a character pointer with initial value of 2800, then after executing the p1++ statement, the value of p1 will be 2801.

Example: Assume p1 is a float pointer with initial value of 2800, then after executing the p1++ statement, the value of p1 will be 2804.

Example: Assume p1 is a double pointer with initial value of 2800, then after executing the p1++ statement, the value of p1 will be 2808.

Rules for pointer operations:

- ✚ A pointer variable can be assigned to another pointer variable, if both are pointing to the same data type.
- ✚ A pointer variable can be assigned a NULL value.
- ✚ A pointer variable cannot be multiplied or divided by a constant.
Example: $p*3$ or $p/3$ where p is a pointer variable
- ✚ One pointer variable can be subtracted from another provided both point to elements on same array.
- ✚ C allows us to add integers to or subtract integers from pointers.
Example: $p1+4$, $p2-2$, $p1-p2$ If both $p1$, $p2$ are pointers to same array then $p2-p1$ gives the number of elements between $p1, p2$
- ✚ Pointer variable can be incremented or decremented $p++$ & $p--$

When a pointer to an integer is incremented by one and if 2 bytes are used for int, the address is incremented by 2. such scale factors necessary for the pointer arithmetic are taken care by the compiler.

Pointers and other operations: Operations performed on pointers is

- ✚ Adding an integer to a pointer.
- ✚ Subtracting an integer from pointer
- ✚ Subtracting two pointers
- ✚ Comparing two pointers

Adding an integer to a pointer: Here, one operand is a pointer and other operand is an integer.

Consider the following declaration:

```
int a[5] = { 10,20,30,40,50};
int *p1, *p2;
p1 = a;
p2 = a;
```

the various valid and invalid statements are

$p1 = p1 + 1;$	valid
$p1 = p1 + 3;$	valid
$p1 + p2;$	invalid: two pointers cannot be added
$p1++;$	valid

Example: Display array elements using pointers:

```
#include<stdio.h>
void main ( )
{
    int a [ ] = { 10,20,30,40,50};
    int *p;
    int i;
    p = a;
```



```

        for (i = 0; i<=4; i++)
        {
            printf("%d\n", *p);
            p++;
        }
    }

```

Example: Sum of n numbers using pointers

```

#include<stdio.h>
void main ( )
{
    int a [ ] = { 10,20,30,40,50}; int *p;
    int i, sum; p=a;
    sum = 0;
    for (i = 0; i<=4; i++)
    {
        sum = sum + *p;
        p++;
    }
    printf("sum = %d", sum);
}

```

Subtracting an integer from a pointer: Subtracting can be performed when first operand is a pointer and the second operand is an integer.

Example: int a[5] = { 10,20,30,40,50};
 int *p1;
 p1 = &a[4];

the various valid and invalid statements are

```

p1 = p1 - 1;    it is valid
p1 = p1 - 3;    it is valid
p1--;          it is valid
--p1;          it is valid
p1 = 1 - p1    it is invalid (illegal pointer subtraction)

```

Example: #include<stdio.h>
 main()
 {

```

            int a[5]={ 10,20,30,40,50};
            int *p;
            p=&a[4];
            for(i=0;i<5;i++)
            {
                printf("%d", *p);
                p--;
            }
        }

```

Output: 50 40 30 20 10

Subtracting two pointers: If two pointers are associated with the same array, then subtraction of two pointers is allowed. But, if the two pointers are associated with different arrays, even though subtraction of two pointers is allowed. The result is meaningless.

Example: `int a[5] = { 10,20,30,40,50};`
 `int *p1;`
 `int *p2;`
 `float *f;`
 `p1= a;`
 `p2= &a[4];`

The various valid and invalid statements are

<code>p2-p1;</code>	it is valid
<code>p1-p2;</code>	it is valid
<code>f-p1</code>	it is invalid since type of both operands is not same.

10	20	30	40	50
100	102	104	106	108

But, `p2-p1` is not `0108 - 0100`. Actually, it is $(0108-0100)/\text{sizeof}(\text{int})$ i.e; $(0108 - 0100)/2 = 4$.

Comparing two pointers: If two pointers are associated with the same array. Then comparison of two pointers is allowed using operators.

Example: `int a[5] = { 10,20,30,40,50};`
 `int *p1, *p2;`
 `float *f;`
 `p1= a;`
 `p2 = &a[4];`
 `p2!= p1;` ----- valid
 `p1== p2;` ----- valid
 `p1<= p2;` ----- valid
 `p1>= p2;` ----- valid
 `f!= p1;` -----invalid

Example: `#include<stdio.h>`
 `main()`
 `{`
 `int a=20,b=10;`
 `int *p, *q;`
 `p=&a;`
 `q=&b;`
 `printf("addition of two pointers is:%d",(*p)+(*q));`
 `printf("subtraction of two pointers is:%d",(*p)-(*q));`
 `printf("multiplication of two pointers is:%d",(*p)*(*q));`
 `printf("division of two pointers is:%d",(*p)/(*q));`
 `printf("mod of two pointers is:%d",(*p)%(*q));`
 `getch();`
 `}`

Pointers and functions:

Pointers as function arguments: Here, we can pass the addresses to a function, and then the parameters receiving the address should be pointers. The process of calling a function using pointers to pass the address of variables is known as call by reference (or) call by address (or) pass by pointers. The function which is called by reference can change the value of the variable used in the call.

Example:

```
#include<stdio.h>
main()
{
    int x;
    x=20;
    change(&x);
    printf("%d",x);
}
change(int *p)
{
    *p=*p+10;
}
```

Example:

```
#include<stdio.h>
void exchange(int *m, int *n);
void main()
{
    int a=10,b=20;
    printf("values before exchange a=%d,b=%d",a,b);
    exchange(&a,&b);
    printf("values after exchange a=%d,b=%d",a,b);
}
void exchange (int *m, int *n)
{
    int temp;
    temp=*m;
    *m=*n;
    *n=temp;
}
```


Pointers and arrays: when an array is declared, the compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations. The name of an array itself indicates the starting address of an array or address of first element of an array. That means array name is the pointer to starting address or first elements of the array. If x is an array then address of first element can be expressed as &x[0]. The compiler defines array name as a constant pointer to the first element.

Example: int x[5]={ 1,2,3,4,5};

In the above declaration, assume the base address of x is 1000 and assuming that each integer requires two bytes, the five elements will be stored as follows

Elements-----> x[0] x[1] x[2] x[3] x[4]

Value----->	1	2	3	4	5
Address----->	1000	1002	1004	1006	1008


 Base address

The name `x` is defined as a constant pointer pointing to the first element, `x[0]` and therefore the value of `x` is 1000, the location where `x[0]` is stored. That is

$$x = x[0] = 1000$$

if we declare `p` as an integer pointer, then we can make the pointer `p` to point to the array `x` by the following assignment:

```
p=x;
```

The above statement is equivalent to `p=&x[0]`.

Now, we can access every value of `x` using `p++` to move from one element to another. The relationship between `p` and `x` is:

```
p=&x[0](=1000)
```

```
p+1=&x[1](=1002)
```

```
p+2=&x[2](=1004)
```

```
p+3=&x[3](=1006)
```

```
p+4=&x[4](=1008)
```

Note that the address of an element is calculated using its index and the scale factor of its data type.

Example: address of `x[3]` = base address + (3 * scale factor of int)

$$= 1000 + (3 * 2)$$

$$= 1006$$

When handling arrays, instead of using array indexing, we can use pointers to access array elements.

Note that `*(p+3)` gives the value of `x[3]`, that is 1006.

The pointer access method is faster than array indexing.

Example:

```
#include<stdio.h>
#include< conio.h>
main( )
{
    int *p,i,sum;
    int x[5]={5,9,6,3,7};
    i=0;
    p=x;
    printf("Element      value      address");
    while(i<5)
    {
        printf("x[%d]      %d      %u",i,*p,p);
        sum=sum+ *p;
        i++;
    }
}
```

```

        p++;
    }
    printf("\n sum=%d\n",sum);
    printf("\n &x[0]=%u\n",&x[0]);
    printf("\np=%u\n",p);
}

```

Output:

element	value	address
X[0]	5	166
X[1]	9	168
X[2]	6	170
X[3]	3	172
X[4]	7	174

```

Sum=55
&x[0]=166
P=176

```

Example:

```

#include <stdio.h>
main( )
{
    int *p;
    int val[7] = { 11, 22, 33, 44, 55, 66, 77 } ;
    p = &val[0];
    for ( int i = 0 ; i <= 6 ; i++ )
    {
        printf("val[%d]: value is %d and address is %u", i, *(p+i), (p+i));
    }
}

```

Example:

```

#include <stdio.h>
main( )
{
    int a[5]={ 10,20,30,40,50};
    int i=3;
    printf("%d%d%d%d",*(a+i),a[i],*(a+i),*(i+a));
}

```

Output: 40 40 40 40

Example:

```

#include <stdio.h>
main( )
{
    int a[10],n,i,big;
    printf("enter no of elements");
    scanf("%d",&n);
    printf("enter array elements");
    scanf("%d",a);
    big=a[0];
    for(i=0;i<n;i++)
    {
        if(a[i]>big)
        {
            big=a[i];
        }
    }
}

```

```

    }
    }
    printf("largest no is%d:",big);
}
Example: #include <stdio.h>
main( )
{
    int a[10],n,i,sum;
    int *p=a;
    printf("enter no of elements");
    scanf("%d",&n);
    printf("enter array elements");
    for(i=0;i<n;i++)
    {
        scanf("%d",(p+i));
        p++;
    }
    for(i=0;i<n;i++)
    {
        sum=sum+*p;
        p++;
    }
    printf("sum of n no is is%d:",sum);
}

```

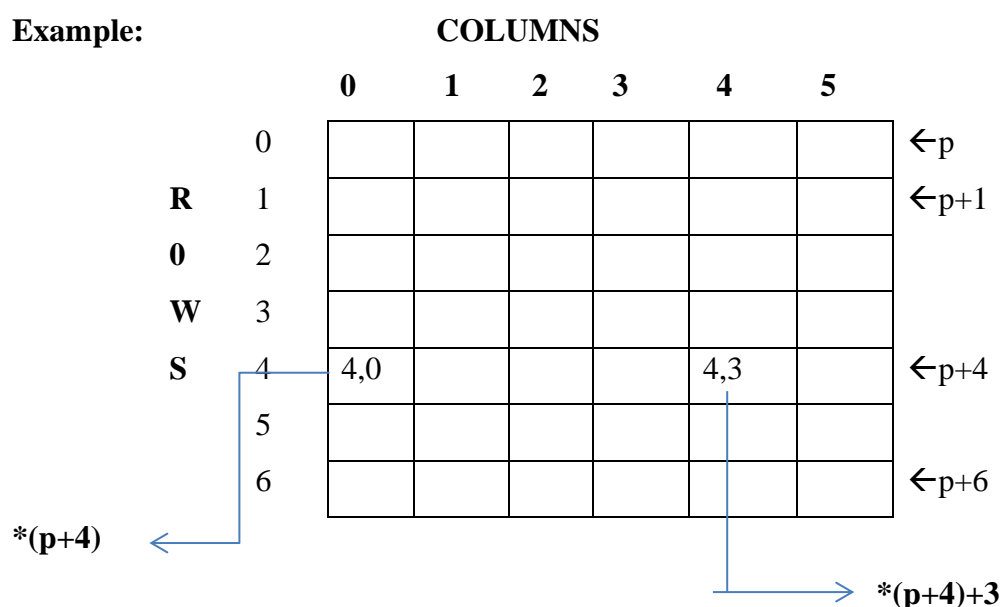
Pointers to two dimensional arrays: pointers can be used to manipulate two dimensional arrays as well. we know that in a one dimensional array x, the expression is

$*(x+i)$ or $*(p+i)$

The above declaration represents the element $x[i]$, similarly an element in 2D array can be represented by the pointer expression is

$*(*(a+i)+j)$ or $*(*(p+i)+j)$

Example:



Here, $p \rightarrow$ pointer to the first row

$p+i \rightarrow$ pointer to the i^{th} row

$*(p+i) \rightarrow$ pointer to 1st element in the i^{th} row

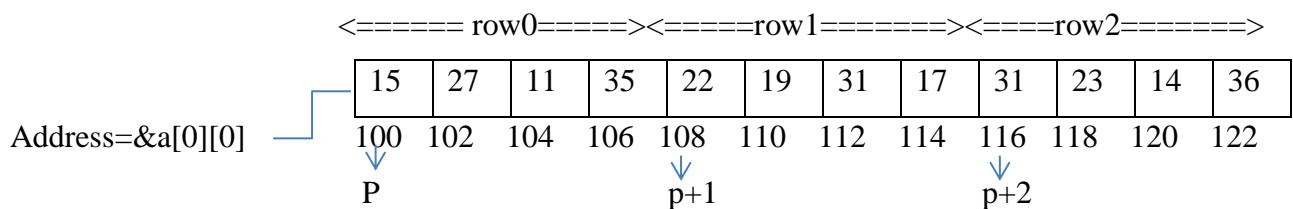
$*(p+i)+j \rightarrow$ pointer to j^{th} element in the i^{th} row

$*(*(p+i)+j) \rightarrow$ pointer to the i^{th} row j^{th} cell

The above example, base address of the array a is $\&a[0][0]$ and starting at this address. The compiler allocates contiguous space for all the elements row wise. That is the first element of the second row is placed immediately after the last element of the first row and so on.

Example: `int a[3][4]={ { 15,27,11,35}`
 `{ 22,19,31,17}`
 `{ 31,23,14,36}`
 `};`

The elements a will be stored as



✚ By using $\&p[i][j]$ or $*(p+i)+j$ we can obtain address of i^{th} row j^{th} column.

✚ By using $p[i][j]$ or $*(*(p+i)+j)$ we can obtain value i^{th} row and j^{th} column.

Example:

```
#include<stdio.h>
void main ( )
{
    int i,m,n, a[10][20], b[10][20];
    printf ("Enter the size of the matrix m & n: \n");
    scanf ("%d%d", &m, &n);
    printf ("enter the matrix a \n");
    read_matrix (a, m, n);
    printf ("enter the matrix b\n");
    read_matrix (b, m,n);
    printf (" the matrix is \n");
    write-matrix (c ,m, n);
}
void read_matrix(int *a[10], int m, int n)
{
    int i, j;
    for (i = 0; i<m; i++)
    {
        for (j=0; j<n; j++)
        {
```

```

scanf ("%d", (*(a+i) + j) );
    }
}
}
void write_ matrix(int *b[10], int m, int n)
{
    int i, j;
    for(i = 0; i<m; i++)
    {
        for (j=0; j<n; j++)
        {
            printf ("%d", (*(a+i) + j) );
        }
        printf ("\n");
    }
}

```

Example: Program to add two matrices using pointers.

```

#include<stdio.h>
void read_ matrix(int *a[10], int m, int n)
{
    int i, j;
    for (i = 0; i<m; i++)
    {
        for (j=0; j<n; j++)
        {
            scanf ("%d", (*(a+i) + j) );
        }
    }
}
void write_ matrix(int *b[10], int m, int n)
{
    int i, j;
    for(i = 0; i<m; i++)
    {
        for (j=0; j<n; j++)
        {
            printf ("%d", (*(a+i) + j) );
        }
        printf ("\n");
    }
}
void add_ matrix ( int * a[10], int *b[10], int *c[10], int m, int n)
{
    int i, j;
    for (i = 0; i<m; i++)
    {
        for (j=0; j<n; j++)
        {
            *( *( c+i ) + j) = (*(a+i) + j) + * ( *(b+i) + j);
        }
    }
}

```



```

    }
}
void main ( )
{
    int i,m,n, a[10][20], b[10][20], c[10][20];
    printf ("Enter the size of the matrix m & n: \n");
    scanf ("%d%d", &m, &n);
    printf ("enter the matrix a \n");
    read_matrix (a, m, n);
    printf ("enter the matrix b\n");
    read_matrix (b, m,n);
    add_matrix(c,m,n);
    printf(" the resultant matrix is \n");
    write-matrix (c ,m, n);
}

```

Pointers and strings: we have seen earlier that strings are treated like character arrays and therefore, they are declared and initialized as follows.

Example: char array[11] = "Love India";

The compiler will automatically inserts the null character ('\0') at the end of the string.

C language supports to create strings using pointer variables of type char

Example: char *p= "Love India";

Here the pointer p now points to the first character of the string love india as

L	O	V	E		I	N	D	I	A	\0
---	---	---	---	--	---	---	---	---	---	----

We can also use the runtime assignment for giving the values to a string pointer.

Example: char *p;
P="good";

We can print the content of string p using printf() and puts().

Example: printf("%s",p);
puts(p);

Remember p is a pointer to the string, it is also name of the string. Therefore, we do not need to use indirection operator * here.

We can also use pointers to access the individual characters in a string.

Example: #include<stdio.h>
#include<conio.h>
main()
{
 int i;
 char *p= " Love India";
 clrscr();

```

while (*p! = "\0")
{
    printf ("%c ", *p);
    p++;
}

```

Output: Love India

Array of pointers: one important advantage of pointers is in handling of a table of strings using array of pointers.

Example: char name[3][25];

The above syntax tells that the name is a table containing three names, each with maximum length of 25 characters (including null character). The total storage requirements for the name table are 75 bytes. We know that the individual strings will be of equal lengths. Therefore, instead of making each row a fixed number of characters, we can make it a pointer to a string of varying length.

Example: char *name[3]={ "new Zealand",
"Australia"
"india"
};

In the above example declares name to be an array of three pointers to characters, each pointer pointing to a particular name as:

Name[0]----->new Zealand

Name[1]----->india

Name[2]----->Australia

This declaration allocates 28 bytes, sufficient to hold all the characters.

N	E	W		Z	E	A	L	A	N	D	\0
A	U	S	T	R	A	L	I	A	\0		
I	N	D	I	A	\0						

The following statement would print out all the three names

```

for(i=0;i<=2;i++)
{
    printf("%s\n",name[i]);
}

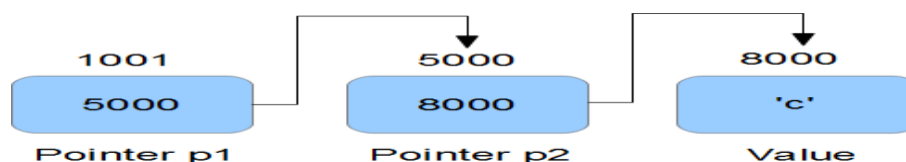
```

To access the jth character in the ith name, we may write it as

*(name[i]+j)

The character arrays with the rows of varying length are called ragged arrays

Pointers to pointers (or) chain of pointers: A pointer to point to another pointer is called pointer to a pointer (or) chain of pointers.



Here the pointer variable p1 contains the address of the pointer variable p2, which points to the location that contains the desired value. This is known as multiple indirections.

A variable that is a pointer to a pointer must be declared using additional indirection operator symbols in front of the name.

Syntax: `int **p1;`

In the above syntax tells the compiler that p1 is a pointer to a pointer of int type. Remember, the pointer p1 is not a pointer to an integer, but rather a pointer to an integer pointer.

Example:

```

#include<stdio.h>
main()
{
    int x,*p2,**p1;
    x=100;
    p2=&x;
    p1=&p2;
    printf("%d",**p1);
}
  
```

Example:

```

#include <stdio.h>
void main()

    int k = 5;
    int *p = &k;
    int **m = &p;
    printf("%d%d%d\n", k, *p, **m);
}
  
```

Output: 5 5 5

void pointer (or) generic pointer: a void pointer is a special type of pointer, that can be pointed to any data type. Hence, void pointer is also called as universal (or) generic pointer. A void pointer is declared like a normal pointer, using the void keyword as the pointer's type. void pointers are useful for pointing different data types.

Syntax: `void *identifier;`

Example: `void *p;`

Now the pointer variable p can contain address of data variable of any type such as int, float, char and double etc. The only limitation is that the pointed data cannot be referenced directly using indirection operator. This is because; its length is always undetermined.

Example:

```

#include<stdio.h>
main()
  
```

```

{
    int i;
    char ch;
    void *p;
    i=6;
    ch='a';
    p=&i;
    printf("the value of I is %d",*(int*)p);
    p=&ch;
    printf("the value of ch is %c",*(ch*)p);
}

```

Pointers to functions: a function, like a variable, has a type and an address location in the memory. so it is possible to declare a pointer to a function, which can be used as an argument in another function. A pointer to a function is declared as follows

Syntax: **datatype (*fptr)();**

This tells the compiler that fptr is a pointer to a function, which returns data type value.

We can make a function pointer to point to a specific function by simply assigning the name of the function to the pointer.

Example: double mul(int,int);
 double(*p)();
 p=mul;

Here declare p is pointer to a function and then make p to point the function mul. To call the function mul, we may now use the pointer p with list of parameters. That is

(*p)(x,y); /* function call */

The above statement is equivalent to

mul(x,y);

Example: # include<stdio.h> main()
 {
 int show ()
 int (*p) ();
 clrscr();
 p = show;
 p(); /* equals to (*p)(); */
 printf ("%u", show);
 getch();
 }
 show ()
 {
 printf("the address of function show is : ");
 }
 }

Output: the address of function show is 759

In this program the variable p is pointer to function. Address of function show () is assigned

to pointer p. Using function pointer, the function show () is invoked.

Example:

```
#include<stdio.h>
main()
{
    int(*fptr)(int,int);    /*Function pointer */
    fptr = func;            /* Assign address to function pointer */
    func(2,3);
    fptr(2,3); /* it is equals to (*fptr)(2,3); */
}

int func (int a, int b)
{
    printf("\n a = %d\n",a);
    printf("\n b = %d\n",b);
    return 0;
}
```

Output:

```
a=2
b=3
a=2
b=3
```

Functions returning pointers: we have seen so far that a function can return single value by using return statement (or) return multiple values through pointer parameters. since pointers are a data type in c, we can return a pointer to the calling function.

Syntax: **datatype *fptr();**

Where fptr is a function returning a pointer to data type.

Example:

```
#include<stdio.h>
int *larger(int *,int *);
main()
{
    int a=10;
    int b=20;
    int *p;
    p=larger(&a,&b);
    printf("%d",*p);
}

int *larger(int *x,int *y)
{
    if(*x>*y)
        return (x);    /* return address of a */
    else
        return (y);    /* return address of b */
}
```

Here, the function larger receives the addresses of the variables a and b, decides which one is larger using the pointers x and y and then returns the address of its location. The returned value is then assigned to the pointer variable p in the calling function. In this case, the address of b is returned and assigned to p and therefore the output will be the value of b, that is 20.

Advantages:

- ✚ More than one value can be returned using pointer concept.
- ✚ Very compact code can be written using pointers.
- ✚ Data accessing is much faster when compared to arrays.
- ✚ Using pointers, we can access byte (or) word locations and the CPU registers directly. The pointers in „C“ are mainly useful in processing of non – primitive data structures such as arrays, linked lists etc.,

Disadvantages:

- ✚ Un-initialized pointers or pointers containing invalid address can cause system crash.
- ✚ It is very easy to use pointers incorrectly, causing bugs that are very difficult to identify and correct.
- ✚ They are confusing and difficult to understand in the beginning and if they are misused. The result is not predictable.