Day 1

Topics

Algorithm and Flowchart, Translator and its types, Applications of C programming , transition from algorithm to program, Syntax, logical errors and Runtime errors, Keywords, identifiers, constant, Data types, Type conversion, Operators and their types, Precedence and associativity,

ALGORITHM

- Algorithm refers to the logic of the program. It is a step by step description of how to arrive at the solution of the problem.
- An algorithm is a complete, detailed and precise step by step method for solving a problem independently of the hardware and software.

Characteristics of a good algorithm are:

- **Input** – the algorithm receives input.
- **Output** – the algorithm produces output.
- **Finiteness** – the algorithm stops after a finite number of instructions are executed.
- **Precision** – the steps are precisely stated (defined).
- **Uniqueness** – results of each step are uniquely defined and only depend on the input and the result of the preceding steps.
- **Generality** – the algorithm applies to a set of inputs.

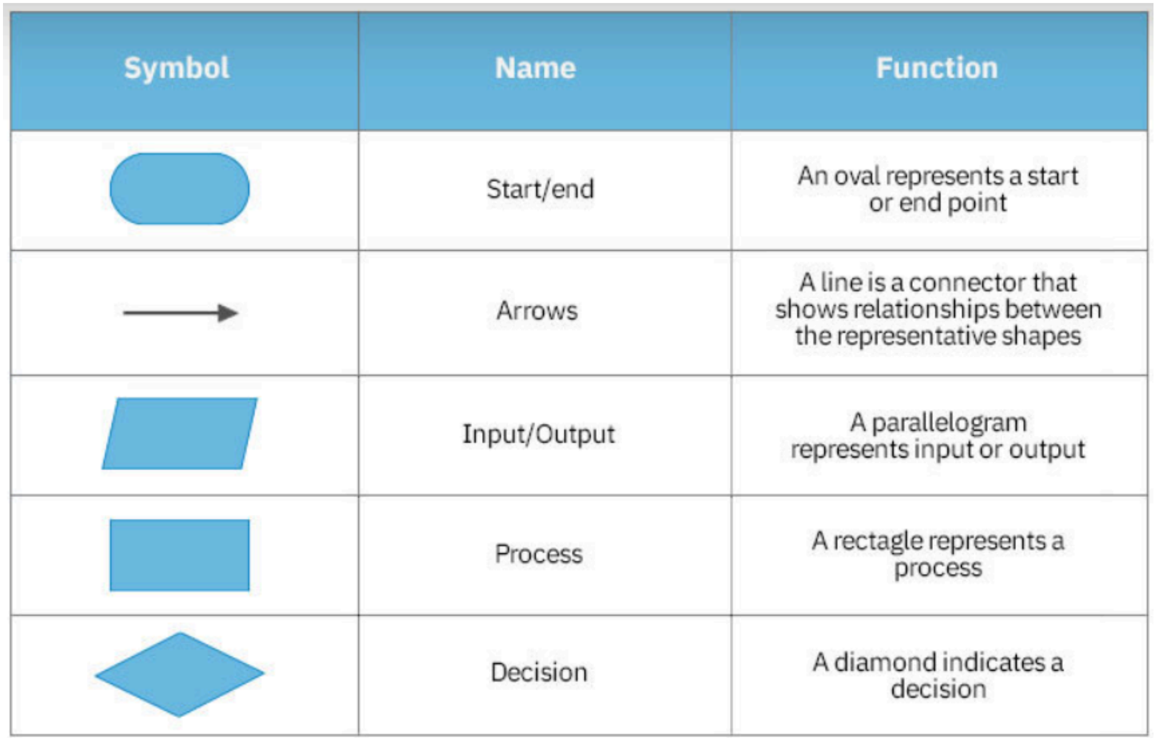
Example: Algorithm to find sum of two numbers:

```
1.    Begin
2.    Input the value of A and B
3.    SUM= A+B
4.    Display SUM
5.    End.
```

FLOWCHART

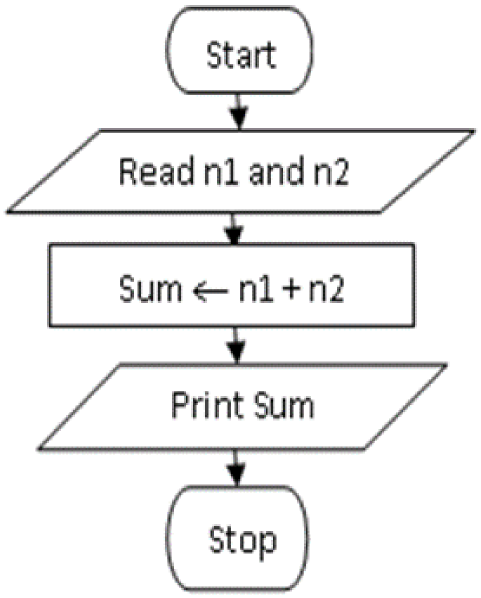
- A flowchart is a pictorial representation of an algorithm or process.

Symbols used:



Symbol	Name	Function
⬭	Start/end	An oval represents a start or end point
→	Arrows	A line is a connector that shows relationships between the representative shapes
▱	Input/Output	A parallelogram represents input or output
▭	Process	A rectagle represents a process
◇	Decision	A diamond indicates a decision

Example: Flow-Chart to find sum of two numbers



Questions: Write an algorithms and design flow charts for the following:

1. Find average of three numbers.
2. Swap two numbers using: i. Using third variable ii. Without using third variable
3. Find area and perimeter of a circle.
4. Check if a number is divisible by 7 or not.
5. Among three numbers: a,b and c, find the greatest number.
6. Check if a number is an Armstrong number or not.
7. Find Factorial of a number.

COMPUTER LANGUAGES

- Computer language or programming language is a coded syntax used by computer programmers to communicate with a computer.
- There are three types of Computer Languages
    - Machine Language
    - Assembly Language
    - High level Language

Machine Language

- It is a low level language.
- Instructions written in the form of 0 and 1.
- Communicate directly with the computer.
- Machine language or machine code is the native language directly understood by the computer’s central processing unit or CPU.
- This type of computer language is not easy to understand for humans, as it only uses a binary system.
- **Advantages:**
    - Computation speed is very fast.
    - Directly understandable by computer.
- **Disadvantages:**
    - Writing a program is very time consuming.
    - Error correction is a tedious process.

Assembly Level Language

- It is also a low level language.
- Written in the form of symbolic codes (mnemonics).
- Symbolic codes are like- ADD, SUB, MUL, DIV, LOAD, STORE, etc.
- Assembly Level Language is a set of codes that can run directly on the computer’s processor.
- This type of language is most appropriate in writing operating systems and maintaining desktop applications.
- With the assembly level language, it is easier for a programmer to define commands. It is easier to understand and use as compared to machine language.
- **Advantages**
    - Easier to understand as compared to machine code.
    - Easy to locate and correct errors.
- **Disadvantages**
    - Like machine language it is also machine dependent.
    - Programmers should have knowledge of hardware.

High Level Language

- High Level Languages are user-friendly languages which are similar to English with vocabulary of words and symbols.
- These are easier to learn and require less time to write.
- They are problem oriented rather than ‘machine’ based.
- Programs written in a high-level language can be translated into many machine languages and therefore can run on any computer for which there exists an appropriate translator.
- Examples: COBOL, FORTRAN, PASCAL, C, C++, JAVA etc.
- **Advantages**
    - User-friendly.
    - Easier to learn.
    - Require less time to code.
    - Easier to maintain.
- **Disadvantages**
    - Slower than low level language.
    - Needs a translator.

LANGUAGE TRANSLATORS

- Used to convert one programming language to another.
- Computers understand only Machine Language. Most of the programming is done in High level languages as it is much easier to code in these languages.
- The need is there to translate a high level language program into machine language.
- The system programs which perform this job are called language translators.
- They are classified into three categories

    - Assembler
    - Interpreter
    - Compiler

Assembler

- This language processor converts the program written in assembly language into machine language.

Interpreter

- It converts High level language into low level.
- It translates line by line.

Compiler

- It converts High level language into low level.
- It converts entire program at once.

Applications of C programming

Applications of C programming is a powerful and versatile language used in various domains due to its efficiency, performance, and flexibility. Here are some key applications of C programming:

1. System Software Development

- **Operating Systems:** Many operating systems, including Unix, Linux, and parts of Windows, are written in C. C provides the low-level access to memory and system processes needed for OS development.
- **Embedded Systems:** C is widely used in embedded systems, which are specialized computing systems that perform dedicated functions within larger systems (e.g., automotive systems, medical devices, consumer electronics).

2. Application Software Development

- **Desktop Applications:** C is used for developing high-performance applications such as graphics editors (e.g., Adobe Photoshop), office suites, and video editors.

- **Compilers and Interpreters:** Compilers for various programming languages, including the C compiler itself, are often implemented in C due to its efficiency and control over system resources.

3. Hardware Drivers and Firmware

- **Device Drivers:** C is used to develop drivers that allow the operating system to communicate with hardware devices like printers, display adapters, and storage devices.
- **Firmware:** Firmware, which provides low-level control for the device's specific hardware, is often written in C to ensure efficient operation and compatibility.

4. Network Programming

- **Network Protocols:** Many network protocols and their implementations are written in C. This includes protocols used for communication over the internet (e.g., TCP/IP).
- **Server and Client Software:** Servers and client software that handle network communications, such as web servers and mail servers, often utilize C for its performance benefits.

5. Game Development

- **Game Engines:** C and its derivative C++ are used in the development of game engines that require real-time processing and high performance.
- **Graphics and Simulation:** C is used to write high-performance graphics and simulation code that is fundamental in game development and other graphic-intensive applications.

6. Database Systems

- **Database Management Systems (DBMS):** Several DBMS systems, such as MySQL, are written in C. C allows fine-tuning of performance-critical database operations and memory management.

7. Scientific Computing

- **Numerical Analysis and Computation:** C is used in scientific computing for tasks that require intensive numerical computation and precise control over hardware.
- **Simulation and Modeling:** Scientific simulations and models that require high performance and efficiency are often written in C.

8. Embedded Software in Consumer Electronics

- **Telecommunications:** C is used in the software that runs on telecommunications devices, including modems and routers.
- **Consumer Devices:** Many consumer electronics, such as smart TVs, microwave ovens, and washing machines, use C for their embedded software.

9. IoT (Internet of Things)

- **IoT Devices:** C is commonly used in programming IoT devices that require efficient code to operate with limited resources (e.g., sensors, actuators, smart home devices).

10. Utility Programs

- **Text Editors:** Many text editors, which are critical tools for software development, are written in C.
- **System Utilities:** Utilities like disk defragmenters, file managers, and performance monitoring tools are often developed using C.

Transitioning from an algorithm to a program

1. Define the Problem

Before writing an algorithm, clearly define the problem you are trying to solve. Understand the inputs, desired outputs, and any constraints or requirements.

2. Design the Algorithm

An algorithm is a step-by-step procedure to solve a problem. It should be clear, unambiguous, and finite. Here are the steps to design an algorithm:

Identify Inputs and Outputs: Specify what data you will receive as input and what data you need to output.

Outline Steps: Break down the problem into a series of logical steps. Use pseudocode or flowcharts to represent these steps.

Consider Edge Cases: Think about possible edge cases and how your algorithm will handle them.

Iterate and Refine: Review and refine your algorithm to ensure it covers all aspects of the problem.

3. Choose a Programming Language

Select an appropriate programming language based on factors such as performance requirements, ease of use, and specific features needed for the task.

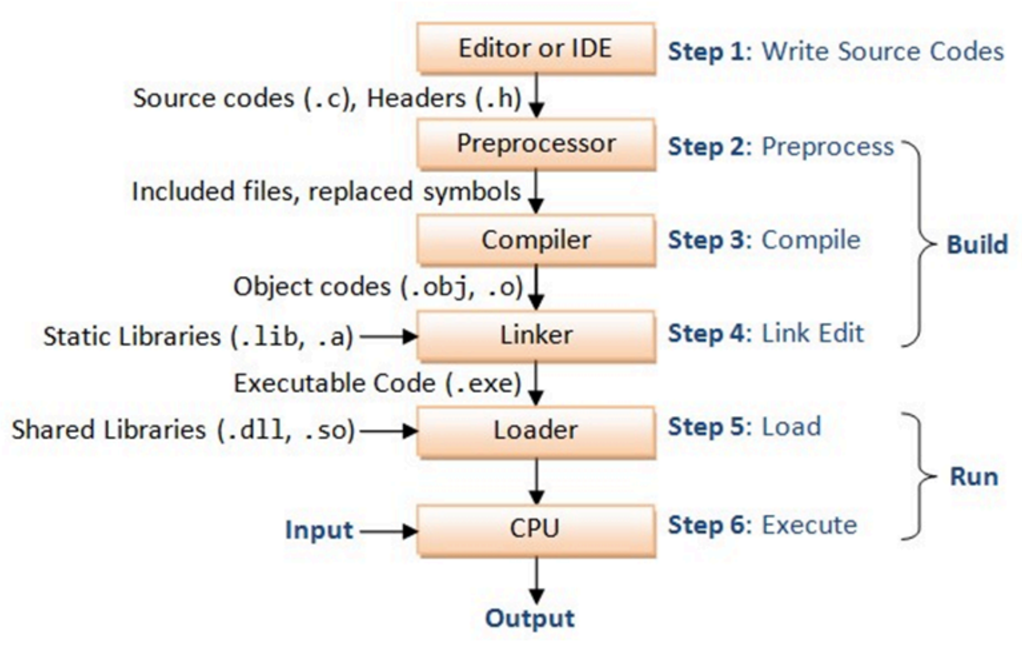
4. Write the Program

Translate the algorithm into a program using the syntax and features of your chosen programming language. Here's how you can do this step-by-step:

Translate the Algorithm to Code
Convert each step of your algorithm into corresponding code. For the example algorithm, here's how you might write it in Python and C.

5. Test the Program
Test the program if it matches with all the test cases.

Steps in Program Execution



Syntax errors, logical errors and Runtime errors

Syntax errors

- Syntax errors occur when the rules of the C language are violated. These are detected by the compiler during the compilation process.

- Common Causes:
    - Misspelled keywords
    - Missing semicolons at the end of statements
    - Mismatched parentheses, brackets, or braces
    - Incorrect use of operators

Logical errors

- Logical errors occur when the program compiles and runs, but the output is not as expected. These are errors in the logic or algorithm used in the program.

- Common Causes:
    - Incorrect algorithm implementation
    - Wrong conditions in loops or if statements
    - Incorrect use of variables or operators

Runtime Errors

- Runtime errors occur while the program is running. These errors are not detected during compilation and typically cause the program to terminate abnormally.

- Common Causes:
    - Division by zero
    - Null pointer dereferencing
    - Array index out of bounds
    - Memory allocation failures

C programming:

Example 1: use semicolon ; at the end of every statement.

```
#include<stdio.h>

void main() {
    printf("Hello World");
}
```

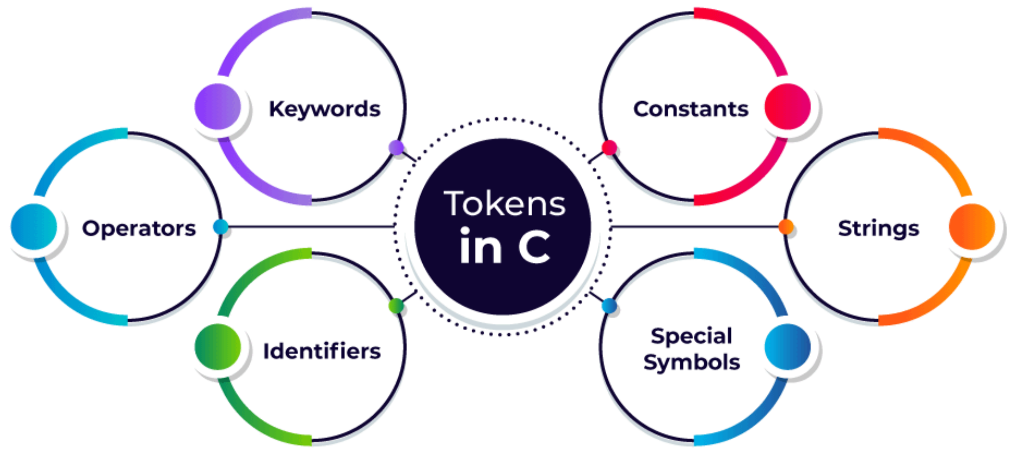
Example 2: variable declaration is important before using them. Use %d for int and %f for float.

```
#include<stdio.h>

void main() {
    int x =7;
    float y = 5.25;
    printf("value of x = %d \n", x);
    printf("value of y = %f", y);
}
```

Features of C

1. Procedural Language
2. Fast and Efficient
3. Modularity
4. General-Purpose Language
5. Rich set of built-in Operators
6. Libraries with Rich Functions
7. Middle-Level Language: C is High level language with some features of low level language, hence, it is called middle level language.
8. Portability: Programs that are written in C language can run and compile on any system with either no or small changes.
9. Easy to Extend

C Tokens



Keywords

Keywords are predefined or reserved words that have special meanings to the compiler. These are part of the syntax and cannot be used as identifiers in the program. A list of keywords in C or reserved words in the C programming language are mentioned below:

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

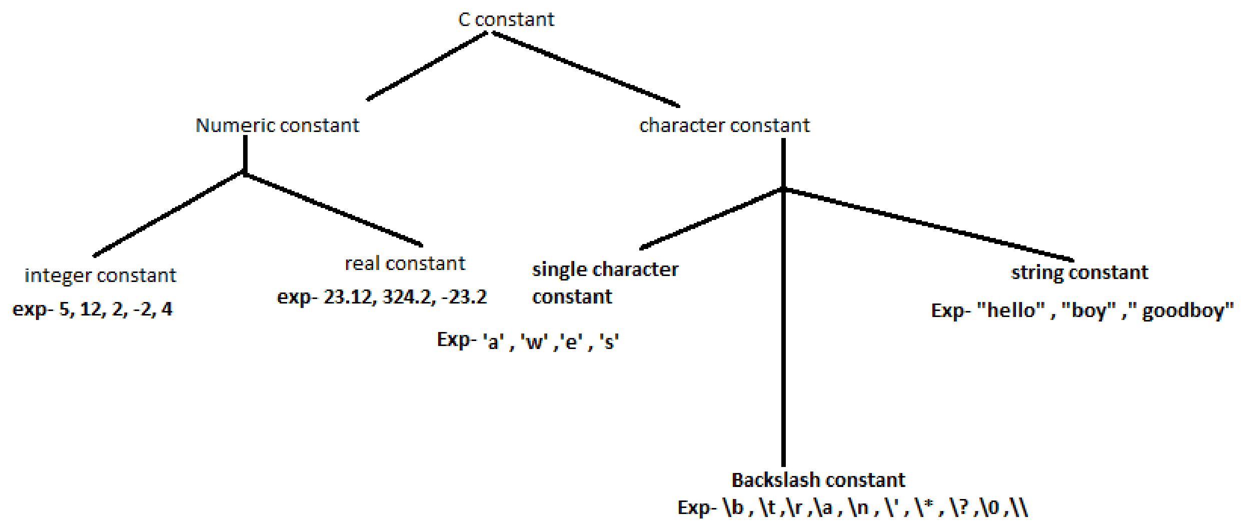
Identifier

Identifier is a name used to identify a variable, function, or array.

Rules for constructing C identifiers

1. Identifiers must start with a letter or an underscore.
2. Identifiers can only contain letters, digits, and underscores.
3. No Commas or blank spaces within an identifier.
4. Identifiers are case-sensitive.
5. Identifiers cannot be a reserved keyword.
6. The length of the identifiers should not be more than 31 characters.

C-Constants



Operators

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

1. Arithmetic Operators

2. Assignment Operators
3. Increment Decrement Operators
4. Relational Operators
5. Logical Operators
6. Bitwise Operators
7. Conditional Operator
8. Special Operator

1. Arithmetic Operators:

For a=9, b=3

OPERATION	OPERATOR	SYNTAX	COMMENT	RESULT
Multiply	*	a * b	result = a * b	27
Divide	/	a / b	result = a / b	3
Addition	+	a + b	result = a + b	12
Subtraction	-	a - b	result = a – b	6
Modulus	%	a % b	result = a % b	0

2. Assignment Operators

OPERATOR	MEANING	EXAMPLE
=	ASSIGNMENT	A=5 WILL ASSIGN 5 TO A

3. Increment Decrement Operators

OPERATOR	MEANING	EXAMPLE
++	INCREMENT	A++ INCREMENTS THE VALUE OF A TO 1
--	DECREMENT	A– DECREMENTS THE VALUE OF A TO 1

4. Relational operators

OPERATOR	MEANING	EXAMPLE
<	LESS THAN	3 < 5 GIVES 1
>	GREATER THAN	7 > 9 GIVES 0
>=	LESS THAN OR EQUAL TO	100 >= 100 GIVES 1
<=	GREATER THAN EQUAL TO	50 >=100 GIVES 0
==	IS EQUAL TO	5==5 GIVES 1
!=	NOT EQUAL TO	5!=5 GIVES 0

5. Logical Operators

OPERATOR	MEANING	EXAMPLE
&&	LOGICAL AND	(6>5) && (7>9) GIVES 0
||	LOGICAL OR	(6>5) || (7>9) GIVES 1
!	LOGICAL NOT	!(6>5) GIVES 0

6. Bitwise Operators

OPERATOR	MEANING	EXAMPLE
&	BITWISE AND	5 & 6 GIVES 4
|	BITWISE OR	5 | 6 GIVES 7
^	BITWISE X-OR	5 ^ 6 GIVES 3
<<	BITWISE LEFT SHIFT	5<<2 GIVES 20
>>	BITWISE RIGHT SHIFT	5>>2 GIVES 1
~	BITWISE NOT	~6 GIVES -7

How 5 & 6 gives 4?

Convert decimal 5 and 4 in binary. Perform bitwise AND operation. Convert result back into Decimal.

5	=	0101
6	=	0110
5 & 6	=	0100 = 4

7. Conditional Operator

Also called ternary operator as it takes 3 operands.

exp1 ? exp2 : exp3

Example:

```c
#include<stdio.h>

void main() {
    int a = 12, b = 7, c;
    c = a > b ? a : b;
    printf("value of c = %d ", c);
   }
```

8. Special Operators

These will be discussed later when we will be using them in upcoming chapters.

A. , comma operator
B. . member access operator
C. sizeof() operator
D. * dereferencing operator

Operator Precedence and Associativity

- Both are used to determine the order of evaluation of Operators in an expression.
- Precedence is always applied before associativity.
- Associativity can be left to right or right to left.

Example1: (precedence)

3 + 7 * 2   (Here * has higher precedence than +. So * will be evaluated first)

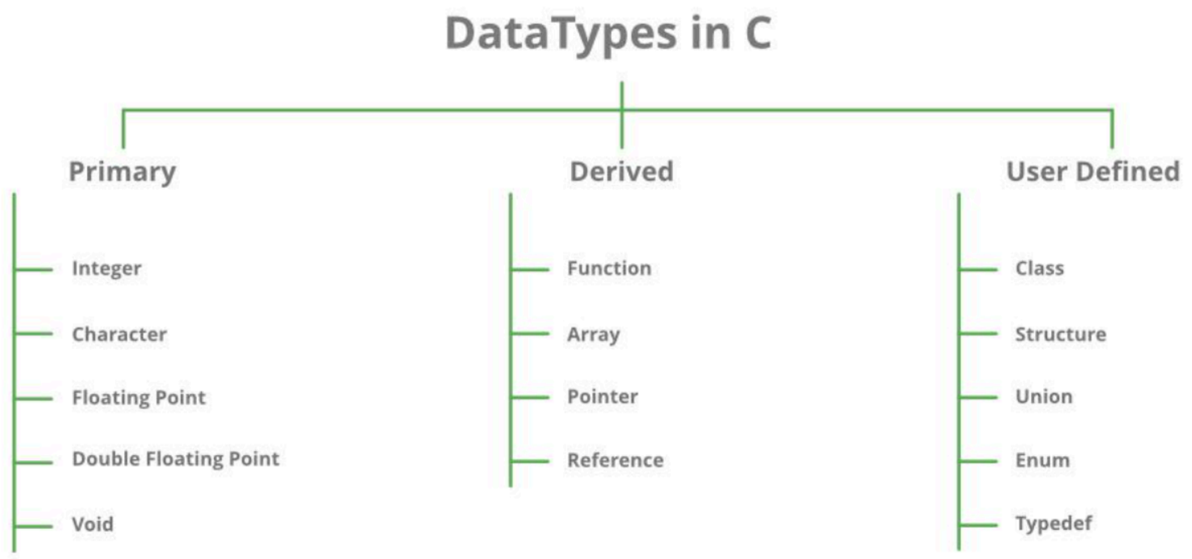
Example2: (associativity)

3 - 4 + 6   (Here + and - both have the same precedence so according to associativity of arithmetic operators, which is left to right, in this example - will be evaluated before +)

Example3: (associativity)

3 + 4 - 6   (Here + and - both have the same precedence so according to associativity of arithmetic operators, which is left to right, in this example + will be evaluated before -)

Data Types



Primary or Fundamental data types:

(Derived and user defined data types will be discussed later during the course)

Data Type	Size (bytes)	Range	Format Specifier
short int	2	-32,768 to 32,767	%hd
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
long long int	8	-(2^63) to (2^63)-1	%lld
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

Type Conversion in C

- Type conversion in C is the process of converting one data type to another.

Type of Type conversion

There are two types of Conversion:

1. Implicit Type Conversion

- Also known as ‘automatic type conversion’.
- Done by the compiler on its own, without any external trigger from the user.
- Generally takes place when in an expression more than one data type is present. In such conditions type conversion (type promotion) takes place to avoid loss of data.

Example:

```
#include <stdio.h>
int main()
{
    int x = 10; // integer x
    char y = 'a'; // character c

    // y implicitly converted to int. ASCII
    // value of 'a' is 97
    x = x + y;

    printf("x = %d, x);
    return 0;
}
```

2. Explicit Type conversion

- This process is also called type casting and it is user-defined.
- Here the user can typecast the result to make it of a particular data type.
- Syntax

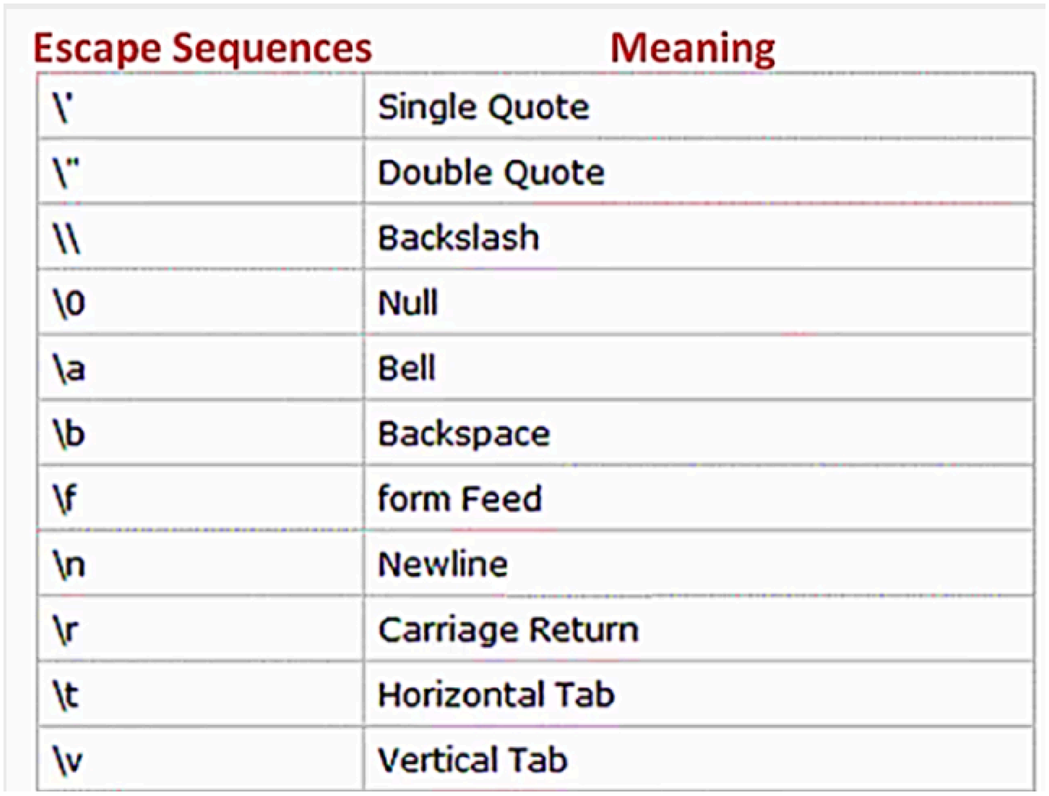
(type) expression

Example :

```c
#include <stdio.h>
int main()
{
    double x = 1.2;

    // Explicit conversion from double to int
    int sum = (int)x + 1;

    printf("sum = %d", sum);

    return 0;
}
```

Escape Sequence

- Escape sequences in C are combinations of characters that represent special characters or actions.
- Escape sequences start with a backslash (\) followed by a character or combination of characters.



Escape Sequences	Meaning
\'	Single Quote
\"	Double Quote
\\	Backslash
\0	Null
\a	Bell
\b	Backspace
\f	form Feed
\n	Newline
\r	Carriage Return
\t	Horizontal Tab
\v	Vertical Tab

Programs for day1 (Hands-on):

1. Find average of three numbers
2. Find Area and perimeter of Circle.
3. Find Area of triangle
    a. using base and height
    b. using Heron’s formula
4. Convert Fahrenheit temperature into celsius. C = (F-32) * 5 / 9
5. Program to swap two numbers.

Day_2 C-Programming

Conditional Program Execution

(RECAP OF PREVIOUS DAY)

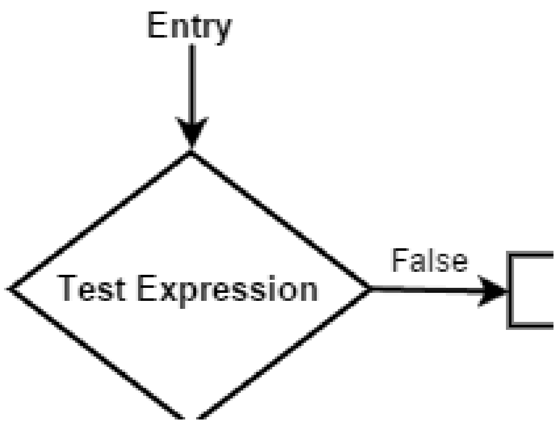
if, else-if, nested if-else, switch statements, use of break, and default with switch, Concept of loops, for, while and do while, multiple loop variables, use of break and continue statements, nested loop.

Conditional Statements: In C Programming we want the program to be executed sequentially. We want a set of instruction to be executed at one time and an entirely set of instruction to be executed in another situation. In this type of situation will have to use the decision control structure.

In C Language decision making capabilities are supported by following statements.

- **if statement**
- **switch statement**
- **Conditional Operator**

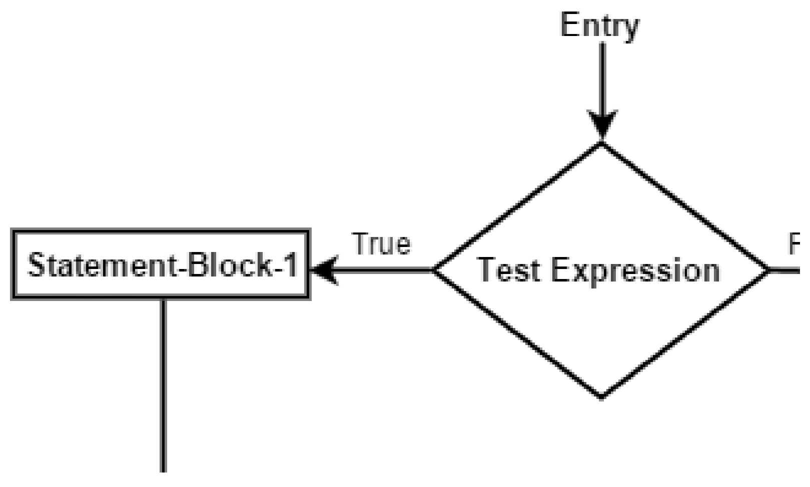
if statements: The if statement is a decision making statements. It is two-way decision statement and used in combination with an expression.

Syntax:
if(test expression)
{
Statement-block;
}
Statement-X;
	

if else Statement: In this, the body of if is executed if the result of expression is true, otherwise the else block is executed and then the control is transferred to the next statement.

Syntax:

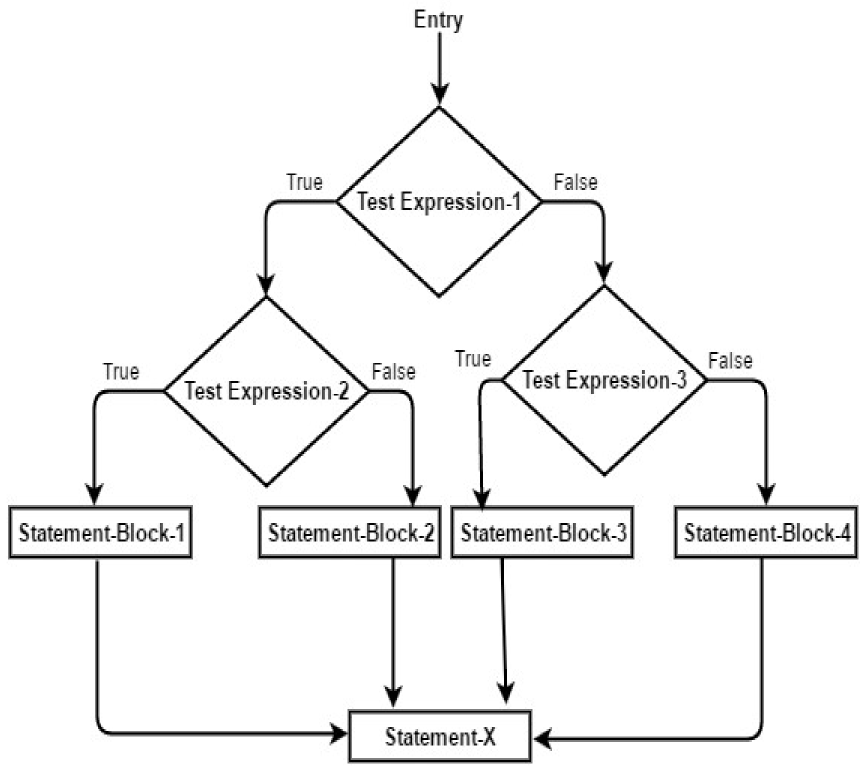
```
if(test expression)
{
statement block-1;
}
else
{
statement block-2;
}
statement-x;
```



Nested if else Statement: When an if statement is written inside another if statement, it is called a nested if statement.

Syntax:

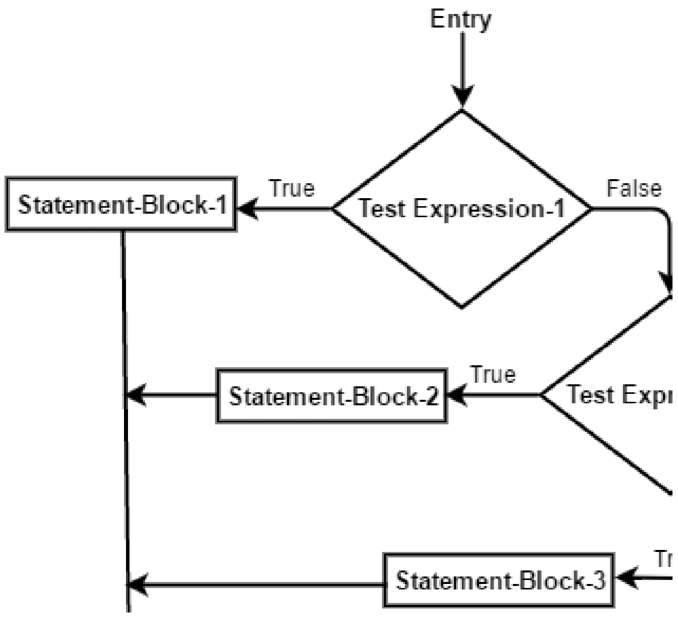
```
if(test expression-1)
{
    if(test expression-2)
     {
       statement block-1;
      }
     else
     {
       statement block-2;
      }
}
else
{
    if(test expression-3)
     {
      statement block-3;
      }
    else
     {
      statement block-4;
     }
}
Statement-x;
```



else if Ladder: It is a multiway branching statement. It start by checking the first logical expression. If the expression is true, then the body of first if is executed. Otherwise the next expression is evaluated and so on. When an expression is found to be true, the statements associated with it are executed and then the control is transferred.

Syntax:

```
if(test expression-1)
{
Statement block-1
}
else if(test expression-2)
{
Statement block-2
}
else if(test expression-3)
{
Statement block-3
}
else
{
statement block-4
}
Statement-x;
```



- ✓ **Write a program to check whether a number is even or odd.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
int n;
printf("Enter a Number");
scanf("%d",&n);
if(n%2==0)
printf("Number is Even");
else
printf("Number is odd");
getch();
}
```

- ✓ **Write a program to check whether a year is a leap year or not.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
int y;
printf("Enter a Year");
scanf("%d",&y);
if((y%4==0 && y%100!=0)||(y%400==0))
printf("Year is a leap year");
else
printf("Year is not a leap year");
getch();
}
```

- ✓ **Write a program to convert an uppercase letter into lowercase and vice-versa.**

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

```
{
char ch;
printf("Eneter a character");
scanf("%c",&ch);
if(ch>='A' && ch<='Z' )
{
ch=ch+32;
printf("%c",ch);
}
else if(ch>='a' && ch<='z')
{
ch=ch-32;
printf("%c",ch);
}
else
printf("Invalid Character");
getch(); }
```

- ✓ **Write a program to find the largest number among three numbers.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b,c;
printf("Enter three Numbers");
scanf("%d%d%d",&a,&b,&c);
if(a>b && a>c)
printf("%d is largest",a);
else if(b>a && b>c)
printf("%d is largest",b);
else
printf("%d is largest",c);
getch();
}
```

- ✓ **Write a program to check whether a number is positive, negative or zero.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
int n;
printf("Enter Number");
scanf("%d",&n);
if(n>0)
printf("Number is Positive");
else if(n<0)
printf("Number is Positive");
else
printf("Number is Zero");
getch();
}
```

- ✓ **Write a program to find the largest number among three numbers**

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

```
#include<conio.h>
void main()
{
int a, b, c, x;
printf("Enter three numbers");
scanf("%d%d%d",&a,&b,&c);
x=a>b?a>c:a:c:b>c?b:c;
printf("Largest number=%d",x);
getch();
}
```

Switch Statement: The switch statement is a multiway decision statement. The switch statement test the value of a given variable or expression against a list of case values and when a match is found a block of statement associated with that case is executed.

Syntax:

```
switch(expression)
{
caes value1: block-1;
              break;
caes value2: block-2;
              break;
caes value3: block-3;
             break;
default:     block-1;
             break;
}
```

Advantages of Switch

(i). Easier to debug.
(ii). Easier to read, understand, and maintain.
(iii). Faster execution.
(iv). More efficient (destination can be computed by looking up in table).

Limitations (Disadvantages) of switch statement

(i). It doesn't work with floats, strings, etc.
(ii). Relational and logical operators are not allowed in switch case.
(iii). It doesn't work with variable conditions i.e. case values cannot be variable.

✓ **Write a program to simulate calculator using switch statement.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
float a,b,c;
char op;
printf("Enter two numbers");
scanf("%f%f",&a,&b);
fflush(stdin);
printf("Enter Operator");
scanf("%c",&op);
switch(op)
{
case '+': c=a+b;
          printf("Addition=%f",c);
          break;
case '-': c=a-b;
          printf("Subtraction=%f",c);
          break;
case '*': c=a*b;
          printf("Multiplication=%f",c);
          break;
case '/': c=a/b;
          printf("Division=%f",c);
          break;
default: printf("Invalid Operator");
          break;
}
getch();
}
```

✓ **Write a menu driven program to calculate the area of different geometrical figures such as rectangle, square, circle and triangle.**

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
void main()
{
float a,b,c,s,area;
int n;
printf("Enter:\n1:For rectangle\n2:For square\n3:For circle\n4:For triangle\n");
scanf("%d",&n);
switch(n)
{
case 1: printf("Enter length and breadth of rectangle");
          scanf("%f%f",&a,&b);
          area=a*b;
          printf("Area of Rectanle:%f",area);
          break;
case 2: printf("Enter the side of square");
          scanf("%f",&a);
          area=a*a;
          printf("Area of Square:%f",area);
          break;
case 3: printf("Enter the radius of circle");
          scanf("%f",&a);
          area= 3.14*a*a;
          printf("Area of Circle:%f",area);
          break;
case 4: printf("Enter the three sides of triangle ");
          scanf("%f%f%f",&a,&b,&c);
          s=(a+b+c)/2;
          area=sqrt(s*(s-a)*(s-b)*(s-c));
```

```
            printf("Area of Triangle:%f",area);
            break;
default: printf("Invalide Choice");
            break;
}
getch();
}
```

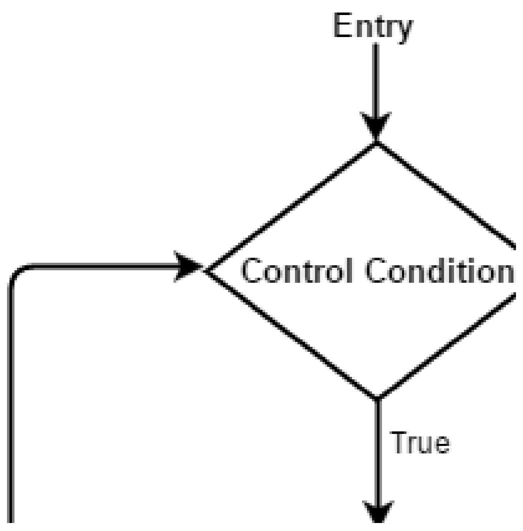
Difference between switch and if statement

switch	if
1) In switch statement the condition is tested only once and jumps to required block. **2)** A switch statement is generally best to use when you have more than two conditional expressions based on a single variable of numeric & character type. **3)** Logical operators are not allowed in case.	**1)** In multiple if statements the conditions are to be checked as many times the if statements are written. **2)** A if else statement can take any kind of value in conditional expression. **3)** All operators are allowed in if else statement.

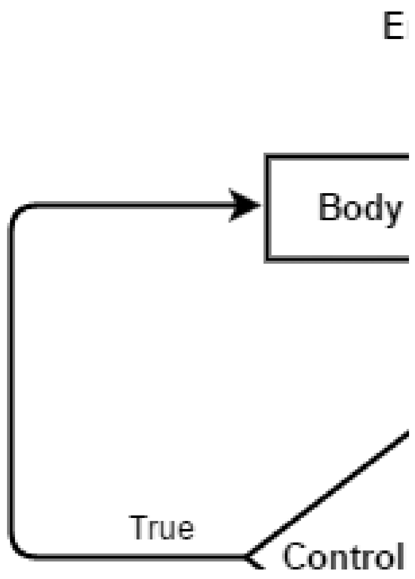
Looping: In looping, a sequence of statements are executed until some condition for termination of the loop are satisfied.
A program loop consist of two parts- one known as control statement and other known as body of loop. Depending on position of control statements there are ***two*** types of loop.

(1) Entry controlled loop
(2) Exit controlled loop

(1) **Entry controlled loop:** In this type of loop the control conditions are tested before the body of loop is executed. Example: ***for*** and ***while***.



(2) **Exit controlled loop:** In this type of loop the control conditions are tested at the end of the body of loop. Example: ***do while***.



for loop: It is an entry controlled loop. This loop is used when the number of iterations are known in advance.

Syntax: *for(initialization; condition; increment/decrement)*
```
{
Body of loop;
}
```

while loop: It is an entry controlled loop. This loop is used when the number of iterations are not known in advance.

Syntax: *while(condition)*
```
{
Body of loop;
}
```

do while loop: It is an exit controlled loop. In this loop the body of loop is always executed at least one time even if the condition is false for the first time .

Syntax: *do*
```
{
Body of loop;
}
while(condition);
```

Differentiate among for, while and do while loop

for loop	while loop	do-while loop
(i). It is an entry controlled loop.	**(i).** It is an entry controlled loop.	**(i).** It is an exit controlled loop.
(ii). Its body may not be executed even once if the condition is false.	**(ii).** Its body may not be executed even once if the condition is false.	**(ii).** Its body is always executed at least once even if the condition is false initially.
(iii). It is used when the number of iterations is known in advance.	**(iii).** It is used when the number of iterations is not known in advance.	**(iii).** It is used when the number of iterations is not known in advance.

- ✓ **Write a program to find the sum of digits of an integer number.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
int n,d,s=0;
printf(“Eneter the number”);
scanf(“%d”,&n);
while(n>0)
{
d=n%10;
s=s+d;
n=n/10;
}
printf(“The Sum of digits:=%d”,s);
getch();
}
```

- ✓ **Write a program to find the sum of digit square of an integer number.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
int n,d,s=0;
printf(“Eneter the number”);
scanf(“%d”,&n);
while(n>0)
{
d=n%10;
s=s+d*d;
n=n/10;
}
printf(“The Sum of Digits Square:=%d”,s);
getch();
}
```

- ✓ **Write a program to find the reverse of an integer number.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
int n,d,s=0;
printf(“Eneter the number”);
scanf(“%d”,&n);
while(n>0)
{
d=n%10;
s=s*10+d;
n=n/10;
}
printf(“The Reverse of Number:=%d”,s);
getch();
}
```

- ✓ **Write a program to check whether an integer number is palindrome or not.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
int n,d,s=0,t;
printf("Eneter the number");
scanf("%d",&n);
t=n;
while(n>0)
{
d=n%10;
s=s*10+d;
n=n/10;
}
if(s==t)
printf("Number is Palindrome Number");
else
printf("Number is not Palindrome Number");
getch();
}
```

- ✓ **Write a program to check whether an integer number is Armstrong or not.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
int n,d,s=0,t;
printf("Eneter the number");
scanf("%d",&n);
t=n;
while(n>0)
{
d=n%10;
s=s+d*d*d;
n=n/10;
}
if(s==t)
printf("Number is Armstrong Number");
else
printf("Number is not Armstrong Number");
getch();
}
```

- ✓ **Write a program to check whether an integer number is Prime number or not.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
int n,i,c=0;
printf("Eneter the number");
scanf("%d",&n);
for(i=1;i<=n;i++)
{
if(n%i==0)
c++;
}
if(c==2)
printf("Number is Prime Number");
else
printf("Number is not Prime Numbe");
getch();
}
```

- ✓ **Write a program to print all Prime number between 1 and 100.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
int n,i,c;
for(n=2;n<100;n++)
{
c=0;
for(i=1;i<=n;i++)
{
if(n%i==0)
c++;
}
if(c==2)
printf("%d\t",n);
}
getch();
}
```

Unconditional or Jumping Statements: The jump statements unconditionally transfer the control from one place to another place in a program. The jumping statements are-

1. ***break***
2. ***continue***
3. ***goto***
4. ***return***
5. ***exit***

break Statement: The break statement transfers the control to outside a loop or switch statement.

continue Statement: The continue statement transfers the control to the beginning of loop. It transfer the control to the increment/decrement in for loop and to the condition in while and do while loop.

Difference between break and continue statements

break	continue

1. It is used to exit from switch case or loop.
2. It transfer the control to the first statement after the loop.
 	
3. It is used with loop and switch.
Example:
void main()
{
int i;
for(i=1;i<=10;i++)
{
if(i==5)
break;
printf("%d",i);
}
getch();
}
	1. It is used to go to the beginning of a loop.
	2. It transfer the control to the iteration in for loop and to the condition in while and do while loop.
	3. It is used with loop.
	Example:
	void main()
	{
	int i;
	for(i=1;i<=10;i++)
	{
	if(i==5)
	continue;
	printf("%d",i);
	}
	getch();
	}

goto Statement: The goto statement is used to transfer the control from one place to another in a program. This statement uses a label.

Syntax:

```
Statement-1;
if(condition)
```

```
{
goto label;
}
Statement-2;
label:
Statement-3;
```

*Where **goto** is a keyword and **label** is an identifier (user defined name).*

- ✓ **Write a program to print 1 to 10 numbers using goto statement.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i=1;
start:
printf("%d",i);
if(i<10)
{
i++;
goto start;
}
getch();
}
```

- ✓ **Write a program to calculate the square root of a number.**

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
void main()
{
float n,s;
read:
printf("Enter a number");
scanf("%d",&n);
if(n<0)
goto read;
s=sqrt(n)
printf("The square root of %f=%f",n,s);
getch();
}
```

return: This statement is used to transfer the control from the called function to the calling function with or without result.

exit(): This function is used to exit the control from the program execution.

- ✓ **Write a program to find the sum of following series:**

(i) **$1^1 + 2^2 + 3^3 + 4^4 +$ ----------n^n.**

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
void main()
{
int n,i,s=0;
printf("Eneter the number of terms");
scanf("%d",&n);
for(i=1;i<=n;i++)
{
s=s+pow(i,i);
}
printf("The Sum of the Series:=%d",s);
getch();
}
```

(ii) **$X^1 + X^2 + X^3 + X^4 +$ ----------X^n.**

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
void main()
{
int n,i,s=0;
printf("Eneter the number and range");
scanf("%d%d",&x,&n);
for(i=1;i<=n;i++)
{
s=s+pow(x,i);
}
printf("The Sum of the Series:=%d",s);
getch();
}
```

(iii) **1 + 2 – 3 + 4 – 5---------n.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
int n,i,s=1,sign= -1;
printf("Eneter the range");
scanf("%d",&n);
for(i=2;i<=n;i++)
{
sign=sign*(-1);
s=s+(i*sign);
}
printf("The Sum of the Series:=%d",s);
getch();
}
```

(iv) **$X^1 + X^2 - X^3 + X^4 + X^5$--------- X^n.**

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
void main()
{
int n,i,s,x,sign= -1;
printf("Eneter the range and value of x");
scanf("%d%d",&n,&x);
for(i=2;i<=n;i++)
{
sign=sign*(-1);
s=s+(pow(x,i)*sign);
}
printf("The Sum of the Series:=%d",s);
getch();
}
```

(v) 1/1! + 2/2! + 3/3! + ------------n/n!.

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
void main()
{
int n,i;
float s=0;
printf("Eneter the range");
scanf("%d",&n);
for(i=1;i<=n;i++)
{
s=s+(i/(float)fact(i));
}
printf("The Sum of the Series:=%f",s);
getch();
}
int fact(int n)
{
if(n==0)
return 1;
else
return(n*fact(n-1));
}
```

✓ **Write a program to print the following sequence of integers.**

(i) 0,5,10,15,20,25

```
 #include<stdio.h>
#include<conio.h>
void main()
{
int i;
for(i=0;i<=25;i=i+5)
{
printf("%d,",i);
}
getch();
}
```

(ii) 0,2,4,6,8,10

```
 #include<stdio.h>
#include<conio.h>
void main()
{
int i;
for(i=0;i<=10;i=i+2)
{
printf("%d,",i);
}
getch();
}
```

(iii) 1,2,4,8,16,32

```
 #include<stdio.h>
#include<conio.h>
void main()
{
int i;
for(i=0;i<=32;i=i*2)
{
printf("%d,",i);
}
getch();
}
```

(iv) -8 -6 -4 -2 0 2 4 6 8

```
 #include<stdio.h>
#include<conio.h>
void main()
{
int i;
for(i=-8;i<=8;i=i+2)
{
printf("%d",i);
}
getch();
}
```

Nested Loop: A loop which is inside the body of another loop is called nested loop.

- ✓ **Write a program to print following patterns.**

(i).

```
*
*       *
*       *       *
*       *       *       *
*       *       *       *       *
```

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,j;
for(i=1;i<=5;i++)
{
for(j=1;j<=i;j++)
{
printf("*");
}
printf("\n");
}
getch();
}
```

(ii).

```
A
B       A
A       B       A
B       A       B       A
A       B       A       B       A
```

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,j;
for(i=1;i<=5;i++)
{
for(j=1;j<=i;j++)
{
if((i+j)%2==0)
printf("A");
else
printf("B");
}
printf("\n");
}
getch();
}
```

(iii). 1

```
2       2
3       3       3
4       4       4       4
5       5       5       5       5
```

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,j;
for(i=1;i<=5;i++)
{
for(j=1;j<=i;j++)
{
printf("%d\t",i);
}
printf("\n");
}
getch();
}
```

(iv).5

```
5       4
5       4       3
5       4       3       2
5       4       3       2       1
```

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,j;
for(i=5;i>=1;i--)
{
for(j=1;j<=i;j++)
{
printf("%d\t",i);
}
printf("\n");
}
getch();
}
```

(v).

```
        *
      * * *
    * * * * *
  * * * * * * *
* * * * * * * * *
```

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,j,k;
for(i=1;i<=5;i++)
{
for(j=5;j>i;j--)
printf(“ ”);
for(k=1;k<=2*i-1;k++)
printf(“*”);
printf(“\n”);
}
getch();
}
```

List of programs to practice (Home work)

1. **Write a program to display numbers 1 to 10.**
2. **Write a program to display all even numbers from 1 to 20**
3. **Write a program to display all odd numbers from 1 to 20**
4. **Write a program to print all the Numbers Divisible by 7 from 1 to 100.**
5. **Write a program to print table of 2.**
6. **Write a program to print table of 5.**
7. **Write a program to print table of any number.**
8. **Write a program to print table of 5 in following format.**

5 X 1 = 5

5 X 2 = 10

5 X 3 = 15

9. **Write a program to Find the Sum of first 50 Natural Numbers using for Loop.**
10. **Write a program to calculate factorial of a given number using for loop.**
11. **Write a program to calculate factorial of a given number using while loop.**
12. **Write a program to calculate factorial of a given number using do-while loop.**
13. **Write a program to count the sum of digits in the entered number.**
14. **Write a program to find the reverse of a given number.**
15. **Write a program to Check whether a given Number is Perfect Number.**
16. **Write a program to Print Armstrong Number from 1 to 1000.**

17. Write a program to Compute the Value of X ^ N

18. Write a program to Calculate the value of nCr

19. Write a program to generate the Fibonacci Series

20. Write a program to Print First 10 Natural Numbers

21. Write a program to check whether a given Number is Palindrome or Not

22. Write a program to Check whether a given Number is an Armstrong Number

23. Write a program to Check Whether given Number is Perfect or Not

24. Write a program to check weather a given number is prime number or not.

25. Write a program to print all prime numbers from 50-500

26. Write a program to find the Sum of all prime numbers from 1-1000

27. Write a program to display the following pattern:

```
* * * * *

* * * * *

* * * * *

* * * * *

* * * * *
```

28. Write a program to display the following pattern:

```
*

* *

* * *

* * * *

* * * * *
```

29. Write a program to display the following pattern:

```
1

1 2

1 2 3

1 2 3 4
```

1 2 3 4 5

30. Write a program to display the following pattern:

```
1

2 2

3 3 3

4 4 4 4

5 5 5 5 5
```

31. Write a program to display the following pattern:

```
A

B B

C C C

D D D D

E E E E E
```

32. Write a program to display the following pattern:

```
* * * * *

* * * *

* * *

* *

*
```

33. Write a program to display the following pattern:

1 2 3 4 5

1 2 3 4

1 2 3

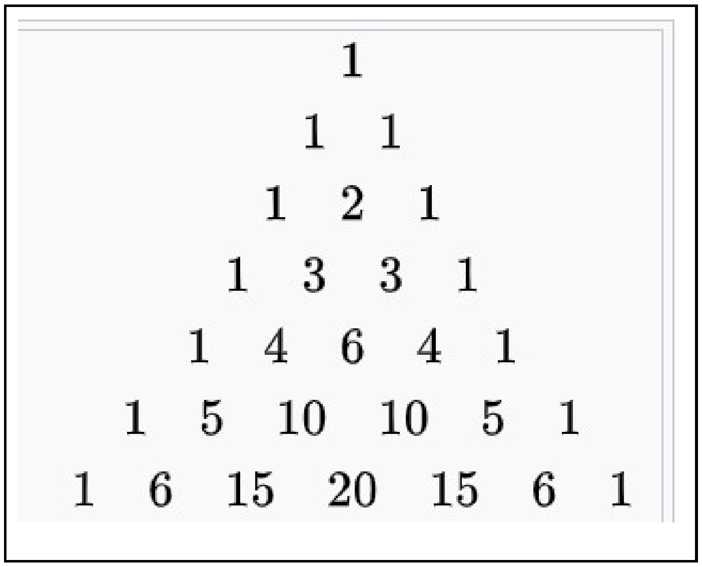
1 2

1

34. Write a program to display the following pattern:

```
*

* * *

* * * * *

* * * * * * *

* * * * * * * * *
```

35. Write a program to display the following pattern (Pascal Triangle):



```
            1
          1   1
        1   2   1
      1   3   3   1
    1   4   6   4   1
  1   5   10  10  5   1
1   6   15  20  15  6   1
```

41. Write a program to display the following pattern:

1

2 3

```
4 5 6
7 8 9 10
```

36. Write a program to display the following pattern:

```
A
B A B
A B A B A
B A B A B A B
```

37. Write a program to display the following pattern:

```
1
0 1 0
1 0 1 0 1
0 1 0 1 0 1 0
```

38. Write a program to display the following pattern:

```
A B C D E F G F E D C B A
A B C D E F   F E D C B A
A B C D E       E D C B A
A B C D           D C B A
A B C               C B A
A B                   B A
A                       A
```

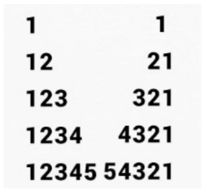
39. Write a program to display the following pattern:

```
**********
****  ****
***    ***
**      **
*        *
```

40. Write a program to display the following pattern:

```
0
01
010
0101
01010
```

41. Write a program to display the following pattern:



```
1           1
12         21
123       321
1234     4321
12345 54321
```

42. Write a program to display the following pattern:

```
A
B C
D E F
G H I J
K L M N O
```

43. Write a program to display the following pattern:

```
A
BAB
CBABC
DCBABCD
EDCBABCDE
```

44. Write a program to display the following pattern:

```
1A2B3C4D5E
1A2B3C4D
1A2B3C
1A2B
1A
```

45. Write a program to display the following pattern:

```
0
1 0 1
2 1 0 1 2
3 2 1 0 1 2 3
4 3 2 1 0 1 2 3 4
```

46. Write a program to print the following sequence of integers.

0, 5, 10, 15, 20, 25

47. Write a program to print the following sequence of integers.

0, 2, 4, 6, 8, 10

48. Write a program to Find the Sum of A.P Series.

49. Write a program to Find the Sum of G.P Series.

50. Write a program to Find the Sum of H.P Series.

51. Write a program to print the following sequence of integers.

1, 2, 4, 8, 16, 32

52. Write a program to print the following sequence of integers.

-8, -6, -4, -2, 0, 2, 4, 6, 8

53. Write a program to find the Sum of following Series:

1 + 2 + 3 + 4 + 5 + ... + n

54. Write a program to find the Sum of following Series:

(1*1) + (2*2) + (3*3) + (4*4) + (5*5) + ... + (n*n)

55. Write a program to find the Sum of following Series:

(1) + (1+2) + (1+2+3) + (1+2+3+4) + ... + (1+2+3+4+...+n)

56. Write a program to find the Sum of following Series:

1! + 2! + 3! + 4! + 5! + ... + n!

57. Write a program to find the Sum of following Series:

(1^1) + (2^2) + (3^3) + (4^4) + (5^5) + ... + (n^n)

58. Write a program to find the Sum of following Series:

(1!/1) + (2!/2) + (3!/3) + (4!/4) + (5!/5) + ... + (n!/n)

59. Write a program to find the Sum of following Series:

[(1^1)/1] + [(2^2)/2] + [(3^3)/3] + [(4^4)/4] + [(5^5)/5] + ... + [(n^n)/n]

60. Write a program to find the Sum of following Series:

[(1^1)/1!] + [(2^2)/2!] + [(3^3)/3!] + [(4^4)/4!] + [(5^5)/5!] + ... + [(n^n)/n!]

61. Write a program to find the Sum of following Series:

1/2 - 2/3 + 3/4 - 4/5 + 5/6 - ...... upto n terms

62. Write a program to print the following Series:

1, 2, 3, 6, 9, 18, 27, 54, ... upto n terms

63. Write a program to print the following Series:

2, 15, 41, 80, 132, 197, 275, 366, 470, 587

64. Write a program to print the following Series:

1, 3, 8, 15, 27, 50, 92, 169, 311

Day_3 C-Programming

(RECAP OF PREVIOUS DAY)

Sub Programming, function, types of functions, passing parameters to functions: call by value Recursion: Definition, Types of recursive functions, Tower of Hanoi problem, scope of variable, local and global variables, Nesting of Scope Storage classes: Auto, Register, Static and Extern,

Modular programming: Dividing the program in to sub programs (modules/function) to achieve the given task is modular approach. More generic functions definition gives the ability to re-use the functions, such as built-in library functions.

Function (Modules): A function is a block of statements to perform a specific task. Functions are the subprograms. There are two types of functions in C programming.

- **Library Functions (Pre-Defined Functions/ Built-in Functions)**
- **User Defined Functions (Programmer Defined Functions)**

Library Functions: The functions that are defined in C library are known as library functions. Examples are- *printf, scanf, getch, strlen.*

User Defined Functions: The functions that are defined by the programmer are known as user defined function.

There are following three components associated with functions:

- Function Declaration (Function Prototype)
- Function Definition
- Function Call

Function Declaration:

Syntax: *return-type function-name(parameter-list);*

Function Definition:

Syntax: *return-type function-name(parameter-list)*

```
{
Statements;
}
```

Function Call:

Syntax: *function-name(parameter-list);*

Advantages of using multiple functions:

(i) Functions reduce the length and complexity of programs.
(ii) Functions are useful when the problem is very complex.
(iii) It is easy to find the errors in functions.
(iv) Functions can be reused in programs.

Actual Parameters (Arguments): The parameters that are passed from the calling function to the called function are known as actual parameters.

Formal Parameters (Arguments): The parameters that hold the value of actual parameters in function definition are known as formal parameters.

Types of functions: There are following types of functions.

(i). *Function with arguments and with return value*
(ii). *Function with arguments and without return value*
(iii).*Function without argument and with return value*
(iv).*Function without argument and with return value*

(i) Function with arguments and with return value

```
#include<stdio.h>
#include<conio.h>
int sum(int, int);
void main()
{
int a,b,s;
printf("Eneter two Numbers");
scanf("%d%d",&a,&b);
s=sum(a,b);
printf("Addition=%d",s);
getch();
}
int sum(int x, int y)
{
int z;
z=x+y;
return z;
}
```

(ii) Function with arguments and without return value

```
#include<stdio.h>
#include<conio.h>
void sum(int, int);
void main()
{
int a,b;
printf("Eneter two Numbers");
scanf("%d%d",&a,&b);
sum(a,b);
getch();
}
void sum(int x, int y)
{
int z;
z=x+y;
printf("Addition=%d",z);
}
```

(iii) Function without argument and with return value

```
#include<stdio.h>
#include<conio.h>
int sum();
void main()
{
int s;
s=sum();
printf("Addition=%d",s);
getch();
}
int sum()
{
int x,y,z;
printf("Enter Two Numbers");
scanf("%d%d",&x,&y);
z=x+y;
return z;
}
```

(iv) Function without argument and with return value

```
#include<stdio.h>
#include<conio.h>
void sum();
void main()
{
sum();
getch();
}
void sum()
{
int x,y,z;
printf("Enter Two Numbers");
scanf("%d%d",&x,&y);
z=x+y;
printf("Addition=%d",z);
}
```

✓ **Write a program to calculate the factorial of a number using function**

```
#include<stdio.h>
#include<conio.h>
int fact(int);
void main()
{
int n,f;
printf("Enter a Number");
scanf("%d",&n);
f=fact(n);
printf("Factorial=%d",f);
getch();
}
int fact(int num)
{
```

```
int i,x=1;
for(i=1;i<=num;i++)
x=x*i;
return x;
}
```

Types of Function Call (Parameter passing Mechanism or Parameter Passing Methods):

There are two types of function call

(i) **Call by value (Pass by value)**
(ii) **Call by reference (Pass by reference)**

Call by value (Pass by value): In this method the value of the variable is passed from the calling function to the called function.

```
#include<stdio.h>
#include<conio.h>
void swap(int, int);
void main()
{
int a=10,b=20;
swap(a,b);
printf("The value of Actual Arguments: a=%d, b=%d",a,b);
getch();
}

void swap(int x, int y)
{
int t;
t=x;
x=y;
y=t;
printf("The value of Formal Arguments: x=%d, y=%d",x,y);
}
```

Call by Reference (Pass by Reference): In this method the address of the variable is passed from the calling function to the called function.

```
#include<stdio.h>
#include<conio.h>
void swap(int *, int *);
void main()
{
int a=10,b=20;
swap(&a,&b);
printf("The value of Actual Arguments: a=%d, b=%d",a,b);
getch();
}
void swap(int *x, int *y)
{
int t;
t=*x;
*x=*y;
*y=t;
printf("The value of Formal Arguments: x=%d, y=%d",*x,*y);
}
```

Difference between Call by Value and Call by Reference

Call by Value	Call by Reference

1. In this method, the value of variable is passed from the calling function to the called function.
2. Any change in formal parameters will not reflect in actual parameters.
3. This method is slow.
Example:
#include<stdio.h>
#include<conio.h>
void swap(int,int);
void main()
{
int a,b;
printf("Enter two Number");
scanf("%d%d",&a,&b);
swap(a,b);
printf("Value of actual parameters=%d,%d",a,b);
getch();
}
void swap(int x,int y)
{
int t;
t=x;
x=y;
y=t;
printf("Value of formal parameters=%d,%d",x,y);
}
	1. In this method, the address of variable is passed from the calling function to the called function.
	2. Any change in formal parameters will reflect in actual parameters.
	3. This method is fast.
	Example:
	#include<stdio.h>
	#include<conio.h>
	void swap(int,int*);*
	void main()
	{
	int a,b;
	printf("Enter two Number");
	scanf("%d%d",&a,&b);
	swap(&a,&b);
	printf("Value of actual parameters=%d,%d",a,b);
	getch();
	}
	*void swap(int *x,int *y)*
	{
	int t;
	*t=*x;*
	**x=*y;*
	**y=t;*
	*printf("Value of formal parameters=%d,%d",*x,*y);*
	}

Recursion

Definition

Recursion is a process where a function calls itself directly or indirectly to solve a larger problem by breaking it into smaller sub-problems.

Types of Recursive Functions

1. **Direct Recursion: A function directly calls itself.**

Example:

```
#include <stdio.h>

void printNumbers(int n) {
  if (n > 0) {
        printf("%d\n", n);
        printNumbers(n - 1); // Function calls itself
    }
}

int main() {
    printNumbers(5);
    return 0;
}
```

2. **Indirect Recursion:** A function calls another function, which in turn calls the first function.

Example:

```
#include <stdio.h>

void functionA(int n);
void functionB(int n);

void functionA(int n) {
    if (n > 0) {
        printf("%d\n", n);
        functionB(n - 1);
    }
}

void functionB(int n) {
    if (n > 0) {
        printf("%d\n", n);
        functionA(n - 1);
    }
}

int main() {
    functionA(5);
    return 0;
}
```

✓ **Write a program to calculate the factorial of a number using recursion**

```
#include<stdio.h>
#include<conio.h>
int fact(int);
void main()
{
int n,f;
printf("Enter a Number");
scanf("%d",&n);
f=fact(n);
printf("Factorial=%d",f);
getch();
}
int fact(int x)
{
if(x==0)
return (1);
else
return (x*fact(x-1));
}
```

✓ **Write a program to display the Fibonacci series using recursion.**

```
#include<stdio.h>
#include<conio.h>
int rec(int);
void main()
{
int i,n;
printf("Enter the Limit");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("%d\t",rec(i));
}
getch();
}
int rec(int x)
{
if(x==0)
return (0);
else if(x==1)
return (1);
else
return (rec(x-1)+rec(x-2));
}
```

Tower of Hanoi Problem

The Tower of Hanoi is a classic problem that demonstrates recursion. The problem involves moving disks from one rod to another, following these rules:

1. Only one disk can be moved at a time.
2. A larger disk cannot be placed on top of a smaller disk.
3. Disks can only be moved between three rods.

Solution Using Recursion:

```
#include <stdio.h>

void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod) {
  if (n == 1) {
    printf("Move disk 1 from %c to %c\n", from_rod, to_rod);
    return;
  }
```

```
    towerOfHanoi(n - 1, from_rod, aux_rod, to_rod);
    printf("Move disk %d from %c to %c\n", n, from_rod, to_rod);
    towerOfHanoi(n - 1, aux_rod, to_rod, from_rod);
}

int main() {
    int n = 3; // Number of disks
    towerOfHanoi(n, 'A', 'C', 'B'); // A, B, and C are the rod names
    return 0;
}
```

Output for n = 3:

```
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
```

---

2. Scope of Variables

Local Variables

- Variables declared inside a function are local to that function.
- They can only be accessed within the function where they are defined.

Example:

```
#include <stdio.h>

void test() {
    int localVar = 10;
    printf("Local variable: %d\n", localVar);
}

int main() {
    test();
    // printf("%d", localVar); // Error: localVar is not accessible here
    return 0;
}
```

Global Variables

- Variables declared outside all functions are global.
- They can be accessed by any function in the program.

Example:

```
#include <stdio.h>

int globalVar = 20; // Global variable

void test() {
    printf("Global variable: %d\n", globalVar);
}

int main() {
    test();
    printf("Global variable in main: %d\n", globalVar);
    return 0;
}
```

Nesting of Scope

- Inner blocks can access variables of outer blocks.
- Inner variables with the same name as outer variables will shadow the outer variables**.**

Example:

```
#include <stdio.h>

int main() {
    int x = 10;

    {
        int x = 20; // Shadows the outer x
        printf("Inner block x: %d\n", x);
    }

    printf("Outer block x: %d\n", x);
    return 0;
}
```

Output:

```
Inner block x: 20
Outer block x: 10
```

---

3. Storage Classes

Storage classes in C define the scope, lifetime, and visibility of variables.

In C, variables have three important properties:

1. **Scope:** Where the variable is accessible in the program.
2. **Lifetime:** How long the variable retains its value in memory.
3. **Visibility:** Which part of the program can see (use) the variable.

Types of Storage Classes

1. **Auto**
    - **Default storage class for local variables.**
    - **Scope: Local to the block in which it is defined (default for all local variables).**
    - **Lifetime: Limited to the execution of the block where it is defined.**
    - **Visibility: Not visible outside the block**

Example:

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

```
void test() {
   auto int a = 10; // Auto is implicit
   printf("Auto variable: %d\n", a);
}

int main() {
   test();
   return 0;
}
```

2. **Register**
    - **Scope: Local to the block in which it is defined.**
    - **Lifetime: Limited to the execution of the block where it is defined.**
    - **Visibility: Not visible outside the block.**
    - **Special Feature: Stored in CPU registers (if available) for faster access. Address-of operator (&) cannot be used.**

Example:

```
#include <stdio.h>

int main() {
   register int x = 10;
   printf("Register variable: %d\n", x);
   return 0;
}
```

3. **Static**
    - **Retains its value between function calls.**
    - **Scope: Local to the block in which it is defined.**
    - **Lifetime: Retains its value between function calls (entire program execution).**
    - **Visibility: Not visible outside the block (local static variable).**

Example:

```
#include <stdio.h>

void test() {
   static int count = 0;
   count++;
   printf("Static variable count: %d\n", count);
}

int main() {
   test();
   test();
   test();
   return 0;
}
```

Output:

Static variable count: 1
Static variable count: 2
Static variable count: 3

4. **Extern**
    - **Used to declare a global variable in another file.**
    - **Scope: Global across all files where it is declared.**
    - **Lifetime: Entire program.**
    - **Visibility: Visible in all files that declare it using the extern keyword.**

Example (Two files):

File1.c:

```
#include <stdio.h>

extern int globalVar;

void display() {
   printf("Extern variable: %d\n", globalVar);
}
```

File2.c:

```
#include <stdio.h>

int globalVar = 50;

void display();

int main() {
   display();
   return 0;
}
```

Practice Problems

1. **Write a program to calculate the factorial of a number using recursion.**
2. **Implement the Tower of Hanoi problem for n disks.**
3. **Demonstrate the use of local and global variables in a program.**
4. **Write a program to demonstrate the difference between auto and static storage classes.**
5. **Implement a program to calculate the Fibonacci series using recursion.**
6. **Demonstrate the use of extern variables across two files.**
7. **Write a program to calculate the sum of digits of a number using recursion.**
8. **Demonstrate the use of register variables in a program.**
9. **Implement nested scopes and show variable shadowing.**
10. **Write a program to reverse a string using recursion.**
11. **Write a program to find out G.C.D. of two numbers using recursion.**

Day_5 C-Programming

(RECAP OF PREVIOUS DAY)

Pointer,pointer arithmetic and scaling, Pointer Aliasing. call by reference.

Array(one dimensional),array using pointers,manipulating array elements, Linear Search,Binary Search, Bubble Sort.

Pointers

Definition

A pointer is a variable that stores the memory address of another variable. Pointers allow for efficient memory management and enable dynamic memory allocation, passing data by reference, and creating complex data structures like linked lists and trees.

· Pointers are nothing but memory addresses.

· Every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory.

· A pointer is a variable that contains the memory address of another variable.

Declaration and Initialization

- To declare a pointer, use the * operator.
- To initialize a pointer, assign it the address of a variable using the & operator.

Syntax:

```
<datatype> *pointerName;
```

Example:

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

```
int main() {

    int x = 10;

    int *ptr = &x; // Pointer stores the address of x


    printf("Value of x: %d\n", x);

    printf("Address of x: %p\n", &x);

    printf("Pointer ptr points to address: %p\n", ptr);

    printf("Value at address stored in ptr: %d\n", *ptr); // Dereferencing


    return 0;

}
```

Output:

```
Value of x: 10

Address of x: 0x7ffee1a4c98c

Pointer ptr points to address: 0x7ffee1a4c98c

Value at address stored in ptr: 10
```

Pointer Arithmetic and Scaling

Pointer arithmetic involves operations like addition and subtraction on pointers. When performing arithmetic, the size of the data type being pointed to determines how much the pointer moves in memory.

Valid Operations:

1. Increment (`ptr++`) and decrement (`ptr--`) pointers.
2. Add/subtract integers to/from pointers (`ptr + n`, `ptr - n`).
3. Subtract one pointer from another to calculate the number of elements between them.

Example:

```
#include <stdio.h>


int main() {

    int arr[5] = {10, 20, 30, 40, 50};

    int *ptr = arr; // Points to the first element

    printf("Pointer arithmetic:\n");

    printf("Value at ptr: %d\n", *ptr);

    ptr++;

    printf("Value at ptr++: %d\n", *ptr);

    ptr += 2;

    printf("Value at ptr+2: %d\n", *ptr);

    return 0;

}
```

Output:

```
Pointer arithmetic:

Value at ptr: 10

Value at ptr++: 20

Value at ptr+2: 40
```

Scaling

- The pointer moves in increments of the size of the data type.
- For example, if `int` is 4 bytes, `ptr++` moves the pointer by 4 bytes.

Pointer Aliasing

Pointer aliasing occurs when two or more pointers point to the same memory location. Modifying the data through one pointer affects the data accessed through the other pointer(s).

Example:

```
#include <stdio.h>

int main() {

    int x = 100;

    int *ptr1 = &x;

    int *ptr2 = &x; // Alias of ptr1

    printf("Value of x using ptr1: %d\n", *ptr1);

    printf("Value of x using ptr2: %d\n", *ptr2);

    *ptr1 = 200; // Modifying x using ptr1

    printf("Value of x after modification using ptr1: %d\n", *ptr2);

    return 0;

}
```

Output:

```
Value of x using ptr1: 100

Value of x using ptr2: 100
Value of x after modification using ptr1: 200
```

Call by Reference

In the call by reference method, the function receives the address of the actual parameter. Changes made to the parameter inside the function affect the original value.

Example:

```
#include <stdio.h>

void swap(int *a, int *b) {

    int temp = *a;

    *a = *b;

    *b = temp;

}


int main() {

    int x = 10, y = 20;

    printf("Before swapping: x = %d, y = %d\n", x, y);

    swap(&x, &y);

    printf("After swapping: x = %d, y = %d\n", x, y);

    return 0;

}
```

Output:

```
Before swapping: x = 10, y = 20

After swapping: x = 20, y = 10
```

One-Dimensional Array

Definition

An array is a collection of elements of the same data type stored in contiguous memory locations. Arrays in C are zero-indexed, meaning the first element is at index 0.

Declaration and Initialization

Syntax:

```
<datatype> arrayName[size];
```

Example:

```
#include <stdio.h>

int main() {

    int arr[5] = {10, 20, 30, 40, 50}; // Declaration and initialization

    printf("Elements of the array:\n");

    for (int i = 0; i < 5; i++) {

        printf("arr[%d] = %d\n", i, arr[i]);

    }

    return 0;

}
```

Output:

```
Elements of the array:

arr[0] = 10

arr[1] = 20

arr[2] = 30

arr[3] = 40

arr[4] = 50
```

Array Using Pointers

Accessing Array Elements Using Pointers

Array elements can be accessed using pointers because the array name represents the address of the first element.

Example:

```
#include <stdio.h>

int main() {

   int arr[5] = {10, 20, 30, 40, 50};

   int *ptr = arr; // Pointer to the first element
```

```
    printf("Accessing array elements using pointers:\n");

    for (int i = 0; i < 5; i++) {

        printf("Element %d: %d\n", i, *(ptr + i));

    }

    return 0;

}
```

Output:

```
Accessing array elements using pointers:

Element 0: 10

Element 1: 20

Element 2: 30

Element 3: 40

Element 4: 50
```

Manipulating Array Elements

Array elements can be modified using direct indexing or pointer arithmetic.

Example:

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

```
int main() {

    int arr[5] = {1, 2, 3, 4, 5};

    printf("Original array:\n");

    for (int i = 0; i < 5; i++) {

        printf("arr[%d] = %d\n", i, arr[i]);

    }

    // Modify elements

    for (int i = 0; i < 5; i++) {

        arr[i] *= 2;

    }

    printf("Modified array:\n");

    for (int i = 0; i < 5; i++) {

        printf("arr[%d] = %d\n", i, arr[i]);

    }

    return 0;

}
```

Output:

```
Original array:
```

```
arr[0] = 1

arr[1] = 2

arr[2] = 3

arr[3] = 4

arr[4] = 5


Modified array:

arr[0] = 2

arr[1] = 4

arr[2] = 6

arr[3] = 8

arr[4] = 10
```

Linear Search

Linear search involves searching for an element by checking each array element sequentially.

Algorithm:

1. Start from the first element.
2. Compare each element with the target value.
3. Return the index if found, or return -1 if not found.

Example:

```
#include <stdio.h>

int linearSearch(int arr[], int size, int key) {

    for (int i = 0; i < size; i++) {

        if (arr[i] == key) {

            return i; // Return the index

        }

    }

    return -1; // Not found

}



int main() {

    int arr[5] = {10, 20, 30, 40, 50};

    int key = 30;

    int result = linearSearch(arr, 5, key);

    if (result != -1) {

        printf("Element %d found at index %d\n", key, result);

    } else {
        printf("Element %d not found\n", key);
    }
    return 0;
}
```

Output:

```
Element 30 found at index 2
```

Binary Search

Binary search is an efficient search algorithm for sorted arrays. It repeatedly divides the search interval in half.

Algorithm:

1. Compare the middle element with the target.
2. If equal, return the index.
3. If smaller, search the right half; if larger, search the left half.
4. Repeat until the target is found or the interval is empty.

Example:

```
#include <stdio.h>

int binarySearch(int arr[], int size, int key)

{
    int low = 0, high = size - 1;

    while (low <= high) {

        int mid = (low + high) / 2;

        if (arr[mid] == key) {

            return mid;

        } else if (arr[mid] < key) {

            low = mid + 1;

        } else {

            high = mid - 1;

        }

    }

    return -1; // Not found

}



int main() {

    int arr[5] = {10, 20, 30, 40, 50};

    int key = 40;

    int result = binarySearch(arr, 5, key);
    if (result != -1) {

        printf("Element %d found at index %d\n", key, result);

    } else {

        printf("Element %d not found\n", key);

    }

    return 0;

}
```

Output:

```
Element 40 found at index 3
```

Bubble Sort

Bubble Sort is a simple sorting algorithm that repeatedly swaps adjacent elements if they are in the wrong order.

Algorithm:

1. Compare adjacent elements and swap if needed.
2. Repeat for all elements, reducing the range after each pass.

Example:

```
#include <stdio.h>

void bubbleSort(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

int main() {
    int arr[5] = {50, 20, 40, 10, 30};
    printf("Original array:\n");
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    bubbleSort(arr, 5);

    printf("Sorted array:\n");

    for (int i = 0; i < 5; i++) {

        printf("%d ", arr[i]);

    }

    printf("\n");

    return 0;

}
```

Output:

```
Original array:

50 20 40 10 30

Sorted array:

10 20 30 40 50
```

Practice Problems

1. Write a program to find the maximum and minimum elements in an array using pointers.
2. Implement linear search to count the occurrences of a target value in an array.
3. Write a program to reverse an array using pointers.
4. Implement binary search on a user-input sorted array.
5. Modify the bubble sort algorithm to sort in descending order.
6. Write a program to find the sum and average of array elements using pointers.
7. Implement a program to merge two sorted arrays into one sorted array.
8. Create a program to find the second largest element in an array.
9. Implement bubble sort using a while loop instead of nested for loops.
10. Write a program to count the number of even and odd elements in an array.
11. Write a program to find the largest of three numbers using call by reference.
12. Implement pointer arithmetic to traverse an array and find the sum of its elements.
13. Demonstrate pointer aliasing by swapping two numbers without using a temporary variable.
14. Write a program to reverse a string using pointers.
15. Implement a function to find the factorial of a number using call by reference.
16. Demonstrate the difference between pointer arithmetic on int and char types.
17. Write a program to dynamically allocate memory for an array and find its average using pointers.
18. Implement matrix multiplication using pointers.
19. Write a program to compare two strings using pointers.
20. Implement a program to find the length of a string using pointer arithmetic.

Day_6 C-Programming

(RECAP OF PREVIOUS DAY)

Array(Two dimensional), its uses in matrix computation(Addition, multiplication,sum of diagonal elements, transpose). Passing array to function.

Two-Dimensional Arrays

Definition

A two-dimensional array is an array of arrays. It is used to store data in a tabular format, where data is organized in rows and columns.

Declaration and Initialization

Syntax:

<datatype> arrayName[rows][columns];

Example:

```
#include <stdio.h>

int main() {

    int matrix[2][3] = {

        {1, 2, 3},

        {4, 5, 6}

    };
    printf("Two-dimensional array elements:\n");

    for (int i = 0; i < 2; i++) {

        for (int j = 0; j < 3; j++) {

            printf("%d ", matrix[i][j]);

        }

        printf("\n");

    }

    return 0;

}
```

Output:

```
Two-dimensional array elements:

1 2 3

4 5 6
```

Uses in Matrix Computation

Matrix Addition

Addition of two matrices involves adding corresponding elements of the matrices.

Example:

```
#include <stdio.h>

int main() {

    int A[2][2] = {{1, 2}, {3, 4}};

    int B[2][2] = {{5, 6}, {7, 8}};

    int C[2][2];

    printf("Matrix Addition:\n");

    for (int i = 0; i < 2; i++) {

        for (int j = 0; j < 2; j++) {

            C[i][j] = A[i][j] + B[i][j];

            printf("%d ", C[i][j]);

        }

        printf("\n");

    }

    return 0;

}
```

Output:

```
Matrix Addition:
```

```
6 8

10 12
```

Matrix Multiplication

Matrix multiplication involves taking the dot product of rows and columns.

Example:

```
#include <stdio.h>

int main() {

    int A[2][2] = {{1, 2}, {3, 4}};

    int B[2][2] = {{5, 6}, {7, 8}};

    int C[2][2] = {0};

    printf("Matrix Multiplication:\n");

    for (int i = 0; i < 2; i++) {

        for (int j = 0; j < 2; j++) {

            for (int k = 0; k < 2; k++) {

                C[i][j] += A[i][k] * B[k][j];

            }

            printf("%d ", C[i][j]);

        }
        printf("\n");
    }
    return 0;
}
```

Output:

```
Matrix Multiplication:
19 22
43 50
```

Sum of Diagonal Elements

Diagonal elements are those where the row index equals the column index.

Example:

```
#include <stdio.h>
int main() {
    int matrix[3][3] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };

    int sum = 0;

    for (int i = 0; i < 3; i++) {

        sum += matrix[i][i]; // Primary diagonal

    }


    printf("Sum of diagonal elements: %d\n", sum);

    return 0;

}
```

Output:

```
Sum of diagonal elements: 15
```

Matrix Transpose

Transpose of a matrix is obtained by interchanging its rows and columns.

Example:

```
#include <stdio.h>

int main() {

    int matrix[2][3] = {
```

```
    {1, 2, 3},

    {4, 5, 6}

};

int transpose[3][2];

for (int i = 0; i < 2; i++) {

    for (int j = 0; j < 3; j++) {

        transpose[j][i] = matrix[i][j];

    }

}



printf("Transpose of the matrix:\n");

for (int i = 0; i < 3; i++) {

    for (int j = 0; j < 2; j++) {

        printf("%d ", transpose[i][j]);

    }

    printf("\n");

}
```

```
    return 0;

}
```

Output:

```
Transpose of the matrix:

1 4

2 5

3 6
```

Passing Arrays to Functions

Two-dimensional arrays can be passed to functions as arguments by specifying the size of the second dimension.

Example:

```
#include <stdio.h>

void printMatrix(int matrix[2][3], int rows, int cols) {

    printf("Matrix:\n");

    for (int i = 0; i < rows; i++) {

        for (int j = 0; j < cols; j++) {

            printf("%d ", matrix[i][j]);

        }
        printf("\n");
    }
}


int main() {
    int matrix[2][3] = {
        {1, 2, 3},
        {4, 5, 6}
    };
    printMatrix(matrix, 2, 3);
    return 0;
}
```

Output:

```
Matrix:
1 2 3
4 5 6
```

Practice Problems

1. Write a program to find the trace (sum of diagonal elements) of a 4x4 matrix.
2. Implement a function to compute the determinant of a 3x3 matrix.
3. Write a program to check if a given matrix is symmetric.
4. Implement addition and subtraction of two matrices using functions.
5. Create a program to find the row-wise and column-wise sum of a matrix.
6. Write a program to rotate a square matrix 90 degrees clockwise.
7. Implement a function to multiply two matrices of size MxN and NxP.
8. Write a program to find the largest element in each row of a matrix..
9. Write a program to check if two matrices are equal.
10. Write a program to find addition of Lower Triangular Elements in a Matrix.
11. Write a program to calculate sum of Upper Triangular Elements.

Day_7 C-Programming

(RECAP OF PREVIOUS DAY)
Strings: Introduction, initializing strings, accessing string elements, Array of strings, Passing strings to functions, String functions like strlen, strcat, strcmp, strcpy, strrev and other string functions.

1. Introduction to Strings

- A string in C is a one-dimensional array of characters terminated by a null character (`\0`).
- Strings are used to store text data.

Example:

```
char str[] = "Hello, World!";
```

Here:

- `str` is a string of size 14 (13 characters + 1 null character `\0`).

---

2. Initializing Strings

Strings can be initialized in multiple ways:

Method 1: Using a String Literal

```
char str1[] = "Hello";
```

Method 2: Character by Character

```
char str2[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

Method 3: Using `scanf` or `gets` (deprecated)

```
char str3[20];
scanf("%s", str3);  // Note: Stops reading at whitespace
```

---

3. Accessing String Elements

- Strings are arrays, so elements can be accessed using indices.

Example:

```
char str[] = "C Programming";
printf("First character: %c\n", str[0]);  // Output: C
printf("Fifth character: %c\n", str[4]);  // Output: r
```

---

4. Array of Strings

An array of strings is a 2D array where each row represents a string.

Example:

```
#include <stdio.h>

int main() {
    char colors[3][10] = {"Red", "Green", "Blue"};
    for (int i = 0; i < 3; i++) {
        printf("Color %d: %s\n", i + 1, colors[i]);
    }
    return 0;
}
```

Output:

```
Color 1: Red
Color 2: Green
Color 3: Blue
```

---

5. Passing Strings to Functions

Strings can be passed to functions as arguments.

Example:

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

```
void printString(char str[]) {
    printf("String: %s\n", str);
}

int main() {
    char message[] = "Hello, Functions!";
    printString(message);
    return 0;
}
```

Output:

```
String: Hello, Functions!
```

---

6. Common String Functions in C

The `<string.h>` library provides various functions for string manipulation.

1. `strlen`: Calculate the length of a string

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

int main() {
    char str[] = "C Programming";
    printf("Length: %zu\n", strlen(str));  // Output: 13
    return 0;
}
```

2. `strcpy`: Copy one string to another

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

int main() {
    char source[] = "Hello";
    char destination[10];
    strcpy(destination, source);
    printf("Copied String: %s\n", destination);  // Output: Hello
    return 0;
}
```

3. `strcat`: Concatenate two strings

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

int main() {
    char str1[20] = "Hello";
    char str2[] = " World";
    strcat(str1, str2);
    printf("Concatenated String: %s\n", str1);  // Output: Hello World
    return 0;
}
```

4. `strcmp`: Compare two strings

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

int main() {
    char str1[] = "Apple";
    char str2[] = "Orange";
    int result = strcmp(str1, str2);
    if (result == 0)
        printf("Strings are equal\n");
    else if (result < 0)
        printf("%s comes before %s\n", str1, str2);
    else
        printf("%s comes after %s\n", str1, str2);
    return 0;
}
```

5. `strrev`: Reverse a string

`strrev` is not part of the standard C library but is available in some compilers (e.g., Turbo C). You can implement it yourself:

```c
#include <stdio.h>
#include <string.h>

void reverseString(char str[]) {
    int n = strlen(str);
    for (int i = 0; i < n / 2; i++) {
        char temp = str[i];
        str[i] = str[n - i - 1];
        str[n - i - 1] = temp;
    }
}

int main() {
    char str[] = "Reverse Me";
    reverseString(str);
    printf("Reversed String: %s\n", str);  // Output: eM esreveR
    return 0;
}
```

6. `strncpy`: Copy first N characters

```c
#include <stdio.h>
#include <string.h>

int main() {
    char source[] = "Hello";
    char destination[10];
    strncpy(destination, source, 3);
    destination[3] = '\0';  // Ensure null termination
    printf("Copied String: %s\n", destination);  // Output: Hel
    return 0;
}
```

7. `strchr`: Find the first occurrence of a character

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

int main() {
    char str[] = "Locate the first 't'.";
    char *pos = strchr(str, 't');
    if (pos)
        printf("Found 't' at position: %ld\n", pos - str);
    else
        printf("Character not found\n");
    return 0;
}
```

8. `strstr`: Find a substring

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

int main() {
    char str[] = "Find the substring here.";
    char *substr = strstr(str, "substring");
    if (substr)
        printf("Substring found at position: %ld\n", substr - str);
    else
        printf("Substring not found\n");
    return 0;
}
```

---

7. Summary of String Functions

Function	Purpose

`strlen`	Calculate the length of a string
`strcpy`	Copy one string to another
`strncpy`	Copy the first N characters of a string
`strcat`	Concatenate two strings
`strcmp`	Compare two strings
`strchr`	Find the first occurrence of a character
`strstr`	Find a substring within a string
`strrev`	Reverse a string (if supported)

Program to input a string and print it:

```
#include <stdio.h>
int main() {
    char str[100];
    printf("Enter a string: ");
    scanf("%s", str);
    printf("You entered: %s\n", str);
    return 0;
}
```

Program to count the number of characters in a string (without `strlen`):

```
#include <stdio.h>
int main() {
    char str[100];
    int count = 0;
    printf("Enter a string: ");
    scanf("%s", str);
    while (str[count] != '\0') {
        count++;
    }
    printf("Length of the string: %d\n", count);
    return 0;
}
```

Program to reverse a string:

```
#include <stdio.h>
#include <string.h>
int main() {
    char str[100], temp;
    int i, j;
    printf("Enter a string: ");
    scanf("%s", str);
    int n = strlen(str);
    for (i = 0, j = n - 1; i < j; i++, j--) {
        temp = str[i];
        str[i] = str[j];
        str[j] = temp;
    }
    printf("Reversed string: %s\n", str);
    return 0;
}
```

Program to check if a string is a palindrome:

```
#include <string.h>
int main() {
    char str[100];
    int i, n, flag = 1;
    printf("Enter a string: ");
    scanf("%s", str);
    n = strlen(str);
    for (i = 0; i < n / 2; i++) {
        if (str[i] != str[n - i - 1]) {
            flag = 0;
            break;
        }
    }
    if (flag)
        printf("The string is a palindrome.\n");
    else
        printf("The string is not a palindrome.\n");
    return 0;
}
```

String Functions

Program to implement `strlen`:

```
#include <stdio.h>
int stringLength(char str[]) {
    int count = 0;
    while (str[count] != '\0') {
        count++;
    }
    return count;
}
int main() {
    char str[100];
    printf("Enter a string: ");
    scanf("%s", str);
    printf("Length of the string: %d\n", stringLength(str));
    return 0;
}
```

Program to concatenate two strings (without `strcat`):

```
#include <stdio.h>
void stringConcat(char str1[], char str2[]) {
    int i = 0, j = 0;
    while (str1[i] != '\0') {
        i++;
    }
    while (str2[j] != '\0') {
        str1[i] = str2[j];
        i++;
        j++;
    }
    str1[i] = '\0';
}
int main() {
    char str1[100], str2[50];
    printf("Enter first string: ");
    scanf("%s", str1);
    printf("Enter second string: ");
    scanf("%s", str2);
    stringConcat(str1, str2);
    printf("Concatenated string: %s\n", str1);
    return 0;
}
```

Program to compare two strings (without `strcmp`):

```c
#include <stdio.h>
int stringCompare(char str1[], char str2[]) {
    int i = 0;
    while (str1[i] != '\0' && str2[i] != '\0') {
        if (str1[i] != str2[i]) {
            return str1[i] - str2[i];
        }
        i++;
    }
    return str1[i] - str2[i];
}
int main() {
    char str1[100], str2[100];
    printf("Enter first string: ");
    scanf("%s", str1);
    printf("Enter second string: ");
    scanf("%s", str2);
    int result = stringCompare(str1, str2);
    if (result == 0)
        printf("Strings are equal.\n");
    else if (result < 0)
        printf("String 1 comes before String 2.\n");
    else
        printf("String 1 comes after String 2.\n");
    return 0;
}
```

Program to find the first occurrence of a character (using `strchr`):

```c
#include <stdio.h>
#include <string.h>
int main() {
    char str[100], ch;
    printf("Enter a string: ");
    scanf("%s", str);
    printf("Enter the character to search for: ");
    scanf(" %c", &ch);
    char *pos = strchr(str, ch);
    if (pos)
        printf("Character found at position: %ld\n", pos - str);
    else
        printf("Character not found.\n");
    return 0;
}
```

Program to find all occurrences of a substring (using `strstr`):

```c
#include <stdio.h>
#include <string.h>
int main() {
    char str[100], substr[50];
    printf("Enter the main string: ");
    scanf("%s", str);
    printf("Enter the substring to search: ");
    scanf("%s", substr);
    char *pos = strstr(str, substr);
    if (!pos) {
        printf("Substring not found.\n");
        return 0;
    }
    while (pos) {
        printf("Substring found at position: %ld\n", pos - str);
        pos = strstr(pos + 1, substr);
    }
    return 0;
}
```

Program to count vowels, consonants, digits, and spaces in a string:

```
#include <stdio.h>
int main() {
    char str[100];
    int vowels = 0, consonants = 0, digits = 0, spaces = 0;
    printf("Enter a string: ");
    scanf(" %[^\n]", str);
    for (int i = 0; str[i] != '\0'; i++) {
        if ((str[i] >= 'a' && str[i] <= 'z') || (str[i] >= 'A' &&
str[i] <= 'Z')) {
            if (str[i] == 'a' || str[i] == 'e' || str[i] == 'i' ||
str[i] == 'o' || str[i] == 'u' ||
                str[i] == 'A' || str[i] == 'E' || str[i] == 'I' ||
str[i] == 'O' || str[i] == 'U') {
                vowels++;
            } else {
                consonants++;
            }
        } else if (str[i] >= '0' && str[i] <= '9') {
            digits++;
        } else if (str[i] == ' ') {
            spaces++;
        }
    }
    printf("Vowels: %d, Consonants: %d, Digits: %d, Spaces: %d\n",
vowels, consonants, digits, spaces);
    return 0;
}
```

Practice programs (Home Work)

1. Write a program to find the frequency of each character in a given string.
2. Write a program to count the number of words in a string.
3. Write a program to remove all vowels from a string.
4. Write a program to check if two strings are anagrams of each other.

5. Write a program to find the second most frequent character in a string.
6. Write a program to replace all occurrences of a given character with another character in a string.
7. Write a program to find and replace a substring within a string.
8. Write a program to reverse each word in a string while keeping the word order intact.
9. Write a program to find the longest word in a string.
10. Write a program to convert all lowercase letters in a string to uppercase and vice versa.
11. Write a program to count the number of palindromic substrings in a string.
12. Write a program to sort the characters of a string in alphabetical order.
13. Write a program to check if a string contains only unique characters.
14. Write a program to implement your own version of `strtok` to tokenize a string based on a given delimiter.
15. Write a program to calculate the edit distance (Levenshtein distance) between two strings.
16. Write a program to dynamically allocate memory for a string and reverse it.
17. Write a program to merge two strings alternatively (e.g., merge "abc" and "123" to get "a1b2c3").
18. Write a program to remove duplicate characters from a string.
19. Write a program to remove all whitespace characters from a string.
20. Write a program to find the lexicographically smallest and largest substrings of a string.
21. Write a program to implement `strncmp` to compare the first `n` characters of two strings.
22. Write a program to implement `strncat` to concatenate the first `n` characters of one string to another.
23. Write a program to implement `memcpy` for copying a specified number of bytes between two memory locations.
24. Write a program to implement `strrchr` to find the last occurrence of a character in a string.
25. Write a program to implement `strpbrk` to find the first occurrence of any character from one string in another string.
26. Write a program to check if a string follows a given pattern (e.g., "aabb" matches pattern "xxyy").
27. Write a program to count the number of occurrences of a substring in a string.
28. Write a program to find all permutations of a given string.
29. Write a program to implement the KMP (Knuth-Morris-Pratt) string matching algorithm.
30. Write a program to implement the Rabin-Karp string matching algorithm.
31. Write a program to generate all possible substrings of a string.
32. Write a program to compress a string using Run Length Encoding (e.g., "aaabbc" becomes "a3b2c1").
33. Write a program to check if a string can be rearranged to form a palindrome.
34. Write a program to remove all characters from the first string that are present in the second string.
35. Write a program to count the number of times each word appears in a sentence.

36. Write a program to reverse the order of words in a sentence (e.g., "C is fun" becomes "fun is C").
37. Write a program to find the first non-repeating character in a string.
38. Write a program to calculate the longest prefix that is also a suffix in a string.
39. Write a program to check if a string can be segmented into a space-separated sequence of dictionary words.
40. Write a program to generate all possible valid permutations of parentheses for a given n.
41. Write a program to encode a string by shifting characters by a given value (Caesar Cipher).
42. Write a program to decode a Caesar Cipher string.
43. Write a program to validate if a string is a valid email address.
44. Write a program to parse a CSV file and extract specific data fields from it.
45. Write a program to determine if a string is a valid hexadecimal number.

Day_9 C-Programming

(RECAP OF PREVIOUS DAY)
Structure, Structure within structure, Array of structure, Union, Difference between Structure and Union, Dynamic Memory Allocation: malloc, calloc, realloc and free.

Structure

A Structure is a collection of elements, which are of different data types.
It is a user-defined data type that allows grouping variables of different data types under a single name.

Syntax:

```
struct StructureName {
   dataType member1;
   dataType member2;
   ...
};
```

Example:

```
#include <stdio.h>

struct Student {
    int id;
    char name[50];
    float marks;
};

int main() {
    struct Student s1 = {1, "Alice", 95.5};

    printf("ID: %d\n", s1.id);
    printf("Name: %s\n", s1.name);
    printf("Marks: %.2f\n", s1.marks);

    return 0;
}
```

Accessing Members:

- Use the dot operator (`.`) for direct access and use arrow operator(->) to access members through pointer
- Example: `s1.id or s1->id` **`(We will see this after malloc function today)`**

Differences Between Structures and Arrays:

Aspect	Structure	Array
Data Type	Can store multiple variables of **different data types**.	Stores multiple variables of the **same data type**.
Definition Syntax	Defined using the **struct** keyword.	Declared with the data type followed by the size. **int a[10]**
Memory Allocation	Memory is allocated for each member individually.	Memory is allocated as a contiguous block for all elements.
Accessing Elements	Accessed using the dot (.) operator and the member name.	Accessed using an index (e.g., array[index]).

Structure Within Structure

A structure can have another structure as a member.

Example:

```
#include <stdio.h>

struct Address {
    char city[50];
    int pin;
};

struct Employee {
    int id;
    char name[50];
    struct Address addr; // Nested structure
};

int main() {
    struct Employee e1 = {1, "John", {"New York", 12345}};

    printf("ID: %d\n", e1.id);
    printf("Name: %s\n", e1.name);
    printf("City: %s\n", e1.addr.city);
    printf("PIN: %d\n", e1.addr.pin);

    return 0;
}
```

Array of Structures

An array of structures allows storing multiple records of the same structure type.

Example:

```
#include <stdio.h>

struct Book {
    int id;
    char title[50];
    float price;
};

int main() {
    struct Book books[3] = {
        {1, "C Programming", 300.50},
        {2, "Data Structures", 450.75},
        {3, "Algorithms", 500.00}
    };

    for (int i = 0; i < 3; i++) {
        printf("Book ID: %d\n", books[i].id);
        printf("Title: %s\n", books[i].title);
        printf("Price: %.2f\n\n", books[i].price);
    }

    return 0;
}
```

Union

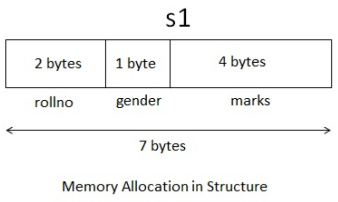
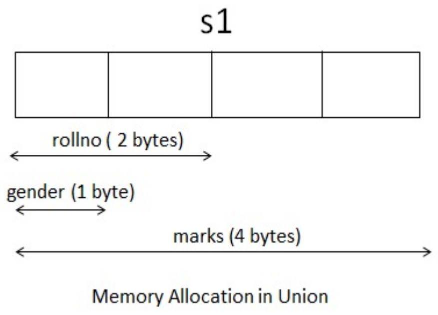
A union is similar to a structure, but it uses a single shared memory location for all its members. Only one member can store a value at a time.

Syntax:

```
union UnionName {
    dataType member1;
    dataType member2;
    ...
};
```

Difference between Structure and union

	Structure	**Union**
Definition	A structure is a user-defined data type that groups different data types into a single entity.	A union is a user-defined data type that allows storing different data types at the same memory location.

Keyword	The keyword struct is used to define a structure	The keyword union is used to define a union
Size	The size is the sum of the sizes of all members, with padding if necessary.	The size is equal to the size of the largest member, with possible padding.
	 s1 2 bytes 1 byte 4 bytes rollno gender marks 7 bytes Memory Allocation in Structure	 s1 rollno (2 bytes) gender (1 byte) marks (4 bytes) Memory Allocation in Union
Memory Allocation	Each member within a structure is allocated a unique storage area of location.	Memory allocated is shared by individual members of the union.
Data Overlap	No data overlap as members are independent.	Full data overlap as members share the same memory.
Accessing Members	Individual members can be accessed at a time.	Only one member can be accessed at a time.

Example:

```
#include <stdio.h>

union Data {
    int i;
    float f
    char str[20];
};

int main() {
    union Data data;

    data.i = 10;
    printf("Integer: %d\n", data.i);

    data.f = 220.5;
    printf("Float: %.2f\n", data.f);

    strcpy(data.str, "C Programming");
    printf("String: %s\n", data.str);

    return 0;
}
```

Dynamic Memory Allocation

Static memory allocation	Dynamic memory allocation
It is the process of allocating memory at compile time.	It is the process of allocating memory during execution of program.
Fixed number of bytes will be allocated.	Memory is allocated as and when it is needed.
The memory is allocated in memory stack.	The memory is allocated from free memory pool (heap).

Dynamic memory allocation allows memory to be allocated at runtime. The standard library provides functions like **`malloc`**, **`calloc`**, **`realloc`**, **and `free`** in `<stdlib.h>`.

1. malloc()

Allocates a single block of memory without initializing it.

Syntax: `void* malloc(size);`

Example:

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

int main() {
    int *ptr;
    ptr = (int *)malloc(5 * sizeof(int)); // Allocate memory for 5 integers, 5x2 =10 byte

    if (ptr == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }

    for (int i = 0; i < 5; i++) {
        ptr[i] = i + 1;
    }

    for (int i = 0; i < 5; i++) {
        printf("%d ", ptr[i]);
    }

    free(ptr); // Free the allocated memory
    return 0;
}
```

Example:
Structure Using arrow operator(->)

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

struct Student {
    int id;
    char name[50];
    float marks;
};

int main() {
    // Dynamically allocate memory for a structure
    struct Student *s1 = (struct Student *)malloc(sizeof(struct Student));

    // Assign values using the arrow operator
    s1->id = 1;
    snprintf(s1->name, sizeof(s1->name), "Alice"); // Use snprintf for strings
    s1->marks = 95.5;

    // Access and print values using the arrow operator
    printf("ID: %d\n", s1->id);
    printf("Name: %s\n", s1->name);
    printf("Marks: %.2f\n", s1->marks);

    // Free allocated memory
    free(s1);

    return 0;
}
```

2. calloc()

Allocates memory in multiple contiguous blocks(like in Array) and initializes all elements to zero.

0	0	0	0	0	0

Syntax: `void* calloc(n, size);`

`Difference between malloc() and calloc()`

Feature	malloc()	calloc()
Prototype	**void *malloc(size);**	**void *calloc(n, size);**

Number of Arguments	**Accepts 1 argument: the total memory size in bytes.**	**Accepts 2 arguments: the number of blocks (n) and size of each block.**
Memory Allocation	**Allocates one block of memory.**	**Allocates multiple blocks each of specified size.**
Memory Initialization	**Does not initialize memory (contains garbage values).**	**Initializes memory to 0 (all allocated bytes are set to zero).**
Usage	**Commonly used for a single block of memory.**	**Commonly used when allocating memory for arrays or multiple items.**

Example:

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

int main() {
    int *ptr;
    ptr = (int *)calloc(5, sizeof(int)); // Allocate memory for 5 integers, 5 blocks each of 2 Byte

    if (ptr == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }

    for (int i = 0; i < 5; i++) {
        printf("%d ", ptr[i]); // All elements initialized to 0
    }

    free(ptr); // Free the allocated memory
    return 0;
}
```

3. realloc() – (to Reallocate Memory)

Changes the size of previously allocated memory.

Syntax: `void* realloc(void* ptr, size);`

Example:

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

int main() {
    int *ptr = (int *)malloc(3 * sizeof(int));

    if (ptr == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }

    for (int i = 0; i < 3; i++) {
        ptr[i] = i + 1;
    }

    // Reallocate memory for 5 integers
    ptr = (int *)realloc(ptr, 5 * sizeof(int));

    for (int i = 3; i < 5; i++) {
        ptr[i] = i + 1;
    }

    for (int i = 0; i < 5; i++) {
        printf("%d ", ptr[i]);
    }

    free(ptr); // Free the allocated memory
    return 0;
}
```

4. free() (to free previously allocated memory)

Releases dynamically allocated memory back to the system.

Syntax: `void free(void* ptr);`

Important Notes:

1. Always use `free` to release memory allocated with `malloc`, `calloc`, or `realloc`.
2. Accessing freed memory leads to undefined behavior.

Programs to Practice (HW):

1. Define a structure to store employee details and display them.
2. Write a program to calculate the total and average marks of a student using a structure.
3. Create a program to store and display details of 5 books using an array of structures.
4. Implement a structure to store date (day, month, year) and write a program to calculate the difference between two dates.
5. Create a program to store details of a car (model, price, color) and display the most expensive car.
6. Define a structure for a `Student` with a nested structure for `Address` (street, city, pin).
7. Create a program to store and display details of players, including their personal details (nested structure).
8. Implement a structure for a company with nested structures for employee details and their department details.
9. Write a program to calculate the total salary of employees, where salary details are part of a nested structure.
10. Create a structure for a `Library` with a nested structure for `Book` and display all the books by a specific author.
11. Store and display details of 10 students using an array of structures.
12. Create a program to store and display details of products in a shop using an array of structures.
13. Write a program to calculate the average marks of students stored in an array of structures.
14. Implement a program to display the details of employees earning above a specific salary using an array of structures.
15. Store and sort details of movies (title, year, rating) using an array of structures.
16. Create a union to store different types of data (int, float, char) and demonstrate memory sharing.
17. Write a program to store and display data of a variable length (e.g., an integer, a float, or a string) using a union.
18. Compare and demonstrate memory usage in structures and unions.

19. Implement a union to represent a record that can store either a student’s marks or attendance.
20. Write a program to use a union to store and display a value as both integer and floating-point number.
21. Allocate memory for an integer array using `malloc` and input values dynamically.
22. Write a program to create a dynamically allocated 2D array using `malloc`.
23. Use `calloc` to allocate memory for an array of integers and initialize all elements to 0.
24. Demonstrate the use of `realloc` to resize a dynamically allocated array.
25. Implement a program to create a linked list using `malloc` for dynamic memory allocation.
26. Create a program to dynamically allocate memory for a string and copy a user-provided string into it.
27. Write a program to allocate memory for `n` students using `calloc` and input their details.
28. Demonstrate freeing allocated memory using `free` after a program finishes execution.
29. Implement a program to find the sum of an array of numbers entered by the user, using dynamic memory allocation.
30. Dynamically allocate memory for a structure (e.g., Student) and input details.
31. Create a structure to store employee details and dynamically allocate memory for an array of employees.
32. Write a program to store book details using an array of structures, allocating memory dynamically.
33. Implement a nested structure for storing details of students and dynamically allocate memory for `n` students.
34. Use a union and dynamically allocate memory to store either an integer or a float, based on user input.
35. Write a program to store employee details and calculate the average salary, using both structures and `malloc`.
36. Demonstrate the use of `calloc` to initialize an array of structures with default values.
37. Create a program to input and display an array of employee details, resizing it using `realloc`.
38. Write a program to store and manipulate date details using dynamic memory allocation and structures.
39. Use a structure with dynamically allocated memory to store strings of variable lengths.
40. Create a dynamic array of unions to store mixed data types and display their contents.

Day_10 C-Programming

(RECAP OF PREVIOUS DAY)
File Types, File operations, File pointer, File opening modes, File handling functions, Command Line Arguments, File handling through command line argument, Record I/O in files, Preprocessor directives: Macros and File inclusion

File Types in C

Files in C are used for permanent storage of data. C supports two types of files:

1. **Text Files**:

    - Contain data in human-readable format.
    - Example: `.txt`, `.csv`
2. **Binary Files**:

    - Contain data in machine-readable format (0s and 1s).
    - Example: `.bin`, `.dat`

File Operations in C

C provides a set of operations to handle files:

1. **Create a file**: Create a new file for storing data.
2. **Open a file**: Open an existing file for reading or writing.
3. **Read from a file**: Retrieve data from the file.
4. **Write to a file**: Write data to the file.
5. **Close a file**: Close the file to free resources.

File Pointer

- A file pointer is a pointer to a **`FILE`** structure, used to control file operations.

Syntax:

```
FILE *fp;
fp = fopen("example.txt", "r");
if (fp == NULL) {
    printf("File could not be opened\n");
}
fclose(fp);
```

File Opening Modes

C provides multiple modes for opening files:

Mode	Description
`"r"`	Open for reading. File must exist.
`"w"`	Open for writing. Creates/truncates.
`"a"`	Open for appending. Creates if absent.
`"r+"`	Open for reading and writing.
`"w+"`	Open for reading and writing.
`"a+"`	Open for reading and appending.
`"rb"`	Open for reading in binary mode.
`"wb"`	Open for writing in binary mode.

File Handling Functions

1. **Opening a file**:
    - `FILE *fopen(const char *filename, const char *mode);`

Example:

```
FILE *fp = fopen("data.txt", "r");
```

2. **Closing a file**:
    - `int fclose(FILE *fp);`

Example:

```
fclose(fp);
```

3. **Reading from a file**:
    - `fgetc()`: Read a single character.
    - `fgets()`: Read a string.
    - `fread()`: Read binary data.

Example:

```
char ch = fgetc(fp);
```

4. **Writing to a file**:
    - `fputc()`: Write a single character.
    - `fputs()`: Write a string.
    - `fwrite()`: Write binary data.

Example:

```
fputc('A', fp);
```

5. **Positioning Functions**:
    - `fseek()`: Move the file pointer to the desired location.
    - `ftell()`: tells the current position of the file pointer.
    - `rewind()`: Set the file pointer to the start.
      We will see examples later in this session

Some File Programs

Example:
Program that writes content into a file and then reads it back to display on the screen

```
#include <stdio.h>

int main() {
    FILE *file;
    char content[] = "This is a sample text written into the file.\nThis is the second line.\n";

    // Open the file for writing
    file = fopen("sample.txt", "w");
    if (file == NULL) {
        printf("Error opening the file for writing.\n");
        return 1;
    }

    // Write content into the file
    fprintf(file, "%s", content);

    // Close the file after writing
    fclose(file);
    printf("Content has been written to the file.\n");

    // Open the file for reading
    file = fopen("sample.txt", "r");
    if (file == NULL) {
        printf("Error opening the file for reading.\n");
        return 1;
    }

    // Display the content of the file
    printf("\nReading the content from the file:\n");
    char ch;
    while ((ch = fgetc(file)) != EOF) {
        putchar(ch);  // Print each character to the screen
    }

    // Close the file after reading
    fclose(file);

    return 0;
}
```

Example:

Write a program that takes 30 numbers as input, writes them into a file, then separates even and odd numbers into two different files named **`EVEN.txt`** and **`ODD.txt`**

```
#include <stdio.h>

int main() {
    FILE *inputFile, *evenFile, *oddFile;
    int numbers[30];
    int i;

    // Open the input file for writing
    inputFile = fopen("numbers.txt", "w");
    if (inputFile == NULL) {
        printf("Error opening numbers.txt file.\n");
        return 1;
    }

    // Taking 30 numbers as input from the user
    printf("Enter 30 numbers:\n");
    for (i = 0; i < 30; i++) {
        scanf("%d", &numbers[i]);
        fprintf(inputFile, "%d\n", numbers[i]);  // Write each number to the file
    }

    // Close the input file after writing
    fclose(inputFile);

    // Open the EVEN and ODD files for writing
    evenFile = fopen("EVEN.txt", "w");
    oddFile = fopen("ODD.txt", "w");
    if (evenFile == NULL || oddFile == NULL) {
        printf("Error opening EVEN.txt or ODD.txt file.\n");
        return 1;
    }

    // Separate the even and odd numbers and write them to respective files
    for (i = 0; i < 30; i++) {
        if (numbers[i] % 2 == 0) {
            fprintf(evenFile, "%d\n", numbers[i]);
        } else {
            fprintf(oddFile, "%d\n", numbers[i]);
        }
    }

    // Close the EVEN and ODD files after writing
    fclose(evenFile);
    fclose(oddFile);
    printf("The numbers have been separated into EVEN.txt and ODD.txt files.\n");

    return 0;
}
```

Example:
example of fseek, ftell, and rewind

```
#include <stdio.h>

int main() {
    // Open a file for writing
    FILE *file = fopen("example.txt", "w");

    if (file == NULL) {
        printf("Error opening file\n");
        return 1;
    }

    // Write some data to the file
    fprintf(file, "Hello, World!\nThis is an example of fseek, ftell, and rewind.");

    // Move the file pointer 6 bytes from the beginning
    fseek(file, 6, SEEK_SET);
    printf("File pointer moved to position: %ld\n", ftell(file)); // Prints 6

    // Move the file pointer 5 bytes ahead from the current position
    fseek(file, 5, SEEK_CUR);
    printf("File pointer moved to position: %ld\n", ftell(file)); // Prints 11

    // Move the file pointer 5 bytes from the end
    fseek(file, -5, SEEK_END);
    printf("File pointer moved to position: %ld\n", ftell(file)); // Prints position 47 (based on file
size)

    // Rewind the file pointer to the start
    rewind(file);
    printf("File pointer after rewind: %ld\n", ftell(file)); // Prints 0

    // Close the file
    fclose(file);

    return 0;
}
```

Command Line Arguments

- Command-line arguments allow passing inputs to the program at runtime.

Syntax:

```
int main(int argc, char *argv[])
```

  - **`argc`: Number of arguments.**
  - **`argv`: Array of arguments.**

Example:

```
 #include <stdio.h>

int main(int argc, char *argv[]) {
    printf("Program name: %s\n", argv[0]);
    if (argc > 1) {
        printf("Argument: %s\n", argv[1]);
    }
    return 0;
}
```

To Run it from command line

- Lets name it as MyProg.c
- Compile it from command line
- Run it without argument” MyProg.c
  - Output: Program name:MyProg.c
- Run it with argument: MyProg.c Hello
  - Output: Program name:MyProg.c
    Argument: Hello

File Handling Through Command Line Arguments

Example program to copy the contents of one file to another:

```
 #include <stdio.h>

int main(int argc, char *argv[]) {
    if (argc != 3) {
        printf("Usage: %s source_file target_file\n", argv[0]);
        return 1;
    }

    FILE *src = fopen(argv[1], "r");
    FILE *dest = fopen(argv[2], "w");

    if (src == NULL || dest == NULL) {
        printf("Error opening file.\n");
        return 1;
    }

    char ch;
    while ((ch = fgetc(src)) != EOF) {
        fputc(ch, dest);
    }

    fclose(src);
    fclose(dest);
    printf("File copied successfully.\n");
    return 0;
}
```

Record I/O in Files

- Used to store and retrieve structured data (e.g., records of employees).

Example:

```
 #include <stdio.h>

struct Employee {
    int id;
    char name[50];
    float salary;
};

int main() {
    struct Employee e = {1, "Alice", 50000.0};

    FILE *fp = fopen("employee.dat", "wb");
    fwrite(&e, sizeof(struct Employee), 1, fp);
    fclose(fp);
    struct Employee e_read;
    fp = fopen("employee.dat", "rb");
    fread(&e_read, sizeof(struct Employee), 1, fp);
    fclose(fp);

    printf("ID: %d, Name: %s, Salary: %.2f\n", e_read.id, e_read.name, e_read.salary);
    return 0;
}
```

Preprocessor Directives

a. Macros:

- Macro is a piece of code in a program that is replaced by the value of the macro.
- Macros are constants or functions defined using the `#define` directive.

Example:

```
#define PI 3.14159
#define SQUARE(x) ((x) * (x))

int main() {
    printf("PI: %.2f\n", PI);
    printf("Square of 5: %d\n", SQUARE(5));
    return 0;
}
```

Program: Calculate Area of a Circle Using a Macro

```
#include <stdio.h>

#define PI 3.14159
#define AREA_OF_CIRCLE(radius) (PI * (radius) * (radius))

int main() {
    float radius;

    printf("Enter the radius of the circle: ");
    scanf("%f", &radius);

    float area = AREA_OF_CIRCLE(radius);
    printf("The area of the circle with radius %.2f is: %.2f\n", radius, area);

    return 0;
}
```

b. File Inclusion:

- Used to include header files or other files into the program.

Two main types of include file directives in C:

1. #include <filename.h>:
The use of this syntax is to include system header file in C program. It tells the compiler to look for or find the file in the standard system directory.

2. #include "filename":
This syntax is used to add a header file. It tells the compiler to first look for the file in the current directory and then look in the system directory.

```
#include <stdio.h>
#include "myheader.h" // User-defined header file
```

Programs to practice(HW)

1. File Types and File Operations

- **Program 1**: Create a text file and write data into it using `fwrite()`.
- **Program 2**: Read data from a file using `fread()` and display it on the screen.
- **Program 3**: Append data to an existing file without overwriting using `fopen()` in append mode.

2. File Pointer and File Operations

- **Program 4**: Move the file pointer using `fseek()`, `ftell()`, and `rewind()`, and display the current file pointer position at various stages.
- **Program 5**: Implement a program that uses `fseek()` to search for a specific word or pattern in a file.
- **Program 6**: Read and display file content in reverse order using `fseek()`.

3. File Opening Modes

- **Program 7**: Open a file in different modes (`"r"`, `"w"`, `"a"`, `"r+"`, `"w+"`, `"a+"`) and handle errors for each mode.
- **Program 8**: Create a program that handles errors gracefully when trying to open a file that doesn't exist in `"r"` mode, or when trying to write in `"r"` mode.

4. File Handling Functions

- **Program 9**: Use `fopen()`, `fclose()`, `fgetc()`, and `fputc()` to read and write one character at a time in a file.
- **Program 10**: Implement a program that reads from a file line-by-line using `fgets()` and writes the content to another file.
- **Program 11**: Write a program that reads and writes a structured record (e.g., `struct` type) to/from a binary file using `fwrite()` and `fread()`.

5. Command Line Arguments

- **Program 12**: Write a program that takes two numbers from the command line, adds them, and displays the sum.
- **Program 13**: Create a program that takes a list of strings as command-line arguments and prints each string.
- **Program 14**: Implement a program that takes a file name as a command-line argument, reads the content of that file, and prints it.

6. File Handling through Command Line Arguments

- **Program 15**: Write a program that accepts a file name as a command-line argument, opens the file, reads the content, and displays it.
- **Program 16**: Create a program that accepts file names from the command line, and copies content from the source file to the destination file.
- **Program 17**: Implement a program that accepts a file name and a word from the command line and counts how many times the word appears in the file.

7. Record I/O in Files

- **Program 18**: Implement a program to write student records (name, roll number, marks) into a file using `struct` and binary file operations (`fwrite()`, `fread()`).
- **Program 19**: Write a program to read and display student records from a file, and update a student record (e.g., modify marks) in the file.
- **Program 20**: Create a program that writes multiple records (students) into a file, then reads the file and displays the records in tabular format.

8. Preprocessor Directives: Macros and File Inclusion

- **Program 21**: Implement a program that uses macros to find the maximum of two numbers.
- **Program 22**: Create a program with a header file (`.h`) that contains function prototypes for file operations, and implement the functions in a separate `.c` file.
- **Program 23**: Write a program that includes an external header file with preprocessor directives to define constants and use them in the program.
- **Program 24**: Implement a program that defines a macro for a factorial and uses it to compute the factorial of a number.

9. (Mixed Topics)

- **Program 25**: Write a program that takes a list of students' names and grades from the user, writes them to a file, and later reads and sorts the data in ascending order based on grades.
- **Program 26**: Create a program to count the number of lines, words, and characters in a file using `fgetc()` or `fgets()`.
- **Program 27**: Write a program that reads integers from a file, computes their average, and writes the result to a new file.