Q1. Define Algorithm and Characteristics of a Good Algorithm.

Ans: **Algorithm** : An algorithm is a finite sequence of well-defined and unambiguous steps or instructions to solve a problem and to get the desired output.

Characteristics of a Good Algorithm :

① **Input** : An algorithm must have zero or more input(s) which are clearly specified.

② **Output** : An algorithm must have at least one output which is clearly defined.

③ **Definiteness** : Each step of an algorithm must be clear, precise and unambiguous.

④ **Finiteness** : An algorithm must terminate after a finite number of steps.

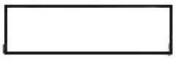
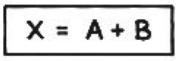
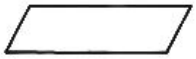
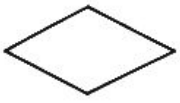
⑤ **Effectiveness** : Each step must be basic and feasible to perform in finite time.

⑥ **Generality** : An algorithm should be applicable to all problems of the same type.

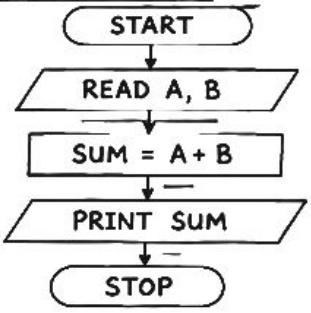
Q2. Explain Flowchart Symbols with Examples.

Ans: **Flowchart** : A flowchart is a graphical representation of an algorithm or process using standard symbols connected by arrows to show the sequence of steps.

Common Flowchart Symbols :

Symbol	Name	Meaning / Example
	Terminator (Start / Stop)	Indicates the beginning or end of a program. 
	Process	Indicates a processing or calculation step. 
	Input / Output	Indicates input or output operation. 
	Decision	Indicates a condition test (True / False). 
	Flow Line	Shows the direction of flow of control.
	Connector	Used to join flow lines from different parts. 

Example Flowchart :



Explanation :

- START → Program begins.
- READ A, B → Input two numbers.
- SUM = A + B → Add the numbers.
- PRINT SUM → Output the result.
- STOP → Program ends.

Q3. History and Importance of C Language.

Ans: **History of C Language :**

- C language was developed in 1972 at AT & T Bell Laboratories by Dennis Ritchie.
- It was developed to overcome the limitations of earlier languages like B, BCPL and Assembly language.
- The language "C" is a successor of B language which was developed by Ken Thompson.
- C was originally used for developing system software like operating systems (UNIX).
- Today, C is one of the most widely used and influential programming languages.

Importance of C Language :

① C is a simple, structured and mid-level programming language.
② It is machine independent but provides low-level features.
③ It is portable, i.e., programs written in C can run on different platforms with little or no modification.
④ It is very fast and efficient.
⑤ It can be used to develop system software as well as application software.
⑥ It forms the base for other languages like C++, Java, Python, etc.
⑦ It has a rich set of operators, data types and control structures.

Q4. Explain Character Set and C Tokens.

Ans: **Character Set in C :**

- The character set in C consists of all the symbols that can be used in a C program.
- It includes the following :
  - ① Alphabets : A to Z, a to z
  - ② Digits : 0 to 9
  - ③ Special Symbols : + - * / % = < > ! & | ^ ~ etc.
  - ④ White Space Characters : space, tab (\t), new line (\n)
  - ⑤ Other Characters : ( ) { } [ ] ; , . : ' " \ # etc.

C Tokens :

- A token is the smallest individual unit in a C program.
- The C tokens are of the following types :

① Keywords : int, float, if, else, while, return, etc.
② Identifiers : Names given by the user such sum, value, main, etc.
③ Constants : Fixed values such as 10, 3.14, 'A', "Hello", etc.
④ String Literals : Series of characters enclosed in double quotes, e.g., "Hello".
⑤ Operators : Symbols used to perform operations, e.g., + - * / = etc.
⑥ Punctuators : Symbols used to separate or group other tokens, e.g., ; , ( ) { } [ ].

Q5. Keywords, Identifiers, Constants and Variables.

Ans: ① **Keywords** : Keywords are predefined (reserved) words in C language which have special meaning and cannot be used as names for variables, functions or any other user defined items.
Examples : int, float, if, else, while, return, for, break, continue, char, void, etc.

② **Identifiers** : Identifiers are names given by the user to variables, functions, arrays, structures etc.
Rules for Identifiers :
→ It can contain letters (A–Z, a–z), digits (0–9) and underscore (_).
→ The first character must be a letter or underscore.
→ It cannot be a keyword.
→ It is case sensitive.
→ No special symbols are allowed except underscore.
Examples : sum, total_marks, value1, _count, temp2

③ **Constants** : Constants are fixed values which do not change during the execution of a program.
Types of Constants :
- Integer Constants : 10, −25, 0, 100 etc.
- Real (Floating) Constants : 3.14, −0.25, 6.02e23 etc.
- Character Constants : 'A', 'b', '7' etc.
- String Constants : "Hello", "C Language" etc.

④ **Variables** : Variables are named memory locations used to store data that can change during the execution of a program.
Example : `int age = 20;  float price = 99.50;`
`char grade = 'A';`

Q6. Data Types and Storage Classes in C.

Ans: • Data Types in C :
C provides different data types to store different kinds of data. They are classified as follows :

	Type	Description	Examples	Size (bytes)
1. Basic (Primary) Types	int	Stores integers	10, −25, 100	2 or 4
	char	Stores characters	'A', '7', '$'	1
	float	Stores single precision real numbers	3.14, −0.25	4
	double	Stores double precision real numbers	3.14159, −12.3456	8
	void	Represents no value (used for functions)	void (no value)	—
2. Derived Types	array	Collection of similar elements	int a[10];	Depends on implementation
	pointer	Stores address of a variable	int *p;	
	structure	Group of different data	struct student	
	union	Group of different data (one at a time)	union data	
	function	Block of code	int add();	
3. Enumeration Type	enum	User defined data type with named integer constants	enum day {sun, mon, tue};	2 or 4

• Storage Classes in C :
Storage classes specify the scope, lifetime and visibility of variables.

Storage Class	Scope	Lifetime	Default Value	Used For
auto	Block (local)	Within block	Garbage value	Local variables
register	Block (local)	Within block	Garbage value	Local variables (in CPU register)
static	Block (local)	Entire program	Zero	Local variables, functions
extern	File (global)	Entire program	Zero	Global variables declared in other files

Q7. Symbolic Constants, const and Volatile Keywords.

Ans: ① **Symbolic Constants :**

- Symbolic constants are identifiers whose values do not change during program execution.
- They are defined using **#define** preprocessor directive.
- **Syntax :** #define NAME value
- **Example :**

```
#define PI  3.14159
#define MAX  100
```

> **Advantage :**
> Improves readability, easy to modify and reduces chances of errors.

② **const Keyword :**

- The const keyword is used to declare constants in C.
- A const variable cannot be modified once it is initialized.
- **Syntax :** const data_type variable_name = value;
- **Example :** const int age = 18;

> **Note :**
> const variables are checked by compiler and provide better type safety.

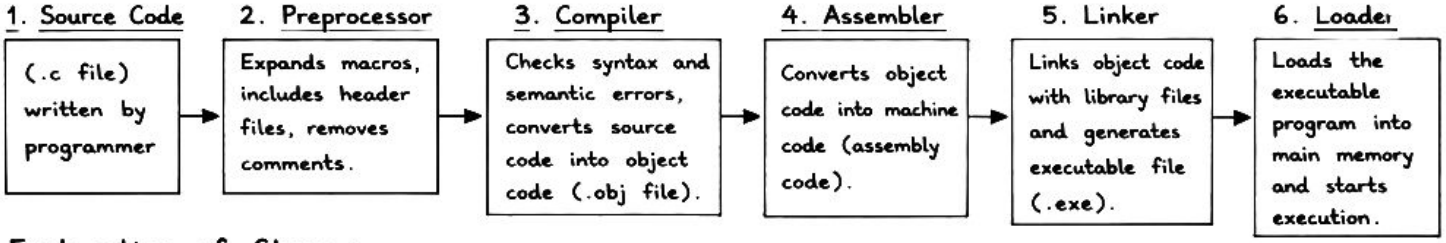
③ **volatile Keyword :**

- The volatile keyword tells the compiler that the value of a variable may change unexpectedly (e.g., by hardware or other programs).
- It prevents the compiler from optimizing the variable.
- **Syntax :** volatile data_type variable_name;
- **Example :** volatile int flag;

> **Use :**
> Used in system programming, hardware registers, interrupt service routines, etc.

Q8. C Program Compilation Process.

Ans: The process of converting a C program (source code) into an executable program is called compilation process.
It involves the following steps :



Explanation of Steps :

① **Source Code (.c file) :** The program is written in C and saved with .c extension.
② **Preprocessor :** Processes all #define, #include, conditional compilation directives and produces expanded source code.
③ **Compiler :** Checks the program for errors and converts it into object code (.obj). If errors are found, it reports them.
④ **Assembler :** Converts object code into machine (assembly) code.
⑤ **Linker :** Combines object code with library functions and other necessary modules to create an executable file (.exe).
⑥ **Loader :** Loads the executable file into memory and transfers control to the program. The program starts execution.

> **Note :**
> If any error occurs in any step, the compilation process stops and error messages are displayed.

Q1. Arithmetic, Relational and Logical Operators.

Ans:

(A) Arithmetic Operators :

- Used to perform mathematical operations.
- Operators : + , - , * , / , %
- + (addition), - (subtraction)
  * (multiplication), / (division)
  % (modulus - remainder)
- Operate on numeric operands (int, float, double).
- Example :

```
int a = 10, b = 3;
a + b = 13
a - b = 7
a * b = 30
a / b = 3
a % b = 1
```

(B) Relational Operators :

- Used to compare two operands.
- Result is either 1 (true) or 0 (false).
- Operators : == , != , > , < , >= , <=
- == (equal), != (not equal)
  > (greater than), < (less than)
  >= (greater or equal)
  <= (less or equal)
- Example :

```
int a = 5, b = 7;
a == b  →  0 (false)
a != b  →  1 (true)
a > b   →  0 (false)
a <= b  →  1 (true)
```

(C) Logical Operators :

- Used to combine or modify conditional expressions.
- Operators : && (logical AND)
  || (logical OR)
  ! (logical NOT)
- Result is 1 (true) or 0 (false).
- Example :

```
if (a > 0 && b < 10)
    printf("Valid");
```

Q2. Assignment, Increment-Decrement Operators.

Ans:

(A) Assignment Operators :

- Used to assign value to a variable.
- Basic operator : =
- Compound assignment operators :
  += , -= , *= , /= , %= ,
  <<= , >>= , &= , ^= , |=
- Example :

```
int a = 10;
a = 5;     //  a = 5
a += 5;    //  a = 10
a *= 2;    //  a = 20
a -= 3;    //  a = 17
```

(B) Increment and Decrement Operators :

- Used to increase or decrease value of a variable by 1.
- Operators : ++ (increment),
  -- (decrement)
- Two forms :
  1. Prefix : ++a , --a (operation first, then use)
  2. Postfix : a++ , a-- (use first, then operation)
- Example : int a = 5;

(i) Prefix

```
++a  →  6   (then a = 6)
--a  →  4   (then a = 4)
```

(ii) Postfix

```
a++  →  5   (then a = 6)
a--  →  5   (then a = 4)
```

Q3. Conditional and Bitwise Operators.

Ans:

(A) Conditional (Ternary) Operator :

- Also called ternary operator.
- Used as a shortcut for if - else statement.
- Syntax : condition ? expression1 : expression2 ;
- If condition is true, expression1 is executed ;
  otherwise expression2.
- Example : int max = (a > b) ? a : b ;
- Only one operator (? :) but takes three operands.

(B) Bitwise Operators :

- Used to perform operations on individual bits of operands.
- Operators : &, |, ^, ~, <<, >>
- & (bitwise AND), | (bitwise OR), ^ (bitwise XOR),
  ~ (bitwise NOT), << (left shift), >> (right shift)
- Operands must be of integer type.
- Example : int a = 5, b = 3; // a = 0101, b = 0011

  a & b = 0001 (1) a | b = 0111 (7)

  a ^ b = 0110 (6) ~a = ... (bitwise complement)

  a << 1 = 1010 (10) a >> 1 = 0010 (2)

Q4. Special Operators in C.

Ans:

① sizeof : Returns the size (in bytes) of a data type or variable.
Example : sizeof(int), sizeof(a)

② & (Address-of) : Returns the memory address of a variable.
Example : int a = 10; int *p = &a;

③ * (Pointer/Indirection) : Accesses the value stored at the address.
Example : int *p = &a; int b = *p;

④ , (Comma Operator) : Evaluates expressions from left to right
and returns the value of the last expression.
Example : int a = (x = 5, x + 2); // a = 7

⑤ [] (Array Subscript) : Used to access array elements.
Example : int arr[5]; arr[2] = 10;

⑥ -> (Structure Pointer Access) : Used to access members
of a structure using pointer.
Example : ptr->age;

Q5. Arithmetic Expressions and Evaluation.

Ans:

- An arithmetic expression is a combination of operands, operators and parentheses.
- Operators used : + , - , * , / , %
- Operands must be numeric (int, float, double etc.).
- Evaluation is the process of computing the value of an expression according to precedence and associativity.
- Parentheses ( ) are used to change the default order of evaluation.
- Order of evaluation follows operator precedence and associativity.
- Example :

```
int a = 10, b = 3, c = 5;
Expression : a + b * c - (a / b) % c
Step 1 : b * c = 3 * 5 = 15
Step 2 : a / b = 10 / 3 = 3 (integer division)
Step 3 : (a / b) % c = 3 % 5 = 3
Step 4 : a + 15 - 3 = 10 + 15 - 3 = 22
Result = 22
```

Q6. Operator precedence and associativity.

Ans:

- Precedence determines which operator is evaluated first in an expression.
- Higher precedence operators are evaluated before lower precedence operators.
- Associativity determines the direction of evaluation when operators have the same precedence.
- Most operators are left associative (evaluated from left to right).
- Some operators are right associative (evaluated from right to left).
- Precedence and associativity ensure unambiguous evaluation of expressions.
- Table of precedence (high to low) :

Precedence	Operators	Associativity
1 (Highest)	( )	Left to Right
2	*, /, %	Left to Right
3	+, -	Left to Right
4	<<, >>	Left to Right
5	<, <=, >, >=	Left to Right
6	==, !=	Left to Right
7	& (bitwise AND)	Left to Right
8	^ (bitwise XOR)	Left to Right
9	| (bitwise OR)	Left to Right
10	=, +=, -=, *=, /=, %=	Right to Left

- Example :

```
int a = 10, b = 5, c = 2;    Expression : a + b * c - a / c
Evaluation : b * c = 10 → a / c = 5 → a + 10 - 5 = 15
```

Name : _____

Roll No. : _____

Date : 25 / 05 / 2024

FPL (Fundamentals of Programming Languages)

Q7. Mathematical Functions in C

C language provides a rich set of mathematical functions in the math.h header file. These functions can be used to perform mathematical operations.

To use these functions, we include the header file :

```
#include <math.h>
```

Some commonly used mathematical functions are :

Function	Description	Example	Returns
sqrt(x)	Square root of x	sqrt(16.0)	4.0
pow(x, y)	x raised to the power y	pow(2.0, 3.0)	8.0
fabs(x)	Absolute value of x	fabs(-7.5)	7.5
sin(x)	Sine of x (in radians)	sin(3.1416 / 2)	1.0
cos(x)	Cosine of x (in radians)	cos(0.0)	1.0
tan(x)	Tangent of x (in radians)	tan(0.0)	0.0
log(x)	Natural logarithm of x (ln x)	log(2.71828)	1.0
log10(x)	Common logarithm of x (log base 10)	log10(1000.0)	3.0
ceil(x)	Smallest integer ≥ x	ceil(4.2)	5.0
floor(x)	Largest integer ≤ x	floor(4.8)	4.0
exp(x)	e raised to the power x	exp(1.0)	2.71828

Example Program :

```
#include <stdio.h>
#include <math.h>
int main()
{
    double a = 16.0, b = 2.0;
    printf("sqrt(%.1lf) = %.1lf\n", a, sqrt(a));
    printf("pow(%.1lf, %.1lf) = %.1lf\n", b, 3.0, pow(b, 3.0));
    printf("fabs(-7.5) = %.1lf\n", fabs(-7.5));
    return 0;
}
```

Output :

```
sqrt(16.0) = 4.0
pow(2.0, 3.0) = 8.0
fabs(-7.5) = 7.5
```

Q8. Infix, Prefix and Postfix Expressions.

An expression may be written in different notations depending on the position of the operator with respect to operands.

1) Infix Notation (Middle Notation)
   - Operator is written between the operands.
   - It is the most common form.
   - Example : (A + B) * C - D / E
2) Prefix Notation (Polish Notation)
   - Operator is written before the operands.
   - No parenthesee are required.
   - Example : - * + A B / D E C
3) Postfix Notation (Reverse Polish Notation)
   - Operator is written after the operands.
   - No parentheste are required.
   - Example : A B + D E / * C -

Infix	Prefix	Postfix
A + B	+ A B	A B +
A - B	- A B	A B -
A * B	* A B	A B *
A / B	/ A B	A B /
(A + B) * C	* + A B C	A B + C *
(A+B) * (C-D)	* + A B - C D	A B + C D - *
A + B * C - D	- + A * B C D	A B C * + D -

Note : Prefix and Postfix expressions are widely used in compilers, expression evaluation, and stack based processing.

UNIT III – CONTROL FLOW

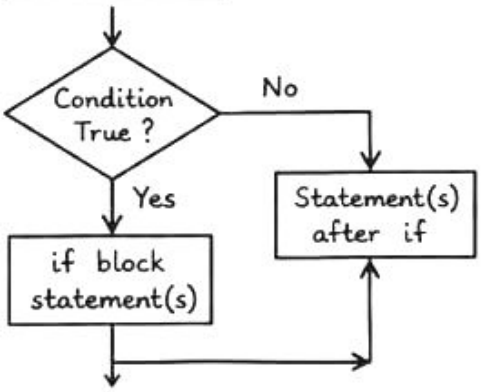
Roll No. : _____
Date : 25 / 05 / 2024

Q1. Decision making statements.

Decision making statements are used to execute different statements based on a condition. In C, the decision making statements are :

(i) if statement (ii) if-else statement

(i) if statement



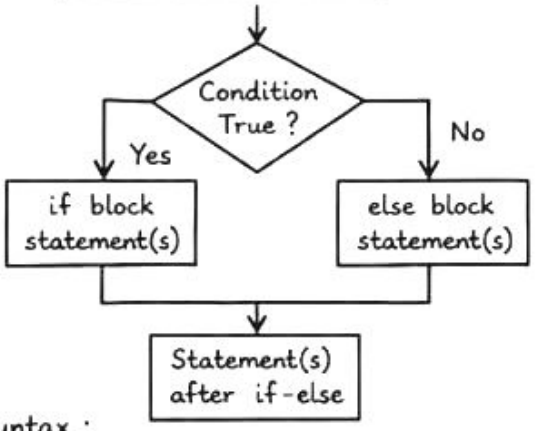
Syntax :

```
if (condition)
{
    // statements
}
```

Example :

```
if (marks >= 40)
{
    printf("Pass\n");
}
```

(ii) if-else statement



Syntax :

```
if (condition)
{
    // if part statements
}
else
{
    // else part statements
}
```

Example :

```
if (marks >= 40)
    printf("Pass\n");
else
    printf("Fail\n");
```

Q2. if, if-else, else-if ladder.

These are selection (decision making) statements used to control the flow of program based on conditions.

(i) if statement

```
if (condition)
{
    // statements
}
```

Example :

```
if (x > 0)
{
    printf("Positive\n");
}
```

(ii) if-else statement

```
if (condition)
{
    // if part statements
}
else
{
    // else part statements
}
```

Example :

```
if (x % 2 == 0)
    printf("Even\n");
else
    printf("Odd\n");
```

(iii) else-if ladder

```
if (condition1)
{
    // statements
}
else if (condition2)
{
    // statements
}
:
else
{
    // statements
}
```

← Any number of else if parts can be used.

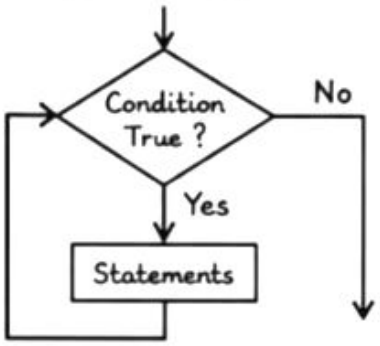
Example :

```
if (marks >= 90)
    printf("Grade A\n");
else if (marks >= 75)
    printf("Grade B\n");
else if (marks >= 50)
    printf("Grade C\n");
else
    printf("Grade D\n");
```

Q5. while, do-while and for loops.

Loops are used to execute a block of statements repeatedly as long as a given condition is true.

(i) while loop



Syntax :

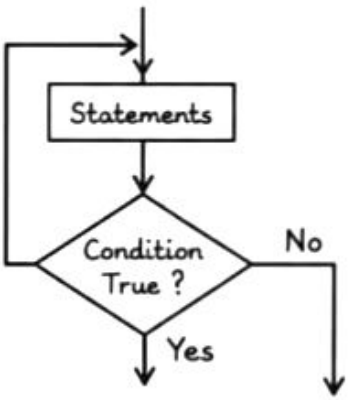
```
while (condition)
{
   // statements
}
```

Example :

```
int i = 1;
while (i <= 5)
{
   printf("%d ", i);
   i = i + 1;
}
```

Output : 1 2 3 4 5

(ii) do-while loop



Syntax :

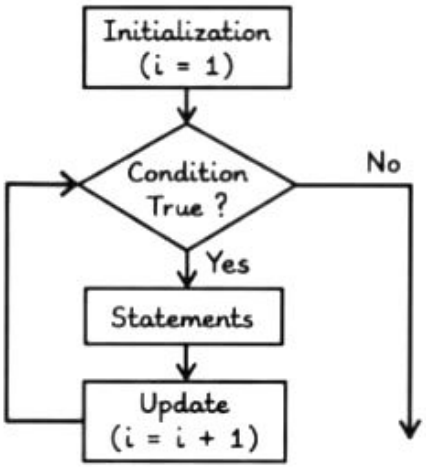
```
do
{
   // statements
} while (condition);
```

Example :

```
int i = 1;
do
{
   printf("%d ", i);
   i = i + 1;
} while (i <= 5);
```

Output : 1 2 3 4 5

(iii) for loop



Syntax :

```
for (initialization; condition; update)
{
   // statements
}
```

Example :

```
for (int i = 1; i <= 5; i = i + 1)
{
   printf("%d ", i);
}
```

Output : 1 2 3 4 5

Q6. break and continue statements.

These statements are used to alter the normal flow of loops.

(i) break statement

- It is used to terminate the loop immediately.
- Control transfers to the statement immediately after the loop.

Example : Print numbers from 1 to 10 but stop when 5 is encountered.

```
for (int i = 1; i <= 10; i++)
{
   if (i == 5)
      break;
   printf("%d ", i);
}
```

Output : 1 2 3 4

(ii) continue statement

- It is used to skip the remaining statements of the current iteration and proceed to the next iteration.

Example : Print numbers from 1 to 10 but skip printing 5.

```
for (int i = 1; i <= 10; i++)
{
   if (i == 5)
      continue;
   printf("%d ", i);
}
```

Output : 1 2 3 4 6 7 8 9 10

Q7. Program : Simple calculator.

Write a C program to perform basic arithmetic operations using switch case statement.

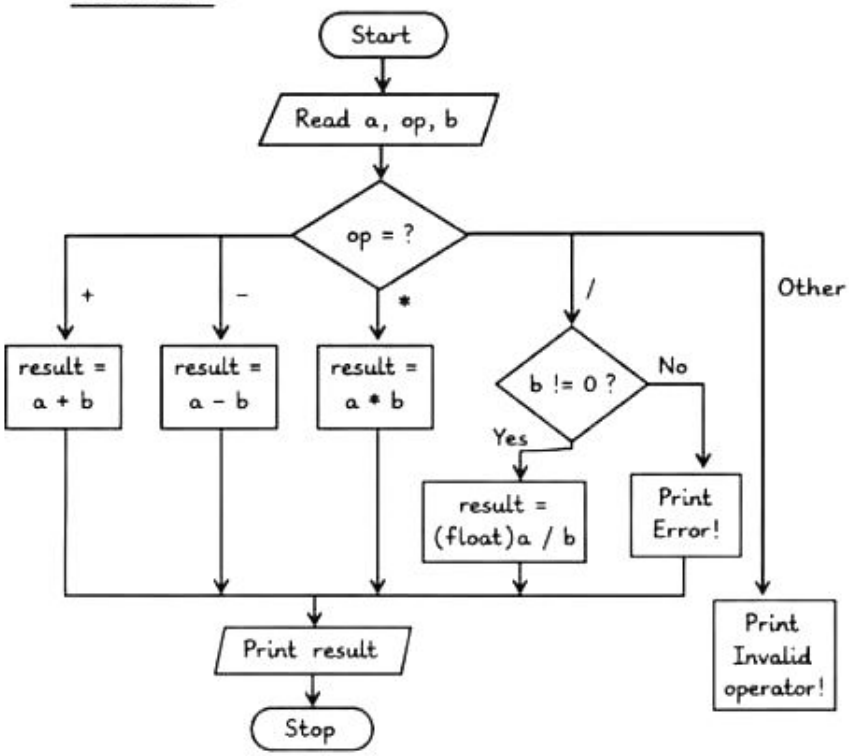
Program :

```c
#include <stdio.h>
int main()
{
    int a, b, choice;
    char op;
    float result;

    printf("Simple Calculator\n");
    printf("Enter first number : ");
    scanf("%d", &a);
    printf("Enter operator (+, -, *, /) : ");
    scanf(" %c", &op);
    printf("Enter second number : ");
    scanf("%d", &b);

    switch (op)
    {
      case '+':  result = a + b; break;
      case '-':  result = a - b; break;
      case '*':  result = a * b; break;
      case '/':
         if (b != 0)
             result = (float)a / b;
         else
         {
           printf("Error! Division by zero.\n");
           return 0;
         }
         break;
      default:
         printf("Invalid operator!\n");
         return 0;
    }
    printf("Result = %.2f\n", result);
    return 0;
}
```

Flowchart :



Example :

Input :	12 + 7
Output :	Result = 19.00
Input :	20 / 0
Output :	Error! Division by zero.
Input :	5 $ 3
Output :	Invalid operator!

Q8. Program : Calendar generation.

Write a C program to input month and year and print the calendar of that month. (Assume starting day of the month using Zeller's congruence.)

Program :

```c
#include <stdio.h>
int main()
{
   int day, month, year, i, j, days;
   printf("Enter month (1-12) : ");
   scanf("%d", &month);
   printf("Enter year : ");
   scanf("%d", &year);

   /* day = starting day (0=Sunday, 1=Monday, ..., 6=Saturday)
      Calculate using Zeller's congruence */
   day = zeller(month, year);  // assume user-defined function
   days = no_of_days(month, year); // 28/29/30/31

   printf("\nSun  Mon  Tue  Wed  Thu  Fri  Sat\n");

   for (i = 0; i < day; i++)
      printf("     ");    // spaces for starting day
   for (i = 1; i <= days; i++)
   {
      printf("%5d", i);
      if ((i + day) % 7 == 0)
         printf("\n");
   }
   printf("\n");
   return 0;
}
```

Example Output : (May 2024)

Sun	Mon	Tue	Wed	Thu	Fri	Sat
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

Notes :

- Zeller's congruence is used to find the day of the week for a given date.
- no_of_days() returns number of days in the given month and year (considering leap year for February).
- 0 = Sunday, 1 = Monday, ..., 6 = Saturday.

Q.1 One-dimensional arrays.

An array that stores elements of the same data type in a single row is called one-dimensional array.

Declaration : type array_name[size];

Example : int a[5];

Memory Representation :

Index →	0	1	2	3	4
a →	a[0]	a[1]	a[2]	a[3]	a[4]

Accessing elements :

Array elements are accessed using the index.

General form : array_name[index]

Example : a[0], a[1], a[2], a[3], a[4]

Program Example : (Read n numbers and display them)

```
#include <stdio.h>
int main()
{
    int n, i, a[10];
    printf("Enter n: ");
    scanf("%d", &n);
    printf("Enter %d numbers:\n", n);
    for(i = 0; i < n; i++)
       scanf("%d", &a[i]);
    printf("\nThe numbers are: ");
    for(i = 0; i < n; i++)
       printf("%d ", a[i]);
    return 0;
}
```

Notes :

- Array index always starts from 0.
- For array of size n, valid indices are 0 to n-1.
- All elements in an array are of the same data type.
- Size of array must be known at the time of declaration.

Q.2 Two-dimensional arrays.

An array that stores elements in the form of rows and columns (matrix form) is called two-dimensional array.

Declaration : type array_name[rows][columns];

Example : int a[3][4]; // 3 rows and 4 columns

Memory Representation :

Column index →		0	1	2	3
Row index ↓	0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
	1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
	2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Accessing elements :

Each element is accessed using two indices.

General form : array_name[i][j] (i = row index, j = column index)

Example : a[0][0], a[1][2], a[2][3] are valid elements.

Q.3 Character arrays and strings.

A string is a sequence of characters terminated by the null character '\0'. In C, strings are stored in character arrays.

Declaration : `char str[size];`

Example : `char name[20];`

String Representation :

Index →	0	1	2	3	4	5	...	n-2	n-1
str →	s[0]	s[1]	s[2]	s[3]	s[4]	s[5]	...	s[n-2]	'\0'

Notes :

- The last character of every string is the null character '\0'.
- The length of the string is the number of characters before '\0'.
- Size of array should be at least (length of string + 1).
- String can be initialized at the time of declaration.

Example :

```
char city[] = "PUNE";      // Automatically adds '\0' at the end
char msg[10] = "HELLO";    // Stored as H E L L O \0
```

Q.4 String input/output functions.

C provides library functions to read and write strings.

1. String Input Functions :

Function	Header File	Description
scanf("%s", str)	<stdio.h>	Reads a string (up to first whitespace). Example : scanf("%s", str);
fgets(str, size, stdin)	<stdio.h>	Reads a line of text (including spaces) up to (size-1) characters. Example : fgets(str, 50, stdin);

2. String Output Functions :

Function	Header File	Description
printf("%s", str)	<stdio.h>	Displays a string. Example : printf("%s", str);
puts(str)	<stdio.h>	Displays a string followed by a new line. Example : puts(str);

Note : All these functions work with null-terminated strings.

Q.5 String handling functions.

C provides various library functions to manipulate strings.

Function	Header File	Description	Example
strlen(str)	<string.h>	Returns the length of string str (excluding the null character '\0').	strlen("PUNE") → 4
strcpy(dest, src)	<string.h>	Copies string src into dest (including null character).	strcpy(a, b);
strcat(dest, src)	<string.h>	Appends string src at the end of dest.	strcat(a, b);
strcmp(str1, str2)	<string.h>	Compares str1 and str2. Returns 0 if equal, < 0 if str1 < str2, > 0 if str1 > str2.	strcmp(a, b);
strrev(str)	<string.h>	Reverses the string str.	strrev(a);
strupr(str)	<string.h>	Converts all lowercase letters in str to uppercase.	strupr(a);
strlwr(str)	<string.h>	Converts all uppercase letters in str to lowercase.	strlwr(a);

Note : These functions require #include <string.h> header file.

Q.6 Program: Matrix multiplication.

To multiply two matrices, the number of columns in the first matrix must be equal to the number of rows in the second matrix.
If A is an m x n matrix and B is an n x p matrix, then the product C = A x B is an m x p matrix where

$$C[i][j] = \sum_{k=0}^{n-1} A[i][k] \times B[k][j]$$

C Program :

```
#include <stdio.h>
int main()
{
   int i, j, k, m, n, p;
   int A[10][10], B[10][10], C[10][10];
   printf("Enter order of first matrix (m x n): ");
   scanf("%d %d", &m, &n);
   printf("Enter elements of first matrix:\n");
   for(i = 0; i < m; i++)
      for(j = 0; j < n; j++)
         scanf("%d", &A[i][j]);
   printf("Enter order of second matrix (n x p): ");
   scanf("%d %d", &m, &p);
   printf("Enter elements of second matrix:\n");
   for(i = 0; i < n; i++)
      for(j = 0; j < p; j++)
         scanf("%d", &B[i][j]);
   for(i = 0; i < m; i++)
      for(j = 0; j < p; j++)
      {
         C[i][j] = 0;
         for(k = 0; k < n; k++)
           C[i][j] += A[i][k] * B[k][j];
      }
   printf("\nProduct matrix C = A x B:\n");
   for(i = 0; i < m; i++)
   {
      for(j = 0; j < p; j++)
         printf("%d\t", C[i][j]);
      printf("\n");
   }
   return 0;
}
```

Sample Input / Output :

```
Enter order of first matrix (m x n): 2 3
Enter elements of first matrix:
1 2 3
4 5 6
Enter order of second matrix (n x p): 3 2
Enter elements of second matrix:
7 8
9 10
11 12
Product matrix C = A x B:
58    64
139   154
```

Note : Matrix multiplication is not commutative, i.e., in general A x B ≠ B x A.

1. Need and advantages of user-defined functions.

◆ **Need of user-defined functions:**

C language provides a rich set of library functions, but these functions cannot always solve every problem. In such cases, we can create our own functions according to our need. These are called user-defined functions.

◆ **Advantages of user-defined functions:**

1. **Modularity:**
   A large program can be divided into small modules (functions). This makes the program easier to design and understand.
2. **Reusability:**
   Once a function is written and tested, it can be used multiple times in the same program or in other programs.
3. **Reduced coding:**
   Repetitive code can be written in a single function and called whenever required. This reduces the size of the program.
4. **Easier debugging:**
   Errors can be located and corrected in a particular function without affecting other parts of the program.
5. **Maintainability:**
   Programs become easier to modify and maintain.
6. **Better readability:**
   Using functions makes the program structure clear and improves readability.

2. Function declaration, definition and call.

A user-defined function in C consists of three parts:

1. **Function declaration (or prototype):**
   It informs the compiler about the function name, return type and parameter types before using the function.

```
return_type function_name(type1 param1, type2 param2, ...);
```

Example:

```
int add(int, int);
```

2. **Function definition:**
   It contains the actual body of the function.

```
return_type function_name(type1 param1, type2 param2, ...)
{
   // local variable declaration
   // statements
   return value;     // if return_type is not void
}
```

Example:

```
int add(int a, int b)
{
   int sum;
   sum = a + b;
   return sum;
}
```

3. **Function call:**
   It transfers the control to the called function.

Example:

```
int result;
result = add(10, 20);    // function call
```

3. Categories of functions.

C library functions can be classified into the following categories:

No.	Category	Description	Example Functions
1.	**Input Functions**	Read data from the standard input (keyboard).	`scanf()`, `getchar()`, `gets()`
2.	**Output Functions**	Display data on the standard output (screen).	`printf()`, `putchar()`, `puts()`
3.	**String Handling Functions**	Perform operations on strings.	`strlen()`, `strcpy()`, `strcat()`, `strcmp()`, `strrev()`, `strupr()`, `strlwr()`
4.	**Mathematical Functions**	Perform mathematical calculations.	`sqrt()`, `pow()`, `sin()`, `cos()`, `log()`, `exp()`, `fabs()`
5.	**Character Handling Functions**	Perform operations on characters.	`isalpha()`, `isdigit()`, `isupper()`, `islower()`, `toupper()`, `tolower()`
6.	**Memory Management Functions**	Allocate and deallocate memory during program execution.	`malloc()`, `calloc()`, `realloc()`, `free()`
7.	**File Handling Functions**	Create, read, write and close files.	`fopen()`, `fclose()`, `fscanf()`, `fprintf()`, `fread()`, `fwrite()`
8.	**Miscellaneous Functions**	General utility functions.	`exit()`, `system()`, `rand()`, `srand()`, `time()`

4. Functions returning multiple values.

In C, a function can return only one value directly. To return multiple values from a function, we can use one of the following methods:

1. **Using Global Variables:**
   The function assigns values to global variables, which can be accessed in the calling function.

```c
int x, y;                // global variables
void getValues(void)
{
    x = 10;
    y = 20;
}
int main(void)
{
    getValues();
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

Note:
Global variables can be used anywhere in the program, but excessive use may reduce modularity.

2. **Using Call by Reference (Using Pointers):**
   The function accepts addresses of variables and stores the results in those variables.

```c
void getValues(int *a, int *b)
{
    *a = 10;
    *b = 20;
}
int main(void)
{
    int x, y;
    getValues(&x, &y);
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

Note:

This is the most commonly used and efficient method.

3. **Using Structures:**
   The function returns a structure that contains multiple values.

```c
struct Pair {
    int x;
    int y;
};
struct Pair getValues(void)
{
    struct Pair p;
    p.x = 10;
    p.y = 20;
    return p;
}
int main(void)
{
    struct Pair result = getValues();
    printf("x = %d, y = %d\n", result.x, result.y);
    return 0;
}
```

Note:

This method is useful when we want to return related data items together.

5. Nesting of functions.

In C, a function can call another function. This is called function nesting.
The called function may again call another function. This process can continue.

Key Points:

- The function that calls another function is called the calling function.
- The function that is called is called the called function.
- Nesting can be of any level, but it must not create an infinite loop.
- Each function has its own local variables and memory.

Example:

Function C is called inside Function B, and Function B is called inside Function A.

`main()` → Function A() → Function B() → Function C()

Illustration:

Level	Function Called	Called By	Description
1	Function A()	main()	main() calls Function A().
2	Function B()	Function A()	Function A() calls Function B.
3	Function C()	Function B()	Function B() calls Function C.
4	Return to B	Function C()	After completion, control returns to Function B.
5	Return to A	Function B()	After completion, control returns to Function A.
6	Return to main	Function A()	After completion, control returns to main().

Advantages:

- Improves modularity of the program.
- Breaks a large problem into smaller sub-problems.
- Improves code organization and readability.

6. Recursion with example.

Recursion is a technique in which a function calls itself directly or indirectly
to solve a problem by breaking it into smaller instances of the same problem.

Key Points:

- A recursive function must have a base case (termination condition).
- It should move towards the base case in each recursive call.
- It may cause stack overflow if base case is missing.

Example: Factorial of a number using recursion.

```c
#include <stdio.h>

int factorial(int n)
{
    if (n == 0 || n == 1)
        return 1;          // base case
    else
        return n * factorial(n - 1);  // recursive call
}

int main(void)
{
    int num, fact;

    printf("Enter a non-negative integer: ");
    scanf("%d", &num);

    fact = factorial(num);
    printf("Factorial of %d = %d\n", num, fact);

    return 0;
}
```

Explanation:

For n > 1,

factorial(n) = n * factorial(n - 1)

The recursion stops when n becomes 0 or 1.

Example Execution:

Input: 5

Recursive calls:

```
factorial(5)
= 5 * factorial(4)
= 5 * 4 * factorial(3)
= 5 * 4 * 3 * factorial(2)
= 5 * 4 * 3 * 2 * factorial(1)
= 5 * 4 * 3 * 2 * 1
= 120
```

Output: Factorial of 5 = 120

Advantages:

- Easy to write and understand for problems with recursive nature (e.g., factorial, Fibonacci).
- Often leads to simpler and shorter code.

Disadvantages:

- Uses more memory (function calls are stored on stack).
- May be slower due to repeated function calls.

7. Structure definition and initialization.

◆ **Structure:**

A structure is a user-defined data type that allows grouping of different data items (possibly of different types) under a single name.

◆ **Structure definition:**

```
struct structure_name
{
   data_type member1;
   data_type member2;
   ....
   data_type memberN;
};
```

Example:

```
struct Student
{
   int roll_no;
   char name[30];
   float marks;
};
```

◆ **Structure initialization:**

Structure variables can be initialized in the following ways:

1. **Initialization at the time of declaration:**

```
struct Student s1 = {101, "Amit", 85.50};  // Order must match
```

2. **Initialization after declaration:**

```
struct Student s2;
s2.roll_no = 102;
strcpy(s2.name, "Neha");
s2.marks = 91.25;
```

3. **Partial initialization (remaining members are set to 0 by default):**

```
struct Student s3 = {103, "Rahul"};   // marks will be 0.0
```

4. **Initialization using designated initializers (C99 and later):**

```
struct Student s4 = {.name = "Pooja", .roll_no = 104, .marks = 88.75};
```

◆ **Accessing structure members:**

Structure members are accessed using the **dot (.)** operator.

```
printf("Roll No: %d, Name: %s, Marks: %.2f", s1.roll_no, s1.name, s1.marks);
```

8. Accessing structure members.

Members of a structure can be accessed using the **dot (.)** operator.

Syntax:

```
structure_variable.member_name
```

Example:

```
struct Student
{
   int roll_no;
   char name[30];
   float marks;
};
```

Accessing members:

```
struct Student s = {101, "Amit", 85.50};

printf("Roll No: %d\n", s.roll_no);     // Access integer member
printf("Name   : %s\n", s.name);        // Access string member
printf("Marks  : %.2f\n", s.marks);     // Access float member
```

■ **Modifying members:**

```
s.marks = 90.25;              // Modify float member
strcpy(s.name, "Rohan");      // Modify string member
s.roll_no = 105;              // Modify integer member
```

■ **Accessing structure members using pointer to structure:**

A structure variable can also be accessed using a pointer.

```
struct Student s = {101, "Amit", 85.50};
struct Student *ptr = &s;

printf("Roll No: %d\n", ptr->roll_no);   // Using -> operator
printf("Name   : %s\n", ptr->name);      // Using -> operator
printf("Marks  : %.2f\n", ptr->marks);   // Using -> operator
```

Note:

Use dot (.) operator with structure variables.

Use arrow (->) operator with pointers to structure.

■ **Advantages of using structures:**

- Group related data items under a single name.
- Improves code readability and organization.
- Easy to handle complex data.