UNIT-I: Introduction to Programming and Problem Solving

Introduction to Programming Languages, Basics of a Computer Program- Algorithms, Algorithmic approach, characteristics of algorithm, Problem solving strategies: Top-down approach, Bottom-up approach, Time and space complexities of algorithms. flowcharts (Using Dia Tool), pseudo code.

Structure of C Program Introduction to Compilation and Execution, Primitive Data Types, Variables, and Constants, Basic Input and Output, Operators, keywords, identifiers, Type Conversion, and Casting.

Language is a mode of communication that is used to share ideas and opinions with each other. For example, if we want to teach someone, we need a language that is understandable by both communicators.

What is a Programming Language?

A programming language is a computer language that is used by programmers (developers) to communicate with computers.

It is a set of instructions written in any specific language (C, C++, Java, Python) to perform a specific task.

Programming languages are often categorized into three levels based on their **abstraction from the machine hardware.**

1. Low-Level
2. Middle-Level
3. High-Level Languages.

1) Low-Level Programming Language:

Low-level languages are closer to machine code or hardware. They provide little to no abstraction and are highly dependent on the machine architecture, making them efficient but more difficult to write and understand for humans.

Characteristics:

Close to hardware: Offers minimal abstraction from the computer's hardware.

Efficient: Programs written in low-level languages run fast since they communicate directly with the hardware.

Difficult to write and understand: Programming in low-level languages requires knowledge of hardware details, making the code harder to write, debug, and maintain

Example: Machine Code, Assembly Code

2) Middle-Level Programming Language:

A middle-level programming language is a type of programming language that has features of both low-level and high-level languages. It provides some abstraction from the hardware, like high-level languages, but also allows for direct interaction with hardware, like low-level languages.

Characteristics:

Closer to the machine than high-level languages: It can access memory and hardware directly.

Easier to use than low-level languages: It includes more human-readable constructs, making it simpler to write code.

Balanced: It strikes a balance between efficiency (performance) and ease of programming. .

Examples: C, C++

3) High-Level Programming Language:

High-level programming languages are programming languages that are designed to be easy for humans to read and write. High-level languages are user-friendly, abstract away hardware complexities, and allow developers to focus on writing logical, efficient code.

Characteristics:

Human-readable syntax: The syntax is closer to natural language or mathematical notation, making it easier to understand and write.

Portable: High-level languages are not tied to a specific type of computer hardware, so the same code can often be run on different machines.

Execution: Slower execution compared to low- and middle-level languages because of higher abstraction.

Examples: Java, Python, Ruby, Php etc….

Types of Programming Paradigm:

1. Procedural programming languages
2. Functional programming languages
3. Object-oriented programming languages (OOP)
4. Scripting languages
5. Logic programming languages

Procedural programming languages

A procedural language follows a sequence of statements or commands to achieve a desired output. Each series of steps is called a procedure, and a program written in one of these languages will have one or more procedures within it.

Common examples of procedural languages include C and C++, Pascal, BASIC

Object Oriented Programming Languages

The programming languages which are used to develop the applications by using class and object are called Object Oriented Programming Languages.

Common examples of OOP languages include Java, Python, PHP, C++, Ruby

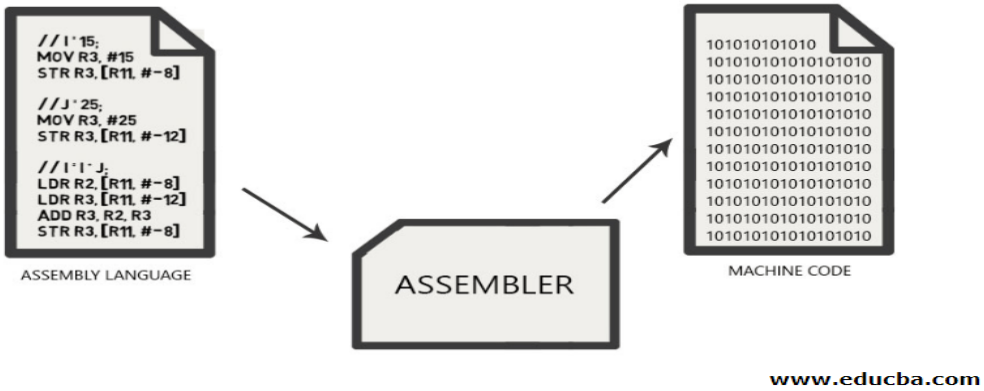
Translator:

A translator refers to a type of software that converts code written in one programming language into another language or into a machine-readable format.

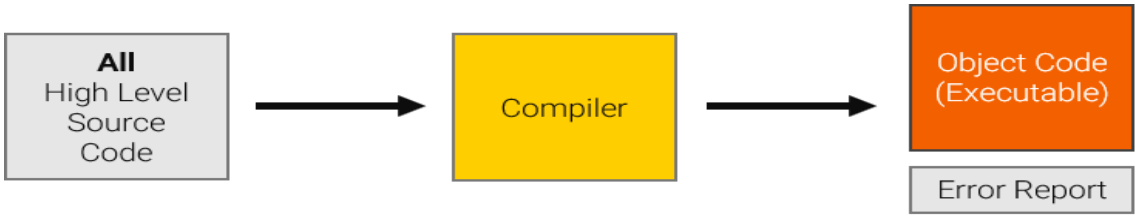
There are different types of translators in programming.

a) **Assembler**
b) **Compiler**
c) **Interpreter**

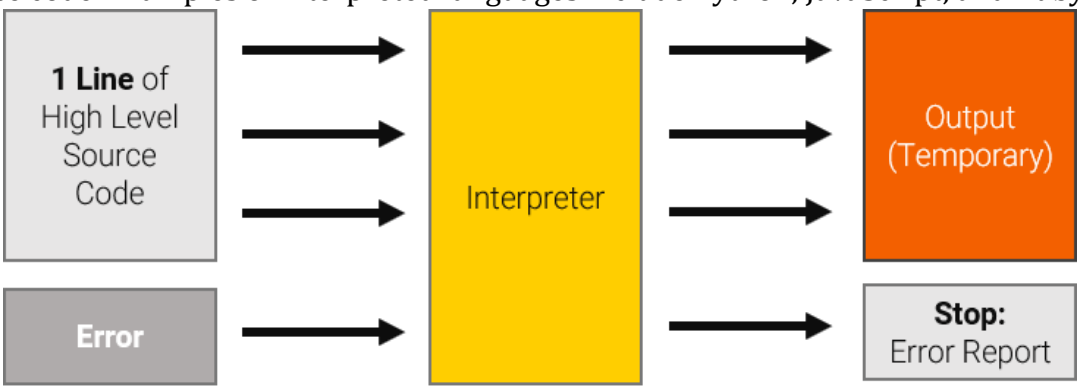
Assembler: An assembler translates assembly language, which is a low-level language closely related to machine code into actual machine code (binary code) that the computer's CPU can execute.

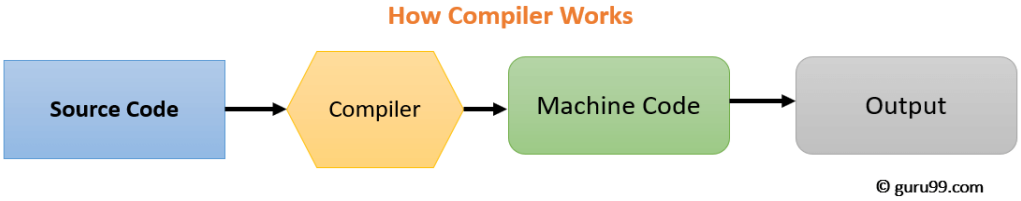


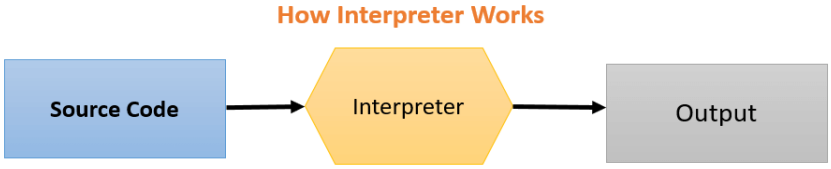
Compiler: A compiler translates code from a high-level programming language (like C, Java, or Python) into machine code, which is a low-level language that the computer's hardware can execute directly. The entire source code is typically translated at once, producing an executable file.



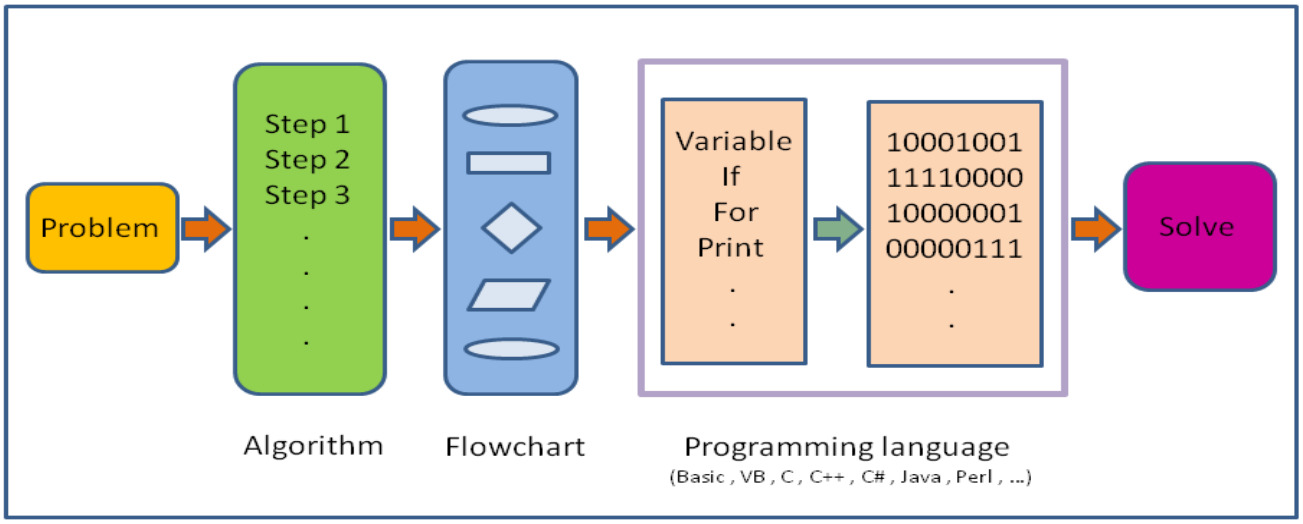
Interpreter: An interpreter translates and executes code line-by-line or statement-by-statement. Instead of producing a separate executable file, an interpreter directly executes the instructions in the source code. Examples of interpreted languages include Python, JavaScript, and Ruby.







Problem Solving:



Algorithm:
An algorithm is a step-by-step process, or a set of rules designed to perform a specific task or solve a particular problem. Algorithms serve as a blueprint for writing code, guiding the program through a sequence of operations to achieve a desired outcome.

Characteristics of an Algorithm:
Input: The algorithm takes zero or more inputs.
Output: The algorithm produces one or more outputs.
Definiteness: Each step of the algorithm is clear and unambiguous.
Finiteness: The algorithm must terminate after a finite number of steps.
Effectiveness: Each step of the algorithm is basic enough to be performed, ideally by a computer.

Examples:
Algorithm-1: Addition of Two Numbers
Step-1: Start
Step-2: Input: Read two numbers, num1 and num2.
Step-3: Process: Calculate the sum of the two numbers by performing the operation:
sum = num1 + num2
Step-4: Output: Display or return the result, sum.
Step-5: End

Algorithm-2: Finding Simple Interest
Step-1: Start
Step-2: Input: Read the values for P (Principal), R (Rate of Interest), and T (Time in years).
Step-3: Process:
Calculate the Simple Interest using the formula:
SI = (P * R * T) / 100
Step-4: Output: Display or return the result, SI.
Step-5: End

Algorithm-3: Finding the Biggest of Two Numbers
Step-1: Start
Step-2: Input: Read two numbers, num1 and num2.
Step-3: Process:
- If num1 > num2, then num1 is the biggest.
- Otherwise num2 is the biggest.

Step-4: Output: Display or return the biggest number.
Step-5: End

Algorithm-4: Check if a Number is Even or Odd
Step-1: Start
Step-2: Input: Read the number num.
Step-3: Process:
- If num % 2 == 0, then the number is **even**.
- Otherwise, the number is **odd**.

Step-4: Output: Display whether the number is "Even" or "Odd".
Step-5: End

Algorithm-5: Finding the Factorial of a Given Number
Step-1: Start
Step-2: Input: Read the number n.
Step-3: Process:
- Initialize fact = 1.
- If n == 0 or n == 1, fact = 1 (since 0! = 1 and 1! = 1).
- For i = 2 to n,
  do fact = fact * i.

Step-4: Output: Display or return fact.
Step-5: End

Algorithm-6: Check if a Number is Prime
Step-1: Start
Step-2: Input: Read the number n.
Step-3: Process:
- If n <= 1, then the number is **not prime**.
- For i = 2 to sqrt(n):
  - If n % i == 0, then n is **not prime** (it has a divisor other than 1 and itself).
- If no divisors are found, n is **prime**.

Step-4: Output: Display or return whether the number is "Prime" or "Not Prime".
Step-5: End

Algorithm-7: Finding the Sum of Digits of a Given Number
Step-1: Start
Step-2: Input: Read the number num.
Step-3: Process:
- Initialize sum = 0.
- While num > 0:
  - Extract the last digit using digit = num % 10.
  - Add the digit to sum.
  - Remove the last digit from num using num = num // 10 (integer division).

Step-4: Output: Display or return the value of sum.
Step-5: End

Algorithm-8: Check if a Number is an Armstrong Number
Step-1: Start
Step-2: Input: Read the number num.
Step-3: Process:
- Initialize sum = 0.
- Determine the number of digits n in num.
- Initialize a variable temp to num (to preserve the original number).
- While temp > 0:
  - Extract the last digit using digit = temp % 10.
  - Calculate the power of the digit raised to n: power = digit^n.
  - Add the power to sum: sum = sum + power.
  - Remove the last digit from temp using temp = temp // 10.
- After the loop, compare sum with num.

Step-4: Output: Display whether the number is "Armstrong" or "Not Armstrong".

Step-5: End

Algorithm-9: Implement the Simple Calculator
Steps for the Algorithm:
Step-1: Start.
Step-2: Input two numbers (num1 and num2).
Step-3: Input an operator (+, -, *, /).
Step-4: Check the operator:

- If the operator is +, then calculate num1 + num2.
- If the operator is -, then calculate num1 - num2.
- If the operator is *, then calculate num1 * num2.
- If the operator is /, then:
  - Check if num2 is not zero (since division by zero is not allowed).
  - If num2 is not zero, calculate num1 / num2.
  - If num2 is zero, output an error message ("Division by zero error").
- If the operator is invalid (i.e., not one of +, -, *, /), output an error message ("Invalid operator").

Step-5: Output the result of the operation.
Step-6: End.

Algorithm-10: Check for the given year is leap year or not.
Step-1: Start.
Step-2: Input the year.
Step-3: Process: Check the leap year condition using a single logical statement:

- If the year is divisible by 400 **OR** (divisible by 4 **AND** not divisible by 100), then it is a leap year.
- Otherwise, it is not a leap year.

Step-4: Output the result ("Leap year" or "Not a leap year").
Step-5: End.

Problem Solving Approaches:
When solving problems, especially in computer science and software engineering, two common approaches are Top-Down and Bottom-Up. These approaches are widely used in algorithm design, dynamic programming, system development, and programming in general.
1. Top-Down Approach:
Definition: The **top-down approach** starts by breaking down a complex problem into smaller, more manageable sub-problems. The idea is to start from the highest level of abstraction and then progressively refine and decompose it into more specific components or steps.
Steps:

1. **Start with the big problem** and think about its general solution.
2. **Divide the problem** into smaller sub-problems.
3. **Solve each sub-problem** individually, refining each until you get to the smallest, most specific steps.
4. **Integrate the sub-problems** to solve the original, larger problem.

Advantages:

- Promotes clear, structured thinking.
- Easier to understand the big picture before diving into details.
- Helps in breaking down complex tasks, making them easier to manage.

Disadvantages:

- Can lead to redundant sub-problems if not careful (without optimization).
- Might overlook specific optimizations in lower-level components.

2. Bottom-Up Approach

Definition: The **bottom-up approach** starts by solving the simplest or smallest sub-problems first, and then building up solutions to more complex sub-problems by combining the solutions of the simpler ones.

Steps:

1. **Identify the base cases** or smallest sub-problems.
2. **Solve the smallest problems** first.
3. **Combine the solutions** of these smaller problems to build up to the solution of the larger problem.
4. **Repeat until** you reach the solution to the original, large problem.

Advantages:

- Avoids redundancy by solving each sub-problem once (more efficient).
- Often leads to more optimized solutions.
- Well-suited for **iterative** implementations.

Disadvantages:

- Can be harder to conceptualize, especially for large problems.
- May involve more initial work in defining and identifying base cases and how to combine them.

Examples of Usage

1. Fibonacci Sequence Calculation:

- **Top-Down (Recursive with Memoization):**
  - Break the problem as fib(n) = fib(n-1) + fib(n-2), and recursively calculate fib(n-1) and fib(n-2), while storing the results to avoid re-computation.
- **Bottom-Up (Iterative with Tabulation):**
  - Start with fib(0) and fib(1) and build up to fib(n) by iteratively solving from the base cases.

Time and Space Complexity of an algorithm:

Generally, there is always more than one way to solve a problem in computer science with different algorithms. Therefore, it is highly required to use a method to compare the solutions in order to judge which one is more optimal.

There are two such methods used, **time complexity** and **space complexity**

Time Complexity: The time required by the algorithm to solve given problem is called ***time complexity*** of the algorithm. The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input.

Ex-2: Given an array of n numbers, find if a specific number exists in the array.

Time Complexity Analysis:

Algorithm:

Step 1: Start

Step2: Initialize the loop: i = 0.

Step 2: for i=0 to n n-1:

Compare each element: arr[i] = target.

Continue until the element is found or until all elements have been checked.

Step3: if target found print 'YES'

Otherwise print 'NO'

Step 4: Stop

Time Complexity:

- **Best Case**: O(1) (if the target is at the beginning).
- **Worst Case**: O(n) (if the target is at the end or not present).
- **Explanation**: In the worst case, the algorithm iterates through every element of the array, so the time complexity is **O(n)**.

Space Complexity:

The amount of memory required by the algorithm to solve given problem is called **space complexity** of the algorithm.

Ex: Find the Space Complexity of an algorithm to find the element through linear search.

Space Complexity Analysis:

- The algorithm uses a few fixed variables:
    - arr[] (the input array) is given as input and does not affect the space complexity.
    - target (the element being searched for) is also given.
    - A counter variable i is used to traverse the array.

Since no additional memory is used other than these fixed variables, the space complexity remains constant, regardless of the array's size.

Space Complexity:

- **O(1)** (constant space).

C Programming:

C is a general-purpose programming language created by Dennis Ritchie at the Bell Laboratories in 1972. It is a very popular language, despite being old. The main reason for its popularity is because it is a fundamental language in the field of computer science.

C is strongly associated with UNIX, as it was developed to write the UNIX operating system.

Features of C Programming:

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

2) System Programming Language: C language is a system programming language because it can be used to do low-level programming (for example driver and kernel).

3) Structured programming language: C is a structured programming language in the sense that we can break the program into parts using functions. So, it is easy to understand and modify. Functions also provide code reusability.

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

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

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

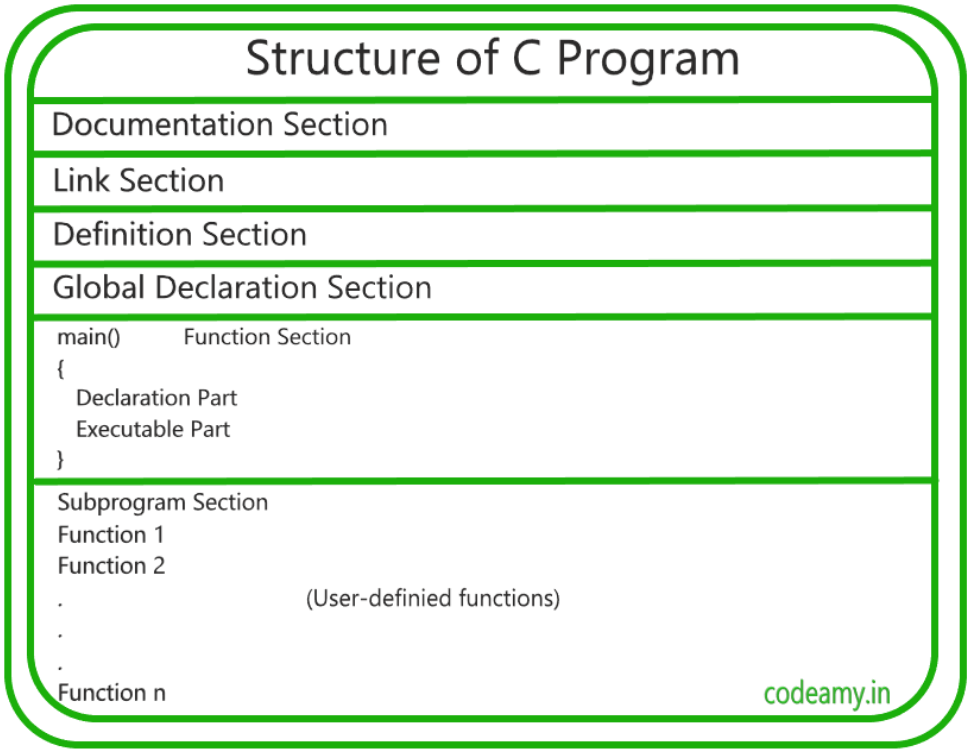
7) Pointer: C provides the feature of pointers. We can directly interact with the memory by using the pointers. We can use pointers for memory, structures, functions, array, etc.

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

9) Recursion: In C, we can call the function within the function. It provides code reusability for every function. Recursion enables us to use the approach of backtracking.

Structure of C Program:

A The structure of a **C program** consists of several sections that follow a standard organization. While some parts are optional depending on the complexity of the program, every C program generally adheres to a similar format.



1) Documentation Section

This section consists of comment lines which include the name of the program, the name of the programmer, the author and other details like time and date of writing the program. Documentation section helps anyone to get an overview of the program.

```
/*Documentation Section:
  Program Name: Program to find the area of circle
  Author: M.Srinu
  Date : 12/09/2024
  Time : 10 AM
*/
```

2) **Link Section**

The link section consists of the header files of the functions that are used in the program. It provides instructions to the compiler to link functions from the system library such as using the #include directive.

Example:

```
#include<stdio.h>                    //link section
#include<conio.h>                    //link section
```

3) Definition Section

All the symbolic constants are written in the definition section. Macros are known as symbolic constants.

```
#define PI 3.14                      //definition section
```

4) Global Declaration Section

The global variables that can be used anywhere in the program are declared in the global declaration section. This section also declares the user defined functions.

```
float area;                          //global declaration section
```

5) main() Function Section

It is necessary to have one main() function section in every C program. This section contains two parts, declaration and executable part. The declaration part declares all the variables that are used in executable part. These two parts must be written in between the opening and closing braces. Each statement in the declaration and executable part must end with a semicolon (;). The execution of the program starts at opening braces and ends at closing braces.

```
int main()
{
        float r;                                        //declaration part
        printf("Enter the radius of the circle\n"); //executable part start here
        scanf("%f",&r);
        area=PI*r*r;
        printf("Area of the circle=%f \n",area);
        message();
}
```

6) Subprogram Section

The subprogram section contains all the user defined functions that are used to perform a specific task. These user defined functions are called in the main() function.

If the program is a multifunction program, then the sub program section contains all the user-defined functions that are called in the main () function. User-defined functions are generally placed immediately after the main () function, although they may appear in any order.

```
 void message()
{
        printf("This Sub Function \n");
        printf("we can take more Sub Function \n");
}
```

Program-001:

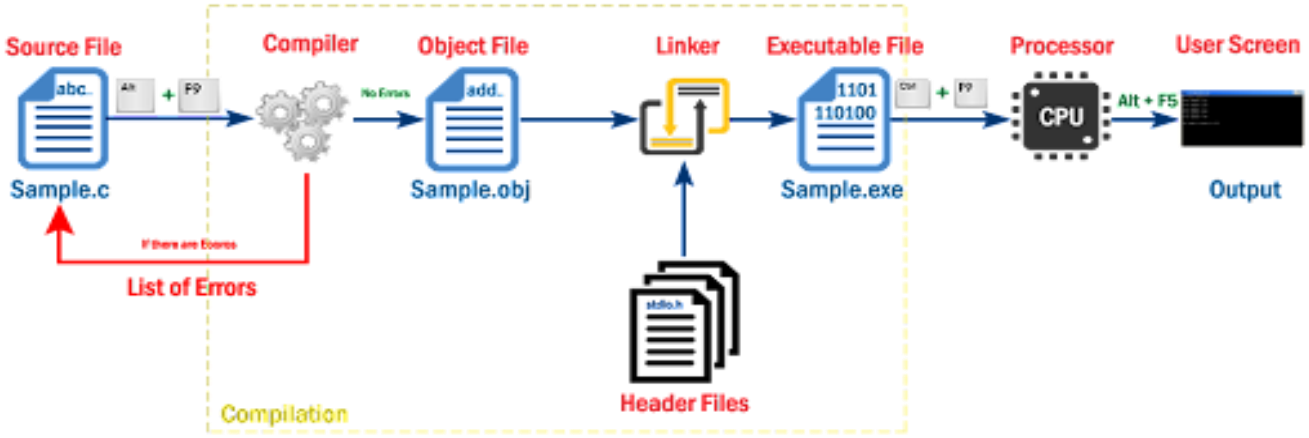
```
/*Documentation Section:
   Program Name: Program to find the area of circle
   Author: M.Srinu
   Date  : 12/09/2024
   Time  : 11 AM
*/
#include<stdio.h>                       //link section
#include<conio.h>                       //link section
#define PI 3.14                         //definition section
float area;                             //global declaration  section
void message();

int main()
{
 float r;                                       //declaration part
 printf("Enter the radius of the circle\n");    //executable part
 scanf("%f",&r);
 area=PI*r*r;
 printf("Area of the circle=%f \n",area);
 message();
}
void message()
{
        printf("This Sub Function \n");
        printf("we can take more Sub Functions \n");
}
```

Program-002:

```
#include                /* Link section */
int total=0 ;           /* Global declaration, definition section */
int sum(int,int);       /* Function declaration section */
int main()              /* Main function */
{
        printf("C programming basics & structure of C programs \n");
        total=sum(6,6);
        printf("sum=%d\n",total);
}
```

```
int sum(int a, int b) /* User defined function */
{
        return a+b;
}
```

Compilation and Execution of C Program:



Token:

A token in C can be defined as the smallest individual element of the C programming language that is meaningful to the compiler. It is the basic component of a C program.

Types of Tokens in C

The tokens of C language can be classified into six types based on the functions they are used to perform. The types of C tokens are as follows:

Keywords:

- The keywords are pre-defined or reserved words in a programming language. Each keyword is meant to perform a specific function in a program.
- Keywords are the words whose meaning has already been explained to the C compiler and their meanings cannot be changed.
- Keywords can be used only for their intended purpose.
- Keywords cannot be used as user-defined variables.
- All keywords must be written in lowercase.

C language supports **32** keywords which are given below:

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

Identifiers:

Identifiers are user-defined names consisting of an arbitrarily long sequence of letters and digits with either a letter or the underscore(_) as a first character.

Certain rules should be followed while naming c identifiers which are as follows:

- They must begin with a letter or underscore(_).
- They must consist of only letters, digits, or underscore. No other special character is allowed.
- It should not be a keyword.
- It must not contain white space.
- It should be up to 31 characters long as only the first 31 characters are significant.

Ex: int a,b,c; //where a,b and c are identifiers and int a keyword.

Constants:

The constants refer to the variables with fixed values. They are like normal variables but with the difference that their values cannot be modified in the program once they are defined. Constants may belong to any of the data types.

Ex: const int x=10;

const float pi=3.142f;

Strings:

Strings are nothing but an array of characters ended with a null character ('\0'). This null character indicates the end of the string. Strings are always enclosed in double quotes. Whereas, a character is enclosed in single quotes in C.

Ex: char branch[20]={'C','S','E'};

char name[]="Rajesh";

Special Symbols:

The following special symbols are used in C having some special meaning and thus, cannot be used for some other purpose. Some of these are listed below:

- **Brackets[]:** Opening and closing brackets are used as array element references. These indicate single and multidimensional subscripts.

- **Parentheses():** These special symbols are used to indicate function calls and function parameters.
- **Braces{}:** These opening and ending curly braces mark the start and end of a block of code containing more than one executable statement.
- **Comma (,):** It is used to separate more than one statement like for separating parameters in function calls.
- **Colon(:):** It is an operator that essentially invokes something called an initialization list.
- **Semicolon(;):** It is known as a statement terminator. It indicates the end of one logical entity. That's why each individual statement must be ended with a semicolon.
- **Asterisk (*):** It is used to create a pointer variable and for the multiplication of variables.
- **Assignment operator(=):** It is used to assign values and for logical operation validation.
- **Pre-processor (#):** The preprocessor is a macro processor that is used automatically by the compiler to transform your program before actual compilation.
- **Period (.):** Used to access members of a structure or union.
- **Tilde(~):** Bitwise One's Complement Operator.

Operators:

Operators are symbols that trigger an action when applied to C variables The data items on which operators act are called operands.

Depending on the number of operands that an operator can act upon, operators can be classified as follows:

Unary Operators: Those operators that require only a single operand to act upon are known as unary operators. For example, **increment and decrement** operators

Binary Operators: Those operators that require two operands to act upon are called binary operators. Binary operators can further be classified into:

1. Arithmetic operators
2. Relational Operators
3. Logical Operators
4. Assignment Operators
5. Bitwise Operator

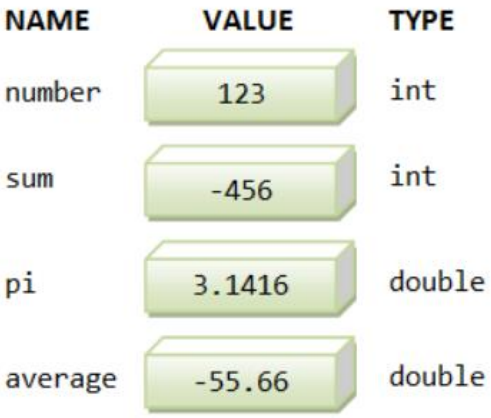
Ternary Operator: The operator that requires three operands to act upon is called the ternary operator. Conditional Operator(?) is also called the ternary operator.

Variable:

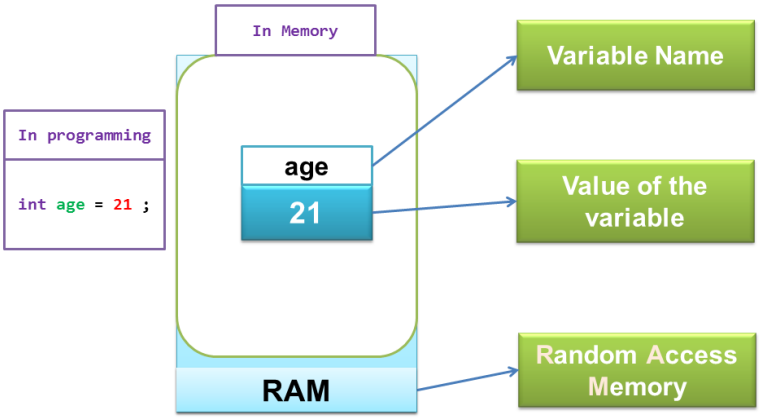
Variable is a name of the memory location where the actual value is to be stored. A **variable in C** is a memory location with some name that helps store some form of data and retrieves it when required.

Syntax:

```
        datatype variable_name;          //Variable declaration
        datatype variable_name=value;    //Variable initialization
Ex:     int number=123,sum=-456;
        double pi=3.1416,average=-55.66;
```



A variable has a *name*, stores a *value* of the declared *type*



The C variables can be classified into the following types:

1. Local Variables
2. Global Variables
3. Static Variables
4. Automatic Variables
5. Extern Variables
6. Register Variables

1) Local Variables:

A **Local variable in C** is a variable that is declared inside a function or a block of code. Its scope is limited to the block or function in which it is declared.

Program-003:

```
#include <stdio.h>
void function();
int main()
{
    function();
}
void function()
{
  int x = 10;        // local variable
  printf("%d", x);
}
```

Output:

10

2) Global Variables:

A Global variable in C is a variable that is declared outside the function or a block of code. Its scope is the whole program i.e. we can access the global variable anywhere in the C program after it is declared.

Program-004:

```
// C program to demonstrate use of global variable
#include <stdio.h>
void function1();
void function2();
int x = 20;                          // global variable
int main()
{
   function1();
   function2();
   return 0;
}
void function1()
{
     printf("Function 1: %d\n", x);
}


void function2()
{
     printf("Function 2: %d\n", x);
}
```

3) Static Variables:

A static variable in C is a variable that is defined using the static keyword. It can be defined only once in a C program and its scope depends upon the region where it is declared (can be global or local).

The default value of static variables is zero.

Syntax:

```
static data_type variable_name = initial_value;
```

As its lifetime is till the end of the program, it can retain its value for multiple function calls as shown in the example.

Program-005:

```
// C program to demonstrate use of static variable
#include <stdio.h>
void function()
{
   int x = 20;                  // local variable
   static int y = 30;           // static variable
   x = x + 10;
   y = y + 10;
   printf("\tLocal Variable X: %d\n\tStatic Variable Y: %d\n", x, y);
}
int main()
{
```

```
    printf("First Call\n");
    function();
    printf("Second Call\n");
    function();
    printf("Third Call\n");
    function();
    return 0;
}
```

Output:

```
First Call
    Local: 30
    Static: 40
Second Call
    Local: 30
    Static: 50
Third Call
    Local: 30
    Static: 60
```

4) Automatic Variable:

All the local variables are automatic variables by default. They are also known as auto variables. Their scope is local, and their lifetime is till the end of the block. If we need, we can use the auto keyword to define the auto variables.

The default value of the auto variables is a garbage value.

Syntax: auto data_type variable_name;

or

data_type variable_name; //(in local scope)

Program-006:

```
// C program to demonstrate use of automatic variable
#include <stdio.h>
void function()
{
    int x = 10;              // local variable (also automatic)
    auto int y = 20;         // automatic variable
    printf("Auto Variable: %d", y);
}
int main()
{
    function();
    return 0;
}
```

Output:

```
Auto Variable: 20
```

Note: In the above example, both x and y are automatic variables. The only difference is that variable y is explicitly declared with the auto keyword.

5) External Variables:

External variables in C can be shared between multiple C files. We can declare an external variable using the extern keyword. Their scope is global and they exist between multiple C files.

Syntax: extern data_type variable_name;

Example:

Program-007:

File1: myFile1.h

```
extern int x=10;                    //external variable (also global)
```

File2: myFile2.C

```
#include "myfile.h"
#include <stdio.h>
void printValue()
{
        printf("Global variable: %d", x);
}
int main()
{
        printValue();
}
```

Output:

Global variable: 10

6) Register Variables:

Register variables in C are those variables that are stored in the CPU register instead of the conventional storage place like RAM. Their scope is local and exists till the end of the block or a function. These variables are declared using the register keyword.

The default value of register variables is a garbage value.

Syntax:

register data_type variable_name = initial_value;

Example:

Program-008:

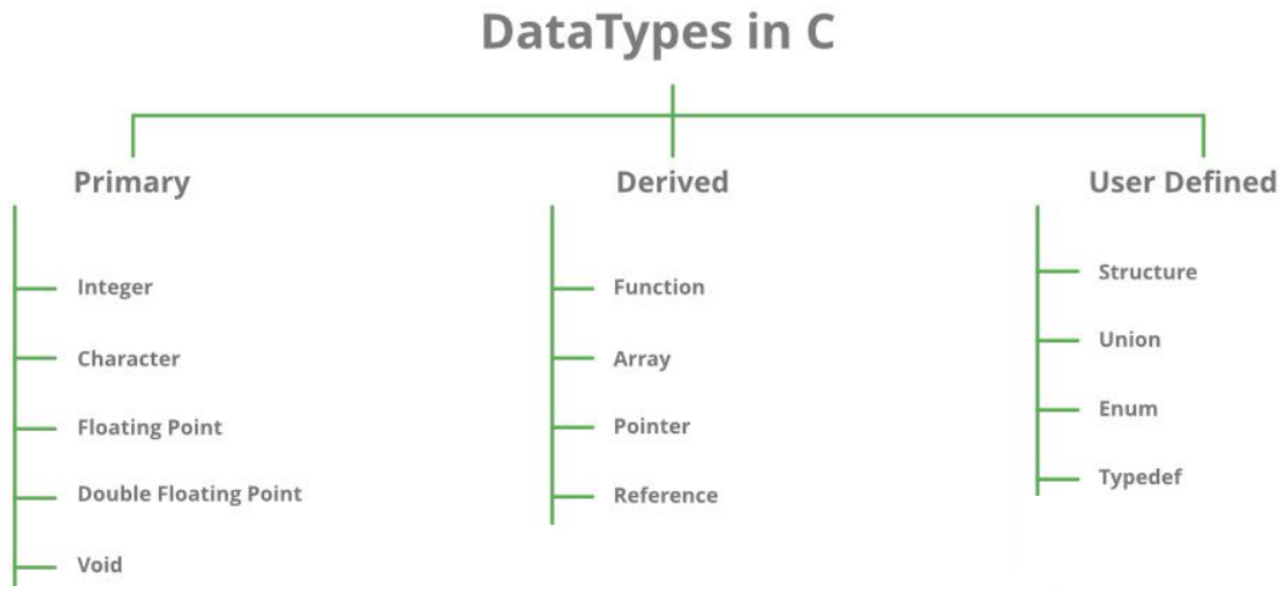
```
// C program to demonstrate the definition of register variable
#include <stdio.h>
int main()
{
        register int var = 22;
        printf("Value of Register Variable: %d\n", var);
        return 0;
}
```

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

Data types:

Each variable in C has an associated data type. It specifies the type of data that the variable can store like integer, character, floating, double, etc. Each data type requires different amounts of memory and has some specific operations which can be performed over it.

Types	Description
Primitive Data Types	Primitive data types are the most basic data types that are used for representing simple values such as integers, float, characters, etc.
User Defined Data Types	The user-defined data types are defined by the user himself.
Derived Types	The data types that are derived from the primitive or built-in datatypes are referred to as Derived Data Types.



Different data types also have different ranges up to which they can store numbers. These ranges may vary from compiler to compiler. Below is a list of ranges along with the memory requirement and format specifiers.

Data Type	Size (bytes)	Range	Format Specifier
short int	2	-32,768 to 32,767	%hd
unsigned short int	2	0 to 65,535	%hu
unsigned int	4	0 to 4,294,967,295	%u
int	4	-2,147,483,648 to 2,147,483,647	%d
long int	4	-2,147,483,648 to 2,147,483,647	%ld
unsigned long int	4	0 to 4,294,967,295	%lu

long long int	8	-(2^63) to (2^63)-1	%lld
unsigned long long int	8	0 to 18,446,744,073,709,551,615	%llu
signed char	1	-128 to 127	%c
unsigned char	1	0 to 255	%c
float	4	1.2E-38 to 3.4E+38	%f
double	8	1.7E-308 to 1.7E+308	%lf
long double	16	3.4E-4932 to 1.1E+4932	%Lf

Datatype Modifiers:
Modifiers are keywords in C which changes the meaning of basic data type in C. It specifies the amount of memory space to be allocated for a variable. Modifiers are prefixed with basic data types to modify the memory allocated for a variable.

Need of Datatype Modifiers:
We use int to store the Salary of the employee as we are assuming that the salary will be in whole numbers. An integer data type takes 4 bytes of memory, and we are aware that the Salary of any of the employee cannot be "Negative".
We are using "4 Bytes" to store the salary of an employee and we can easily save 2 Bytes over there by removing the "Signed Part" in the integer. This leads us to the use of Data Type Modifiers.

Types of Datatype Modifiers:
- Signed
- Size
- Const

Signed modifier:
All data types are "signed" by default. Signed modifier implies that the data type variable can store positive values as well as negative values. For example, if we need to declare a variable to store temperature, it can be negative as well as positive.

signed int temperature; Or int temperature;

If we need to declare a variable to store the salary of an employee, we will use "Unsigned" Data modifier.

unsigned int salary;

Size Modifier:
Sometimes we need to increase the Storage Capacity of a variable so that it can store values higher than its maximum limit which is there as default.
We need to make use of the **"long"** data type qualifier. **"long"** type modifier doubles the "length" of the data type when used along with it.
For example, if we need to store the "annual turnover" of a company in a variable, we will make us of this type qualifier.

long int turnover;

A **"short"** type modifier does just the opposite of "long" with 2 bytes in memory. If one is not expecting to see high range values in a program and the values are both positive & negative.

For example, if we need to store the "age" of a student in a variable, we will make use of this type qualifier as we are aware that this value is not going to be very high.

short int age;

Const modifier:

In C all variables are by default not constant. Hence, you can modify the value of variable by program. You can convert any variable as a constant variable by using modifier const which is keyword of C language.

Properties of constant variable:

You can assign the value to the constant variables only at the time of declaration.

For example:

const int i=10; float const f=0.0f; unsigned const long double ld=3.14L;

Uninitialized constant variable is not cause of any compilation error. But you cannot assign any value after the declaration.

Example: const int i;

Note: The long, short, signed and unsigned are datatype modifiers that can be used with some primitive data types to change the size or length of the datatype.

Literal:

In C, Literals are the constant values that are assigned to the variables. Literals represent fixed values that cannot be modified.

There are 4 types of literals in C:

- **Integer Literal**
- **Float Literal**
- **Character Literal**
- **String Literal**

Integer literals are used to represent and store the integer values only. Integer literals are expressed in two types i.e,

A) Prefixes: The Prefix of the integer literal indicates the base in which it is to be read.

a. **Decimal-literal(base 10):** A **non-zero decimal digit** followed by zero or more decimal digits(0, 1, 2, 3, 4, 5, 6, 7, 8, 9). Ex: 15, 29 etc…

b. **Octal-literal(base 8):** a **0** followed by zero or more octal digits(0, 1, 2, 3, 4, 5, 6, 7). **Ex:** 056, 0123 etc..

c. **Hex-literal(base 16): 0x** or **0X** followed by one or more hexadecimal digits(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, A, b, B, c, C, d, D, e, E, f, F). **Ex:** 0x5A, 0XAABB etc….

d. **Binary-literal(base 2): 0b** or **0B** followed by one or more binary digits(0, 1). **Ex:** 0b101, 0B11011 etc…

B) Suffixes: The Suffixes of the integer literal indicates the type in which it is to be read. These are represented in many ways according to their data types.

int: No suffix is required because integer constant is by default assigned as an int data type.

unsigned int: character u or U at the end of an integer constant.

long int: character l or L at the end of an integer constant.

unsigned long int: character ul or UL at the end of an integer constant.

long long int: character ll or LL at the end of an integer constant.

unsigned long long int: character ull or ULL at the end of an integer constant.

Real Or Floating-Point Literal:
Integer numbers are inadequate to represent quantities that vary continuously such as distances, heights, temperatures, prices and so on. These quantities are represented by numbers containing fractional parts like17.548. Such numbers are called real or floating-point constants.
Examples: 0.0083, -0.75, 435.36, 2e-8, 0.006e-2
Character Literal: Literal that is used to store a single character within a single quote.
Ex: char ch='A', ch1='9', ch2='+' etc….
String Literal: String literals are like that of character literals, except that they can store multiple characters and use a double quote to store the same. A string constant is a sequence of characters enclosed with in double quotes. The characters may be letters, numbers, special characters and blank space. Example: "Hello", "1994", "Well done", "5+3", "M".
Ex: char ch[]={'A','D','I','T','Y','A'};

Input and Output:
C language has standard libraries that allow input and output in a program. The **stdio.h** or **standard input output library** in C that has methods for input and output.
scanf(): The scanf() method, in C, reads the value from the console as per the type specified and store it in the given address.

Syntax: int scanf("*Control String*",*&arg1,&arg2,…*);

The return value of scanf function is the number of successful data inputs
printf(): The printf() method, in C, prints the value passed as the parameter to it, on the console screen.

Syntax: int printf ("*Control String*", arguments_list);

The return value of printf function is the total number of characters printed is returned. On failure, a negative number is returned.
The format specifier in C is used to tell the compiler about the type of data to be printed or scanned in input and output operations. They always start with a **%** symbol and are used in the formatted string in functions like printf(), scanf etc.

Format Specifier	Description
%c	For character type.
%d	For signed integer type.
%e or %E	For scientific notation of floats.
%f	For float type.
%i	signed integer
%ld or %li	Long
%lf	Double
%Lf	Long double
%lu	Unsigned int or unsigned long
%lli or %lld	Long long
%llu	Unsigned long long
%o	Octal representation
%p	Pointer
%s	String
%u	Unsigned int
%x or %X	Hexadecimal representation
%n	Prints nothing
%%	Prints % character

Examples on printf():

printf("Welcome to C Programming\n"); //**Output:** Welcome to C Programming
int a=10, b=20; printf("%d %d\n",a,b); //**Output:** 10 20
int a=10,b=20,c=a+b; printf("sum of %d and %d is: %d\n",a,b,c);//**Output**:sum of 10 and 20 is: 30

Examples on Scanf():

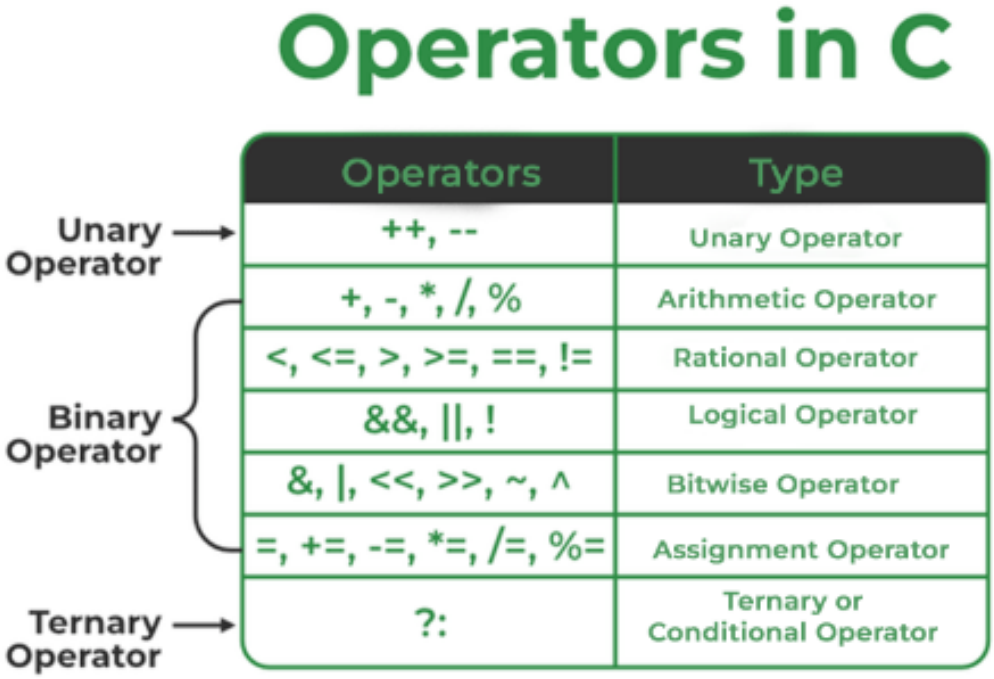
int n; scanf("%d",&n); **//Read an integer value**
int a,b,c; scanf("%d %d %d",&a,&b,&c); **//Read there integer values**
float p,q; scanf("%f %f",&p,&q); **//Read two float values.**
Char ch; scanf("%c",&ch); **//Read a Character**

/* Program to find the Simple Interest*/

```
#include<stdio.h>
int main()
{
        int P,T,R;
        float I;
        printf("Enter Principle, Time and Rate of interest\n");
        scanf("%d%d%d",&P,&T,&R);
        I=(P*T*R)/100;
        printf("P = %d  T = %d R = %d\n",P,T,R);
        printf("Simple Interest = %f",I);
        return 0;
}
```

Operators:

In C language, operators are symbols that represent operations to be performed on one or more operands. C has a wide range of operators to perform various operations.



Types of Operators in C

C language provides a wide range of operators that can be classified into 6 types based on their functionality:

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment Operators
6. Other Operators

1) Arithmetic Operators:

The arithmetic operators are used to perform arithmetic/mathematical operations on operands. Those are +, -, *, / and %.

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

Output:

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

2) Relational Operators:

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

```
// Working of relational operators
#include <stdio.h>
int main()
{
   int a = 5, b = 5, c = 10;
   printf("%d == %d is %d \n", a, b, a == b);
   printf("%d == %d is %d \n", a, c, a == c);
   printf("%d > %d is %d \n", a, b, a > b);
   printf("%d > %d is %d \n", a, c, a > c);
```

```
    printf("%d < %d is %d \n", a, b, a < b);
    printf("%d < %d is %d \n", a, c, a < c);
    printf("%d != %d is %d \n", a, b, a != b);
    printf("%d != %d is %d \n", a, c, a != c);
    printf("%d >= %d is %d \n", a, b, a >= b);
    printf("%d >= %d is %d \n", a, c, a >= c);
    printf("%d <= %d is %d \n", a, b, a <= b);
    printf("%d <= %d is %d \n", a, c, a <= c);
    return 0;
}
```

Output:

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

3) Logical Operators:

Logical Operators are used to combine two or more conditions to make complex expressions. An expression containing logical operator returns either 0 or 1 depending upon whether expression results true or false. Logical operators are commonly used in decision making in C programming.

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

// Working of logical operators

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

```
    printf("!(a != b) is %d \n", result);
    result = !(a == b);
    printf("!(a == b) is %d \n", result);
    return 0;
}
```

Output:

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

4) Bit wise Operators:

The Bitwise operators are used to perform bit-level operations on the operands. The operators are first converted to bit-level and then the calculation is performed on the operands. Mathematical operations such as addition, subtraction, multiplication, etc. can be performed at the bit level for faster processing.

There are 6 bitwise operators in C

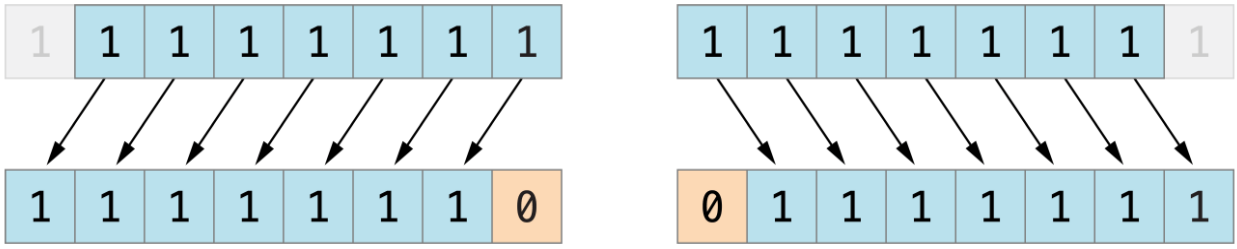
S. No.	Symbol	Operator	Syntax
1	&	Bitwise AND	a & b
2	|	Bitwise OR	a | b
3	^	Bitwise XOR	a ^ b
4	~	Bitwise One's Complement	~a
5	<<	Bitwise Left shift	a << b
6	>>	Bitwise Right shilft	a >> b

Table *Evaluation of bitwise operators on 1 bit values*

Inputs		and	or	xor
a	b	a & b	a | b	a ^ b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Input	not
a	~a
0	1
1	0

In bitwise shift operations, the digits are moved, or shifted, to the left or right.



// C program to illustrate the bitwise operators

```
#include <stdio.h>
int main()
{
   int a = 25, b = 5;
   printf("a & b: %d\n", a & b);
   printf("a | b: %d\n", a | b);
   printf("a ^ b: %d\n", a ^ b);
   printf("~a: %d\n", ~a);
   printf("a >> b: %d\n", a >> b);
   printf("a << b: %d\n", a << b);
   return 0;
}
```

Output:

```
a & b: 1
a | b: 29
a ^ b: 28
~a: -26
a >> b: 0
a << b: 800
```

5) Assignment Operators:

Assignment operators are used to assign value to a variable. The left side operand of the assignment operator is a variable and the right-side operand of the assignment operator is a value. The value on the right side must be of the same data type as the variable on the left side otherwise the compiler will raise an error.

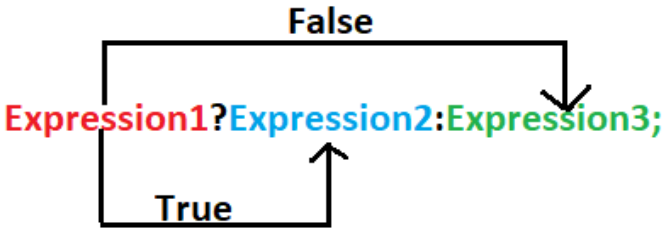
S. No.		Symbol	Operator	Syntax	Equalent to
1		=	Simple Assignment	a = b	a = b
2	Short hand Operators	+=	Plus and assign	a += b	a=a+b
3		-=	Minus and assign	a -= b	a=a-b
4		*=	Multiply and assign	a *= b	a=a*b
5		/=	Divide and assign	a /= b	a=a/b
6		%=	Modulus and assign	a %= b	a=a%b
7		&=	AND and assign	a &= b	a=a&b
8		|=	OR and assign	a |= b	a=a|b
9		^=	XOR and assign	a ^= b	a=a^b
10		>>=	Rightshift and assign	a >>= b	a=a>>b
11		<<=	Leftshift and assign	a <<= b	a=a<<b

6) Other Operators:

a) Conditional Opertor(?:)

The conditional operator is the only ternary operator in C.

Syntax: Expression1 ? Expression2 : Expression3



Here, Expression1 is the condition to be evaluated first. If the condition(Expression1) is *True* then we will execute and return the result of Expression2 otherwise if the condition(Expression1) is *false* then we will execute and return the result of Expression3.
Note: We may replace the use of if..else statements with conditional operators.

Example:

```
#include<stdio.h>
int main()
{
    int x,y;
    scanf("%d%d",&x,&y);
    int max=x>y ? x : y;
    printf("Maximum of %d and %d is: %d",x,y,max);
    return 0;
}
```

Input:

20 36

Output:

Maximum of 20 and 36 is: 36

b) **sizeof():** Basically, the sizeof the operator is used to compute the size of the variable or datatype. The result of sizeof is of the unsigned integral type which is usually denoted by size_t.

Example:

```
#include <stdio.h>
int main()
{
  int a;
   float b;
   double c;
   char d;
   printf("Size of int=%lu bytes\n",sizeof(a));
   printf("Size of float=%lu bytes\n",sizeof(b));
   printf("Size of double=%lu bytes\n",sizeof(c));
   printf("Size of char=%lu byte\n",sizeof(d));
   return 0;
}
```

Output:

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

c) Comma(,), dot(.), addressof(&), dereference(*), arrow(->) etc….

Operator Precedence and Associativity:

Precedence: The precedence of operators determines which operator is executed first if there is more than one operator in an expression.

For Example:

10+20*30 => 610 [Right]
10+20*30 => 900 [Wrong]

Note: Precedence tells which operator is to be executed first in an expression.

Associativity: It defines the order in which operators of the same precedence are evaluated in an expression. Associativity can be either from left to right or right to left.

EXAMPLE:	Example:
Let a=30, b=10, c=11, d=5, e=10 With the above values evaluate the following expression A%3 – b/2 + (c*d-5)/e SOLUTION: A%3 – b/2 + (c*d-5)/e Substitute the values in the expression As per hierarchy rules the sub-expression with in parenthesis will be first evaluated. 30 % 3 – 10/2 + (11*5-5)/10 =30 %3 – 10/2 + (55-5)/10 (operation with in parenthesis) =30 %3 – 10/2 + 50/10 (operation with in parenthesis) =0 -10/2 + 50/10 (% operation) =0 – 5 + 50/10 (/ operation) =0 – 5 + 5 (/operation) =-5 + 5 (- operation) =0 (+operation)	(3 * 4) / 2 + 4 / 2. Solution: (3 * 4) / 2 + 4 / 2. =12/2 + 4/2(operation with in parentheses) = 6 + 4/2 (/ operation) = 6 + 2 (/ operation) = 8 (+ operation)

OPERATOR	TYPE	ASSOCIAVITY
( ) [ ] . ->		left-to-right
++ -- + - ! ~ (*type*) * & sizeof	Unary Operator	right-to-left
* / %	Arithmetic Operator	left-to-right
+ -	Arithmetic Operator	left-to-right
<< >>	Shift Operator	left-to-right
< <= > >=	Relational Operator	left-to-right
== !=	Relational Operator	left-to-right
&	Bitwise AND Operator	left-to-right
^	Bitwise EX-OR Operator	left-to-right
|	Bitwise OR Operator	left-to-right
&&	Logical AND Operator	left-to-right
||	Logical OR Operator	left-to-right
? :	Ternary Conditional Operator	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Operator	right-to-left
,	Comma	left-to-right

Type Conversion, and Casting:

Conversion of the value of one data type to another type is known as **type conversion**.

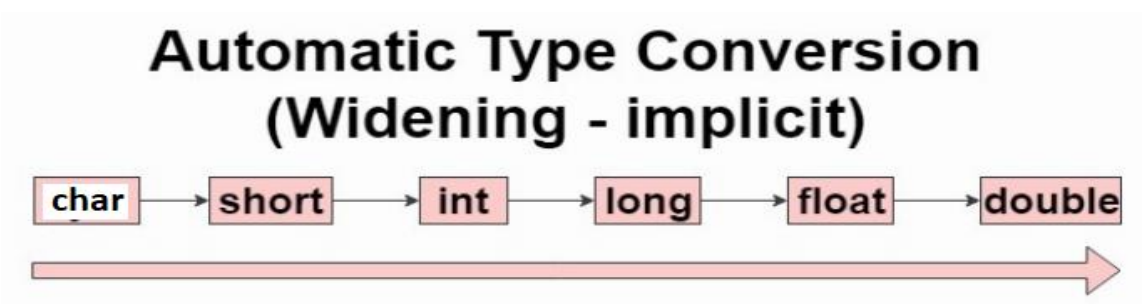
There are two types of conversion in C:

- **Implicit Conversion** (automatically)
- **Explicit Conversion** (manually)

1) Implicit Conversion:

Implicit Type Conversion is also known as **'automatic type conversion'**. It is done by the compiler on its own, without any external trigger from the user.

In implicit typecasting, the conversion involves a **smaller data type to the larger data type.**

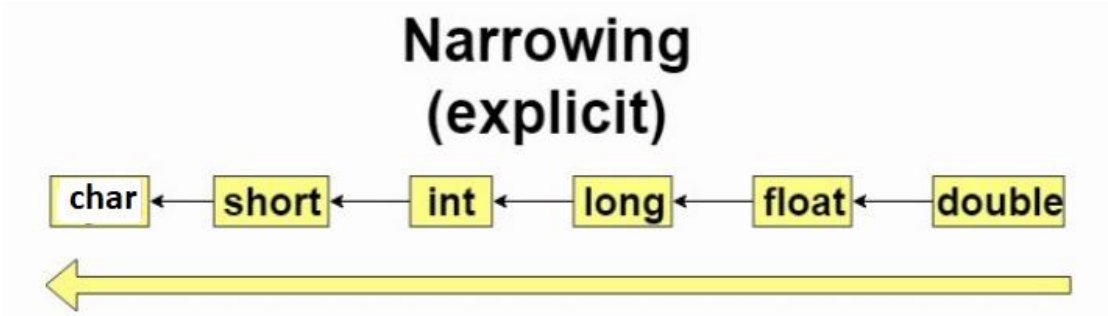


```
#include<stdio.h>
int main()
{
        int x=10;
        char y='a';
        float z;
        x=x+y;              //y implicitly converted to int. ASCII value of 'a' is 97
        z=x+1.0;
        printf("x=%d,z=%f",x,z);
        return 0;
}
```

Output:
x=107, z=108.000000

2) Explicit Conversion

This conversion is done by user. This is also known **as typecasting**. In this one data type is converted into another data type forcefully by the user. In explicit typecasting, the conversion involves a larger data type to the smaller data type.



Here is the syntax of explicit type casting in C language,

Syntax:

(datatype) Variable/expression

Ex:

float salary = 10000.00;

int sal= **(int)** salary;

Example:

```
#include<stdio.h>
int main()
{
        float x=1.2;
        int sum;
        sum=(int)x+1;
        printf("sum=%d",sum);
        return 0;
}
```

Output:

Sum=2

Important Questions:

1. Differentiate Hight Level, Middle Level and Low-Level Programming Languages.
2. Define an algorithm. List the characteristics of an algorithm.
3. List and explain the programming strategies or approaches.
4. What is a Time and Space complexity of an algorithm.
5. Differentiate Compiler, Assembler and interpreter.
6. What is the structure of C program and explain in detail of each section.
7. Describe the following
   a) C Character set b) Token c) Identifier d) Variable e) Constant
8. What is data type and explain different data types supported by C.
9. Define an operator. List and describe different types of operators in C programming with an example program.
10. Explain Operator Precedence and Associativity with an example program.
11. Demonstrate the use of printf() and scanf() with an example programs.
12. What is type casting? Demonstrate with an example program.

Programs to Practice:

1. Program to find the simple interest.
2. Program to find the conversion from Fahrenheit to Celsius.
3. Program to find the distance between two points.
4. Program to find the square root of a given number.
5. Program to find the area of Triangle using Heron's formula.
6. Program to find the BMI value.

Practice the flow chart design for above programs also using dia tool.

MCQ Question and Answers for Practice

1. Which of the following is NOT a characteristic of a good algorithm?

a) Finiteness **b) Ambiguity** c) Input d) Output

2. What does it mean for an algorithm to be finite?

a) It produces an incorrect output. b) It runs indefinitely without stopping.

c) It completes after a certain number of steps. d) It doesn't require any input.

3. Which of the following characteristics defines the performance efficiency of an algorithm?

a) Time complexity b) Uniqueness c) Input size d) Clarity

4. What is meant by the 'definiteness' characteristic of an algorithm?

a) The algorithm must be written in English

b) Each step of the algorithm must be clear and unambiguous

c) The algorithm should handle multiple tasks at once d) It must run forever

5. Which of the following is an example of a high-level programming language?

a) Assembly **b) Python** c) Machine code d) Binary

6. Which of the following best describes a low-level language?

a) Provides close control over hardware b) Easy to read and write

c) Independent of hardware architecture d) Automatically manages memory

7. What is a key feature of a middle-level programming language?

a) It is highly abstracted from hardware b) It only works on specific hardware architectures

c) It combines features of both low-level and high-level languages

d) It does not require compilation

8. Which of these languages is considered a middle-level language?

a) Assembly b) Python **c) C** d) JavaScript

9. What symbol is used to represent the start or end of a flowchart?

a) Rectangle b) Diamond **c) Oval** d) Parallelogram

10. Which symbol is used to represent a decision point in a flowchart?

a) Rectangle **b) Diamond** c) Oval d) Circle

11. In a flowchart, what does the rectangle symbol represent?

a) Input/output b) Start/End **c) Process or action step** d) Decision

12. What is the purpose of a flowchart?

a) To write code for a program **b) To visually represent the flow of a process or algorithm**

c) To store data for a program d) To manage memory in a system

13. Which of the following symbols represents input/output in a flowchart?

a) Rectangle **b) Parallelogram** c) Diamond d) Oval

14. Which of the following is not a valid C data type?

a) int b) float c) char **d) string**

15. What is the size of an int data type in C on a 32-bit system?

a) 2 bytes **b) 4 bytes** c) 8 bytes d) 1 byte

16. Which of the following data types can store a value of 3.14159?

a) int b) char **c) float** d) unsigned int

17. Which format specifier is used to print a char value in C?

a) %d b) %f **c) %c** d) %s

18. What is the range of values that can be stored in an unsigned char?

a) -128 to 127 **b) 0 to 255** c) 0 to 128 d) -255 to 255

19. What is the default data type of a floating-point number in C?

a) double b) float c) long double d) int

20. Which of the following is the correct format specifier for a double in C?

a) %d **b) %lf** c) %c d) %u

21. What is the size of a char data type in C?

a) 1 byte b) 2 bytes c) 4 bytes d) 8 bytes

22. Which data type would you use to store large integers (greater than int)?

a) float b) double **c) long int** d) char

23. Which of the following is not a C token?

a) Keyword b) Identifier c) Constant **d) Variable**

24. How many types of C tokens are there?

a) 3 b) 5 **c) 6** d) 8

25. Which of the following is a valid identifier in C?

a) 1variable b) $sum **c) total_sum** d) float

26. Which symbol is used to terminate a statement in C?

a) . b) , c) : **d) ;**

27. Which of the following is a valid C operator?

a) & b) @ c) # d) %&

28. Which token is used for preprocessor directives in C?

a) $ **b) #** c) @ d) %

29. Which of the following is a string constant in C?

a) 'C' **b) "C Programming"** c) 100 d) True

30. What is typecasting in C?

a) Changing the variable's name **b) Converting one data type to another**

c) Assigning a variable to a constant d) Comparing two variables

31. Which of the following is an example of explicit typecasting?

a) float a = 10.5; b) int b = 3.14; **c) int c = (int)4.5;** d) double d = 5;

32. Check the output for the following code: int a = 10;

float b = (float)a / 4;

printf("%f", b);

a) 2.0 **b) 2.5** c) 2.75 d) 2.0f

33. Which of the following conversions is NOT allowed in C?

a) int to float b) float to int c) char to int **d) string to int**

34. What happens during implicit type conversion?

a) The programmer must specify the conversion b) Data is lost in the conversion process

c) The compiler automatically converts one data type to another d) It causes a compilation error

35. Which of the following is a valid variable declaration in C?

a) int 1stNumber; **b) float number;** c) char $name; d) double my variable;

36. What is the default value of an uninitialized static variable in C?

a) 0 b) Undefined c) Garbage value d) NULL

37. Which of the following is a constant in C?

a) #define PI 3.14 b) int a = 5; c) float b = 10.0; d) char name = 'A';

38. What will happen if you try to change the value of a constant defined using const?

a) Compilation will succeed b) The program will terminate

c) A compilation error will occur d) The constant will be changed

39. Which of the following statements about variables in C is FALSE?

a) Variables must be declared before they are used.

b) Variable names can contain digits, but must not start with one.

c) Variables can be assigned new values at any point in the program.

d) Variable names can be the same as C keywords.