COMPUTATIONAL PROBLEM SOLVING

Unit-01

Problem Solving and Introduction to C

Introduction To C

History of C language is interesting to know. Here we are going to discuss a brief history of the c language. **C programming language** was developed in 1972 by Dennis Ritchie at bell laboratories of AT&T (American Telephone & Telegraph), located in the U.S.A. **Dennis Ritchie** is known as the **founder of the c language**. It was developed to overcome the problems of previous languages such as B, BCPL, etc.

Initially, C language was developed to be used in **UNIX operating system**. It inherits many features of previous languages such as B and BCPL.

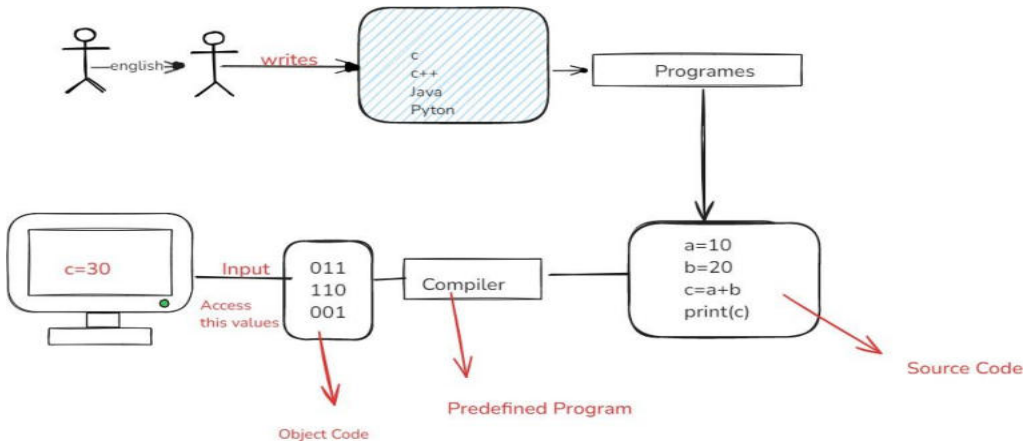
What is Language: Language is Nothing but a Communication Channel b/w 2 People.

What is a Computer Language: Computer Language is also a Communication Channel B/W Person of the System/Electronic device.

What is the need of Computer Language:-

User → High Level Language.(English or Telugu).

System → Machine Language.(0+1 Binary Language).



- Header File Section

- Define Section
- Main Function
  {
  - - - -
  }

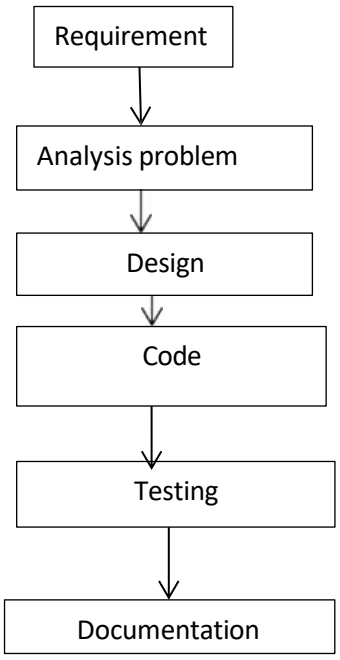
Ex:-// Write a Sample C program to print HELLO WORLD

```
#include<stdio.h>

Int main()

{

Printf("HELLO WORLD");

return  0;

}
```

Steps to Solve Problems

Sometimes it is not sufficient just to cope with problems. We have to solve that problem. Most people are involving to solve the problem. These problem are occur while performing small task or making small decision. So, Here are the some basic steps to solve the problems

1. Defing the Problem
2. Designing the Algorithm
3. Coding the program
4. Testing and Debugging the program
   - Syntax error
   - Logical error
   - Runtime error
5. Implementing the Program
6. Maintaing and upgrading the Program



Algorithm:-

Program design process has 2/phases

1. **Problem Solving Phase:** Creates on Algorithms that Solves Problem.
2. **Implementation Phase:** Translates algorithm into programing language.

Algorithm:-or (Step-Form)

An algorithm in C is a step-by-step procedure to solve a specific problem or perform a task, implemented using the C programming language. It's a set of instructions that takes some input, processes it, and produces an output. In C, algorithms are typically implemented as functions or a series of functions that work together to achieve a goal.

It Algorithm written in English like sentence then it is called "Pseudo Code".

Properties of algorithm

Algorithms in C programming are distinguished by several key features that make them fundamental to creating effective & efficient software. Here's a detailed look at these features:

Definiteness: Each step of an algorithm is clear & unambiguous. In C programming, this means that every operation, from variable declaration to loops and conditionals, must be precisely defined to avoid errors during execution.
Finiteness: Algorithms must have a finite number of steps. This ensures that they eventually terminate & produce a result after executing their defined steps a certain number of times.
Input: Algorithms take zero or more inputs. In the context of C programming, these inputs could be data like numbers, array elements, or any valid data type, provided to the algorithm to process a specific task.

Output: For every input, an algorithm should produce at least one output. This output is the result of processing the given inputs through the set of well-defined instructions.
Effectiveness: Each step of an algorithm must be basic enough to be carried out, in principle, by a person using only pencil & paper. While computers process these steps much faster, the concept emphasizes that steps should be simple & executable.

Ex:-1. A logarithm for finding the Average of 3no's

Step:-1 Start
Step:-2 Read 3 numbers a,b,c
Step:-3 compute sum =a+b+c
Step:-4 compute avg=sum/3

Step:-5 print avg value
Step:-6 stop

Ex:-2. A logarithm for finding the sum of 2no's

Step:-1 Start
Step:-2 Declare 3 integer variables as: a, b and sum
Step:-3 Read a, b Values
Step:-4 calculate sum=a+b
Step:-5 print "sum"
Step:-6 stop

Advantages:-

- It provides core solutions to given problem
- It cases identifications & removal of logical errors in program.
- It facilities algorithm analysis to find out the more efficient solutions to a given problem.
- Promotes effective communication b/w team members.
- Enables analysis of problem at hand.
- Acts as blue print for coding.
- It is easy to debug.
- It uses a definite procedure.
- It is easier to implement.

Dis Advantages: -

- It large algorithms the flow of program controls becomes difficult to track.
- Algorithm lack visual representation of programming constructs like flow chart relatively difficult.
- It is time-consuming.
- The big task is difficult to put in algorithms.

Pseudo code:-

A Pseudocode is defined as a step-by-step description of an algorithm. Pseudocode does not use any programming language in its representation instead it uses the simple English language text as it is intended for human understanding rather than machine reading. Pseudocode is the intermediate state between an idea and its implementation(code) in a high-level language.

What is the need for Pseudocode

Pseudocode is an important part of designing an algorithm, it helps the programmer in planning the solution to the problem as well as the reader in understanding the approach to the problem. Pseudocode is an intermediate state between algorithm and program that plays supports the transition of the algorithm into the program.

pseudo code is a method of representing logic in programming that is independent of any programming language.

Example:

Let's consider an example of implementing a simple program that calculates the average of three numbers using pseudo code in C:

Pseudo code:

1. Start
2. Input three numbers
3. Calculate the sum of the three numbers
4. Divide the sum by 3 to get the average
5. Display the average
6. End

FLOWCHART: -

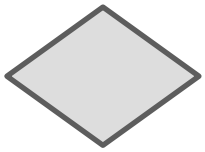
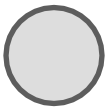
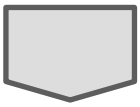
The flowcharts are simple visual tools that help us understand and represent processes very easily. They use shapes like arrows, rectangles, and diamonds to show steps and decisions clearly. If someone is making a project or explaining a complex task, flowcharts can make complex ideas easier to understand. The main purpose of a flow chart is to analyze different processes.

(or)

Graphical representation of any program is called as a flowchart.

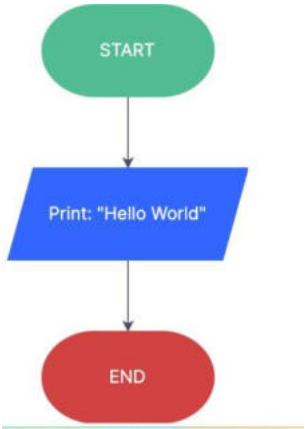
Flowchart Symbols:-

Symbols	Symbol Name	purpose
	Start/stop	Used to represent start and end of the program.
	Flow line	Used to indicate the flow of logic by connecting symbols.
	Input/output	Used for input and output operations.

	processing	Used for Arithmetic/ Mathematical operations.
	Decision	Stands for decision statements in a program where answer is usually Yes or No.
	One-page connector	Used to join different flowline.
	Off-page connector	Used to connect flowchart patterns on different pages.

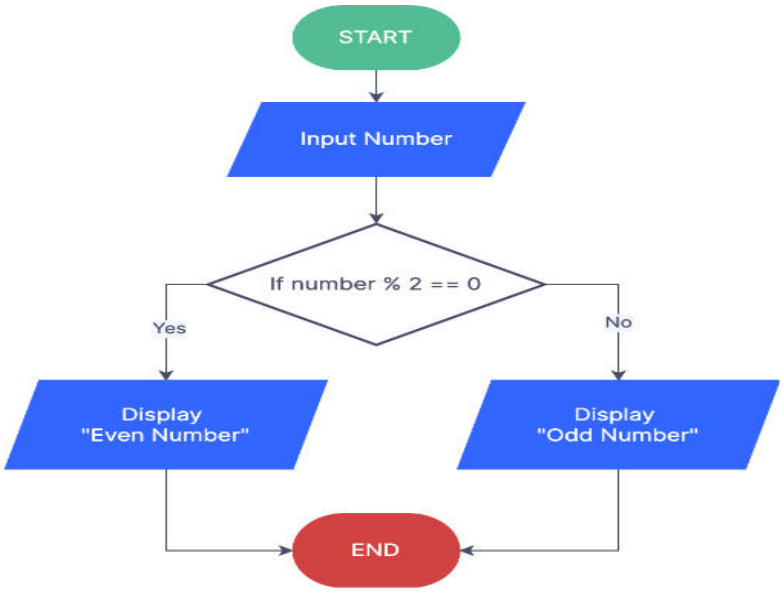
Example 1: Print "Hello World." On-screen

When you first learn a programming language, whether **Python**, **Pascal**, or C, your first program is often a simple "Hello World" program. So, a chart of a simple program of printing a "Hello World" message onto the screen should be like this:



Example 2: Input Number And Check If They Are Odd Or Even

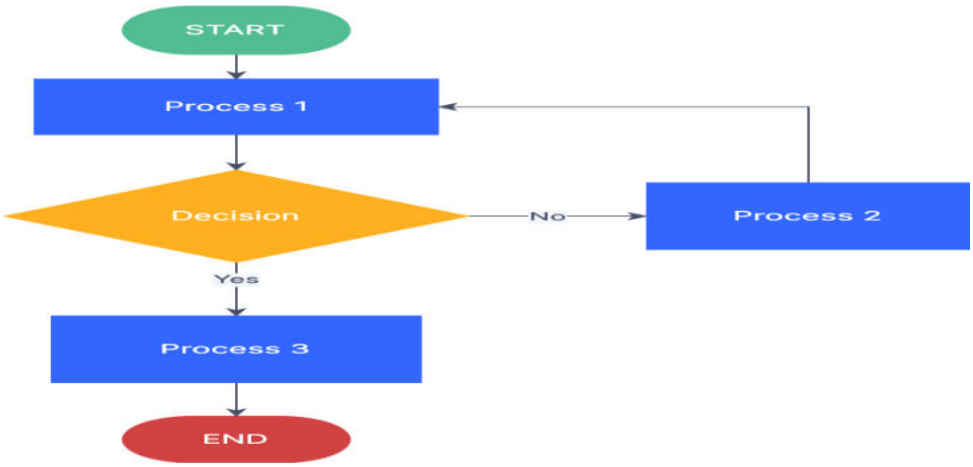
Another relatively simple program is checking odd/even numbers. It is a basic conditional operation that involves: inputting a number, determining whether it is odd or even, printing the result on-screen. The chart should be something like this:



Flowchart Guidelines

To create a flowchart, you must follow the following current standard guideline:

- Step 1: Start the program.
- Step 2: Begin Process 1 of the program.
- Step 3: Check some conditions and take a Decision ("yes" or "no").
- Step 4: If the decision is "yes", proceed to Process 3. If the decision is "no", proceed to Process 2 and return to Step 2.
- Step 5: End of the program.



Additionally, the following can be helpful tips in creating flowcharts as well:

- You can only have one start and one end in your chart, no more, no less.
- On-page connectors are referred to by numbers, while off-page connectors are referred to by alphabetical letters.
- The flow of processes is generally from top to bottom or left to right, not the other way around.
- It would be best not to make the arrows cross each other.

Program Development Environments

Program Development Environments (PDEs) are tools that help you write, compile, and debug C programs. Here's a straightforward look at some popular types of PDEs you can use for C programming.

1. Integrated Development Environments (IDEs)

IDEs combine various tools into one application to simplify the development process. Here are some popular IDEs for C programming:

- **Code::Blocks**
    - **Overview**: A free, open-source IDE that is highly customizable.
    - **Features**:
        - Supports multiple compilers (e.g., GCC, MinGW).
        - Built-in debugging tools.
        - Project management capabilities.
        - User-friendly interface.
    - **Best For**: Beginners and intermediate programmers.
- **Dev-C++**
    - **Overview**: A lightweight IDE designed for C and C++ development.
    - **Features**:
        - Simple interface with basic functionality.
        - Integrated debugger.

        - Easy project setup.
    - **Best For**: Learning and small projects.
- **Eclipse CDT**
    - **Overview**: A powerful IDE that is part of the Eclipse ecosystem.
    - **Features**:
        - Advanced code editing with auto-completion.
        - Integrated version control.
        - Extensive plugins available for added functionality.
    - **Best For**: Larger projects and experienced developers.
- **Visual Studio**
    - **Overview**: A comprehensive IDE mainly for Windows.
    - **Features**:
        - Rich debugging tools and profiling options.
        - Extensive libraries and frameworks.
        - Supports multiple programming languages, including C.
    - **Best For**: Professional development and complex applications.
- **CLion**
    - **Overview**: A modern IDE from JetBrains tailored for C and C++.
    - **Features**:
        - Smart code completion and suggestions.
        - Integrated testing and debugging tools.
        - Built-in support for CMake.
    - **Best For**: Developers looking for advanced features and efficiency.

2. Text Editors with Compilation Support

Text editors provide a simpler interface and are often more lightweight compared to IDEs.

- **Visual Studio Code**
    - **Overview**: A popular, highly customizable code editor.
    - **Features**:
        - Extensions for C/C++ support.
        - Integrated terminal for running commands.
        - Git integration for version control.
    - **Best For**: Flexible and modular development.
- **Sublime Text**
    - **Overview**: A fast and responsive text editor.
    - **Features**:
        - Syntax highlighting for C.
        - Custom build systems for compiling code.
        - Extensible through plugins.
    - **Best For**: Quick edits and lightweight development.
- **Notepad++**
    - **Overview**: A simple text editor for Windows.
    - **Features**:
        - Syntax highlighting and code folding.
        - Can be extended with plugins for additional features.
    - **Best For**: Basic editing tasks and small scripts.

3. Command-Line Tools

Command-line tools are essential for developers who prefer working in terminal environments.

- **GCC (GNU Compiler Collection)**
    - **Overview**: A powerful compiler system for C and other languages.
    - **Features**:

        - Highly configurable with many optimization options.
        - Cross-platform support.
    - **Best For**: Developers comfortable with command-line interfaces.

- **Make**
    - **Overview**: A build automation tool.
    - **Features**:
        - Uses Makefiles to define build processes.
        - Manages dependencies between files.
    - **Best For**: Large projects with multiple source files.

4. Online IDEs

Online IDEs provide a convenient way to code without needing to install software.

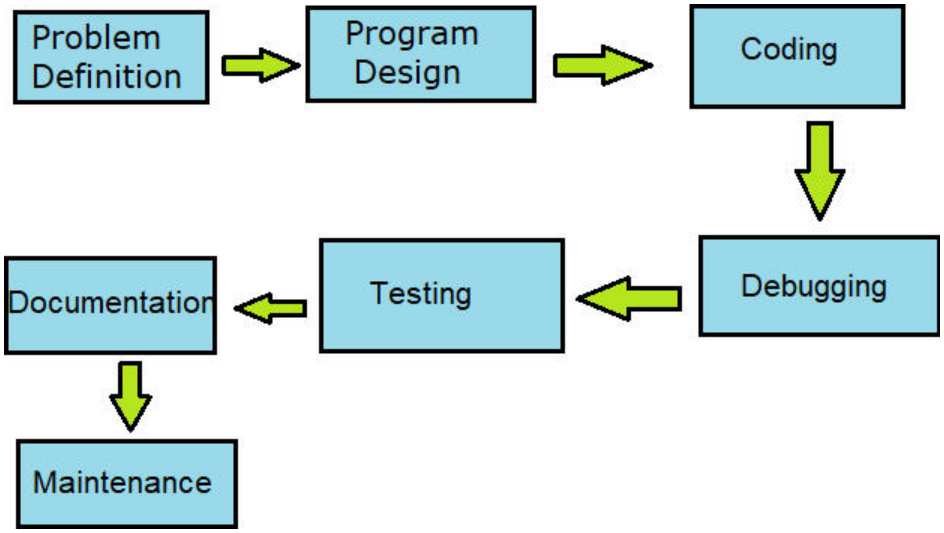
- **Repl.it**
    - **Overview**: An online coding platform supporting multiple languages.
    - **Features**:
        - Run and test code directly in the browser.
        - Collaborative features for teamwork.
    - **Best For**: Learning and quick prototyping.
- **CoderPad**
    - **Overview**: A real-time coding environment for interviews.
    - **Features**:
        - Supports multiple programming languages.
        - Allows for collaborative coding sessions.
    - **Best For**: Technical interviews and coding assessments.

Steps in development of a program

Whenever we develop a program, we follow a sequence of steps. These steps in program development are called phases.

In C, there are following 7 steps to develop a program:-

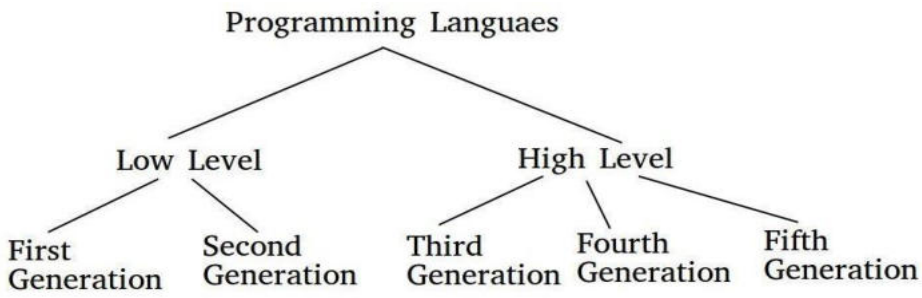
1. Problem Definition
2. Program Design
3. Coding
4. Debugging
5. Testing
6. Documentation
7. Maintenance



Introduction to programming: Programming language and generations

Programming language: - Programming language is a computer language programmers use to develop software programs, scripts or other sets of instructions for computers to execute.

Each phase of development has made the programing language, more user-friendly, easier to use and more powerful.



1. First-Generation Language :(1940-1950's)

The first-generation languages are also called machine languages/ 1G language. This language is machine-dependent. The machine language statements are written in binary code (0/1 form) because the computer can understand only binary language.

Advantages :

1. Fast & efficient as statements are directly written in binary language.

2. No translator is required.

Disadvantages :

1. Difficult to learn binary codes.

2. Difficult to understand – both programs & where the error occurred.

2. Second Generation Language :(1950-1960's)

The second-generation languages are also called assembler languages/ 2G languages. Assembly language contains human-readable notations that can be further converted to machine language using an assembler.

Assembler – converts assembly level instructions to machine-level instructions.

Programmers can write the code using symbolic instruction codes that are meaningful abbreviations of mnemonics. It is also known as low-level language.

Advantages :

1. It is easier to understand if compared to machine language.

2. Modifications are easy.

3. Correction & location of errors are easy.

Disadvantages :

1. Assembler is required.

2. This language is architecture /machine-dependent, with a different instruction set for different machines.

3. Third-Generation Language :-(1960-1970's)

The third generation is also called procedural language /3 GL. It consists of the use of a series of English-like words that humans can understand easily, to write instructions. It's also called High-

Level Programming Language. For execution, a program in this language needs to be translated into machine language using a Compiler/ Interpreter. Examples of this type of language are C, PASCAL, FORTRAN, COBOL, etc.

Advantages :

1. Use of English-like words makes it a human-understandable language.

2. Lesser number of lines of code as compared to the above 2 languages.

3. Same code can be copied to another machine & executed on that machine by using compiler-specific to that machine.

Disadvantages :

1. Compiler/ interpreter is needed.

2. Different compilers are needed for different machines.

4. Fourth Generation Language :(1980-1990's)

The fourth-generation language is also called a non – procedural language/ 4GL. It enables users to access the database. Examples: SQL, Foxpro, Focus, etc.

These languages are also human-friendly to understand.

Advantages :

1. Easy to understand & learn.

2. Less time is required for application creation.

3. It is less prone to errors.

Disadvantages :

1. Memory consumption is high.

2. Has poor control over Hardware.

3. Less flexible.

5. Fifth Generation Language :(1990-Present)

The fifth-generation languages are also called 5GL. It is based on the concept of artificial intelligence. It uses the concept that rather than solving a problem algorithmically, an application can be built to solve it based on some constraints, i.e., we make computers learn to solve any

problem. Parallel Processing & superconductors are used for this type of language to make real artificial intelligence.

Examples: PROLOG, LISP, etc.

Advantages :

1. Machines can make decisions.

2. Programmer effort reduces to solve a problem.

3. Easier than 3GL or 4GL to learn and use.

Disadvantages :

1. Complex and long code.

2. More resources are required & they are expensive too.

Introduction to c:Introduction

History of C Programming Language: C is a procedural programming language initially developed by Dennis Ritchie in the year 1972 at Bell Laboratories of AT&T Labs. It was mainly developed as a system programming language to write the UNIX operating system.

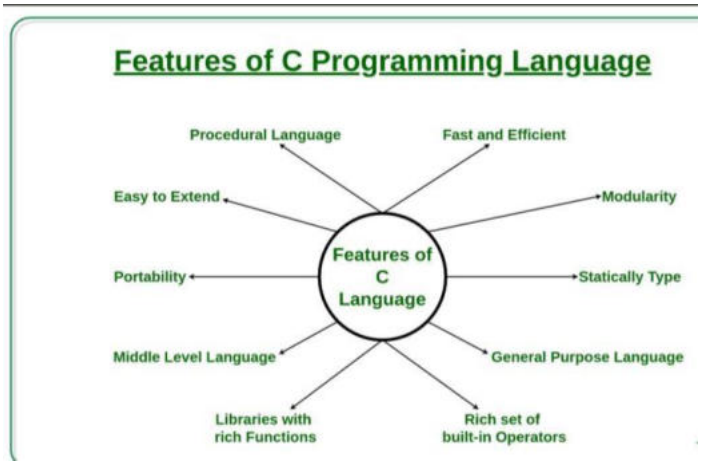
1. The C Program from the beginning, C Was intended to be useful to allow busy programmers to get things done because C is such a Powerful Language.

2. Its Use quickly spread beyond Bell Labs in the late 70's because of its long list of strong features.

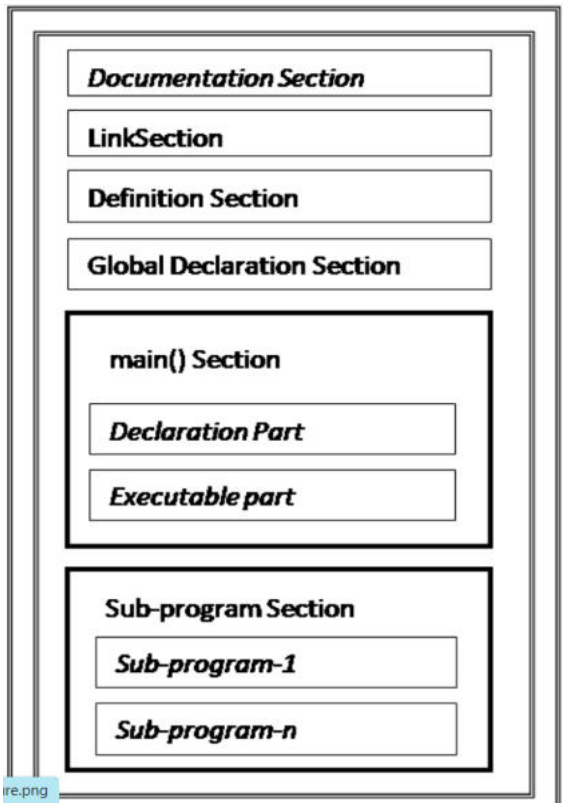
3. 'C' Language is called middle level language because 'C' supports both features of low level language and high level language.

Programming language	Development Year	Developed by
ALGOL	1960	International Group
BCPL	1967	Martin Richards
B	1970	Ken Thompson
Traditional C	1972	Dennis Ritchie
ANSI C	1989	ANSI committee
ANSI/ISO C	1990	ISO Committee

Important Features of C Language?



STRUCTURE OF C PROGRAM



- Every C program follows block structure.
- It is written as a collection of function also known as sub-routine.
- The basic structure of a C program is as follow:

[1] Documentation Section:

- This section contains information about the program such as author name, creation data & time etc.
- It provides guidelines to the program reader.
- It may also include logic of the program.
- This information is written as a comment.

[2] Link Section:

- It includes different library and header files that are required by the program.
- These header files are linked with the program during the linking steps.
- Some standard header files are **<stdio.h>,<conio.h>, <string.h>** etc.

[3] Definition Section:

- This section includes all the constant variables.
- They are defined with the **"#define"**

4] Global Declaration Section:

- All the global variables are defined in this section.
- Global variables can be accessed by any function of the program.
- Their scope is entire program.

[5] main() Section:

- The **main()**section is the main section of the C- program.
- Each C- program must have one and only one main function.
- It is define using **main()**
- Every statement of the C- program should be written in the main section.

- Opening **'{'**specifies starting of the main function and closing **'}'** specifies ending of the main function.
- main section is divided into the following two sections.

1. **Declaration part Executable part**

- Declaration part contains all the local variables to be used in the main section.
- Executable section contains all the executable statement to perform the task.

[6] Sub-program Section:

- This section includes all the sub-program definitions.
- It is used when program is divided into different functions.
- Its order is not important; it can be place before or after the main section.Normally it is written after the main section.

```
#include <stdio.h>

// definition
#define num1 20
#define num2 30

// Global variables
int sum = 0, product = 1;

int main() {

  // subprograms
  // Utility function to perform addition
  int performAddition(int a, int b){
     return a + b;
  }

  // Utility function to perform multiplication
  int performMultiplication(int a, int b){
     return a * b;
  }

  // Perform calculations
  sum = performAddition(num1, num2);
```

```
  product = performMultiplication(num1, num2);

  // Display output
  printf("Sum = %d\n", sum);
  printf("Product = %d\n", product);
  return 0;
}
```

Explanation of the above Program

S.no	Snippet	Type
1.	/* C program to add and product two numbers Author : Alisha */	Documentation
2.	#include <stdio.h>	Link
3.	#define num1 20 #define num2 30	Definition
4.	int sum = 0, product = 1;	Global Variable

5.	int main() {
	…
	…
	}
		Main() function

6.	int performAddition(int a, int b){
	return a + b;
	}
	
	int performMultiplication(int a, int b){
	return a * b;
	}
		Subprograms

KEYWORDS:-

Every 'C' word is classified as either keyword or an identifier. All keywords have fixed meanings and these meanings cannot be changed. keywords are also called as reserved word or predefined word. all keywords must be written in lower case and upper case .,

32 Keywords in C Programming Language

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

C program to demonstrate the use of while keyword

```
#include<stdio.h>

int main(){

int x=1; //initializing the value of x

//loop will execute until the value of x turns greater than 15

while(x<=15){

printf("%d \t",x);

x++; //in each iteration ++operator increment the value of x by 1.

}

return 0;

}
```

Output:-

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

Explanation

In the above example, the value of x is initialized as 1, and the control of the program jumps to the expression of while; if it is true, then it runs the code written in while till the condition fails.

IDENTIFIERS:-

In C programming language, identifiers are the building blocks of a program. Identifiers are unique names that are assigned to variables, structs, functions, and other entities. They are used to uniquely identify the entity within the program. In the below example "section" is an identifier assigned to the string type value.

[OR]

An identifier is a name given to a program element such as variables, constants, functions names, array names and so on.,

Rules to Name an Identifier in C

A programmer has to follow certain rules while naming variables. For the valid identifier, we must follow the given below set of rules.

1. An identifier can include letters (a-z or A-Z), and digits (0-9).
2. An identifier cannot include special characters except the '_' underscore.
3. Spaces are not allowed while naming an identifier.
4. An identifier can only begin with an underscore or letters.
5. We cannot name identifiers the same as keywords because they are reserved words to perform a specific task. For example, print f, scan f, int, char, struct, etc. If we use a keyword's name as an identifier the compiler will throw an error.
6. The identifier must be unique in its namespace.
7. C language is case-sensitive so, 'name' and 'NAME' are different identifiers.

EX:-a+b (special character present).

2abc (first letter is not an alphabet).

Float (key word).

VARIABLES: -

A variable is one whose values does change during the execution of a program. It is a name given to a memory location where the data is stored temporarily. the user is allowed to access the

data from the memory location through the variable name. Variable can store only one value in its location

(or)

Variable is a named memory location.

(or)

Variable is a container which contains values.

Go Variable Naming Rules

A variable can have a short name (like x and y) or a more descriptive name (age, price, car name, etc.).

Go variable naming rules:

- A variable name must start with a letter or an underscore character (_).
- Commas are not allowed, and case sensitive.
- A variable name cannot start with a digit
- A variable name can only contain alpha-numeric characters and underscores (a-z, A-Z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)
- There is no limit on the length of the variable name
- A variable name cannot contain spaces
- The variable name cannot be any Go keywords

Ex:-some valid identifiers:-Salary,sum,stud-name,c123,r,R etc.,

some valid identifiers: - a+b(special character present

2abc(first letter is not on alphabet)

Float(keyword)

There are 3 aspects of defining a variable:

1. Variable Declaration
2. Variable Initialization

3. Variable Assigning values

Variable Declaration

A variable must be declared before it is used in the program. The declaration of variable tells the compiler to reverse the memory space with specified variable names depending on the specified data type.

Syntax:- Data type var1,var2,var3….....var name;

Eg:-int a;
Float b;
Char c;

int,float,char is a data types

a, b, c is variables

Rules:-

1. It can contain Alphabets, digits & Underscore symbols.
2. Should not start with a digit
3. should not use white spaces.
4. variable name should not be a keyword
5. commas are not allowed.
6. these are case sensitive.

Initialization of variable:-

Data type variable name=value;

Ex: int a=10.

Assigning values to a variables

Using pre-defined assignment

Variable=constant Ex:a=10

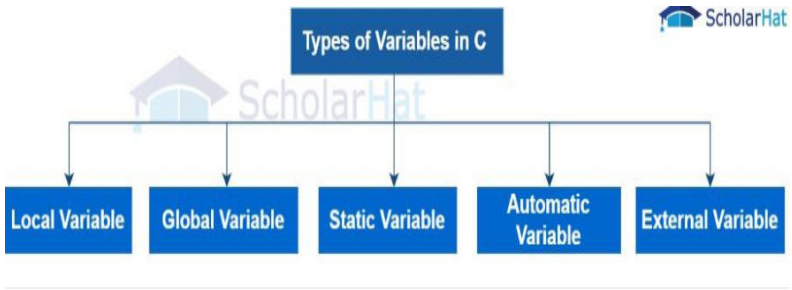
" =variable a=b

" =expression c=a+b

TYPES OF VARIABLES

There are 5 types of variables in C language, which are

1. **Local Variable**
2. **Global Variable**
3. **Static Variable**
4. **Automatic Variable**
5. **External Variable**



1. **Local Variable**

Local variables are declared and initialized at the start of a function or block and allocated memory inside that execution scope. The statements only inside that function can access that local variable. Such variables get destroyed when the control exits from the function.

Ex:-

```
#include <stdio.h>
int main()
{
 int a = 10;
if()
{
Int a=20;
 printf("a”);
}
 Else
 printf(a);
}
```

2. **Global Variable**

- The variables which are defined outside of all functions, usually on top of the program are called as global variables.

- Global Variable hold their values through out the life time of your program and can be accessed in side any of the function defined for the program.
- A Global var can be accessed by the fun
- Global var are initialized automatically by the system.

3. **Static Variable**

   A Variable that is declared with the static keyword called static variable, It retains its value b/w multiple function calls.

   Ex:- void fun()
   {
   Int a=10;
   Static int b=20;
   a++;
   b++;
   pf("%d,%d",a,b);
   }

4. **Automatic Variable**

All variables are declared inside the block are automatic variable by default, use can explicitly declare as automatic variable using auto keyword.

Ex: int main()

{

Int a=10;

Auto int a=20;

}

5. **External Variable**

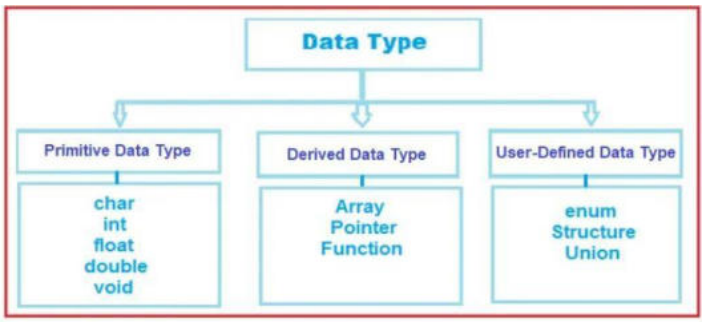
We can share a variable in multiple source files by using an external variable.

DATA TYPES:-

The kind of data that a variable may hold in a program is called as data types (or) The type of data that programmer is going to store in a variable is called data type. Data type specify the size and type of values that can be stored. C language is rich in its data types

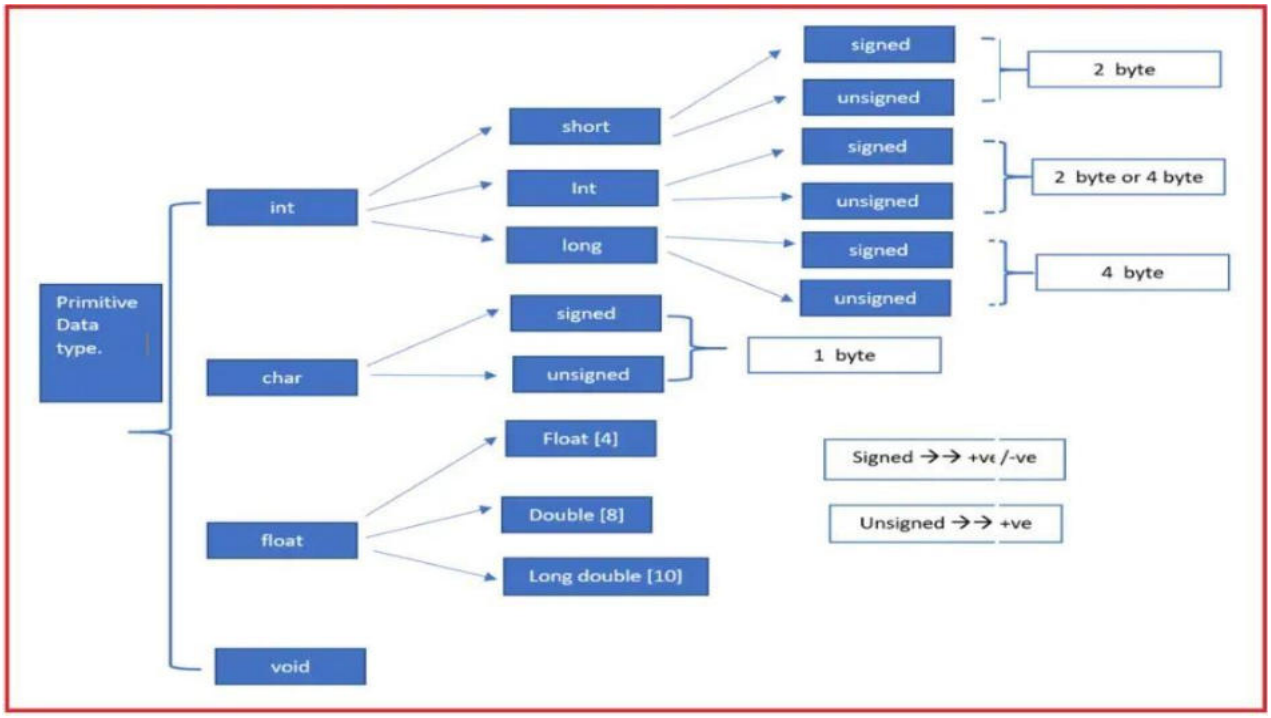
(or)

The kind of data is stored in a variable is called as data type.



Primary Data Types:- It is used for representing a single value only.It is also known as Built- in Data Types.

Classification of Primitive Data Types in C Language:



Integer Family:- The key word 'int' is used to declare integer type variables.

Data type	Size	Range	Format Character
Int(or)Short int(or)Short Signed int	2 Bytes	-32768 to +32767	%d or %i
Unsigned int (or) short signed int	2 Bytes	0 to 65535	%u or %ud
Long int(or) Long Signed Int	4 Bytes	-2147483648 to +2147483647	%ld
Long unsigned int	4 bytes	0 to 4294967295	%lu

Float Family:-

The Keyword ‘float’ and ‘double’ are used to declare float type variables.

Data type	Size	Range	Format Character
float	4 Bytes	3.4e-38 to 3.4e+38	%f or %e
Double	8 Bytes	1.7e-308 to 1.7e+308	%1f or %1e
Long Double	10 Bytes	3.4e-4932 to 1.1e+4932	%L or %Le

Character Family:-

The Keyword char is used to declare character data type variables.it contains alphabets,digit(or) special character.

Data type	Size	Range	Format Character
Char	1 Bytes	-128 to +127	%c
Unsigned	1 Bytes	0 to 255	%c

Void Dat Type:-

The data type void is used to specify an empty set of values and do not occupy memory

Derive Data Type:-

The Data Types which are derived from the primary data types are called as derived data type.

Ex:-arrays, pointers etc.,

User defined data types:

The data types which are created by the user are called as user data type.

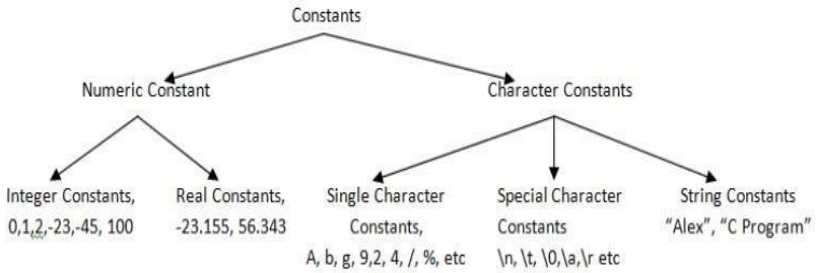
Ex:-unions, enum.

CONSTANTS:-

Constant is an entity that refers to a **fixed value** of data and which cannot be modified. During execution/computation of programming, the value of a constant cannot be altered, it remains constant.

The most common example which can be considered to understand the concept of a constant is “PI”

PI = 3.1415927 whose value is universal and does not change.



Constants are mainly classified into two types

- Numaric :- A constant which is having only digits 0-9 are called as numeric constants. Numeric constant is 2 types
  1. Integer constant 2. Floating point constant.

1. **Integer Constant:-**

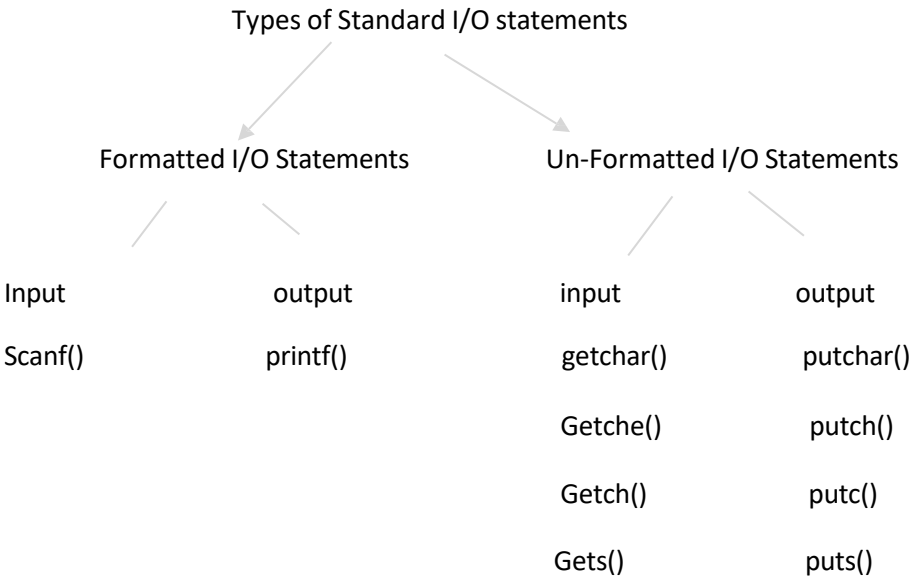
   Any number without a decimal point is called an integer constant.
   Ex:-124,-321 etc.,
2. **Floating point constant:-**
   Floating point constant are also called real numbers. Any number with decimal point is called as floating point constant.
   E.g.: – 0.2334, 45.565, 2334.54456, -0.2323, -0.562E-5

- **Non-Numaric constant:-**
  Character constants have either a single character or group of characters or a character with backslash used for special purpose.
  1. Single character
  2. String character

1. **Single character constant:-**
   A character constant contains only one character enclosed within single quotes. A character constant may contain any alphabet,digit or special symbol.
   Ex:-'a',;&'.
2. **String Character constant:-**
   A String character constant is sequence of one or more character enclosed within double quotes. A character constant may contain any alphabet,digit or special symbol.
   Ex:-"hello","%d","abc".
   Special Symbols:-
   The following special symbols are used C Language. [],(){};:*#etc.

I/O STATEMENTS in C language:-

In language ,the input and output functions are used to receive data from the input device or for sending data to the output device. These functions are standard built-in functions which are available in "stdio.h" and "conio.h" header files.
The I/O functions are classified into 2 types. Shown in below.



Formatted I/O Functions

Formatted I/O functions are used to take various inputs from the user and display multiple outputs to the user. These types of I/O functions can help to display the output to the user in different formats using the format specifiers. These I/O supports all data types like int, float, char, and many more.

Why they are called formatted I/O?

These functions are called formatted I/O functions because we can use format specifiers in these functions and hence, we can format these functions according to our needs.

The following formatted I/O functions will be discussed in this section-

1. printf()
2. scanf()

- **printf():**printf() function is used in a C program to display any value like float, integer, character, string, etc on the console screen. It is a pre-defined function that is already declared in the stdio.h(header file).

Syntax 1: To display any variable value.
Printf ("Format Specifier", var1, var2, …., varn);

- **scanf():**

scanf() function is used in the C program for reading or taking any value from the keyboard by the user, these values can be of any data type like integer, float, character, string, and many more. This function is declared in stdio.h(header file), that's why it is also a pre-defined function. In scanf() function we use &(address-of operator) which is used to store the variable value on the memory location of that variable.

Syntax:

scanf("Format Specifier", &var1, &var2, …., &varn);

Unformatted Input/Output functions

Unformatted I/O functions are used only for character data type or character array/string and cannot be used for any other datatype. These functions are used to read single input from the user at the console and it allows to display the value at the console.

Why they are called unformatted I/O?

These functions are called unformatted I/O functions because we cannot use format specifiers in these functions and hence, cannot format these functions according to our needs.

The following unformatted I/O functions will be discussed in this section-

1. getch()
2. getche()
3. getchar()
4. putchar()
5. gets()
6. puts()
7. putch()

- **getch():**getch() function reads a single character from the keyboard by the user but doesn't display that character on the console screen and immediately returned without pressing enter key. This function is declared in conio.h(header file). getch() is also used for hold the screen.
  Syntax: getch ();
  or
  variable-name = getch();

- **Get cahar():-**this function is used to accept a single character from the keyword and stores it in a variable.
  Syntax:-variable=getchar();

- **Getc():-**this function is used to read character from the file and stores it in a variable if specified.
  Syntax:-variable=getc(filepointer);
- **Gets():-**This function is used to accept a string from the keyboard and stores it in a array variable.it reads a Sequence of characters until ENTER key is pressed.it stores a null character "I/O" at the end of the string.
  Syntax:-gets(variable);
- **Putchar():-**This function is used to print a character on the standard output device i.e. monitor.
  Syntax:-putchar(arg);
- **Putc():-**This function is used to store a character in the file.
  Syntax:-putc(variable,file pointer);
- **Puts():-**This function is used to print a string on the monitor.
  Syntax:-puts(variable);

OPERATOR:-

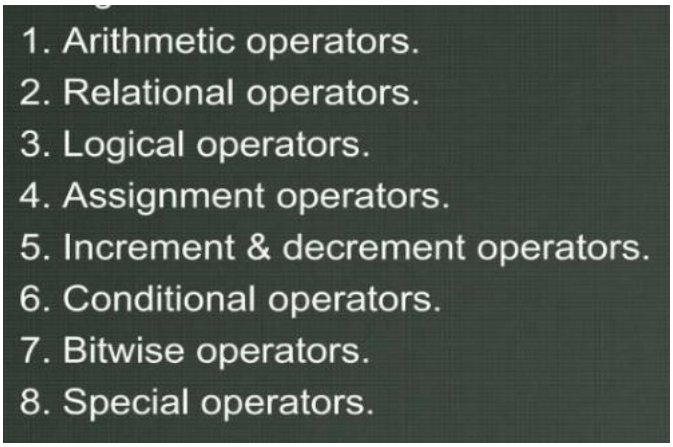
OPERATOR is a symbol which represents some particular action. Operator operates on operands. Operator may be a variable or constant.

Ex:-a+b

A,b ⟶ operands

\+ ⟶ operator

Types of Operators in c:-



1. Arithmetic Operators:-Arithmetic operators are used to perform arithmetic operations like addition, subtraction, multiplication, division etc.,

operator	meaning	example	result
+	addition	A+b 13+5	18
-	subtraction	a-b 13-5	8
*	multiplication	A*b 13*5	65
/	division	A/b 13/5	3
%	Modula(here reminder value)	A%b 13%5	3

2. Relational operators:-The Operator are used to compare one value with the another value. It gives either true (or) false (0) result. are also called as comparison operator.

operator	meaning
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or Equal to
<=	Less than or Equal to

3. **Logical Operators:-**These are used to combine two or more conditions and give the result either true or false. They are used to form compound conditions. Logical operators are also called as connectors.

operator	meaning
&&	Logical AND
||	Logical OR
!	Logical NOT

Logical AND:-It returns true when all conditions are true

Condition 1	Condition 2	Condition 1&& Condition 2

T	T	T
T	F	F
F	T	F
F	F	F

Logical OR:-It returns false when all conditions are false.

Condition 1	Condition 2	Condition || Condition 2
T	T	T
T	F	T
F	T	T
F	F	F

Logical NOT:-If the condition is false then it returns true.

Condition	!
T	F
F	T

4. **Assignment operator:-**Assign (store) a value to a variable.
   Variable=constant/variable/expression.
   Ex:-32; y=a+b+c;
5. **Increment and decrement operators (unary operator):-**
   Increment Operator(++):- 2 types
   Pre-Increment:-If the "++" operator "precedes" then operand than its is called "pre-increment".
   Ex:-c=++a;
   Post-increment:- If the "++" operator "succeeds" then operand than its is called "post-increment".
   Ex:-c=a++;
   Pre-decrement:-If the "--" operator "precedes" then operand than its is called "pre-decrement".
   Ex:-c=--a;
   Post-decrement:- If the "--" operator "succeeds" then operand than its is called "post-decrement".
   Ex:-c=a--;

6. **Conditional operator(Ternary operator):**-? And : are called conditional operator.

Operator	meaning
?	Question mark
:	colon

Syntax:-(Expression1)? (Expression2): (expression3)

7. **Bitwise operator:-**this operator can perform the operation on bits.

operator	meaning
&	Bitwise AND
!	Bitwise OR
^	Bitwise exclusive AND
<<	Bitwise left shift
>>	Bitwise right shift

PRECEDENCE AND ASSOCIATIVELY:-

The concept of **operator precedence and associativity in C** helps in determining which operators will be given priority when there are multiple operators in the expression. It is very common to have multiple operators in C language and the compiler first evaluates the operator with higher precedence. It helps to maintain the ambiguity of the expression and helps us in avoiding unnecessary use of parenthesis.

In this article, we will discuss **operator precedence, operator associativity, and precedence table** according to which the priority of the operators in expression is decided in C language.

Precedence and Associativity Table

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type) * & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	Left to right

UNIT-02

Introduction to decision control statements and Arrays

Selective, looping and nested statements, jumping statements.
Arrays: Introduction, declaration of arrays, accessing and storage of array elements, searching (linear and binary search algorithms) and sorting (selection and bubble) algorithms, multidimensional arrays, matrix operations

---

Introduction to decision control statements and Arrays:-

There are also called as Conditional/Branching/Selection/Decision making Statements.

Control Statements:-The Statements which are used to control the sequential execution of the program is called as control statements. Control statements are mainly classified into three types.

1. Conditional Control statements or Decision making or branching control statements.

2. Looping statements or Iterative control statements.

3. Jumping or unconditional statements.

The **conditional statements** (also known as decision control structures) such as if, if else, switch, etc. are used for decision-making purposes in C programs.

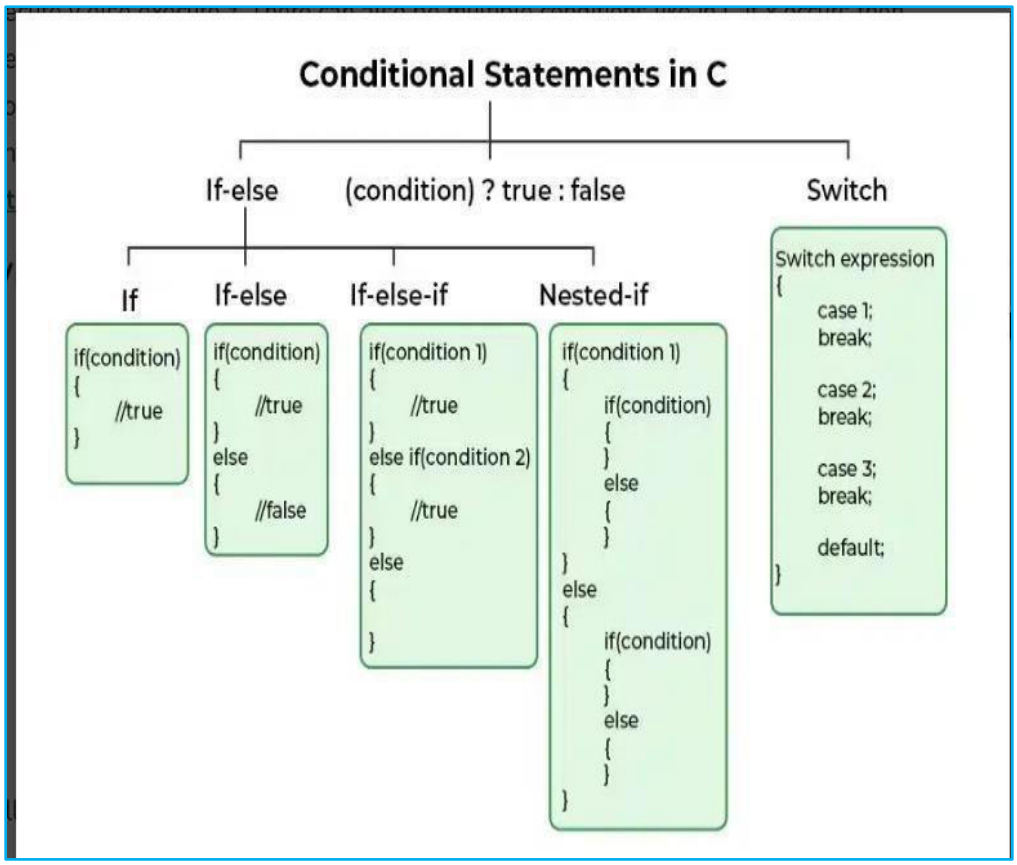
1. Conditional Control statements or Decision making:-

There come situations in real life when we need to make some decisions and based on these decisions

Or

These Statements execute a particular group of statements from several groups depending on the given conditions.

Types of Conditional Statements in C



Following are the decision-making statements available in C:

1. **if Statement [or] Simple if**
2. **if-else Statement**
3. **Nested if Statement**
4. **if-else-if Ladder**
5. **switch Statement**
6. **Conditional Operator**
7. **Jump Statements:**
   - **break**
   - **continue**
   - **goto**
   - **return**

1. **if Statement [or] Simple if:-**It may or may not execute block of statements

[or]

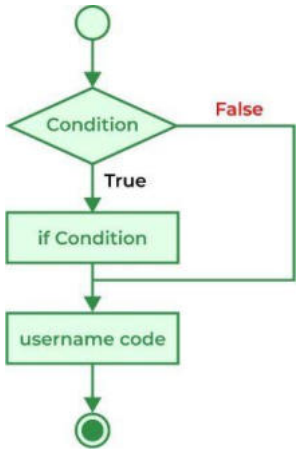
The if statement is the most simple decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statements is executed otherwise not.

Syntax:-

```
if(condition)
{
  // Statements to execute if
  // condition is true
}
```

Here, the **condition** after evaluation will be either true or false. C if statement accepts Boolean values – if the value is true then it will execute the block of statements below it otherwise not. If we do not provide the curly braces '{' and '}' after if(condition) then by default if statement will consider the first immediately below statement to be inside its block.

Flowchart of if Statement



Sample Program Simple if in C:-

```
// C program to illustrate If statement
#include <stdio.h>

int main()
{
   int i = 10;

   if (i > 15) {
      printf("10 is greater than 15");
   }

   printf("I am Not in if");
}
```

Out put:- I am Not in if

2. **if-else :-**In this conditional statement if the condition is true then it executes one group of statements otherwise it will execute another group of statements.

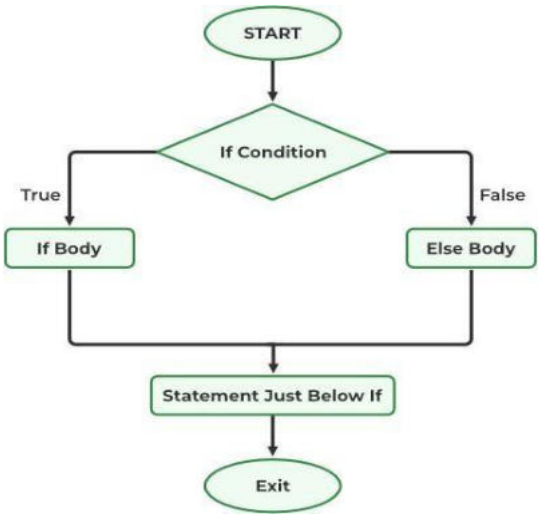
[or]

It executes one block of statements out of two blocks,

Syntax of if else

```
if (condition)
{
   // Executes this block if
   // condition is true
}
else
{
   // Executes this block if
   // condition is false
}
```

Flowchart of if-else Statement



Sample Program Simple if in C:-

```
// C program to illustrate If statement
#include <stdio.h>

int main()
{
   int i = 20;
```

```
    if (i < 15) {

        printf("i is smaller than 15");
    }
    else {

        printf("i is greater than 15");
    }
    return 0;
}
```

Out put:- i is greater than 15

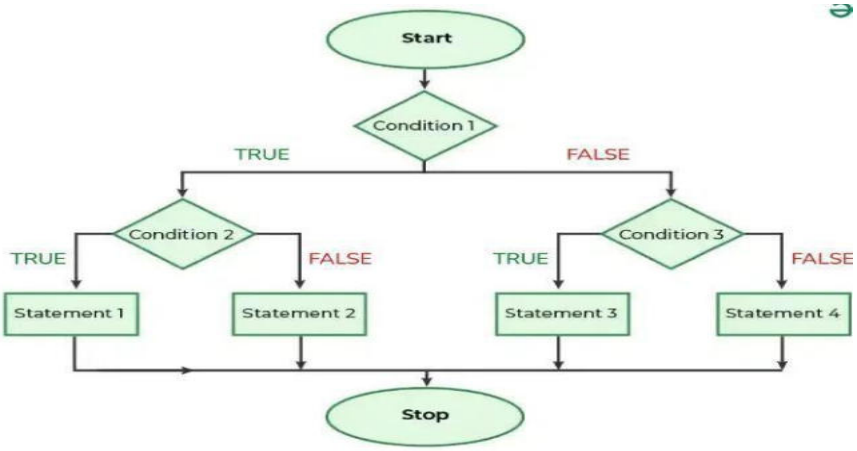
3. Nested if-else in C

A nested if in C is an if statement that is the target of another if statement. Nested if statements mean an if statement inside another if statement

Syntax of Nested if-else

```
if (condition1)
{
  // Executes when condition1 is true
  if (condition_2)
  {
    // statement 1
  }
  else
  {
      // Statement 2
   }
}
else {
    if (condition_3)
  {
    // statement 3
  }
  else
  {
      // Statement 4
   }
}
```

Flowchart of Nested if-else



Sample Program in Nested if:-

```
// C program to illustrate nested-if statement
#include <stdio.h>

int main()
{
    int i = 10;

    if (i == 10) {
        // First if statement
        if (i < 15)
            printf("i is smaller than 15\n");

        // Nested - if statement
        // Will only be executed if statement above
        // is true
        if (i < 12)
            printf("i is smaller than 12 too\n");
        else
            printf("i is greater than 15");
    }
    else {
```

```
        if (i == 20) {

            // Nested - if statement
            // Will only be executed if statement above
            // is true
            if (i < 22)
                printf("i is smaller than 22 too\n");
            else
                printf("i is greater than 25");
        }
    }

    return 0;
}
```

Output

i is smaller than 15

i is smaller than 12 too

4. else-if Ladder:-In these conditional statement it will execute the group of statements based on its respecting condition.

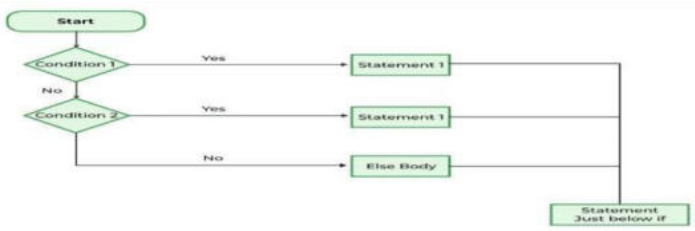
[or]

It executes one block of statements out of 'N' blocks,

Syntax of else-if Ladder

```
if (condition)
   statement;
else if (condition)
   statement;
.
.
else
   statement;
```

Flowchart of else-if Ladder



Sample Program else-if Ladder:-

```
// C program to illustrate nested-if statement
#include <stdio.h>

int main()
{
   int i = 20;

   if (i == 10)
      printf("i is 10");
   else if (i == 15)
      printf("i is 15");
   else if (i == 20)
      printf("i is 20");
   else
      printf("i is not present");
}
```

Output

i is 20

2. Conditional Expression in C:- The conditional Expression is used to add conditional code in our program. It is similar to the if-else statement. It is also known as the ternary operator as it works on three operands.

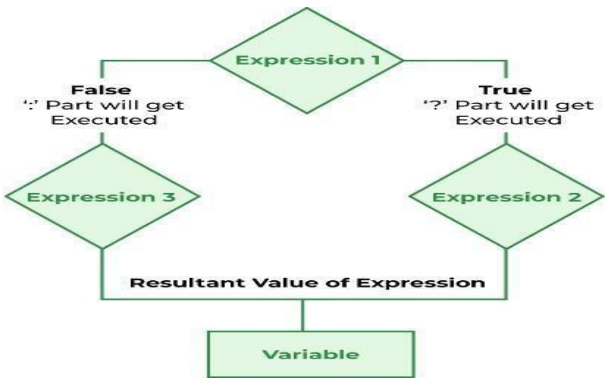
Syntax:- (condition) ? [true_statements] : [false statements];

[or]

Expression1? Expression 2 : Expression N;

Flowchart

:-



Program:-

```
// C Program to illustrate the use of conditional operator
#include <stdio.h>

// driver code
int main()
{

   int var;
   int flag = 0;

   // using conditional operator to assign the value to var
   // according to the value of flag
   var = flag == 0 ? 25 : -25;
   printf("Value of var when flag is 0: %d\n", var);

   // changing the value of flag
   flag = 1;
   // again assigning the value to var using same statement
   var = flag == 0 ? 25 : -25;
   printf("Value of var when flag is NOT 0: %d", var);

   return 0;
}
```

Output

```
Value of var when flag is 0: 25
Value of var when flag is NOT 0: -25
```

3. switch Statement in C:-

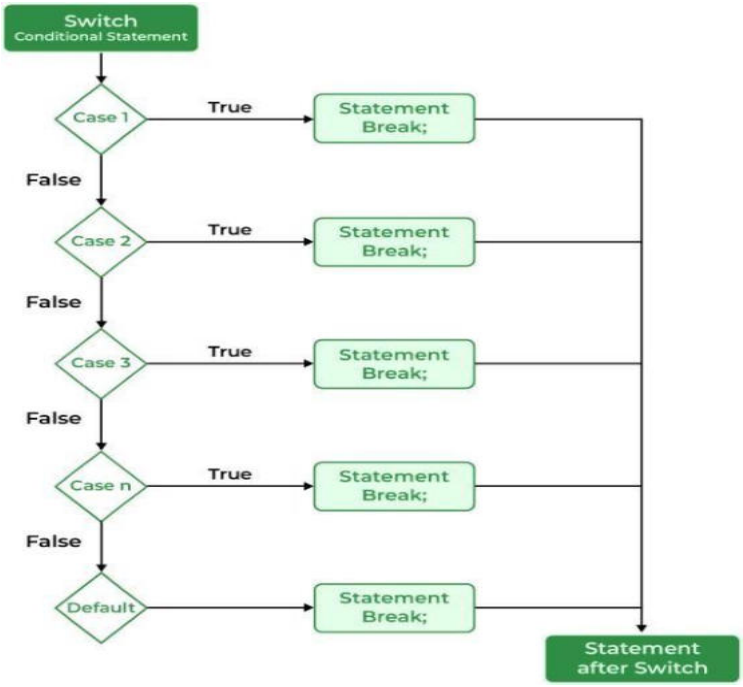
The switch case statement is an alternative to the if else if ladder that can be used to execute the conditional code based on the value of the variable specified in the switch statement. The switch block consists of cases to be executed based on the value of the switch variable.

Syntax of switch

```
switch (expression) {
   case value1:
      statements;
   case value2:
      statements;
   ....
   ....
```

```
    ....
    default:
        statements;
}
```

Flowchart



Program:-

```
// C Program to illustrate the use of switch statement
#include <stdio.h>

int main()
{
    // variable to be used in switch statement
    int var = 2;

    // declaring switch cases
    switch (var) {
    case 1:
```

```
        printf("Case 1 is executed");
        break;
    case 2:
        printf("Case 2 is executed");
        break;
    default:
        printf("Default Case is executed");
        break;
    }

    return 0;
}
```

Output

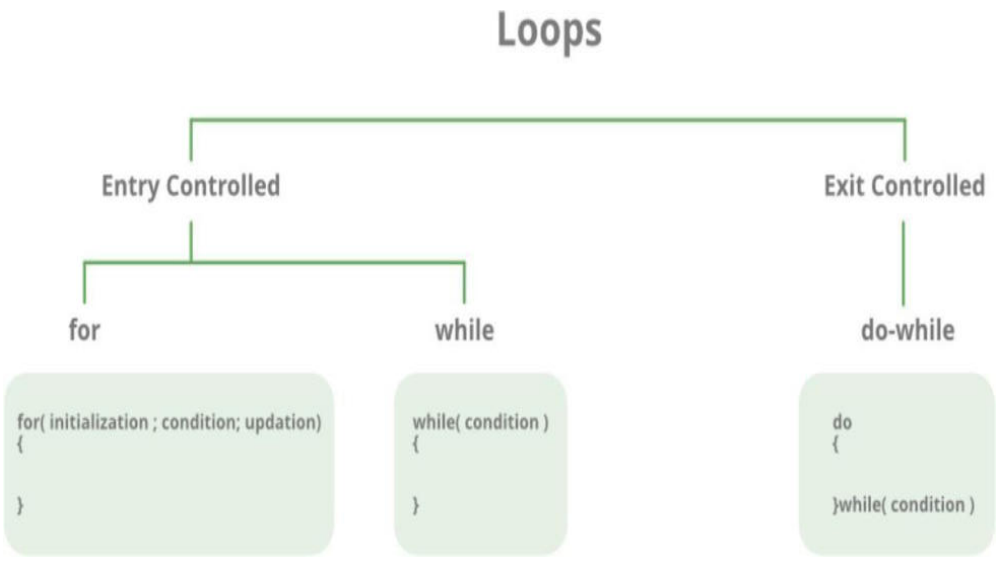
Case 2 is executed

Looping and nested statements [or] Iterative Statements:-

Generally in programming languages like C,C++,Java etc. Loops are used to executed a group of (or) A block of statements repeatedly until some conditions is satisfied.

(or)

Executing a statement or group of statements for a repeated number of times is called as a loop.The process of executing a block of statements repeatedly is known as looping.



Loops are mainly classified into 3 types.

1. While loop 2. Do while loop 3. For loop

1. While loop

While loop is also called entry controlled loop. IN this loop a statement or group of statements can be executed for a repeated number of times up to the specified conditions is false.

Syntax:-

```
While(condition)
{
Statement1;
Statement2;
.
Statement;
Changing value;
}
```

Example:-

```
#include <stdio.h>
 int main()
{
  int i = 2;
  while(i < 10)
  {
    printf( "Hello World\n");
    i++;
  }
  return 0;
}
```

2. Do while loop

It is also called as Exit Controlled loop. In this loop a statement or group of statements will be executed once automatically even through the condition is false for the first time. It is the main drawback for the do while loop. In this loop a statement or group of statements will be executed for a repeated numbers of times up to the specified condition is false.

Syntax:-

```
do
{
```

```
Statement1;
Statement2;
.
Statement;
Changing value;
} While(condition);
```

Example:-

```
#include <stdio.h>
int main()
{
 int i = 2;
 do
 {
  printf( "Hello World\n");
   i++;
 } while (i < 1);
 return 0;
}
```

For Loop:-

for loop in C programming is a repetition control structure that allows programmers to write a loop that will be executed a specific number of times. for loop enables programmers to perform n number of steps together in a single line.

Syntax:

```
for (initialize expression; test expression; update expression)
{
   //
   // body of for loop
   //
}
```

Example:

```
for(int i = 0; i < n; ++i)
{
   printf("Body of for loop which will execute till n");
}
```

Example:-

```
// C program to illustrate for loop
```

```
#include <stdio.h>

// Driver code
int main()
{
  int i = 0;

  for (i = 1; i <= 10; i++)
  {
    printf( "Hello World\n");
  }
  return 0;
}
```

Difference b/w While & Do while:-

while	Do while
It is a pretesting	It is post Testing
Entry Controlled loop	Exit Controlled loop

Syntax:-
While(text condition)
{
Body of the loop
}
	Syntax:-
	do
	{
	Body of the loop
	}
	While(text condition);

While loop should not terminate with in semicolon	While loop terminate with in semicolon.
In this loop the condition becomes false, at first time the loop will be executed Zero times.	In this loop the condition becomes false, at first time the loop will be executed One times.
Condition test First.	Condition test Last.

Jumping Statements (or) UN Conditional Statements:-

These statements are used in C for the unconditional flow of control throughout the functions in a program. They support Three types of jump statements:

- Break
- Continue
- goto

Break:-

This loop control statement is used to terminate the loop. As soon as the break statement is encountered from within a loop, the loop iterations stop there, and control returns from the loop immediately to the first statement after the loop.

Syntax:-

```
While(Condition 1)
{
Statements;
If(condition 1)
Break,
}
Statements;
}
Next statements;
```

Program:-

```
#include<stdio.h>
Int main()
{
Int I;
For(i=1;i<=10;i++)
{
If(i==5)
{
Break;
}
Printf("/n,%d",i);
}
```

Continue statement:-

The continue statement in C is used to skip the remaining code after the continue statement within a loop and jump to the next iteration of the loop. When the continue statement is encountered, the loop control immediately jumps to the next iteration, by skipping the lines of code written after it within the loop body.

Syntax:-

```
While(Boolean expression)
{
--------
if(test condition)
{
Continue;
}
```

```
}
```

Program:-

C Program to illustrate the continue statement

```
#include <stdio.h>

int main()
{
    int i;
    // loop
    for (i = 0; i < 5; i++) {
        if (i == 2) {
            // continue to be executed if i = 2
            printf("Skipping iteration %d\n", i);
            continue;
        }
        printf("Executing iteration %d\n", i);
    }
    return 0;
}
```

Output:-

Executing iteration 0
Executing iteration 1
Skipping iteration 2
Executing iteration 3
Executing iteration 4

Goto statement:-

The goto statement is used to jump to a specific point from anywhere in a function. It is used to transfer the program control to a labeled statement within the same function.

Syntax of goto Statement:-

```
Gotolabel;
.
.
label:
```

Program:-

```
// C program to check if a number is
// even or not using goto statement
#include <stdio.h>

// function to check even or not
void check Even Or Not (int num)
{
    if (num % 2 == 0)
```

```
        // jump to even
        goto even;
    else
        // jump to odd
        goto odd;

even:
    printf("%d is even", num);
    // return if even
    return;
odd:
    printf("%d is odd", num);
}

int main()
{
    int num = 26;
    checkEvenOrNot(num);
    return 0;
}
```

Output

26 is even

Arrays: Introduction, declaration of arrays, accessing and storage of array elements,

Arrays:-

Array is a derived data type; it is a group of elements.

(or)

An array is a collection of elements of same type that share a common name.

Introduction:-An array is a finite group.

- An array is a collection of variables of the same type that are referenced by a common name.
- In C, all arrays consist of contiguous memory locations.
- The lowest address corresponds to the first element and the highest address to the last element.
- Arrays may have from one to several dimensions. A specific element in an array is accessed by an index.

  Declaration of Arrays :-**Array Index Start with "Zero".**

  The general form of single-dimension array declaration is:

  Syntax: data type array name [size];

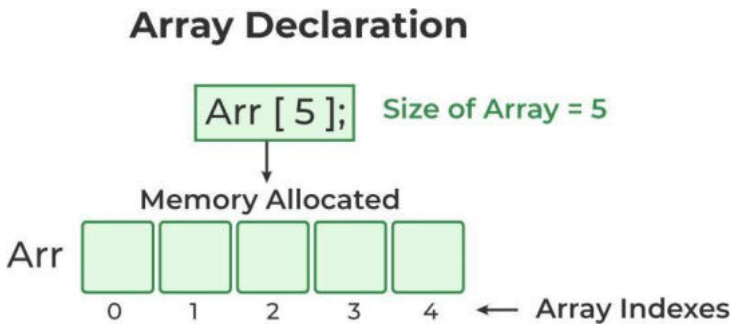
Here, type declares the base type of the array; size defines how many elements the array will hold. For example, the following declares as integer array named sample that is ten elements long:

Ex:-int a[5];

Ends with ⟶ size-1 5-1=4

Declares an array, named a, consisting of ten elements, each of type int. simply speaking, an array is a variable that can hold more than one value.

Ex: int x[100]; float mark[50]; char name[30];



The C arrays are static in nature, i.e., they are allocated memory at the compile time.

Example of Array Declaration C

```
// C Program to illustrate the array declaration
#include <stdio.h>
int main()
{
   // declaring array of integers
   int arr_int[5];
   // declaring array of characters
   char arr_char[5];

   return 0;
}
```

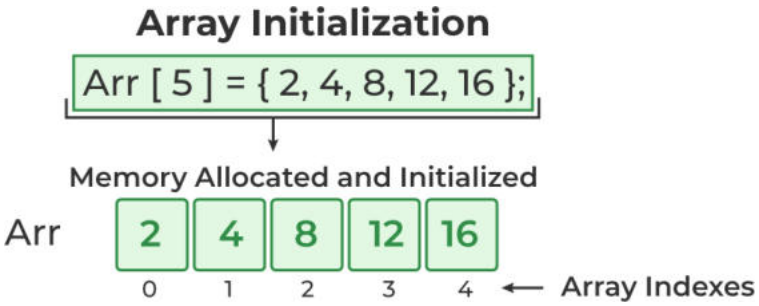
Array Initialization:- while declaring an array ,we can Initialize it with some values,.

If we initialize an array using an initializer list, we can skip declaring the size of the array as the compiler can automatically deduce the size of the array in these cases. The size of the array

in these cases is equal to the number of elements present in the initializer list as the compiler can automatically deduce the size of the array.

Syntax:-Data type array name[size]={v0,v1,……vn};
EX:-int a[6];

GV	GV	GV	GV	GV	GV
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]



Program:- Example of Array Initialization in C

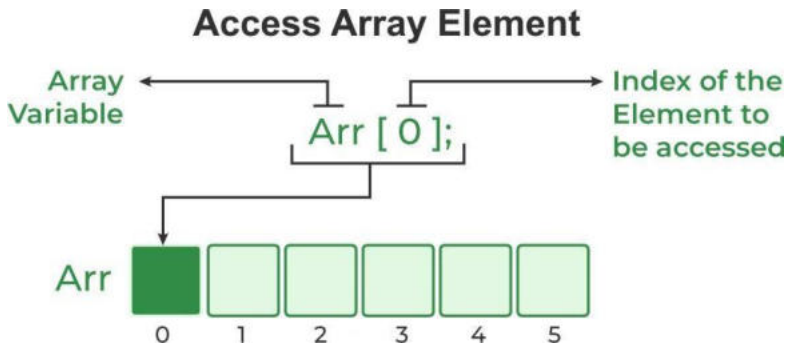
```
// C Program to demonstrate array initialization
#include <stdio.h>

int main()
{

   // array initialization using initializer list
   int arr[5] = { 10, 20, 30, 40, 50 };

   // array initialization using initializer list without
   // specifying size
   int arr1[] = { 1, 2, 3, 4, 5 };

   // array initialization using for loop
   float arr2[5];
   for (int i = 0; i < 5; i++) {
      arr2[i] = (float)i * 2.1;
   }
   return 0;
}
```

Accessing and storage of array elements:-

We can access any element of an array in C using the array subscript operator **[ ]** and the index value ***i*** of the element.

Syntax:-array name [index];

One thing to note is that the indexing in the array always starts with 0, i.e., the **first element** is at index **0** and the **last element** is at **N – 1** where **N** is the number of elements in the array.



Example of Accessing Array Elements using Array Subscript Operator:-

```
// C Program to illustrate element access using array
// subscript
#include <stdio.h>

int main()
{

   // array declaration and initialization
   int arr[5] = { 15, 25, 35, 45, 55 };

   // accessing element at index 2 i.e 3rd element
   printf("Element at arr[2]: %d\n", arr[2]);

   // accessing element at index 4 i.e last element
   printf("Element at arr[4]: %d\n", arr[4]);

   // accessing element at index 0 i.e first element
   printf("Element at arr[0]: %d", arr[0]);

   return 0;
```

}

Output

Element at arr[2]: 35

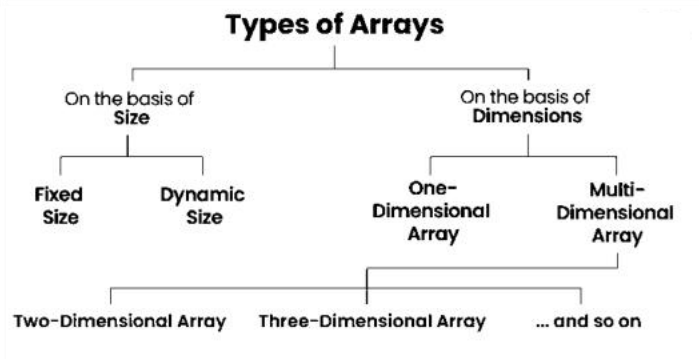
Element at arr[4]: 55

Element at arr[0]: 15

Types of Arrays :-

However, these array types can be broadly classified in two ways:

1. On the basis of Size
2. On the basis of Dimensions



1. One-dimensional array (1-D arrays):

When an array is declared with only one dimension (subscript) then it is called "one Dimensional array" or Single Dimensional Array**.** You can imagine a 1d array as a row, where elements are stored one after another.

Declaring 1D Array:-

The general format of declaring a one dimensional Array is a follows.

Syntax:-data type array name [size];

Ex: int a[6];

The above declaration reserves 6 contiguous memory locations of integer data type for the array 'a'. The conceptual view for the above declaration is as shown below.

0 1 2 3 4 5

a[0] a[1] a[2] a[3] a[4] a[5]

Initialization of 1D array:-

While declaring an array, we can initialize it with some values. The general format to initialize one dimensional array is as follows.

Syntax: data type array name [size]={v0,v1,v2…….vn};

Ex:- data type array name [6]={5,10,20,25,30,35};

where,

- **data type:** is a type of data of each array block.
- **array name:** is the name of the array using which we can refer to it.
- **array size:** is the number of blocks of memory array going to have.

Two-dimensional (2D) array:

Multidimensional arrays can be considered as an array of arrays or as a matrix consisting of rows and columns.

Ex:- data type array name [3] [3]={5,10,20,25,30,35};

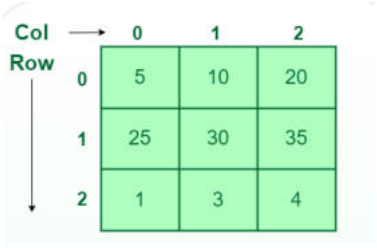
Syntax for Declaration of Two-Dimensional Array

Below is the syntax to declare the Two-dimensional array

data type array name[sizeof_1st_dimension][sizeof_2nd_dimension];

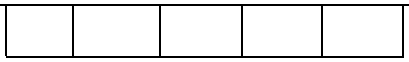
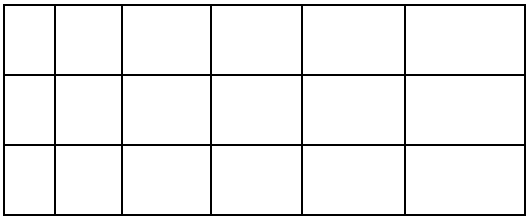
where,

- **data type:** is a type of data of each array block.
- **array name:** is the name of the array using which we can refer to it.
- **sizeof_dimension:** is the number of blocks of memory array going to have in the corresponding dimension.



Difference b/w 1-D and 2-D arrays:-

One dimensional array	Two dimensional array
A one dimensional array is a group of element of same type.	A two dimensional array is a group of one dimensional array of same type.
A one dimensional array must be declared Array name followed by only one.	A Two dimensional array must be declared by Array name followed by two.
Subscript	Subscripts

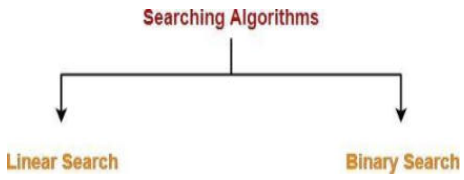
A one dimensional array is declared as follows Data type array name[size];	A two dimensional array is declared as follows Data type array name[row size] [col size];
The conceptual view for the above ex is follow 	
Ex:-int num[5]; Here variable 'num' can hold 5 values.	Ex:-int num[3][6]; Here variable 'num' can hold (3*6)18 values.

Searching (linear and binary search algorithms) and sorting (selection and bubble) algorithms

Searching: It is the one of the most important operation, Searching means finding the element whether the given element is present in the list or not is called searching.

- Finding the required element from a group of elements.
- The Searching element is used as a key element.
  There are Two Types of Searching

1. Linear Search/sequential Search ⟶ Sorted/Unsorted Array
2. Binary search ⟶ Sorted Array



1. Linear Search (Sequential Search)

In this search we search the element in required order from first element to last element. The linear search compares the key element by an array element one by one order until a math has been found.

[or]

Linear search is the simplest searching algorithm. It checks each element of the array or list in sequence until the target element is found or the list ends.

Algorithm:

- Start from the first element.
- Compare each element with the target.
- If the element matches the target, return its index.
- If the end of the list is reached without finding the element, return a special value (like -1) indicating that the element is not in the list.

Example program:-

```
#include <stdio.h>
// Function for linear search
int linearSearch(int arr[], int size, int target) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == target) {
            return i; // Return the index of the found element
        }
    }
    return -1; // Return -1 if the target is not found
}

int main() {
    int arr[] = { 10, 20, 30, 40, 50 }; // Sample array
    int target = 30; // Element to search for
    int size = sizeof(arr) / sizeof(arr[0]); // Calculate array size
    // Perform linear search
    int result = linearSearch(arr, size, target);
    // Output the result
    if (result != -1) {
        printf("Element %d found at index %d.\n", target, result);
    } else {
        printf("Element %d not found in the array.\n", target);
    }

    return 0;
}
```

Out put:- Element 50 found at index 4.

Binary Search algorithm: - Only on sorted Array

Binary Search is an interval searching algorithm that searches for an item in the sorted list. It works by repeatedly dividing the list into two equal parts and then searching for the item in the part where it can possibly exist. Searching order can be reduce "Divide & Conquer".

Binary Search Algorithm:-

Below is the step-by-step algorithm for Binary Search:

- Divide the search space into two halves by **finding the middle index "mid"**.
- Compare the middle element of the search space with the **key**.
- If the **key** is found at middle element, the process is terminated.
- If the **key** is not found at middle element, choose which half will be used as the next search space.
    - If the **key** is smaller than the middle element, then the **left** side is used for next search.
    - If the **key** is larger than the middle element, then the **right** side is used for next search.
- This process is continued until the **key** is found or the total search space is exhausted.

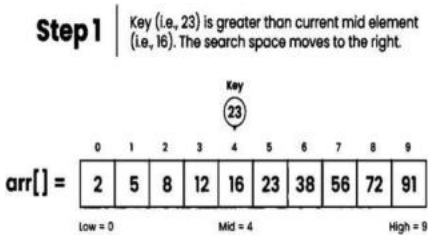
Binary Search Algorithm

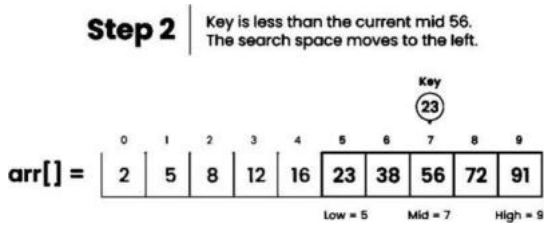
Below is the step-by-step algorithm for Binary Search:

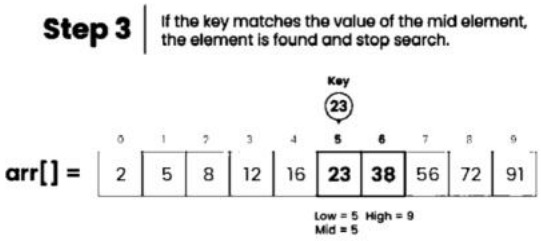
- Divide the search space into two halves by **finding the middle index "mid"**.
- Compare the middle element of the search space with the **key**.
- If the **key** is found at middle element, the process is terminated.
- If the **key** is not found at middle element, choose which half will be used as the next search space.
    - If the **key** is smaller than the middle element, then the **left** side is used for next search.
    - If the **key** is larger than the middle element, then the **right** side is used for next search.
- This process is continued until the **key** is found or the total search space is exhausted.

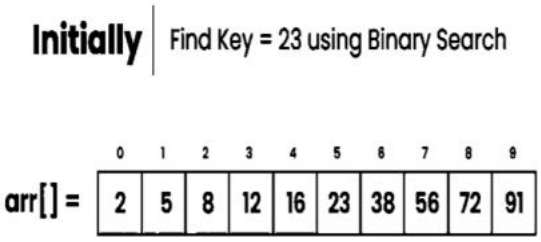
How does Binary Search Algorithm work?

To understand the working of binary search, consider the following illustration:
Consider an array **arr[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91}**, and the **target = 23**.









Binary search Program:-

```
#include <stdio.h>

// Function for binary search

int binarySearch(int arr[], int size, int target) {

    int left = 0, right = size - 1;

    while (left <= right) {

        int mid = left + (right - left) / 2; // Find the middle index

        // Check if the target is at the mid index

        if (arr[mid] == target) {

            return mid; // Target found, return the index
```

```
        }
        // If the target is smaller than the middle element, search in the left half
        else if (arr[mid] > target) {
            right = mid - 1;
        }
        // If the target is larger than the middle element, search in the right half
        else {
            left = mid + 1;
        }
    }
    return -1; // Target not found
}
int main() {
    int arr[] = { 10, 20, 30, 40, 50 }; // Sample sorted array
    int target = 30; // Element to search for
    int size = sizeof(arr) / sizeof(arr[0]); // Calculate array size
    // Perform binary search
    int result = binarySearch(arr, size, target);
    // Output the result
    if (result != -1) {
        printf("Element %d found at index %d.\n", target, result);
    } else {
        printf("Element %d not found in the array.\n", target);
```

```
    }

    return 0;

}
```

Output: - Element 30 found at index 2.

Sorting (selection and bubble) algorithms: -

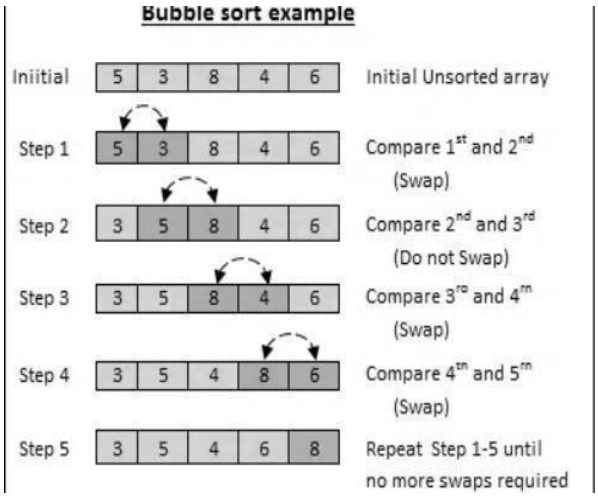
Arranging the elements of an array either in ascending or descending order.

There are 3 types of Arrays:-

1.Bubble Sort 2. Insertion Sort 3. Selection Sort

1. Bubble Sort:-

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity are quite high.



Program on Bubble Sort:-

```
// Optimized implementation of Bubble sort
#include <stdbool.h>
#include <stdio.h>

void swap(int* xp, int* yp){
    int temp = *xp;
```

```
    *xp = *yp;
    *yp = temp;
}

// An optimized version of Bubble Sort
void bubbleSort(int arr[], int n){
    int i, j;
    bool swapped;
    for (i = 0; i < n - 1; i++) {
        swapped = false;
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(&arr[j], &arr[j + 1]);
                swapped = true;
            }
        }

        // If no two elements were swapped by inner loop,
        // then break
        if (swapped == false)
            break;
    }
}

// Function to print an array
void printArray(int arr[], int size){
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", arr[i]);
}

int main(){
    int arr[] = { 64, 34, 25, 12, 22, 11, 90 };
    int n = sizeof(arr) / sizeof(arr[0]);
    bubbleSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

Out put:- Sorted array:

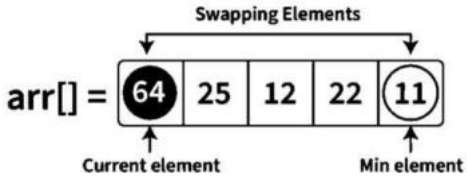
11 12 22 25 34 64 90

Selection Sort algorithm: -

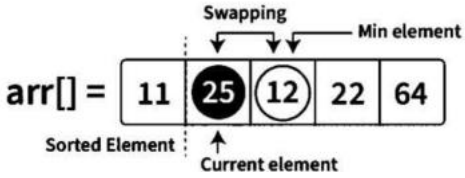
Selection Sort is a comparison-based sorting algorithm. It sorts an array by repeatedly selecting the **smallest (or largest)** element from the unsorted portion and swapping it with the first unsorted element. This process continues until the entire array is sorted.

1. First we find the smallest element and swap it with the first element. This way we get the smallest element at its correct position.
2. Then we find the smallest among remaining elements (or second smallest) and move it to its correct position by swapping.
3. We keep doing this until we get all elements moved to correct position.

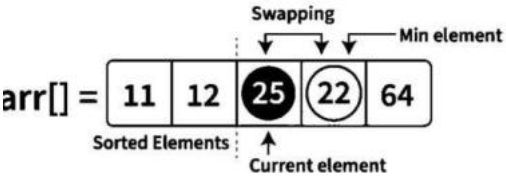
Start from the first element at index 0, find the smallest element in the rest of the array which is unsorted, and swap (11) with current element(64).



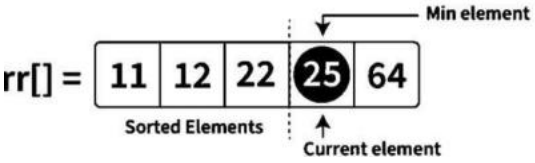
Move to the next element at index 1 (25). Find the smallest in unsorted subarray, and swap (12) with current element (25).



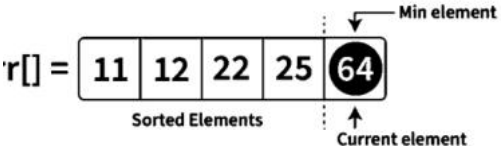
Move to element at index 2 (25). Find the minimum element from unsorted subarray, Swap (22) with current element (25).



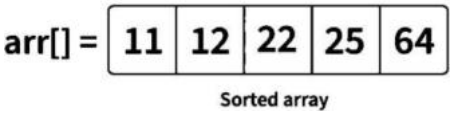
Move to element at index 3 (25), find the minimum from unsorted subarray and swap (25) with current element (25).



Move to element at index 4 (64), find the minimum from unsorted subarray and swap (64) with current element (64).



We get the sorted array at the end.



Program:-

```
// C program for implementation of selection sort
#include <stdio.h>
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {

        // Assume the current position holds
        // the minimum element
        int min_idx = i;

        // Iterate through the unsorted portion
         // to find the actual minimum
         for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {

                // Update min_idx if a smaller element is found
                min_idx = j;
            }
        }

        // Move minimum element to its
        // correct position
        int temp = arr[i];
        arr[i] = arr[min_idx];
        arr[min_idx] = temp;
    }
}

void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
```

```
    printArray(arr, n);

    selectionSort(arr, n);

    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}
```

Out put:-

Original array: 64 25 12 22 11
Sorted array: 11 12 22 25 64

Multi dimensional arrays:-

A multi-dimensional array can be defined as an array that has more than one dimension. Having more than one dimension means that it can grow in multiple directions. Some popular multidimensional arrays are 2D arrays and 3D arrays. In this article, we will learn about multidimensional arrays in C programming language.

Syntax:- *type arr_name[size1][size2]….[sizeN];*

Declaration of 3D Array in C

We can declare a 3D array with **x** 2D arrays each having **m** rows and **n** columns using the syntax shown below:

type arr_name[x][m][n];

Ex:- int a[2] [3] [4]

It holds 2*3*4=24 elements. [2 arrays 3 rows 4 columns].

A[0]=	a[0][0]	a[0][1]	a[0][2]	a[0][3]
	a[1][0]	a[1][1]	a[1][2]	a[1][3]
	a[2][0]	a[2][1]	a[2][2]	a[2][3]
A[1]=	a[0][0]	a[0][1]	a[0][2]	a[0][3]
	a[1][0]	a[1][1]	a[1][2]	a[1][3]
	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Initialization of 3D Array in C

Initialization in a 3D array is the same as that of 2D arrays. The difference is as the number of dimensions increases so the number of nested braces will also increase.

```
int arr[2][3][2] = {0, 1, 2, 3, 4, 5, 6, 7 , 8, 9, 10, 11}
```

or

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

Program:-

```
// C program to print elements of a Three-Dimensional Array
#include <stdio.h>

int main() {

   // Create and Initialize the 3-dimensional array
   int arr[2][3][2] = { { { 1, 1 }, { 2, 3 }, { 4, 5 } },
                        { { 6, 7 }, { 8, 9 }, { 10, 11 } } };

     // Loop through the depth
   for (int i = 0; i < 2; ++i) {

        // Loop through the rows of each depth
      for (int j = 0; j < 3; ++j) {

           // Loop through the columns of each row
         for (int k = 0; k < 2; ++k)
            printf("arr[%i][%i][%i] = %d  ", i, j, k,
                  arr[i][j][k]);
          printf("\n");
      }
    printf("\n\n");
   }
   return 0;
}
```

Output:-

```
arr[0][0][0] = 1   arr[0][0][1] = 1
arr[0][1][0] = 2   arr[0][1][1] = 3
arr[0][2][0] = 4   arr[0][2][1] = 5
```

arr[1][0][0] = 6 arr[1][0][1] = 7
arr[1][1][0] = 8 arr[1][1][1] = 9
arr[1][2][0] = 10 arr[1][2][1] = 11

Matrix Operations: -

Matrix is a collection of numbers organized in rows and columns, represented by a two-dimensional array in C.

[or]

Matrices are rectangular arrays of numbers, symbols, or characters where all of these elements are arranged in each row and column. An array is a collection of items arranged at different locations.

There are 3 types of matrix

- Matrices Addition
- Matrices Subtraction
- Matrices Multiplication

1. Matrices Addition:-

The addition of two matrices **A** $_{m*n}$ and $\mathbf{B}_{m*n}$ gives a matrix $\mathbf{C}_{m*n}$. The elements of C are the sum of corresponding elements in **A** and **B** which can be shown as:

$$\begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix} + \begin{bmatrix} 5 & 6 \\ 8 & 9 \end{bmatrix} = \begin{bmatrix} 6 & 8 \\ 12 & 14 \end{bmatrix}$$

Program:-

```
#include <stdio.h>
int main() {
   int row, col;

   // Input the number of rows and columns
   printf("Enter the number of rows and columns: ");
   scanf("%d %d", &row, &col);

   // Declare matrices
   int matrix1[row][col], matrix2[row][col], sum[row][col];
```

```
    // Input elements for the first matrix
    printf("Enter elements of matrix 1:\n");
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            printf("Enter element at [%d][%d]: ", i+1, j+1);
            scanf("%d", &matrix1[i][j]);
        }
    }

    // Input elements for the second matrix
    printf("Enter elements of matrix 2:\n");
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            printf("Enter element at [%d][%d]: ", i+1, j+1);
            scanf("%d", &matrix2[i][j]);
        }
    }

    // Adding corresponding elements of the two matrices
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            sum[i][j] = matrix1[i][j] + matrix2[i][j];
        }
    }

    // Printing the sum matrix
    printf("Sum of the matrices:\n");
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            printf("%d ", sum[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

Out put:-

```
Enter the number of rows and columns: 2 3
Enter elements of matrix 1:
Enter element at [1][1]: 1
```

```
Enter element at [1][2]: 2
Enter element at [1][3]: 3
Enter element at [2][1]: 4
Enter element at [2][2]: 5
Enter element at [2][3]: 6
Enter elements of matrix 2:
Enter element at [1][1]: 6
Enter element at [1][2]: 5
Enter element at [1][3]: 4
Enter element at [2][1]: 3
Enter element at [2][2]: 2
Enter element at [2][3]: 1
Sum of the matrices:
7 7 7
7 7 7
```

Matrices Subtraction:

The subtraction of two matrices $\mathbf{A}_{m*n}$ and $\mathbf{B}_{m*n}$ give a matrix $\mathbf{C}_{m*n}$. The elements of C are difference of corresponding elements in **A** and **B** which can be represented as:

$$\begin{bmatrix} 5 & 6 \\ 8 & 9 \end{bmatrix} - \begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix} = \begin{bmatrix} 4 & 4 \\ 4 & 4 \end{bmatrix}$$

Program:-

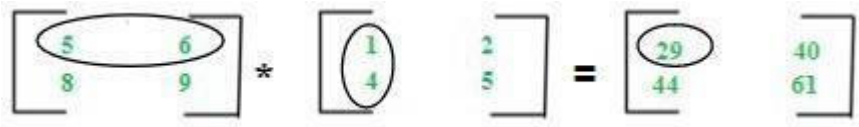
```
#include <stdio.h>

#define MAX_ROWS 10
#define MAX_COLS 10

void subtractMatrices(int A[MAX_ROWS][MAX_COLS], int B[MAX_ROWS][MAX_COLS],
int result[MAX_ROWS][MAX_COLS], int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            result[i][j] = A[i][j] - B[i][j];
        }
    }
}

void printMatrix(int matrix[MAX_ROWS][MAX_COLS], int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%d ", matrix[i][j]);
```

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

int main() {
    int rows, cols;

    // Input the size of the matrices
    printf("Enter the number of rows and columns: ");
    scanf("%d %d", &rows, &cols);

    intA[MAX_ROWS][MAX_COLS],B[MAX_ROWS][MAX_COLS],
result[MAX_ROWS][MAX_COLS];

    // Input first matrix
    printf("Enter elements of first matrix A:\n");
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            scanf("%d", &A[i][j]);
        }
    }

    // Input second matrix
    printf("Enter elements of second matrix B:\n");
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            scanf("%d", &B[i][j]);
        }
    }

    // Perform matrix subtraction
    subtractMatrices(A, B, result, rows, cols);

    // Display the result of subtraction
    printf("Result of A - B:\n");
    printMatrix(result, rows, cols);

    return 0;
}
```

Out put:-
Enter the number of rows and columns: 2 3
Enter elements of first matrix A:
1 2 3
4 5 6
Enter elements of second matrix B:
6 5 4
3 2 1
Result of A - B:
-5 -3 -1
1 3 5

Matrix Multiplication:-

The multiplication of two matrices A_{m*n} and B_{n*p} give a matrix C_{m*p}. It means a number of columns in A must be equal to the number of rows in B to calculate C=A*B. To calculate element c11, multiply elements of 1st row of A with 1st column of B and add them (5*1+6*4) which can be shown as:



Program:-

```
#include <stdio.h>

int main() {
    int A[10][10], B[10][10], result[10][10];
    int i, j, k;
    int A_rows, A_cols, B_rows, B_cols;

    // Input the size of the matrices
    printf("Enter number of rows and columns for matrix A: ");
    scanf("%d %d", &A_rows, &A_cols);

    printf("Enter number of rows and columns for matrix B: ");
    scanf("%d %d", &B_rows, &B_cols);

    // Matrix multiplication is only possible if A_cols == B_rows
    if (A_cols != B_rows) {
        printf("Matrix multiplication is not possible.\n");
        return 1;  // Exit if multiplication is not possible
    }
```

```
// Input elements for matrix A
printf("Enter elements of matrix A:\n");
for (i = 0; i < A_rows; i++) {
    for (j = 0; j < A_cols; j++) {
        scanf("%d", &A[i][j]);
    }
}

// Input elements for matrix B
printf("Enter elements of matrix B:\n");
for (i = 0; i < B_rows; i++) {
    for (j = 0; j < B_cols; j++) {
        scanf("%d", &B[i][j]);
    }
}

// Initialize result matrix to 0
for (i = 0; i < A_rows; i++) {
    for (j = 0; j < B_cols; j++) {
        result[i][j] = 0;
    }
}

// Matrix multiplication
for (i = 0; i < A_rows; i++) {
    for (j = 0; j < B_cols; j++) {
        for (k = 0; k < A_cols; k++) {
            result[i][j] += A[i][k] * B[k][j];
        }
    }
}

// Display the result of matrix multiplication
printf("Result of A * B:\n");
for (i = 0; i < A_rows; i++) {
    for (j = 0; j < B_cols; j++) {
        printf("%d ", result[i][j]);
    }
    printf("\n");
}
```

```
    return 0;
}
```

Out put:-

Enter number of rows and columns for matrix A:
2 3 Enter number of rows and columns for matrix B: 3 2 Enter elements of matrix A:
1 2 3
4 5 6
Enter elements of matrix B:
7 8
9 10
11 12
Result of A * B:
58 64
139 154

How Matrix Multiplication Works:

- The element result[i][j] is calculated by taking the dot product of the i-th row of matrix A and the j-th column of matrix B. For example:
  - result[0][0] = (1*7 + 2*9 + 3*11) = 58
  - result[0][1] = (1*8 + 2*10 + 3*12) = 64
  - Similarly for other elements.

UNIT-03

Strings and Functions

Strings: Declaration and Initialization, String Input / Output functions, String manipulation functions. Functions: Types of functions, recursion, scope of variables and storage classes. Preprocessor Directives: Types of preprocessor directives, examples.

Strings: Declaration and Initialization:-

String:-String is nothing but an array of characters that is Characters Array is called a string. Always string end switch (/0) null character.

Syntax:- char string name

[size]; Ex:-char name [5];

Any group of character enclosed with in double quotes is a string

constant. EX:- " HELLO JAN" etc..,

Operation on string:

The common operations are performed on strings are..,

1. Reading and writing strings.
2. Combining string together (Concatenation).
3. Copying one string to another.
4. Comparing string for equality.
5. Extracting a portion of string (sub String).
6. Finding the number of characters in a string(length)

Declaring a string Variable: To manipulate string an array must be declared with character data type. A string can be declared as follows.

Syntax:- Char string name[size];

The 'size' is the number of characters in the arrays.

Ex:- char name[40]; char city[15];etc..,

Initializing a string variable:- A string of characters can be stored in an array as follows:

Ex:-char name [ ]= "janu";

Two phases of strings:-

I by using char Array:- Character may be an

Alphabet/digit/symbols. Syntax:- Char string name[size];

(or)

Char string name[size]={'char',

'char',……}; Eg:-Char

name[5]={'A','I','T','S','\0'};

0	1	2	3	4
A	I	T	S	\0

Size is optional ⟶ char name [ ] = {'A','I','T','S','\0'};

II by using string Literal:-

We should use double Quotes, the null character is not needed, the C - Compiler automatically Inserts'\0' Character at the end.

Char string name[size]={''str value'', "str value'',……};

Eg:-Char name[5]={"A","I","T","S","\0"};

0	1	2	3	4
A	I	T	S	\0

Arrays of strings (or) two dimensional character type array:

To store more than one string in the array, then the array can be declared as follows.

Char array name[size1] [size2];

Here 'size1' specifies the number of strings and 'size2' refers the number of characters in each string.

Example:- char city [3][10] = {"DELHI","BANGLORE", "RAYACHOTY"};

This example reserves 3 memory locations to store 3 strings. Each string may contain 10 characters (10 bytes), the conceptual view for the above declaration is,

D	E	L	H	I	\0				

B	A	N	G	A	L	O	R	E	\0
R	A	Y	A	C	H	O	T	Y	\0

String Input/ Output Functions:-

String can be read and write using formative I/O functions by the format character "%s", while using "%s" character.

When %s character is used with scanf ( ) function its terminates its output at the first white space character. For example, if we input "NEW DELHI" then it takes only "NEW".

Ex:-

```
include<stdio.h>
int main ( )
{
Char name [30];
Printf("Enter a string");
Scanf ("%s", name);
Printf ("%s", name);
}
```

Gets () and puts () functions:-

These functions area also used to read and write strings the gets function read a string up to carriage return that is "n" pressed. The puts function writes the specified the string on the monitor.

These functions will not require any format character is read and write strings.

Ex:-

```
#include<stdio
.h>
#include<coni
o.h> int main
() {
char name[50];
printf("\nEnter your name: ");
gets(name, sizeof(name), stdin); // Safer than gets()
printf("\nYour name is: ");
puts(name); // Print the name
getch(); // Wait for a key press before closing
}
```

String Manipulation Function

(or)

Explain about built-in string function or string handling functions or character functions?

In 'C' Language there are several function are used to manipulate strings. These functions are included in "string.h" file. The following are some of the most commonly used string function.

There are 7 types:

- Strlen()
- Strrev()
- Strlwr()
- Strupr()
- Strcpy()
- Strcat()
- Strcmp()

1. **Strlen ():** This function finds the length of the given string. It means it counts and returns the number of characters present in the string. The output of this function is an integer.

Syntax:- Strlen(string)

Where 'string' is constant or array variable.

Example:-

```
#include<stdio
.h>
#include<coni
o.h> int main()
{
Int n:
Char
name[25]="Annamacharya";
n=strlen(name);
printf("length=%d",n);
}
```

Output:-12

2. **Strrev:** This function is used to reverse all the characters in the given string except null character.

   Syntax:- strrev(string); where 'string' is constant or array variable.

Example:-

```
#include<stdio.h>
#include<conio.h>
int main()
{
Char name[10]=”JANU”;
strrev(name);
puts (name);
}
```

Output:-UNAJ

3. **strlwr:-** This function converts all the uppercase letters in the given string into lower case.
   Syntax:- strlwr(string); where ‘string’ is constant or array variable.
   Example:-

```
#include<stdio.h>
#include<conio.h>
int main()
{
Char name[10]=”JANU”;
strlwr(name);
puts (name);
}
```

   Output:-janu
4. **strupr:-** This function converts all the lowercase letters in the given string into upper case.
   Syntax:- strupr(string); where ‘string’ is constant or array variable.
   Example:-

```
#include<stdio.h>
#include<conio.h>
int main()
{
Char name[10]=”janu”;
strupr(name);
puts (name);
}
```

   Output:-JANU
5. **strcpy:-** This function copies the contents of the one string to another string. The contents of the string are unchanged.
   Syntax:- strcpy(string); where ‘string’ is constant or array variable.

Example:-

```
#include<stdio.h>
#include<conio.h>
int main()
{
Char a[10]=”janu”;
Char b[10];
strcpy(a,b);
puts (b);
}
```

Output:-janu

6. **strcat:-** This function appends the contents of one string (Source) to anthor string (destination). This called Concatenation of two strings. The contents of the source string are unchanged.

   Syntax:- strcat (string1,string2);

   Where ‘string1’ is destination string and ‘string2’ is source. The contents of string2 are appended. To string1.

   Example:-

```
#include<stdio.h>
#include<conio.h>
int main()
{
Char a[10]=”janu”;
Char b[10]= “Rayachoty”;
strcpy(a,b);
puts (b);
}
```

   Output:-januRayachoty

7. **strcmp:-**The function is used to compare two strings.
   Syntax:-stracmp(string1,string2);
   1. If string1 is eqal to string2 then it returns value is 0.
   2. If string 1 is less than string2 then it return negative(less than o) value.
   3. If string1 is greater than string2 then it returns positive (greater than 0)value.

      Example:-

```
#include<stdio.h>
#include<string.h>
Int main()
```

```
{
Char a[30],b[30];
Chat c:
Printf("Enter first string");
Gets (a);
Printf("Enter second string");
Gets (b);
C= strccmp(a,b);
If(c==0)
Printf("string1 and string2 and Equal");
Else
If(c<0)
Printf("string1 is less than string2");
Else
If(c>0)
Printf("string1 is greater than string2");
}
```

Functions:-Types of Functions:-

What is a Function:-A function is a block of statements which can be used to perform a particular task.

(or)

Function definition:-A function is self-contained program segment that perform some specific and well-defined task. Every 'C' program consists of one or more functions. One of these functions must be called main ().Execution of the program will always begin with main().

Advantages of Functions:-

- Modularity
- Reusable
- More readable 7 easy to understand
- Debugging &Editing is easy
- Avoid rewriting same logic.

In C functions are divided into two categories.

1. Built-in or library function /Predefined/standard / Library Function.
2. User-defined Functions.
    1. Built-in or library function /Predefined/standard / Library Function:-These Functions are available to Programmer in the Standard 'C' Library.

Ex:-clrscr (),printf(),scanf().

Advantages of C library functions

- C Library functions are easy to use and optimized for better performance.
- C library functions save a lot of time i.e, function development time.
- C library functions are convenient as they always work.

2. User-Defined Function:-The programmer has to write a own function is called user defined functions. The use functions of user-defined functions allows a large program to be broken down in a number of smaller functions(modules).

Advantages of User-Defined Functions

- Changeable functions can be modified as per need.
- The Code of these functions is reusable in other programs.
- These functions are easy to understand, debug and maintain.

Some parts are there.

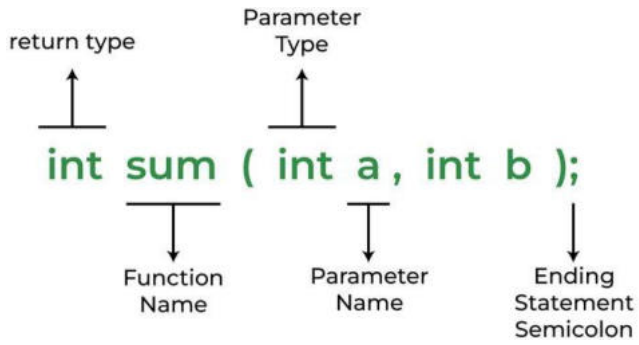
1. Function Declaration
2. Function Definition
3. Function Calling
4. Function Parameters
5. Return Statement

1.Function Declaration:- In a function declaration, we must provide the function name, its return type, and the number and type of its parameters. A function declaration tells the compiler that there is a function with the given name defined somewhere else in the program.

Syntax:- return type function name;

The parameter name is not mandatory while declaring functions. We can also declare the function without using the name of the data variables.

EX:- int sum(int a, int b); // Function declaration with parameter names int sum(int , int); // Function declaration without parameter names



2. *Function Definition:-*

The function definition consists of actual statements which are executed when the function is called (i.e. when the program control comes to the function).A C function is generally defined and declared in a single step because the function definition always starts with the function declaration so we do not need to declare it explicitly. The below example serves as both a function definition and a declaration.

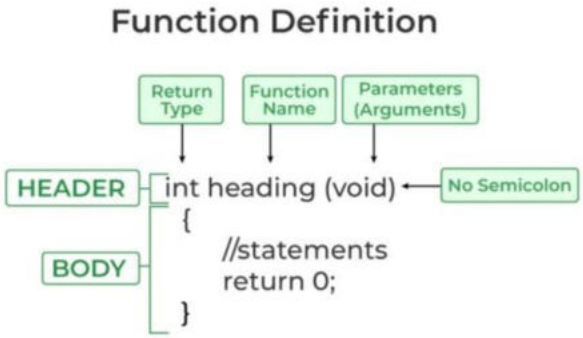
Syntax:-

return_type **function name** (para1_type *para1_name,* para2_type *para2_name*)

{

// body of the function

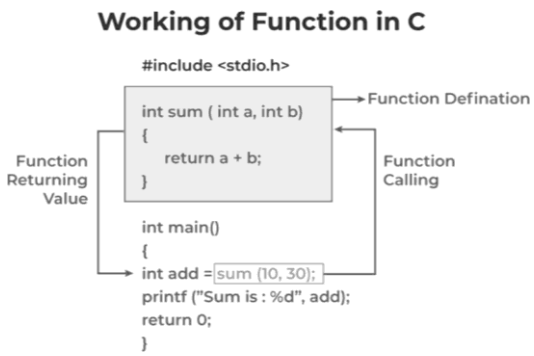
}



3. *Function Calling*

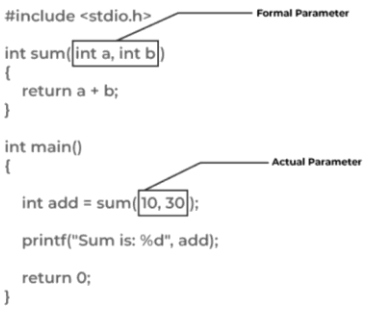
A function call is a statement that instructs the compiler to execute the function. We use the function name and parameters in the function call. In the below example, the first sum function is called and 10,30 are passed to the sum function. After the function call sum of a and b is returned and control is also returned back to the main function of the program.

Syntax:-function name (arg1,arg2……arg n);



4. *Function Parameters:-*

The data passed when the function is being invoked is known as the Actual parameters. In the below program, 10 and 30 are known as actual parameters. Formal Parameters are the variable and the data type as mentioned in the function declaration. In the below program, a and b are known as formal parameters.



sum of two numbers using user defined function in C…./

```
#include<stdio.h>
Int sum(int ,int);  \\fun
declaration Int main()
{
Int x,y,result;
Pf("\n Enter x,y value:");
Sf("%d %d",&x,&y);
Result=sum(x,y);  \\function calling (x,y are Actyual)
Pf("\n the addition=%d",result);
}
Int sum(int a,int b)//fun definition
{
Int c;        \\local
variable C=a+b;
```

```
    return c;
}
```

5. Function Return Type

Function return type tells what type of value is returned after all function is executed. When we don't want to return a value, we can use the void data type.

Example:

Syntax:-int fun c(parameter_1,parameter_2);

The above function will return an integer value after running statements inside the function.

Note: Only one value can be returned from a C function. To return multiple values, we have to use pointers or structures.

Recursion: -

A function that calls itself is known as a recursive function and this process is called as recursion.

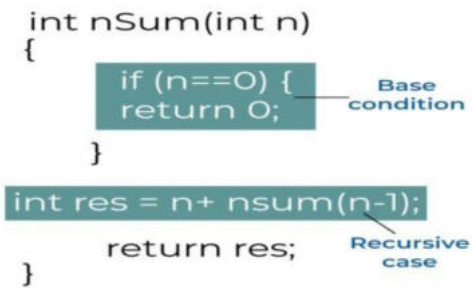
In C, recursion is used to solve complex problems by breaking them down into simpler sub-problems. We can solve large numbers of problems using recursion in C. For example, factorial of a number, generating Fibonacci series, generating subsets, etc.

Let's discuss some basic terminologies and fundamentals of recursion before going into working and implementation.

Fundamentals of C Recursion

The fundamental of recursion consists of two objects which are essential for any recursive function. These are:

1. Recursion Case
2. Base Condition



```
    EX:- robo ()
{
    Printf("hai");
    Robo ();  // fun calling
    Printf ("bye");
}
```

EX PROGRAM:-

```
#include<stdio.h>
    Int
    fact(int)
    ; Int
    main();
{
    Int n,result;
    Printf("\n enter n
    value:");
    Scanf("%d",&n);
    Result=fact (n);
    Printf("\n the factorial of %d is :%d",n,result);
}
    Int fact(int n);
    {
    If(
    n>
    1)
{
    Return  N*fact(n-1);
}
    Else
}
    Return 1;
}
```

Scops of variables and storage classes: - Scops of variables: -

It means the place/region/section of a program where a variable has been declared and has its existence and that variable can't be used or accessed beyond that region.

- Local Scope
- Global scope

Local scope:- The majority of variables you use are local variables. These are created within a function or a block of code. Local variables can't be accessed by other functions.

Ex:- #include <stdio.h>
int main(void)
{

```
  int room_key = 237;                    Local variable
  printf("%d ", room_key);
}
```

Global Scope:- These are created without a function or a block of code.

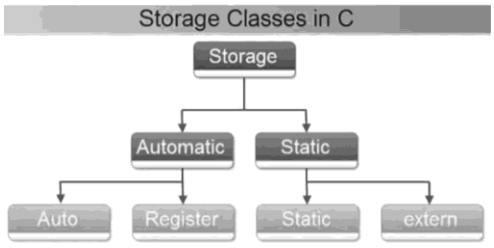
```
Ex:- #include<stdio.h>
A=10;
Int main();
{
Int a,b,c
A=100;
B=20;
C=a+b;
Printf ("\n The Result %d", c);
}
```

Out put:- 120

Storage Classes in C:-

A storage class in c is used to describe the following things

1. The Variable Scope
2. The Location where the variable will be stored
3. The Initialized value of a variable
4. life time of a variable
5. Who can access a variable

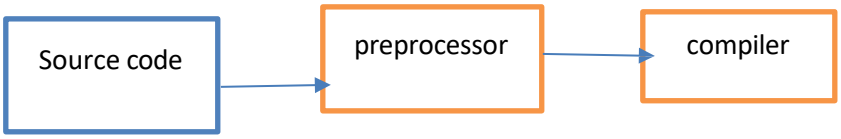


Storage Classes in C

Class	Specifier	Storage	Initial Value	Scope	Life
Automatic	auto	Stack	garbage	within block	End of Block
Register	register	CPU Register	garbage	within block	End of Block
Static	static	Data Segment	zero	within block	Till end of program
External	extern	Data Segment	zero	global multiple files	Till end of program

Preprocessor Directives: -Types of preprocessor Directives

The preprocessor is not a part of the compiler but is a separate step in the compilation process.



It is just a text Substitution Tool and it in struct the compiler to do required pre-processing before the actual compilation.

Preprocessor always start with the symbol is "#".

Ex program: -

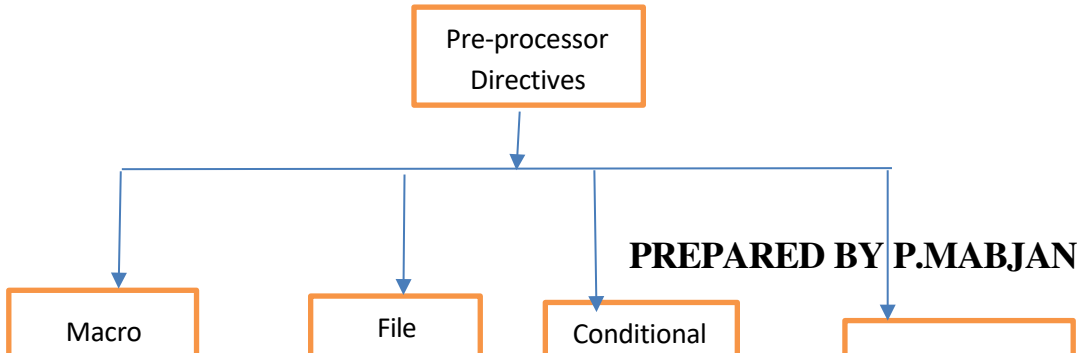
#include<stdio.h>

```
Void main()
{
Int a,b,c;
Scanf{"%d,%d",&a,&b);
C=a+b;
Printf("sum is %d\n",c);
}
```

Types of preprocessor Directives: -

Statements which are used in pre-processor program are called pre-processor

directives. There are mainly 4 pre-processor Directives



Miscella
neous

1. ***Macro Substitution Directives:-***

Macro is a small piece of code which is replaced by macro value, macro is defined by #define.

Macro are used to define constants.

There are 2 types of Macros: -

- Simple Marcos/object link macros.
- Macros with arguments/function like macros.

Simple Macros:-Simple macro is an identifier that is replaced by a constant value.it is mainly used to represent numeric constants .

Here are the few examples of simple macros.

#define PI+3.14,#define RATE#2,#define Size10.

Example Program: -

```
#include<stdio.h>

#define size10 10  // Define size as 10

int main() {
   int i;

   // Correct for loop to iterate from 1 to size10-1
   for(i = 1; i < size10; i++) {
      printf("\n%d Hello...", i);  // Corrected printf with proper syntax
   }

   printf("\n");  // To print a newline after the loop ends
   return 0;  // Return statement to end the program
}
```

Output:-

```
1 Hello...
```

2 Hello...
3 Hello...
4 Hello...
5 Hello...
6 Hello...
7 Hello...
8 Hello...
9 Hello...

Macros with arguments/function like macros.

Ex:-1. Program:-

```
#include<stdio.h>
#define MAX(a,b)a>b?a:b
int main();
{
Printf("maximum of 10 and 20 is :%d",MAX(10,20));
Printf("/n");
Return 0:
}
```

Ex:-

2. Program:-

```
\* program name :Macros args.c task:program to work with macros*/
#include<stdio.h>
#define MAX(a,b)a>b?a:b
int main();
{
Int a=10,b=20;
Printf("maximum of two numbers 10 and 20 is :%d",max(a,b));
Printf('\n");
return 0:
}
```

Output:- maximum of two numbers 10 &20 is : 20

2. File inclusive Directives:-

The #include pre-processor Directives include file contents in the current file . we can use system defined and user-defined header files.

There are two Syntaxes:-

1. #include <header file> ⟶ #include<stdio.h> tells the cpp to get system libraries.
2. #include "file.name" ⟶ #include "filename.h" tells the cpp to include filename.h.

Ex:- #include<stdio.h>

#include<conio.h>

#include "myfunc.h"

#include "sample.c"

3. Conditional Compilation

Condition completion allows the compiler to produce different executable based upon the parameters provided during compilation.
It is done by compiling specific portion of the source code or by skipping some specific portion of the program based on some conditions.

Directive	Descriptions
#if	Uses the value of macro
#if def	Return true if this macro is defined
#ifndef	Return true if this macro is defined
#elif	Its equal to else and if
#else	Executes if # if is false
#end if	End of pre-processor conditional
#error	Prints error message and halts the complation.

UNIT-04

POINTERS

Pointers: Understanding computer's memory, introduction to pointers, declaration pointer variables, pointer arithmetic, pointers and strings, array of pointers, function pointers, dynamic memory allocation, advantages and drawbacks of pointers

---

Pointers: Understanding computer's memory

A pointer is defined as a derived data type that can store the **memory address** of other variables, functions, or even other pointers. It is one of the core components of the C programming language allowing low-level memory access, dynamic memory allocation, and many other functionalities in C.

Syntax:-datatype *ptr;

What is mean by Variable:-Variable is used to store a data value.

Syntax:-datatype

Identifier; Introduction

to pointers:-

C pointer is the derived data type that is used to store the address of another variable and can also be used to access and manipulate the variable's data stored at that location. The pointers are considered as derived data types.

With pointers, you can access and modify the data located in the memory, pass the data efficiently between the functions, and create dynamic data structures like linked lists, trees, and graphs.

Advantages of Pointers

Following are the major advantages of pointers in C:

- Pointers are used for dynamic memory allocation and deallocation.
- Execution of program will be faster.
- An Array or a structure can be accessed efficiently with pointers
- Pointers are useful for accessing memory locations.
- Pointers are used to form complex data structures such as linked lists, graphs, trees, etc.
- Pointers reduce the length of the program and its execution time as well.

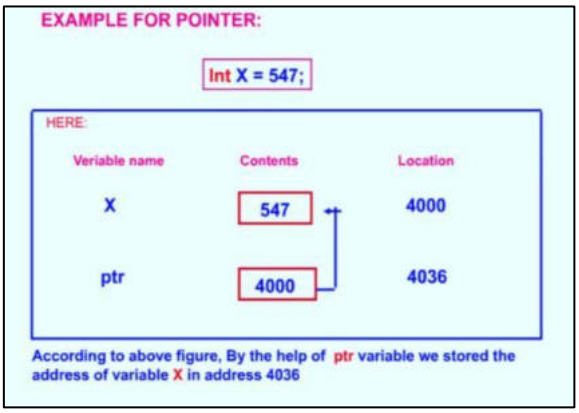
Disadvantages of Pointers

Pointers are vulnerable to errors and have following disadvantages:

- Memory corruption can occur if an incorrect value is provided to pointers.

- Pointers are a little bit complex to understand.

- Pointers are majorly responsible for memory leaks in C.
- Pointers are comparatively slower than variables in C.
- Uninitialized pointers might cause a segmentation fault.
- Using pointers sometimes be confusions.



Declaration pointer variables:-

1. Pointer Declaration

To declare a pointer, we use the **(*) dereference operator** before its name. In pointer declaration, we only declare the pointer but do not initialize it.

2. Pointer Initialization

Pointer initialization is the process where we assign some initial value to the pointer variable. We use the **(&) addressof operator** to get the memory address of a variable and then store it in the pointer variable.

Note: *We can also declare and initialize the pointer in a single step. This is called* ***pointer definition.***

Syntax:-Datatype *variable;

Ex:-int *p; float *m;

Pointer Operations:-C language provides two special operators to manipulate the data items directly from memory location using pointer variables.

1. **Address Operators(&):-** It gives address of given variable.(Indicates the address).

Ex:- where 'pv' is a pointer variable which holds the address of the variable 'v' so 'pv' is a pointer to 'v' that pointer to the location of 'v' as shown below.

pointer

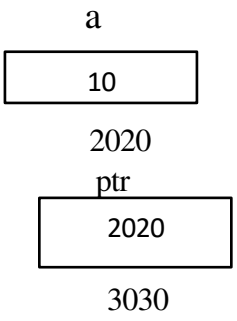
The above diagram shown the relationship between 'pv' and 'v' hence pv=&v and v=*pv.

2. Indirection operator (*):- It displays the value of pointer. To declare a pointer and for accessing the value of the pointer to which it is pointing.

Ex:- v=*pv;

Example program:-

```
/* pointer using in c….*/
#include<stdio.h>
#include<conio.h>
Void main()
{
Int a=10;
Int
*ptr
;
Ptr=
&a;
Printf("\n the value of
a=%d",a); Pf("\n the address of
a =%u,&a);
Pf("\n the value of
p;ointer,ptr=%u",ptr); Pf("\n the address
of pointer,ptr=%u',&ptr);
Pf("\n the value of the pointer pointing=%d",*ptr);
Pf("\n the value of a =%d",*(&a));
}
```



Output:-

Value of a=10

Address of a

=2020 Value of

p=2020 Address

of p =3030 Value

of a=*p=10

Types of pointers:- there mainly classified into 4 types:

1. Typed pointers:-A pointer is a variable which can be used to store the memory address of another variable.

Syntax:-datatype *identifier;

Ex:-int *p; float *f; char *c; double *d;.

2. Untyped pointers (or) Generic pointer:- The pointers declared with “void” type are called as untyped pointer.

Syntax:- void *Identifier;

Ex:-void *ptr;

Here "ptr" can store the address of int/float/double etc..,

3. Null pointer:-A pointer which has been assigned with null value is called null pointer.

Syntax:-datatype *identifier=null;

Ex:-int *ptr=null; void *p=null;

4. double pointer:-pointer to pointer

Double pointer is used to store the memory address of another pointer variable. A double pointer is also a variable, which can be used to store the memory address of another pointer variable.

Syntax:- datatype ** identifier;

Ex:-int **dp;

```
/*double pointer using in c….*/
#include<stdio.h>
int main()
{
Int
a=10
; Int
*p;
Int
**dp
;
P=&
a;
dp=
&p;
Printf("\n the value of
a=%d",a); Pf("\n the address of
a =%u,&a); Pf("\n the value of
p=%u",p); Pf("\n the address of
p=%u",&p);
Pf("\n the value of pointed by
p=%d",*p); Pf("\n the value of
dp=%u",dp);
Pf("\n the address dp=%u',&dp);
Pf("\n the value pointed by dp=%d",**dp);
}
```

pointer arithmetic:-

Pointer Arithmetic is the set of valid arithmetic operations that can be performed on pointers. The pointer variables store the memory address of another variable. It doesn't store any value.

Hence, there are only a few operations that are allowed to perform on Pointers in C language. The C pointer arithmetic operations are slightly different from the ones that we generally use for mathematical calculations. These operations are:

1. Increment Operation(++)
2. Decrement Operation(--)
3. Addition Operation Pointer(+)
4. Subtraction Operation Pointer(-)

1. Increment Operation(++):-It is a condition that also comes under addition. When a pointer is incremented, it actually increments by the number equal to the size of the data type for which it is a pointer.

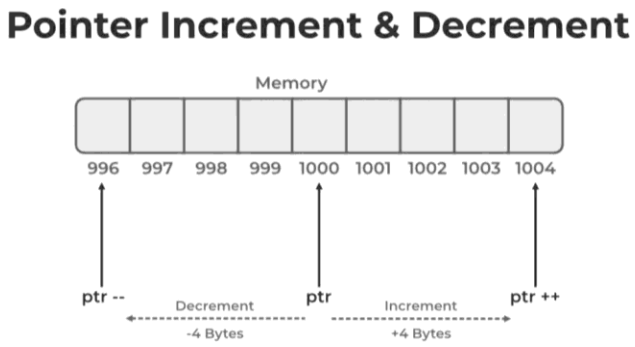
ForExample:

If an integer pointer that stores **address 1000** is incremented, then it will increment by 4(**size of an int**), and the new address will point to **1004**. While if a float type pointer is incremented then it will increment by 4(**size of a float**) and the new address will be **1004**.

2. Decrement Operation(--):-It is a condition that also comes under subtraction. When a pointer is decremented, it actually decrements by the number equal to the size of the data type for which it is a pointer.

For Example:

If an integer pointer that stores **address 1000** is decremented, then it will decrement by 4(**size of an int**), and the new address will point to **996**. While if a float type pointer is decremented then it will decrement by 4(**size of a float**) and the new address will be **996**.



Example Program:-

```
#include <stdio.h>
// pointer increment and decrement
//pointers are incremented and decremented by the size of the data type they point
to int main()
{
  int a = 22;
  int *p = &a;
  printf("p = %u\n", p); // p = 6422288
  p++;
  printf("p++ = %u\n", p); //p++ = 6422292     +4    // 4 bytes
```

```
    p--;
    printf("p-- = %u\n", p); //p-- = 6422288     -4   // restored to original value
   return 0;
}
```

3. **Addition Pointer:-** When a pointer is added with an integer value, the value is first multiplied by the size of the data type and then added to the pointer.

Ex:-

```
//*pointer Arithmetic -Addition…*//

#include<stdio.h>

Int main()

{

Int a[5]={10,20,30,40,50};

Int
*p
;

P=
a;

Pf{"\n value of p=%d",*p);

Pf("\n the address of the value=%u",p);

P=p+2;

Pf{"\n value of p=%d",*p);

Pf("\n the address of the value=%u",p);

return o;

}
```

Output:-

Value of p=10

The address of the value=2645617840

Value of p=30

The address of the value=2645617848.

4. **Subtraction of Integer to Pointer:-** When a pointer is subtracted with an integer value, the value is first multiplied by the size of the data type and then subtracted from the pointer similar to addition.

```
Ex:-//pointer Arthematic
Subtraction….*// #include<stdio.h>
Int main()
{
Int a[5]={10,20,30,40,50};
Int
*p1,*p2
;
P1=&a[
0];
P2=&a[2];
Printf('\n value pointed by
p1=%d",*p1); Printf('\n value pointed
by p2=%d",*p2); Printf('\n value of
pointer  p1=%u",p1); Printf('\n value of
pointer  p2=%u",p2); Printf('\n p2-
p1=%d",p2-p1);
Printf('\n p1-p2=%d",p1-
p2); P2=p2-1;
Printf("\n value of p2=%d",*p2);
Printf("\n Address of value =%u",p2);
return 0;
}
```

Output:-

Value of p1=10

Value of p2=30

Value of pointer p1=3143928784 Value of pointer p2=3143928782

Pointers and strings: -

In C, **strings** are represented as arrays of characters, and **pointers** play a key role in

manipulating and managing strings. Understanding the relationship between pointers and strings is fundamental to working with text in C.

Pointer:-Pointer is a variable ,which stores the memory address of another variable.to declare a pointer, you use the *symbol.

Ex:-int a=10;

int *p=&a;

In above example :-

A is a regular integer

variable. P is a pointer to

an integer.

& a memory address of a.

String:-This character Arrays is also called as a string , always string ends with null- character('\0').,

Arrays of characters:- char str[]="HELLO WORLD!"; Pointer of character:- char 8str= "HELLO WORLD!"; **Ex program:-** Pointer and String Together:-

```
#include
<stdio.h> int
main()
{
    char *str =
      "Hello";
printf("%s\n", str); // Output: Hello
str++; // Move the pointer to the next
character printf("%s\n", str); // Output: ello
return 0; } return 0;
}
```

Output:-

Hello

ello

Arrays of Pointers: -

There is a close relationship between pointers and arrays. a pointer can also be declared as an array. As a pointer variable always contains an address, an array of pointers contains a collection of address. The general from of an array of pointers is as follows.

Syntax:- datatype *array name[size];

(or)

pointer_type *array_name [array_size];

Here,

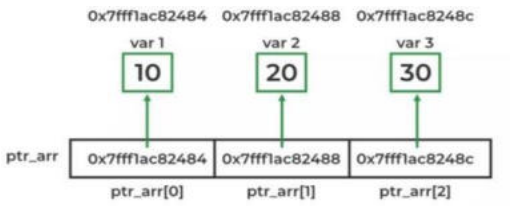
- **pointer_type:** Type of data the pointer is pointing to.
- **array_name:** Name of the array of pointers.
- **array_size:** Size of the array of pointers.

Ex:-
```
// C program to demonstrate the use of array of pointers
#include <stdio.h>
int main()
{
  // declaring some temp variables
  int var1 = 10;
  int var2 = 20;
  int var3 = 30;

  // array of pointers to integers
  int* ptr_arr[3] = { &var1, &var2, &var3 };

  // traversing using loop
  for (int i = 0; i < 3; i++) {
    printf("Value of var%d: %d\tAddress: %p\n", i + 1, *ptr_arr[i], ptr_arr[i]);
  }

  return 0;
}
```

Output:-

Value of var1: 10 Address: 0x7fff1ac82484 Value of var2: 20 Address: 0x7fff1ac82488 Value of var3: 30 Address: 0x7fff1ac8248c

Explanation:



Function Pointers:-

A **function pointer** in C is a pointer that points to the address of a function. This allows you to call functions dynamically at runtime. Function pointers are powerful and flexible, enabling dynamic behavior such as callback functions, event handling, or table-driven approach.

Declaration of Function Pointer:

The syntax for declaring a function pointer is as follows:

Syntax:- return_type (*pointer_name)(parameter_type1, parameter_type2, ...);

- return_type is the type of the value the function returns.
- pointer_name is the name of the pointer.
- parameter_type1, parameter_type2, etc., are the types of parameters the function takes

Ex:- int (*func_ptr)(int, int);

Here are two simple programs demonstrating **function pointers in C**: one where the function pointer points to a function without arguments, and another where the function pointer points to a function with arguments.

1. Function Pointer Without Arguments

In this example, the function does not take any arguments, and we use a function pointer to call it.

Code:

```
#include <stdio.h>

// Function without
arguments void greet() {
printf("Hello, world!\n");
}

int main() {
// Declare a function pointer
void (*func_ptr)();

// Point to the 'greet' function
func_ptr = greet;

// Call the function using the pointer
func_ptr();  // Output: Hello, world!

return 0;
}
```

Output:- Hello, world!

2. Function Pointer With Arguments

In this example, the function takes arguments, and the function pointer is used to call the function with those arguments.

Ex:-

```
#include <stdio.h>

// Function with
arguments int add(int a,
int b) {
return a + b;
}

int main() {
// Declare a function pointer that takes two integers and returns an integer
int (*func_ptr)(int, int);

// Point to the 'add' function
func_ptr = add;

// Call the function using the pointer with arguments
int result = func_ptr(5, 3);  // Passing arguments 5 and 3
printf("Addition result: %d\n", result);  // Output: 8

return 0;
}
```

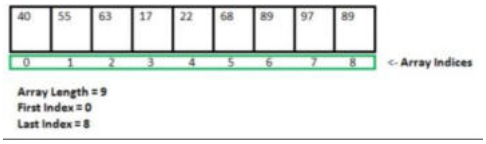
Output:- Addition result: 8

Explanation:

- add(int a, int b) is a function that takes two integers as arguments and returns their sum.
- func_ptr is a pointer to a function that takes two int arguments and returns an int.
- func_ptr = add; assigns the address of the add function to func_ptr.
- func_ptr(5, 3); calls the function with arguments 5 and 3, and stores the result in result.

Dynamic Memory Allocation (DMA): -

Dynamic Memory location means allocation of memory at run time.., During program execution. Since C is a structured language, it has some fixed rules for programming. One of them includes changing the size of an array. An array is a collection of items stored at contiguous memory locations.



As can be seen, the length (size) of the array above is 9. But what if there is a requirement to change this length (size)? For example,

- If there is a situation where only 5 elements are needed to be entered in this array. In this case, the remaining 4 indices are just wasting memory in this array. So there is a requirement to lessen the length (size) of the array from 9 to 5.
- Take another situation. In this, there is an array of 9 elements with all 9 indices filled. But there is a need to enter 3 more elements in this array. In this case, 3 indices more are required. So the length (size) of the array needs to be changed from 9 to 12.

Therefore, C **Dynamic Memory Allocation** can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime. C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under **<stdlib.h>** header file to facilitate dynamic memory allocation in C programming. They are:

1. malloc()
2. calloc()
3. free()
4. realloc()

malloc() method

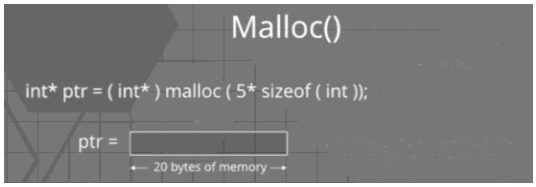
The **"malloc"** or **"memory allocation"** method in C is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It doesn't Initialize memory at execution time so that it has initialized each block with the default garbage value initially.

Syntax of malloc() in C

ptr = (cast-type*) malloc(byte-size)

For Example: ***ptr = (int*) malloc(100 * sizeof(int));***

Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.



Program:-

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

int main()
{

    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;

    // Get the number of elements for the array
    printf("Enter number of elements:");
    scanf("%d",&n);
    printf("Entered number of elements: %d\n", n);

    // Dynamically allocate memory using malloc()
    ptr = (int*)malloc(n * sizeof(int));

    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {
```

```
        // Memory has been successfully allocated
        printf("Memory successfully allocated using malloc.\n");

        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }

        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }
    }

    return 0;
}
```

Output:-

Enter number of elements:2

Entered number of elements: 2

Memory successfully allocated using malloc.

The elements of the array are: 1, 2,

C calloc() method

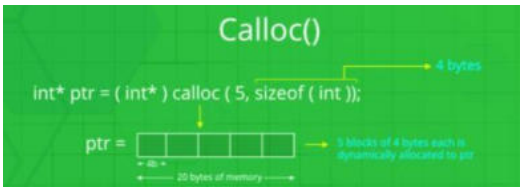
1. **“calloc”** or **“contiguous allocation”** method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. it is very much similar to malloc() but has two different points and these are:
2. It initializes each block with a default value ‘0’.
3. It has two parameters or arguments as compare to malloc().

Syntax of calloc() in C

ptr=(cast-type*)calloc(n,element-size);
here, n is the no. of elements and element-size is the size of each element.

For Example:

ptr=(float*)calloc(25,sizeof(float));
This statement allocates contiguous space in memory for 25 elements each with the size of the float.



Program:-

```
#include
<stdio.h>
#include
<stdlib.h> int
main()
{
   // This pointer will hold the
   // base address of the block created
   int* ptr;
   int n, i;

   // Get the number of elements for the array
   n = 5;
   printf("Enter number of elements: %d\n", n);

   // Dynamically allocate memory using calloc()
   ptr = (int*)calloc(n, sizeof(int));

   // Check if the memory has been successfully
   // allocated by calloc or not
   if (ptr == NULL) {
      printf("Memory not allocated.\n");
      exit(0);
   }
   else {
```

// Memory has been successfully allocated

```
        printf("Memory successfully allocated using calloc.\n");

        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }

        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }
    }

    return 0;
}
```

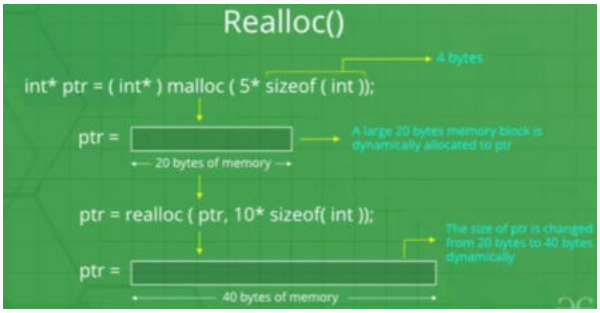
Output:- Enter number of elements: 5
Memory successfully allocated using
calloc. The elements of the array are: 1,
2, 3, 4, 5,

C realloc() method

"realloc" or **"re-allocation"** method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to **dynamically re-allocate memory**. re-allocation of memory maintains the already present value and new blocks will be initialized with the default garbage value.

Syntax of realloc() in C

ptr = realloc(ptr, newSize);
where ptr is reallocated with new size 'newSize'.



Program:-

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

    int main()
```

```
{

    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;

    // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);

    // Dynamically allocate memory using calloc()
    ptr = (int*)calloc(n, sizeof(int));

    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {

        // Memory has been successfully allocated
        printf("Memory successfully allocated using calloc.\n");

        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }

        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }

        // Get the new size for the array
        n = 10;
        printf("\n\nEnter the new size of the array: %d\n", n);

        // Dynamically re-allocate memory using realloc()
        ptr = (int*)realloc(ptr, n * sizeof(int));
```

```
        if (ptr == NULL) {
        printf("Reallocation Failed\n");
        exit(0);
       }

       // Memory has been successfully allocated
       printf("Memory successfully re-allocated using realloc.\n");

       // Get the new elements of the array
       for (i = 5; i < n; ++i) {
          ptr[i] = i + 1;
       }

       // Print the elements of the array
       printf("The elements of the array are: ");
       for (i = 0; i < n; ++i) {
          printf("%d, ", ptr[i]);
       }

       free(ptr);
    }

    return 0;
}
```

Output:-

```
Enter number of elements: 5
Memory successfully allocated using
calloc. The elements of the array are: 1,
2, 3, 4, 5,

Enter the new size of the array: 10
Memory successfully re-allocated using realloc.
The elements of the array are: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
```

Advantages and drawbacks of pointers: -

Advantages of Pointers in C:

1. **Direct Memory Access:**
    - Pointers allow direct access to memory addresses, enabling efficient manipulation of data stored in memory.
2. **Dynamic Memory Allocation:**

- Pointers enable dynamic memory allocation (using functions like malloc(), calloc(), realloc()), allowing memory to be allocated at runtime.

3. **Efficient Function Argument Passing:**
   - Large data structures (e.g., arrays or structs) can be passed to functions by reference, avoiding unnecessary copying and improving performance.
4. **Array and String Manipulation:**
   - Pointers allow easy and efficient manipulation of arrays and strings by pointing to specific memory locations.
5. **Flexible Data Structures:**
   - Pointers are essential for implementing dynamic data structures such as linked lists, trees, and graphs.
6. **Memory Efficiency:**
   - Pointers enable the precise allocation and deallocation of memory, helping optimize memory usage.

Drawbacks of Pointers in C:

1. **Complexity and Error-Prone:**
   - Pointer usage can make code complex and harder to understand, leading to potential mistakes, especially for beginners.
2. **Memory Leaks:**
   - Improper memory management (e.g., failing to free() allocated memory) can cause memory leaks, reducing system performance.
3. **Dangling Pointers:**
   - If a pointer continues to reference memory after it has been freed, it becomes a dangling pointer, leading to undefined behavior.
4. **Pointer Arithmetic Risks:**
   - Incorrect pointer arithmetic can lead to memory corruption, accessing invalid memory, or causing crashes.
5. **Security Vulnerabilities:**
   - Pointers can introduce security risks like buffer overflows, which can be exploited by attackers to execute malicious code.
6. **Difficult to Debug:**

- Debugging pointer-related errors, such as memory corruption or dereferencing null pointers, is more challenging compared to higher-level languages.

7. **Lack of Built-in Safety:**
   - C does not have garbage collection or automatic memory management, requiring manual handling of memory, which increases the potential for errors.
8. **Portability Issues:**
   - Pointer operations may vary across different systems (e.g., 32-bit vs 64-bit architectures), potentially causing compatibility issues in portable code.

UNIT-05

Structures and Files

Structures: Structure definition, initialization and accessing the members of a structure, nested structures, array of structures, structures and functions, structures and pointers, self-referential structures, unions and enumerated data types.

Files: Introduction to files, file operations, reading and writing data on files, error handling during file operations.

Structures: Structure definition:-

A structure is a user defined data type in C and C++ language. Which allows you to combine different data types to store a particular type of record?

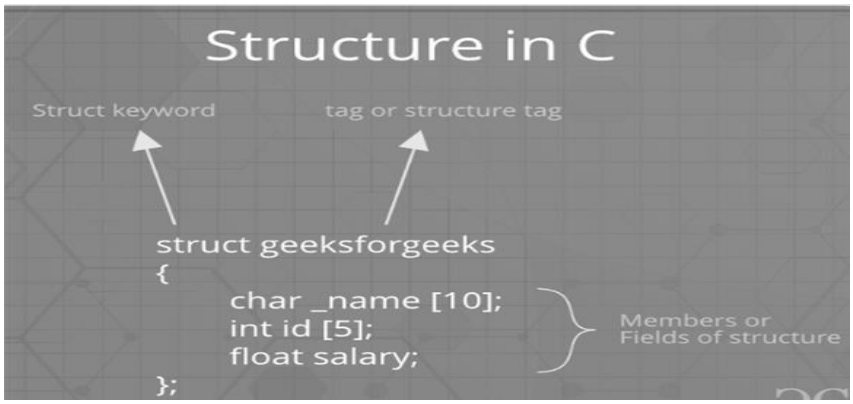
Arrays are used to stored similar type of "N" no.of elements.

In structure we can store 'N' no.of elements of different data types.

That is a structure is a collection of different type of data elements like int, float,double,char etc..,

(Or)

The structure in C is a user-defined data type that can be used to group items of possibly different types into a single type. The **struct keyword** is used to define the structure in the C programming language. The items in the structure are called its **member** and they can be of any valid data type. Additionally, the values of a structure are stored in contiguous memory locations.



How to create a structure?

'struct' keyword is used to create a structure. An example is given below

Syntax: struct { data_type variable1;

data_type variable2;

data_type variable3;

- - - - - - - -

};

Example: struct address

```
{
 char name[50];
char city[50];
char state[20];
int pin;
 };
```

There are four methods of structures:-

1. Define a structure
2. Declare structure variables
3. Initialize structure members/elements
4. Access the structure members

- **Define a structure:-** "struct" is the required keyword. Struct keyword is used to define a structure. Struct define a new data type which is a collection of different type of data.

  Syntax:-Struct structure name
  {
  Data type element1;
  ……..
  Data type element;
  }
  Eg: -
  Struct Book
  {
  Char name [15];
  Int price;
  Int pages;
  };
  Here the **struct Book** declare a structure to hold the details of book which consists of three data fields, namely name, price and pages. These fields are called **Structure elements or members.** Each member can have different data type, like in this case, **name** is of char type and **price** is of int type etc. **Book** is the name of the structure and is called structure tag.

- **Declaring structure variables:-**It is possible to declare variable of a structure ,after the structure is defined. structure variable declaration is similar to the declaration of variables of any other sdata types. Structure variables can be declared in following two ways.

  1. Declaring structure variables separately:-

```
Syntax:-struct structure_name {
   data_type member_name1;
   data_type member_name1;
   ....
   ....
}variable1, varaible2, ...;
```

2. Declaring structure variables with structure definition

Syntax:-struct identity var1,var2,……var-n;

Eg:-

```
main()
{
Struct employee e1,e2,e3;
……
}e1,e2,e3;
```

- **Initialization of structure elements: -**Like any other datatype, Structure variable can also be initialized at compile time.

```
Ex:-Struct Employee
{
Int emp no;
Char ename[20];
Float salary;
};
Main()
{
Struct employee e={101,”jan”,30000};
}
```

- **Accessing Structure Members:-**members Access operator dot(.).

Syntax:-Struct variable . member

```
e.empno
e.empname
e.empsalary
```

Overall Programming for Structures:-

```
/*..Using Structures in C…*/
#include<stdio.h>
#include<conio.h>
Struct employee
{
Int emp no;
Char ename[20];
Float salary;
};
Main()
{
Struct Employee e={101,”jan”,30000};
Clrscr();
```

```
Printf("\n The size of the Structure=%d",size of (e));  (or) Printf("\n The size of the Structure=%d",size of
(struct employee));
Printf("\n\n\n Employee Details");
Printf("\n\n Employee number=%d",e.empno);
Printf("\n\n Employee name=%s",e.ename);
Printf("\n\n Employee salary=%f",e.salary);
}
```

Output:-

The size of structure variable=26

Employee details

……

Employee Number=10 Employee

Name=Jan Employee

Salary=30000.00000

Nested structures:-(With in Structures)

Structures within a structure means nesting of structures. Nesting of structures is permitted in C. A Structure can be embedded with in another structure.

(Or)

Specifying a structure with in another structure, That is one structure has another structure as number variable

Eg:- To Store the "Address" or "Date of Birth".

Syntax:-

```
struct name_1
{
   member1;
  member2;
   .
   .
  member n;
  struct name_2
  {
     member_1;
     member_2;
     .
     .
    member_n;
  } var1;
} var2;
```

Different ways of nesting structure

The structure can be nested in the following different ways:

1. **By separate nested structure**

2. **By embedded nested structure**.
    1. **By separate nested structure:** In this method, the two structures are created, but the dependent structure(Student) should be used inside the main structure(Organization) as a member. Below is the C program to implement the approach:

Ex:-

```
Struct dob
{
int  day;
int mm;
int yy;
};
Struct Student
{
Int id;
Char name[20];
Struct dob d;
};
```

2. By Embedded nested structure: Using this method, allows to declare structure inside a structure and it requires fewer lines of code.

Ex:-

```
Struct Student
{
int id;
Char name[20];
Struct dob;
{
Int dd;
Int mm;
Int yy;
}d;
}s;
```

Accessing Nested Structures members:- Outer Structure variable name, Inner Struct Variable name. Member;

Ex:-s.id; s.name;s.d.dd;s.d.mm;s.d.yy;.

C Nested Structure Example

```
// C program to implement
// the nested structure
#include <stdio.h>
```

```
#include <string.h>

// Declaration of Outer structure
struct College
{
  char college_name[20];
  int ranking;

  // Declaration of Inner structure
  struct Student
  {
    int student_id;
    char name[20];
    int roll_no;

  // Inner structure variable
  } student1;
};


// Driver code
int main()
{
  struct College c1 = {"AITS", 7,
                  {111, "Janu", 278}};

  printf("College name : %s\n",
         c1.college_name);
  printf("Ranking : %d\n",
         c1.ranking);
  printf("Student id : %d\n",
         c1.student1.student_id);
  printf("Student name : %s\n",
         c1.student1.name);
  printf("Roll no : %d\n",
         c1.student1.roll_no);
  return 0;
}
```

Output:-

College name : AITS
Ranking : 7
Student id : 111
Student name : Janu
Roll no : 278

Array Structures:- An array of structures is simply an array where each element is a structure. It allows you to store several structures of the same type in a single array.

Syntax:-Structure tag array name[size];

Ex:-

```
Struct Employee
{
Int empno;
Char ename[20];
Float salary;
};
int main()
{
Struct employeee1,e2,e3;
Read e1 details
Read e2 details
Read e3 details
Printf e1 details
Printf e2 details
Printf e3 details
```

Sample example program:-

```
#include<stdio.h>
Struct employee
{
Int empno;
Char ename[20];
Int sal;
};
Int main()
{
Struct employee e;
printf("\n Enter Employee details…\n");
printf("\n Enter number:");
scanf("%d",&e.empno);
printf("\n Enter name:");
scanf("%d",&e.ename);
```

```
printf("\n Enter salary:");
scanf("%d",&e.sal);
printf("\n employee details are …..\n");
printf("\n\n number \t name\t\t salary");
printf("\n…       );
printf("\n%d\t%s\t\t%d",e.empno,e.name,e.sal);
return 0;
}
```

Output:-

```
Enter employee Details…..
Enter Number:1
"       Name:Janu
"       Salary:25000
Employee details are….
Number      Name      Salary
……..
1           Janu      25000
```

Structures and functions:-

1.Structures in C:

A structure in C is a user-defined data type that allows grouping of variables of different types into a single unit. Each element in a structure is called a member or field.

Syntax:-

```
struct Structure Name
 {
dataType member1;
dataType member2;
 // other members
 };
```

2. Functions in C:

A **function** in C is a block of code that performs a specific task. Functions allow code reusability and modularity. Functions can be defined and called with or without arguments, and they can return values or be void.

```
returnType functionName(parameter1, parameter2, ... ) {
// function body return returnValue;
// if returnType is not void
}
```

In C, you can pass structures to functions in two ways:

1. **Passing structure members to a function** (passing individual elements of a structure to a function).

2. **Passing the entire structure to a function** (passing the whole structure to a function, either by value or by reference).

Let's look at both approaches in more detail:

1. Passing Structure Members to a Function:

This involves passing individual members of the structure as arguments to the function.

```
#include <stdio.h>

struct Student {
    char name[50];
    int age;
    float grade;
};

// Function to display student details by passing individual members
void displayStudentDetails(char name[], int age, float grade) {
    printf("Name: %s\n", name);
    printf("Age: %d\n", age);
    printf("Grade: %.2f\n", grade);
}

int main() {
    struct Student student1 = {"John", 20, 85.5};

    // Passing structure members to the function
    displayStudentDetails(student1.name, student1.age, student1.grade);

    return 0;
}
```

Explanation:

- Here, the individual members of the Student structure (name, age, and grade) are passed separately as arguments to the displayStudentDetails() function.
- This method does not pass the structure as a whole, only the members are passed one by one.

2. Passing the Entire Structure to a Function:

- In this approach, you pass the whole structure to a function. The structure can be passed either **by value** or **by reference (using pointers)**.
  a. Passing Structure by Value:

- When a structure is passed by value, a copy of the structure is made, and the function operates on this copy. Any changes made to the structure inside the function won't affect the original structure.

b. Passing Structure by Reference (Using Pointers):

- When you pass a structure by reference (using a pointer), the function operates directly on the original structure. Any changes made to the structure inside the function will affect the original structure.

Structures and pointers: -

A structure pointer is a pointer variable that stores the address of a structure. It allows the programmer to manipulate the structure and its members directly by referencing their memory location rather than passing the structure itself. In this article let's take a look at structure pointer in C.

(Or)

Pointer holds the address of another variable.

Syntax: - *struct struct_name* ***ptr_name;***

Let's take a look at an example:

```
#include <stdio.h>
struct A {
        int var;
    };

int main() {
        struct A a = {30};

        // Creating a pointer to the structure
        struct A *ptr;

        // Assigning the address of person1 to the pointer
        ptr = &a;

        // Accessing structure members using the pointer
        printf("%d", ptr->var);

        return 0;
    }
```

Output:-30

Accessing Member using Structure Pointers

There are two ways to access the members of the structure with the help of a structure pointer:

1. Differencing and Using (.) Dot Operator.
2. Using (->) Arrow operator.

Differencing and Using (.) Dot Operator

First method is to first dereference the structure pointer to get to the structure and then use the dot operator to access the member. Below is the program to access the structure members using the structure pointer with the **help of the dot operator.**

Using (->) Arrow Operator

C language provides an array operator (->) that can be used to directly access the structure member without using two separate operators. Below is the program to access the structure members using the structure pointer with the help of the Arrow operator.

Help of the dot operator.

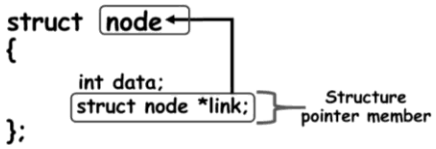
```
#include <stdio.h>
#include <string.h>
struct Student {
    int roll_no;
    char name[30];
    char branch[40];
    int batch;
};

int main() {

    struct Student s1 = {1, "Degree",
"AI&ML,COM", 2024};

     // Pointer to s1
    struct Student* ptr = &s1;

     // Accessing using dot operator
    printf("%d\n", (*ptr).roll_no);
    printf("%s\n", (*ptr).name);
    printf("%s\n", (*ptr).branch);
    printf("%d", (*ptr).batch);

    return 0;
}
Output:- 1
Degree
AI&ML,COM
2024
```

Help of the Arrow operator.

```
#include <stdio.h>
#include <string.h>
struct Student {
    int roll_no;
    char name[30];
    char branch[40];
    int batch;
};

int main() {

    struct Student s1 = {1, "Degree",
"AI&ML,COM", 2024};

     // Pointer to s1
    struct Student* ptr = &s1;

     // Accessing using dot operator
    printf("%d\n", ptr->roll_no);
    printf("%s\n", ptr->name);
    printf("%s\n", ptr->branch);
    printf("%d", ptr->batch);

    return 0;
}
Output:- 1
Degree
AI&ML,COM
2024
```

Self-Referential Structures: -

Self Referential structures are those structures that have one or more pointers which point to the same type of structure, as their member.



How it Works:- Consider a structure node it contains two parts 1. Data 2. Link

Types of Self Referential Structures

1. Self Referential Structure with Single Link
2. Self Referential Structure with Multiple Links

Declaring a structure:-Example Program

```
#Include<stdio.h>
Void main()
{
Struct node
{
Int data;
Struct node *link;
};
Struct node a;
a.data=3;
a.link=null;
Struct node b;
b.data=5;
b.link=null;
a.link=&b;
printf("%d",a.data);
printf("%d",a.link->dat);
}
```

Output:- 3 5

Applications: Self-referential structures are very useful in creation of other complex data structures like:

- Linked Lists
- Stacks
- Queues

- Trees
- Graphs etc.

Unions and Enumerated Data Types: - To Store different data items. May be of Different types or same data types referenced under one name.

(Or)

Union is a collection of different data items.

{Or}

In C, **union** is a user-defined data type that can contain elements of the different data types just like structure. But unlike structures, all the members in the C union are stored in the same memory location. Due to this, only one member can store data at the given point in time.

Mainly Classified into 4 types:-

1.Declaration of a Union 2.Declaration of a Union Variable 3.Initialization of Union Members 4.Accessing Union Members

1. Declaration of a Union:- In this part, we only declare the template of the union, i.e., we only declare the members' names and data types along with the name of the union. No memory is allocated to the union in the declaration.

Syntax:- ***union*** *name {*
type1 member1;
type2 member2;
.
.
};

Ex:- Union Employee
{
Int Emp no;
Char e name[20]; Float salary;
};

2. Declaration of a Union Variable:- We need to define a variable of the union type to start using union members. There are two methods using which we can define a union variable:

Creating Union Variable with Declaration

union name{
type member1;
type member2;

…
} **var1, var2, …;**
Creating Union Variable after Declaration
union name **var1, var2, var3…;**
3.Initialization of Union Members:-

EX:-

```
Union Employee
    {
Int emp no;
Char ename[20];
Float salary;
    };
Int main()
    {
Union employee e={101,”jan”,30000};
e.empno=101;
e.ename=”jan”;
e.salary=30000;
```

4.Accessing Union Members

We can access the members of a union by using the (.) dot operator just like structures.

Syntax:- var1.member1;

Ex:-e.empno; e.ename; e.salary; #include

```
<stdio.h>
// Declaring multiple unions union
A{
        int x;
        char y;
    };
union B{
        int arr[10];
        char y;
    };
int main() {
        // Finding size using sizeof() operator
        printf("Sizeof A: %ld\n", sizeof(union A));
        printf("Sizeof B: %ld\n", sizeof(union B));
        return 0;
    }
```

Output:- Sizeof A: 4

Sizeof B: 4
Sizeof C: 40

Difference between Structures and Unions

Structures	unions
Structure is a collection of data item may or different data types referenced under one name.	Union is a collection of a data item may of different data types referenced under one name.
Structures is defined by "struct" keyword as Struct tag { Dat type member 1; Data type member 2; " " Data type member-n; } var1,var2,….,var-n;	Union is defined by "union" keyword as Union tag { Dat type member 1; Data type member 2; " " Data type member-n; } var1,var2,….,var-n;
In a structure memory is reserved for all members of that structure	In a union memory is reserved only for the largest member of that union.
All the members of structure cannot share the same storage area.	All the members of union can share the same storage area.
All members of structure can be accessed at a time	All members of union cannot be accessed at a time
We can store data for all members in the structure	We can store data for only one member of an union
The Structure variable can be initialized and can give values for all members. Ex:- Struct temp m={10,50.25,'m'};	The union variable cannot be initialized, but if we want to initialize, we can give value to the first member only Ex;- Union Temp m={10};

FILES:-(A file is collection of related records. For example employee file contains employee records.)

Introduction to Files:-

A file represents a sequence of bytes on the disk, where a group of related data is stored; file is created for permanent storage of data.

(Or)

A File is a collection of bytes that contains some information permanently. A file is created in secondary storage device.

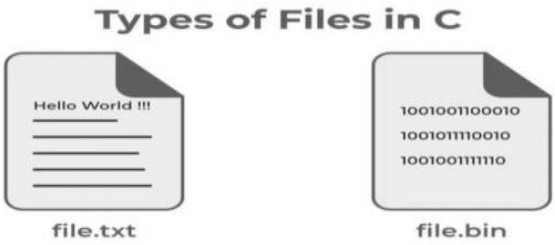
Types of Files:-

A file can be classified into two types based on the way the file stores the data. They are as follows:

- **Text Files**
- **Binary Files**

Text Files:-Text files are normal.txt file 5that you can easily create using notepad, file contains plain text you can easily edit or delete the content.

Binary Files:-Binary files are mostly. In files in your computer, Instead of storing data in plain txt. They store it in binary form (0's and 1's)



FILE OPERATIONS:- In C ,you can perform five major operations on the file, whether text or binary file.

1. Creating a new file → f open()
2. Opening an existing file → f open()
3. Reading data from the file → fgets(),fgets(),fprintf().
4. Writing data into a file → fputs(),fputs(),fscanf()
5. Closing a file → fclose()

Character I/O Functions (f getc() & f putc()):-

The Smallest unit of data written to a file is a character, which is 1 byte long, these functions Read/Write only single character from to a file is called character I/O functions.

The following are different file modes used in C language:

MODE	MEANING
"r"	Opens text file for reading
"w"	Opens text file for writing
"a"	Opens text file in append mode
"rb"	Opens binary file for reading
"wb"	Opens binary file for writing
"ab"	Opens binary file for append
"r+"	Opens text file for reading and writing
"w+"	Opens text file for writing and reading

1. **f getc():-** This function is used to **Read** a characters from a file that has been opend in read mode.
   - **f open():-** f open() function is used to **create a new file** or to **open an existing file.**
     Syntax:-file*p=f open ("file name","mode");

EX:-fp=f open ("c:\janu\123.txt","mode");

- **f close():-**f close() function is used to close an already opened file.
  Syntax:-fp=f close (file pointer);
  EX:-fp=f close (ptr);

fgetc():- This function is used to read a single character from the file and stores it in a variable.

Syntax:- variable=fgetc (file pointer);

Sample program:-

```
#include<stdio.h>
#include<conio.h>
void main()
{
File*ptr;
Char ch;
Ptr=fopen("c:\robo\xyz.txt",'r');
if(ptr==null)
{
Printf("file unable to open");
}
Else
{
Printf("file content is :");
}
While (ch=fgetc (ptr))!=eof)
{
Printf("%c",ch);
}
F close (ptr);
getch ();
}
```

2. **f putc():**-This function is used to **write** a characters to the file
   - **f putc() function:-** This function is used to store a single character into file. The general format is as follows.
     Syntax:-fp=f putc (char variable/value. File pointer);
     EX:-fp=f putc('R'.ptr);

Sample program:-

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

```
File*ptr;
Char ch='R';
Ptr=fopen("c:\robo\xyz.txt",'w');
If(ptr==null)
{
Print f("file unable to open");
}
Else
Printf("file opend successfully");
f putc(ch,ptr);
f putc('O',ptr);
f putc('B',ptr);
f putc('O',ptr);
f close(ptr);
getch ();
}
```

Output:-ROBO

String I/O Functions (f gets () & f puts ()):-

f puts():- The f puts () function is used to write string (array of characters) into the file.

Syntax:- f puts(char str[],file*ptr);

The f puts () function takes two arguments, first is the string to be written to the file and second is the file pointers, where the string will be written.

f gets():- The f gets () function is used to read string (array of characters) from the file.

Syntax:- f gets(char str[],int n, file*ptr);

The f gets() function takes three arguments.

1. First is the string read from the file
2. Second is size of string.
3. Third is the file pointer from where the string will be read.
   F gets() function will return null value when it reads "EOF" (End of file).

f puts() function sample program:-

```
# include <stdio.h>

#include<conio.h>
void main()
{
File*fptr;
Char str[100];
f ptr=f open("janu.txt","w");
if (f ptr== null)
```

```
{
Printf("file unable open");
}
Else
{
Printf ("Enter text:/n");
gets(str);
f puts(str.f ptr);
f close(fptr);
}
getch ();
}
```

f gets() function sample program:-

```
# include <stdio.h>

#include<conio.h>
void main()
{
File*fptr;
Char str[100];
f ptr=f open("janu.txt","r");
if (f ptr== null)
{
Printf("file unable open");
}
Else
{
Printf ("Data in the file:/n");
f gets(str,50,f ptr);
puts(str);      (or) printf("%s',str);
f close(fptr);
}
getch ();
}
```

Reading and Writing data on files:-Reading and Writing data to files in C is a fundamental task that allows programs to store and retrieve data.

Reading and Writing Strings:- #include

<stdio.h>

```
Int main()
    {
FILE *file=fopen('example.txt","w");
If (file == null)
    {
Printf("error opening file!\n");
Return 1;
}   //write a string to the file
F puts("This is a sample text.\n",file);
f close(file);  //close the file after writing
file = f open("example.txt","r");  //now read the string from the file.
if( file==null){
printf9'Error opening file\n");
return 1;
    }
Char buffer[100];  //read a line of text from the file
If(fgets9buffer,size of (buffer),file)!=null){ Printf("read
from file:%s",buffer);
    }
f close(file); return
o;
    }
```

Out put:- Error Opening File.

ERROR HANDLING DURING FILE OPERATIONS:-

In C programming, proper **error handling during file operations** is essential for ensuring that the program behaves as expected, even when issues arise with file access or manipulation. Below are detailed explanations and sample code snippets to handle errors during file operations such as opening, reading, writing, and closing files.

Key Functions for File Handling in C:

1. **Opening a File**: fopen()
2. **Reading from a File**: fgetc(), fgets(), fread()
3. **Writing to a File**: fputc(), fputs(), fwrite()
4. **Closing a File**: fclose()
5. **Error Checking**: ferror(), feof()

Error Handling

It's essential to check for errors after file operations using ferror() and feof().

- **ferror(file)**: Checks if an error occurred during the last file operation.
- **feof(file)**: Checks if the end of the file has been reached.

Program me:-

```
#include
<stdio.h> int
main() {
    FILE *file =
    fopen("example.txt", "r"); if
    (file == NULL) {
       perror("Error opening
       file"); return 1;
    }
    char ch;
    while ((ch = fgetc(file)) !=
       EOF) { putchar(ch);
    }
    if (ferror(file)) {
       perror("Error reading file");
    } else if (feof(file)) {
       printf("\nEnd of file reached.\n");
    }
    fclos
    e(file
    );
    retur
```

```
    n 0;
}
```

Output:-Error opening file:No such file or directory.

ALL THE BEST