Unit - I:

Introduction to Programming: Problem analysis, Introduction to algorithms, flow charts, structured programming and modular programming.
Overview of C: History of C, Importance of C, Basic Structure of C program, Writing and Executing a C Program. Constants, Variables and Data types, Input/output statements, Operators and Expressions.

1. Introduction

A computer is a very powerful and versatile machine capable of performing a multitude of different tasks, yet it has no intelligence or thinking power. The intelligence Quotient (I.Q) of a computer is zero. A computer performs many tasks exactly in the same manner as it is told to do. This places responsibility on the user to instruct the computer in a correct and precise manner, so that the machine is able to perform the required job in a proper way. A wrong or ambiguous instruction may sometimes prove disastrous.

In order to instruct a computer correctly, the user must have clear understanding of the problem to be solved. A part from this he should be able to develop a method, in the form of series of sequential steps, to solve it. Once the problem is well-defined and a method of solving it is developed, then instructing the computer to solve the problem becomes relatively easier task.

Thus, before attempt to write a computer program to solve a given problem. It is necessary to formulate or define the problem in a precise manner. Once the problem is defined, the steps required to solve it, must be stated clearly in the required order.

Operating Systems

The set of instructions which resides in the computer and governs the system are called operating systems, without which the machine will never function. They are the medium of communication between a computer and the user. DOS, Windows, Linux, UNIX etc. are Operating Systems.

Computer Languages

They are of three types –

- Machine language (Low level language)

- Assembly language
- User Oriented language (Higher level language)

Machine language depends on the hard ware and comprises of 0 and 1. This is tough to write as one must know the internal structure of the computer. At the same time assembly language makes use of English like words and symbols. With the help of special programs called Assembler, assembly language is converted to machine oriented language. Here also a programmer faces practical difficulties. To overcome this hurdles user depends on higher level languages, which are far easier to learn and use. To write programs in higher level language, programmer need not know the characteristics of a computer. Here he uses English alphabets, numerals and some special characters.

Some of the higher level languages are FORTRAN, BASIC, COBOL, PASCAL, C, C++, ADA etc. We use C to write programs. Note that higher level languages cannot directly be followed by a computer. It requires the help of certain software to convert it into machine coded instructions. This software is called **Compiler, Interpreter, and Assembler**. **The major difference between a compiler and an interpreter** is that compiler compiles the user's program into machine coded by reading the whole program at a stretch where as Interpreter translates the program by reading it line by line.

Problem Analysis & Solving:

It can be said that whatever activity a human being or machine do for achieving a specified objective comes under problem solving. To make it more clear, let us see some other examples.

Example1: If you are watching a news channel on your TV and you want to change it to a sports channel, you need to do something i.e. move to that channel by pressing that channel number on your remote. This is a kind of problem solving.

Example 2: One Monday morning, a student is ready to go to school but yet he/she has not picked up those books and copies which are required as per time-table. So here picking up books and copies as per time-table is a kind of problem solving.

Example 3: Some students in a class plan to go on picnic and decide to share the expenses among them. So calculating total expenses and the amount an individual has to give for picnic is also a kind of problem solving.

Now, broadly we can say that problem is a kind of barrier to achieve something and problem solving is a process to get that barrier removed by performing some sequence of activities. A computer cannot solve a problem on its own. One has to provide step by step solutions of the problem to the computer**.** In fact, the task of problem solving is not that of the computer. It is the programmer who has to write down the solution to the problem in terms of simple operations which the computer can understand and execute. In order to solve a problem by the computer, one has to pass though certain stages or steps. They are:

1. Understanding the problem.
2. Analyzing the problem.
3. Developing the solution.
4. Coding and implementation.

1. Understanding the problem: Here we try to understand the problem to be solved in totally. Before with the next stage or step, we should be absolutely sure about the objectives of the given problem.

2. Analyzing the problem: After understanding thoroughly the problem to be solved, we look different ways of solving the problem and evaluate each of these methods. The idea here is to search an appropriate solution to the problem under consideration. The end result of this stage is a broad overview of the sequence of operations that are to be carries out to solve the given problem.

3. Developing the solution: Here the overview of the sequence of operations that was the result of analysis stage is expanded to form a detailed step by step solution to the problem under consideration.

4. Coding and implementation: The last stage of the problem solving is the conversion of the detailed sequence of operations in to a language that the computer can understand. Here each step is converted to its equivalent instruction or instructions in the computer language that has been chosen for the implantation.

Algorithm & Flowchart:

Algorithm and flowchart are the powerful tools for learning programming. An algorithm is a step-by-step analysis of the process, while a flowchart explains the steps of a program in a graphical way. Algorithm and flowcharts helps to clarify all the steps for solving the problem.

Algorithm

The word "algorithm" relates to the name of the mathematician Al-khowarizmi, which means a procedure or a technique. Software Engineer commonly uses an algorithm for planning and solving the problems. An algorithm is a sequence of steps to solve a particular problem or algorithm is an ordered set of unambiguous steps that produces a result and terminates in a finite time. **In other words,** a set of sequential steps usually written in Ordinary Language to solve a given problem is called A**lgorithm**.

It may be possible to solve to problem in more than one ways, resulting in more than one algorithm. The choice of various algorithms depends on the factors like reliability, accuracy and easy to modify. The most important factor in the choice of algorithm is the time requirement to execute it, after writing code in High-level language with the help of a computer. The algorithm which will need the least time when executed is considered the best.

Steps involved in algorithm development

An algorithm can be defined as **"a complete, unambiguous, finite number of logical steps for solving a specific problem "**

Step1- Define input: For an algorithm, there are quantities to be supplied called input and these are fed externally. The input is to be identified first for any specified problem.

Step2- Outline the algorithm operations: All the calculations to be performed in order to lead to output from the input are to be identified in an orderly manner.

Step3- Process the finiteness, definiteness and Effectiveness of algorithm: If we go through the algorithm, then for all cases, the algorithm should terminate after a finite number of steps.

Step4-Define output: From an algorithm, at least one quantity is produced, called for any specified problem

Properties of an Algorithm:

1. **Finiteness:** An algorithm must always terminate after a finite number of steps. It means after every step one reach closer to solution of the problem and after a finite number of steps algorithm reaches to an end point.

2. **Definiteness:** Each step of an algorithm must be precisely defined. It is done by well thought actions to be performed at each step of the algorithm. Also the actions are defined unambiguously for each activity in the algorithm.
3. **Input:** Any operation you perform need some beginning value/quantities associated with different activities in the operation. So the value/quantities are given to the algorithm before it begins.
4. **Output:** One always expects output/result (expected value/quantities) in terms of output from an algorithm. The result may be obtained at different stages of the algorithm. If some result is from the intermediate stage of the operation, then it is known as intermediate result and result obtained at the end of algorithm is known as end result. The output is expected value/quantities always have a specified relation to the inputs.
5. **Effectiveness:** Algorithms to be developed/written using basic operations. Actually operations should be basic, so that even they can in principle be done exactly and in a finite amount of time by a person, by using paper and pencil only.

Any algorithm should have all these five properties otherwise it will not fulfill the basic objective of solving a problem in finite time. As you have seen in previous examples, every step of an algorithm puts you closer to the solution.

Example

1. Suppose we want to find the average of three numbers, the algorithm is as follows

    Step 1: Read the numbers a, b, c

    Step 2: Compute the sum of a, b and c

    Step 3: Divide the sum by 3

    Step 4: Store the result in variable d

    Step 5: Print the value of d

    Step 6: End of the program

2. Write an algorithm to calculate the simple interest using the formula Simple interest = P*N* R/100. Where P is principle Amount, N is the number of years and R is the rate of interest.

    Step 1: Read the three input quantities’ P, N and R.

    Step 2: Calculate simple interest as Simple interest = P* N* R/100

Step 3: Print simple interest.

Step 4: Stop.

Advantages of algorithm:

- It is a step-wise representation of a solution to a given problem, which makes it easy to understand.
- An algorithm uses a definite procedure.
- It is not dependent on any programming language, so it is easy to understand for anyone even without programming knowledge.
- Every step in an algorithm has its own logical sequence so it is easy to debug.

Flowchart

The first design of flowchart goes back to 1945 which was designed by **John von Neumann**. Unlike an algorithm, Flowchart uses different symbols to design a solution to a problem. It is another commonly used programming tool. By looking at a Flowchart one can understand the operations and sequence of operations performed in a system. Flowchart is often considered as a blueprint of a design used for solving a specific problem.

A **flow chart is** a step by step diagrammatic representation of the logic paths to solve a given problem or A flowchart is visual or graphical representation of an algorithm. The flowcharts are pictorial representation of the methods to be used to solve a given problem and help a great deal to analyze the problem and plan its solution in a systematic and orderly manner. A flowchart when translated in to a proper computer language, results in a complete program.

Advantages of Flowcharts

1. The flowchart shows the logic of a problem displayed in pictorial fashion which felicitates easier checking of an algorithm.
2. The Flowchart is good means of communication to other users. It is also a compact means of recording an algorithm solution to a problem.
3. The flowchart allows the problem solver to break the problem into parts. These parts can be connected to make master chart.

4. The flowchart is a permanent record of the solution which can be consulted at a later time.

Symbols used in Flow-Charts:

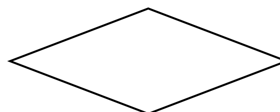
The symbols that we make use while drawing flowcharts as given below are as per conventions followed by International Standard Organization (ISO).

a. **Oval**: Rectangle with rounded sides is used to indicate either START/STOP of the program.

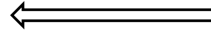
b. **Input and output indicators**: Parallelograms are used to represent input and output operations. Statements like INPUT, READ and PRINT are represented in these Parallelograms.

c. **Process Indicators**: - Rectangle is used to indicate any set of processing operation such as for storing arithmetic operations.

d. **Decision Makers**: The diamond is used for indicating the step of decision making and therefore known as decision box. Decision boxes are used to test the conditions or ask questions and depending upon the answers, the appropriate actions are taken by the computer. The decision box symbol is



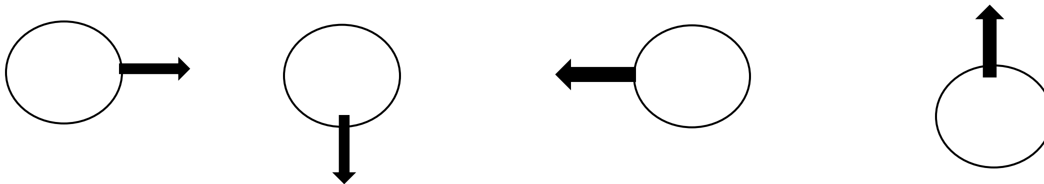
e. **Flow Lines**: Flow lines indicate the direction being followed in the flowchart. In a Flowchart, every
line must have an arrow on it to indicate the direction. The arrows may be in any direction



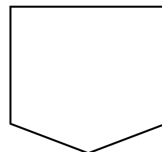
f. **On- Page connectors**: Circles are used to join the different parts of a flowchart and these circles are

called on-page connectors. The uses of these connectors give a neat shape to the flowcharts. In a

complicated problems, a flowchart may run in to several pages. The parts of the flowchart on different pages are to be joined with each other. The parts to be joined are indicated by the circle.



g. **Off-page connectors**: This connector represents a break in the path of flowchart which is too large to fit on a single page. It is similar to on-page connector. The connector symbol marks where the algorithm ends on the first page and where it continues on the second.



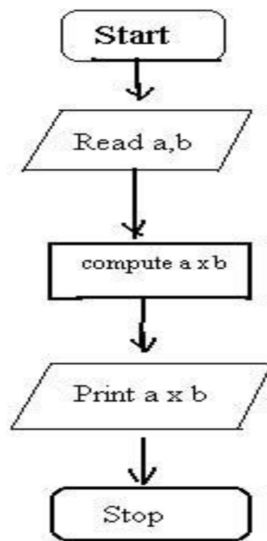
Example: Consider a problem of multiplying two numbers

Algorithm

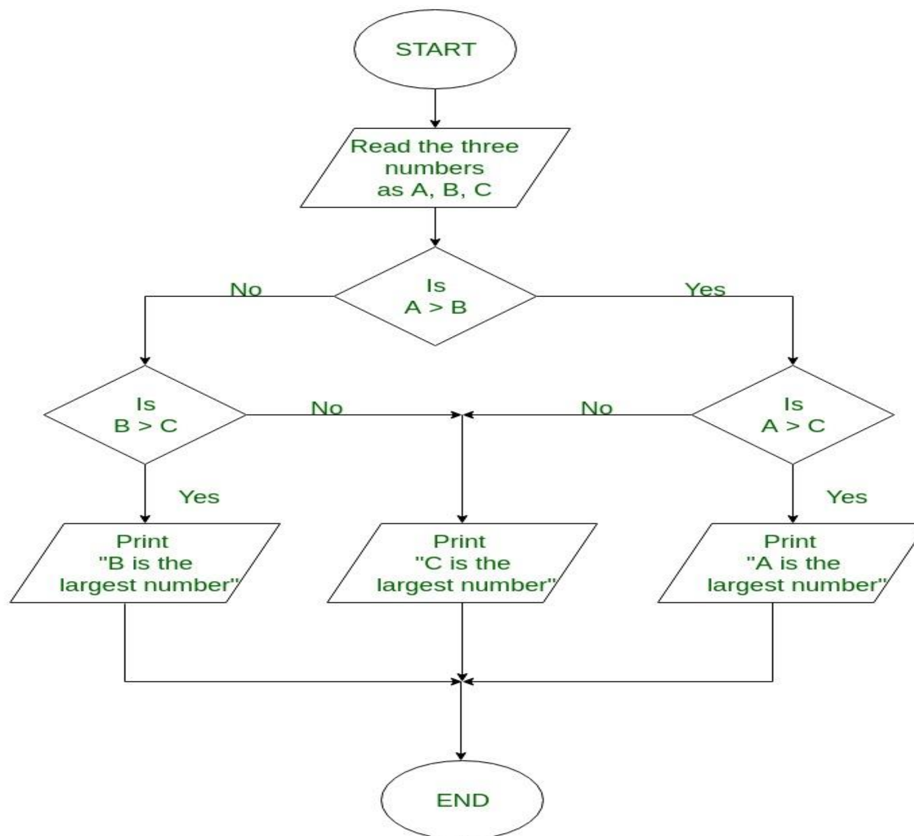
Step1: Input numbers as a and b.

Step2: Find the product of a and b.

Step3: Print the result.



Example: flowchart to find the biggest of three numbers



Differences between Algorithm and Flowchart

Algorithm	Flowchart
1. A method of representing the step-by-step logical procedure for solving a problem. 2. It contains step-by-step English descriptions, each step representing a particular operation leading to solution of problem. 3. These are particularly useful for small problems. 4. For complex programs, algorithms prove to be Inadequate.	1. Flowchart is diagrammatic representation of an algorithm. It is constructed using different types of boxes and symbols. 2. The flowchart employs a series of blocks and arrows, each of which represents a particular step in an algorithm. 3. These are useful for detailed representations of complicated programs. 4. For complex programs, Flowcharts prove to be adequate

Structured programming & Modular Programming:

➞**Modular programming** means separating your programs into separate functional units (modules). Each module does a well-defined task and the modules are called by one-another as needed. Typically, the state of each module is encapsulated and only supposed to be altered by the functions from that module.

➞**Structured programming** is a technique where you use subprograms (functions) and control-structures (like if-statements or loops) to structure the control-flow of your programs instead of doing so by liberal use of go to. If you learn programming today, you won't ever have learned how to write unstructured programs.

Structured programming:

- Structured programming is a subset of procedural programming that enforces a logical structure on the program being written to make it more efficient and easier to understand and modify. Certain

languages such as Ada, Pascal, and dBASE are designed with features that encourage or enforce a logical program structure.

- Structured programming frequently employs a **top-down design** model, in which developers map out the overall program structure into separate subsections. A defined function or set of similar functions is coded in a separate module or submodule, which means that code can be loaded into memory more efficiently and that modules can be reused in other programs. After a module has been tested individually, it is then integrated with other modules into the overall program structure.
- Almost any language can use structured programming techniques to avoid common pitfalls of unstructured languages.
- Unstructured programming must rely upon the discipline of the developer to avoid structural problems, and as a consequence may result in poorly organized programs.
- Most modern procedural languages include features that encourage structured programming. Object-oriented programming (OOP) can be thought of as a type of structured programming, uses structured programming techniques for program flow, and adds more structure for data to the model.

Modular Programming:

- Modular programming is the process of subdividing a computer program into separate sub-programs (Modules).
- A module is a separate software component. It can often be used in a variety of applications and functions with other components of the system. Similar functions are grouped in the same unit of programming code and separate functions are developed as separate units of code so that the code can be reused by other applications.
- Modules in modular programming enforce logical boundaries between components and improve maintainability. They are incorporated through interfaces. They are designed in such a way as to minimize dependencies between different modules. Teams can develop modules separately and do not require knowledge of all modules in the system.

- Each and every modular application has a version number associated with it. This provides developers flexibility in module maintenance. If any changes have to be applied to a module, only the affected subroutines have to be changed. This makes the program easier to read and understand.
- Modular programming has a main module and many auxiliary modules. The main module is compiled as an executable (EXE), which calls the auxiliary module functions. Auxiliary modules exist as separate executable files, which load when the main EXE runs. Each module has a unique name assigned in the PROGRAM statement. Function names across modules should be unique for easy access if functions used by the main module must be exported
- Languages that support the module concept are IBM Assembler, COBOL, RPG, FORTRAN, Morpho, Zonnon and Erlang, among others.

Difference between Structured programming and Modular programming

Modular programming means separating your programs into separate functional units (modules). Each module does a well-defined task and the modules are called by one-another as needed. Typically, the state of each module is encapsulated and only supposed to be altered by the functions from that module.

On the other hand, structured programming is a technique where you use subprograms (functions) and control-structures (like if-statements or loops) to structure the control-flow of your programs instead of doing so by liberal use of go to. If you learn programming today, you won't ever have learned how to write unstructured programs.

Introduction to C: -

C is a programming language developed at AT & T's Bell laboratories of USA in 1972, it was designed and developed by **"Dennis Ritchie**". C was evolved from ALGOL, BCPL and B, and it uses many concepts from these languages and added the concept of data type s and other powerful features.

Since it was developed along with the UNIX operating system, it is strongly associated with UNIX. The UNIX operating system was also developed at Bell Laboratories was coded almost entirely in C.

The various stages in evolution of C language are as follows:

Year	Language	Developed By	Remarks
1960	ALGOL	International Committee	too general, too abstract
1963	CPL	Cambridge University	Hard to learn, difficult to implement
1967	BCPL	Martin Richards at Cambridge University	Too specific
1970	B	Ken Thompson at AT & t Bell Labs	Too specific
1972	**C**	***Dennis Ritchie* at AT & T bell labs**	General purpose structured programming language.

Features of C Language: C is the widely used language. Below are some of the Features of C Programming

1. **Low Level Features:**
   - C Programming provides low level features that are generally provided by the Lower level languages. C is Closely Related to Lower Level Language such as "**Assembly Language** ".
   - **It is easier to write assembly language codes in C programming.**
2. **Portability:**
   - C Programs are portable i.e they can be run on any Compiler with Little or no Modification
   - Compiler and Preprocessor make it Possible for C Program to run it on Different PC
3. **Powerful**
   - Provides Wide verity of 'Data Types '
   - Provides Wide verity of 'Functions'

- Provides useful Control & Loop Control Statements

4. **Bit Manipulation:**
   - C Programs can be manipulated using bits. We can perform different operations at bit level. We can manage memory representation at bit level. (Eg. We can use Structure to manage Memory at Bit Level)
   - It provides wide verity of bit manipulation Operators. We have bitwise operators to manage Data at bit level.
5. **High Level Features**:
   - It is more User friendly as compare to previous languages. Previous languages such as BCPL, Pascal and other programming languages never provide such great features to manage data.
   - Previous languages have their **pros and cons** but C Programming collected all useful features of previous languages thus C become **more effective language**.
6. **Modular programming** is a software design technique that increases the extent to which software is composed of separate parts, called **modules**. C Program Consist of Different Modules that are integrated together to form complete program
7. **Efficient Use of Pointers**: Pointers has direct access to memory. C Supports efficient use of pointer.
8. **It is a robust language** with rich set of built-in functions and operators that can be used to write any complex program.
9. Programs Written in C are efficient and fast. This is due to its variety of data type and powerful operators and It is **loosely-typed** language.
10. Another important feature of C program is its **ability to extend itself**. A C program is basically a collection of functions that are supported by C library. We can also create our own function and add it to C library.

Importance of 'C' language:

C language is a famous programming language due to its qualities. Some qualities are:

1. It is robust language whose rich setup of built in functions and operator can be used to write any complex program.

2. Program written in C are efficient due to several variety of data types and powerful operators.
3. The C compiler combines the capabilities of an assembly language with the feature of high level language. Therefore, it is well suited for writing both system software and business package.
4. There are only 32 keywords; several standard functions are available which can be used for developing program.
5. C is portable language; this means that C programs written for one computer system can be run on another system, with little or no modification.
6. C language is well suited for structured programming; this requires user to think of a problem in terms of function or modules or block. A collection of these modules make a program debugging and testing easier.
7. C language has its ability to extend itself. A c program is basically a collection of functions that are supported by the C library. We can continuously add our own functions to the library with the availability of the large number of functions.
8. C is a **free-form language**; we can group statements together in one line.

Structure of C Program: - C program can be viewed as a group of building blocks called functions. A function is a subroutine that may include one or more statements designed to perform a specific task. To write a C program, we first create functions and then put them together. A C program may contain one or more sections.

```
Documentation section
Link Section
Definition Section
Global declaration Section
main ( ) function section
{
        Declaration part;
        Executable part
}
Sub-programming section

Function -1       }
Function-2        }
:                 }    User-defined functions
:                 }
```

Function -n

Documentation section: - It provides the details about the program like name of the program, author name, date when it is created and when it is modified. This section consists of a set of comment lines to provide all these details.

Comments are two types: i. single line comments ii. Multi-line comments

Single line comment: It is supported by the symbol //. The starting of the comment is with the symbol //.

Multi-line comment: This is used the symbol /* and */. The starting of the comment is with /* and ending with */.

Eg: // PROGRAM FOR ADDITION OF TWO NUMBERS □Single line comment

/* PROGRAM FOR ADDITION □Multi-line comment
TWO NUMBERS */

Linking Section: - It provides instructions to the compiler to link the specified files from the standard C library. It is accomplished by the pre-processor directive: *#include* <file name>

Eg: #include <stdio.h>

Definition Section: - this section defines all symbolic constants and macros by using a compiler directive *#define*. Symbolic constants appear in capital letters, and these are the values which don't change during the execution of the program.

Eg: #define PI 3.14

Global declaration Section: - In this section we can declare some variables that are used in more than one function. Such variables are called as global variables. It also declares the user-defined functions. Global declaration is outside of all the functions.

Eg: int a, b; □ where a, b are global variables
int add(int, int); □add function declaration or prototyping of add function.

main( ) function:- Every C program must contain at least one function , and that should be the main( ) function. If a program contain more functions, then one of those functions should be the main( ) function. Main( ) function consists of two parts:

Declaration part: Declaration part declares all the variables and user-defined functions which are used in executable part.

Executable part*:* There is at least one statement in executable part. And it consists of the statements which are towards the solution of the given program.

These two parts must appear in between opening brace ({) and closing brace (}) of the main ( ) function section. Always program execution begins at the opening brace {and ends at the closing brace} of the main ( ) function.

Sub-program section: - The sub-program section contains all user defined functions that are called in the main function.

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

Example1: program for finding the area of a circle.

```
// PROGRAM FOR FINDING THE AREA OF A CIRCLE          ? Documentation section
#include <stdio.h>                                   ? Link section
#include <conio.h>
#define    PI    3.14                                ? definition section(symbolic constant)
void main( )                                         ? main ( ) function section
{
        float r, area;                               ? Declaration part

        clrscr( );                  }
        printf("Enter radius ");    }
        scanf("%f", &r);            }
        area = PI * r * r;          }  ? Executable part
        printf(" Area = %f", area): }

        getch( );
}
```

Here in this example program, it doesn't have global declaration section and sub-program section. These are optional, based on the user requirement one can use these sections, but main function is must. The actual execution of the program starts from the main function only.

Example 2: Displaying Hello World

```
#include <stdio.h>
int main()
{
   /* printf function displays the content that is
    * passed between the double quotes.
    */
   printf("Hello World");
   return 0;
}
```

Output:

```
Hello World
```

Writing Executing a C Program: Step by Step Execution of C Program

Step 1: Edit

1. This is First Step i.e **Creating and Editing Program**.
2. First Write C Program using Text Editor, such as Turbo C++, **Borland C/C++ 3.0**, Notepad++ etc.
3. Save Program by using **.C Extension**. Ex: sample.c
4. File Saved with .C extension is called "**Source Program** ".

Step 2: Compiling source code (Alt + F9)

1. For compiling source program, we use Alt + F9 command. Whenever we press **Alt + F9**, the source file is going to be submitted to the Compiler.
2. On receiving a source file, the compiler first checks for the Errors.
3. If there are any Errors then compiler returns List of Errors, if there are no errors then the source code is converted into **object code** and stores it as file with **.obj** extension.
4. Then the object code is given to the **Linker**. The Linker combines both the **object code** and specified **header file** code and generates an **Executable file** with **.exe** extension.

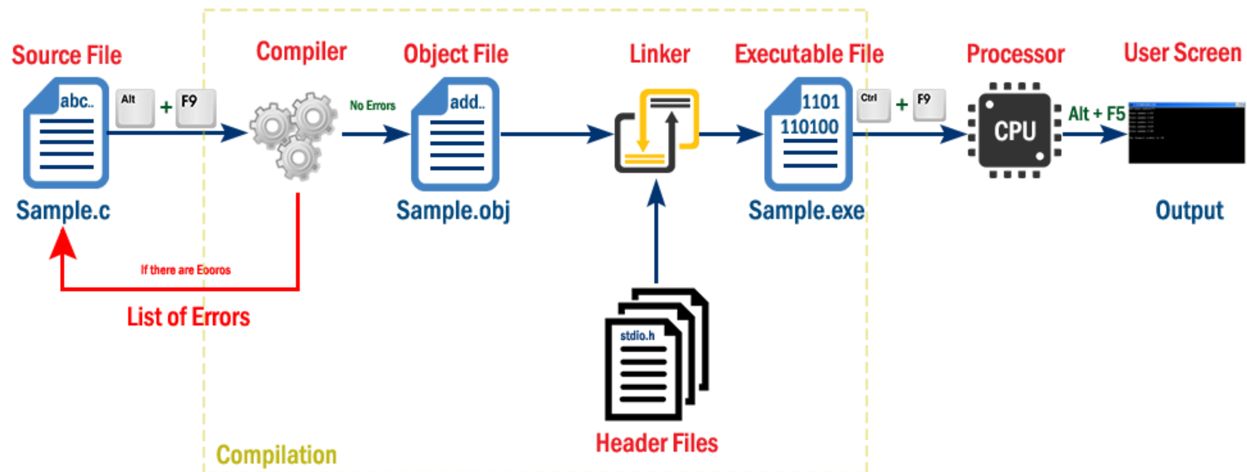
Step 3: Executing/Running the file (Ctrl + F9)

1. After completing compilation successfully, an executable file is created with **.exe** extension. The processor can understand this **.exe** file content so that it can perform the task specified in the source file.
2. We use a shortcut key **Ctrl + F9** to run a C program.
3. Whenever we press **Ctrl + F9**, the **.exe** file is submitted to the **CPU**. On receiving **.exe** file, **CPU** performs the task according to the instruction written in the file. The result generated from the execution is placed in a window called **User Screen**.

Step 4: Checking Results (Alt + F5)

After running the program, the result is placed into **User Screen**. Just we need to open the User Screen to check the result of the program execution. We use the shortcut key **Alt + F5** to open the User Screen and check the result.

The following diagram illustrates the compilation and execution process of a C program



The file which contains c program instructions in high level language is said to be source code. Every c program is a source file.

C Character Set: -

Like any other language, C has its own vocabulary and grammar. The characters in C are grouped into the following categories.

1. Letters.
2. Digits.
3. Special Symbols.
4. White Spaces.

Letters: Uppercase letters : A-Z
Lowercase letter : a-z

Digits: All Decimal digits 0-9

Special Symbols: There are so many special symbols, some of them are:

$-Dollar Sign	(-Left Parenthesis
%-Percent	)-Right Parenthesis
^-caret	[-Left Bracket
*-asterisk	{-Right Brace.
#-numeric sign	

White Spaces: Blank Space, New Line (\n), Horizontal tab (\t), Form feed, Carriage return etc are white space characters. White spaces may be used to separate words, but are prohibited between characters if keywords and identifiers.

C Tokens: -

In a Passage of text, individual words and punctuation marks are called as tokens.
In a C program, the smallest individual units are known as C tokens. The following are different categories of C tokens.

1) Keywords 2) constants 3) Identifiers 4) Operators.

1) Keywords: Every C word is classified as either a keyword or an identifier Keywords are also called as Reserved Words, which are having fixed meaning and these meanings cannot be changed. Keywords are used as basic building blocks for program Statements. All keywords must be written in lowercase letters. ANSI C supports 32 keywords.
Eg: auto, break, double, for, int, short, void, static, volatile, switch, while, union etc. are some keywords.

2) Constants: Constant refers to a fixed value that does not change during the execution of a program .C language support several types of constants.
1) Numeric constants 2) Character constants.

1) ***Numeric constants***: The quantity which can be expressed in numbers is numeric constant.
Eg: 123, -86, 3.14, 238.42

Numeric constants are expressed in two types.
i) Integer constants: It refers to a sequence of digits. There are three types of integers.

a) **Decimal Integers:** consists of a set of digits 0 through 9, preceded by an optional + or –
Eg: 823,28, -32486, +5.
NOTE: Spaces, commas and non-digit characters are not permitted between digits.
Eg: Rs 1000, $1000,25,538, are invalid.
b) **Octal Integers**: These are any combination of digits from the set 0 through 7, with a leading 0.
Eg: 037,0,0435.
c) **HexaDecimal Integers**: sequence of digits preceded by ox or OX from the set 0 through 9 and a through f.
eg: 0X5,0XAB, ox5D, ox28f.

NOTE: We rarely use octal and hexadecimal numbers in programming.

ii) Real Constants: The quantities, such as distances, heights, temperatures, prices and so on, are represented by numbers containing fractional parts. Such number are called real (or floating point) constants.
Eg: 0.75, +28.923, 128, 0.5.
Real constants are represented by using:
a) Decimal Notation b) Scientific Notation (Exponential Notation)
Decimal Notation: This notation is in the form of whole number followed by a decimal point and the fractional part. It is possible to omit digits before the decimal point or digits after the decimal point.

Scientific Notation: A real Number may also be expressed in exponential Notation. It is in the form:

Mantissa e exponent.

Eg: The value 215.65 may be written as 2.1565e2 in exponential notation.e2 means multiply by 10^2.
0.65e4, 12e2,-1.2E-3,1.5e+3.

2) Character constant: Sometimes, the quantities which are expressed in characters. These are 2 types.

i) Single character constants: It contains a single character enclosed within a pair of single quote marks.

Eg: ‘s’,’x’,’M’,’;”

Character constants have integer values known as ASCII values (ASCII-American Standard code for Information Interchange). Since each character represents an integer value, it is also possible to perform arithmetic operations on character constants.

ii) String constants: A String constant is a sequence of characters enclosed in double quotes. The characters may be letters, numbers, special characters and blank spaces.

Eg: “Hello”,”AZAD”,”58”, “$ABC”, “x”

NOTE: The character constant ‘X’ is not equivalent to the string constant “X”

Backslash Character constants/Escape Sequences: C supports some special backslash character constants that are used in output functions. These are also known as ***escape sequences***.

‘\a’- audible alert(bell)
‘\b’- backspace.
‘\f’- form feed
‘\n’- new line
‘\r’- carriage return
‘\v’- vertical tab
‘\’’- single quote.
‘\”’- double quote.
‘\?’- question mark
‘\0’- Null

3) **Identifiers**: Identifiers refers to the **names of variables**, functions and arrays. These are user-defined names are consisting of a sequence of letters & digits. **Identifier is a user defined name of an entity to identify it uniquely during the program execution.** It is a collection of characters which acts as name of variable, function, array, pointer, structure, label etc... In other words, an identifier can be defined as user defined name to identify an entity uniquely in c programming language that name may be of variable name, function name, array name, pointer name, structure name or a label.

Variables: Variable is an identifier. A Variable is a data name that may be used to store a data value. A variable may take different values at different times during execution, means the value of a variable may vary (change) during the execution.

Variable names may consist of letters, digits and under score (_) character. At the time of variables or identifiers, some rules are to be followed.

- First character must be an alphabet or underscore.
- ANSI standard recognizes a length of variable name is up to 31 characters. However, length should not be normally more than 8 characters.
- Keywords are not allowed to take as variable name or identifier name.
- White Space is not allowed.

Eg:

Valid variable names	Invalid Names.
A	123
Abc	$abc
xy	int
distance	10th
count	add num
sum10	#2
even_count	

Declaration of variables: A variable can be used to store value of any data type. The declaration of variables must be done before they are used in the program.

Syntax: data type variable, variable2, variable3 …. variable n

Variables are separated by commas and declaration statement ends with a semicolon.

Eg: int x, y, z;
Float f1, f2;
Double p, q, r;
Char ch;

Assigning values to variables/Defining variables:

Values can be assigned to variables using the assignment operator (=) i.e

Syntax: **variable= value**

The left side of the above syntax is any variable but only one variable. Right side part is to assign a value to the left side variable.

Eg: x=100 ch='m'; f1=8.932;

Sometimes declaration and definition takes place in single statement.

Eg: int a=5; declaration & definition both in single statement.

int a; declaration in one line
a = 5; definition in another line

Declaring a variable as constant:

The value of certain variables can be made to remain constant during the execution of C program by declaring the variable with data type qualifier **const** at the time of initialization. The constants will not change its value during the execution of the program.

Eg: const int x=100;

In the above example the value of x is made as constant and it will not change throughout the execution of the program.

Difference between a variable and a constant:

The **difference between variables and constants** is that **variables** can change their value at any time during the execution of the program that is Variables may vary but **constants** can never change their value. (The **constants** value is locked for the duration of the program). Constants can be very useful, Pi (**π**) for instance is a good example to declare as a **constant.**

Example 3: Program to add three numbers

```
void main()
{
int a=8, b=4, c=12, x;
clrscr();

x=a+b+c;
printf("The sum of a,b and c is %d", x);

getch();
}
```

Data types: -

- In c programming language, **Datatype** can be defined as a set of values with predefined characteristics.
- Datatypes are used to declare variable, constants, arrays, pointers and functions. All the values in a datatype have the same properties like **type and size.**
- The type of data or information which is used in the application should be declared by its appropriate **data type**.
- The set of values for each data type is known as the **domain** for the data type.
- Data types in C are classified into **different categories**.

1) Primary/pre-defined/ Fundamental Data type: **char, int, float, double.**
2) Secondary/user-defined data type: **enum, typedef**
3) Derived data type: **Arrays, structures, unions etc**
4) Empty data type: **void**

1) <u>Primary/Fundamental/scalar data type</u>: - These are basically four fundamental data types, integer(int), character(char), floating point(float) and double precision floating point(double). Once we declare a variable with its data type then that variable is holding some properties like

character types: A single character can be defined as a character (char) type data. Characters are usually stored in 8 bits of internal storage, that is it occupies 1-byte storage in memory.
"%c " is a format specification for character variables

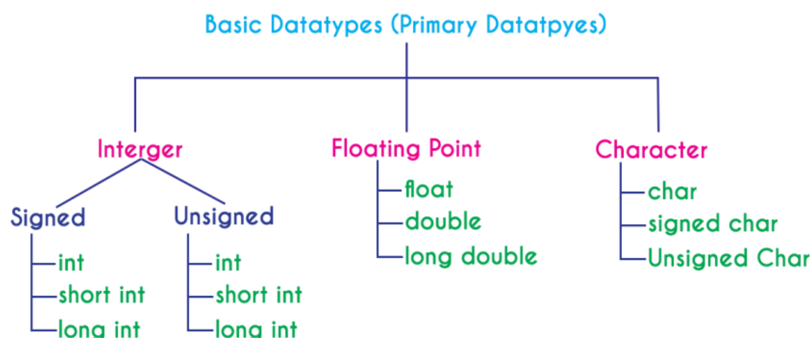
Int (Integer): To represent numbers without decimal point we use integer data type. These are signed or unsigned. These data types are **short int, int, long int**. The Integer variables have a limited value. They range from –32768 to 32767.
Syntax: int variable1, variable2…….variable n;
"%d " is a format specification for integer variables.

long int: In order to provide some control over the range of numbers and storage space, C has three classes of integer storage, namely short int, int and long int, in both signed and unsigned forms. We declare long and unsigned integers to increase the range of values.
"%ld " is a format specification for long int variables

floating point: Floating point numbers are stored in 32 bits with 6 digits of precision. Floating point numbers are defined in 'C' by keyword float. When the accuracy provided by a float number is not sufficient, the type double can be used to define the number.
"%f " is a format specification for float variables

double: A double data type number uses 64 bits giving a precision of 14 digits. These are known as double precision numbers. Remember that double type represents the same data type that float represents, but with a greater precision.
"%lf " is a format specification for double variables

long double: To extend the precision further for double variables, we may use long double which uses 80 bits.
"%Lf " is a format specification for long double variables



Data Type	Size in Bytes	Range	Format
signed char	1	-128 to 127	%c
unsigned char	1	0 to 255	%c
short signed int	2	-32768 to -32767	%d
short unsigned int	2	0 to 65535	%u
signed int	2	-32768 to -32767	%d
unsigned int	2	0 to 65535	%u
long signed int	4	-2147483648 to +2147483647	%ld
long unsigned int	4	0 to 4294967295	%lu
Float	4	3.4e-38 to 3.4e+38	%f
Double	8	1.7e-308 to 1.7e+308	%lf
long double	10	3.4e-4932 to 1.1e+4932	%LF

2) Secondary/Derived data type: The derived data types are complex structures that are built using the standard data types.
They are: function, pointer, structure, union, and array.

3) User-define data type: - The user defined data types are:

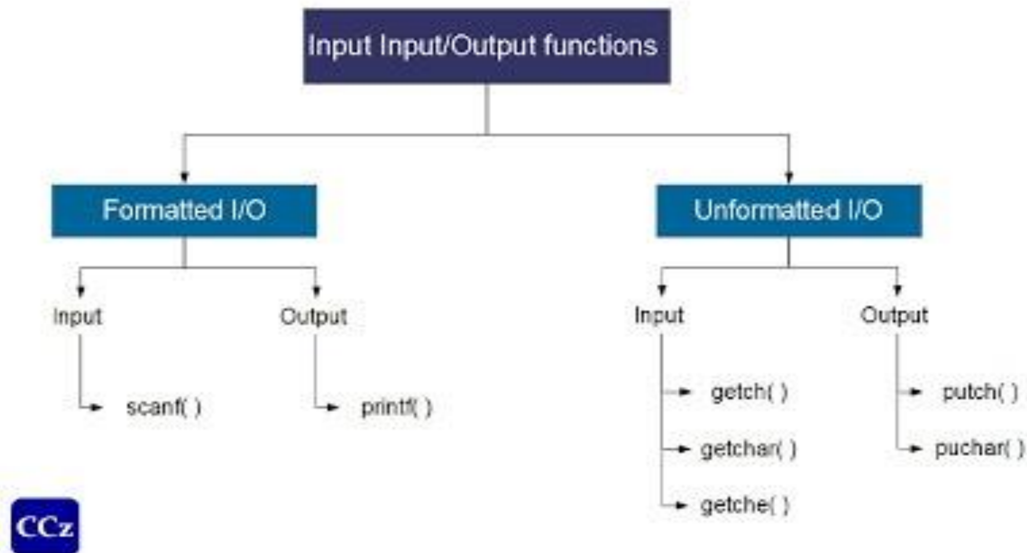
i) type definition (**typedef**)
ii)Enumerated data type(**enum**).

4) Empty data set: ANSI (American National Standards Institute) C supports the type void.
It specifies an empty set of values. It is used to specify the return type of a function when it is not returning any value, and also to indicate any empty arguments list to a function.
Eg: void display (void);

Input/output statements: -

- C programming language provides many **built-in functions** to read any given input and to display data on screen when there is a need to output the result.
- All these built-in functions are present in **C header files**; we will also specify the name of header files in which a particular function is defined.

- An input/output function can be accessed from anywhere within a C program by writing the function name, followed by a list of arguments enclosed in parentheses. **<stdio.h>** is the header file required by the standard input/output library functions.
- The input/output functions permit the transfer of information between the computer and the standard input/output devices.
- The input/output statements are as follows:



1. **Unformatted input/output functions:**
   getchar( ): to read/input a character
   putchar( ): to print/output a character.
   gets( ): to read/input a string
   puts( ): to print/output a string
   getch():to read/input a character
   getche():to read/input a character with echo
2. **Formatted input/output functions:**
   scanf( ): to read/input multiple values
   printf( ): to print multiple values.

Reading and Writing Character:

- **The getchar( ) function:-** The getchar( ) function is used to read a single character from the standard input device(keyboard) and does not require any arguments. The general format is:

Syntax: **Character-variable=getchar( );**

Eg: char c;

c = getchar();

- **The getch( ) function:** This is also an input function. This is used to read a single character from the keyboard like getchar() function. The character data read by this function is directly assign to a variable rather it goes to the memory buffer, the character data directly assign to a variable without the need to press the Enter key. Another use of this function is to maintain the output on the screen till you have not press the Enter Key.
  The general syntax is as: **v = getch();** where v is the variable of character type.

- **getche() function:** This is same as getch() function, except it is an echoed function. It means when you type the character data from the keyboard it will visible on the screen.

The general syntax is as: **v = getche();** where v is the variable of character type.

- **The putchar () function: -** Single characters can be displayed by using the putchar function. It is complementary to the getchar function. It transmits a character type variable to a standard output device and must be expressed as an argument, enclosed in parentheses to the function.

Syntax: putchar(character variable name);
Eg: char c;
putchar(c);

Example 4: A simple C program to illustrate the use of getch() function:

```
/*Program to explain the use of getch() function*/
#include<stdio.h>
#include<conio.h>
void main()
{
char n;
clrscr();

puts("Enter the Char");
n = getche();
puts("Char is :");
putchar(n);

getch();
}
```

```
OUTPUT IS:
Enter the Char L
Char is L
```

- **gets() function:** This function is an input function. It is used to read a string from the keyboar. It is also buffered function. It will read a string, when you type the string from the keyboard and press

the Enter key from the keyboard. It will mark nulll character ('\0') in the memory at the end of the string, when you press the enter key.
The general syntax is as: **gets(v);**
where v is the variable of character type. For example:

Ex: char n[20];
gets(n);

- **puts() function:** This is an output function. It is used to display a string inputted by gets() function. It is also used to display an text (message) on the screen for program simplicity. This function appends a newline ("\n") character to the output.

  The general syntax is as:

  puts(v);
  or
  puts("text line");

where v is the variable of character type.

Example 5:A simple C program to illustrate the use of gets() and puts() function:

```
/*Program to illustrate the concept of puts() with gets() functions*/
#include<stdio.h>
#include<conio.h>
void main()
{
char name[20];
clrscr();

puts("Enter the Name");
gets(name);
puts("Name is :");
puts(name);
}
```

```
OUTPUT IS:
Enter the Name
Laura
Name is:
Laura
```

Formatted Functions:

- **The scanf( ) function**:- It is an input function. Input data can be entered into the computer from a standard input device (keyboard) by means of scanf (scanf means scan formatted) function and can

be used to read multiple values of any type at time. It returns the number of data items that have been entered successfully.

Syntax: scanf("control string" , arg1,arg2,arg3….argn);

The control string specifies the field format in which the data is to be entered and the arguments arg1, arg2,…..specify the address of locations where the data is stored. Control string and arguments are separated by comma (,).

The control string (also known as format string) contains field specifications. Each specifications consists of the **%** (percent sign), followed by **conversion character** which indicates the data item name with an optional number that specifying field width. Each variable name in the arguments list must be preceded by **&** (ampersand) symbol as the arguments values are to be stored in some particular address.

Commonly used format specifications

%c	data item is a single character
%d	data item is a decimal integer
%e	data item is a floating point value(exponential form)
%f	data item is a floating point value (decimal form)
%g	data item is a floating point value
%h	data item is a short integer
%i	data item is a decimal, hexadecimal, or octal integer
%o	data item is an octal integer
%s	data item is a string (group of characters)
%u	data item is an unsigned integer
%x	data item is a hexadecimal integer

The data items must correspond to the arguments in the scanf function in number, in type and in order. Each variable to be read must have a field specification.

Eg:

- We can read multiple values at a time by using scanf

```
scanf("%d %d", &num1,&num2);
```

will read the data : 23479 50 i.e num1 is assigned by 23479 and num2 is assigned by 50.

- We can read mixed type of values by using scanf function.

```
int     count;
char    code;
float   ratio;
char    itemname[20];
scanf("%d %c %f %s", &count, &code, &ratio, itemname);
```

will read the data like : 28 M 2.932 coffee

➢ **<u>The printf( ) function</u>**:- It is an output function. Output data can be written from the computer onto a standard output device (monitor) by using the printf( ) function. The printf( ) function can be used

to print multiple values of any type at time. It returns the number of data items that have been printed successfully.

Syntax: printf("control string" , arg1,arg2,arg3….argn);

Where the control string refers to a string thet contains formatting information. And the arguments arg1, arg2,……..represent the individual output data items. The arguments can be written as constants, single variable names or array names, or more complex expressions. Control string and arguments are separated by comma (,).

The arguments in a printf function do not represent memory address and therefore the are not preceded by ampersand (&) symbol. We can print data items with formatted output by using field width option in printf function.

Eg:

- We can print multiple values at a time by using printf
  ```
  int num1=3467, num2=28;
  printf("%d %d", num1,num2);
  will display the data: 3467 28
  ```
- We can print mixed type of values by using printf function.
  ```
  int     count=23;
  char    code='X';
  float   ratio=2.35;
  printf("%d %c %f %s", count, code, ratio);
  will display the data like: 23   X    2.35
  ```
- We can print formatted output by using printf function. It is possible to limit the number of characters by specifying a maximum field width for that data item and by taking the format string.
  ```
  Int a,b;
  a=3457;
  b=123;
  printf("%3d %2d",a,b);
  it will display the data items a and b as:
  3457  123
  ```

Example 6: program for printf() and scanf() functions

```
/*Program to explain the use of scanf(), printf() function*/
#include<stdio.h>
#include<conio.h>
void main()
{
char sname[20];
int rno;
clrscr();

//reading student name and roll number
```

```
printf("Enter your  name");
scanf("%s",sname);
printf("Enter your roll number");
scanf("%d", &rno);

//displaying student details
printf("Name %s",sname);
printf("Roll Number %d",rno);

getch();
}
```

Operators: - Operator is a symbol which represents a particular action that can be performed on operands .C is extremely rich in operators, and are classified as follows:

1. Arithmetic Operators (+ - * / %)
2. Relational Operators (< > <= >= == !=)
3. Logical Operators (&& | | !)
4. Assignment Operators (= += -= *= /= %=)
5. Conditional Operator (?:)
6. Increment & Decrement Operators (++ --)
7. Bitwise Operators (& | ^ ~ << >>)
8. Special Operators (( ) [ ] , . -> (type) sizeof( ))

Operators can also be designated as ***unary, binary and ternary*** operators depending on whether they operate on one, two or three operands respectively.

Unary operators: Only one operand is required to perform calculation.
++, **--**, **-**, +, **!** are some unary operators.
Eg: - a - a is operand, - is unary operator, which operates on only one operand.
Binary Operators: Two operands are required to perform calculation.
Eg1: a + b: a, b are operands
+ is operator (binary) which operates on two operands on a, b.
Eg2: x * y * z : x, y and z are operands, and * is binary operator which operates on two operands at a time.

Ternary operator: Three operands are required to perform calculation.in C language we have one ternary operator that is conditional operator (**? :**)

Eg: (expression 1)? (expression 2): (expression 3);

```
a > b?printf("a"):printf("b");
```

The following are the different types of operators in C -

1. <u>Arithmetic Operators</u>: These are 5 arithmetic operators in C. They are

+ : Addition or unary plus.
- : Subtraction or unary minus.
* : Multiplication
/ : Division
% : Modulus division (Remainder value).

If an expression involves all integer values, then it is called integer arithmetic. If an expression involves all real (float) operands, then it is called real arithmetic, and when some of the operands are integer and some other are real, then expression is called Mixed-mode arithmetic and result is always a real number.

Eg: int a=5 b =2;

C = a + b c = 7
C = a – b c = 3
C = a * b c = 10
C = a % b c = 1

2) <u>Relational Operators</u>: These are 4 relational Operators and 2 equality operators in C. They are

< : less than
>= : less than or equal to
> : greater than
>= : greater than or equal to
= = : equal to
!= : not equal to

Relational Operators are used to compare two quantities and depending on their relation certain decisions are made. If the defined relation is true or correct then the value is one otherwise 0, it is false. These are used in decision making constructs like if….else, while, for etc.

Eg: 1)a=5,b=2 2)a=5,b=4

```
If (a>b)                          if (a==b)
   Big=a                              printf (“a,b are equal”);
else                              else
   Big=b                              printf (“not”);
```

3) <u>Logical Operators</u> These are three logical operators. They are
.

```
&&   :   logical AND
||   :   logical OR
!    :   logical NOT
```

- The Logical AND (&&) and logical OR (||) are used to test more than one condition.
- In the logical AND (&&) operator, when both the conditions are true, then it is true.
- In logical OR(||) operator, if one of the expression is true, then it is true.
- The logical NOT (!) operator is used to change the value from true to false or false to true.
  i.e !(true)=false and !(false)=true.

The truth table for logical AND and logical OR:

Truth table for OR , AND, and NOT operations
True = 1, False = 0

A	!A	B	!B	A || B	A&&B
0	1	0	1	0	0
1	0	0	1	1	0
0	1	1	0	1	0
1	0	1	0	1	1

illustrates the applications of Boolean operators

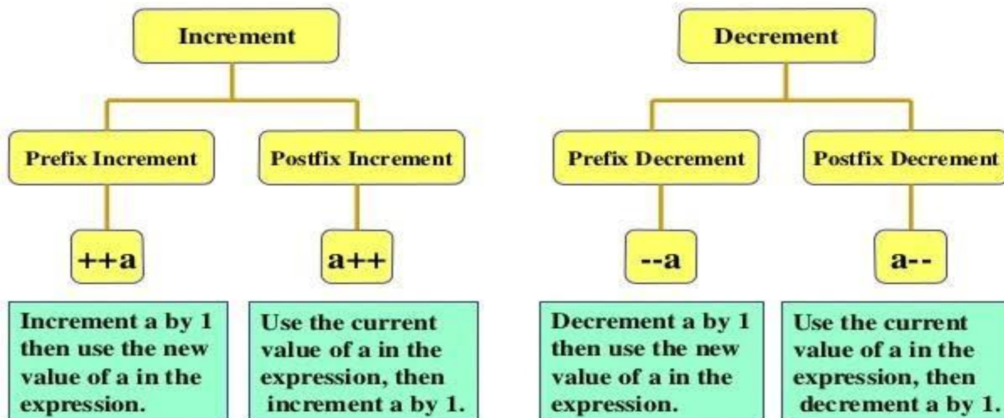
Eg: a=8, b=3, c=9

```
If((a>b) && (a>c))
   Printf(“a is big”);
```

This expression yields false because a is not greater than c. i.e both should be true then only the express -ion give true, otherwise false.

i.e T && F is F
1 && 0 is 0

4) Increment/Decrement operators: -



Increment operator (++) is used to increment the value by one. There are two ways to increment a value: Pre-increment and Post-increment

- **++ x is pre-increment** and **x++ is post increment**. Both gets evaluated to x = x +1.

  i.e. + + x => x = x + 1

  x + + => x = x + 1

- **In** ***pre-increment***, first the value is incremented by one and then the required operation is performed with the *updated value*. Where as **in** ***post-increment***, first the operation is performed with the *present value* and then it is incremented by one.

Eg: x = + + y; where y = 5

After this expression: x = 6 and y = 6. I.e. the value of y is incremented first (y = y + 1 => y = 5 + 1 => 6) and then it is assigned to x (x = 6).

x = y + +; where y = 5

After this expression: x = 5 and y = 6. I.e. first, the value of y is assigned to x (x = 5) and then y is incremented by one (y = y +1 => y = 5 +1 => 6) .

Decrement operator (--) is used to decrement the value by one. There are two ways to decrement a value: pre-decrement and post- decrement.

- **--x id pre-decrement** and **x-- is post-decrement**. Both gets evaluated to x = x – 1.

  i.e. --x => x = x -1

  x-- => x = x -1

- In *pre-decrement*, first the value is decremented by one and then the required operation is performed with the *updated value*. Where as in *post-decrement*, first the operation is performed with the *present value* and then it is decremented by one.

Eg: x = - - y; where y = 10

After this expression: x = 9 and y = 9. I.e. the value of y is decremented first (y = y - 1 => y = 10 - 1 => 9) and then it is assigned to x (x = 9).

x = y - -; where y = 10

After this expression: x = 10 and y = 9. I.e. first, the value of y is assigned to x (x = 10) and then y is decremented by one (y = y -1 => y = 10 -1 => 9).

5) <u>Assignment operators</u>: - Assignment operators are used to assign the result of an expression or a value to a variable. The assignment operator is denoted by the symbol **=**. C also has a set of shorthand assignment operators of the form:

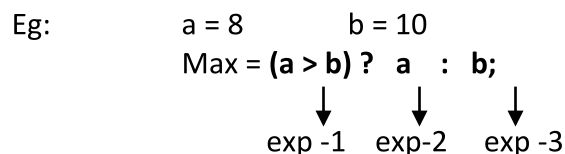
Variable operator = expression

=	: Assign right hand side value to the left hand side variable.
+ =	: Addition assignment
- =	: Subtraction assignment
* =	: Multiplication assignment
/ =	: Division assignment

Eg:
a = 5;
a += 2 => a = a +2 => a = 5 +2 => a = 7.
a *= 10 => a = a * 10 => a = 5 * 10 => a = 50.

6) <u>Conditional operator</u>: - This can be represented by the symbol **" ? : ".** It is also called as ternary operator as it has three operands, and two operators. Each operand is an expression. The first two operands are separated by **?** and **:** separates the last two operands. If the first expression yields **true,** then **second expression** is evaluated. If the first expression yields **false** value, then **third expression** is evaluated.

Eg: a = 8 b = 10
Max = **(a > b) ? a : b;**



exp -1 exp-2 exp -3

Here a > b (8 is not greater than 10) expression is false so expression 3 is evaluated. I.e. the value of b is assigned to max. So max = 10.

a = 8 b =10
c = (a < b) ? ++ a : ++b;

Here a < b (8 is less than 10) expression is true, so expression 2 is evaluated. i.e. first, the value of a is incremented by one (pre-increment) and then assigned to c. So c = 9 and a = 9.

7) <u>Bitwise operators:</u> - Bit-wise operators are low level operators, and these operators can work with only integer type operands. These operators are used for testing the bits. They are:

& Bitwise AND

| Bitwise OR
^ Bitwise XOR (exclusive OR)
<< Bitwise left shift
>> Bitwise right shift
~ Bitwise one's complement

Truth table for bitwise AND, bitwise OR and bitwise XOR is as follows:

Op1	**Op2**	**Op1 & Op2**	**Op1 | Op2**	**Op1 ^ Op2**
0	**0**	**0**	**0**	**0**
0	**1**	**0**	**1**	**1**
1	**0**	**0**	**1**	**1**
1	**1**	**1**	**1**	**0**

- The one's complement operator is a unary operator, and inverts all the bits represented by its operand. i.e. 0s becomes 1s and 1s become 0s.
  Eg: a = 5 a = 0101 ~a = 1010
- The shift operators are used to move bit patterns either to the left or to the right and are used in the following form:
  Op << n; Op >> n;

Where Op is an integer expression and n is the number of bits to be shifted.

- The left shift operation causes all the bits in the operand Op to be shifted to the left by n positions. The left most n bits in the original bit pattern will be lost and the right most n bit positions that are vacated will be filled with 0s. Similarly, the right shift operation also takes the same process but shifted towards right side, and left most vacated positions are filled with 0s.
- There are two restrictions on the value of n.
  i. It may not be negative
  ii. It may not exceed the number of bits used to represent the operand Op.

Eg: a = 5 (0101) b = 8 (1000)
a & b => 0101 & 1000 => 0000 => 0
a | b => 0101 | 1000 => 1101 => 13
a ^ b => 0101 ^ 1000 => 1101 => 13
a << 1 => 0101 << 1 => 1010 =>10
a >> 1 => 0101 >> 1 => 0010 => 2

8) <u>Special operators</u>: - The special operators are:

, (comma operator): separates two values or expressions.
Sizeof operator: gives the size of a given variable in bytes.
***** (Pointer operator): dereference operator or value at address operator

& (address of operator): gives the address of a particular location

. (Direct member selection operator): used in structure member selection

→ (indirect member selection operator): used in structure member (pointer) selection

Eg: c = (a =5, b = 8, a +b);

Here first a becomes 5 and b becomes 8 and then a+b is evaluated to 5+8 => 13 and the right most expression (a + b) value is assigned to c. So c = > 13.

int a = 8; sizeof (a); → gives the value 2, that is the size of integer in bytes.

float x; sizeof(x); → gives the value 4, that is the size of float in bytes.

&x → gives address of the variable x.

*x → gives the value, which is pointed by x.

Operator precedence (Arithmetic operator precedence):

Operator precedence describes the order in which the expression is to evaluate by providing some priorities to the operators.

Rules of Operator Precedence:

C applies the operators in arithmetic expressions in a precise sequence determined by the following **rules:**

1. **First priority - Parentheses**- Operators in expressions may contained within pairs of parentheses. These parentheses are evaluated first. Parentheses are said to be at the "highest level of precedence". In cases of **nested** such as: ((a + b) + c), The operators in the *innermost* pair of parentheses are applied first.
2. **Second priority - Multiplication, division** and **remainder operations** are applied next. If an expression contains several multiplications, division and remainder operations, evaluation proceeds from **left to right**. Multiplication, division and remainder are said to be on the same level of precedence.
3. **Third priority - Addition and subtraction** operations are evaluated next. If an expression contains several addition and subtraction operations, evaluation proceeds from left to right. Addition and subtraction also have the same level of precedence, which is lower than the precedence of the multiplication, division and remainder operations.
4. **Last priority - The assignment operator** (=) is evaluated last.

Example 7: program on operators.

//Program to implement different arithmetic operators with operator precedence.

```
void main()
{
int a=20;
int b=10;
int c=15;
int d=5;
int e;
 clrscr();

   e = (a + b) * c / d;
   printf("Value of (a + b) * c / d is : %d\n",  e );

   e = ((a + b) * c) / d;                          // (30 * 15 ) / 5
   printf("Value of ((a + b) * c) / d is  : %d\n" ,  e );

   e = (a + b) * (c / d);                          // (30) * (15/5)
   printf("Value of (a + b) * (c / d) is  : %d\n",  e );

   e = a + (b * c) / d;                            //  20 + (150/5)
   printf("Value of a + (b * c) / d is  : %d\n" ,  e );

getch();
}
```

Expressions: -

An expression is a sequence of operands and operators that reduce to a single value. An expression is a combination of variables, constants and operators following the syntactic rules of the language. Expressions are evaluated using an assignment statement of the form:

Variable=expression;

Eg:

Valid expressions	Invalid expressions
a = x * y;	a + b = c;
b = (x * x) – 2 * a * b;	a, b = x * y;
c = a / b;	a * b = x * y;

Evaluation of Expressions:

Expressions are evaluated using an assignment statement of the form:

Variable = expression;

Where **variable** is any valid C variable name. When the statement is encountered, the expression is evaluated first and the result then replaced the previous value of the variable on the left-hand side. All variables used in the expression must be assigned values before evaluation is attempted.

Examples:

```
X = a * b – c;
Y = b / c * a;
Z = a – b / c + d;
```

Example 8:example program on expressions

```
void main()
{
float a, b, c, x, y, z;
a=9;
b =12;
c = 3;
x = a – b / 3 + c * 2 – 1;
y = a – b / (3 + c) * (2- 1);
z = a –(b / (3 + c) * 2) – 1;
printf("x = %f\n", x);
printf("y= %f\n", y);
printf("z = %f\n", z);
}
```

=====================*************************=======================

Prepared by Sunitha Mutchinthala

Assistant Professor

Dept of Computer Science

St Joseph's Degree & PG College

Unit - II:

Decision Making and Branching: Introduction, if, if-else, Nested-If statements, Switch-case statement. **Decision Making and Looping:** Introduction, *While* Statement, *Do-while statement,* For Statements, break and continue statements.

Decision Making and Branching:

A program is nothing but the execution of sequence of one or more instructions. Sometimes it is desirable to alter the execution of sequence of statements in the program depending upon certain circumstances. This involves **decision making** through **branching** and **looping.** Control statements specify the order in which the various instructions in a program are to be executed by the computer. i.e. they determine the 'flow of control' in a program.

Decision making and branching structures require that the programmer to specifies one or more conditions to be evaluated or tested by the program along with a statement(s) to be **executed** if the condition is determined to be **true**, and optionally, other statements to be executed if the condition is determined to be **false**. There are different decision making statements in C language:

1. **If statement** – simple if statement, If-else statement, Nested if statement and else-if ladder
2. **Switch-case statement**

- **if statement**: - The if statement is a powerful decision making statement and is used to control the flow of execution of statements. It can be achieved through different forms of if statement.

a) simple if b) if – else c) nested if-else d) else-if ladder

a) simple if statement: - C uses the keyword "if" to execute a set of statements or one statements when the logical condition is true.

Syntax:
```
if(condition or expression)
{
        Statements-block
}
next statement
```

It allows the computer to evaluate the expression first and then depending on the value of expression, the control is transfers to the particular statement. If the expression is **true**(non-zero value) then the *statement-block* is executed and *next statement* is executed. If the expression is **false**(zero), directly next statement is executed without executing the *statement-block*. *Statement-block* may be one or more statements. If more than one statement, then keep all those statements in compound block({ }).

Eg: 1.
```
if( age > 60)
        printf("Old person");
```

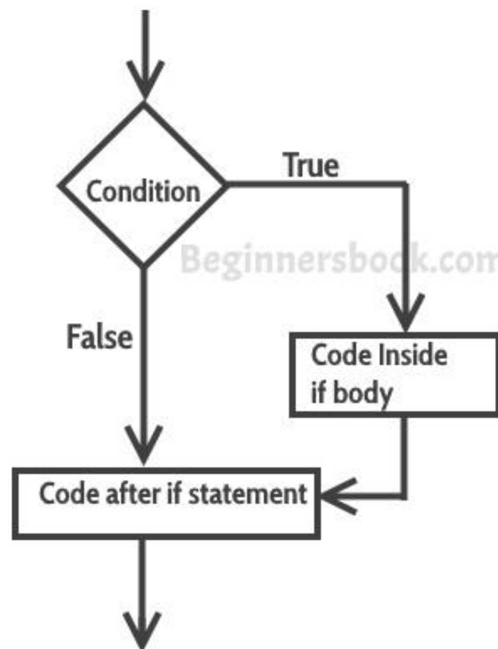
2.
```
if( marks >100)
{
```

```
        printf(" Invalid marks");
        printf(" Check your marks once again ");
    }
```

b) if –else statement: - It is an extension of simple if. It takes care of both the true as well as false conditions. It has two blocks. One is for if and it is executed when the condition is **true**, the other block is for else and it is executed when the condition is **false**. No multiple else statements are allowed with one if.

```
Syntax:if(test condition)
        {
                True block statements
        }
        else
        {
                False block statements
        }
        Next statement
```

Flow Diagram of if statement:



If the condition is true then true block statement(s) is executed, otherwise if the condition is false, then false-block statement(s) is executed. The else cannot be used without if.

```
Eg:     if(a%2 = = 0)
        {
                Printf(" even number");
        }
        else
```

```
{
        Printf("odd number");
}
```

c) <u>Nested if-else statement</u>: - When a series of decisions are involved, we have to use more than one if-else in nested form.

Syntax:

```
if( condition -1)
{
        if(condition -2)
        {
                .........
                if(condition-3)
                {
                        Statement -1
                }
                else
                {
                        Statement -2
                }
        }
        else
        {
                Statement-3
        }
}
else
{
        Statement-4
}
Next statement
```

In this kind of statements, number of logical conditions is checked for executing various statements. If any condition is true then associated block will be executed, otherwise it skips and executes else block statements. We can add repetitively if statements in else block also.

Eg:

```
if(a>b)
{
        if(a>c)
        {
                printf(" ais big");
        else
        {
                printf("c is big");
```

```
        }
    }
    else
    {
        if(c>b)
        {
            printf("c is big");
        }
        else
        {
            printf(" b is big');
        }
    }
```

d) else-if ladder: - The else-if ladder is used when multipath decisions are involved.

Syntax:

```
if(condition-1)
        Statement-1
else    if(condition-2)
                Statement-2
        else    if(condition-3)
                        Statement-3
                ..
                ..
                ..
                else
                        Statement-x
Next statement
```

Eg:

```
if(avg>=70)
        Printf(" distinction");
else    if(avg>=60)
                Printf(" first class");
        else    if(avg>=50)
                        Printf("second class");
                else    if(avg>=40)
                                Printf("third class");
                        else
                                Printf("fail");
```

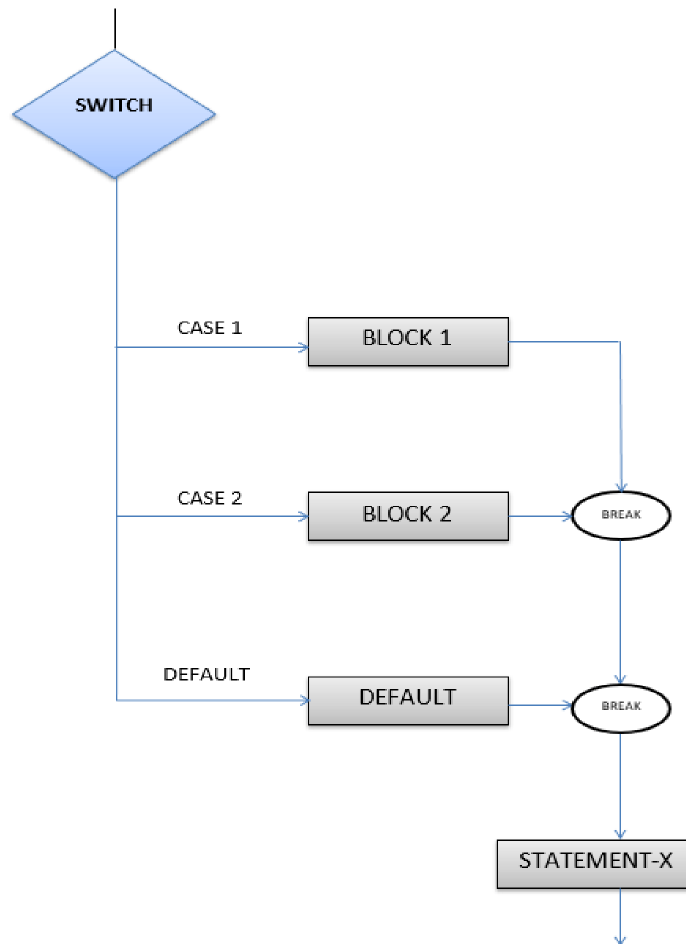
Switch-Case statement: - At times, the if condition may increase the complexity of the program when one of many alternatives is to be selected. C has built-in multiway decision statement known as ***switch-case.*** A switch statement tests the value of a variable and compares it with multiple cases. Once the case match is found, a block of statements associated with that particular case is executed. The switch statement requires only one argument variable or expression. It tests the value of a given variable against a list of case values and when a match is found, a block of statements associated with that case is executed, if not such match, then default statement is executed.

Syntax:

```
switch(variable or expression)
{
 case value-1:        block-1
                      break;
 case value-2:        block-2
                      break;
..
..
..
..
default:  default-block
}
next statement
```

The following are the rules to follow in switch –case statement implementation

- The expression in switch is an integer expression or character.
- Value-1, value-2 these are either integer constants or character constants.
- Case labels always end with a colon (:). Each of these cases is associated with a block.
- These values should be unique with in a switch statement. **Case** labels end with colon (:)
- The **break** statement signals the end of a particular case and causes an exit from the switch statement, transferring the control to the next statement following the switch.
- The **default** is an optional case when present. It will be executed if the value of the expression doesn’t match with any of the case values. If not present, no action takes place if all matches fail and control goes to the next statement.



Eg: Following program illustrates the use of switch:

```
#include <stdio.h>
    int main() {
        int num = 8;
        switch (num) {
            case 7:
                printf("Value is 7");
                break;
            case 8:
                printf("Value is 8");
                break;
            case 9:
                printf("Value is 9");
```

```
                break;
            default:
                printf("Out of range");
                break;
        }
        return 0;
    }
```

Output:

```
Value is 8
```

Decision making and looping – Sometimes many tasks are needed to be done with the help of a computer and they are repetitive in nature. Such type of actions can be easily done by using loop control statements. A loop statement allows us to execute a statement or group of statements multiple times. A loop is defined as a block of statements which are repeatedly executed for certain number of times to do a specific task.

Eg: calculation of salary of employs of an organization for every month.

Steps in writing loops: -

1. **Loop variable**: It is a variable used in loop to evaluate, **Initialization of a loop variable** with starting value or final values in the loop.
2. **Test-condition**: It is to check the condition to terminate the loop. It is any relational expression with the help of logical operators.
3. **Update statement**: It is the numerical value added or subtracted to the loop variable in each round of the loop.

C language supports three types of loop control statements.

- **while**
- **do-while**
- **for**

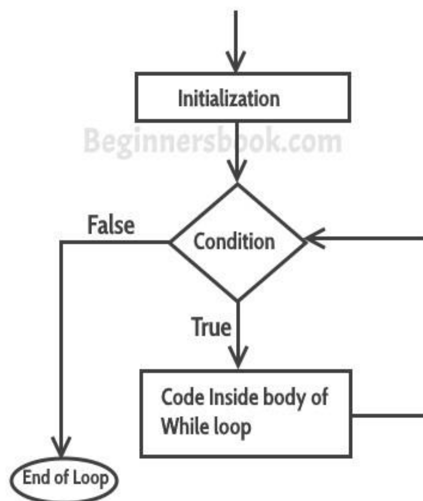
- while:- This is the simplest looping structure in C. the while is an entry-controlled loop statement.

Syntax:

```
initial statement
while(test condition)
{
        Statement(s)
        Update statement
}
```

The test condition may be any expression, is evaluated and if it is true then the body of the loop is executed. The test condition is once again executed for updated values, and if it is true the body of the loop is executed once again. This process is repeated until the test condition is finally becomes false and control is transferred out of the loop to the next statement. The body of the loop may contain one or more statements. The braces are needed if the body of the loop contains more than one statement.

Flow Diagram of while loop:



Eg:

```
main()
{
        int i,sum;
        i = 1; sum=0;
        while(i<=10)
        {
                sum = sum + i ;
                i + +;
        }
        printf("sum=%d",sum);
}
```

Here the value, sum of first ten numbers is stored into the variable sum; **i** is called as loop variable. The loop is repeated for ten times to do that process, each time by incrementing the value of i by one. Once the value of i becomes 11 then the test condition becomes false and the control is out of the loop.

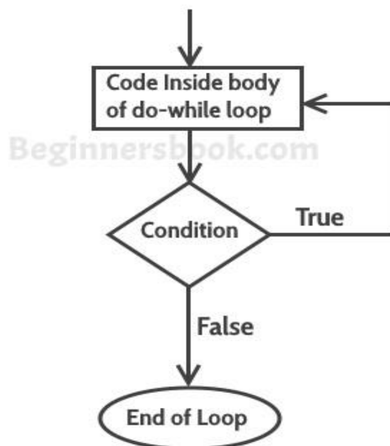
- <u>do-while</u>:- On some occasions it might be necessary to execute the body of the loop before the test condition is performed. Such situations can be handled by the do-while statement. Do-while id exit controlled loop statement.

Syntax:initial statement

```
        do
        {
                Statement(s)
                Update statement
        } while(test – condition);

        Next statement
```

The body of the loop is executed first, and then at the end of the loop the test condition is evaluated, if it is true then the statements are executed once again. The process of execution continues until the test condition finally becomes false and the control is transferred to the next statement

Flow diagram of do while loop



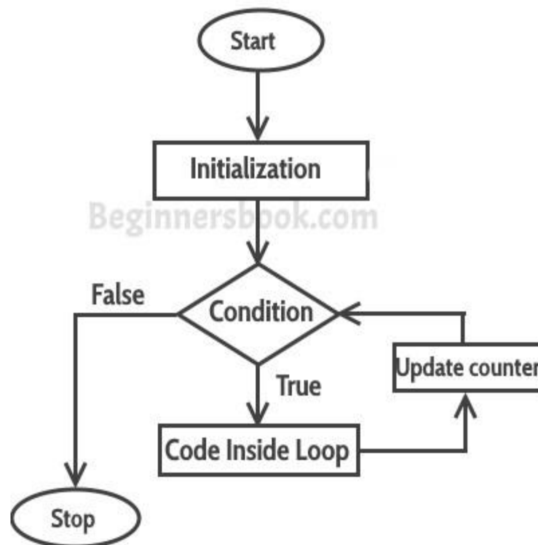
Eg:
```
main()
{
        int i, sum;
        i =1;
        sum=0;


        do
        {
                sum = sum + i;
                i + +;
        } while( i<=10);
```

```
        printf("Sum=%d", sum);
    }
```

➢ <u>The for statement</u>: - The for loop entry controlled loop that provides a more concise loop control structure.

Syntax:
```
for(initialization ; test-condition ; increment/decrement)
{
        Statement(s)
}
Next statement
```

Flow diagram of do while loop



The for loop allows to specify three things about the loop in a single line.

i. Setting a loop counter variable to an initial value using assignment statement.
   Eg: i=1 count=0;
ii. the test condition is a relational expression that determines the number of iterations desired or it determines when to exit from the loop. If the test condition is true, the body of the loop is executed, otherwise the loop is terminated and execution continues with the next statement after the loop.
   Eg: i<=10
iii. After evaluating the last statement of the body the loop, the control is transferred to the increment/decrement statement of the loop. And the new value is again tested to see whether it satisfies the loop condition or not.
   Eg: i++ ++i i+=2

- The body of the loop may contain one or more statements. In case there is more than one statement then braces
- The three sections of for loop must be separated by semicolons (;). Initialization and incr/decr parts may contain more than one statement must be separated by commas.

  Eg: for(i=1, j=10 ; i<=10 ; i++, j--)
- The test-condition may have any compound relation and the testing need not be limited only to the loop variable.

Eg: for(i=1; i<20 && sum <100 ; i++)

- It is permissible to use expressions in the assignment statement of initialization and incr/decr sections.

Eg: for(k=(a+b)/2; k>0;k=k/2)

- The sections of for loop my absent depends on requirement in the program. But it leads to take some extra care about those sections.

Eg: for(**;** **;**)

This statement leads to infinite loop or never–ending process.

<u>Nesting of loops</u>: - The way if statement can be nested, similarly whiles and for can also be nested.

Eg:
```
for(i=1;i<=3;i++)          //outer for loop
{
        for(j=1;j<=2;j++)          // inner for loop
        {
                Printf("\t %d   %d", i,j);
        }
        Printf("\n");
}
```

Here in this example nested for loop is used, and the total process is executed for 6(3 * 2) times. And the output of this example will be:

```
1  1    1   2
2  1    2   2
3  1    3   2
```

The way for loops have been nested here, similarly while and do-while can also be nested. Not only this, a for loop can occur within a while loop, or a while within a for.

Eg: **Comparison of three loops. Finding the sum of first 10 numbers by using all loop statements**.

//sum of first 10 numbers by using *while* loop.

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int i,sum;
        clrscr();
        sum=0;
        i=1;
        while(i<=10)
        {
                sum=sum+i;
                i++;
        }
printf("Sum of first 10 numbers = %d", sum);
getch();
}
```

//sum of first 10 numbers by using *do-while* loop.

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int i,sum;
        clrscr();
        sum=0;
        i=1;
        do
        {
                sum=sum+i;
                i++;
        }while(i<=10);
printf("Sum of first 10 numbers = %d", sum);
getch();
}
```

//sum of first 10 numbers by using *for* loop.

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int i,sum;
        clrscr();
for(i=1,sum=0;i<=10;i++)
        {
                sum=sum+i;
        }
printf("Sum of first 10 numbers = %d", sum);
getch();
}
```

<u>Jumps in loops</u>: - We often come across some situations where we want to make a jump from one statement to other statement, jump out of a loop or to jump to next iteration of the loop instantly, this can be accomplished by the statements like:

- **break**
- **continue**

<u>The *break* statement</u>: -

- When we want to jump out of a loop instantly without waiting to get back to the condition test, then the keyword *break* allows us to do this.
- The break statement provides an early exit from the loop.
- A break is usually associated with an if.

Eg:
```
for(i=1;i<=3;i++)
{
        for(j=1;j<=5;j++)
        {
                if(j = = 3)
                        break;
                else
```

```
                    printf(" %d %d", i, j);
            }
            Printf("\n");
    }
```

Output:
```
1   1    1  2
2   1    2  2
3   1    3  2
```

In this example when j value equals 5, break takes the control outside the inner for loop only, since it is placed inside the inner loop.

The *continue* statement: -

- When we want to take the control to the beginning of the loop by passing the statements inside the loop which are not yet been executed, then the keyword *continue* allows us to do this.
- It causes the next iteration of the loop to begin and it applies only to loops.
- A continue is usually associated with an if.

Eg:
```
for(i=1;i<=2;i++)
{
        for(j=1;j<=2;j++)
        {
                if(i = = j)
                        continue;
                else
                        printf(" \n%d  %d", i, j);
        }
}
```

Output:
```
1  2
2  1
```

In this example when i and j values are equal, *continue* takes the control to the inner for loop by ignoring rest of the statements in the inner for loop.

Arrays: Introduction, Defining, creating and initializing an array One-Dimensional Array, Two-Dimensional Array.
Pointers: Introduction, Understanding Pointer-creating and assigning pointer, Accessing address of a variable. Pointers to arrays.

ARRAYS

Introduction:

C supports a *derived data type* known as *array* that can be used to handle large amounts of data (multiple values) at a time.

- An ***array*** is collection of elements of the *same data type*, which share a common name.
- Array elements are always stored in *contiguous* memory locations.
- Each element in the group is referred by its position called *index*.
- The first element in the array is *numbered 0*, so the last element is one less than the size of the array.
- Before using an array its type and dimension must be declared, so that the compiler will know what kind of an array and how large an array.

Each array element is referred by specifying the array name followed by one or more subscripts, with each subscript enclosed in square brackets. The value of each subscript must be expressed as a *non-negative integer*, variable or expression. The number of subscript determines the dimensionality of the array. We can use arrays to represent not only list of values but also tables of data in two or more dimensions. We have different types of arrays based on its dimension.

- One-dimensional arrays / Single-dimensional arrays
- Two-dimensional arrays / Double-dimensional arrays
- Multi-dimensional arrays

One – Dimensional arrays: -

A list of items can be given one variable name using only one subscript and such a variable is called one-dimensional array or single-dimensional variable. C array is beneficial if you have to store similar elements. For example, if we want to store the marks of a student in 6 subjects, then we don't need to define different variables for the marks in the different subject. Instead of that, we can define an array which can store the marks in each subject at the contiguous memory locations. By using the array, we can access the elements easily. Only a few lines of code are required to access the elements of the array.

Syntax: The general form of array declaration is:

data-type array-name[size];

The data-type specifies the type of elements such as int, float or char. And the size indicates the maximum number of elements that can be stored in that array.

Eg:

int	a[5] ;	here a is an array containing 5 integer elements with index values 0 to 4.
float	height[10];	here height is array containing 10 real elements with index values 0 to 9.
char	name[20];	here name is an array containing 20 characters with index values 0 to 19.

Note: C language character *strings are* as simple as *array of characters*. And every string should be terminated by *null character* ***('\0').***

Example: Array representation in memory

int a[5];

And the computer reserves five storage locations as shown below

Location a[0]
Location a[1]
Location a[2]
Location a[3]

For example if the values to the array elements can be assigned as follows-

a[0]=35

a[1]=40

a[2]=20

a[3]=57

a[4]=19

This would cause the array 'a' to store the values as shown below.

a[0]	35
a[1]	40
a[2]	20
a[3]	57
a[4]	19

<u>Declaration of One Dimensional Arrays:</u>

The general form of array declaration is: **datatype variable-name[size];**

The data type specifies the type of element that will be contained in the array such as int, float or char and size indicates the maximum number of elements that can be elements that can be stored inside the array (or size of an array)

For example:

1) **float** height[50];

'float' declares the height to be an array containing 50 real elements. Any subscripts 0 to 49 are valid.

2) int group[10];

'int' declares the group as an array to contain a maximum of 10 integers constants

3)char name[10];

'char' declares the size of the name array to be only 10 characters.

Initializing of One-Dimensional

We can initialize (store values into array) an array in two methods.

Direct Initialization (Compile time initialization)

Indirect Initialization (Runtime Initialization).

Method1 (Direct Initialization/Compile time Initialization): We can initialize the elements of arrays in the same way as the ordinary variables when they are declared. The general form of initialization of arrays is:

Syntax:

Datatype array-name[size] = {list of values};

The values in the list are separated by commas.

Examples:

int number[3] = {0, 0, 0};

will declare the variable number as an array of size 3 and will assign zero to each element.

float total[5] = {0.5, 15.75,-10};

will initialize the first three elements to 0.5, 15.75, and -10.0 and the remaining two elements to zero.

int counter[ ] = {1,1,1,1};

will counter array to contain four elements with values 1. This approach works fine as long as we initialize every element in the array.

Char name[ ] = { ‘J’, ‘o’, ‘h’, ‘n’, ‘\0’};

Declares the name to be array of five characters, initialized with string “John” ending with null character.

Program:

```
#include<stdio.h>
void main( )
{
 int  a[10] ={10,20,30,40,50,60,70,80,90,100};
printf(“%d\n”,a[0]);
printf(“%d\n”,a[1]);
printf(“%d\n”,a[2]);
printf(“%d\n”,a[3]);
printf(“%d\n”,a[4]);
printf(“%d\n”,a[5]);
printf(“%d\n”,a[6]);
printf(“%d\n”,a[7]);
printf(“%d\n”,a[8]);
printf(“%d\n”,a[9]);
}
```

Output:

```
10
20
30
40
50
60
70
80
90
100
```

Method2 (Indirect Initialization/Runtime Initialization):

An array can be explicitly initialized at run time. This approach is usually applied for initializing large arrays. We can use a **scanf to initialize an array to initialize at runtime.**

```
int x[3];
scanf(“%d %d %d”, &x[0], &x[1], &x[2]);
```

Will initialize array elements with the values entered the through the keyboard.

Program:

Write a C program to find the total of the marks given in an array.

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

Output:

enter array elements:
1
2
3
4
5
The sum of the marks is:
15

```
 int  a[5] , sum=0, i;
 printf("enter array elements:");
for(i=0; i<5; i++)
{
 scanf("%d", &a[i]);
}
for(i=0;  i<5; i++)
{
sum=sum+a[i];
}
printf("the sum of the marks is: %d", sum);
}
```

Two Dimensional Arrays (2-D Array):

A two-dimensional array can be considered as a table which will have x number of rows and y number of columns. In other words, a two dimensional array is collection of single dimensional arrays. It also collection of elements of similar type stored in sequential rows and columns format.

The general form of 2-D array declaration is:

datatype variable-name[row-size][column-size];

A two-dimensional array a, which contains three rows and four columns can be shown as follows:

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Example:

int a[3][4];

Here row size is 3 and column size is 4 and data type is int(integer)

Consider the following matrix.

```
       11    12    13    20
A=   14    15    16    21
       17    18    19    22
```

The above mentioned matrix is 3 X 4. The matrix elements can be accessed using an array A[m][n], where m represents row number (here m=3) and n represents column number(n=4).
The array elements are represented as follows:
Similarly,
A[0][0]=11
A[0][1] = 12
A[0][2] = 13
A[0][3] = 20
A[1][0] = 14
A[1][1] = 15
A[1][2]=16
A[1][3]=21
A[2][0]=17
A[2][1]=18
A[2][2]=19
A[2][3]=22

So we can declare 2-dimensional array for above matrix as A [3][3].

Initializing of Two-Dimensional Array: two dimensional arrays also can be initialized directly at **1. Compile time** along with the declaration of array itself and **2. Runtime** by using scanf function.

Two-dimensional arrays may be initialized by following their declarations with a list of intial values enclosed in braces.
For example:

```
int marks[2][3] = {0, 0, 0, 1, 1, 1};
```

Initializes the elements of the first row to zero and second row to one. The initialization is done by row. The above statement is equivalently written as

```
int marks[2][3] = {{0,0,0}, {1,1,1}};
                or
int table[2][3] = {
                        {0,0,0},
                        {1,1,1}
                      };
```

When the array is completely initialized with all values, explicitly we need not specify of the first dimension, like below,

```
int marks[ ][3] = {
```

```
                {0,0,0},
                {1,1,1}
              };
```

If the values are missing in an initializer, they are automatically set to zero. For example:

```
int table[2][3] = {
                {1,1},
                {2}
              };
```

This will initialize the first two elements of the first row to one, the first element of the second row to two, and all other elements to zero.

When all the elements are to be initialized to zero, the following short-cut method may be used.

```
int m[3][5] = { {0}, {0}, {0} };
```

<u>Program:</u>

Write a C program to add two matrices using two dimensional arrays. (Runtime initialization).

```
#include <stdio.h>
void main()
{
int a[3][3], b[3][3],c[3][3], i, j;

printf("Enter the elements of matrix1:");
for(i=0;i<3;i++)
{
        for(j=0;j<3;j++)
        {
                scanf("%d",&a[i][j]);
        }
}

printf("Enter the elements of matrix2:");
for(i=0;i<3;i++)
{
        for(j=0;j<3;j++)
        {
                scanf("%d",&b[i][j]);
        }
}

for(i=0;i<3;i++)
{
        for(j=0;j<3;j++)
```

```
        {
                c[i][j] = a[i][j] +b[i][j];
        }
}

printf("The sum of the two matrices is: \n");
for(i=0;i<3;i++)
{
        for(j=0;j<3;j++)
        {
                printf("%d ",c[i][j]);
        }
        printf("\n");
}
getch();
}
```

Advantage of C Arrays

1) Code Optimization: Less code to the access the data.
2) Ease of traversing: By using the for loop, we can retrieve the elements of an array easily.
3) Ease of sorting: To sort the elements of the array, we need a few lines of code only.
4) Random Access: We can access any element randomly using the array.

Disadvantage of C Arrays

1) **Fixed Size:** Whatever size, we define at the time of declaration of the array, we can't exceed the limit. So, it doesn't grow the size dynamically like Linked List which we will learn later.

Pointers: - A pointer is a derived data type in C. It is built from one of the fundamental data type available in C. Pointer is a variable which contains memory address as its value. Pointers are frequently used in C language, as they offer a number of advantages to the programmers. They are:

- Pointers are more efficient in handling arrays and data tables.
- Pointers can be used to return multiple values from a function via arguments.
- The use of pointers to character strings results in saving of data storage space in memory.
- Pointers allow C to support dynamic memory management.
- Pointers reduce the length and complexity of programs.
- They increase the execution speed and thus reduce the program execution time.

Pointers provide an efficient tool for manipulating dynamic data structures such as linked lists, stacks, queues, and trees.

Declaration and Initialization of pointer variables: -

Since pointer is a variable, it should be declared before we use. The declaration causes the compiler to allocate memory location for the pointer variable. The declaration of a pointer variable takes the following form.

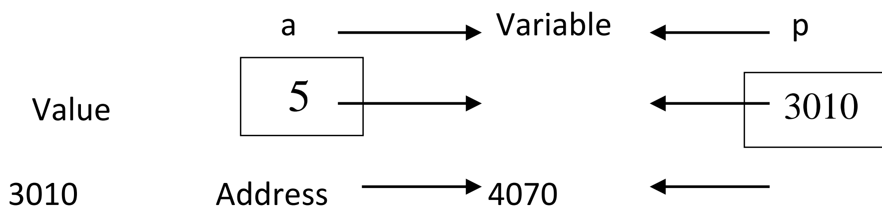
Syntax: data-type * pointer variable name;

Here, the pointer variable preceded by the symbol *, which tells the compiler that it is a pointer variable, and as it is a variable it needs a memory location.

* - This operator is called as “value at address operator” or “indirection operator” or “dereferencing operator”, and it gives the value stored at a particular address.
& - This is called as “Address of operator”, and it gives address of a particular memory.

Eg: int * p; p is a pointer variable that points to an integer data type.
float *q; q is a pointer to a floating-point variable.

Eg: int a = 5;
int *p; declaration of pointer variable
p = & a; initialization of pointer variable



- We can also combine the declaration and initialization as follows:
  int a = 5;
  int *p = &a;

- It is also possible to combine declaration of data variable, pointer variable and the initialization of the pointer variable in one step as follows:
  int a, *p = &a;

- But the following statement is not valid.
  int *p = &a, a;

- We must ensure that the pointer variable always pointing to the corresponding type of data.

```
Eg:     float   a;
        int     x, *p;
          p = &a   // this is not allowed as p is integer pointer and a is float variable.
```

- We could also define a pointer variable with an initial value of NULL or zero.

```
int     *p = NULL;
int     *p = 0;
```

Accessing a variable through its address/pointer: - Once a pointer has been assigned by the address of a variable, then we can access the value of the variable using its pointer.
In C language we have two special operators as

- **Value at address operator/Pointer operator - ***
- **Address of operator - &**

```
Eg:         int     a, *p, x;
            a = 5;
            p = &a;
            x = *p;
```

Here, a, x are integer variables and p is pointer variable pointing to an integer ‘a’. and ‘x’ is assigned by *p. Thus the value of x would be 5. So we can access the value of a variable (a) through its pointer (p) by using value at address operator (*p).

The following program illustrate the use of an indirection operator (*) to access the value pointed to by an integer in different ways.

```
void main( )
{

int     a, *p;
        float   b, *q;

clrscr();

        a = 5;
        p = &a;

        b = 8.2;
        q = &b;
```

```
    printf("\n Value of a is : %d", a);
    printf("\n Value of a is : %d", *p);
    printf("\n Value of a is : %d", *(&a));

    printf("\n %d is stored at %u", a, &a);
    printf("\n %d is stored at %u", *p, p);

    printf("\n Value of b is : %f", b);
    printf("\n Value of b is : %f", *q);
    printf("\n Value of b is : %f", *(&b));

    getch();
}
```

OUTPUT:

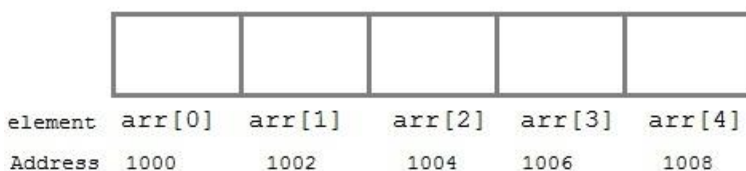
```
 Value of a is : 5
 Value of a is : 5
 Value of a is : 5

 5 is stored at 65522
 5 is stored at 65522

 Value of b is : 8.200000
 Value of b is : 8.200000
 Value of b is : 8.200000
```

Pointers to 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:



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];    //both the statements are equivalent.
```

Now we can access every element of the array arr using p++ to move from one element to another.

NOTE: You cannot decrement a pointer once incremented. p-- won't work.

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", *p);
      p++;
   }

   return 0;
}
```

In the above program, the pointer *p will print all the values stored in the array one by one. We can also use the Base address (a in above case) to act as a pointer and print all the values.

Example – Array and Pointer Example in C

```
#include <stdio.h>
int main( )
{
  /*Pointer variable declaration */
  int *p;
```

```
/*Array variable declaration*/
int val[7] = { 11, 22, 33, 44, 55, 66, 77 } ;

/*Pointer variable initialization */
p = &val[0];

for ( int i = 0 ; i<7 ; i++ )
{
  printf("val[%d]: value is %d and address is %p \n", i, *p, p);

  p++;
}
return 0;
}
```

Output:

val[0]: value is 11 and address is 0x7fff51472c30
val[1]: value is 22 and address is 0x7fff51472c34
val[2]: value is 33 and address is 0x7fff51472c38
val[3]: value is 44 and address is 0x7fff51472c3c
val[4]: value is 55 and address is 0x7fff51472c40
val[5]: value is 66 and address is 0x7fff51472c44
val[6]: value is 77 and address is 0x7fff51472c48

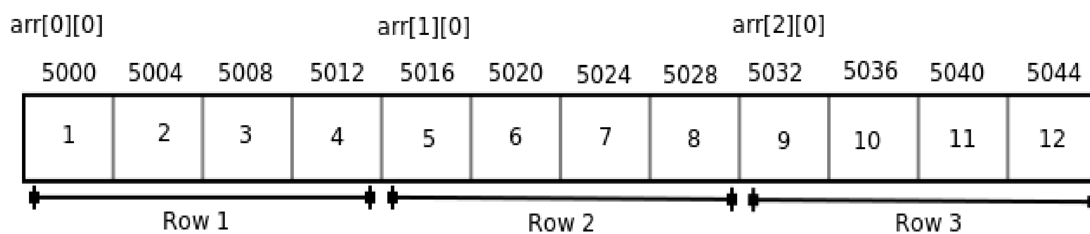
Points to Note:

1) While using pointers with array, the data type of the pointer must match with the data type of the array.
2) You can also use array name to initialize the pointer like this:
p = var;
because the array name alone is equivalent to the base address of the array.
val==&val[0];
3) In the loop the increment operation(p++) is performed on the pointer variable to get the next location (next element's location), this arithmetic is same for all types of arrays (for all data types double, char, int etc.) even though the bytes consumed by each data type is different.

Pointers and two dimensional Arrays: In a two dimensional array, we can access each element by using two subscripts, where first subscript represents the row number and second subscript represents the column number. The elements of 2-D array can be accessed with the help of pointer notation also. Suppose arr is a 2-D array, we can access any element arr[i][j] of the array using the pointer expression *(*(arr + i) + j). Now we'll see how this expression can be derived. Let us take a two dimensional array arr[3][4]:

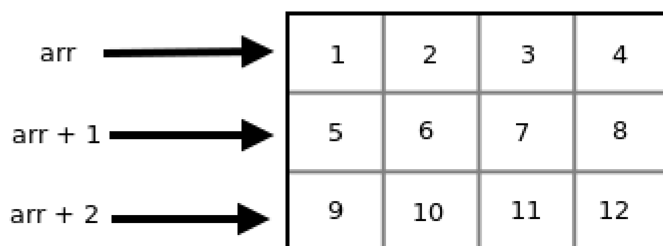
```
int arr[3][4] = { {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12} };
```

The following figure shows how the above 2-D array will be stored in memory.



Each row can be considered as a 1-D array, so a two-dimensional array can be considered as a collection of one-dimensional arrays that are placed one after another.

We know that the name of an array is a constant pointer that points to 0th 1-D array and contains address 5000. Since arr is a 'pointer to an array of 4 integers', according to pointer arithmetic the expression arr + 1 will represent the address 5016 and expression arr + 2 will represent address 5032.
So we can say that arr points to the 0th 1-D array, arr + 1 points to the 1st 1-D array and arr + 2 points to the 2nd 1-D array.



arr - Points to 0th element of arr - Points to 0th 1-D array - 5000
arr + 1 - Points to 1th element of arr - Points to 1nd 1-D array - 5016
arr + 2 - Points to 2th element of arr - Points to 2nd 1-D array - 5032

arr + i Points to i^{th} element of arr -> Points to i^{th} 1-D array

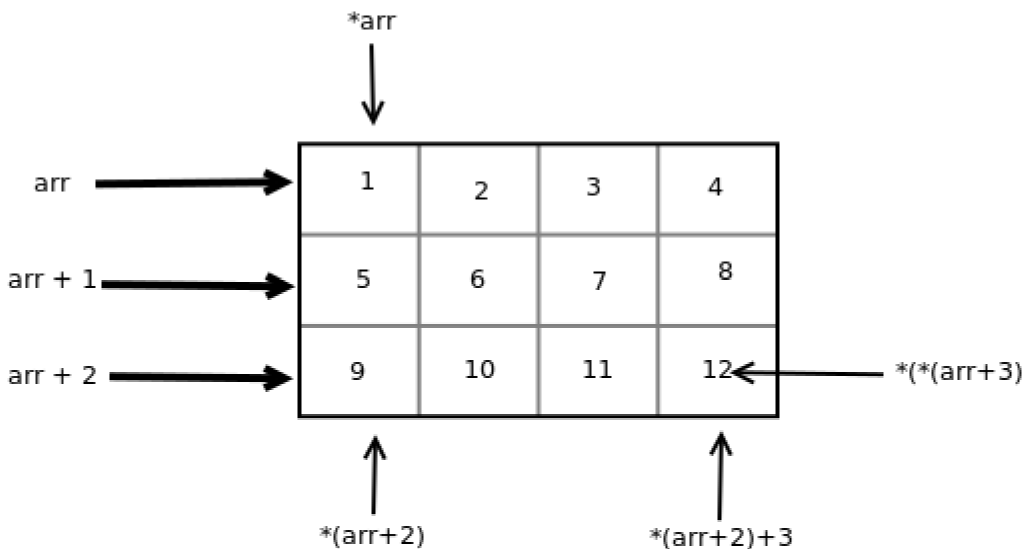
- Since arr + i points to ith element of arr, on dereferencing it will get ith element of arr which is of course a 1-D array. Thus the expression *(arr + i) gives us the base address of ith 1-D array.

- We know, the pointer expression *(arr + i) is equivalent to the subscript expression arr[i]. So *(arr + i) which is same as arr[i] gives us the base address of ith 1-D array.

***(arr + 0) - arr[0] - Base address of 0th 1-D array - Points to 0th element of 0th 1-D array - 5000**
***(arr + 1) - arr[1] - Base address of 1st 1-D array - Points to 0th element of 1st 1-D array - 5016**
***(arr + 2) - arr[2] - Base address of 2nd 1-D array - Points to 0th element of 2nd 1-D array - 5032**

- To access an individual element of our 2-D array, we should be able to access any jth element of ith 1-D array.
- Since the base type of *(arr + i) is int and it contains the address of 0th element of ith 1-D array, we can get the addresses of subsequent elements in the ith 1-D array by adding integer values to *(arr + i).
- For example *(arr + i) + 1 will represent the address of 1st element of 1stelement of ith 1-D array and *(arr+i)+2 will represent the address of 2nd element of ith 1-D array.
- Similarly *(arr + i) + j will represent the address of jth element of ith 1-D array. On dereferencing this expression we can get the jth element of the ith 1-D array.

arr	**Points to 0th 1-D array**
***arr**	**Points to 0th element of 0th 1-D array**
(arr + i)	**Points to i^{th} 1-D array**
***(arr + i)**	**Points to 0th element of ith 1-D array**
***(arr + i) + j)**	**Points to j^{th} element of ith 1-D array**
***(*(arr + i) + j)**	**Reprents the value of jth element of ith 1-D array**



Unit – IV

Functions: Introduction, types of functions, built-in Functions-String Functions, Date Functions.

User-defined functions: Introduction, Elements of Functions-Function Declaration, function definition, function calls, Parameter Passing-call by value and call by reference, Recursion, Passing arrays to functions.

Functions

Introduction

It is possible to code any program using main function, but it may become too large and complex and as a result, the task of debugging, testing and maintaining becomes difficult. If a program is divided into functional parts, then each part may be independently coded and later combined into a single unit. These sub programs called functions, which are much easier to understand, debug and test.

A **function** is a self-contained block of one or more statements that perform a particular task. Every C program can be thought of as a collection of these functions, and one of which must be main( ) function.

The execution of the program always starts and ends with main function and main function can call other functions. A **called function** can receive control from a **calling function**, when the called function completes its execution it returns the control back to the calling function. The communication between calling function and called function is by **passing parameters** at the time of calling.

Advantages of functions:

- It facilitates top-down modular programming; here the problem can be factored into understandable and manageable steps.
- Reusing of the code: The length of the source program can be reduced by using functions at appropriate places. i.e writing functions avoids rewriting of same code over and over.
- Using functions, it becomes easier to write programs and keep track of what they are doing.
- The functions are much easier to understand and test.
- C comes with rich and valuable set of library functions that makes programmer’s work easier.

Types of functions:

C functions can be classified into two categories.

- Library functions (built-in functions)
- User-defined functions

The main distinction between these two categories is that library functions are not required to be written by the user where as a user-defined function has to be developed by the user at the time of writing a program.

Library function	User defined functions
Library function are pre-defined functions	User defined functions are the function which are created by user as per the requirement.
Library function are part of header file	User defined functions are part of a program
In Library function, the name of function is given by the developers	In User defined functions, the name of function is decided by the user.
In Library function, name of function cannot be changed. Example : SIN, COS, Power	In User defined function, name of function can be changed any time Example : fibo, add
Examples: printf(),scanf(),sqrt(),strcat() etc..,	Examples: main(), Add(),Sub(),Mul() etc..,

Built-in functions or Library functions: we have various library functions categorized as mathematical functions, date functions, I/O functions, string handling functions conversion functions etc., but here we will have date functions and string functions.

- **String functions:**

A string is an array of characters. (Or) A string is a one-dimensional array of characters terminated by a null character (‘\0’).

Eg: “INDIA” “WELCOME”

Each character of the string occupies one-byte of memory. The characters of the string are stored in contiguous memory locations.

Declaration and Initialization of Strings: -

Syntax: char string-name [size];

Eg: char city [20];

Character arrays (strings) may be initializing when they are declared (compile time initialization).

Eg:

- We can initialize a character array as element by element and last character should be the null character.
  char city [10] = { ‘H’, ‘Y’, ‘D’, ‘E’, ‘R’, ‘A’, ‘B’, ‘A’, ‘D’, ‘\0’ } ;
- We can initialize a character array as total string at a time.
  char city [10] = “HYDERABAD”;
- We can initialize a character array without specifying the size, the compiler automatically determines the size.
  char city [ ] = “HYDERABAD”;

The memory representation of this character array is as follows:

0	1	2	3	4	5	6	7	8	9
H	**Y**	**D**	**E**	**R**	**A**	**B**	**A**	**D**	**\0**

Reading Strings: -

➢ The input function scanf ( ) can be used with %s format specification to read a string.

Eg: char name[30];

scanf(“%s”, name);

➢ The scanf() function to read multi word strings is doesn’t consider white spaces. It terminates the input string at first occurrence of white space. So multi word strings can’t read by using scanf() function.

Eg: char city[30];

scanf(“%s”, city);

If the input string is “NEW DELHI”, then the variable city can store only “NEW”, it ignores the rest because white space is occurred after NEW.

- **To read multi word strings**, we can use **gets( )** function
  Eg: char city[20];

  gets(city);

- **To print strings**, we can use either **gets( )** function or printf function.
  Eg: char city[20];

  gets(city);

  puts(city);

String handling functions: - Operators can’t work with strings directly. So to manipulate strings we have a large number of string handling functions in C standard library and the responsible header file is *“string.h”.* Some of those functions are:

String functions	Description
strcat ( )	Concatenates str2 at the end of str1
strncat ( )	Appends a portion of string to another
strcpy ( )	Copies str2 into str1
strncpy ( )	Copies given number of characters of one string to another
strlen ( )	Gives the length of str1
strcmp ( )	Returns 0 if str1 is same as str2. Returns <0 if strl < str2. Returns >0 if str1 > str2

strcmpi ( )	Same as strcmp() function. But, this function negotiates case. “A” and “a” are treated as same.
strstr ( )	Returns pointer to first occurrence of str2 in str1
strrstr ( )	Returns pointer to last occurrence of str2 in str1
strlwr ( )	Converts string to lowercase
strupr ( )	Converts string to uppercase
strrev ( )	Reverses the given string
strtok ( )	Tokenizing given string using delimiter

1. **strcpy( ):-** It is to copy one string into another, and it returns the resultant string.

Syntax: **strcpy (string1, string2);**

String2 is copied into string1.

Eg:
```
char city1[10] = “HYDERABAD”;

char city2[12] = “BANGLORE”;

strcpy (city1, city2);
```

Here city2 is copied into city1. So city1= “BANGLORE” and city2=” BANGLORE”. The size of the array city1 should be large enough to receive the contents of city2.

2. **strcmp( ):-** It is to compare two strings to check their equality. If they are equal it returns zero, otherwise it returns the numeric difference between the first non-matching characters in the strings. (i.e. +ve if first one is greater, -ve if first one is lesser).

Syntax: **strcmp (string1, string2);**

Eg: char city1[10] = “HYDERABAD”;

char city2[12] = “BANGLORE”;

strcpy (city1, city2);

Here city1 and city2 are compared, and returns the numeric difference between ASCII value of ‘H’ and ASCII value of ‘B’ as they are not equal.

Strcmp(“RAM”,ROM”); ⍰ It returns some –ve value.

Strcmp(“RAM”,RAM”): ⍰ It returns 0 as they are equal.

3. **strcat ( ):-** This function is used to join two strings together, and it returns the resultant string.

Syntax: **strcat (string1, string2);**

String1 is appended with string2 by removing the null character of string1 and string2 remains unchanged. The resultant string is stored in string1.

Eg: char city1[10] = “HELLO”;

char city2[12] = “WORLD”;

strcat (city1, city2);

Here city2 is appended to city1. So city1= “HELLOWORLD” and city2=”WORLD”.

4. **strlen( ):-** It is to find out the length of the given string and it returns an integer value, that is the number of characters in the given string. It takes only one parameter

Syntax: **strlen (string1);**

It gives the length of string1.

Eg: char city[10] = “HYDERABAD”;

strlen (city); it gives the value 9.

5. **strrev ( ):-**This function is to find out the reverse of a given string.

Syntax: **strrev (string);**

Eg: char city[10] = “HYDERABAD”;

strrev (city); it give the string “DABAREDYH”

Sample program using various string functions

```
#include <stdio.h>
#include <string.h>
void  main()
{
        char s1[10] = "Hello";
        char s2[10] = "World";
                printf("Length of string s1: %d", strnlen(s1));
                printf("Concatenation using strncat: %s", strncat(s1,s2, 3));
                printf("Output string after concatenation: %s",strcat(s1,s2));
                printf("String s1 is: %s", strcpy(s1,s2));
                printf("String s1 is: %s", strncpy(s1,s2,3));

        getch();
}
```

Date Functions:

The C date and time functions are a group of functions in the standard library of the C programming language implementing date and time manipulation operations.

S.no	Function	Description
1	setdate()	This function used to modify the system date
2	getdate()	This function is used to get the CPU time
3	clock()	This function is used to get current system time
4	time()	This function is used to get current system time as structure
5	difftime()	This function is used to get the difference between two given times
6	strftime()	This function is used to modify the actual time format
7	mktime()	This function interprets tm structure as calendar time
8	localtime()	This function shares the tm structure that contains date and time information.
9	gmtime()	This function shares the tm structure that contains date and time information.
10	ctime()	This function is used to return string that contains date and time information

Note: This programs are run in TurboC/C++ compiler

```
// C program to demonstrate setdate() and getdate() methods
#include <dos.h>
#include <stdio.h>
int main()
{
   struct date dt;

   // This function is used to get system's current date
   getdate(&dt);

   printf("System's current date\n");
   printf("%d/%d/%d", dt.da_day, dt.da_mon, dt.da_year);

   printf("Enter date in the format (date month year)\n");
   scanf("%d%d%d", &dt.da_day, &dt.da_mon, &dt.da_year);

   // This function is used to change system's current date
   setdate(&dt);

   printf("System's new date (dd/mm/yyyy)\n")
   printf("%d%d%d", dt.da_day, dt.da_mon, dt.da_year);

   return 0;
}
```

Output

```
System's current date
18/4/2019

Enter date in the format (date month year)
20 4 2018

System's new date (dd/mm/yyy)
20/4/2018_
```

EXAMPLE PROGRAM FOR TIME() FUNCTION IN C. This function is used to get current system time as structure.

```
#include <stdio.h>
#include <time.h>
int main ()
```

```
{
   time_t seconds;
   seconds = time (NULL);

   printf ("Number of hours since 1970 Jan 1st is %ld \n", seconds/3600);
   return 0;
}
```

OUTPUT:

```
Number of hours since 1970 Jan 1st is 374528
```

User-defined functions: -

It is possible to code any program utilizing only main function, but it may become too large and complex and as a result, the task of debugging, testing and maintaining becomes difficult. If a program is divided into functional parts, then each part may be independently coded and later combined into a single unit. These sub programs called functions are much easier to understand, debug and test. These are defined by the user according to the requirement in the application. The user can modify and can create any number of functions based on requirement. The user can certainly understand the internal working of the function.

The syntax of C user-defined function:

```
return-type function-name (type arg1, type arg2, ……)
{
    local variable declarations;
    executable statement1;
    executable statement2;
   return statement;
}
```

- A **function declaration** tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function. A function can also be referred as a method or a sub-routine or a procedure, etc.

- 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 Body** − the function body contains a collection of statements that define what the function does.
- The ***return*** is a keyword which is followed by some expression or value. The return statement returns a value to the calling function and is optional. When there is no return statement, then no value is being returned to the calling function. We can **return only one value at a time**.

Example:

Given below is the source code for a function called max(). This function takes two parameters num1 and num2 and returns the maximum value between the two −

```
/* function returning the max between two numbers */
int max(int num1, int num2) {

  /* local variable declaration */
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

Function Declarations

A function declaration tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts −

return_type function_name (parameter list);

For the above defined function max(), the function declaration is as follows −

int max(int num1, int num2);

Parameter names are not important in function declaration only their type is required, so the following is also a valid declaration −

int max(int, int);

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

Function Calling (Calling a Function):

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When 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.

To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value. For example –

```
#include <stdio.h>

/* function declaration */
int max(int num1, int num2);

int main () {

  /* local variable definition */
  int a = 100;
  int b = 200;
  int ret;

  /* calling a function to get max value */
```

```
  ret = max(a, b);

  printf( "Max value is : %d\n", ret );

  return 0;
}

/* function returning the max between two numbers */
int max(int num1, int num2) {

  /* local variable declaration */
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

<u>Forms of Functions</u>- A function depending on whether the arguments are present or not, and a value is returned or not may belong to one of the following categories.

1. Functions *without* **arguments** and *without* **return values**.

2. Functions *with* **arguments** and *without* **return values**.

3. Functions *without* **arguments** and *with* **return values**.

4. Functions *with* **arguments** and *with* **return values**.

1. <u>Functions *without* **arguments** and *without* **return values**</u>**: -**
   - When a function has no arguments, it does not receive any data from calling function.
   - When a function does not return any value, the calling function doesn't receive any data from called function.
   - There is no data transfer in between the calling function and called function.

Eg:

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

```
        displayline( );
        message( );
        displayline( );

        getch( );
}
displayline( )  // no arguments and no return type for displayline
{
        int i;
        for(i=1; i<=80; i++)
                printf("-");
        printf("\n");
}
message( )
{
        printf(" ***** COME TO LEARN **** \n ");
        printf(" *****  GO TO SERVE ***** \n ");
}
```

OUTPUT:-

```
--------------------------------------------------------------------------
            ***** COME TO LEARN *****
             ***** GO TO SERVE *****
--------------------------------------------------------------------------
```

2. Functions *with* **arguments** and *without* **return values: -**

- The nature of data communication between the calling function and the called function is with arguments but no return values.
- One-way data communication between calling function and called function through arguments.
- The control is transferred to the called function by passing some arguments, after performing the task; the control is back to the calling function without returning any value.

3. Functions *without* **arguments** and *with* **return values: -**

- The function is called without passing any arguments, but after performing the task, it returns some value to the calling function.
- One-way data communication between calling function and called function through return statement.

- The control is transferred to the called function without passing any arguments, after performing the task; the control is back to the calling function by passing some value.

Eg:
```
#include<stdio.h>
void main( )
{
        int f;
        clrscr( );

        f = factorial( );
        printf(" \n Factorial f = %d", f);
        getch( );
}
int factorial( )// without  arguments, but returns some value.
{
        int i, f =1,n;
        printf(" enter a value ");
        scanf("%d", &n);
        for(i=1; i<=n; i++)
                f= f * i ;
        return f;
}
```

OUTPUT:-
```
enter a value  4
Factorial f = 24
```

4. Functions *with* **arguments** and *with* **return values: -**

- Two-way data communication between calling function and called function is by passing arguments and sending back some value.
- The control is transferred to the called function by passing some arguments, after performing the task; the control is back to the calling function by returning some value.
- At the time of function calling the actual arguments are dumped into formal arguments.

Eg:
```
#include<stdio.h>
void main( )
{
        int n, f;
        int factorial(int);                    // function declaration or prototyping
        clrscr( );

        printf(" enter a value ");
```

```
        scanf("%d", &n);
            f = factorial (n);          // function calling statement
        printf(" \n Factorial f = %d", f);
        getch( );
    }
    int factorial( int x)         // function definition with argument, with return value.
    {
        int i, f =1;

        for(i=1; i<=x; i++)
            f= f * i;
        return f;
    }
```

OUTPUT:- enter a value 4
Factorial f = 24

Recursion: -

- When a function *calls itself*, then that process is called as **recursion** and that function is called as recursive function.
- Recursive functions can be used to solve problems where the solution is expressed in terms of successively applying the same solution to subsets of the problems.
- Every recursive functions must has to have two basic properties:
  (i) A termination condition called ***anchor step*** (usually if statement) to avoid infinite process. (ii) A repetition statement called ***recurrence step*** to repeat the function calling process.

- Recursion is of two types. **Direct recursion** and **indirect recursion**.

- When a function calls itself, then it is *direct recursion*. But when function-1 calls function-2 and in turn function-2 calls function-1 then it is *indirect recursion*.

Eg: To calculate the factorial value of number 4, the recursion process is as follows.

```
    void main( )
    {
        int n, f;
        int factorial(int);          // function declaration or prototyping
        clrscr( );
        n = 4;
        f = factorial (n);           // function calling statement
        printf(" \n Factorial f = %d", f);
```

```
        getch( );
}
int factorial( int x)    // function definition with argument, with return value.
{
        int f;
                if( (x = = 1) | | (x = = 0))
                        return 1;
                else
                        return (x * factorial ( x - 1) );
}
```

i.e x * factorial (x - 1) is processed as follows:

```
4 * factorial( 3 )
    3 * factorial ( 2 )
        2 * factorial ( 1 )
                1

4 * 3 * 2 * 1 = 24
```

<u>Parameter passing techniques</u>: -

Parameter passing is a technique used to pass data to a function. Data are passed to a function using one of two techniques: **pass by value** (**call by value**) and **pass by reference** (call **by reference**).

1. <u>Call by value: -</u>

- In call by value mechanism a copy of the data is sent to the function. That is **the values** of actual arguments are being **copied** into formal arguments. And ensures that the original data in the calling function cannot be changed.
- Memory is allocated temporarily for formal parameters and local variables.
- Whatever the modifications are done for formal parameters **will not affect** the actual parameters.

2. <u>Call by Reference-</u>

- In call by reference mechanism the address of the data rather than a copy is sent to the function. That is **the address** of actual arguments is being passed into formal arguments. The called function can change the original data in the calling function.

- When we pass the addresses, the receiving parameters should be pointers to hold these addresses.
- Whatever the modifications are done for formal parameters **will directly affect** the actual parameters.
- C language does not have a true call by reference and it is stimulated by call by address.

Eg:

```
// program for call by value mechanism
#include<stdio.h>
#include<conio.h>
void main( )
{
        int a, b;
        void swap(int, int);            // function declaration or prototyping
        clrscr( );

        printf(" enter two values ");
        scanf("%d %d", &a, &b);
                swap(a, b);                     // function calling statement
        printf(" Values after swap function call are: ");
        printf(" a=  %d \t b = %d ", a, b);
        getch( );
}
void swap( int x, int y)                // function definition.
{
        int temp;

        temp = x;
        x = y;
        y = temp;
}
```

OUTPUT:- enter two values 10 20
Values after swap function call are: a = 10 b=20

Here the **copy of the actual data** a, b is sent to formal parameters of swap function, in swap function these are swapped but the changes made to x, y are **not affect** the values of a, b in main function. Because the copy the data is sent but not original location.

Eg:

```
// program for call by reference mechanism
#include<stdio.h>
#include<conio.h>
```

```
void main( )
{
        int a, b;
        void swap(int, int);            // function declaration or prototyping
        clrscr( );

        printf(" enter two values ");
        scanf("%d %d", &a, &b);
        swap(&a, &b);                   // function calling statement
        printf(" Values after swap function call are: ");
        printf(" a=  %d \t b = %d ", a, b);
        getch( );
}
void swap( int *x, int *y)              // function definition.
{
        int temp;

        temp = *x;
        *x = *y;
        *y = temp;
}
```

```
OUTPUT:-        enter two values     10      20
                Values after swap function call are:  a=20          b=10
```

Here the **address of the actual data** (&a, &b) is sent to formal parameters (*x, *y) of swap function, in swap function instead of creating temporary memory locations for x and y, the same memory locations of a, b referenced by x, y. So changes made to x, y are directly **affects a, b** in main function. So finally swapped values are stored in a, b .

Passing Array to Function in C

In C, when we pass an array to a function. In C, there are various general problems which requires passing more than one variable of the same type to a function. For example, consider a function which sorts the 10 elements in ascending order. Such a function requires 10 numbers to be passed as the actual parameters from the main function. Here, instead of declaring 10 different numbers and then passing into the function, we can declare and initialize an array and pass that into the function. This will resolve all the complexity since the function will now work for any number of values.

If you want to pass a single-dimension array as an argument in a function, you would have to declare a formal parameter in one of following three ways and all three declaration methods

produce similar results because each tells the compiler that an integer pointer is going to be received. Similarly, you can pass multi-dimensional arrays as formal parameters.

Consider the following syntax to pass an array to the function.

Function_name(arrayname); *//passing array*

Methods to declare a function that receives an array as an argument

There are 3 ways to declare the function which is intended to receive an array as an argument.

First way:

return_type function (type arrayname [])

Declaring blank subscript notation [] is the widely used technique.

Second way:

return_type function (type arrayname[SIZE])

Optionally, we can define size in subscript notation [].

Third way:

*return_type function (type *arrayname)*

Example Program: passing an array to function

```
#include<stdio.h>
int minarray (int arr[], int size)
{
	int min=arr[0];
	int i=0;
		for(i=1;i<size;i++)
		{
			if(min>arr[i])
			{
			min=arr[i];
			}
		} //end of for
return min;
} //end of function

int main ()
{
int i=0, min=0;
int numbers [] = {4,5,7,3,8,9}; //declaration of array

min=minarray(numbers,6); //passing array with size
```

```
printf("minimum number is %d \n", min);
return 0;
}
```

UNIT-V

STRUCTURES AND UNIONS: Introduction, Definition of Structure, Declaring Structure Variable, Structure Initialization, Accessing Structure members, Structure Vs Unions, Enumerated Data types

---

STRUCTURES

INTRODUCTION:

Structures help to organize complex data in a more meaning way. It is a powerful concept that we may often need to use in our program design.

Structure is a derived data type whose individual elements can have different data types. Thus a single structure might contain integer elements, floating–point elements and character elements. Pointers, arrays and other structures can also be included as elements within a structure. The individual structure elements are referred to as members.

Definition: A structure is a sequenced collection of heterogeneous related data items that share a common name. Simply the structure is a collection of logically related elements of different data types.

Or

A structure provides a mechanism for packaging data of different data types.

DEFINING & DECLARING A STRUCTURE:

In defining a structure the following points has to be considered.

- The keyword **struct** is used to declares a structure.
- The structure is terminated with a semicolon.
- While the entire definition is considered as a statement, each member is declared independently for its name and type in a separate statement inside the template.

Declaring a Structure:

The **syntax** of a structure may be defined as

```
Struct structure_name
{
datatype member 1;
datatype member 2;
datatype member 2;
------------
datatypemember m;
 };
```

In this declaration, **struct** is a required keyword; structure_name is a name that identifies structures.

The individual members can be ordinary variables, pointers, arrays or other structures. The member names within a particular structure must be different from one another.

For example:

```
struct student
{
intrno;
char name [80];
float percent;
};
```

DECLARING A STRUCTURE VARIABLE:

After defining a structure format we can declare variables of that type. A structure variable declaration is similar to the declaration of variable of any other data types. It includes the following elements.

1. The keyword struct
2. The structure name.
3. List of variable names separated by commas.
4. A terminating semicolon.

Method I:

Struct tag_name var1, var2, …….,var n;

Example:

To declare the structure variable s1 and s2 as follows:

struct student stu1, stu2;

s1 and s2 are structure 00type variables whose composition is identified by the structure student.

Method II:

It is possible to combine the declaration of the structure composition with that of the structure variable as shown below.

```
struct tag_name
{
member 1;
member 2;
-------
member m;
} variable 1, variable 2 --------- variable n;
```

Example:

```
struct student
 {
  int rno;
```

```
    char name [80];
    float percent;
}stu1,stu2;
```

The s1, s2, are structure variables of type student.

STRUTURE INITIALIZATION:

Like any other data type, a structure variable can be initialized at compile time. To initialize structure variables we have use structure-variable & member using ‘.’ Operator.

Method I: Direct initialization

a) **Struct _variable.member1=value;**
   Struct _variable.member2=value;
   .
   .

b) **Struct** tag_name var_name={val1, val2, ….., val n};

Example1:

```
struct student
{
    int rno;
    char name [20];
    float percent;
}s1;
```

a)

```
s1.rno=101;
s1.name="Ramana";
s1.percent=88.8;
```

b)

```
struct student s1 = { 105," ravi", 80.9};
```

Example2:

```
struct Patient
{
   float height;
   int weight;
   int age;
};
```

```
struct Patient p1;
a)
p1.height = 180.75;     //initialization of each member separately
p1.weight = 73;
p1.age = 23;

b)
struct Patient p1 = { 180.75 , 73, 23 };    //initialization
```

Method II: Runtime initialization

Runtime initialization of a structure variable can be done by using scanf( ) statement.

Eg:

```
struct Patient
{
   float height;
   int weight;
   int age;
};

printf("Enter the values for structure");
Scanf("%d%s%f",&s1.rno, s1.name, &s1.percent);
```

DECLARING STRUCTURE MEMBERS USING ARRAY:

It is also possible to define an array of structure that is an array in which each element is a structure. The procedure is shown in the following example:

```
struct student
{
        intrno;
        char name [20];
        float percent;
} stu [5];
```

In this declaration stu is a 5- element array of structures.
It means each element of stu represents an individual student record.

ACCESSING STRUCTURE MEMBERS:

The members of a structure are usually processed individually, as separate entities. Therefore, we must be able to access the individual structure members. A structure member can be accessed by writing

variable. member;

E.g. if we want to print the detail of a member of a structure then we can write as

printf("%s", st.name);
printf("%d", st.rno) and so on.

Example program:

// program on storing Student info using structures

```
#include<stdio.h>
struct student
{
intrno;
char name[50];
float percentage;
}student1,student2;

void main()
{
printf("Enter student 1 details: RNO, NAME, PERCENTAGE:\n");
scanf("%d%s%f",&student1.rno,&student1.name,&student1.percentage);

printf("Enter student 2 details: RNO, NAME, PERCENTAGE:\n");
scanf("%d%s%f",&student2.rno,&student2.name,&student2.percentage);

printf("The student1 details are:\n");
printf("%d\t%s\t%f",student1.rno,student1.name,student1.percentage);

printf("\n The student2 details are:\n");
printf("%d\t%s\t%f",student2.rno,student2.name,student2.percentage);
}
```

Output:

```
Enter student 1 details: RNO, NAME, PERCENTAGE:
10
Sailaja
85.5
Enter student 2 details: RNO, NAME, PERCENTAGE:
20
Raj
87.5
The student1 details are:
10	Sailaja	85.5
The student2 details are:
20	Raj	87.5
```

Array of Structure

We can also declare an array of structure variables. in which each element of the array will represent a structure variable.

Example : struct employee emp[5];

The below program defines an array emp of size 5. Each element of the array emp is of type Employee.

```
#include<stdio.h>
#include<conio.h>
struct Employee
{
    char ename[10];
    int sal;
};

struct Employee emp[5];
void main()
{
int i, j;
    for(i = 0; i < 3; i++)
    {
        printf("\nEnter %dst Employee record:\n", i+1);
        printf("\nEmployee name:\t");
        scanf("%s", emp[i].ename);
        printf("\nEnter Salary:\t");
        scanf("%d", &emp[i].sal);
    }
    printf("\nDisplaying Employee record:\n");
    for(i = 0; i < 3; i++)
    {
        printf("\n Employee name is %s", emp[i].ename);
        printf("\n Slary is %d", emp[i].sal);
    }
getch();
}
```

UNIONS

INTRODUCTION:

Union, like structures, contains members whose individual data types may differ from one another. However, the all the members of a union share the same storage area within the computer's memory, whereas each member within a structure is assigned its own unique storage area. Thus, unions are used to conserve memory.

Definition: A union is a sequenced collection of heterogeneous related data items that share a common name but differs in memory allocation. Each member in structure owns its storage space whereas all the members of a union use same location.

DECLARING A UNION:

The **syntax** of a structure may be defined as

```
Union union_name
{
datatype member 1;
datatype member 2;

------------
member m;
 };
```

In this declaration, **union** is a required keyword; union_name is a name that identifies union of this type.

The individual members can be ordinary variables, pointers, arrays or other structures. The member names within a particular union must be distinct from one another, though a member name can be same as the name of a variable defined outside of the union.

For example:

```
union student
{
intrno;
char name [80];
float percent;
};
```

Declaring a union variable:

Method I:

uniontag_name var1, var2, ……., var n;

Example:

To declare the union variable s1 and s2 as follows:

union student s1, s2;

s1 and s2 are union type variables whose composition is identified by the union student.

Method II:
It is possible to combine the declaration of the union composition with that of the union variable as shown below.

```
uniontag_name
{
member 1;
member 2;
-------
member m;
} variable 1, variable 2 --------- variable n;
```

Example:

```
union student
 {
  intrno;
  char name [80];
  float percent;
} s1,s2;
```

The s1, s2, are union variables of type student.

ASSIGNING OR INITIALIZING VALUES:

Method I:
union _var. member=value;

Example:

```
s1.rno=101;
s1.name=”ravi”;
s1.percent=80.9;
```

Method II:

Uniontag_namevar_name={val1, val2, ….., val n};

Example:
union student s1 = { 101,“ ravi”, 80.9};

It is also possible to define an array of union that is an array in which each element is a union. The procedure is shown in the following example:

```
union student
{
	intrno;
	char name [80];
	float percent;
} st [100];
```

In this declaration st is a 100- element array of union.

It means each element of st represents an individual student record.

ACCESSING A UNION:

The members of a union are usually processed individually, as separate entities. Therefore, we must be able to access the individual union members. A union member can be accessed by writing

variable.member;

E.g. if we want to print the detail of a member of a union then wecan write as
printf(“%s”, st.name);
or
printf(“%d”, st.rno)

Example program:

```
// Employee details using union

#include <stdio.h>
union employee
{
	longintempid;
	charempname[50];
	float salary;
}emp;
void main()
{

	printf("\n\t Enter employee id : ");
	scanf("%d", &emp.empid);
	printf("\n\n Employee id : %d", emp.empid);

	printf("\n\n\t Enter employee name : ");
	scanf("%s", emp.empname);
	printf("\n\n Employee name : %s", emp.empname);
```

```
        printf("\n\n\t Enter employee salary : ");
        scanf("%f", &emp.salary);
        printf("\n\n Employee salary: %f", emp.salary);

}
```

DIFFERENCES BETWEEN STRUCTURE AND UNION:

Structure	Union
A structure is a sequenced collection of heterogeneous related data items that share a common name.	A union is a sequenced collection of heterogeneous related data items that share a common name but differs in memory allocation
The keyword **struct** is used to define a structure	The keyword **union** is used to define a union.
Example structure declaration: struct student { int rno; float percentage; }s;	Example union declaration: union stu { int rno; float percentage; }u;
In Structures, the compiler allocates the memory for each member i.e., The size of structure is greater than or equal to the sum of sizes of its members.	In Union, the compiler allocates the memory by considering the size of the largest member memory i.e., size of union is equal to the size of largest member.
Each member within a structure is assigned unique storage	In union memory allocated is shared by individual members of union.
In Structure, the address of each member will be in ascending order	In unions the address is same for all the members of a union.
In Structure, altering the value of a member will not affect other members of the structure.	In union altering the value of any of the member will alter other member values.

Enumerated Data Types

An enumerated type (also called enumeration or enum) is a data type consisting of a set of named values called elements, members or enumerators of the type. The enumerator names are usually identifiers that behave as constants in the language. A variable that has been declared as having an enumerated type can be assigned any of the enumerators as a value.

enum identifier {value_1, value_2,…….., value_n};

Here the "identifier" is a user-defined enumerated data type which can be used to declare variables that can have one of the values enclosed within the braces. These values are known as enumerated constants.

enum identifier v1, v2, …. vn;

The enumerated variables v1, v2, …vn can only have one of the values value_1, value_2, …. value_n.

The assignment of the following types are valid:

```
V1=value_3;
V5=value_1;
```

An example:

```
enum day {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
enum day week_start, week_end;

week_start= Monday;

week_end= Friday;
```

The compiler automatically assigns integer digits beginning with 0 to all the enumeration constants, i.e. value_1 assigned to 0, value_2 assigned to 1, and so on. However, the automatic assignments can be overridden by assigning values explicitly to the enumeration constants.
For example:

```
Enum day {Monday=1, Tuesday, Wednesday, Thursday, Friday, Saturday};
```

Here, the constant Monday is assigned the value of 1. The remaining constants are assigned values that increase successively by 1.

Example Program:

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

```
{
enum Days{Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};

Days TheDay;

int j = 0;
printf("Please enter the day of the week (0 to 6)\n");
scanf("%d",&j);
TheDay = Days(j);

if (TheDay == Sunday || TheDay == Saturday)
printf("It is the weekend\n");
else
printf("It is the working day\n");

}
```