Syllabus

Unit-I

Logic Development: Data Representation, Flowcharts, Problem Analysis, Decision Trees/Tables, Pseudo code and algorithms. Fundamentals: Character set, Identifiers and Key Words, Data types, Constants, Variables, Expressions, Statements, Symbolic Constants.

Operations and Expressions: Arithmetic operators, Unary operators, Relational Operators, Logical Operators, Assignment and Conditional Operators, Library functions.

Unit-II

Data Input and Output: formatted & unformatted input output.

Control Statements: While, Do–while and For statements, Nested loops, If–else, Switch, Break – Continue statements.

Unit-III

Functions: Brief overview, defining, accessing functions, passing arguments to function, specifying argument data types, function prototypes, recursion.

Arrays: Defining, processing arrays, passing arrays to a function, multi–dimensional arrays.

Strings: String declaration, string functions and string manipulation Program Structure Storage Class: Automatic, external and static variables.

Unit-IV

Structures & Unions: Defining and processing a structure, user defined data types, structures and pointers, passing structures to functions, unions.

Pointers: Understanding Pointers, Accessing the Address of a Variable, Declaration and Initialization of Pointer Variables, Accessing a Variable through its Pointer, Pointers and Arrays

File Handling: File Operations, Processing a Data File

INDEX

UNIT –I

❖ Data Representation

•Data refers to the symbols that represent people, events, things, and ideas. Data can be a name, a number, the colors in a photograph, or the notes in a musical composition.

•Data Representation refers to the form in which data is stored, processed, and transmitted.

Number systems are the technique to represent numbers in the computer system architecture, every value that you are saving or getting into/from computer memory has a defined number system.

Computer architecture supports following number systems.

- **Binary number system**
- **Octal number system**
- **Decimal number system**
- **Hexadecimal (hex) number system**

1) Binary Number System

A Binary number system has only two digits that are **0 and 1**. Every number (value) represents with 0 and 1 in this number system. The base of binary number system is 2, because it has only two digits.

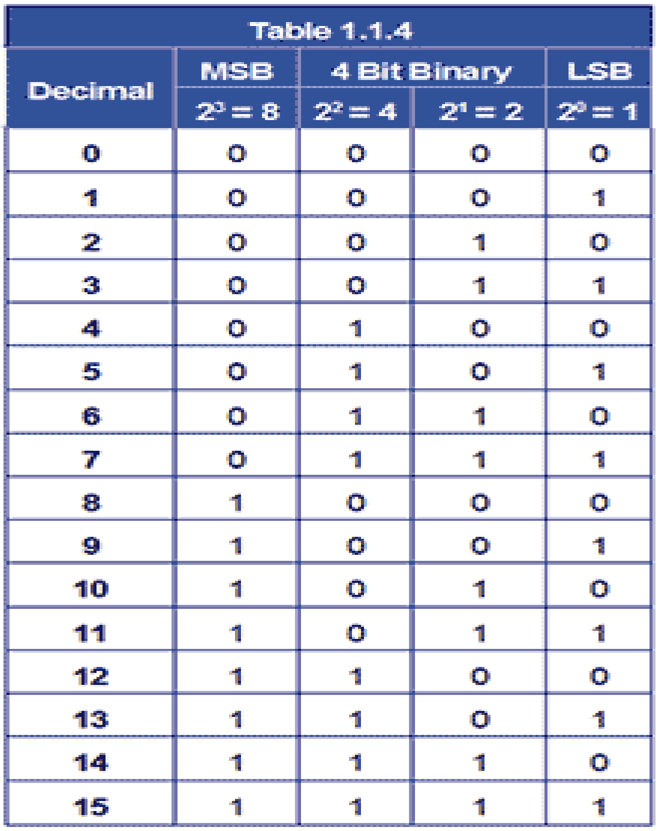
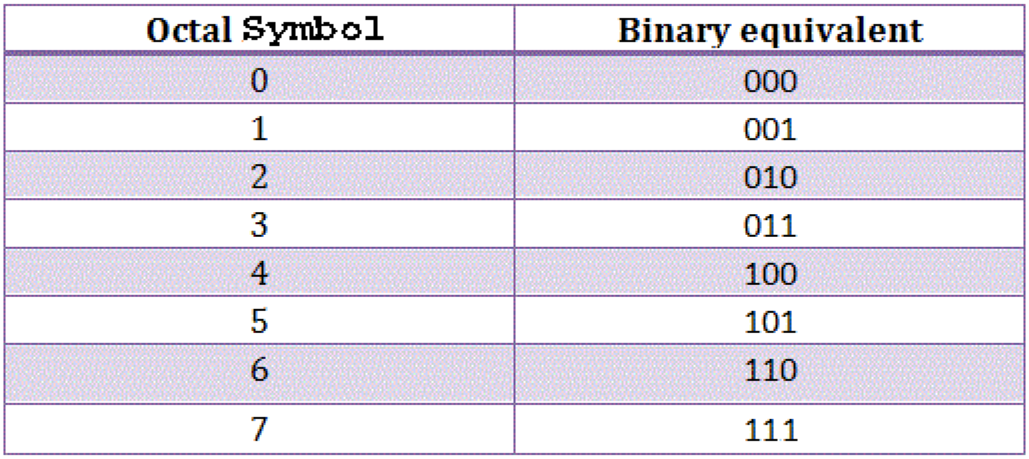


Table 1.1.4

Decimal	MSB	4 Bit Binary		LSB
	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1

2) Octal number system

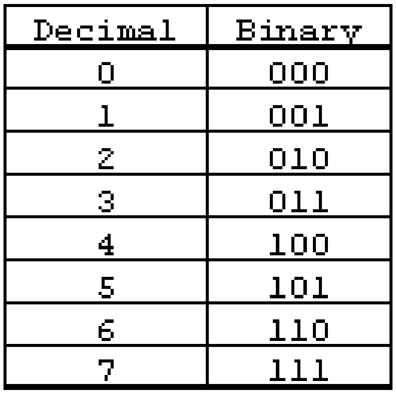
Octal number system has only eight (8) digits from **0 to 7**. Every number (value) represents with 0,1,2,3,4,5,6 and 7 in this number system. The base of octal number system is 8, because it has only 8 digits.



Octal Symbol	Binary equivalent
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

3) Decimal number system

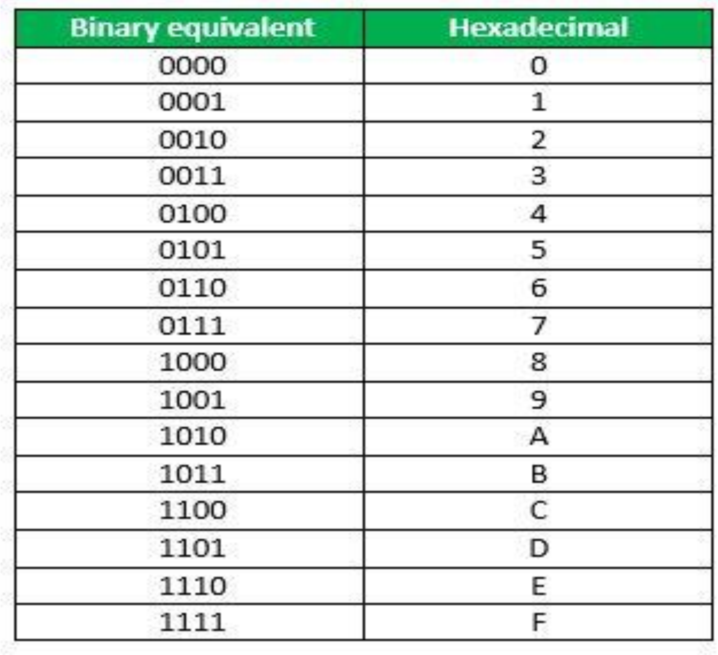
Decimal number system has only ten (10) digits from **0 to 9**. Every number (value) represents with 0,1,2,3,4,5,6, 7,8 and 9 in this number system. The base of decimal number system is 10, because it has only 10 digits.



Decimal	Binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

4) Hexadecimal number system

A Hexadecimal number system has sixteen (16) alphanumeric values from **0 to 9** and **A to F**. Every number (value) represents with 0,1,2,3,4,5,6, 7,8,9,A,B,C,D,E and F in this number system. The base of hexadecimal number system is 16, because it has 16 alphanumeric values. Here **A is 10**, **B is 11**, **C is 12**, **D is 13**, **E is 14** and **F is 15**.

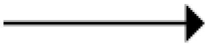
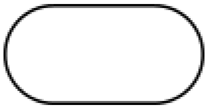


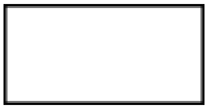
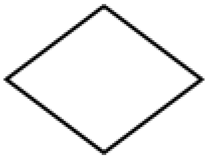
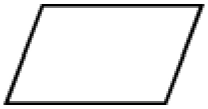
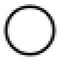
Binary equivalent	Hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

❖ Flowchart

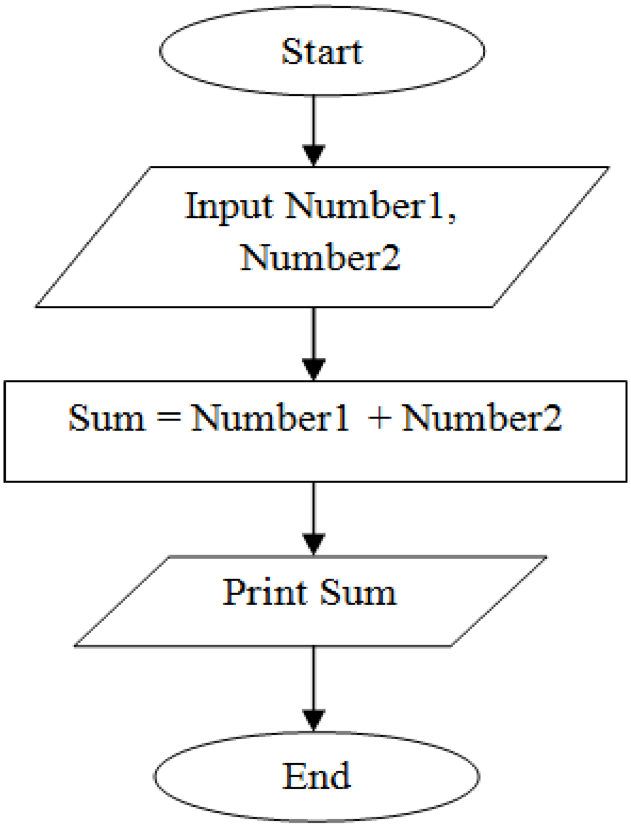
A **flowchart** is a type of diagram that represents a workflow or process. A flowchart can also be defined as a diagrammatic representation of an algorithm, a step-by-step approach to solving a task.

The flowchart shows the steps as boxes of various kinds, and their order by connecting the boxes with arrows. This diagrammatic representation illustrates a solution model to a given problem. Flowcharts are used in analyzing, designing, documenting or managing a process or program in various fields.

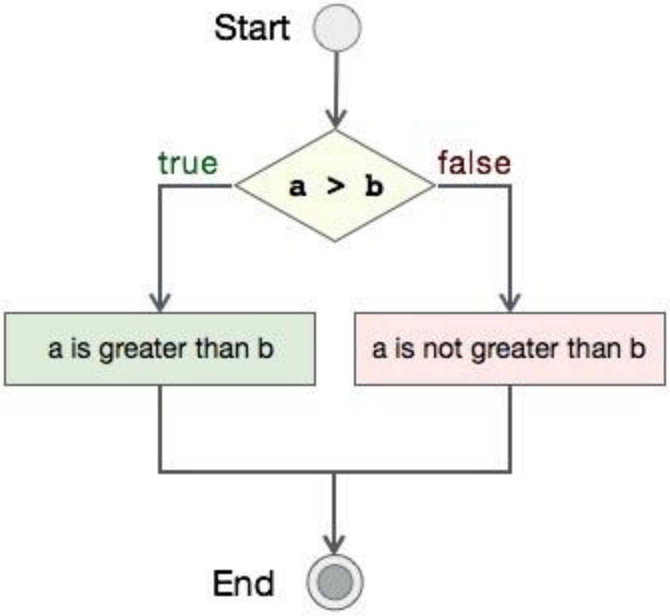
ANSI/ISO Shape	Name	Description
	Flow line (Arrowhead)	Shows the process's order of operation. A line coming from one symbol and pointing at another. Arrowheads are added if the flow is not the standard top-to-bottom, left-to right.
	Terminal	Indicates the beginning and ending of a program or sub-process. They usually contain the word "Start" or "End", or another phrase signaling the start or end of a process, such as "submit inquiry" or

		"receive product".
	Process	Represents a set of operations that changes value, form, or location of data. Represented as a rectangle.
	Decision	Shows a conditional operation that determines which one of the two paths the program will take. The operation is commonly a yes/no question or true/false test. Represented as a diamond (rhombus).
	Input/output	Indicates the process of inputting and outputting data, as in entering data or displaying results. Represented as a parallelogram.
	On-page Connector	Pairs of labeled connectors replace long or confusing lines on a flowchart page. Represented by a small circle with a letter inside.

Flow Chart Add Two number



Flow Chart compare two number



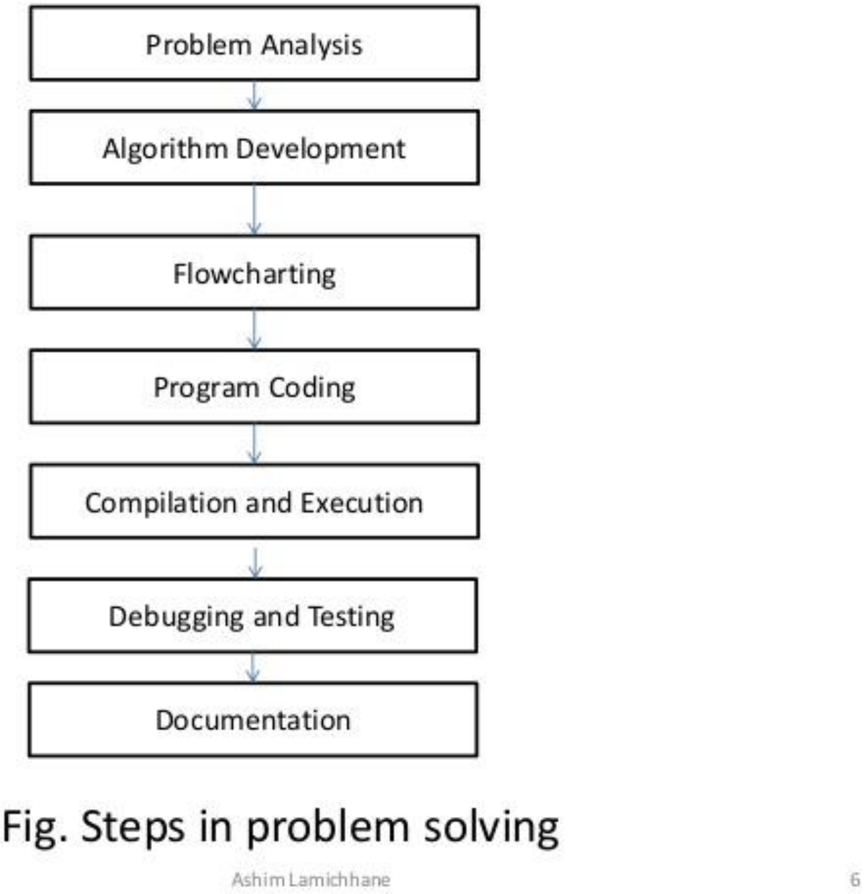
Advantages of using Flowcharts

- **Communication: -** Flowcharts are a better way of communicating the logic of a system to all concerned.
- **Effective analysis:-** With the help of flowcharts, problems can be analyzed more effectively.
- **Proper documentation**:-Program flowcharts serve as a good program documentation needed for various purposes.
- **Efficient coding:-** Flowcharts act as a guide or blueprints during the systems analysis and program development phase.
- **Proper debugging**:- Flowcharts help in the debugging process.
- **Efficient program maintenance:-** The maintenance of an operating program becomes easy with the help of a flowchart.

Disadvantages of using flowcharts

- **Complex logic: -** Sometimes , the program logic is quite complicated . In such a case a flowcharts become complex.
- **Alterations and modification**: **-** If alterations are required the flowcharts may need to be redrawn completely.
- **Reproduction: -** Since the flowcharts symbols cannot be typed in, the reproduction of flowcharts become a problem.
- The essentials of what has to be done can easily be lost in the technical details of how it is to be done.

❖ Problem Analysis



Problem Analysis

Algorithm Development

Flowcharting

Program Coding

Compilation and Execution

Debugging and Testing

Documentation

Fig. Steps in problem solving

Ashim Lamichhane 6

1. **Problem Analysis**: Identify the issues. Be clear about what the **problem** is. ...

Understand everyone's interests. ...

List the possible solutions (options) ...

Evaluate the options.

Select an option or options. ...

Document the agreement(s). ...

Agree on contingencies, monitoring, and evaluation.

2. Algorithm Development

An **algorithm** in general is a sequence of steps to solve a particular problem. **Algorithms** are universal. The **algorithm** you use in **C** programming language is also the same **algorithm** you use in every other language. An algorithm produces the same output information given the same input information, and several short algorithms can be combined to perform complex tasks such as writing a computer program.

3. Flow Chart

A flowchart is a formalized graphic representation of a logic sequence, work or manufacturing process, organization chart, or similar formalized structure. The purpose of a flow chart is to provide people with a common language or reference point when dealing with a project or process. Flowcharts use simple geometric symbols and arrows to define relationships.

4. Program Coding

A **Programming** (or **coding**) language is a set of syntax rules that define how code should be written and formatted. Thousands of different **programming** languages make it possible for us to create computer **software**, apps and websites.

5. Compile and Execution

Compilation

First, the source '.java' file is passed through the compiler, which then encodes the source code into a machine independent encoding, known as Bytecode. The content of each class contained in the source file is stored in a separate '.class' file. While converting the source code into the bytecode.

Execution

The class files generated by the compiler are independent of the machine or the OS, which allows them to be run on any system. To run, the main class file (the class that contains the method main) is passed to the JVM, and then goes through three main stages before the final machine code is executed.

6. **Debugging and testing**

Testing means verifying correct behavior. Testing can be done at all stages of module development: requirements analysis, interface design, algorithm design, implementation, and integration with other modules. In the following, attention will be directed at implementation testing. Implementation testing is not restricted to execution testing. An implementation can also be tested using correctness proofs, code tracing, and peer reviews, as described below.

Debugging is a cyclic activity involving execution testing and code correction. The testing that is done during debugging has a different aim than final module testing. Final module testing aims to demonstrate correctness, whereas testing during debugging is primarily aimed at locating errors. This difference has a significant effect on the choice of testing strategies.

7. **Documentation**

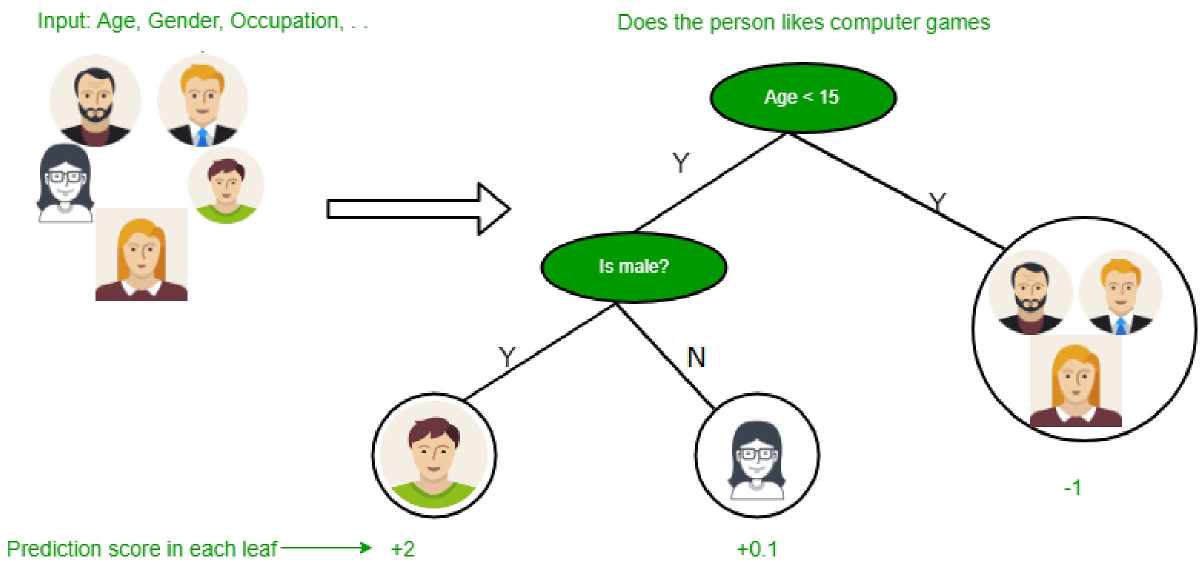
The documentation section contains a set of comment including the name of the program other necessary details. Comments are ignored by compiler and are used to provide documentation to people who reads that code.

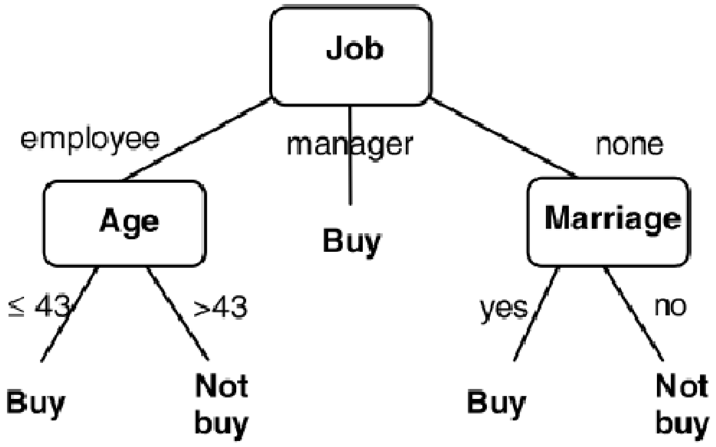
- **Decision tree**

A **decision tree** is a decision support tool that uses a tree-like model of decisions and their possible consequences, including chance event outcomes, resource costs, and utility. It is one way to display an algorithm that only contains conditional control statements.

Decision trees are commonly used in operations research, specifically in decision analysis, to help identify a strategy most likely to reach a goal, but are also a popular tool in machine learning.

- Decision tree algorithm falls under the category of supervised learning. They can be used to solve both regression and classification problems.
- Decision tree uses the tree representation to solve the problem in which each leaf node corresponds to a class label and attributes are represented on the internal node of the tree.
- We can represent any Boolean function on discrete attributes using the decision tree.


Input: Age, Gender, Occupation, . .
Does the person likes computer games
Age < 15
Y
Y
Is male?
Y
N
-1
Prediction score in each leaf
+2
+0.1



Below are some assumptions that we made while using decision tree:

- At the beginning, we consider the whole training set as the root.
- Feature values are preferred to be categorical. If the values are continuous then they are discredited prior to building the model.
- On the basis of attribute values records are distributed recursively.
- We use statistical methods for ordering attributes as root or the internal node.

As you can see from the above image that Decision Tree works on the Sum of Product form which is also known as Disjunctive Normal Form. In **the above image**, we are predicting the use of computer in the daily life of the people.

In Decision Tree the major challenge is to identification of the attribute for the root node in each level. This process is known as attribute selection.

❖ Pseudo code

Pseudocode is an informal way of programming description that does not require any strict programming language syntax or underlying technology considerations. It is used for creating an outline or a rough draft of a program. Pseudocode summarizes a program’s flow, but excludes underlying details. System designers write pseudocode to ensure that programmers understand a software project's requirements and align code accordingly.

Advantages of pseudocode –

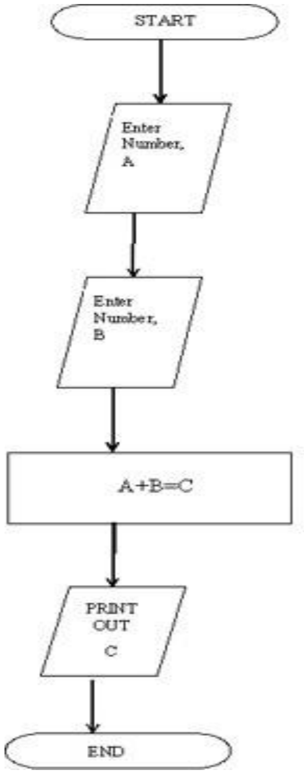
• Pseudocode is understood by the programmers of all types.

• It enables the programmer to concentrate only on the algorithm part of the code development.

• It cannot be compiled into an executable program. Example, Java code : if (i < 10) { i++; } pseudocode :if i is less than 10, increment i by 1.
Let's review an example of pseudocode to **create a program to add 2 numbers together and then display the result**.

```
Start Program
Enter two number A,B
Add the number together
Print add
End program
```

Compare that pseudocode to an example of a flowchart to add two numbers



Now, let's look at a few more simple examples of pseudo code. Here is a pseudocode to **compute the area of a rectangle**:

```
Get the length, l, and width, w
Compute the area = l*w
Display the area
```

Now, let's look at an example of pseudocode to **compute the perimeter of a rectangle**:

```
Enter length, l
Enter width, w
Compute Perimeter = 2*l + 2*w
Display Perimeter of a rectangle
```

Remember, writing basic pseudocode is not like writing an actual coding language. It cannot be compiled or run like a regular program. Pseudocode can be written how you want. But some companies use specific pseudocode syntax to keep everyone in the company on the same page. **Syntax** is a set of rules on how to use and organize statements in a programming language. By adhering to specific syntax, everyone in the company can read and understand the flow of a program. This becomes cost effective and there is less time spent finding and correcting errors.

Advantages of Pseudocode

1. Pseudocode uses English like statements. Therefore, it can be written easily and quickly.
2. In pseudocode, each step is independent of other steps. Therefore, if any modification is done in any step, it does not affect the codes of other module i.e. the code has not to be rewritten if any change occurs in any other step.
3. The format of pseudocode is similar to that of the programs. Both pseudocode and program consist a set of sequential statements and use defined set of keywords. Therefore, pseudocode and be converted to a full-fledged program using any programming language.
4. The learning curve of pseudocode is very smooth. Since, it is written in highly simple language and thus can be understood by any naïve user also.

Disadvantages of Pseudocode

1. Pseudocode is textual representation of an algorithm. It does not provide graphical representation. Therefore, sometimes, it becomes difficult to understand the complex logic written in pseudocode.
2. When too many nested conditions are used in the pseudocode, the level of difficulty to understand the code increases.
3. Since pseudocode focus on detailed description, a lot of practice and concentration is required.

❖ Algorithm in Programming

In programming, algorithm is a set of well defined instructions in sequence to solve the problem. An algorithm is defined as a step-by-step procedure or method for solving a problem by a computer in a finite number of steps. Steps of an algorithm definition may include branching or repetition depending upon what problem the algorithm is being developed for. While defining an algorithm steps are written in human understandable language and independent of any programming language. We can implement it in any programming language of our choice.

Qualities of a good algorithm

1. Input and output should be defined precisely.
2. Each steps in algorithm should be clear and unambiguous.
3. Algorithm should be most effective among many different ways to solve a problem.
4. An algorithm shouldn't have computer code. Instead, the algorithm should be written in such a way that, it can be used in similar programming languages.

Write an algorithm to add two numbers entered by user.

```
Step 1: Start
Step 2: Declare variables num1, num2 and sum.
Step 3: Read values num1 and num2.
Step 4: Add num1 and num2 and assign the result to sum.
        sum←num1+num2
Step 5: Display sum
Step 6: Stop
```

Write an algorithm to find the largest among three different numbers entered by user.

```
    Step 1: Start
    Step 2: Declare variables a,b and c.
    Step 3: Read variables a,b and c.
    Step 4: If a>b
           If a>c
             Display a is the largest number.
           Else
             Display c is the largest number.
         Else
           If b>c
          Display b is the largest number.
           Else
             Display c is the greatest number.
Step 5: Stop
```

Advantages of Algorithms:

1. It is a step-wise representation of a solution to a given problem, which makes it easy to understand.
2. An algorithm uses a definite procedure.
3. It is not dependent on any programming language, so it is easy to understand for anyone even without programming knowledge.
4. Every step in an algorithm has its own logical sequence so it is easy to debug.
5. By using algorithm, the problem is broken down into smaller pieces or steps hence, it is easier for programmer to convert it into an actual program.

Disadvantages of Algorithms:

1. Algorithms is Time consuming.
2. Difficult to show Branching and Looping in Algorithms.
3. Big tasks are difficult to put in Algorithms.

Characteristics of Algorithms:

1. **Precision** – the steps are precisely stated(defined).
2. **Uniqueness** – results of each step are uniquely defined and only depend on the input and the result of the preceding steps.
3. **Finiteness** – the algorithm stops after a finite number of instructions are executed.
4. **Input** – the algorithm receives input.
5. **Output** – the algorithm produces output.

6. **Generality** – the algorithm applies to a set of inputs.

❖ Introduction to C

C is a programming language that has been developed and designed by Dennis Ritchie in 1970's at AT & T's Bell laboratories of USA. It was originally written under UNIX operating system which itself was rewritten later in C language. C seems to be so popular because it is reliable, simple, easy to use and potable. Programs written in C language are fast and very efficient than its predecessors. So before starting C language , user must have an idea of its historical development .

Historical development of C

By 1960's many computer language exist but each has been developed for a specific purpose. Eg. COBOL for business and commercial applications, FORTRAN for engineering and scientific purposes etc.

The C language derives its name from the fact that it is based on a language developed by ken Thompson, another programmer at Bell laboratories . He adapted it from a language known as basic combined programming language(BCPL). To distinguish his version of language from BCPL, Thompson named it B language , which was the first letter of BCPL. When the language was modified and improved to its present state, the second letter of BCPL, C was chosen to represent the new version by Dennis Ritchie.

Merits of C

- C is a general purpose programming language.
- C is a structural programming language.
- C is a standardized programming language.
- It is system independent.
- It has limited data types with great efficiency.
- C has high efficiency.
- It contains a powerful instruction set for manipulating of data.
- It contains the modern methods of coding loops.

❖ What is a Character set?

Like every other language 'C' also has its own character set. A program is a set of instructions that when executed, generate an output. The data that is processed by a program consists of various characters and symbols. The output generated is also a combination of characters and symbols.

A character set in 'C' is divided into,

- Letters
- Numbers
- Special characters
- White spaces (blank spaces)

A compiler always ignores the use of characters, but it is widely used for formatting the data. Following is the character set in 'C' programming:

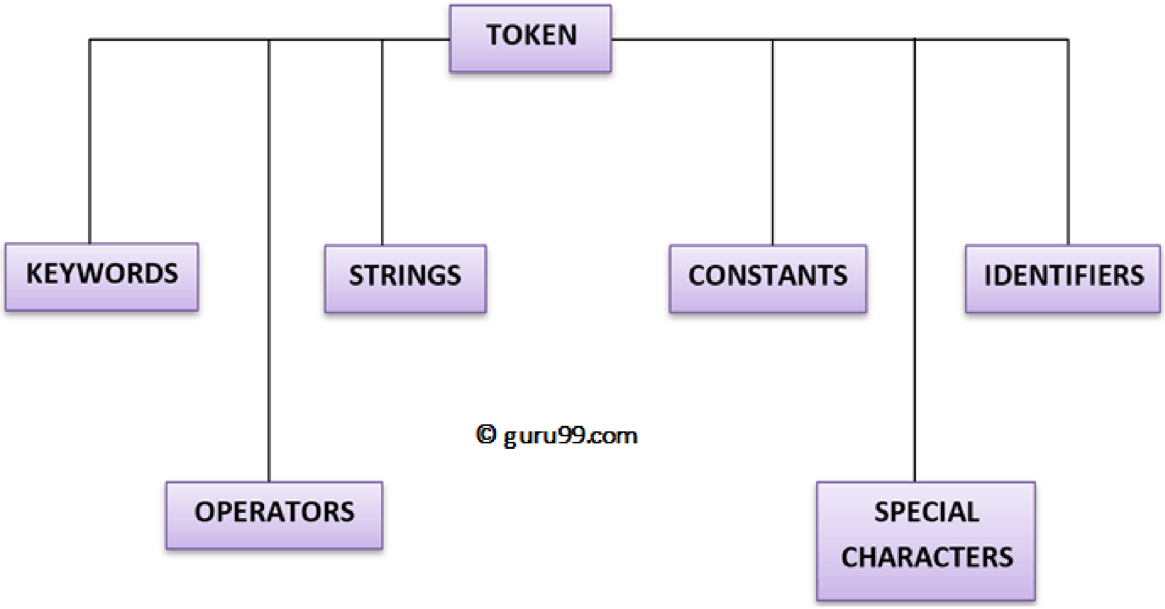
1. Letters
   - Uppercase characters (A-Z)
   - Lowercase characters (a-z)
2. Numbers
   - All the digits from 0 to 9
3. White spaces
   - Blank space
   - New line
   - Carriage return
   - Horizontal tab
4. Special characters
   - Special characters in 'C' are shown in the given table,

, (comma)	{ (opening curly bracket)
. (period)	} (closing curly bracket)
; (semi-colon)	[(left bracket)
: (colon)	] (right bracket)
? (question mark)	((opening left parenthesis)
' (apostrophe)	) (closing right parenthesis)
" (double quotation mark)	& (ampersand)
! (exclamation mark)	^ (caret)
|(vertical bar)	+ (addition)
/ (forward slash)	- (subtraction)
\ (backward slash)	* (multiplication)
~ (tilde)	/ (division)
_ (underscore)	> (greater than or closing angle bracket)

$ (dollar sign)
% (percentage sign)
< (less than or opening angle bracket)
(hash sign)

❖ Token

A token is the smallest unit in a 'C' program. A token is divided into six different types as follows,



Tokens in C

❖ Keywords and Identifiers

In 'C' every word can be either a keyword or an identifier.

Keywords have fixed meanings, and the meaning cannot be changed. They act as a building block of a 'C' program. There are total 32 keywords in 'C'. Keywords are written in lowercase letters.

Following table represents the keywords in 'C',

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	short	float	unsigned
continue	for	signed	void

default	goto	sizeof	volatile
do	if	static	while

An identifier is nothing but a name assigned to an element in a program. **Example**, name of a variable, function, etc. Identifiers are the user-defined names consisting of 'C' standard character set. As the name says, identifiers are used to identify a particular element in a program. Each identifier must have a unique name. Following rules must be followed for identifiers:

1. The first character must always be an alphabet or an underscore.
2. It should be formed using only letters, numbers, or underscore.
3. A keyword cannot be used as an identifier.
4. It should not contain any whitespace character.
5. The name must be meaningful.

❖ What is a Variable?

A variable is an identifier which is used to store some value. Constants can never change at the time of execution. Variables can change during the execution of a program and update the value stored inside it.

A single variable can be used at multiple locations in a program. A variable name must be meaningful. It should represent the purpose of the variable.

Example: Height, age, are the meaningful variables that represent the purpose it is being used for. Height variable can be used to store a height value. Age variable can be used to store the age of a person

Following are the rules that must be followed while creating a variable:

1. A variable name should consist of only characters, digits and an underscore.
2. A variable name should not begin with a number.
3. A variable name should not consist of whitespace.
4. A variable name should not consist of a keyword.
5. 'C' is not a case sensitive language that means a variable named 'age' and 'AGE' are different.

Following are the **examples** of valid variable names in a 'C' program:

height or HEIGHT

_height
_height1
My_name

Following are the **examples** of invalid variable names in a 'C' program:

1height
Hei$ght
My name

For example, we declare an integer variable **my_variable** and assign it the value 48:

```
int my_variable;
my_variable = 48;
```

By the way, we can both declare and initialize (assign an initial value) a variable in a single statement:

```
int my_variable = 48;
```

Local variables

Before learning about the local variable, we should learn about the function block and function parts. There are two parts of the function block (block means region of the function between curly braces in C)

1. **Declaration part -** Region where we declare all variables which are going to be used within the function (this part starts from starting curly brace **"{"**).
2. **Executable part -** Other statements except the declarations are the executable statements.

Global variables

Global variables are the variables which are declared or defined below the header files inclusion section or before the main () function. These variables have global scope to the program in which they are declared. They can be accessed or modified in any function of the program.

Global variable can also be accessed in another files too (for this, we have to declare these variables as extern in associate header file and header file needs to be included within particular file).#include <stdio.h>

```
#include<stdio.h>

/*global variables*/
```

```
int a,b;

int main()

{

        /*local variables*/

        int x,y;



        x=10;

        y=20;

        setValues();



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

        printf("x=%d, y=%d\n",x,y);

        return 0;

}
```

- **Constants**

Constants are the fixed values that never change during the execution of a program. Following are the various types of constants:

I. Integer constants

An integer constant is nothing but a value consisting of digits or numbers. These values never change during the execution of a program. Integer constants can be octal, decimal and hexadecimal.

1. Decimal constant contains digits from 0-9 such as,

Example, 111, 1234

Above are the valid decimal constants.

2. Octal constant contains digits from 0-7, and these types of constants are always preceded by 0.

Example, 012, 065

Above are the valid decimal constants.

3. Hexadecimal constant contains a digit from 0-9 as well as characters from A-F. Hexadecimal constants are always preceded by 0X.

Example, 0X2, 0Xbcd

Above are the valid hexadecimal constants.

The octal and hexadecimal integer constants are very rarely used in programming with 'C'.

II. Character constants

A character constant contains only a single character enclosed within a single quote (''). We can also represent character constant by providing ASCII value of it.

Example, 'A', '9'

Above are the examples of valid character constants.

III. String constants

A string constant contains a sequence of characters enclosed within double quotes ("").

Example, "Hello", "Programming"

These are the examples of valid string constants.

IV. Real Constants

Like integer constants that always contains an integer value. 'C' also provides real constants that contain a decimal point or a fraction value. The real constants are also called as floating point constants. The real constant contains a decimal point and a fractional value.

V. Symbolic Constants

Symbolic Constant is a name that substitutes for a sequence of characters or a numeric constant, a character constant or a string constant.

When program is compiled each occurrence of a *symbolic constant* is replaced by its corresponding character sequence.

The syntax of Symbolic Constants in C

#define name text

where name implies symbolic name in caps**.**
text implies value or the text.

For example,

```
#define printf print
#define MAX 50
#define TRUE 1
#define FALSE 0
#define SIZE 15
```

❖ Expressions

An expression is a formula in which operands are linked to each other by the use of operators to compute a value. An operand can be a function reference, a variable, an array element or a constant.

There are four types of expressions exist in C:

- Arithmetic expressions
- Relational expressions
- Logical expressions
- Conditional expressions

Each type of expression takes certain types of operands and uses a specific set of operators. Evaluation of a particular expression produces a specific value.

❖ Statement

A **statement** is a command given to the computer that instructs the computer to take a specific action, such as display to the screen, or collect input. A computer **program** is made up of a series of **statements**.

Following are the statements which is used in programming In C

1. IF statement

2. IF Else Statement

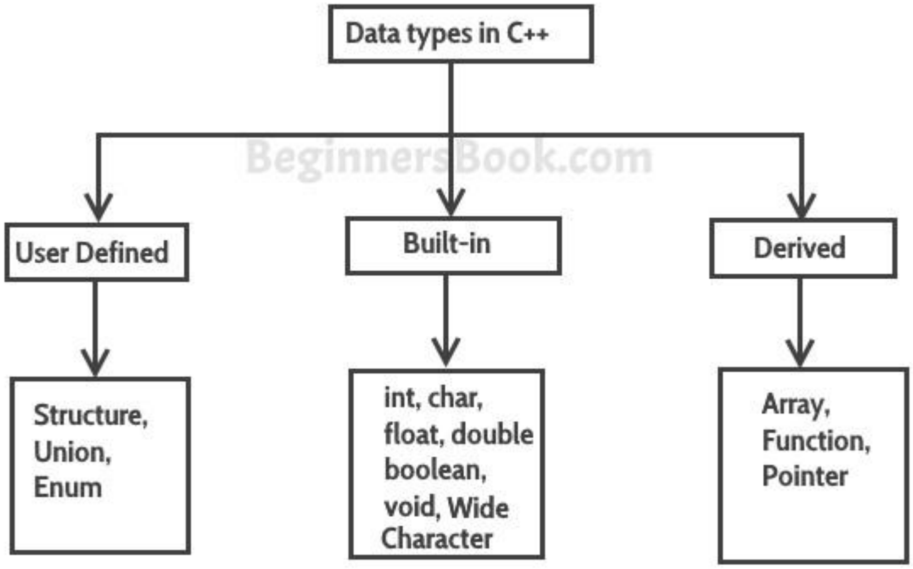
3. Break statement

4. Goto statement

5. Switch statement

6. Continue statement

❖ Data types



'C' provides various data types to make it easy for a programmer to select a suitable data type as per the requirements of an application. Following are the three data types:

I. Build-in and primary data types
II. Derived data types
III. User-defined data types

I. Build-in .and Primary Data Types

There are five primary fundamental data types,

1. int for integer data
2. char for character data
3. float for floating point numbers
4. double for double precision floating point numbers
5. void

Data type	Size in bytes	Range
Char or signed char	1	-128 to 127
Unsigned char	1	0 to 255
int or signed int	2	-32768 to 32767
Unsigned int	2	0 to 65535
Short int or Unsigned short int	2	0 to 255
Signed short int	2	-128 to 127
Long int or Signed long int	4	-2147483648 to 2147483647
Unsigned long int	4	0 to 4294967295
float	4	3.4E-38 to 3.4E+38
double	8	1.7E-308 to 1.7E+308
Long double	10	3.4E-4932 to 1.1E+4932

Note: In C, there is no Boolean data type.

1. Integer data type

Integer is nothing but a whole number. The range for an integer data type varies from machine to machine. The standard range for an integer data type is -32768 to 32767.

An integer typically is of 2 bytes which means it consumes a total of 16 bits in memory. A single integer value takes 2 bytes of memory. An integer data type is further divided into other data types such as short int, int, and long int.

Each data type differs in range even though it belongs to the integer data type family. The size may not change for each data type of integer family.

The short int is mostly used for storing small numbers, int is used for storing averagely sized integer values, and long int is used for storing large integer values.

Whenever we want to use an integer data type, we have place int before the identifier such as,

int age;

Here, age is a variable of an integer data type which can be used to store integer values.

2. Floating point data type

Like integers, in 'C' program we can also make use of floating point data types. The 'float' keyword is used to represent the floating point data type. It can hold a floating point value which means a number is having a fraction and a decimal part. A floating point value is a real number that contains a decimal point. Integer data type doesn't store the decimal part hence we can use floats to store decimal part of a value.

Generally, a float can hold up to 6 precision values. If the float is not sufficient, then we can make use of other data types that can hold large floating point values. The data type double and long double are used to store real numbers with precision up to 14 and 80 bits respectively.

While using a floating point number a keyword float/double/long double must be placed before an identifier. The valid **examples** are,

```
float division;
double BankBalance;
```

3. Character data type

Character data types are used to store a single character value enclosed in single quotes.

A character data type takes up-to **1 byte** of memory space.

Example,

```
Char letter;
```

4. Void data type

A void data type doesn't contain or return any value. It is mostly used for defining functions in 'C'.

Example,

Type declaration of a variableand using data types

```
int main()
 {
int x, y;
float salary = 13.48;
char letter = 'K';
x = 25;
y = 34;
int z = x+y;
printf("%d \n", z);
printf("%f \n", salary);
printf("%c \n", letter);
return 0;}
```

```
Output: 59
13.480000
K
```

We can declare multiple variables with the same data type on a single line by separating them with a comma. Also, notice the use of format specifiers in **printf** output function float (%f) and char (%c) and int (%d).

II. User Define Data Types

C allows the feature called type definition which allows programmers to define their identifier that would represent an existing data type. There are three such types:

Data types	Description
Structure	It is a package of variables of different types under a single name. This is done to handle data efficiently. "struct" keyword is used to define a structure.
Union	These allow storing various data types in the same memory location. Programmers can define a union with different members, but only a single member can contain a value at a given time. It is used for
Enum	Enumeration is a special data type that consists of integral constants, and each of them is assigned with a specific name. "enum" keyword is used to define the enumerated data type.

III. Derived Data Types

C supports three derived data types:

Data types	Description
Arrays	Arrays are sequences of data items having homogeneous values. They have adjacent memory locations to store

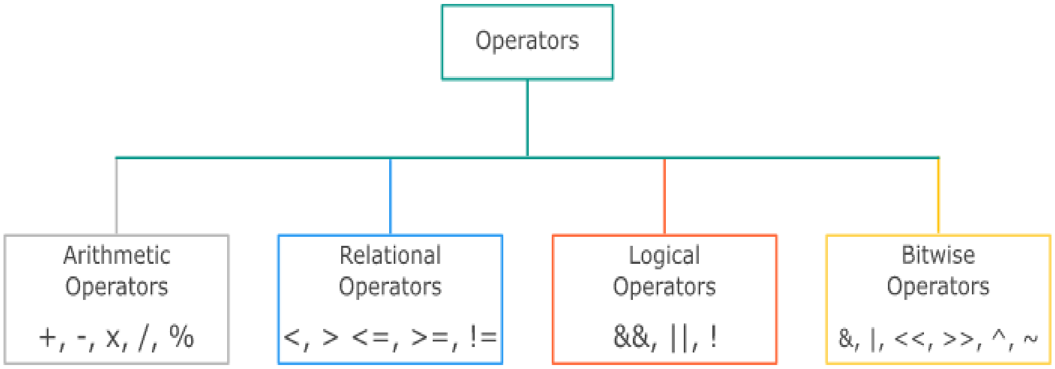
	values.
Function	Function pointers allow referencing functions with a particular signature.
Pointers	These are powerful C features which are used to access the memory and deal with their addresses.

❖ C Operators

Operators are used in c for manipulating the data store into the variables .

Types of Operators

I. Arithmetic
II. Relational
III. Logical
IV. Bitwise



I. C Arithmetic Operators

An arithmetic operator performs mathematical operations such as addition, subtraction, multiplication, division etc on numerical values (constants and variables).

Operator	**Meaning of Operator**
+	addition or unary plus
-	subtraction or unary minus
*	multiplication
/	division
%	remainder after division (modulo division)

Example 1: Arithmetic Operators

```
// Working of arithmetic operators
#include<stdio.h>
int main()
{
int a =9,b =4, 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("a/b = %d \n",c);
c = a%b;
printf("Remainder when a divided by b = %d \n",c);
return0;
}
```

Output

```
a+b = 13
a-b = 5
a*b = 36
a/b = 2
```

II. C Relational Operators

A relational operator checks the relationship between two operands. If the relation is true, it returns 1; if the relation is false, it returns value 0.

Relational operators are used in decision making and loops.

Operator	Meaning of Operator	Example
==	Equal to	5 == 3 is evaluated to 0
>	Greater than	5 > 3 is evaluated to 1
<	Less than	5 < 3 is evaluated to 0
!=	Not equal to	5 != 3 is evaluated to 1
>=	Greater than or equal to	5 >= 3 is evaluated to 1
<=	Less than or equal to	5 <= 3 is evaluated to 0

Example 4: Relational Operators

```
1.  // Working of relational operators

#include<stdio.h>

int main()

{

int a =5, b =5, c =10;

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

   printf("%d == %d is %d \n", a, c, a == c);

   printf("%d > %d is %d \n", a, b, a > b);

   printf("%d > %d is %d \n", a, c, a > c);

   printf("%d < %d is %d \n", a, b, a < b);
```

```
    printf("%d < %d is %d \n", a, c, a < c);

    printf("%d != %d is %d \n", a, b, a != b);

    printf("%d != %d is %d \n", a, c, a != c);

    printf("%d >= %d is %d \n", a, b, a >= b);

    printf("%d >= %d is %d \n", a, c, a >= c);

    printf("%d <= %d is %d \n", a, b, a <= b);

    printf("%d <= %d is %d \n", a, c, a <= c);

 return0;

 }
```

Output

```
5 == 5 is 1
5 == 10 is 0
5 > 5 is 0
5 > 10 is 0
5 < 5 is 0
5 < 10 is 1
5 != 5 is 0
5 != 10 is 1
5 >= 5 is 1
5 >= 10 is 0
5 <= 5 is 1
5 <= 10 is 1
```

III. C Logical Operators

An expression containing logical operator returns either 0 or 1 depending upon whether expression results true or false. Logical operators are commonly used in decision making in C programming.

Operator	**Meaning**	**Example**

Operator	Meaning	Example
&&	Logical AND. True only if all operands are true	If c = 5 and d = 2 then, expression ((c==5) && (d>5)) equals to 0.
||	Logical OR. True only if either one operand is true	If c = 5 and d = 2 then, expression ((c==5) || (d>5)) equals to 1.
!	Logical NOT. True only if the operand is 0	If c = 5 then, expression !(c==5) equals to 0.

Example 5: Logical Operators

```
// Working of logical operators


#include<stdio.h>
int main()
{
int a =5, b =5, c =10, result;
    result =(a == b)&&(c > b
    printf("(a == b) && (c > b) is %d \n", result);
    result =(a == b)&&(c < b);
    printf("(a == b) && (c < b) is %d \n", result);
    result =(a == b)||(c < b);
    printf("(a == b) || (c < b) is %d \n", result);
    result =(a != b)||(c < b);
    printf("(a != b) || (c < b) is %d \n", result);
    result =!(a != b);
    printf("!(a == b) is %d \n", result);

    result =!(a == b);

    printf("!(a == b) is %d \n", result);

return0;

}
```

Output

```
(a == b) && (c > b) is 1
(a == b) && (c < b) is 0
(a == b) || (c < b) is 1
(a != b) || (c < b) is 0
!(a != b) is 1
!(a == b) is 0
```

Explanation of logical operator program

- (a == b) && (c > 5) evaluates to 1 because both operands (a == b) and (c > b) is 1 (true).
- (a == b) && (c < b) evaluates to 0 because operand (c < b) is 0 (false).
- (a == b) || (c < b) evaluates to 1 because (a = b) is 1 (true).
- (a != b) || (c < b) evaluates to 0 because both operand (a != b) and (c < b) are 0 (false).
- !(a != b) evaluates to 1 because operand (a != b) is 0 (false). Hence, !(a != b) is 1 (true).
- !(a == b) evaluates to 0 because (a == b) is 1 (true). Hence, !(a == b) is 0 (false).

IV. C Bitwise Operators

During computation, mathematical operations like: addition, subtraction, multiplication, division, etc are converted to bit-level which makes processing faster and saves power.

Bitwise operators are used in C programming to perform bit-level operations.

Operators	**Meaning of operators**
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
~	Bitwise complement
<<	Shift left
>>	Shift right

V. Other Operators

- **Comma Operator**

Comma operators are used to link related expressions together. For example:

1. int a, c =5, d;

- **The sizeof operator**

The sizeof is a unary operator that returns the size of data (constants, variables, array, structure, etc).

Example 6: sizeof Operator

```
#include<stdio.h>

int main()

{

int a;

float b;

double c;
char d;

   printf("Size of int=%lu bytes\n",sizeof(a));

   printf("Size of float=%lu bytes\n",sizeof(b));

   printf("Size of double=%lu bytes\n",sizeof(c));

   printf("Size of char=%lu byte\n",sizeof(d));

return0;

}
```

Output

```
Size of int = 4 bytes
Size of float = 4 bytes
Size of double = 8 bytes
Size of char = 1 byte
```

Other operators such as ternary operator ? Reference operator &, dereference operator * and member selection operator -> will be discussed in later tutorials.

- **C Increment and Decrement Operators**

C programming has two operators increment ++ and decrement -- to change the value of an operand (constant or variable) by 1.

Increment ++ increases the value by 1 whereas decrement -- decreases the value by 1. These two operators are unary operators, meaning they only operate on a single operand.

Example 2: Increment and Decrement Operators

```
#include<stdio.h>

int main()
```

```
    {
    int a =10, b =100;
    float c =10.5, d =100.5;
printf("++a = %d \n",++a);
        printf("--b = %d \n",--b);
        printf("++c = %f \n",++c);
        printf("--d = %f \n",--d);
    return0;
    }
```

Output

```
++a = 11
--b = 99
++c = 11.500000
--d = 99.500000
```

Here, the operators ++ and -- are used as prefixes. These two operators can also be used as postfixes like a++ and a--. Visit this page to learn more about how increment and decrement operators work when used as postfix.

- **C Assignment Operators**

An assignment operator is used for assigning a value to a variable. The most common assignment operator is =

Operator	**Example**	**Same as**
=	a = b	a = b
+=	a += b	a = a+b
-=	a -= b	a = a-b

Operator	**Example**	**Same as**
*=	a *= b	a = a*b
/=	a /= b	a = a/b
%=	a %= b	a = a%b

Example 3: Assignment Operators

```
// Working of assignment operators
#include<stdio.h>
int main()
{
int a =5, c;
c = a;
printf("c = %d\n", c);
    c += a;
    printf("c = %d\n", c);
    c -= a;
    printf("c = %d\n", c);
    c *= a;
    printf("c = %d\n", c);
    c /= a;
    printf("c = %d\n", c);
    c %= a;
    printf("c = %d\n", c);
return0;
}
```

Output

```
c = 5
c = 10
c = 5
c = 25
c = 5
c = 0
```

❖ Conditional Operator

The conditional operator is also known as a **ternary operator**. The conditional statements are the decision-making statements which depends upon the output of the expression. It is represented by two symbols, i.e., '?' and ':'.

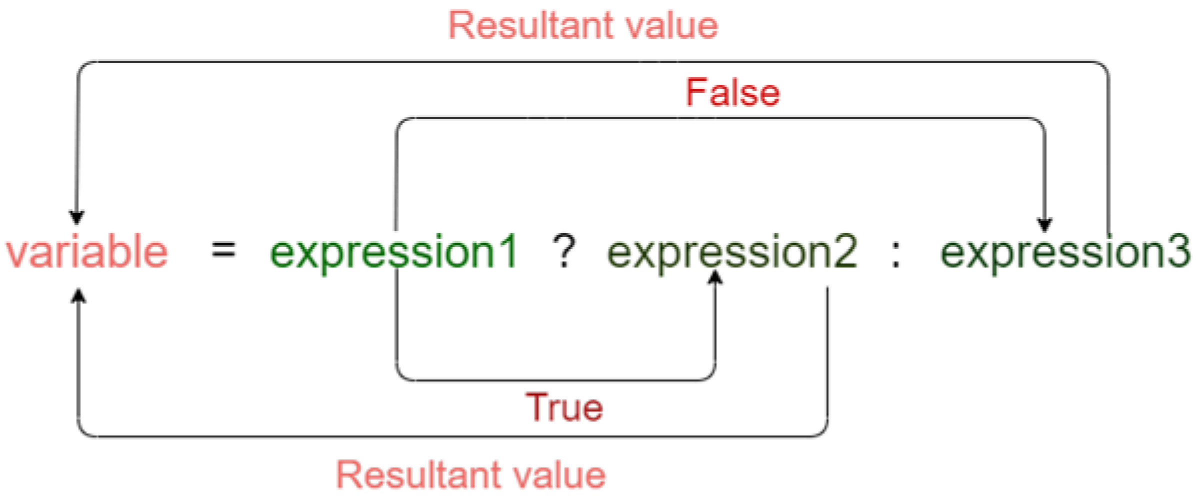
As conditional operator works on three operands, so it is also known as the ternary operator.

The behavior of the conditional operator is similar to the 'if-else' statement as 'if-else' statement is also a decision-making statement.

Syntax of a conditional operator

1. Expression1? expression2: expression3;

The pictorial representation of the above syntax is shown below:



Meaning of the above syntax.

- In the above syntax, the expression1 is a Boolean condition that can be either true or false value.
- If the expression1 results into a true value, then the expression2 will execute.
- The expression2 is said to be true only when it returns a non-zero value.
- If the expression1 returns false value then the expression3 will execute.
- The expression3 is said to be false only when it returns zero value.

❖ C Standard Library Functions

In this tutorial, you'll learn about the standard library functions in C. More specifically, what are they, different library functions in C and how to use them in your program.
C Standard library functions or simply C Library functions are inbuilt functions in C programming.

The prototype and data definitions of these functions are present in their respective header files. To use these functions we need to include the header file in our program. For example,

If you want to use the printf() function, the header file <stdio.h> should be included.

```
#include <stdio.h>
int ma in()
{
  printf("Catch me if you can.");
}
```

If you try to use printf() without including the stdio.h header file, you will get an error.

Advantages of Using C library functions

1. They work

One of the most important reasons you should use library functions is simply because they work. These functions have gone through multiple rigorous testing and are easy to use.

2. The functions are optimized for performance

Since, the functions are "standard library" functions, a dedicated group of developers constantly make them better. In the process, they are able to create the most efficient code optimized for maximum performance.

3. It saves considerable development time

Since the general functions like printing to a screen, calculating the square root, and many more are already written. You shouldn't worry about creating them once again.

4. The functions are portable

With ever-changing real-world needs, your application is expected to work every time, everywhere. And, these library functions help you in that they do the same thing on every computer.

Library Functions in Different Header Files

C Header Files	
<assert.h>	Program assertion functions
<ctype.h>	Character type functions
<locale.h>	Localization functions

C Header Files

<math.h>	Mathematics functions
<setjmp.h>	Jump functions
<signal.h>	Signal handling functions
<stdarg.h>	Variable arguments handling functions
<stdio.h>	Standard Input/Output functions
<stdlib.h>	Standard Utility functions
<string.h>	String handling functions
<time.h>	Date time functions

UNIT-II

❖ Formatted & Unformatted input output

• Unformatted Input/Output functions

I. getchar()
II. putchar()
III. getch()
IV. putch()
V. gets()
VI. puts()
VII. prinf()
VIII. scanf()

I.getchar(): This function reads a character-type data from standard input. • It reads one character at a time till the user presses the enter key.

Example: char c; c = getchar();

```
#include<stdio.h>

 void main()

{

char c;

 printf(“enter a character”);

 c=getchar();

 printf(“c = %c ”,c);

 }
```

II.putchar() : This function prints one character on the screen at a time which is read by standard input.

Example: char c= ‘c’;

```
 putchar (c);

#include<stdio.h>

void main()


{

char ch;

 printf(“enter a character: ”);

 scanf(“%c”, ch);

 putchar(ch);

 }
```

```
enter a character: r

r
```

III. getch() & getche(): These functions read any alphanumeric character from the standard input device The character entered is not displayed by the getch() function until enter is pressed .The getche() accepts and displays the character.

```
#include

void main()

{

printf(“Enter two alphabets:”);

getche();

 getch();

 }
```

```
Enter two alphabets a
```

IV.putch(): This function prints any alphanumeric character taken by the standard input device.

Example:

```
#include<stdio.h>

void main()

{

char ch;

printf(“Press any key to continue”);

ch = getch();

printf(“ you pressed:”);

putch(ch);

}
```

```
Press any key to continue

You pressed : e
```

V. gets(): This function is used for accepting any string until enter key is pressed (string will be covered later).

```
#include <stdio.h>

#include <string.h>

void main()

{

char ch[30];
```

```
 printf(“Enter the string:”);

 gets(ch);

 printf(“Entered string: %s”, ch);

 }

 Enter the string: Use of data!

Entered string: Use of data!
```

VI.puts() : This function prints the string or character array. It is opposite to gets().

```
#include <stdio.h>

#include <string.h>

 void main()

{

   char string[40];

   strcpy(str, "This is a test string");

   puts(string);

}
```

VII.printf():

- In C programming language, printf() function is used to print the “character, string, float, integer, octal and hexadecimal values” onto the output screen.
- We use printf() function with %d format specifier to display the value of an integer variable.
- Similarly %c is used to display character, %f for float variable, %s for string variable, %lf for double and %x for hexadecimal variable.
- To generate a newline,we use “\n” in C printf() statement.

```
int a=10;
```

```
double d=13.4;
printf("%f%d",d,a);
```

VIII.scanf(): In C programming language, scanf() function is used to read character, string, numeric data from keyboard.Consider below example program where user enters a character. This value is assigned to the variable “ch” and then displayed.

Then, user enters a string and this value is assigned to the variable “str” and then displayed.

```
int a;

float b;

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

Example program for printf() and scanf() functions in C programming language:

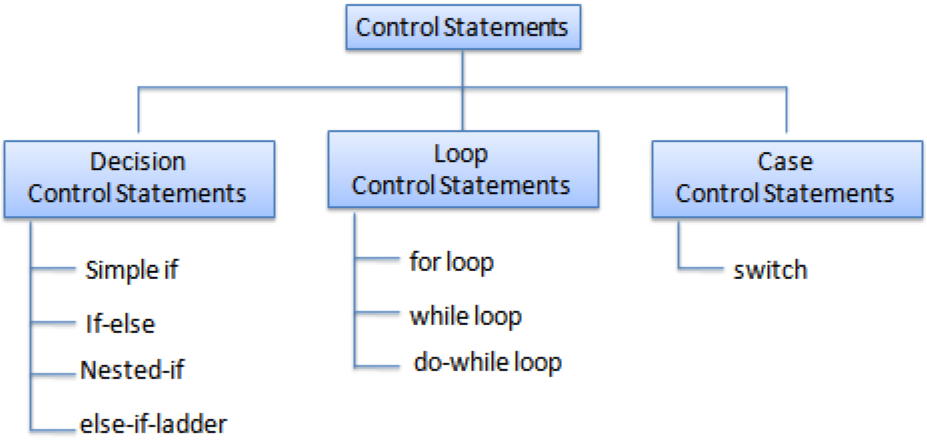
```
#include <stdio.h>
int main()
{
char ch;
char str[100];
   printf("Enter any character \n");
   scanf("%c", &ch);
   printf("Entered character is %c \n", ch);
   printf("Enter any string ( upto 100 character ) \n");
   scanf("%s", &str);
   printf("Entered string is %s \n", str);
}
```

❖ Control Statements:

Control statements enable us to specify the flow of program control; ie, the order in which the instructions in a program must be executed. They make it possible to make decisions, to perform tasks repeatedly or to jump from one section of code to another.

Control Statements are Two Types

I. Decision Making
II. Case
III. Looping

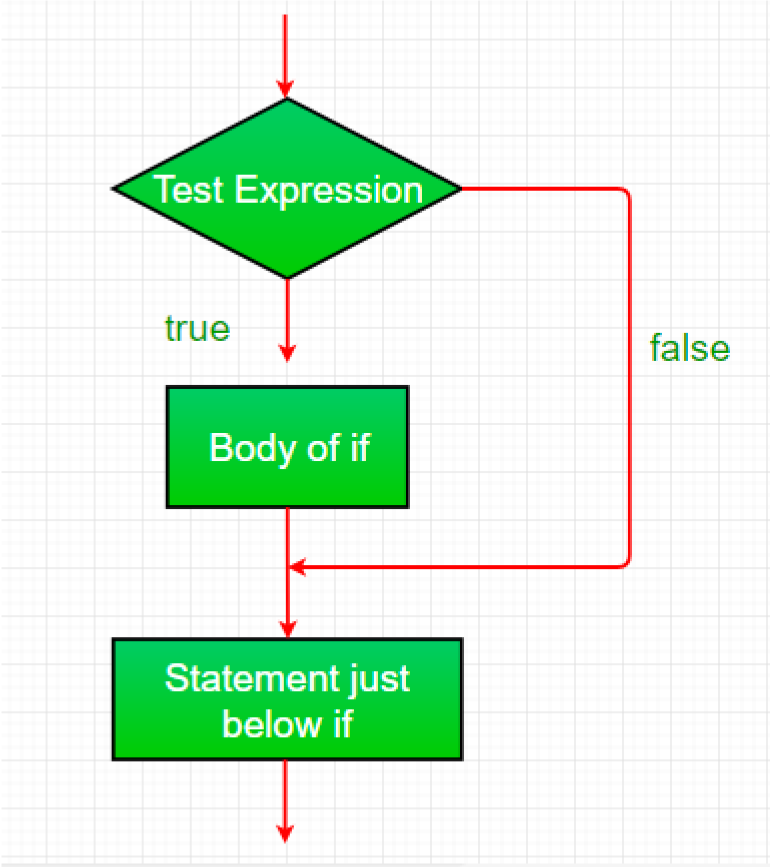


I. **Decision Control Statements:**In decision control statements (if-else and nested if), group of statements are executed when condition is true. If condition is false, then else part statements are executed.

There are 3 types of decision making control statements in C language. They are,

- if statements
- if else statements
- else if ladder

- **if statements**:This is the simplest form of ‘if’ statement. The expression is to be placed in parenthesis. It can be any logical expression.The Block is executed when the given condition is true.



Syntax

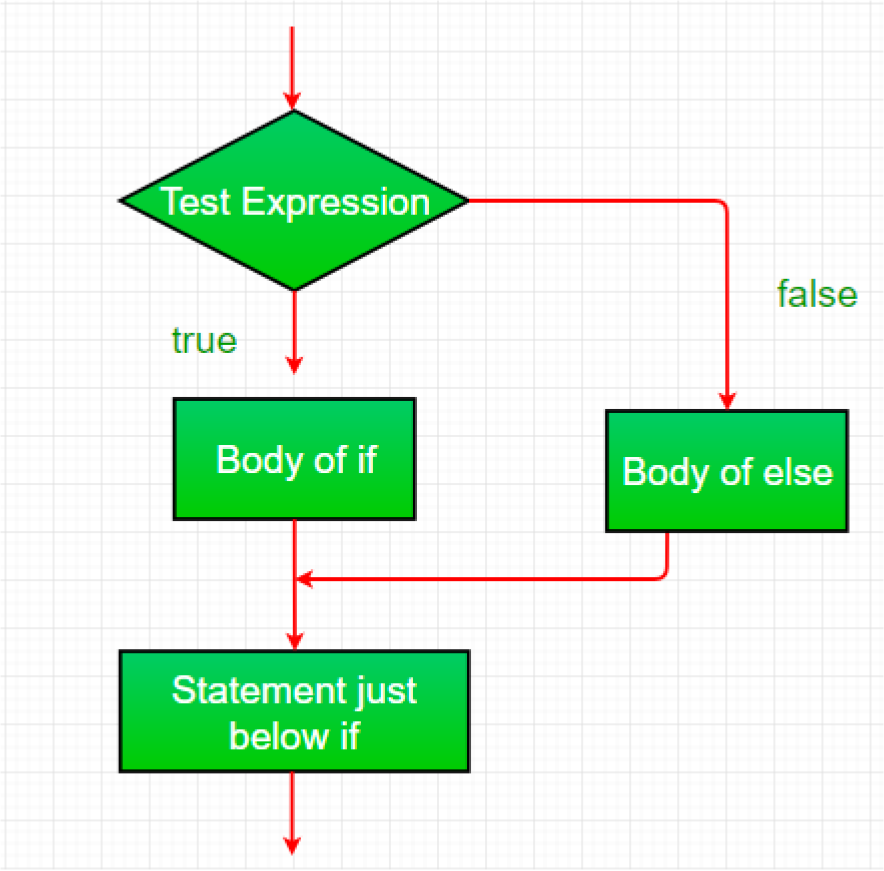
```
If(Condition)
{
Block1;
}
```

Example:

Program check number is Positive

```
#include<stdio.h>
void main()
{
int a;
a=1;
if(a>0)
{
printf("number is positive");
}
}
Output
number is positive
```

- **if else statements** :In the "if" statement seen earlier, we can take some action if expression is true. But if expression is false there is no action. We can include an action for both conditions (i.e. true or false) by using if-else statement. If condition is true block1 is executed otherwise block2.



Syntax

```
If(Condition)

{

Block1;
}
else
{
Block2;
}
```

Example:

Program check number is Positive

```
#include<stdio.h>
void main()
{
int a;
a=1;
if(a>0)
{
printf(“number is positive”);
}
else
{
printf(“number is negative “);
}
}
```

Output

```
number is positive
```

- **else if ladder**

The evolutions of if-else –if ladder or multiple alternative if statement is carried out from top to bottom. Each conditional expression is tested and if found true only then its corresponding statements is executed. In a situation where none of the nested conditions is found true then the final else part is executed.

Syntax

```
if (condition 1)
Block
else if (condition 2)
Block
else if (condition 3)
Block
```

*print days of week input enter by user from 1 to 7

```
#include<stdio.h>
int main()
{
int day;
printf("enter key from 1 to 7 to print days of week");
scanf("%d", &day);
if(day==1)
printf("Today is Monday");
else if(day==2)
printf("Today is Tuesday");

else if(day==3)

printf("Today is Wednesday");

else if(day==4)

printf("Today is thursday");

else if(day==5)

printf("Today is Friday");

else if(day==6)

printf("Today is Saturday");

else if(day==7)

printf("Today is Sunday");

else

printf("wrong input");

}
```

❖ Other decision making statements

jump: Java supports three jump statement: break, continue and goto. These three statements transfer control to other part of the program.

1. **Break**: In Java, break is majorly used for:
   - Terminate a sequence in a switch statement
   - To exit a loop.
   - Used as a “civilized” form of goto.

Using break to exit a Loop

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

```
int main () {

   /* local variable definition */
   int a = 10;

   /* do loop execution */
   do {

      if( a == 15) {
         /* skip the iteration */
         a = a + 1;
break;
      }

      printf("value of a: %d\n", a);
      a++;

   } while( a < 20 );

   return 0;
}
```

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
```

2. **Continue:** The **continue** statement in C programming works somewhat like the **break** statement. Instead of forcing termination, it forces the next iteration of the loop to take place, skipping any code in between.

For the **for** loop, **continue** statement causes the conditional test and increment portions of the loop to execute. For
the **while** and **do...while** loops, **continue** statement causes the program control to pass to the conditional tests.

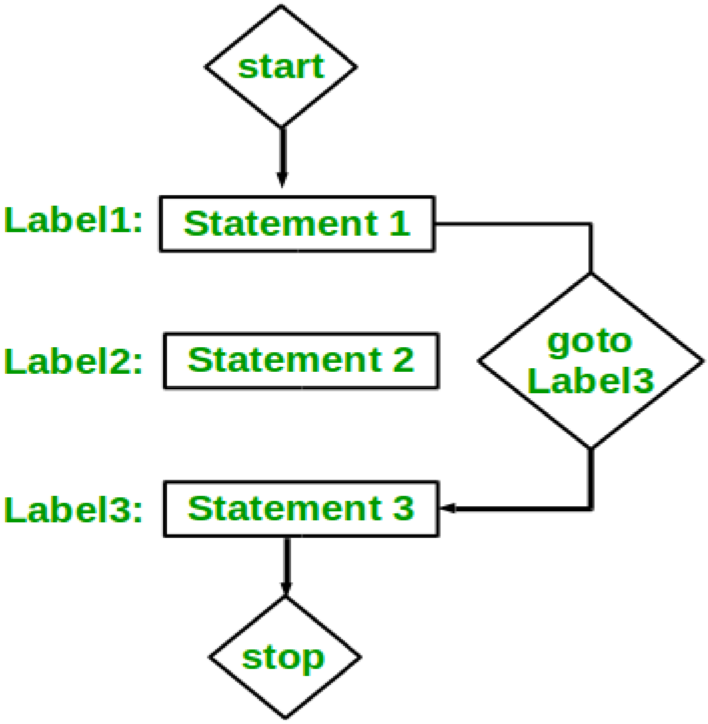
```
#include <stdio.h>

int main () {

   /* local variable definition */
   int a = 10;

   /* do loop execution */
   do {

      if( a == 15) {
         /* skip the iteration */
         a = a + 1;
         continue;
      }

      printf("value of a: %d\n", a);
```

```
     a++;

   } while( a < 20 );

   return 0;
}
```

Output:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

3. **goto**: The goto statement is a jump statement which is sometimes also referred to as unconditional jump statement. The goto statement can be used to jump from anywhere to anywhere within a function.



Example

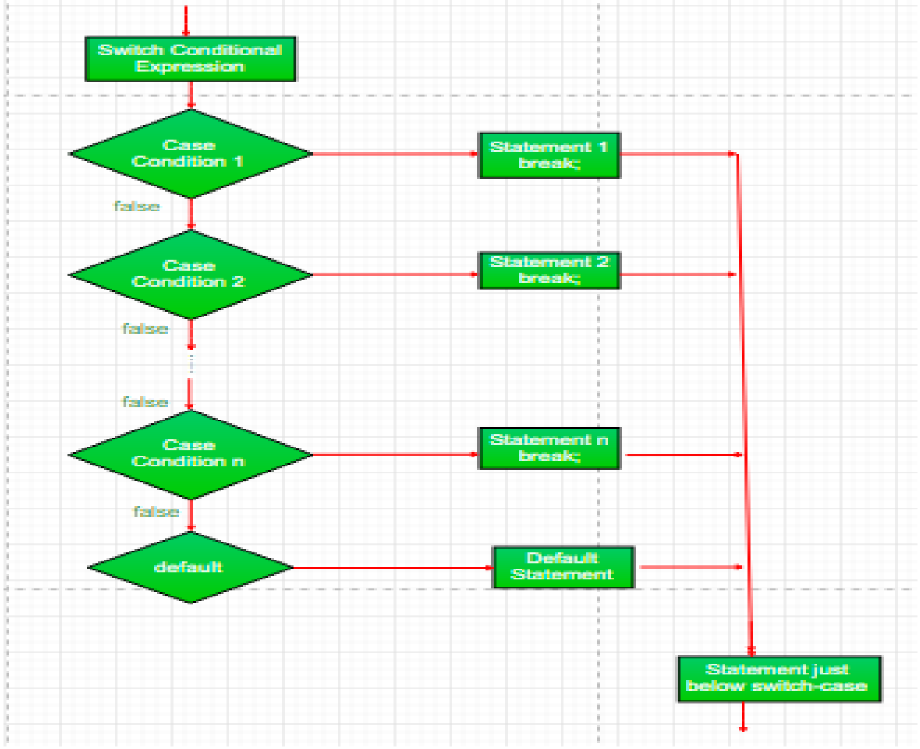
```
#include <stdio.h>

int main()

{

int number=1;

while(number<=10)

{

if(number==4)

        goto end;

   printf("Number is : %d", number);

 end:

   printf("Bye Bye !!!");

}

return 0;

}
```

Output

```
   First run:

   Number is : 1 2 3

   Bye Bye !!!
```

II. **switch – case**

Another form of statement available for selective execution is the switch statement. It causes particular group of statements to be selected from several available group.



The general format is as follows.

switch (expression)

Statement;

Where statement consist of one more case statements followed by a colon and group of statements.

switch (expression)

{

case expression 1 : statement 1;

statement 2;

break;

```
case expression 2 : statement 1;
statement 2;
break;
default :
statement 1;
statement 2;
}
```

The expression should result in an integer value or character value. First the expression following switch is solved.

```
#include<stdio.h>
void main ( )
{
int j = 2;
switch ( j )
{
case 1 :
printf ("nI am in case 1.");
break;
case 2:
printf ("nI am in case 2.");
break;
case 3:
printf ("nI am in case 3.");
```

default:

printf ("nI am in default.");

}

}

III. **Looping:** Sometimes we want some part of our code to be executed more than once. We can either repeat the code in our program or use loops instead. It is obvious that if for example we need to execute some part of code for a hundred times it is not practical to repeat the code. Alternatively we can use our repeating code inside a loop.

Types of Loops in C

- while
- do-while
- for

- **while**: The statements within the while loop would keep on getting executed till the condition being tested remains true. When the condition becomes false, the control passes to the first statement that follows the body of the while loop. It is entry control loop.

Syntax

Initialization;

While(condition)

{

Block;

Increment/decrement;

}

Three important parts of Loop

i) **initialization:**it is the starting Point of loop.

ii) **Test Condition:**it is the given condition which is to be checked.

iii) **Increment/decrement :**for Increment (++) is used and(--) is used for decrement to given statements.

Example:

```
   #include <stdio.h>

   int main ()

 {


int a = 10;


      while( a < 14 ) {

         printf("value of a: %d\n", a);

         a++;

      }

  return 0;

   }
```

output

```
value of a:10

value of a:11

value of a:12

value of a:13
```

- **do while:**The while and for loops test the termination condition at the top. By contrast, the third loop in C, the do-while, tests at the bottom after making each pass through the loop body; the body is always executed at least once.

Syntax

```
Initialization;
do
{
Block;
    Increment/decrement ;
}while (condition);
```

Example

```
#include <stdio.h>
int main () {
  int a = 10;
  do
{
    printf("value of a: %d\n", a);
    a + +;
  }
while( a < 14 );
  return 0;
}
```

output

```
value of a:10

value of a:11

value of a:12

value of a:13
```

- **for loop:** for loop is something similar to while loop but it is more complex. for loop is constructed from a control statement that determines how many times the loop will run and a command section. Command section is either a single command or a block of commands.

  Syntax

```
    for ( initialization; test condition; increment/decrement )
{
Block;
}
```

Example

```
#include <stdio.h>

int main ()

{

  int a;

  for( a = 10; a <14; a + + )

{

     printf("value of a: %d\n", a);

  }
```

```
   return 0;
}
```

```
output
value of a:10
value of a:11
value of a:12
value of a:13
```

BASIS FOR COMPARISON	**WHILE**	**DO-WHILE**

General Form	while (condition) {
	statements; //body of loop
	}
		do{
		.
		statements; // body of loop.
		.
		} while(Condition);

Controlling Condition	In 'while' loop the controlling condition appears at the start of the loop.	In 'do-while' loop the controlling condition appears at the end of the loop.
Iterations	The iterations do not occur if, the condition at the first iteration, appears false.	The iteration occurs at least once even if the condition is false at the first iteration.
type	It entry Control Loop	It Exit Control Loop.

UNIT-III

❖ Function

A function is a group of statements that together perform a task. Every C program has at least one function.You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division is such that each function performs a specific task.

The advantages of using functions are:

- Avoid repetition of codes.
- Increases program readability.
- Divide a complex problem into simpler ones.
- Reduces chances of error.
- Modifying a program becomes easier by using function.

❖ Three parts of Function

✓ **Function declaration(Prototype):**A function declaration tells the compiler about a function's name, return type, and parameters.

Syntax

Returntype Functionname (parameterlist);

A function definition in C programming consists of a function header and a function body. Here are all the parts of a function −

- **Return Type** − A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword void.
- **Function Name** − This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters** − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters

✓ **Function Definition:** function definition in which number of statements are write down into body of the function.

Syntax

Returntype Functionname(parameterlist)

{

Statements;

}

✓ **Function call**:A program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

Syntax

Functionname();

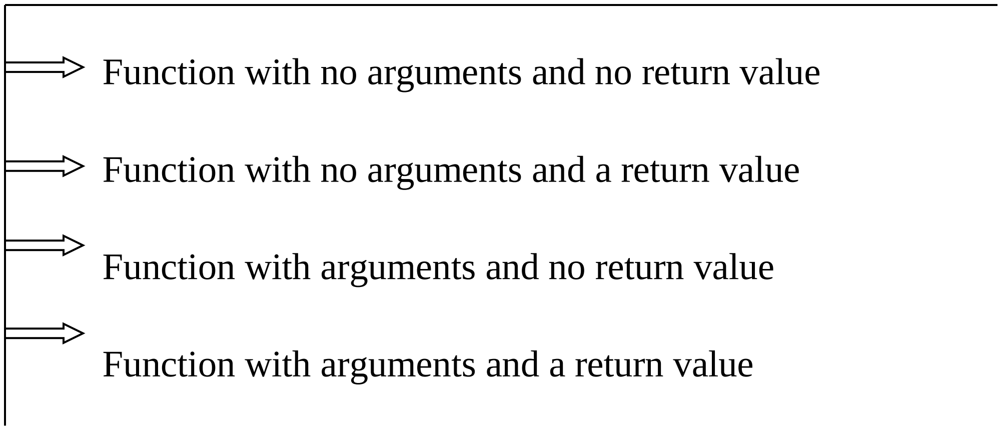
Types of Function in C

- **Predefined Functions:** The Predefined Functions are those which are already defined into C Like printf(),scanf(),main() etc.
- **User defined Functions:** These Functions are made by program itself to perforam any task and solve any problem.

❖ User defined Functions: Categories of User defined Functions are

1) Function with no arguments and no return value
2) Function with no arguments and a return value
3) Function with arguments and no return value/ Specifying argument data types
4) Function with arguments and a return value/ Function with argument

Categories of User defined Functions (Function Prototypes)



Function with no arguments and no return value

Function with no arguments and a return value

Function with arguments and no return value

Function with arguments and a return value

1) **Function with no arguments and no return value**: This Type of Function have no arguments and any return value that void.

Syntax:

```
Void functionname (void);
```

Example:

```
#include<stdio.h>
void sum(void);  //function declaration
void sum(void)
{
int a,b,c;
printf("enter two number\n");   //function defination

scanf("%d%d",&a,&b);
c=a+b;
printf("sum is %d",c);
}
void main()
{
sum(); //function call

}
```

Output

```
enter two number
10 20
sum is 30
```

2) **Function with no arguments and a return value**:This Type of Function have no arguments with any return value that is the calculated back to the main().

Syntax:

```
returnvalue functionname (void);
```

Example:

```
#include<stdio.h>
int sum(void);  //function declaration
int sum(void)
```

```
{
int a,b,c;
printf("enter two number\n");   //function defination
scanf("%d%d",&a,&b);
c=a+b;
return(c);
}
void main()
{
int c;
c=sum(); //function call
printf("sum is %d",c);
}
```

Output

```
enter two number
10 20
sum is 30
```

3) **Function with arguments and No return value**:This Type of Function have arguments that are the value which is enter from the main(), the arguments have datatype and every argument separated by commas(,). These have no any return value .

Syntax:

```
void functionname (parameter list);
```

Example:

```
#include<stdio.h>
void sum(int a,int b);  //function declaration
void sum(int a,int b)
{
```

```
int a,b,c;
c=a+b;                    //function defination
printf(“sum is %d”,c);

}
void main()
{
int a,b;
printf(“enter two number\n”);
scanf(“%d%d”,&a,&b);

sum(a,b); //function call
}
```

Output

```
enter two number
10 20
sum is 30
```

4) **Function with arguments and with return value**:This Type of Function have two way communication between user define function and main().the user define function accept the data from main() and back the result to main().

Syntax:

```
void functionname (parameter list);
```

Example:

```
#include<stdio.h>
int sum(int a,int b);  //function declaration
int sum(int a,int b)
{
```

```
int a,b,c;
c=a+b;                  //function defination
return ( c );
}
void main()
{
Int a,b, c;
printf("enter two number\n");
scanf("%d%d",&a,&b);

c=sum(a,b); //function call
printf("sum is %d",c);
}
```

Output

```
enter two number
10 20
sum is 30
```

what formal and actual arguments

Formal Argument :

The formal arguments are the arguments in the function declaration. The scope of formal arguments is local to the function definition in which they are used. They belong to the called function.

Actual arguments :

The arguments that are passed in a function call are called actual arguments. These arguments are defined in the calling function.

Example:

```
#include<stdio.h>
int sum(int a,int b);  //function declaration
int sum(int a,int b)  →  formal arguments
```

```
{
int a,b,c;
c=a+b;                //function definition
return ( c );
}
void main()
{
Int a,b, c;
printf("enter two number\n");
scanf("%d%d",&a,&b);

c=sum(a,b); //function call  ──────────────────→  Actual Arguments
printf("sum is %d",c);
}
```

- **Call by value** :

In *call by value*, a copy 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 no effect on the values of actual arguments in the calling function.

In call by value, actual arguments will remain safe, they cannot be modified accidentally.

Example using Call by Value

The classic example of wanting to modify the caller's memory is a swapByValue() function which exchanges two values. For C uses call by value, the following version of *swap* swapByValue() will not work...

```
#include <stdio.h>
void swapByValue(int a, int b)
{
 int t;
```

```
  t = a;
a = b;
 b = t;
printf(" Values In user define a: %d, b: %d\n", a,b);
}
 int main() /* Main function */
{
  int a1 = 10;
int  b1 = 20;

/* actual arguments will be as it is */
  swapByValue(a1, b1);
printf ("Values In main a1: %d, b1: %d\n", a1,b1);
}OUTPUT
======
Values In user define a: 20, b: 10
Values In main  a1: 10, b1: 20
```

Call by Reference

In call by reference, to pass a variable n as a reference parameter, the programmer must pass a pointer to n instead of n itself. The formal parameter will be a pointer to the value of interest. The calling function will need to use & to compute the pointer of actual parameter. The called function will need to dereference the pointer with *where appropriate to access the value of interest. Here is an example of a correct *swap* swapByReference() function. So, now you got the difference between call by value and call by reference!

```
#include <stdio.h>
 void swapByValue(int *a, int *b); /* Prototype */
void swapByValue(int *a, int * b)
{
  int t;
  t = *a;
*a = *b;
*b = t;
printf(" Values In user define *a: %d, *b: %d\n", *a,*b);

}
 int main() /* Main function */
{
  int a1 = 10;
int  b1 = 20;

/* actual arguments will be as it is */
  swapByValue(&a1,& b1);
printf("Values In main a1: %d, b1: %d\n", a1, b1);
}
```

OUTPUT

```
======
Values In user define a: 20, b: 10
Values In main  a1: 20, b1: 10
```

CALL BY VALUE	CALL BY REFERENCE
While calling a function, we pass values of variables to it. Such	While calling a function, instead of passing the values of variables, we

functions are known as “Call By Values”.	pass address of variables(location of variables) to the function known as “Call By References.
In this method, the value of each variable in calling function is copied into corresponding dummy variables of the called function.	In this method, the address of actual variables in the calling function are copied into the dummy variables of the called function.
With this method, the changes made to the dummy variables in the called function have no effect on the values of actual variables in the calling function.	With this method, using addresses we would have an access to the actual variables and hence we would be able to manipulate them.

// C program to illustrate
// call by value
 	
#include <stdio.h>
 	
// Function Prototype
void swapx(int x, int y);
 	
// Main function
	// C program to illustrate
	// Call by Reference
	
	#include <stdio.h>
	
	// Function Prototype
	void swapx(int*, int*);
	
	// Main function

```
int main()
{
   int a = 10, b = 20;

   // Pass by Values
   swapx(a, b);

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

   return 0;
}

// Swap functions that swaps
// two values
void swapx(int x, int y)
{
   int t;

   t = x;
   x = y;
   y = t;

   printf("x=%d y=%d\n", x, y);
}
```

Output:

```
x=20 y=10
a=10 b=20
```

```
int main()
{
   int a = 10, b = 20;

   // Pass reference
   swapx(&a, &b);

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

   return 0;
}

// Function to swap two variables
// by references
void swapx(int* x, int* y)
{
   int t;

   t = *x;
   *x = *y;
   *y = t;

   printf("x=%d y=%d\n", *x, *y);
}
```

Output:

```
x=20 y=10
a=20 b=10
```

Thus actual values of a and b remain unchanged even after exchanging the values of x and y.	Thus actual values of a and b get changed after exchanging values of x and y.
In call by values we cannot alter the values of actual variables through function calls.	In call by reference we can alter the values of variables through function calls.
Values of variables are passes by Simple technique.	Pointer variables are necessary to define to store the address values of variables.

Syntax
Returntype
functioname(parameterlist)
{
Body;
}
	Syntax
	Returntype
	functioname(*parameterlist)
	{
	Body;
	}

❖ Recursion

What is Recursion?

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function. Using recursive algorithm, certain problems can be solved quite easily

What is base condition in recursion?

In the recursive program, the solution to the base case is provided and the solution of the bigger problem is expressed in terms of smaller problems.

```
#include <stdio.h>
/* Function declaration */
void printNaturalNumbers(int lowerLimit, int upperLimit);0
int main()
{
    int lowerLimit, upperLimit;

    /* Input lower and upper limit from user */
    printf("Enter lower limit: ");
    scanf("%d", &lowerLimit);
    printf("Enter upper limit: ");
    scanf("%d", &upperLimit);

    printf("All natural numbers from %d to %d are: ", lowerLimit, upperLimit);
    printNaturalNumbers(lowerLimit, upperLimit);

    return 0;
}



/**
 * Recursively prints all natural number between the given range.
 */
void printNaturalNumbers(int lowerLimit, int upperLimit)
```

```
{
    if(lowerLimit > upperLimit)
        return;

    printf("%d, ", lowerLimit);

    // Recursively call the function to print next number
    printNaturalNumbers(lowerLimit + 1, upperLimit);
}
```

Diffrence between Recursion and Iteration

Iteration	**Recursion**
Allows the execution of a sequential set of statements repetitively using conditional loops.	A statement in the function's body calls the function itself.
There are loops with a *control variable* that need to be initialized, incremented or decremented and a **conditional control statement** that continuously gets checked for the *termination of execution*.	A recursive function must comprise of at least one **base case** i.e. a condition for *termination of execution*.
The value of the control variable continuously *approaches* the value in the **conditional statement**.	The function keeps on *converging* to the defined base case as it **continuously calls itself**.
A control variable stores the value, which is then updated, monitored, and compared with the conditional statement.	Stack memory is used to store the current state of the function.
Infinite loops keep utilizing *CPU cycles* until we stop their execution manually.	If there is **no base case** defined, recursion causes a *stack overflow error*.
The execution of iteration is comparatively **faster**.	The execution of recursion is comparatively **slower**.

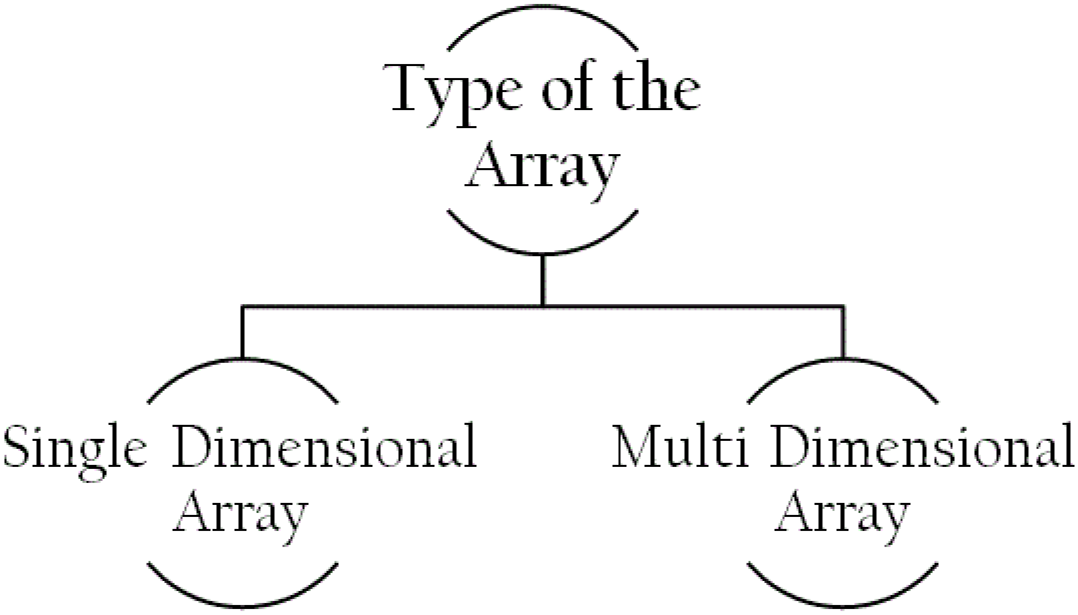
❖ **Arrays:** Defining, processing arrays, passing arrays to a function, multi– dimensional arrays.

1. **Array**:Arrays a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

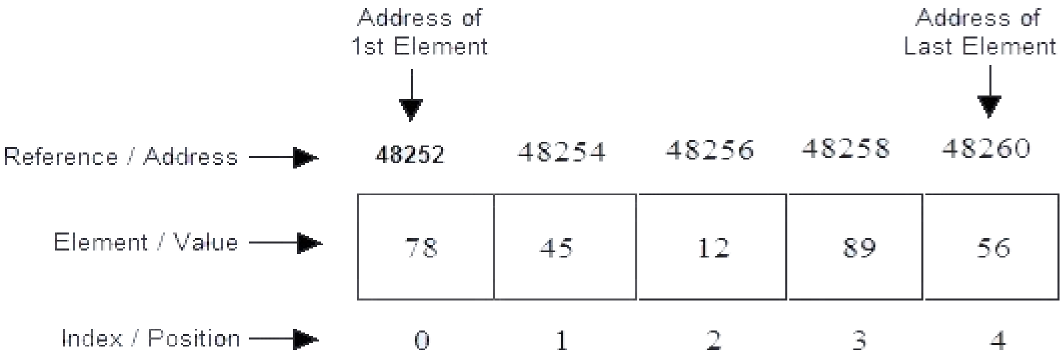
❖ **Types of Arrays in C**

There are three type of array in C language:

- **One dimensional array:**
- **Two dimensional array:**
- **Multi– dimensional arrays:**



- **One dimensional array:**
  - Single or One Dimensional array is used to represent and store data in a linear form.
  - Array having only one subscript variable is called One-Dimensional array
  - It is also called as Single Dimensional Array or Linear Array



Syntax

```
Datatype arrayname[size];
```

Declaring Arrays

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows −

```
type arrayName [ arraySize ];
```

This is called a single-dimensional array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type. For example, to declare a 10-element array called **balance** of type double, use this statement −

```
double balance[10];
```

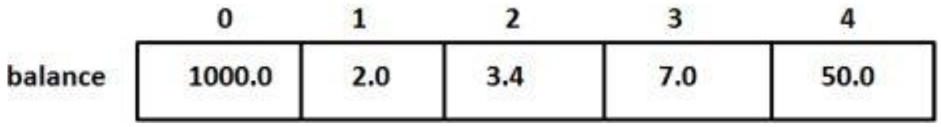
Here balance is a variable array which is sufficient to hold up to 10 double numbers.

Initializing Arrays

You can initialize an array in C either one by one or using a single statement as follows −

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ].



	0	1	2	3	4
balance	1000.0	2.0	3.4	7.0	50.0

Example of storing elements into array

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

```
void main()
{
int a[5];
int i;
printf("enter the number atleast 5\n");
for(i=0;i<5;i++)
{
scanf("%d",&a[i]);
}
printf("elements you enter are\n");
for(i=0;i<5;i++)
{
printf("%d\n",a[i]);
}}
```

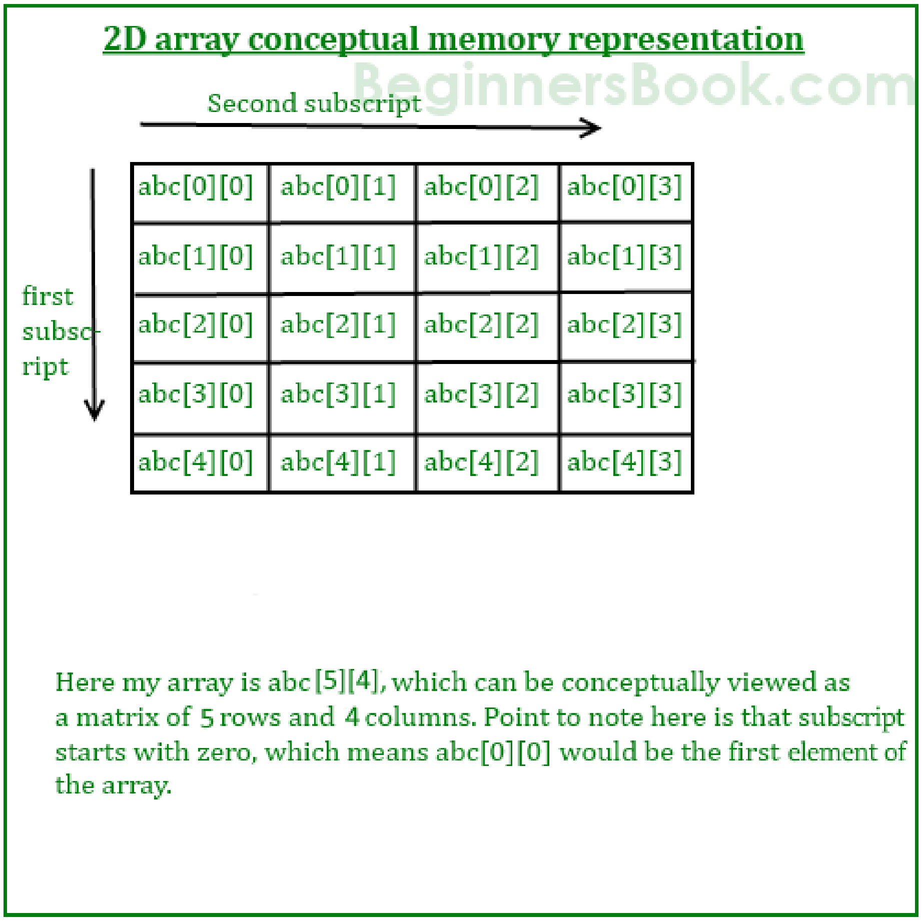
Output

```
enter the number atleast 5
10 20 30 40 50
elements you enter are
10
20
30
40
50
```

- **Two dimensional array:**
  - Array having more than one subscript variable is called Multi-dimensional array.
  - Multi Dimensional Array is also called as Matrix.

Declaration of 2D Array

While declaring the 2D Array there are two subscript values.



2D array conceptual memory representation

abc[0][0]	abc[0][1]	abc[0][2]	abc[0][3]
abc[1][0]	abc[1][1]	abc[1][2]	abc[1][3]
abc[2][0]	abc[2][1]	abc[2][2]	abc[2][3]
abc[3][0]	abc[3][1]	abc[3][2]	abc[3][3]
abc[4][0]	abc[4][1]	abc[4][2]	abc[4][3]

Here my array is abc[5][4], which can be conceptually viewed as a matrix of 5 rows and 4 columns. Point to note here is that subscript starts with zero, which means abc[0][0] would be the first element of the array.

Syntax:

Datatype Arrayname[rowsize][columnsize];

Initialization of 2D Array

There are two ways to initialize a two Dimensional arrays during declaration.

```
int disp[2][4] = {
    {10, 11, 12, 13},
    {14, 15, 16, 17}
};
```

Example of 2D Array

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

```
int main()
{
   /* 2D array declaration*/
   int disp[2][3];
   /*Counter variables for the loop*/
   int i, j;
   for(i=0; i<2; i++) {
      for(j=0;j<3;j++) {
         printf("Enter value for disp[%d][%d]:", i, j);
         scanf("%d", &disp[i][j]);
      }
   }
   //Displaying array elements
   printf("Two Dimensional array elements:\n");
   for(i=0; i<2; i++) {
      for(j=0;j<3;j++) {
         printf("%d ", disp[i][j]);
         if(j==2){
            printf("\n");
         }
      }
   }
```

```
  return 0;
}
```

Output:

```
Enter value for disp[0][0]:1
Enter value for disp[0][1]:2
Enter value for disp[0][2]:3
Enter value for disp[1][0]:4
Enter value for disp[1][1]:5
Enter value for disp[1][2]:6
Two Dimensional array elements:
1 2 3
4 5 6
```

- **Multi– dimensional arrays**:In C, we can define multidimensional arrays in simple words as array of arrays. Data in multidimensional arrays are stored in tabular form (in row major order).

General form of declaring N-dimensional arrays:

```
data_type  array_name[size1][size2]....[sizeN];
```

data_type: Type of data to be stored in the array.

Here data_type is valid Cdata type

array_name: Name of the array

size1, size2,... ,sizeN: Sizes of the dimensions

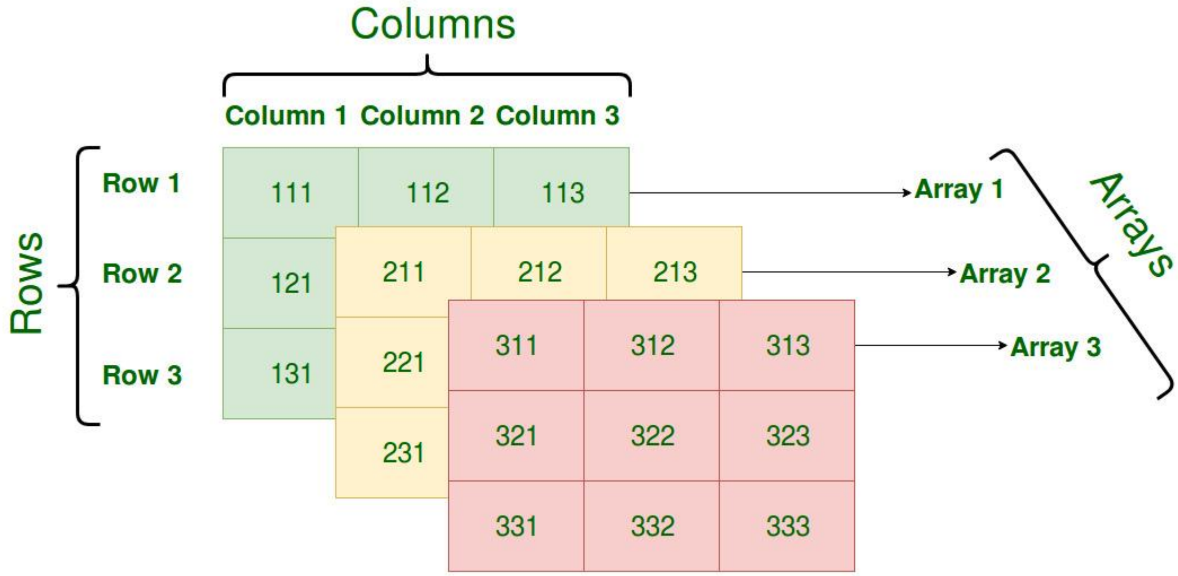
Examples:

Two dimensional array:

```
int two_d[10][20];
```

Three dimensional array:

```
int three_d[10][20][30];
```



Initializing Three-Dimensional Array: Initialization in Three-Dimensional array is same as that of Two-dimensional arrays. The difference is as the number of dimension increases so the number of nested braces will also increase.

Method 1:

```
int x[2][3][4] =

 {

   { {0,1,2,3}, {4,5,6,7}, {8,9,10,11} },
  { {12,13,14,15}, {16,17,18,19}, {20,21,22,23} }
 };
```

```
// C  program to print elements of Three-Dimensional
// Array
#include<stdio.h>
int main()
{
   // initializing the 3-dimensional array
   int x[2][3][2] =
   {
      { {0,1}, {2,3}, {4,5} },
      { {6,7}, {8,9}, {10,11} }
   };

   // output each element's value
   for (int i = 0; i < 2; ++i)
   {
        for (int j = 0; j < 3; ++j)

        {

            for (int k = 0; k < 2; ++k)

            {

               printf("%d", x[i][j][k]);



            }

printf("\n");

        }

    }

    return 0;

}
```

❖ Passing arrays to a function:

Passing a single element of an array to a function is similar to passing variable to a function.

Example

```
// Program to calculate average by passing an array to a function



#include <stdio.h>
```

```
float average(float age[]);

float average(float age[])

{

int i;

float avg, sum = 0.0;

for (i = 0; i < 6; ++i) {

sum += age[i];

}

avg = (sum / 6);

return avg;

}


int main()

{

float avg, age[] = {23.4, 55, 22.6, 3, 40.5, 18};

avg = average(age); // Only name of an array is passed as an argument

printf("Average age = %.2f", avg);

return 0;
}
```

Output

```
Average age = 27.08
```

❖ Advantages of Arrays

- It is better and convenient way of storing the data of same datatype with same size.
- It allows us to store known number of elements in it.
- It allocates memory in contiguous memory locations for its elements. It does not allocate any extra space/ memory for its elements. Hence there is no memory overflow or shortage of memory in arrays.
- Iterating the arrays using their index is faster compared to any other methods like linked list etc.
- It allows to store the elements in any dimensional array - supports multidimensional array.

❖ Disadvantages of Arrays

- It allows us to enter only fixed number of elements into it. We cannot alter the size of the array once array is declared. Hence if we need to insert more number of records than declared then it is not possible. We should know array size at the compile time itself.

- Inserting and deleting the records from the array would be costly since we add / delete the elements from the array, we need to manage memory space too.
- It does not verify the indexes while compiling the array. In case there is any indexes pointed which is more than the dimension specified, then we will get run time errors rather than identifying them at compile time.

❖ **Strings:** String declaration, string functions and string manipulation Program Structure Storage Class: Automatic, external and static variables.

1.String: Strings are defined as an array of characters. The difference between a character array and a string is the string is terminated with a special character '\0'.

int a[3];

❖ **Declaration of strings:** Declaring a string is as simple as declaring a one dimensional array. Below is the basic syntax for declaring a string.

char str_name[size]; char name[10]={'L','U','D','H','I','A','N','A'}

Initializing a String: A string can be initialized in different ways. We will explain this with the help of an example.

char name[5]={'R','A','H','U','L'};

// C program to illustrate strings

```
#include<stdio.h>

int main()

{

   // declare and initialize string
```

```
  char name[5]={‘R’,’A’,’H’,’U’,’L’};

   // print string

   printf("%c",str);

   return 0;

}
```

Output:

```
RAHUL
```

Second way to use String

// C program to illustrate strings

```
#include<stdio.h>

int main()

{

   // declare and initialize string

  char name[5]=”RAHUL”;

   // print string

   printf("%s",str);

    return 0;

}
```

Output:

```
RAHUL
```

Here (“ “)Double quotes is used while initializing the string instead of(‘ ‘)Single quotes.

❖ String Manipulations In C Programming Using Library Functions

To use String Handling function “string.h” Header file is use

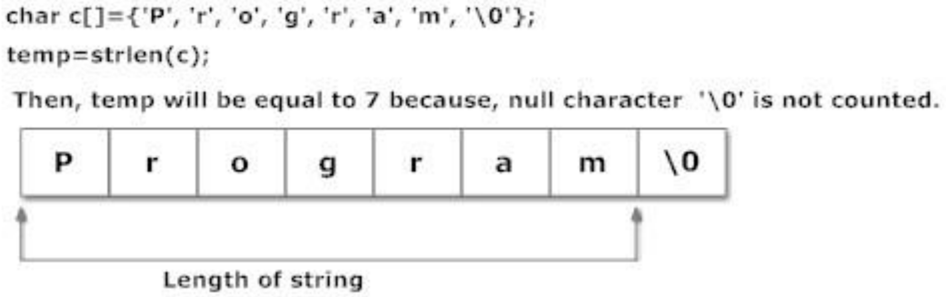
The list ofLibrary Functions is given below

Function	Work of Function
strlen()	Calculates the length of string
strcpy()	Copies a string to another string
strcat()	Concatenates(joins) two strings
strcmp()	Compares two string
strlwr()	Converts string to lowercase
strupr()	Converts string to uppercase

1)strlen(): The function takes a single argument, i.e, the string variable whose length is to be found, and returns the length of the string passed.

Syntax:

size_t strlen(const char *str);



```
char c[]={'P', 'r', 'o', 'g', 'r', 'a', 'm', '\0'};
temp=strlen(c);
```

Then, temp will be equal to 7 because, null character '\0' is not counted.

P	r	o	g	r	a	m	\0

Example:

```
#include <stdio.h>

#include <string.h>

int main()

{

   char a[20];


   printf("Enter string: ");

   gets(c);


   printf("Length of string a = %d \n",strlen(a));




   return 0;

}
```

Output

```
Enter string: rahul
```

```
Length of string a = 5
```

2)strcpy(): The strcpy() function copies the string pointed by source (including the null character) to the character array destination.This function returns character array destination.

Syntax:

char* strcpy(char* destination, const char* source);

Example

```
#include <stdio.h>

#include <string.h>


int main()

{

   char str1[10]= "awesome";

   char str2[10];


   strcpy(str2, str1);


printf("str2= %s",str2);


   return 0;

}
```

Output

str2=awesome

3)strcat(): It takes two arguments, i.e, two strings or character arrays, and stores the resultant concatenated string in the first string specified in the argument.

Syntax

char *strcat(char *dest, const char *src)

Example:

```
 #include <stdio.h>

 #include <string.h>

 int main()

 {

    char str1[5] = "rahul";

 char  str2[6] = "sharma";

  printf(“old  string is = %s”,str1);

 strcat(str1,str2);

printf(“new string is =%s”,str1);

    return 0;

 }
```

Output:

```
old  string is = rahul

new string is =rahulsharma
```

4)strcmp(): The strcmp() function takes two strings and return an integer.

The strcmp() compares two strings character by character. If the first character of two strings are equal, next character of two strings are compared. This continues until the corresponding characters of two strings are different or a null character '\0' is reached.

Return Value from strcmp()

Return Value	Remarks
0	if both strings are identical (equal)
negative	if the ASCII value of first unmatched character is less than second.
positive integer	if the ASCII value of first unmatched character is greater than second.

Example: C strcmp() function

```
#include <stdio.h>

#include <string.h>

int main()

{

    char str1[] = "abcd", str2[] = "abCd", str3[] = "abcd";

    int result;

    // comparing strings str1 and str2

    result = strcmp(str1, str2);

    printf("strcmp(str1, str2) = %d\n", result);

    // comparing strings str1 and str3
    result = strcmp(str1, str3);

    printf("strcmp(str1, str3) = %d\n", result);

    return 0;

}
```

Output

```
strcmp(str1, str2) = 32

strcmp(str1, str3) = 0
```

The first unmatched character between string str1 and str2 is third character. The ASCII value of 'c' is 99 and the ASCII value of 'C' is 67. Hence, when strings str1 and str2 are compared, the return value is 32.

When strings str1 and str3 are compared, the result is 0 because both strings are identical.

5)strupr(): The strupr( ) function is used to converts a given string to uppercase.

Syntax:

```
char *strupr(char *str);
```

Example:

```
// c program to demonstrate

// example of strupr() function.

#include<stdio.h>

#include<string.h>

int main()

{

   char str[ 5] = "hello";

      printf("%s\n", strupr (str));
```

```
    return  0;
}
```

Output:

```
HELLO
```

6)strlwr(): The strlwr( ) function is used to converts a given string to lowercase.

Syntax:

```
char *strlwr(char *str);
```

Example:

```
// c program to demonstrate
// example of strlwr() function.
#include<stdio.h>
#include<string.h>

int main()
{
    char str[ 5] = "HELLO";
        printf("%s\n", strlwr (str));
    return  0;
}
```

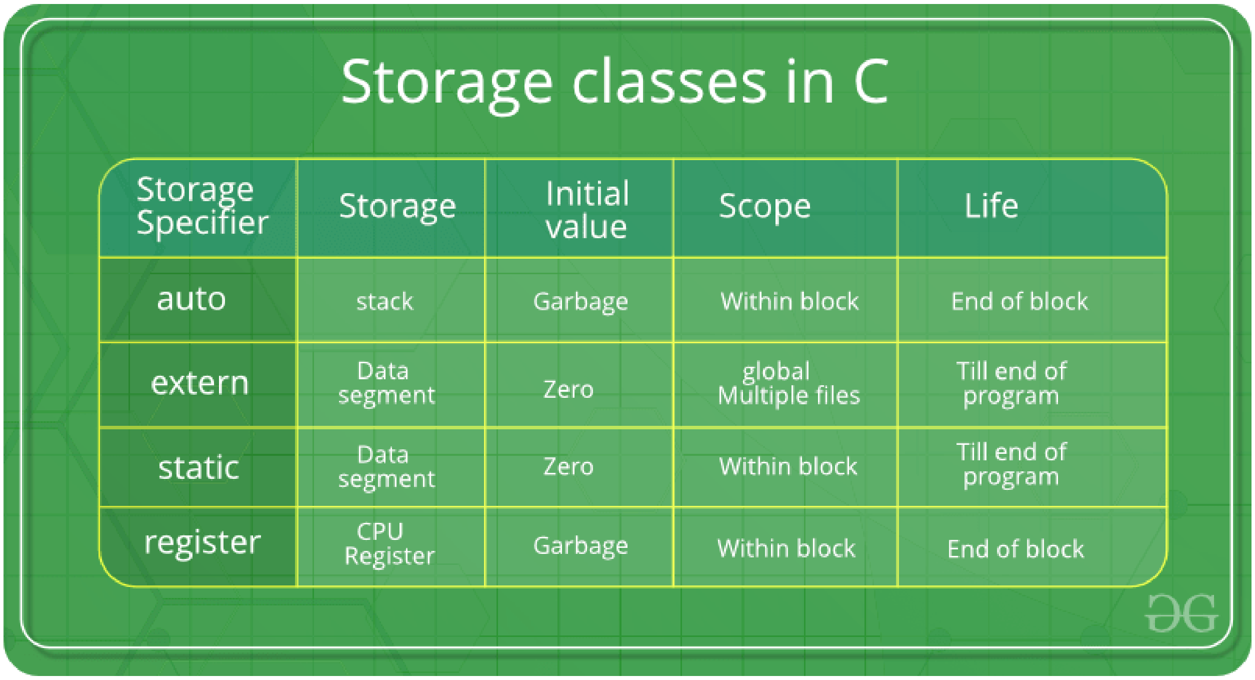
Output:

```
hello
```

❖ Structure Storage Class: Automatic, external and static variables

Structure Storage Class:

Storage Classes are used to describe the features of a variable/function. These features basically include the scope, visibility and life-time which help us to trace the existence of a particular variable during the runtime of a program.



Storage classes in C

Storage Specifier	Storage	Initial value	Scope	Life
auto	stack	Garbage	Within block	End of block
extern	Data segment	Zero	global Multiple files	Till end of program
static	Data segment	Zero	Within block	Till end of program
register	CPU Register	Garbage	Within block	End of block

1)auto : This is the default storage class for all the variables declared inside a function or a block. Hence, the keyword auto is rarely used while writing programs in C language. Auto variables can be only accessed within the block/function they have been declared and not outside them (which defines their scope).

Syntax

auto datatype variablename;

Example:

```
#include <stdio.h>

void main()

{
   auto int a = 32;


    printf("Value of the variable  adeclared as auto: %d\n", a);

}
```

Output

```
Value of the variable  a declared as auto:32
```

2)extern : Extern storage class simply tells us that the variable is defined elsewhere and not within the same block where it is used. Basically, the value is assigned to it in a different block and this can be overwritten/changed in a different block as well. Hence, the keyword extern is rarely used while writing programs in C language.

Syntax

```
extern datatype variablename;
```

Example:

```
#include <stdio.h>

void main()

{

   extern  int a ;


    printf("Value of the variable  a declared as extern: %d\n", a);

   a=34;

    printf("Value of the variable  adeclared as extern: %d\n", a);


}
```

Output

```
Value of the variable  adeclared as extern:0

Value of the variable  adeclared as extern:34
```

3)static: This storage class is used to declare static variables which are popularly used while writing programs in C language. Static variables have a property of preserving their value even after they are out of their scope! Hence, static variables preserve the value of their last use in their scope. So we can say that they are initialized only once and exist till the termination of the program. Hence, the keyword static is rarely used while writing programs in C language.

Syntax

```
static  datatype variablename;
```

Example:

```
#include <stdio.h>

static int a=10;

void main()

{

    static  int b=20 ;

       printf("value of a %d”,a);

        printf("value of b %d”,b);

}
```

Output

```
value of a10

value of b20
```

4)register: This storage class declares register variables which have the same functionality as that of the auto variables. The only difference is that the compiler tries to store these variables in the register of the microprocessor if a free register is available.This makes the use of register variables to be much faster than that of the variables stored in the memory during the runtime of the program. If a free register is not available, these are then stored in the memory only.

Syntax:

```
storage_class var_data_type var_name;

#include <stdio.h>

void main()

{

     register  int b=20 ;


       printf("value of a %d”,a);

        printf("value of b %d”,b);

}
```

Output

```
value of b20
```

UNIT -IV

- **Structures**

A structure is a user defined data type in C. A structure creates a data type that can be used to group items of possibly different types into a single type.

‘**struct**’ keyword is used to create a structure.

Declaration of Structure

A structure variable can either be declared with structure declaration or as a separate declaration like basic types.

Syntax

```
 struct structurename

{

Datatype variablename1;

Datatype variablename2;

.

.

Datatype variablename n;

}
```

Following is an example.

```
struct address

{

   char name[50];

   char street[100];

   char city[50];

   char state[20];

   int pin;

};
```

Initialization of Structure

Structure members **cannot be** initialized with declaration. For example the following C program fails in compilation.it always initialize into main() and user define() by using the object of structure.

Syntax

Struct structurename objectname;

access structure elements

Structure members are accessed using dot (.) operator.

Syntax

Objectname.variablename=value;

Example

```
#include<stdio.h>
struct student
{
int rollno;
char name[20];
};
void main()
{
 struct student  obj; //declaring structure object
printf("enter rollno and name\n");
scanf("%d%s",&obj.rollno,&obj.name);
  printf(" rollno  is %d \n",obj.rollno);
  printf(" name  is %s \n",obj.name);
```

}

Output

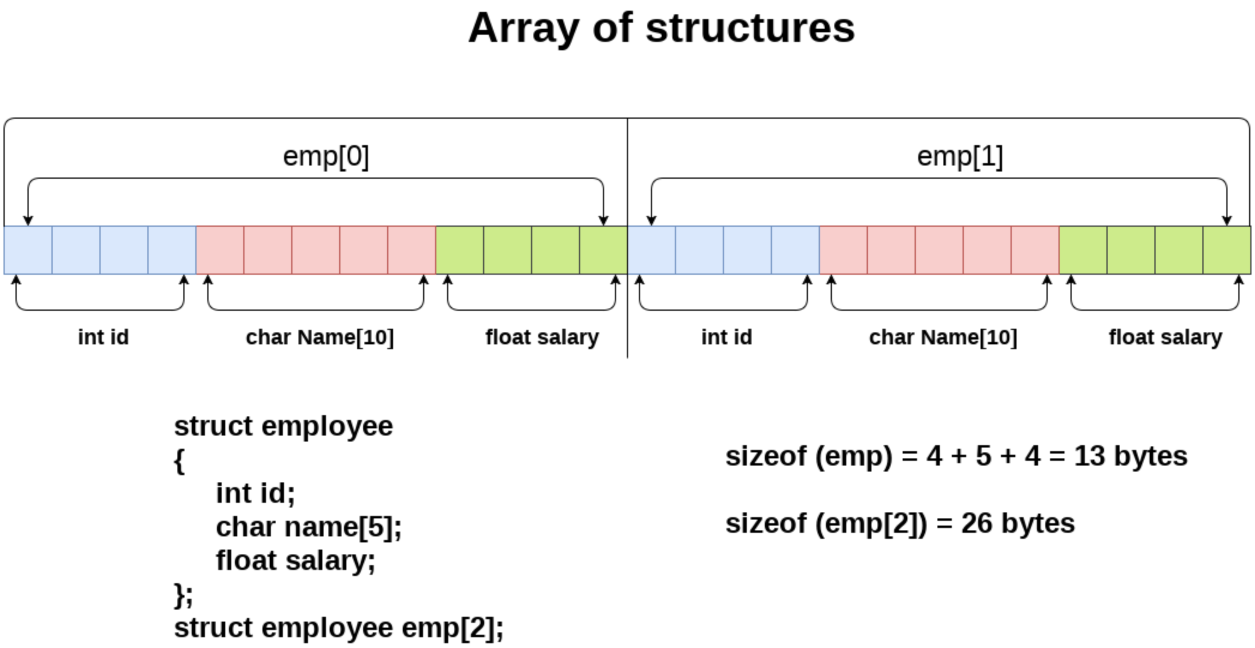
enter rollno and name

23 rahul

rollno is 23

name is rahul

2) Array of Structure : Array of structures in C can be defined as the collection of multiple structures variables where each variable contains information about different entities. The array of structures in C are used to store information about multiple entities of different data types. The array of structures is also known as the collection of structures.



```
struct employee
{
    int id;
    char name[5];
    float salary;
};
struct employee emp[2];
```

sizeof (emp) = 4 + 5 + 4 = 13 bytes

sizeof (emp[2]) = 26 bytes

Syntax

Struct structurename objectname[size];

Example

```
#include<stdio.h>
#include <string.h>
struct student{
int rollno;
char name[10];
};
int main(){
int i;
struct student st[5];
printf("Enter Records of 5 students");
for(i=0;i<5;i++){
printf("\nEnter Rollno:");
scanf("%d",&st[i].rollno);
printf("\nEnter Name:");
scanf("%s",&st[i].name);
}
printf("\nStudent Information List:");
for(i=0;i<5;i++){
printf("\nRollno:%d, Name:%s",st[i].rollno,st[i].name);
}
   return 0;
}
```

Output:

```
Enter Records of 5 students
Enter Rollno:1
Enter Name:Sonoo
Enter Rollno:2
Enter Name:Ratan
Enter Rollno:3
Enter Name:Vimal
Enter Rollno:4
Enter Name:James
Enter Rollno:5
Enter Name:Sarfraz


Student Information List:
Rollno:1, Name:Sonoo
Rollno:2, Name:Ratan
Rollno:3, Name:Vimal
Rollno:4, Name:James
Rollno:5, Name:Sarfraz
```

❖ Passing structure to function

- A structure can be passed to any function from main function or from any sub function.
- Structure definition will be available within the function only.
- It won’t be available to other functions unless it is passed to those functions by value or by address(reference).

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

```
#include <string.h>

struct student
{
            int id;

            float percentage;
};

void func(struct student record);


void func(struct student record)
{
            printf(" Id is: %d \n", record.id);

            printf(" Percentage is: %f \n", record.percentage);
}

int main()
{
            struct student record;

            record.id=1;
            record.percentage = 86.5;

            func(record);
            return 0;
}
```

Output

```
Id is: 1
Percentage is: 86.500000
```

❖ Structure using Pointer

Dot(.) operator is used to access the data using normal structure variable and arrow (->) is used to access the data using pointer variable.

Syntax:

Objectname->variablename=value;

Example

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

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

int main()
{
    int i;
    struct student record1 = {1, "Raju", 90.5};
    struct student *ptr;

    ptr = &record1;

        printf("Records of STUDENT1: \n");
        printf("  Id is: %d \n", ptr->id);
        printf("  Name is: %s \n", ptr->name);
        printf("  Percentage is: %f \n\n", ptr->percentage);

    return 0;
}
```

Output

```
Records of STUDENT1:
Id is: 1
Name is: Raju
Percentage is: 90.500000
```

❖ Union

Like Structures, union is a user defined data type. In union, all members share the same memory location.

A union is a special data type available in C that allows to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time

How is the size of union decided by compiler?

Size of a union is taken according the size of largest member in union.The Union keyword is used to declare the unions in C.

Syntax

```
 union structurename

{

Datatype variablename1;

Datatype variablename2;

.

.

Datatype variablename n;

};
```

Example

```
#include<stdio.h>

union student

{

int rollno;

char name[20];

};

void main()

{
 struct student  obj; //declaring structure object
printf(“enter rollno and name\n”);
scanf(“%d%s”,&obj.rollno,&obj.name);
  printf(“ rollno  is %d \n”,obj.rollno);
  printf(“ name  is %s \n”,obj.name);
}
```

BASIS OF COMPARISON	**STRUCTURE**	**UNION**
Basic	The separate memory location is allotted to each member of the 'structure'.	All members of the 'union' share the same memory location.

Declaration	struct struct_name{
	type element1;
	type element2;
	.
	.
	} variable1, variable2, ...;
		union u_name{
		type element1;
		type element2;
		.
		.
		} variable1, variable2, ...;

keyword	'struct'	'union'
Size	Size of Structure= sum of size of all the data members.	Size of Union=size of the largest members.
Store Value	Stores distinct values for all the members.	Stores same value for all the members.
At a Time	A 'structure' stores multiple values, of the different members, of the 'structure'.	A 'union' stores a single value at a time for all members.

Pointers: Understanding Pointers, Accessing the Address of a Variable, Declaration and Initialization of Pointer Variables, Accessing a Variable through its Pointer, Pointers and Arrays

❖ Pointers

Pointer variable must be declared before using it as we know in C Programming Language, every variable must be declared before using. Generally if we declare an integer type variable that hold a unique memory address from the computer system, we do it in C like below:

Int Variable_name;

Similarly, we can declare a pointer variable but adding an asterisk ('*' sign) after data type and before variable name. For that there are three type of pointer declaration style in C. They are:

1. int* variable_name;
2. int * variable_name;
3. int *variable_name;

The pointer in C language is a variable which stores the address of another variable. This variable can be of type int, char, array, function, or any other pointer. The size of the pointer depends on the architecture. However, in 32-bit architecture the size of a pointer is 2 byte.

Consider the following **example** to define a pointer which stores the address of an integer.

1. **int** n = 10;
2. **int*** p = &n;

❖ Declaring a pointer

The pointer in c language can be declared using * (asterisk symbol). It is also known as indirection pointer used to dereference a pointer.

1. **int** *a;//pointer to int
2. **char** *c;//pointer to char

❖ Initializing a pointer

To initialize the pointer variable &(Address Operator)is used.

int* p = &n;

❖ Accessing a Variable through its Pointer

```
#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);
   printf("%u", ptr);

   return 0;
}
```

Output

```
50
4104
```

Accessing a Variable without Pointer

```
#include <stdio.h>
int main()
{
   int  q;
   q = 50;


   printf("%u",&q);

   return 0;
}
```

Output

```
50
4104
```

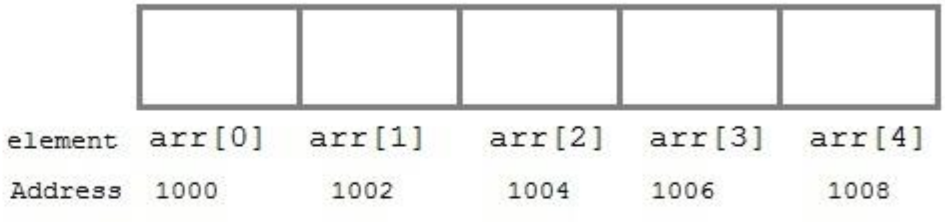
❖ Pointer and Arrays

When an array is declared, compiler allocates sufficient amount of memory to contain all the elements of the array. Base address i.e address of the first element of the array is also allocated by the compiler.

Suppose we declare an array arr,

```
int arr[5] = { 1, 2, 3, 4, 5 };
```

Assuming that the base address of arr is 1000 and each integer requires two bytes, the five elements will be stored as follows:



element	arr[0]	arr[1]	arr[2]	arr[3]	arr[4]
Address	1000	1002	1004	1006	1008

Here variable arr will give the base address, which is a constant pointer pointing to the first element of the array, arr[0]. Hence arr contains the address of arr[0] i.e 1000. In short, arr has two purpose - it is the name of the array and it acts as a pointer pointing towards the first element in the array.

arr is equal to &arr[0] by default

We can also declare a pointer of type int to point to the array arr.

```
int *p;
p = arr;
// or,
p = &arr[0];
```

❖ Pointer to Array

As studied above, we can use a pointer to point to an array, and then we can use that pointer to access the array elements. Let’s have an example,

```
#include <stdio.h>

int main()
{
   int i;
   int a[5] = {1, 2, 3, 4, 5};
   int *p = a;     // same as int*p = &a[0]
   for (i = 0; i < 5; i++)
   {
      printf("%d \t", *p);
      printf("address is %u:\n", p);
```

```
        p++;
    }


    return 0;
}
```

```
Output
1 address is :1000
2 address is :1001
3 address is :1002
4 address is :1003
5 address is :1004
```

Topic-3:
File Handling: File Operations, Processing a Data File.

❖ Why files are needed?

- When a program is terminated, the entire data is lost. Storing in a file will preserve your data even if the program terminates.
- If you have to enter a large number of data, it will take a lot of time to enter them all.

  However, if you have a file containing all the data, you can easily access the contents of the file using few commands in C.
- You can easily move your data from one computer to another without any changes.

❖ Types of Files

When dealing with files, there are two types of files you should know about:

1. Text files
2. Binary files

1. Text files

Text files are the normal .txt files that you can easily create using Notepad or any simple text editors.
When you open those files, you'll see all the contents within the file as plain text.
You can easily edit or delete the contents.
They take minimum effort to maintain, are easily readable, and provide least security and takes bigger storage space.

2. Binary files

Binary files are mostly the .bin files in your computer.
Instead of storing data in plain text, they store it in the binary form (0's and 1's).
They can hold higher amount of data, are not readable easily and provides a better security than text files.

❖ File Handling in C

In programming, we may require some specific input data to be generated several numbers of times. Sometimes, it is not enough to only display the data on the console. The data to be displayed may be very large, and only a limited amount of data can be displayed on the console, and since the memory is volatile, it is impossible to recover the programmatically generated data again and again. However, if we need to do so, we may store it onto the local file system which is volatile and can be accessed every time. Here, comes the need of file handling in C.
File handling in C enables us to create, update, read, and delete the files stored on the local file system through our C program. The following operations can be performed on a file.

- Creation of the new file
- Opening an existing file
- Reading from the file
- Writing to the file
- Deleting the file

❖ File Operations

Mode	Description
r	opens a text file in read mode
w	opens a text file in write mode
a	opens a text file in append mode
r+	opens a text file in read and write mode
w+	opens a text file in read and write mode
a+	opens a text file in read and write mode

❖ Functions for file handling/ Processing a data file

There are many functions in the C library to open, read, write, search and close the file. A list of file functions are given below:

No.	Function	Description
1	fopen()	opens new or existing file
2	fprintf()	write data into the file
3	fscanf()	reads data from the file
4	fputc()	writes a character into the file
5	fgetc()	reads a character from file
6	fclose()	closes the file
8	fputw()	writes an integer to file
9	fgetw()	reads an integer from file
10	ftell()	returns current position
11	rewind()	sets the file pointer to the beginning of the file

1)Opening File: **fopen()**
We must open a file before it can be read, write, or update. The fopen() function is used to open a file.
The **syntax** of the fopen() is given below.

1. **FILE** *fopen(**const char** * filename, **const char** * mode);

The fopen() function accepts two parameters:

- The file name (string). If the file is stored at some specific location, then we must mention the path at which the file is stored. For example, a file name can be like **"c://some_folder/some_file.ext"**.
- The mode in which the file is to be opened. It is a string.

We can use one of the following modes in the fopen() function.

Mode	**Description**
r	opens a text file in read mode
w	opens a text file in write mode
a	opens a text file in append mode
r+	opens a text file in read and write mode
w+	opens a text file in read and write mode
a+	opens a text file in read and write mode

The fopen function works in the following way.

- Firstly, It searches the file to be opened.
- Then, it loads the file from the disk and place it into the buffer. The buffer is used to provide efficiency for the read operations.
- It sets up a character pointer which points to the first character of the file.

Consider the following **example** which opens a file in write mode.

```
#include<stdio.h>
void main( )
{
FILE *fp ;
char ch ;
fp = fopen("file_handle.c","r") ;
while ( 1 )
{
ch = fgetc ( fp ) ;
if ( ch == EOF )
break ;
printf("%c",ch) ;
}
fclose (fp ) ;
}
```

Output

The content of the file will be printed.

```
#include;
void main( )
```

```
{
FILE *fp; // file pointer
char ch;
fp = fopen("file_handle.c","r");
while ( 1 )
{
ch = fgetc ( fp ); //Each character of the file is read and stored in the character file.
if ( ch == EOF )
break;
printf("%c",ch);
}

fclose (fp );
}
```

2)Closing File: **fclose()**

The fclose() function is used to close a file. The file must be closed after performing all the operations on it.

The **syntax** of fclose() function is given below:

1. **int** fclose(**FILE** *fp);

3)fprintf(): The fprintf() function is used to write set of characters into file. It sends formatted output to a stream.

Syntax:

1. **int** fprintf(**FILE** *stream, **const char** *format [, argument, ...])

Example:

```
#include <stdio.h>
main()
{
  FILE *fp;
  fp = fopen("file.txt", "w");//opening file
  fprintf(fp, "Hello file by fprintf...\n");//writing data into file
  fclose(fp);//closing file
}
```

4)fscanf(): The fscanf() function is used to read set of characters from file. It reads a word from the file and returns EOF at the end of file.

Syntax:

1. int fscanf(FILE *stream, const char *format [, argument, ...])

Example:

```
#include <stdio.h>
main()
{
  FILE *fp;
  char buff[255];//creating char array to store data of file
  fp = fopen("file.txt", "r");
  while(fscanf(fp, "%s", buff)!=EOF){
  printf("%s ", buff );
  }
  fclose(fp);
}
```

Output:

```
Hello file by fprintf...
```

5)fgetc():fgetc() is used to obtain input from a file single character at a time. This function returns the number of characters read by the function. It returns the character present at position indicated by file pointer.

Syntax:
int fgetc(FILE *pointer)
pointer: pointer to a FILE object that identifies
the stream on which the operation is to be performed.

```
// C program to illustate fgetc() function
#include <stdio.h>

int main ()
{
   // open the file
   FILE *fp = fopen("test.txt","r");

   // Return if could not open file
   if (fp == NULL)
     return 0;

   do
   {
      // Taking input single character at a time
      char c = fgetc(fp);
```

```
        // Checking for end of file
        if (feof(fp))
            break ;

        printf("%c", c);
    }  while(1);

    fclose(fp);
    return(0);
}
```

Output:
The entire content of file is printed character by character till end of file. It reads newline character as well.

5)fputc():fputc() is used to write a single character at a time to a given file. It writes the given character at the position denoted by the file pointer and then advances the file pointer.
This function returns the character that is written in case of successful write operation else in case of error EOF is returned.

Syntax:

int fputc(int char, FILE *pointer)

char: character to be written.

This is passed as its int promotion.

pointer: pointer to a FILE object that identifies the

stream where the character is to be written.

Example:

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

```
int main () {
  FILE *fp;
  int ch;
fp = fopen("file.txt", "w+");
  for( ch = 33 ; ch <= 100; ch++ )
{
    fputc(ch, fp);
  }
  fclose(fp);
  return(0);
}
```