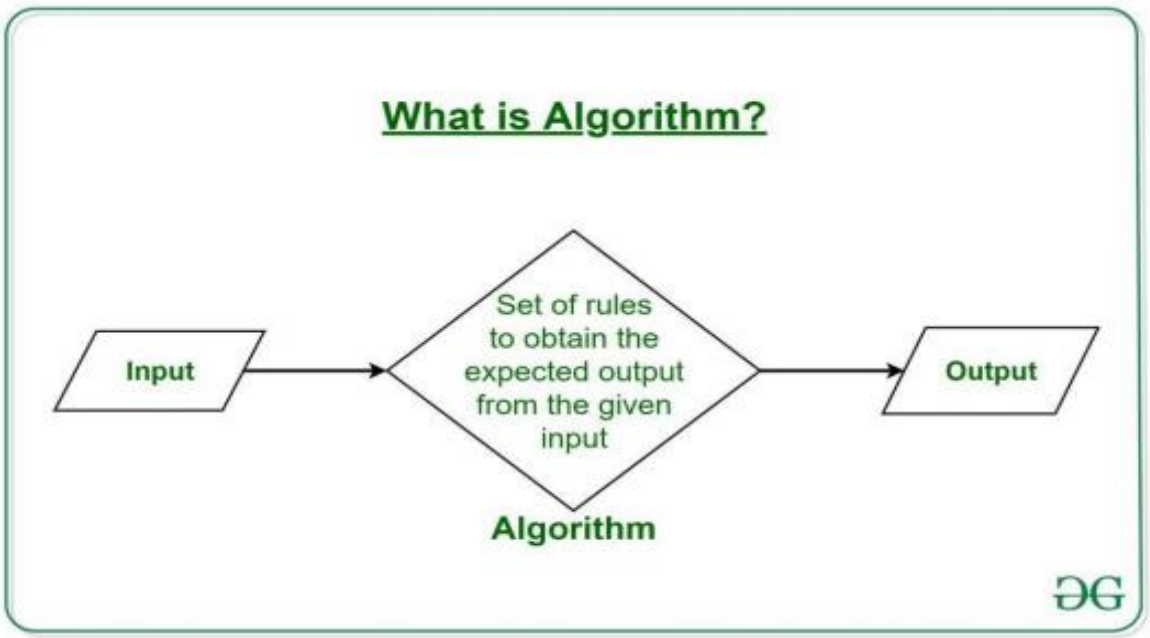
2131 – SCHEME 2021

PROBLEM SOLVING AND PROGRAMMING IN C

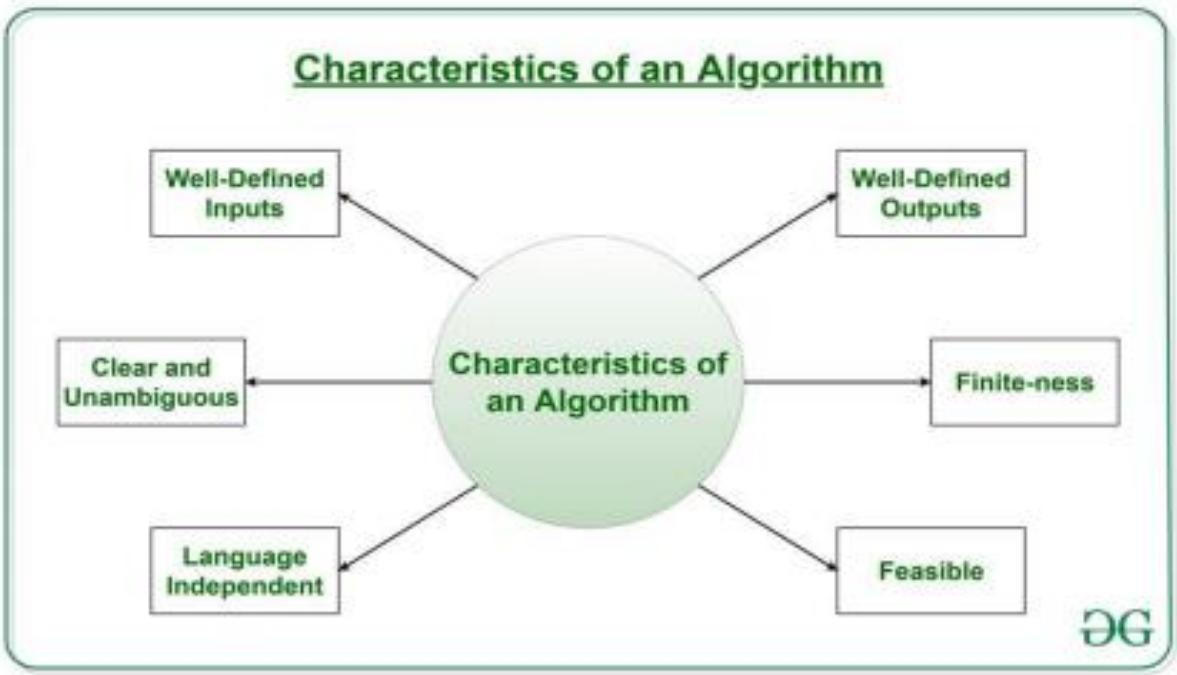
MODULE – I

Algorithms

The word Algorithm means "a process or set of rules to be followed in calculations or other problem-solving operations". Therefore, Algorithm refers to a set of rules/instructions that step-by-step define how a work is to be executed upon in order to get the expected results.



Characteristics of an Algorithm



- **Clear and Unambiguous**: Algorithm should be clear and unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning.
- **Well-Defined Inputs**: If an algorithm says to take inputs, it should be well-defined inputs.
- **Well-Defined Outputs:** The algorithm must clearly define what output will be yielded and it should be well-defined as well.
- **Finite-ness:** The algorithm must be finite, i.e., it should not end up in an infinite loop or similar.
- **Feasible:** The algorithm must be simple, generic and practical, such that it can be executed upon with the available resources. It must not contain some future technology, or anything.
- **Language Independent:** The Algorithm designed must be language-independent, i.e., it must be just plain instructions that can be implemented in any language.

Advantages of Algorithms:

- It is easy to understand.
- Algorithm is a step-wise representation of a solution to a given problem.
- In Algorithm the problem is broken down into smaller pieces or steps hence, it is easier for the programmer to convert it into an actual program.

How to Design an Algorithm?

In order to write an algorithm, following things are needed as a pre-requisite:

1. The **problem** that is to be solved by this algorithm.
2. The **constraints** of the problem that must be considered while solving the problem.
3. The **input** to be taken to solve the problem.
4. The **output** to be expected when the problem the is solved.
5. The **solution** to this problem, in the given constraints.

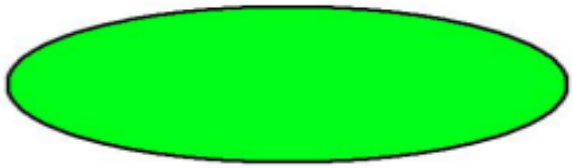
Then the algorithm is written with the help of above parameters such that it solves the problem.

Flowcharts

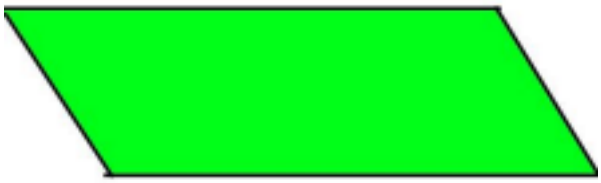
Flowchart is a graphical representation of an algorithm. Programmers often use it as a program-planning tool to solve a problem. It makes use of symbols which are connected among them to indicate the flow of information and processing.

Basic Symbols used in Flowchart Designs

Terminal: The oval symbol indicates Start, Stop and Halt in a program's logic flow.

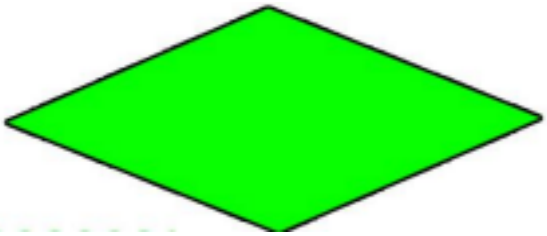


Input/Output: A parallelogram denotes any function of input/output type. Program instructions that take input from input devices and display output on output devices are indicated with parallelogram in a flowchart.

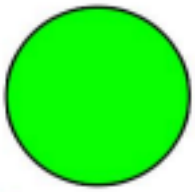


Processing: A box represents arithmetic instructions. All arithmetic processes such as adding, subtracting, multiplication and division are indicated by action or process symbol.

Decision Diamond symbol represents a decision point. Decision based operations such as yes/no question or true/false are indicated by diamond in flowchart.



Connectors: Whenever flowchart becomes complex or it spreads over more than one page, it is useful to use connectors to avoid any confusions. It is represented by a circle.



Flow lines: Flow lines indicate the exact sequence in which instructions are executed. Arrows represent the direction of flow of control and relationship among different symbols of flowchart.

Advantages of Flowchart:

- Flowcharts are better way of communicating the logic of system.
- Flowcharts act as a guide for blueprint during program designed.
- Flowcharts helps in debugging process.
- With the help of flowcharts programs can be easily analysed.
- It provides better documentation.
- Flowcharts serve as a good proper documentation.

Problems:

1. Write an Algorithm and Draw Flowchart for print a message.
2. Read 2 Numbers.
3. Add 2 Numbers.
4. Compare 2 Numbers.
5. Largest value from 3 Numbers.
6. Arithmetic Operations of 2 Numbers.
7. Odd or Even.
8. Sum of first N Numbers.
9. Sum of N Numbers.
10. Largest Value from N Numbers.

Program Development Cycles

Programming is the process of creating a set of instructions that tell a computer how to perform a task. Programming can be done using a variety of computer "languages," such as C,C++,JAVA etc.

Syntax refers to the spelling and grammar of a programming language. Computers are inflexible machines that understand what you type only if you type it in the exact form that the computer expects. The expected form is called the syntax.

Program development is the process of creating application programs. Program development life cycle (PDLC) The process containing the five phases of program development: analysing, designing, coding, debugging and testing, and implementing and maintaining application software.

The following are six steps in the Program Development Life Cycle:

1. **Analyze the problem.** The computer user must figure out the problem, then decide how to resolve the problem.

2. **Design the program.** A flow chart is important to use during this step of the PDLC. This is a visual diagram of the flow containing the program.

3. **Code the program.** This is using the language of programming to write the lines of code. The code is called the source code.

4. **Debug the program.** The bugs are important to find because this is known as errors in a program.
5. **Formalize the solution.** One must run the program to make sure there are no syntax and logic errors. ie testing the program.

6. **Document and maintain the program**. This step is the final step of gathering everything together. Internal documentation is involved in this step because it explains the reasoning one might have made a change in the program or how to write a program.

Programming Languages:

Programming Languages are mainly classified into 2.

High Level Language

Low Level Language

High-level language

- It can be easily interpreted as well as compiled in comparison to low-level language. • It can be considered as a programmer-friendly language.
- It is easy to understand.
- It is easy to debug.
- It is simple in terms of maintenance.
- It requires a compiler/interpreter to be translated into machine code. • It can be run on different platforms.
- It can be ported from one location to another.
- It is less memory efficient, i.e it consumes more memory in comparison to low-level languages.
- Examples of high-level languages include C, C++, Java, Python.

Low-level language

- It is also known as machine level language.
- It can be understood easily by the machine.
- It is considered as a machine-friendly language.
- It is difficult to understand.
- It is difficult to debug.
- Its maintenance is also complex.
- It is not portable.
- It depends on the machine; hence it can't be run on different platforms. • It requires an assembler that would translate instructions.

Translators :

A translator is a programming language processor that modifies a computer program from one language to another. It takes a program written in the source program and modifies it into a machine program.

There are various types of a translators.

Compiler − A compiler is a program that translates a high-level language (for example, C, C++, and Java) into a low-level language (object program or machine program).

Assembler − An assembler is a translator which translates an assembly language program into an equivalent machine language program of the computer.

Interpreter − An interpreter is a program that executes the programming code directly rather than only translating it into another format. It translates and executes programming language statements one by one.

Compiling and Executing a Program :

Step 1 : Open an EDITOR tool. Eg:- Type "**nano pgm.c**" in UBUNTUTerminal.
Step 2 : Type the C Program
Step 3 : Save the Program. Eg :- Press **Control** Key + **O** Key

Step 4 : Exit from NANO EDITOR. Eg :- Press **Control** Key + **X** Key

Step 5 : Compile the Program. Eg :- **CC** or **GCC pgm.c**

Step 6 : Execute or Run the Program. Eg :- **. /a.out**

Example

nano sample.c

#include<stdio.h>

main()

{

printf(“WELCOME”);

}

Save – CONTROL+O

Exit – CONTROL+X

Compile – GCC sample.c

Execute - ./a.out

C Program Structure

A C program involves the following sections:

- Documentations (Documentation Section)
- Preprocessor Statements (Link Section)
- Global Declarations (Definition Section)
- The main() function
  - Local Declarations
  - Program Statements & Expressions
- User Defined Functions

/* Writes the words "Hello, World!" on the screen */

```
#include<stdio.h>
 main()
{
 printf("Hello, World!\n");

}
```

/* Comments */	Comments are a way of explaining what makes a program. The compiler ignores comments and used by others to understand the code.
#include<stdio.h>	stdio is standard for input/output, this allows us to use some commands which includes a file called stdio.h.
main()	main() is the main function where program execution begins. Every C program must contain only one main function.
Braces	Two curly brackets "{...}" are used to group all statements.

The Documentation section usually contains the collection of comment lines giving the name of the program, author's or programmer's name and few other details. The second part is the link-section which instructs the compiler to connect to the various functions from the system library. The Definition section describes all the symbolic-constants. The global declaration section is used to define those variables that are used globally within the entire program and is used in more than one function. This section also declares all the user-defined functions. Then comes the main(). All C programs must have a main() which contains two parts:

- Declaration part
- Execution part

The declaration part is used to declare all variables that will be used within the program. There needs to be at least one statement in the executable part, and these two parts are declared within the opening and closing curly braces of the main(). The execution of the program begins at the opening brace '{' and ends with the closing brace '}'. Also, it has to be noted that all the statements of these two parts need to be terminated with a semi-colon. The sub-program section deals with all user-defined functions that are called from the main(). These user-defined functions are declared and usually defined after the main() function.

C Identifiers

C identifiers represent the name in the C program, for example, variables. An identifier can be composed of letters such as uppercase, lowercase letters, underscore, digits, but the starting letter should be either an alphabet or an underscore.

Rules for constructing C identifiers

- The first character of an identifier should be either an alphabet or an underscore, and then it can be followed by any of the character, digit, or underscore.
- It should not begin with any numerical digit.
- In identifiers, both uppercase and lowercase letters are distinct. Therefore, we can say that identifiers are case sensitive.
- Commas or blank spaces cannot be specified within an identifier.
- Keywords cannot be represented as an identifier.
- The length of the identifiers should not be more than 31 characters.
- Identifiers should be written in such a way that it is meaningful, short, and easy to read.

C Keywords

Keywords are predefined, reserved words used in programming that have special meanings to the compiler. Keywords are part of the syntax and they cannot be used as an identifier.

As C is a case sensitive language, all keywords must be written in lowercase. Here is a list of all keywords allowed in ANSI C.

C Keywords

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

Data Types in C

Each variable in C has an associated data type.

Following are the examples of some very common data types used in C:

- **char:** The most basic data type in C. It stores a single character and requires a single byte of memory in almost all compilers.
- **int:** As the name suggests, an int variable is used to store an integer.
- **float:** It is used to store decimal numbers (numbers with floating point value) with single precision.
- **double:** It is used to store decimal numbers (numbers with floating point value) with double precision.

Data Type	Memory (bytes)	Range	Format Specifier
short int	2	-32,768to32,767	%hd
unsigned short int	2	0 to 65,535	%hu
unsignedint	4	0to4,294,967,295	%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 to18,446,744,073,709,551,615	%llu
signed char	1	-128 to 127	%c
unsigned char	1	0 to 255	%c
float	4		%f
double	8		%lf
long double	16		%Lf

Constants

Constant is a value that cannot be changed during program execution; it is fixed.

In C language, a number or character or string of characters is called a constant. And it can be any data type. Constants are also called as literals.

There are two types of constants −

Primary constants − Integer, float, and character are called as Primary constants.

Secondary constants − Array, structures, pointers, Enum, etc., called as secondary constants. Example for Primary constants

```c
#include<stdio.h>
int main(){
   const int height=20;
   const int base=40;
   float area;
   area=0.5 * height*base;
   printf("The area of triangle :%f", area);
   return 0;
}
```

Output

```
The area of triangle :400.000000
```

Example for secondary constants

```c
include<stdio.h>
void main(){
   int a;
   int *p;
   a=10;
   p=&a;
   printf("a=%d\n",a);//10//
   printf("p=%d\n",p);//address value of p//
   *p=12;
   printf("a=%d\n",a);//12//
   printf("p=%d\n",p);//address value of p//
}
```

Output

```
a=10
p=6422036
a=12
p=6422036
```

C - Variables

A variable is nothing but a name given to a storage area that our programs can manipulate. The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C is case-sensitive.

A variable definition tells the compiler where and how much storage to create for the variable. A variable definition specifies a data type and contains a list of one or more variables of that type as follows −

```
type variable_list;
```

Here, **type** must be a valid C data type including char, w_char, int, float, double, bool, or any user-defined object; and **variable_list** may consist of one or more identifier names separated by commas. Some valid declarations are shown here −

```
int i, j, k;
char c, ch;
float f, salary;
double d;
```

Variable Declaration in C

A variable declaration provides assurance to the compiler that there exists a variable with the given type and name so that the compiler can proceed for further compilation without requiring the complete detail about the variable.

```c
#include <stdio.h>

int main () {

 /* variable declaration: */
 int a, b;
 int c;
 float f;

 /* actual initialization */
 a = 10;
 b = 20;

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

 f = 70.0/3.0;
 printf("value of f : %f \n", f);

 return 0;
}
```

Output

```
value of c : 30
value of f : 23.333334
```

C Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical functions. C language is rich in built-in operators and provides the following types of operators

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Conditional Operators
- Increment or Decrement Operators

Arithmetic Operators

The following table shows all the arithmetic operators supported by the C language. Assume variable **A** holds 10 and variable **B** holds 20 then −

Operator	Description	Example
+	Adds two operands.	A + B = 30
−	Subtracts second operand from the first.	A − B = -10
*	Multiplies both operands.	A * B = 200
/	Divides numerator by de-numerator.	B / A = 2
%	Modulus Operator and remainder of after an integer division.	B % A = 0
++	Increment operator increases the integer value by one.	A++ = 11
--	Decrement operator decreases the integer value by one.	A-- = 9

Relational Operators

The following table shows all the relational operators supported by C. Assume variable **A** holds 10 and variable **B** holds 20 then −

Operator	Description	Example
==	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.	(A != B) is true.

>	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.	(A <= B) is true.

Logical Operators

Following table shows all the logical operators supported by C language. Assume variable **A** holds 1 and variable **B** holds 0, then –

Operator	**Description**	**Example**
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	!(A && B) is true.

Bitwise Operators

Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for &, |, and ^ is as follows −

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume variable 'A' holds 60 and variable 'B' holds 13, then −

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, i.e., 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) = 61, i.e., 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, i.e., 0011 0001
~	Binary One's Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = ~(60), i.e,. -0111101

<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240 i.e., 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15 i.e., 0000 1111

Assignment Operators

The following table lists the assignment operators supported by the C language −

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand	C = A + B will assign the value of A + B to C
+=	Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.	C -= A is equivalent to C = C – A
*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.	C %= A is equivalent to C = C % A

<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

Conditional Operator

? :	Conditional Expression.	If Condition is true ? then value X : otherwise value Y

Operator Precedence and Associativity in C

Operator precedence determines which operator is performed first in an expression with more than one operators with different precedence.
For example: Solve

10 + 20 * 30

10 + 20 * 30 is calculated as **10 + (20 * 30)**

Operators Associativity is used when two operators of same precedence appear in an expression. Associativity can be either **L**eft **t**o **R**ight or **R**ight **t**o **L**eft.
For example: '*' and '/' have same precedence and their associativity is **L**eft **t**o **R**ight, so the expression "100 / 10 * 10" is treated as "(100 / 10) * 10".

For example: Solve

100 + 200 / 10 - 3 * 10

Category	Operator	Associativity

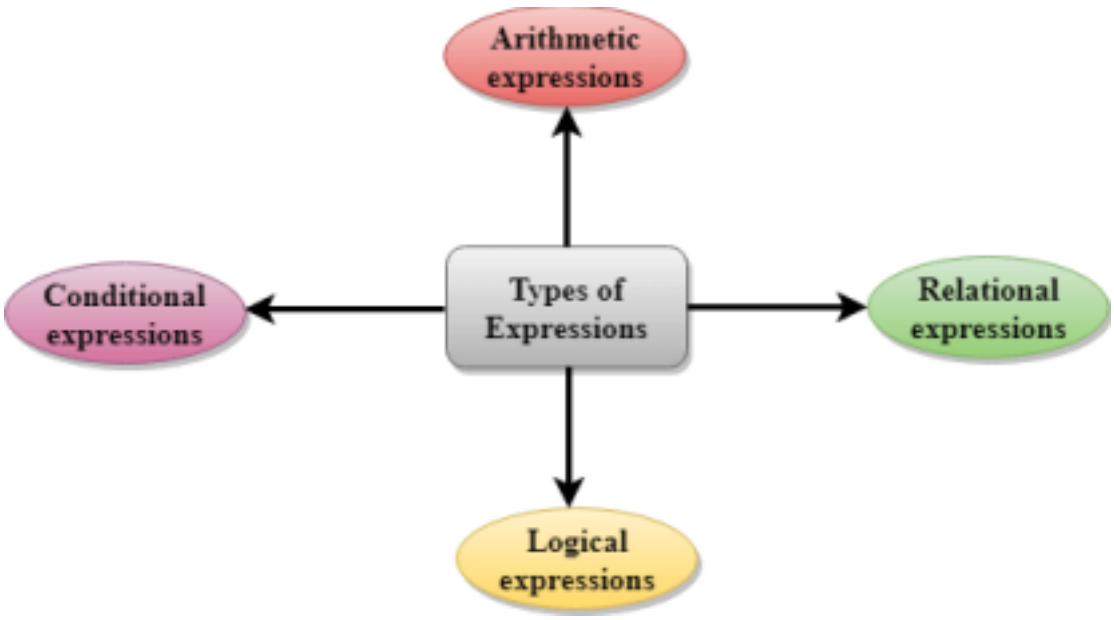
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right

Category	**Operator**	**Associativity**
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

C Expressions

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



Evaluation of Arithmetic Expressions

The expressions are evaluated by performing one operation at a time. The precedence and associativity of operators decide the order of the evaluation of individual operations.

Example.

6*2/ (2+1 * 2/3 + 6) + 8 * (8/4)

Evaluation of expression **Description of each operation**

6*2/(2+1 * 2/3 +6) +8 * (8/4)	An expression is given.
6*2/(2+2/3 + 6) + 8 * (8/4)	2 is multiplied by 1, giving value 2.
6*2/(2+0+6) + 8 * (8/4)	2 is divided by 3, giving value 0.
6*2/ 8+ 8 * (8/4)	2 is added to 6, giving value 8.
6*2/8 + 8 * 2	8 is divided by 4, giving value 2.
12/8 +8 * 2	6 is multiplied by 2, giving value 12.
1 + 8 * 2	12 is divided by 8, giving value 1.
1 + 16	8 is multiplied by 2, giving value 16.
17	1 is added to 16, giving value 17.

C - Type Casting

Converting one datatype into another is known as type casting or, type-conversion.

You can convert the values from one type to another explicitly using the **cast operator** as follows –

(type_name) expression

```
#include <stdio.h>

main() {

 int sum = 17, count = 5;
 float mean;

 mean = (float) sum / count;
 printf("Value of mean : %f\n", mean );
}
```

The cast operator has precedence over division, so the value of sum is first converted to type float and finally it gets divided by count yielding a float value.

C - Input and Output

Input, it means to feed some data into a program. An input can be given in the form of a file or from the command line. C programming provides a set of built-in functions to read the given input.

Output, it means to display some data on screen, printer, or in any file. C programming

provides a set of built-in functions to output the data on the computer screen.
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.
Syntax:

scanf("%X", &variableOfXType);

where **%X** is the format specifier in C. It is a way to tell the compiler what type of data is in a variable and **&** is the address operator in C, which tells the compiler to change the real value of this varia ble, stored at this address in the memory.

printf()

The printf() method, in C, prints the value passed as the parameter to it, on the console screen.

Syntax:

printf("%X", variableOfXType);
where **%X** is the format specifier in C. It is a way to tell the compiler what type of data is in a variable and **&** is the address operator in C, which tells the compiler to change the real value of this varia ble, stored at this address in the memory.

The Syntax for input and output for these are:

- **Integer:**
  Input: scanf("%d", &intVariable);
  Output: printf("%d", intVariable);

- **Float:**
  Input: scanf("%f", &floatVariable);
  Output: printf("%f", floatVariable);

- **Character:**
  Input: scanf("%c", &charVariable);
  Output: printf("%c", charVariable);

Basic Programs

1. Display a message.
2. a=10, b=20 Find a+b;
3. Print 100 five times.
4. a=10, b=4 Find the Quotient and Remainder of a/b.
5. Read a and b then find (a+b)/2.
6. Read a and b then find $(a+b)^2$.
7. Read a and b then perform arithmetic operations such as addition, subtraction, multiplication and division.