C++

Ch-1 Introduction

About C++ programming :-

→ C++ → 1979 → Bjarne Stroustrup → Extension of C.

- **Multi-paradigm Language** :- C++ Supports at least Seven different styles of programming. Developers Can Choose any of the styles.
- **General purpose Language** :- You Can Use C++ to develop games, desktop apps, operating system, and so on.
- **Speed** - Like C programming, the performance of optimized C++ Code is Exceptional.
- **Object oriented** - C++ allows you to divide Complex problems into Smaller sets by using objects.

Why learn C++ ?

- C++ is Used to develop games, desktop apps, operating systems, browsers, and so on because of its performance.
- After learning C++, it will be much Easier to learn other programming language like Java, python, etc.
- C++ helps you to Understand the internal architecture of a Computer, how Computer Stores and retrives information.

C++ Variables, literals and Constants

C++ Variables

In programming, a variable is a container (storage area) to hold data.

To indicate the storage area, each variable should be given a unique name (identifier). For Exr

```
int age = 14;
```

Here, age is a variable of the int data type, and we have assigned an integer value 14 to it.

C++ literals

Literals are data used for representing fixed values. They can be used directly in the code. For example: 1, 2.5, 'C' etc.

C++ Constants

In C++, we can create variables whose value cannot be changed. For that, we use (const) keyword. Here's an example:-

```
const int light_speed = 299792458;
light_speed = 2500 // Error! light_speed is a constant.
```

C++ Data types

In C++, data types are declarations for variables. This determines the type and size of data

associated with Variables. Ex:-

int age = 13;

Here, age is a variable of type int. Meaning, the variable can only store integers of either 2 or 4 bytes.

C++ Fundamental Data types.

Data type	Meaning	Size (in bytes)
int	Integer	2 or 4
float	Floating-point	4
double	Double floating-point	4
char	Character	1
wchar_t	Wide character	2
bool	boolean	1
void	Empty	0

Syntax for declaring variables in C++.

- Data-type variable-name = value;
- Ex. int a = 4, b = 6;
- Based on scope, variable can be classified into two types; (Scope = Kha par uplabdh hai variable)
  1. Local variables
  2. Global variables.

Variable Scope

- Scope of a variable is the region in code where the existence of variable is valid.

- Based on Scope we have local and global variables in C++.
- Local Variables: Local variables are declared inside the braces of any function and can be accessed only from there.
- Global Variables:- Global variables are declared outside any function and can be accessed from anywhere.
- Can global and local variables have same name in C++?: Yes.
- Data types define the type of data a variable can hold, for ex:- an integer variable can hold integer data, a character type variable can hold character data etc.
- Data types in C++ are categorized in three groups:-
  - Built in (int, float, Char, Double, bool etc.)
  - User defined (Struct, Union, Enum) etc.
  - Derived (Array, function, pointer) etc.

Derived data types:- Data types that are derived from fundamental data types are derived data type. Ex:- arrays, pointers, functions types, structure, etc.

C++ Basic Input/output

C++ Output

In C++, cout sends formatted output devices, such as the screen. we use the cout object along with the << operator for displaying output.

Ex:- String Output.

```
#include <iostream>
using namespace std;

int main() {
    // print the string enclosed in double quotes
    cout << "This is C++ programming";
    return 0;
}
```

Output

This is C++ programming.

How does this program work?

- We first include the iostream header file that allows us to display output.
- The cout object is defined inside the std namespace. To use the std namespace, we used the using namespace std; statement

- Every C++ program starts with the main() function. The Code Execution begins from the start of the main() function.
- Cout is a object that prints the string inside quotation marks " ". It is followed by the insertion operator <<.
- return 0; is the "exit status" of the main() function. The program ends with his statement, however, this statement is not mandatory.

Note. If we don't include the using namespace std; statement we need to use std::cout instead of cout.
- The endl manipulator is used to insert a newline. that's why each output is displayed in a new line.

C++ Input

In C++, Cin takes formatted input from standard input devices such as the keyboard. We use the Cin object along with the >> extraction operator for taking input.

C++ type Conversion

C++ Allows us to Convert data of one type to that of another. this is known as type Conversion.

There are two types of type Conversion in C++

1. Implicit Conversion
2. Explicit Conversion (also known as type Casting)

Implicit type Conversion

The type Conversion that is done Automatically done by the Compiler is known as implicit type Conversion. This type Conversion is also known as automatic Conversion.

Ex:- Conversion from int to double.

```
// Working of implicit type-Conversion.
#include <iostream>
Using namespace std;

int main() {
    // assing an int value to num_int.
    int num_int = 9;

    // declaring a double type variable.
    double num_double;
```

```
// implicit Conversion.
// assigning int value to a double variable.
num_double = num_int;

Cout << "num_int = " << num_int << endl;
Cout << "num_double = " << num_double << endl;

return 0;
}
```

Output

num_int = 9
num_double = 9.

C++ Explicit Conversion

When the User manually changes data from one type to another, this is known as Explicit Conversion. This type of Conversion is also known as type Casting.

Ex:- Type Casting

```
int a = 45;
float b = 45.46;

Cout << "The Value of a is " << (float) a << endl;
Cout << "The Value of a is " << float(a) << endl;

Cout << "The Value of b is " << (int) b << endl;
Cout << "The Value of b is " << int (b) << endl;
```

```
int c = int (b);

Cout << " The expression is " << a+b <<endl;
Cout << " The expression is " <<a+int(b) <<endl;
Cout << " The expression is " <<a+(int)b <<endl;
```

There are three major ways in which we can use Explicit Conversion in C++, They are:

1) C-Style type Casting (also known as Cast notation)
   Syntax :- (data-type) expression;

2) Function notation (also known as old C++ Style type Cast)
   Syntax :- data-type (expression);

3) Type Conversion operators:-
   - Static_Cast
   - dynamic_Cast
   - Const_Cast
   - reinterpret_Cast

Operators in C++

Operators are symbol that perform operations on variable and values. for example, (+) is an operator used for addition, while (−) is an operator used for subtraction.

Operators in C++ can be classified into 6 types:

1. Arithmetic operators
2. Assignment operators
3. Relational operators
4. Logical operators
5. Bitwise operators
6. Other operators. or Misc operators.

1:- C++ Arithmetic Operators

Arithmetic Operators are used to perform arithmetic operations on variables and data. for example `a+b;`
Here, the (+) operator is used to add two variables a and b. Similarly there are various other Arithmetic operators in C++.

Operator	Operations.
+	Addition
−	Subtraction
*	Multiplication
/	Division
%	Modulo operation (Remainder after division)

2:) Assignment Operators:

In C++, assignment operators are used to assign value to variables. for Example,

```
// assign 5 to a
a = 5;
```

Here we assign a value of 5 to the variable a.

Operators	Example	Equivalent to
=	a=b;	a=b;
+=	a+=b;	a=a+b;
-=	a-=b;	a=a-b;
*=	a*=b;	a=a*b;
/=	a/=b;	a=a/b;
%=	a%=b;	a=a%b;

3) Relational operators;
A Relational operator is used to check the relationship between two operands. for eg:- a>b;
Here, > is a relational operator. It checks if a is greater than b or not.

If the relation is true, it returns 1 whereas if the relation is false, it returns 0.

Operators	Meaning	Example
==	Is Equal to	3==5 gives us false
!=	Not Equal to	3!=5 gives us true
>	Greater than	3>5 gives us false
<	Less than	3<5 " " true
>=	Greater than or Equal to	3>=5 " " false
<=	Less than or Equal to	3<=5 " " true

4) C++ logical Operators.

Logical operators are used to check whether an expression is true or false. If the expression is true it returns 1 whereas if the expression is false, it returns 0.

Operators	Example	Meaning
&&	Expression && Expression2	logical AND. true only if all the operands are true.
||	Expression1 || Expression2	Logical OR. True if at least one operands is true.
!	! Expression	logical Not. True only if the operand is false.

In C++, logical operators are commonly used in decision making.

5) Bitwise Operators.

In C++, bitwise operators are used to perform operations on individual bits. They can only be used alongside char and int data types.

Operator	Description
&	Binary AND
|	Binary OR
^	XOR
~	one Complement
<<	Shift left
>>	Shift Right

6) Other C++ Operators:

Operators	Description.	Example.
sizeof	returns the size of datatype	sizeof(int);
?:	returns the value based on Condition.	String result = (5>0)? "even": "odd": //even
&	represent memory address of the operand	# //address of num
.	access member of struct variables or class object	S1.marks=92;
->	Used with pointers to access the class or struct variables	ptr->marks=92;
<<	prints the output value	Cout<<5;
>>	gets the input value	Cin>>num;

Precedence of Operators.

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

Decision Making

if/else:- The if block is used to specify the code to be executed if the condition specified in if is true, the else block is executed otherwise

Ex:

```
int age;
Cin >> age

if (age >= 18) {
    Cout << "You Can Vote";
}
else {
    Cout << "Not Eligible for Voting.";
}
return 0;
}
```

else if: To specify multiple if conditions, we first use if and then the consecutive statements use else if.

Ex:

```
int x, y;
Cin >> x >> y;

if (x == y) {
    Cout << "Both the numbers are Equal";
}
else if (x > y) {
    Cout << "X is greater than Y";
}
else {
    Cout << "Y is greater than X";
return 0; }
```

nested if:- To Specify Conditions within Conditions we make the use of nested ifs.

Ex:-
```
int x, y;
Cin >> x >> y;

if (x == y) {
    Cout << "Both the numbers are Equal";
}
else {
    if (x > y) {
        Cout << "X is greater than y";
    }
    else {
        Cout << "y is greater than x";
    }
}

return 0;
}
```

Loops in C++

A loop is used for Executing a block of Statements until a particular Condition is Satisfied. A loop Consists of an initialization Statement, a test Condition and an increament Statement.

There are 3 types of loops in C++.

- for loop
- while loop
- do...while

for loop :- The Syntax of for-loop.

```
for (initialization; Conditions; Update) {
    // body of loop.
}
```

- Here, initialization - initializes Variables and is executed only once.
- Condition - if true, the body of for loop is executed if false, the for loop is terminated.
- Update - Update the value of initialized Variables and again checks the Conditions.

Do while loop: The body of do while loop is Executed once before the Condition is checked.

Its Syntax is:

```
do {
    // body of loop;
}
while (Condition);
```

- The body of loop is Executed first. Then the Condition is Evaluated.
- If the Condition Evaluates to true, the body of the loop inside the do Statement is Executed again.
- The Condition is Evaluated Once again.

- If the Condition Evaluates to true, the body of the loop inside the do statement is Executed again.
- This process Continues Until the Condition Evaluates false. Then the loop Stops.

Ex:

```
{ int n;
  cin >> n;
  int sum=0;
  for(int
```

```
{ int i = 1;
  do
  {
    cout << i << " ";
    i++;
  } while(i<=5);
  return 0;
}
```

While Loop:- The Syntax of the while loop is

```
while (condition) {
    // body of the loop.
}
```

Here;

- A while loop Evaluates the Condition
- If the Condition Evaluates to true, the Code inside the while loop is Executed.
- The Condition is Evaluated again.
- This process Continues Until the Condition is false.

- When the Condition Evaluates to false, the loop terminates.

Ex:

```
{ int i=1;
// While loop from 1 to 5
While (i<=5) {
    Cout << i << " ";
    ++i;
}
return 0;
}
```

Output:-

1, 2, 3 4 5.

Jumps in loops

Jumps in loops are Used to Control the flow of loops. There are two statements Used to implement jumps in loops- Continue and Break. These statements are Used when we need to change the flow of the loop when some specified condition is met.

Continue:- In Computer programming the Continue statement is Used to skip the current iteration of the loop and the Control of the program goes to the next iteration.

The syntax of the Continue statement is:-

```
Continue;
```

Working of C++ Continue Statement.

```
for (init; Condition; Update) {
    // Code
    if (Condition to break) {
        Continue;
    }
    // Code
}
```

```
while (Condition) {
    // Code
    if (Condition to break) {
        Continue;
    }
    // Code
}
```

Ex:- Continue with for loop.

In a for loop, Continue skips the current iteration and the control flow jumps to the Update Expression.

```
// Program to print the value of i.
{
for (int i=1; i<=5; i++) {
    // condition to Continue
    if (i==3) {
        Continue;
    }
    cout<<i<<endl;
}
```

Output

1
2
4
5

Note:- The Continue Statement is almost always Used with decision-making Statement.

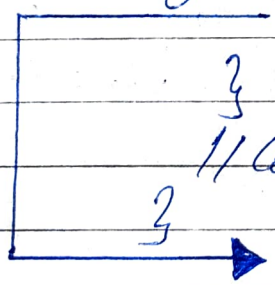
Break: In C++, The break Statement terminates the loop when it is encountered.

The Syntax of the break statement is:

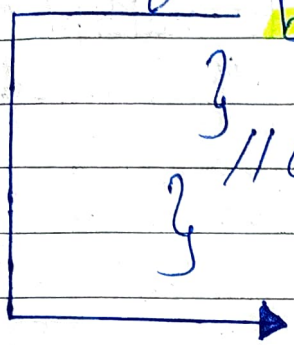
```
break;
```

Working of C++ break Statement

```
for (init; Condition; Update) {
    //Code
    if (Condition to break) {
        break;
    }
    //Code
}
```



```
while (Condition) {
    // Code
    if (Condition to break) {
        break;
    }
    // Code
}
```



Ex:-1 break with for loop

```
#include<iostream>
using namespace std;
int main() {
    for(int i=1; i<=5; i++) {
        // break condition
        if (i==3) {
            break;
        }
        cout<<i<<endl;
    }
    return 0;
}
```

Output

```
1
2
```

Note:- The break statement is usually used with decision making statements.

Switch Statement

Switch Case Statements are a substitute for long if statements that compare a variable to multiple values. After a match is found, it executes the corresponding code of that value case.

Syntax:

```
Switch(n)
{
  Case 1: // Code to be Executed if n==1;
    break;

  Case 2: // Code to be Executed if n==2;
    break;

  default: // Code to be Executed if n doesn't match
           any of the above Cases.
}
```

- The Variable in Switch should have a Constant Value.
- The break statement is optional. It terminates the switch statement and moves control to the next line after switch.
- If break statement is not added, switch will not get terminated and it will continue onto the next line after switch.
- Every Case value should be Unique.

- Default Case is optional. But it is important as it is executed when no Case value could be matched.

Note:- We can do the same thing with the if...else...if ladder. However, the syntax of the switch statement is cleaner and much easier to read and write.

Ex:- Write a program to write a simple Calculator.

```
{ Int n1, n2;
  Char op;

  Cout << "Enter 2 numbers";
  Cin >> n1 >> n2;

  Cout << "Enter operator:";
  Cin << op;

  Switch (op)
  { Case '+':
      Cout << n1+n2 << endl.
      break;
    Case '-':
      Cout << n1-n2 << endl.
      break;
    Case '*':
      Cout << n1 * n2 << endl,
      break;
    Case '%':
      Cout << n1 % n2 << endl;
      break;
    default:
      Cout << "Operator not found!" << endl;
    } break;
```

Output

```
Enter 2 numbers:
3 4
Enter Operator: *
12
```

FUNCTIONS

A function is a block of Code that performs a specific task.

Suppose we need to Create a program ~~that performs~~ to Create a Circle and Color it. we Can Create two functions to Solve this problem.
- a function to draw the Circle
- a function to Color the Circle

Divide a Complex problem into Smaller chunks makes our program Easy to Understand and reusable.

There are two types of function:

1. Standard library function: Predefined in C++.
2. User defined function: Created by User.

C++ User-defined function

C++ allows the programmer to define their own function.

C++ function Declaration

The Syntax to declare a function is:

```
returntype functionname (parameter1, parameter2, ...){
    // function body
}
```

for Ex:-

```
// function declaration
void greet() {
    cout << "Hellow world";
}
```

Calling function

In the above program, we have declared a function named greet(). To use the greet() function, we need to call it.

Here's how we can call the above greet() funtion.

Ex:-

```
int main() {
    // Calling a function
    greet();
}
```

Ex:-

```
#include <iostream>
void greet() {
    //Code
}
int main() {
    .....
    greet();
    .....
}
```

Function Call

How function works in C++.

Function Parameters

As mentioned above, a function can be declared with parameters (arguments). A parameters is value that is passed when declaring a fuction.

Ex:-

```
Void printnum (int num) {
    Cout << num;
}
```

Here, int variable num is a function parameter.

```
#include <iostream>

Void displayNum (int n1, double n2) {
    //code
}

int main() {
    .....
    displayNum(num1, num2);
    .....
}
```

function call

C++ function with parameters

Return Statement

In the above program, we have used void in the function declaration.

This means the function is not returning any value.

It's also possible to return a value from a function. For this, we need to specify the return type of the function during function declaration.

Then, the return statement can be used to return a value from a function.

Example:-

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

The return statement denotes that the function has ended. Any code after return inside the function is not executed.

```
#include <iostream>
int add (int a, int b){
    return (a+b);
}
int main() {
    int sum;
    sum = add(100, 78);
    .....
```

Function Call.

Notice that Sum is a variable of int type. This is because the return value of add() is of int type.

Function Prototype

In C++, the code of function declaration should be before the function call. However, if we want to define a function after the function call, we need to use the function prototype.

Example:-

```
// Function prototype
Void add(int, int);

int main() {
    // Calling the function before declaring it
    add(5, 3);
    return 0;
}

// Function definition
Void add(int a, int b) {
    Cout << (a+b);
}
```

Benefits of Using User-defind Functions.

- Functions make the code reusable. We can declare them once and use them multiple times.
- Functions make the program easier as each small task is divided into a function.
- Function increase readability.

C++ library functions

Library functions are the builtin function in C++ programming.

Programmers can use library function by invoking the function directly; they don't need to write the functions themselves.

Some common library functions in C++ are sqrt(), abs(), isdigit(), etc.

In order to use library function, we usually need to include the header file in which the library functions are defined.

In order to use mathemetical functions such as sqrt() and abs(), we need to include header file cmath.

FUNCTION OVERLOADING

In C++ two functions can have the same name if the number and/or type of arguments passed is different.

These functions having the same name but different arguments are known as overloaded functions.

Ex:-
```
// Same name different arguments
int test() {}
int test(int a) {}
int test(double a) {}
int test(int a, double b) {}
```

Here all 4 functions are overloaded functions.

Notice that the return type of all these 4 functions are not the same. Overload functions may or may not have different return types but they must have different arguments.

Ex:-
```
// Error Code
int test(int a) {}
double test(int b) {}
```

Here both functions have the same name, the same type, and the same number of arguments. Hence, the compiler will through error.

```
void display(int Var1, double Var2) {
        //Code
}

void display(double Var) {
        //Code
}

void display(int Var) {
        //Code
}

int main() {
    int a = 5;
    double b = 5.5;

    display(a);
    display(b);
    display(a,b);
    ...
}
```

Working of Overloading for display function

The return type of all these function is the same but that need not be the case for function overloading.

DEFAULT ARGUMENTS (PARMETERS)

In C++ programming, we can provide default for functions parameters.

If a function with default arguments is called without passing arguments, then the default parameters are used.

However, if arguments are passed while calling the function, the default arguments are ignored.

Working of default arguments

Case 1: No arguments is passed

```
Void temp(int=10, float = 8.8);

int main() {
.....
temp()
.....
}

Void temp(int i, float f) {
    // code
}
```

Case2: First argument is passed

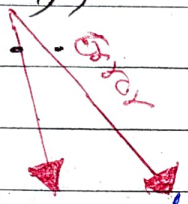
```
Void temp (int=10, float = 8.8);
int main () {
  ..... .....
    temp(6);
  ..... .....
}
Void temp (int i, float f){
    //Code
}
```

Case3: All arguments are passed.

```
Void temp(int =10, float = 8.8);
int main (){
  ..... .....
    temp (6, -2.3);
  ..... .....
}
Void temp (int i , float f) {
    //Code
}
```

Case 4: Second argument is passed.

```
Void temp (int =10, float=8.8);
int main() {
.....
temp (3.4);
.....
}
Void temp (int i, float f) {
//code
}
```



STORAGE CLASS

Type specifies the type of data that can be stored in a variable. for example: int, float, char. etc.

And, Storage Class controls two different properties of a variable: lifetime (determines how long a variable can exist) and scope (determines which part of the program can access it).

Depending upon the storage class of a variable, it can be divided into 4 major types:

- Local Variable
- Global Variable
- Static Local Variable
- Register Variable (Deprected in C++ 11):
- Thread Local Variable

Local Variable

A variable defined inside a function (defined inside function body between braces) is called a local variable.

Ex-
```
int main() {
    // local variable to main()
    int var = 5;
```

Global Variable

If a variable is defined outside of all functions, then it is called a global variable.
The scope of a global variable is the whole program. This means it can be used and changed at any part of the program after its declaration.

Ex-
```
#include <iostream>
using namespace std;

//Global Variable declaration.
int c = 12;

void test();
int main(){
    ++c
    // output 13
    cout << c << endl;
    test();
    return 0;
}
```

Static local variable

Keyword Static is used for specifying a static variable. for Example.

```
.....
int main()
{
    static float a;
    ... ....
}
```

A static local variable exists only inside a function where it is declared (Similar to a local variable) but its lifetime starts when the function is called and ends only when the program ends.

The main difference between local variable and static variable is that, the value of static variable persists the end of the program.

Ex:-

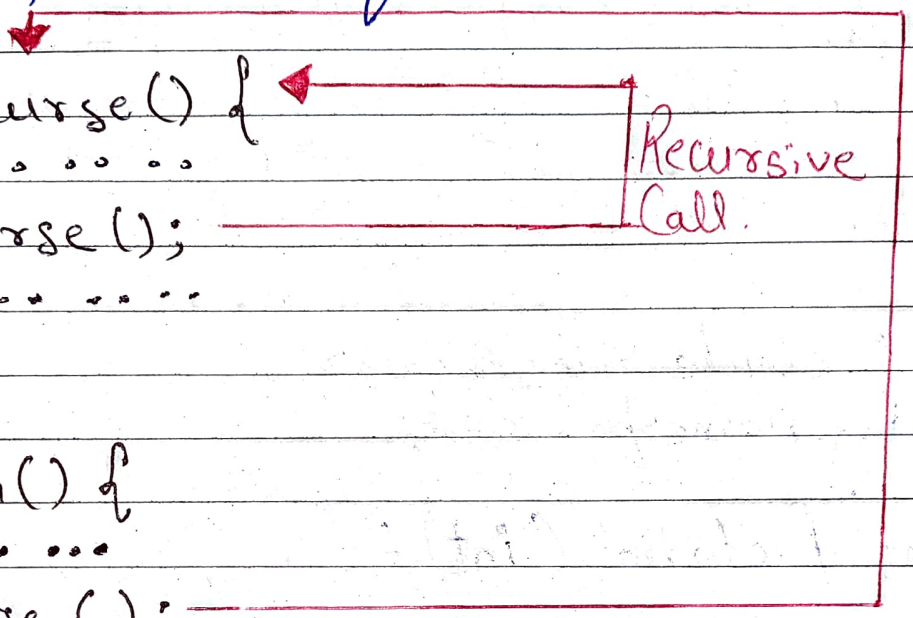
```
void test() {
    //Var is a static variable
    static int Var = 0;
    ++Var;
    Cout << Var << endl;
}
int main() {
    test();
    test();
    return 0;
}
```

output
1
2

Output if Var is not static variable
1
1

RECURSION

A function that Calls itself is Known as a recursive function. And, this technique is Known as Recursion.



The figure shows how recursion works by calling itself over and over again.

The Recursion Continues Until Some Condition is met.

To Prevent infinite recursion, if-else statement (or similar approach) can be Used where one branch makes the recursive call and the other doesn't.

Ex:- Factorial of a Number Using Recursion.

```
// Step by step Calculation of factorial(4)
// factorial(4) = 4 * factorial(3);
// factorial(4) = 4*3* factorial(2);
// factorial(4) = 4*3*2* factorial(1);
// factorial(4) = 4*3*2*1;
// factorial(4) = 24;

#include<iostream>
Using namespace std;
int factorial(int);

int main(){
  int n, result;

Cout<< "Enter a non-negative number:";
Cin >> n;

//Factorial of a number:
// 6! = 6*5*4*3*2*1 = 720
// 0! = 1 by defination
// 1! = 1 by defination
// n! = n*(n-1)!
   result = factorial(n);
Cout; Cout <<" Factorial of " << n << " = "<<result
Cout result return 0;
}
```

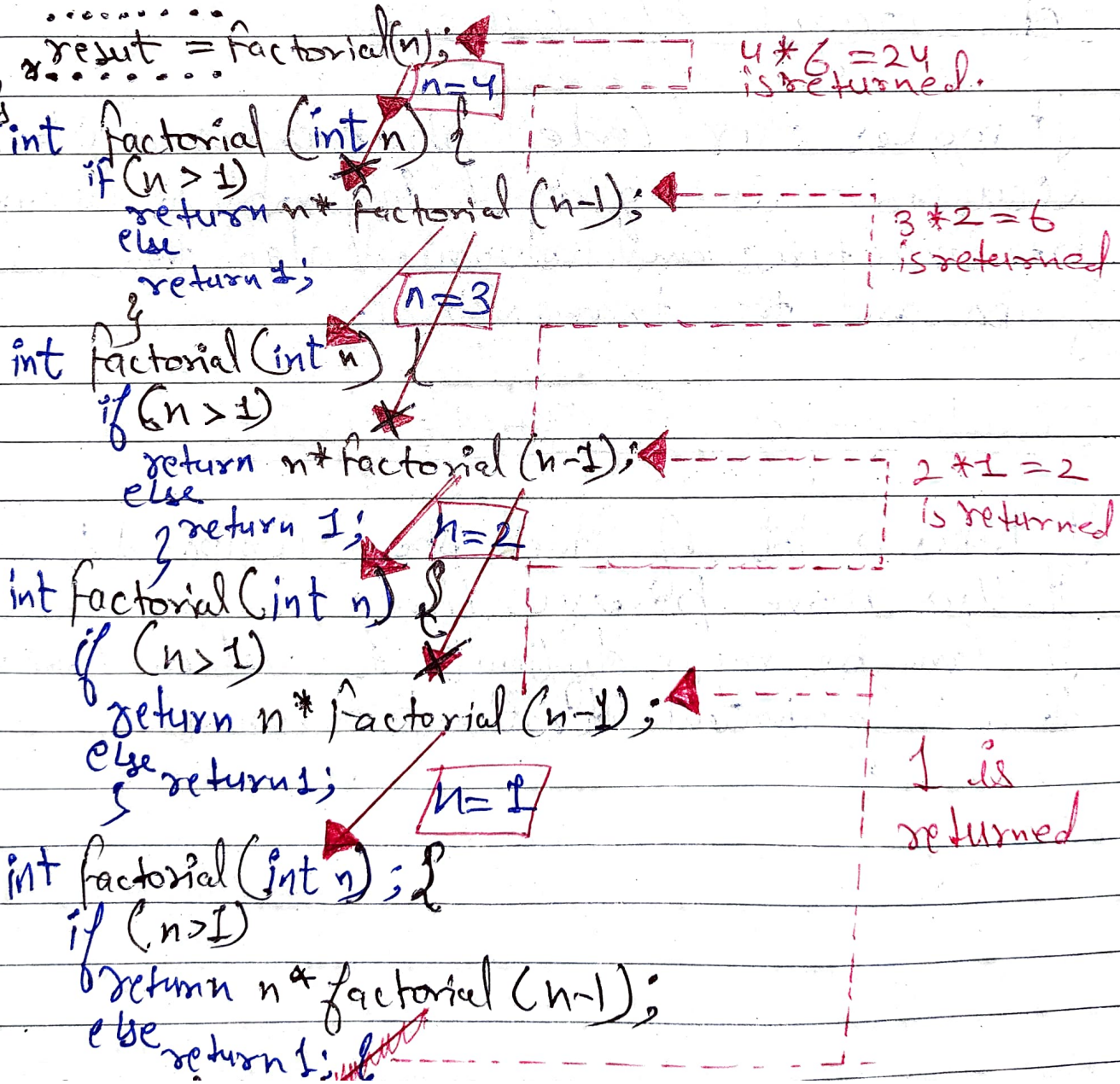
```
int factorial(int n) {
    if (n > 1) {
        return n * factorial(n-1);
    }
    else {
        return 1;
    }
}
```

Output

```
Enter a non-negative numbo: 4
Factorial of 4 = 24
```

Working of factorial program

```
int main() {
    .....
    result = factorial(n);
    .....
}
int factorial (int n) {
    if (n > 1)
        return n* factorial (n-1);
    else
        return 1;
}
int factorial (int n) {
    if (n > 1)
        return n* factorial (n-1);
    else
        return 1;
}
int factorial (int n) {
    if (n > 1)
        return n* factorial (n-1);
    else
        return 1;
}
int factorial (int n) {
    if (n > 1)
        return n* factorial (n-1);
    else
        return 1;
}
```



As we can see, the factorial() function is calling itself. However, during each call, we have decreased the value of n by 1. when n is less than 1, the factorial() function ultimately return the output.

Advantages and disadvantages of recursion

Below are the pros and cons of using recursion in C++.

Advantages of Recursion

- It makes our code shorter and cleaner
- Recursion is required in problems concerning data structures and advanced algorithms, such as Graph and tree transversal.

Disadvantages of C++ Recursion

- It takes a lot of stack space compared to an iterative program.
- It uses more processor time.
- It can be more difficult to debug compared to an equivalent iterative program.

ARRAYS

An Array is a variable that can store multiple values of the same type.
For Example:-
Suppose a class has 27 students, and we need to store the grades of all of them. Instead of creating 27 seperate variables, we can simply create an array.

```
double grade[27];
```

Here grade is an array that can hold a maximum of 27 elements of double type.

Arrays Declaration

```
data type arrayName[array size];
```

Ex:-
```
int x[6];
```

- int - type of element to be stored
- x - name of the array
- 6 - size of the array

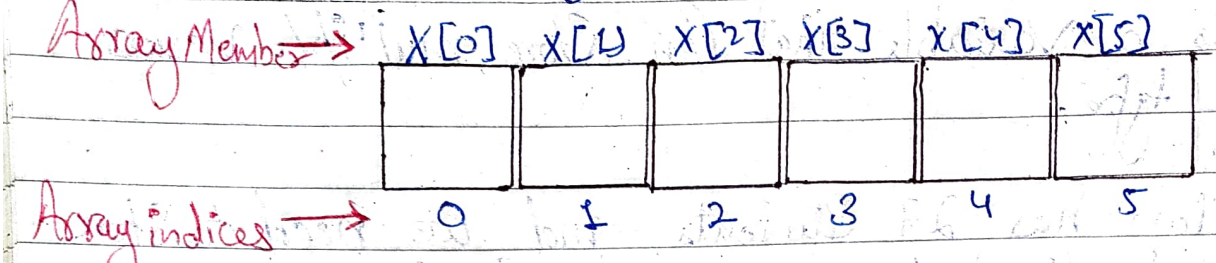
Access Elements in C++ Array

Each element in an Array is associated with a number. The number is know as Array index. We can access elements of an array by those indices.

```
//Syntax of an array Elements

array[index];
```

Consider an Array[X] we have seen.

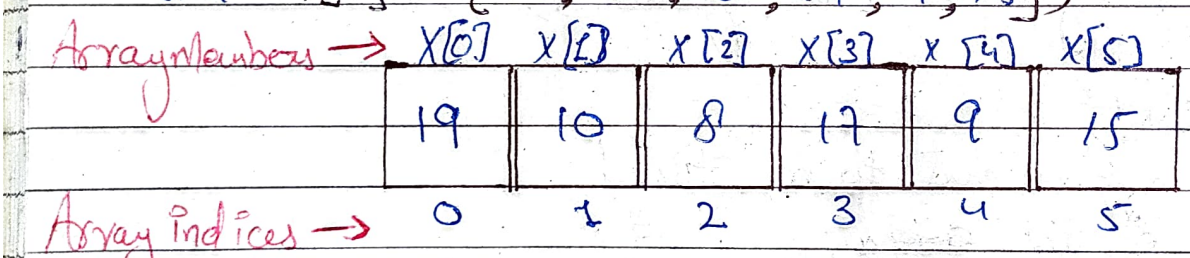


Elements of an Array in C++

Array Initialization

In C++, its possible to initialize an array during declaration. For Example:-

```
//declare and initialize and array.
int x[6] = {19, 10, 8, 17, 9, 15};
```



Array Elements and their data.

Anothe method to initialize array during declaration:

```
//declare and initialize an array
int x[] = {19, 10, 8, 17, 9, 15};
```

Array Out of Bounds

If we declare an array of size 10, then the array will contain elements from index 0 to 9.
However, if we try to access the element at index 10 or more than 10. It will result Undefined Behaviour.

Multi dimensional Arrays

In C++, we can create an array of an array, known as a multidimensional array. For Ex:-

```
int x[3][4];
    (Row)x(Col)
```

x is a two dimensional array. It can hold a maximum of 12 elements.

We can think of this array as a table with 3 rows and each row has 4 columns.

	Col 1	Col 2	Col 3	Col 4
Row 1	x[0][0]	x[0][1]	x[0][2]	x[0][3]
Row 2	x[1][0]	x[1][1]	x[1][2]	x[1][3]
Row 3	x[2][0]	x[2][1]	x[2][2]	x[2][3]

Element in two dimensional Array in C++ programming.

Three dimensional arrays also work in a similar way. For Example.

```
float x[2][4][3];
    (No.of)(R x C)
```

This Array [x] can hold a maximum of 24 elements.

We can find out the total number of elements by multiplying its dimensions:-

2 x 4 x 3 = 24

Multidimensional Array initialization.

Like a normal array we can initialize a multidimensional array in more than one way.

1. Initialization of two dimensional array

```
int test[2][3] = {2,4,5,9,0,19}; -> // Not preferred Method.
int test[2][3] = {{2,4,5},{9,0,19}}; -> // a better way to initialize
```

This array has 2 rows and 3 columns, which is why we have two rows of elements with 3 elements each.

	Col 1	Col 2	Col 3
Row 1	2	4	5
Row 2	9	0	19

Initialization of two-dimensional array in C++

2. Initialization of three-dimensional array.

```
int test[2][3][4] = {3,4,2,3,0,-3,9,11,23,12,23,
                     2,13,4,56,3,5,9,3,5,1,4,9};
```

=> Not a good to Initialize an array

=> A Better way to initialize an array.

```
int test[2][3][4] = {
                     {{3,4,2,13},{0,-3,9,11},
                      {23,12,23,2}},
                     {{13,4,56,3},{5,9,3,5},{5,1,4,9}}
                    };
```

Notice the dimension of this three-dimensional array.

The first dimension has the value 2. So, the two elements comprising the first dimension are:

Element 1 = {{3, 4, 2, 3}, {0, -3, 9, 11}, {23, 12, 23, 2}}
Element 2 = {{13, 4, 56, 3}, {5, 9, 3, 5}, {5, 1, 4, 9}}

The second dimension has the value 3. Notice that each elements of the first dimension has three elements each.

{3,4,2,3}, {0,-3,9,11} and {23,12,23,2} for Element 1.
{13,4,56,3}, {5,9,3,5} and {5,1,4,9} for Element 2.

Finally there are four int members inside each of the elements of the second dimension.

{3, 4, 2, 3}
{0, -3, 9, 11}
... ...
... ...

Passing Arrays to a function.

Syntax for passing Arrays as function Parameters.

The syntax for passing an array to a function is:

returntype Functionname (datatype arrayName[arraySize])
{ //Code.
}

lets See Example.

```
int total (int marks[5]) {
  //Code
```

Strings

String is a Collection of Characters. There are two types of Strings Commonly Used in C++ programming Language.

- String that are objects of string Class (The Standard C++ library string Class)
- C-Strings (C-style Strings)

C-Strings

In C programming, the Collection of Characters is stored in the form of Arrays. This is also Supported in C++ Programming. Hence it's called C-Strings.

C-Strings are arrays of type Char terminated with a null Character, that is, '\0' (ASCII Value of null Character is 0)

How to define C-String?

```
Char Str[] = "C++";
```

In the Above Code, Str is a String and it holds 4 Characters.
Although, "C++" has 3 Characters, the Null Character '\0' is added to the end of the String automatically.
Alternative ways of Defining a String.

```
Char Str[4] = "C++";
Char Str[] = {'C', '+', '+', '\0'};
Char Str[4] = {'C', '+', '+', '\0'};
```

Like Array, it is not necessary to use all the Space allocated for the String. For Ex:— `Char Str[100] = "C++";`

String Object

In C++, you can also create a string object for holding strings.

Unlike using char arrays, string objects has no fixed length and can be extended as per your requirement.

Example

C++ String Using string data type

```
#include <iostream>
Using namespace std;

int main() {
    //Declaring a string object.
    string str;
    cout<<"Enter a String: ";
    getline(cin, str);

    Cout << "You Entered: " << str << endl;
    return 0;
}
```

Output

Enter a String: programming is fun.
You Entered: programming is fun.

In this program, Instead of Using Cin>> or cin.get() function. You can get the entered line of text Using getline().
getline() function takes the input stream as the first parameter which is cin and str as the location of the line to be stored.

Display a String Entered by a User.

```
int main() {
    Char str[100];
    Cout << "Enter a String:";
    Cin >> str;
    Cout << "You Entered" << str << endl;
```

Output:- Enter a String: Programming is fun.
You Entered: Programming.

In this example only "Programming is displayed instead of "Programming is fun". This is because the Extraction operator >> works as scanf() in C and Considers a Space " " has a terminating object.

Program to read and display an entire line Entered by User.

```
int main() {
    Char str[100];
    Cout << "Enter a String:";
    Cin.get(str, 100);
    Cout << "You entered :" << str << endl;
}
```

Output:- Enter a String: Programming is fun.
You Entered: Programming is fun.

To Read the text Containing blank space, cin.get function can be used. This function takes two arguments.
First argument is the name of the String (address of first Element of String) and Second arguments is the maximum Size of the array.

Address in POINTERS

In C++, pointers are Variables that store the memory addresses of other Variables.

Address in C++

If we have Variable Var in our program, &Var will give us its address in the memory.

C++ pointers

Pointers are Used to Store addresses rather than Values. Here, is how we declare pointers.

```
int *pointVar;
```

We Can also declare pointers in the following way.

```
int* pointVar; // preferred syntax
```

Another Example of declaring pointers-

```
int* pointVar, p;
```

Here, we have declare a pointer pointVar and a normal Variable p.

Note: The * operator is Used after the data type to declare pointers.

Assigning Addresses to pointers.

Here is How we assign addresses to pointers.

```
int* pointVar, Var;
Var = 5;
// assign address of Var to pointVar pointer
pointVar = &Var;
```

Here, 5 is assigned to the variable Var. And, the address of Var is assigned to the pointVar pointer with the code pointVar = &Var.

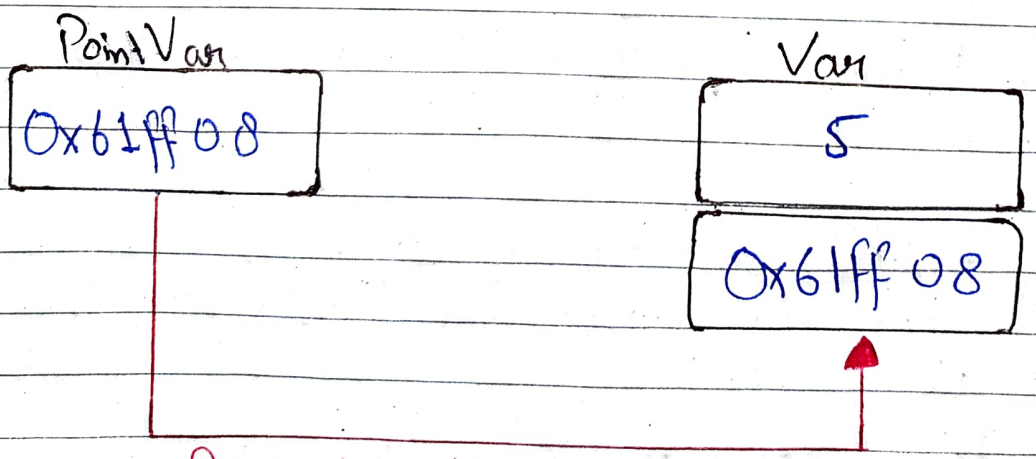
Get the Value from the Address Using pointers

To get the value pointed by a pointer, we use the (*) operator. For Ex:-

```
int* pointVar, Var;
Var = 5;
//assign address of var to pointVar
pointVar = &Var;
//access value pointed by pointVar
Cout << *pointVar << endl;   // Output: 5.
```

In the above code, when (*) is used with pointers, its called the dereference operator. It operates on a pointer and gives the value pointed by the address stored in the pointer. That is *pointVar = Var.

Note:- In C++, pointVar and *pointVar is completely different we cannot do something like *pointVar = &Var.



Working of C++ pointers:-

Changing Value pointed by pointers

If pointvar points to the address of var, we can change the value of var by using *pointvar.

For Example:-

```
int var = 5;
int *, pointvar;

//assign address of var.
pointvar = &var;
//change value at address pointvar.
*pointvar = 1;

Cout << var << endl; //Output: 1
```

Here, pointvar and &var have the same address, the value of var will also be changed when *pointvar is changed.

Common Mistakes when working with pointers:

Suppose, we want a pointer varpoint to point to the address of var. Then,

```
int var, *varpoint;
//wrong!
//Varpoint is an address but Var is not.
Varpoint = var;
//wrong!
// &Var is an address
// *Varpoint is the value stored in &var.
*Varpoint = &Var;
```

```
//Correct!
//Varpoint is an address and so is &Var.
Varpoint = &Var;
//Correct!
// both *VarPoint and Var are Values.
*Varpoint = Var;
```

POINTERS AND ARRAYS

Pointer can also store the address of cells of an array. Ex:-

```
int *ptr;
int arr[5];

// Store the address of the first
// element of arr in ptr.
ptr = arr;
```

Here, ptr is a pointer variable while arr is an int array. The code ptr = arr; Stores the address of the first element of the array in variable ptr.

Notice that arr and &arr[0] are same. So the code below same as the code above.

```
int *ptr;
int arr[5];
ptr = &arr[0];
```

The addresses for the rest of the array elements are given by &arr[1], &arr[2], &arr[3] and &arr[4].

Points To Every Array Elements

Suppose we need to point to the fourth element of the array using the same pointer ptr.
Here, if ptr point to the first Element in the above Example then ptr+3 will point to the fourth element. For Example,

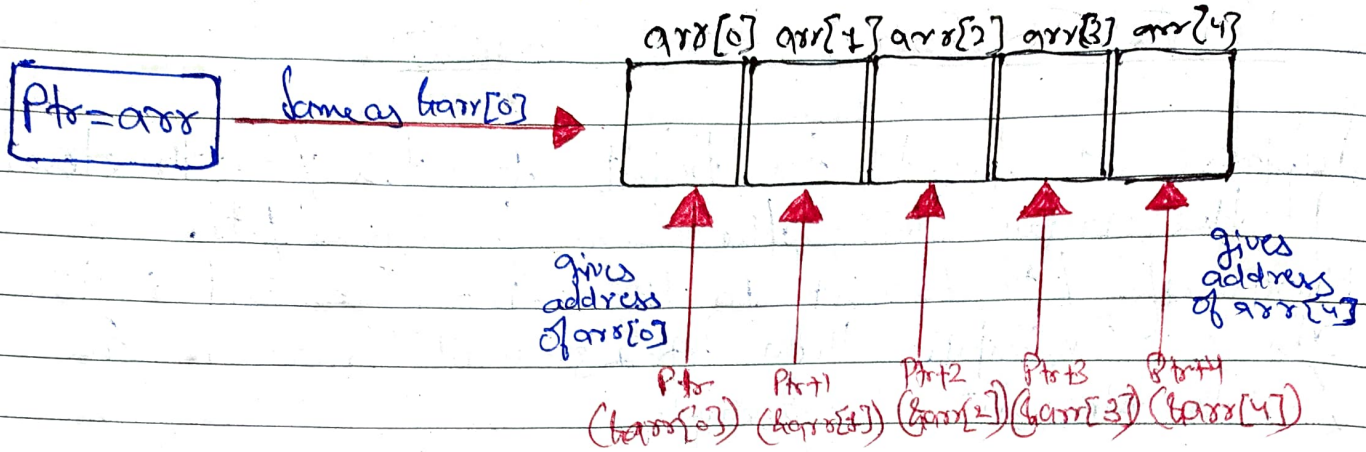
```
int *ptr;
int arr[5];
ptr = arr;
```

ptr+1 is Equivalent to &arr[1];
ptr+2 " " " &arr[2];
ptr+3 " " " &arr[3];
ptr+4 " " " &arr[4];

Similarly we can access element using the single pointer. For Example,

```
//Use dereference operator.
*ptr == arr[0];
*(ptr+1) is Equivalent to arr[1];
*(ptr+2) is Equivalent to arr[2];
*(ptr+3) is equivalent to arr[3];
*(ptr+4) is Equivalent to arr[4];
```

Suppose if we have initialized ptr = &arr[2]; then
ptr-2 is Equivalent to &arr[0];
ptr-1 " " " &arr[1];
ptr+1 " " " &arr[3];
ptr+2 " " " &arr[4];



Working of C++ pointers with arrays.

We learned about passing ~~values~~ arguments to a function. this method used is called passing by value because the actual value is passed.

However, there is another way of passing arguments to a function where the actual values of arguments are not passed. Instead, the reference to values is passed.

For Ex:-

```
// function that takes value as parameter
void func1(int numVal) {
    // code
}
// function that takes reference as parameter
// notice the & before the parameter.
void func2(int &numRef) {
    // code
}

int main() {
    int num = 5;
```

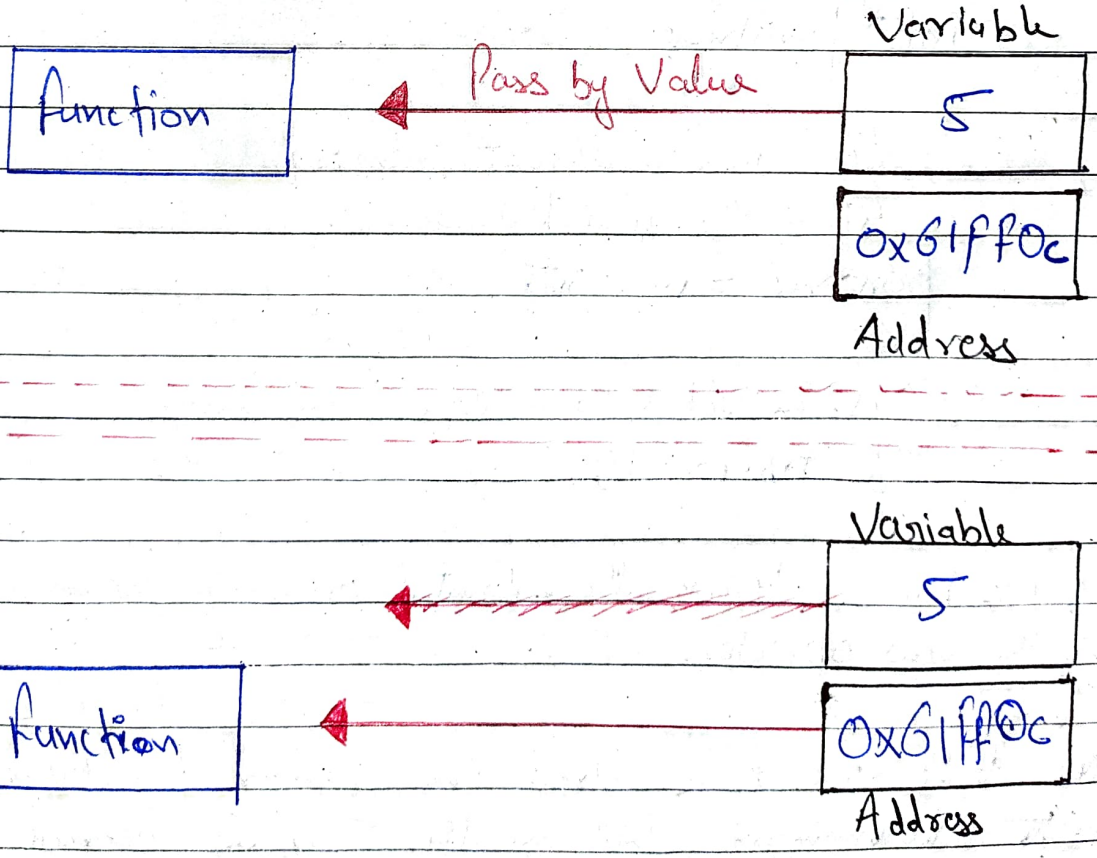
```
//Pass by Value
func1(num);

//Pass by reference;
func2(num);

return 0;
}
```

Notice the & in Void func2(int &numRef). This denotes that we are Using the address of the Variable as our parameter.

So, when we Call the func2() function in main() by passing the variable num as an argument, we are actually passing the address of num Variable instead of the Value 5.



Pass by Value vs. Pass by Reference.

Memory Management: new and delete.

C++ allows us to allocate the memory of a variable or an array in runtime. this is known as dynamic memory allocation.

In C++, we need to deallocate the dynamically allocated memory manually after we have no use for the variable.

We can allocate and then deallocate memory dynamically using the new and delete operators respectively.

C++ new Operator

The new operator allocates memory to a variable.

For Ex:-

```
//declare an int pointer
int* pointVar;

// dynamically allocate memory.
// Using the new keyword.
pointVar = new int;

//assign Value to allocated memory.
*pointVar = 45;
```

Here, we have dynamically allocated memory for an int variable using the new operator.

Notice that we have used the pointer pointVar to allocate the memory dynamically. This is because the new operator returns the address of the memory location.

In the case of an array, the new operator returns the address of the first element of the Array.

The Syntax for Using the new operator is:

pointerVariable = new datatype;

Once we no longer need to use a variable that we have declared dynamically, we can deallocate the memory occupied by the variable.

For this the delete operator is used. It returns the memory to the operating system. This is known as memory deallocation.

The Syntax for this operator is:

delete pointerVariable;

```
//assign value the variable memory
*pointVar = 45;
//Print the value stored in memory
Cout ce *pointVar; //output: 45.

//deallocate the memory
delete pointVar;
```

After printing the Content of pointVar, we deallocated the memory Using delete.

STRUCTURES

Structure is Collection of Variables of different data types Under a Single name. It is similar to a Class in that, both holds a Collection of data of different data types.

This Collection of releated information Under a single name is a Structure.

How to declare a Structure in C++ programming?

```
Struct Person
{
    Char name[50];
    int age;
    float Salary;
};
```

Note: Remember to end the declaration with a Semicolon(;)

Here a Structure person is defined which has three members: name, age and Salary.

How to define a Structure Variable?

Once we declare a structure person as above. you can define a structure Variable as:

```
Person bill;
```

Here, a Structure Variable bill is defined which is of

type Structure Person.
When Structure variable is defined, only then the required memory is allocated by the Compiler.

How to access members of a Structure?

The member of Structure variable is accessed using a dot (.) operator.

Suppose, you want to access age of Structure variable bill and assign it 50 to it.
You can do this by using following code below:-

```
bill.age = 50;
```

Ex:- C++ Structure.
C++ program to assign data to members of a Structure variable and display it.

```
struct person {
    char name[50];
    int age;
    float Salary;
};

int main() {
    Person P1;
    Cout << "Enter full name: ";
    Cin.get(P1.name, 50);
    Cout << "Enter age: ";
    Cin >> P1.age;
    Cout << "Enter Salary: ";
    cin >> P1.Salary;
```

```
cout << "\n Displaying Information." << endl;
cout << "Name: " << p1.name << endl;
cout << "Age: " << p1.age << endl;
cout << "Salary: " << p1.salary;

return 0; }
```

Output

```
Enter Full name: Mohd Raza
Enter Age: 21
Enter Salary: 1024.4

Displaying Information.
Name: Mohd Raza
Age: 21
Salary: 1024.4
```

Passing Structure to function in C++.

A Structure Variable Can be passed to a function in similar way as normal arguments.

Ex: C++ Structure and function

```
- - - - - - - -
void displaydata(Person); // function declaration.

int main() {
    Person P;
- - - - - - -
- - - - - - -
// function Call with Structure variable as an argument.
    displaydata(P);
    return 0; }
```

```
void displaydata(person p) {
  cout << "\n Displaying Information." << endl;
  cout << "Name: " << p.name << endl;
  cout << "Age: " << p.age << endl;
  cout << "Salary: " << p.salary;
  }
```

Pointers to Structures

A pointer variable can be created not only for native type like (int, float, double etc.) but they can also be created for user defined types like structure.

Here is how you can create pointer for structures:

```
#include <iostream>
using namespace std;

struct temp {
    int i;
    float f;
};
int main() {
    temp *ptr;
    return 0;
}
```

This program creates a pointer ptr of type structure temp.

Enumeration

An Enumeration is a User-defined data type that Consists of integral Constants. to define an Enumeration, Keyword Enum is Used.

Syntax:

```
Enum Season { Spring, Summer, autumn, winter};
```

Here, the name of the enumeration is Season.

And, Spring, Summer and winter are values of type Season.

By default, Spring is 0, Summer is 1 and so on. You Can change the default Value of an enum element during declaration (if necessary).

```
Enum Season
{   Spring = 0,
    Summer = 4,
    Autumn = 8,
    Winter = 12 :
};
```

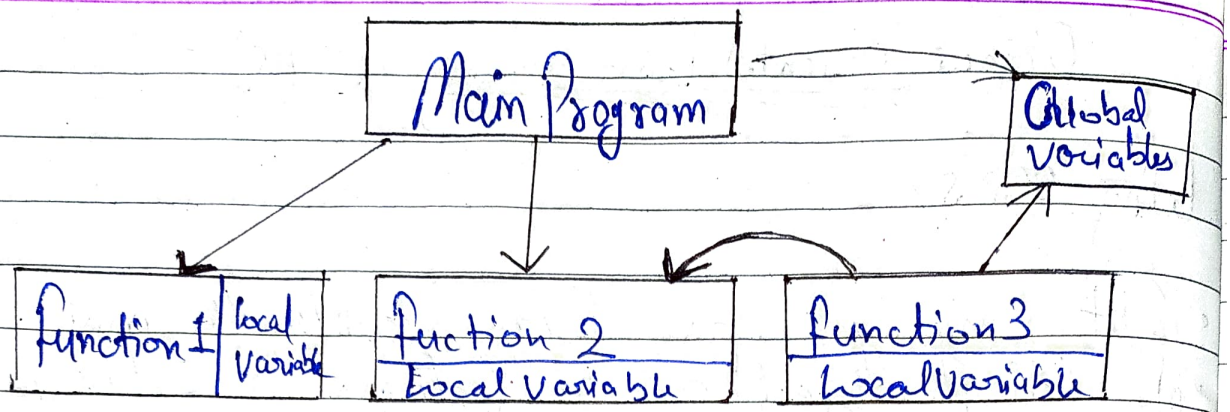
Object Oriented programming (OOPS)

Why OOPs?

- C++ language was designed with the main intention of adding object-oriented features to C language.
- As the size of program increases, readability, maintainability and bug free nature of program decreases.
- This was the major problem with language like C which relied upon functions or procedures (hence the name procedural programming language)
- As a result, the possibility of not addressing the problem in an effective manner was high.
- Also, as data was almost ~~neglected~~ neglected, data security was easily compromised.
- Using classes solves this problem by modelling program as a real world scenario.

Procedure Oriented Programming

- Consists of writing a set of instructions for the computer to follow.
- Main focus is on functions and not on flow of data.
- Functions can either use local or global data.
- Data moves openly from function to function.



FLOW DIAGRAM OF PROCEDURAL PROGRAMMING LANGUAGE.

OBJECT ORIENTED PRORGRAMMIG

- Works on the concept of Classes and objects.
- A Class is a template to Create objects.
- Treats data as a critical element
- Decomposes the problem in objects and builds data and functions around the objects.

BASIC CONCEPT IN OBJECT ORIENTED PROGRAMMING

- Classes - Basic template for Creating Objects.
- Objects - Basic runtime Entities.
- Data Abstraction & Encapsulation - Wrapping data and functions into Single unit.
- Inheritance - Properties of one Class Can be inherited into others
- Polymorphism - Ability to take more than one forms
- Dynamic Binding - Code which will execute is not known Until the Program runs.
- Message passing - Object. Message (Information) Call format.

BENEFITS OF OBJECT ORIENTED PROGRAMMING

- Better Code reusability Using objects and inheritance.
- Principle of data ~~binding~~ hiding helps build secure System.
- Multiple objects Can Co-exist without any interference.
- Software Complexity Can be Easily managed.

Why Uses Classes instead of Structures

Classes and Structures are Somewhat the Same but still, they have Some differences. For Example, we Cannot hide data in structures which means that Everything is public and Can be accessed Easily which is a major drawback of the Structure because Structures Cannot be Used where data Security is a major Concern. Another drawback of Structures is that we Cannot add functions in it.

Classes and Objects

Suppose we need to Store the length, breadth and height of a rectangular room and Calculate its area and Volume.

To handle this task, we Can Create three Variables, Say, length, breadth, and height along with the functions Calculate Area() and Calculate Volume().

However, in C++, rather than Creating Separate Variables and functions, we Can also wrap these related data and functions in a Single place (by Creating object). This programming is known as object-oriented programming.

C++ Classes

or (template)

A Class is a blueprint for the object. Classes consists of variables and functions which are also called Class members. We can think of a Class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows, etc. Based one these description we build the house. House is the object.

Create a Class

```
Class ClassName {
    // Some data
    // Some functions
};
```

for Example;

```
Class Room {
    public:
        double length;
        double breadth;
        double height;

        double CalculateArea() {
            return length * breadth;
        }

        double CalculateVolume() {
            return length * breadth * height;
        }
};
```

Here, we defined a class named Room.

The Variables length, breadth, and height declared inside the class are known as data members. And the, the functions Calculate Area() and calculate Volume() are known as member functions of a class.

Public Access Modifiers in C++

All the Variables and functions declared Under public access modifiers will be avalaible for Everyone. They can be accessed both inside and outside the class. Dot(.) operator is used in the program to access public data members directly.

Private Access Modifiers in C++

All the Variables and functions declared Under a private Access modifiers can only be used inside the class. They are not permissible to be used by any object or function outside the class.

Objects

When a Class is defined, only the Specification for the objects is defined; no memory or storage is allocated.

To use the data and access functions defined in the class, we need to Create objects.

Syntax to Define object in C++

ClassName ObjectVariableName;

We Can Create objects of Room Class (defined in the above example) as follows.

```
//Sample function
void SampleFunction() {
    //Create objects
    Room room1, room2;
}

int main() {
    //Create objects
    Room room3, room4;
}
```

As we Can See we can Create objects of a Class in any function of the program. We Can also create objects of a Class within the Class itself, or in other Classes.
Also, we Can Create as many objects as we want from a single Class.

Access Data Members and Member functions.

We Can Access the data members and member functions of a Class by using a (.) (dot) operator. for Example:-

```
room2.CalculateArea();
```

This will Call the CalculateArea() functions inside the Room Class for objects room2.
Similarly, the Data members Can be accessed as:

```
room1.length = 5.5;
```

In this Case, it initializes the length Variable of room1 to 5.5.

We can also create private ~~keywords~~ members using the private keyword. The private members of a class can only be accessed from with in the class.

For Ex:-

```
Class Test {

    private:
        int a;
        void function1(){}

    Public:
        int b;
        void function2(){}
};
```

Here, (a) and (function1()) are private Thus they cannot be accessed from outside the class.
On the other hand, (b) and (function2()) are accessible from everywhere in the program.

Ex:-

```
Class Room {

    Private:
        double length;
        double Breadth;
        double height;

    Public:
        //function to initialize private Variables.
    void initdata (double len, double Brdth, double hgt) {
            length = len;
            Breadth = Brdth;
            height = hgt;
    }
```

```
---------- }; -> Room Class Bracket Close
int main(){
    Room room1;
    // Pass the Values of private variables as arguments
    room1.initdata(42.5, 30.8, 19.2);
    ----------
    ----------
}
```

Since the Variable are now private, we cannot access them directly from main(). Hence, using the following code would be invalid.

```
// invalid code.
Obj.length = 42.5;
Obj.height = 30.8;
Obj.Breadth = 19.2;
```

NESTING OF MEMBER FUNCTION

If one member function is called inside the other member function of the same class it is called nesting of a member function. A program to demonstrate the nesting of a member function is shown in ref no. Tutorial No. 22.

Objects memory allocation in C++

The way memory is allocated to variables and functions of the class is different even though they both are from the same class.

The memory is only allocated to the variables of the Class when the object is created. The memory is not allocated to the variables when the Class is declared. At the same time, Single variables can have different values for different objects, so Every object has an individual Copy of all the variables of a Class. But the memory is allocated the function only once when the Class is declared. So the objects don't have individual Copies of functions only one Copy is shared among Each objects.

Static data Member in C++

When a Static Data member is Created, there is only a Single Copy of the ~~member~~ data member which is shared between all the objects of the Class. As we have discussed in ~~concept~~ that if the data member are not Static then Every object has an individual Copy of the data member and it is not shared.

Static local Variable

- Concept as it is taken from C
- There are by default initialized to zero
- Their lifetime is throughout the program.

Static Member Variable

- Declared inside the Class body.
- Also know as Class member Variable.
- They must be defined outside the Class.
- Static member Variable does not belong to any object

but, to the whole class.
- There will only be one copy of static member variable for the whole class.
- Any object can use the same copy of Class Variable
- They can also be used with class name.

Static Member function.

- They are qualified with the keyword static.
- They are also called Class member functions
- They can be invoked (called) with or without object.
- They can only access static members of the Class.

CONSTRUCTORS

A Constructor is a special type of members functions that is called automatically when an object is created.

In C++, a Constructor has the same name as that of the Class and it does not have a return type. For Example

```
Class Wall {
public:
    // Create a Constructor.
    wall() {
        //code
    }
};
```

C++ Default Constructor

A Constructor with no parameters is known as a default Constructor. In the example above, Wall() is a default Constructor.

Example:-

```
//declare a class.
Class wall {
private:
    double length;

Public:
  //default Constructor to initialize variable
  wall() {
      length = 5.5;
      Cout << "Creating a Wall." << endl;
      Cout << "Length = " << length << endl;
  }
};

int main() {
Wall wall1;
return 0;
}
```

Output

```
Creating a wall
length = 5.5
```

Here, when the wall1 object is Created, the Wall() Constructor is Called. This Sets the length Variable of the object to 5.5.

C++ Parameterized Constructor.

In C++, a Constructor with parameters is known as a parameterized Constructor. This is the preferred method to initialize member data.

Example:-

```
// C++ program to Calculate the Area of Wall.
Class Wall{
  private:
    double length;
    double height;

  Public:
      // Parameterized Constructor to initialize Variable
  Wall( double len, double hgt){
      length = len;
    } height = hgt;

  double Calculate area() {
      return length * height;
    }
int main(){
    // Create object and initialize data members.
  Wall wall1( 10.5, 8.6);
  Wall wall2( 8.5, 6.3);
  cout << "Area of Wall1: " << wall1.CalculateArea() << endl;
  cout << "Area of Wall2: " << wall2.CalculateArea();
  return 0;
}
```

Output

```
Area of Wall1: 90.3
Area of wall 2: 53.55
```

C++ Copy Constructor.

The Copy Constructor in C++ is used to copy data of one object to another.

- If we want one object to resemble another object we can use a Copy Constructor.
- If no copy Constructor is written in the program Compiler will supply its own Copy Constructor.

Example:-

```
Class Wall {
private:
    double length;
    double height;
Public:
    //Initialize variables with parameterized Constructor
    Wall (doule len, double hgt){
        length = len;
        height = hgt;
    }
    //Copy Constructor with a wall object as parameter
    //Copies data of the obj parameter
    wall (Wall &obj) {
        length = obj.length;
        height = obj.height;
    }
    double CalculateArea () {
        return length * height;
    }
};
int main() {
    //Create an object of Wall Class
    Wall wall1 (10.5, 8.6);
```

```
// Copy Contents of wall1 to wall2
wall wall2 = wall1;

// print areas of wall1 and wall2.
cout << "Area of wall1: " << wall1.calculateArea() <<
cout << "Area of wall2: " << wall2.calculateArea();

return 0;
}
```

Output

```
Area of wall 1: 90.3
Area of wall 2: 90.3
```

Operator Overloading

When an operator is overloaded with multiple jobs, it is known operator overloading.

In C++, we can change the way operators works for User-defined like objects and structures.

Suppose we have created three objects C1, C2, and result from a Class named Complex that represents Complex numbers.

Since Operator overloading allows us to change how operators works we can redefined how the (+) operator works and use it to add the Complex numbers of c1 and c2 by writing the following code.

```
result = c1 + c2;
```

instead of something like.
result = C1.addnumbers(C2);

This makes our code intutive and easy to understand.

Note:- We cannot use operator over loading for fundamental data types like int, float, char and so on.

Syntax of C++ operator overloading.

To overload an operator, we use a special operator function. we define the function inside the class or structure whose object/variables we want to overload operator to work with.

```
Class Classname{
  ---------
  public:
     return type operator Symbol (arguments){
       ---  ---  ---  ---
     }
     ---  ---  ---  ---
};
```

Here,

- return type is the return type of the function.
- operator is a keyword.
- Symbol is the operator we want to overload. like: +, <, -, ++, etc.
- arguments. is the arguments passed to the function.

Operator Overloading in Binary Operators.

Binary Operators works on two operands. For Example.

```
result = num + 9;
```

Here, (+) is a binary operator that works on the operands num and 9.

When we overload the binary operator for User defined types by Using the Code:

```
Obj3 = obj1 + obj2;
```

The operator function is called Using obj1 and obj2 is passed as an arguments to the function.

Example on next page

Operator Overloading in Unary Operators.

Unary operators operate on only one operand. The increament operator (++) and decreament operator (--) are Example of Unary operators.

Example: - - - - - -

```
Class Count {
  private:
    int value;
  Public:
    // Constructor to initialize Count to 5
    Count(): value(5) {}
    // Overload ++ when Used as a prefix.
    Count Operator ++() {
      Count temp;
```

```
// Here, Value is the value Attribute of the Calling object
    temp.Value = ++value;
    return temp;
}
//Overload ++ when used as a prefix.
Count Operator ++ (int) {
    Count temp;

// Here Value is the value attribute of the Calling object
    temp.value = value++;

    return temp;
}

Void Display () {
    Count << "Count: " << Value << endl;
}
};

int main() {
    Count count1, &result;
//Call the "Count operator ++()" function.
    result = ++count1;
    result.display();

//Call the "Count operator ++(int)" function.
    result = Count1++;
    result.display();
    return 0;
}
```

Output

```
Count: 6
Count: 6.
```

Binary Operator Example:-

```
Class Complex
{
    int a, b;
public:
    void Setdata(int x, int y)
    { a = x;
      b = y; }
    void Showdata()
    { cout << "a= " << a << endl;
           << "b= " << b;

/* Complex add(Complex c)
    {
      Complex temp;
      temp.a = a+c.a;
      temp.b = b + c.b;
      return (temp);
    } */

Complex Operator +(Complex c)
  {
    Complex temp;
    temp.a = a + c.a;
    temp.b = b + c.b;
    return (temp);
  }
};
```

```
int main()
{ Complex c1, c2, c3;

  c1.Setdata(3,4);
  c2.Setdata(5,6);

// c3 = c1 + c2; → Give Error! (+) operator is not defined in non-premitive type.
// c3 = c1.add(c2); → we can make a function of Complex type to add c1 and c2.
// c3 = c1.operator+(c2); → we can use operator keyword to make a function of operator (+) name. and give c2 as arguments.

  c3 = c1 + c2; // → Now we can write this it will not give any error because we defined (+) operator in function.
  c3.Showdata();
  return 0;
}
```

Output

a = 8
b = 10

Things to remember in operator overloading

(=) and (&)

1. Two Operator are already overloaded by default in C++. For eg Copy objects of the same class. we directly use the (=) operator.
2. Operator overloading cannot change the precedency and associativity of operator.
3. There are 4 operator that cannot be overloaded in C++.
   a. (::) (Scope resolution)
   b. (.) (member Selection)
   c. (.*) (member Selection through pointer to function)
   d. (?:) (ternary operator

INHERITACE

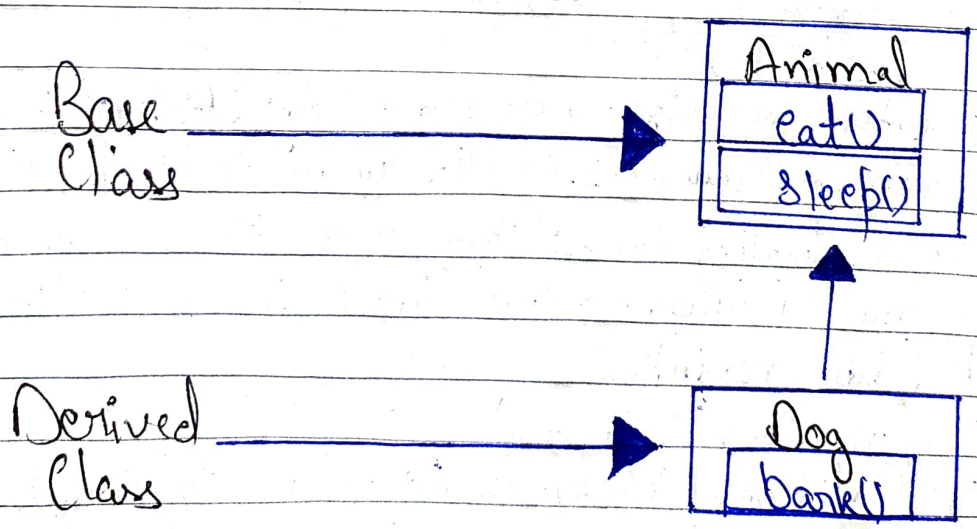
Inheritance is one of the Key features of Object-Oriented programming in C++. It allows us to Create a new class (derived class) from an existing class (base class).

The derived Class inherits the features from the base Class and Can have additional features of its own. For example,

```
Class Animal {
    // eat() function
    // sleep() function
};

Class Dog : public Animal {
    // bark() function.
};
```

Here, the Dog class is derived from Animal Class. Since Dog is derived from Animal, members of Animal are accessible to Dog.



Inheritance is an is-a relationship. We use inheritance only if an is-a relationship is present between the two classes.
Here are some example:

- A Car is a Vehicle
- Orange is a fruit.
- A Surgeon is a doctor.
- A dog is an animal.

Overview of Inheritence

- Reusability is a very important of oop's.
- In C++ we can reuse a class and add additional features to it.
- Reusing Class saves time and money.
- Reusing already tested and debugged class will save a lot of effort of developing and debugging the same thing again.
- The Concept of reusability in C++ is supported using Inheritence.
- We can reuse the properties of an existing class by inheriting from it!
- There are different types of inheritence in C++.
  - Single Inheritance
  - Multiple Inheritence
  - Multilevel Inheritance
  - Hierarchical Inheritance
  - Hybrid Inheritance

C++ protected Members

The access modifier protected is especially relevant when it comes to C++ inheritance.

Like private members, protected members are inaccessible outside of the class. However, they can be accessed by derived classes and friend classes/functions.

We need protected members if we want to hide the data of a class, but still want that data to be inherited by its derived class.

Access/Modes in C++ Inheritance.

The various ways we can derive classes are known as access modes. these access modes have the following effect:

- Default access visibility mode is private.

1- Public: Public members of the base class becomes Public members of the derived class.

2- Private: In this case, ~~all the~~ Public members of the base class become private members in the derived class.

3- protected: The public members of the base class become protected members in the derived class.

Private members are never inherited.

	Public derivation	Private derivation	Protected derivation
1. Private members	Not Inherited	Not Inherited	Not Inherited
2. Protected members	Protected	Private	Protected
3. Public members	Public	Private	Protected

Member function Overriding in Inheritance

Suppose, base class and derived class have members functions with the same name and arguments.

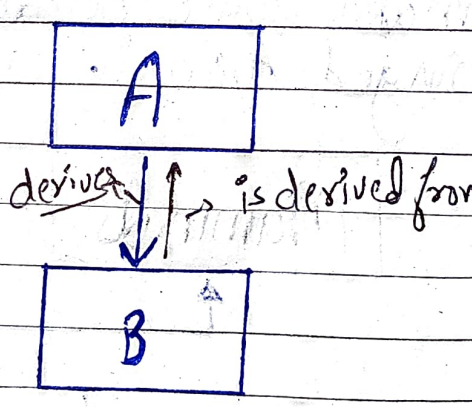
If we create an object of the derived class and try to access that member function, the member function in the derived class is invoked instead of the one in the base class.

The member function of the derived class overrides the member function of base class.

INHERITENCE TYPES

SINGLE INHERITANCE

- A derived class with only one base class.



• MULTILEVEL INHERITANCE

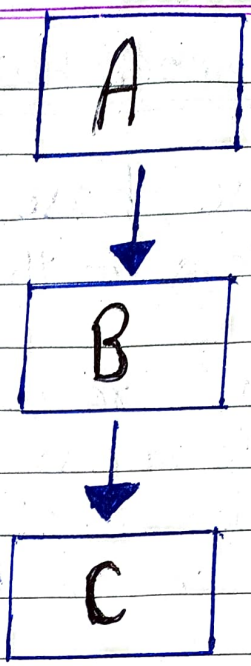
In C++ programming, not only you can derive a class from the base class but you can also derive a class from the derived class. This form of inheritance is known as multilevel inheritance.

Ex: Class A {

};
Class B : public A {

};
Class C : public B {

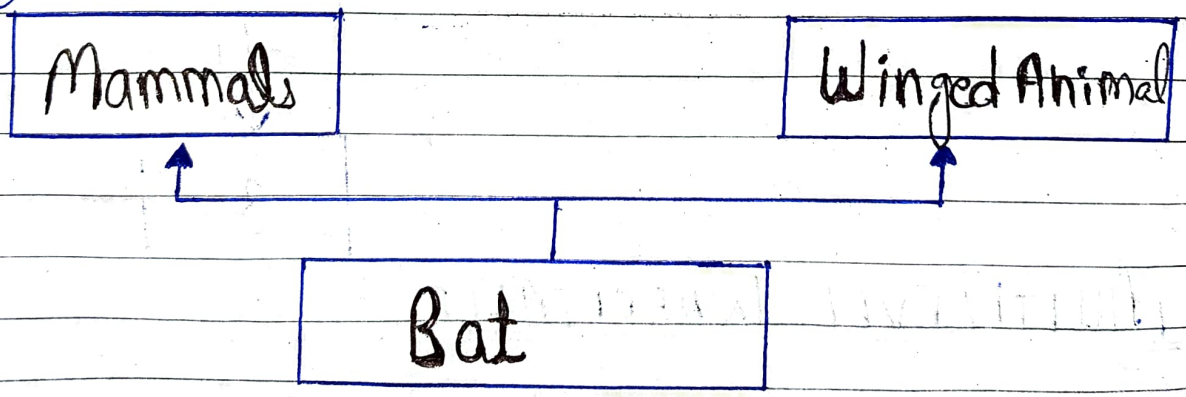
};



Here, Class B is derived from the Base Class A and the Class C is derived from the derived class B.

MULTIPLE INHERITANCE

In C++ programming, a Class can be derived from more than one parent. For Example, A Class Bat is derived from base Classes Mammal and Winged Animal. It makes sense because bat is mammal as well as a winged Animal.



Multiple Inheritence

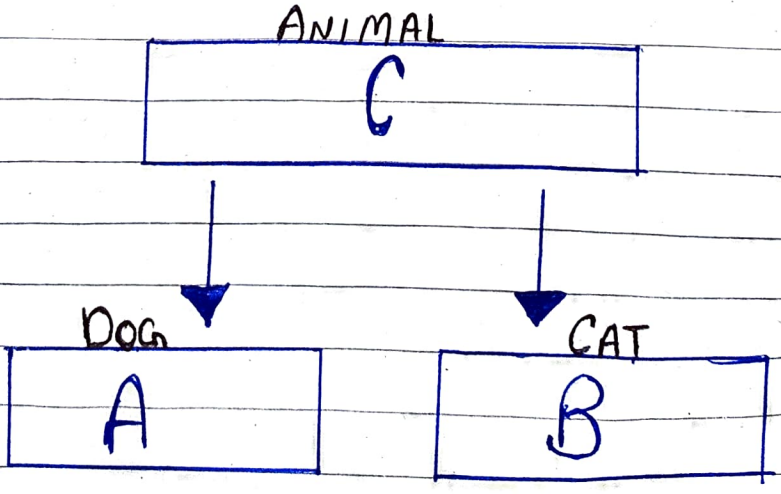
C++ Hierarchical Inheritance

If more than one class is inherited from the base class, it's known as hierarchical inheritance. In hierarchical inheritance, all features that are common in child classes are included in the base class.

For Example, Physics, Chemistry, Biology are derived from Science class. Similarly, Dog, Cat, Horse are derived from Animal class.

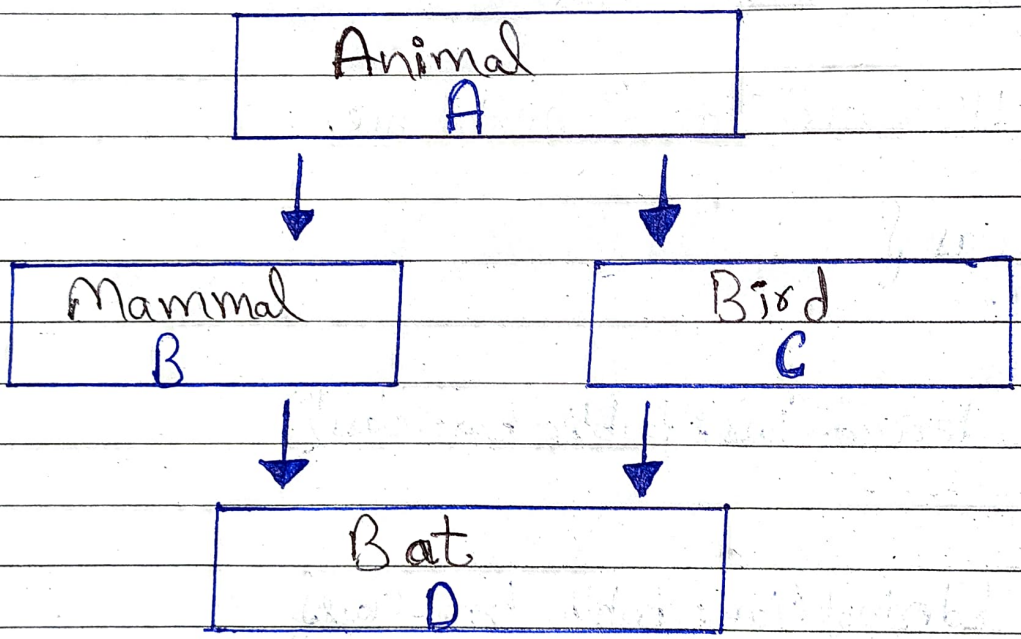
Syntax of Hierarchical Inheritance

```
Class base-class{
   ... .. ...
};
Class First_derived-Class: public base-class{
   ... .. ...
};
Class Second derived Class: public base-Class{
   ... .. ...
};
Class third-derived-Class: public base-class{
   ... .. ...
};
```



Hybrid INHERITANCE IN C++

- Hybrid inheritance is a Combination of multiple inheritance and multilevel inheritance.
- A Class is derived from two classes as in multiple inheritance.
- However, one of the parent classes is not a base class.



Friend function and friend Classes

FRIEND FUNCTIONS

A friend function can access the private and protected data of a class. We declare a friend function using the friend keyword inside the body of the class.

```
Class className {
  ---------
  Friend returntype functionName (arguments);
  ---------
};
```

Properties of friend functions

1. Not in the scope of class. (this function is not a part of class).
2. Since it is not in the scope of the class, it cannot be called from the object of that class. e.g. complex(2,3); is valid.
3. Can be invoked without the help of any object.
4. Usually contains the objects as arguments.
5. Can be declared inside public or private function of a class.
6. It cannot access the members directly by their names and need object_name.member_name to access any member. (dot)

Friend Class in C++

We can also use a friend class in C++ using the friend keyword. For example:

```
class classB;

Class classA{
    // ClassB is a friend class of classA
    friend class classB;
    ----------
};

Class classB{
    ... .. ...
};
```

When a class is declared a friend class, all the member functions of the friend class become friend functions.

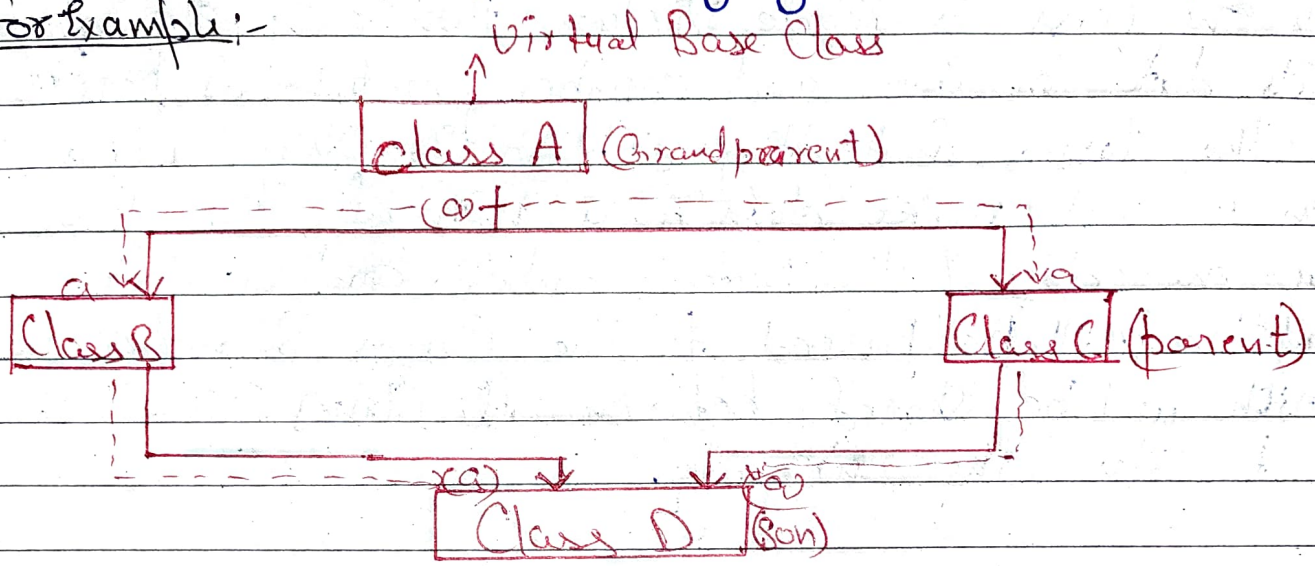
Since ClassB is a friend Class, we can access all members of classA from inside classB.

However, we cannot access members of ClassB from inside classA. It is because friend relation in C++ is only granted, not taken.

Virtual base class in C++

The Virtual base class is a concept used in multiple inheritances to prevent ambiguity between multiple instances.

For Example:-




```
Class B: virtual public A {
    ------
};

Class C: virtual A {
    ----------
};

Class D: public B, public C {
    ---------
};
```

As shown in figure:

1. Class "A" is a parent Class of two classes "B" and "C".
2. And both "B" and "C" ~~are both~~ Classes are the parent of Class "D".

The main thing to note here is that the Data members and member function of Class "A" will be inherited twice in Class "D" because class "B" and "C" are the parent classes of Class "D" and they both are being derived from class "A".

So when the Class "D" will try to access the Data member or member function of Class "A" it will cause ambiguity for the compiler will throw an error. To solve this

ambiguity we will make class "A" as a virtual base Class. To make a virtual base Class "virtual" keyword is used.

When one Class is made virtual then only one copy of its data member and member function is passed to the Classes inheriting it. So in our Example when we will make class "A" a virtual Class then only one copy of the Data member and member function will be passed to the Classes "B" and "C" which will be shared between all Classes. This will help to solve the ambiguity.

Constructors in derived Classes.

- We Can Use Constructors in derived Classes in C++.
- If base class Constructor does not have any arguments, there is no need of any Constructor in derived Class
- But if there are one or more arguments in the base Class Constructor, derived class need to pass arguments to the base class Constructor.
- If both base and derived classes have Constructors, base class Constructor is Executed first.

Constructor in multiple inheritence.

- In multiple Inheritance, base classes are Constructed in the order in which they appear in the Class declaration.
- In multilevel Inheritance, the Constructors are Executed in the order of Inheritance.

Special Syntax

- C++ Supports an special syntax for passing arguments to multiple base classes
- The Constructor of the derived class recieves all the arguments at once and then will pass the calls to the respective base classes.
- The body is called after all the Constructors are finished executing.

```
Derived-Constructor(arg1, arg2, arg3, ---): Base1-Constructor(arg3, arg2), Base2-Constructor(arg3, arg4) {
---
} Base1-Constructor(arg1, arg2).
```

Special Case of Virtual Base Class.

- The Constructors for virtual base classes are invoked before an nonvirtual base class.
- If there are multiple virtual base classes, they are invoked in the order declared.
- Any nonvirtual base class are then Constructed before the derived class Constructor is executed.

Initialization list in Constructor in C++

The Initialization list in Constructor is another Concept of Initializing the data members of the class. The Syntax of the initiazation list in Constructor is shown below

```
// Constructor(argument-list): initialization section {
        assignment to other Code;
};
ex:- Test(int i, int j): a(i), b(j) {}
```

Virtual functions.

A Virtual function is a member function in the base class that we expect to redefine in derived classes.

Basically, a virtual function is used in the base class in order to ensure that the function is overridden. This especially applies to cases where a pointer of base class points to an object of a derived class.

For example:-

```
Class Base{
  public:
    void print() {
        //code
    }
};
Class Derived : public Base{
  public:
    void print() {
        // Code
    }
};
```

Later if we create a pointer of base type to point to an object of derived class and call the print() function it call the print() function of the Base class.

In other words the member function of Base in not Overridden.

```
int main(){
    Derived derived1;
    Base *base1 = &derived1;

    //Calls function of Base class.
    base1 -> print();
    return 0;
}
```

In order to Avoid this, we declare the print() function of the Base class as virtual by using the virtual Keyword.

```
Class Base {
  public:
    virtual void print(){
        //Code
    }
};
```

Virtual functions are an integral part of polymorphism in C++.

Abstract Class and pure Virtual functions

C++ pure Virtual functions

Pure Virtual functions are used

- If a function doesn't have any use in the base class.
- But the function must be implemented by all its derived classes.

Let's take an Example:-
Suppose, we have derived Triangle, Square and Circle

classes from the Shape class, and we want to calculate the area of all these shapes.

In this case we can create a pure virtual function named CalculateArea() in the Shape. Since its a pure virtual function, all derived classes Triangle, Square and Circle must include the CalculateArea() function with implementation.

A pure virtual function doesn't have the function body and it must end with =0. for example,

```
Class Shape {
  public:
    // Creating a pure virtual function.
    virtual void CalculateArea() = 0;
};
```

Abstract Class

A Class that Contains a pure virtual function is known as an abstract class. In the above Example, the Class Shape is an abstract class.

We Cannot create objects of an abstract class. However, we can derive Classes from them, and use their data member and member functions (Except pure virtual functions)

C++ POLYMORPHISM

Polymorphism

Many Forms

- When one thing has many forms it is known as polymorphism.

- Polymorphism is an important concept of object-oriented programming. It simply means more than one form. That is, the same entity (function or operator) behaves differently in different scenarios. for Example:-

The + operator in C++ is used to perform two specific functions. when it is used with numbers (integers and floating point numbers), it performs addition.

```
int a=5
int b=6
int sum = a+b;  // sum=11.
```

And when we use the + Operator with strings, it performs string Concatenation. for example,

```
string firstname = "abc";
string lastname = "xyz";
//name = "abcxyz".
String name= firstname+lastname;
```

We Can implement polymorphism in C++ using the following ways.

1. Function overloading
2. Operator overloading
3. Function overriding
4. Virtual functions

POLYMORPHISM

- Compile time (Static) Polymorphism
  - Function overloading
  - Operator overloading
    - Same name
    - diff. Arg.
    - Same class
- Run time (Dynamic) Polymorphism
  - Function overriding
  - Virtual functions.
    - Same name
    - Arg same
    - Class diff.
    - Class inherit

C++ Function Overloading

In C++, we can use two functions having the same name if they have different parameters (either types or number of arguments).
And, depending upon the number / type of arguments, different functions are called.

It's a Compile time polymorphism because the Compiler knows which function to execute before the program is Compiled.

C++ Operator overloading

In C++ we can overload an operator as long as we are operating on user-defined types like objects and structures.

We cannot use operator overloading for basic types such as int, double, etc.

Operator overloading is basically function overloading where different operator functions have the same symbol but different operands.

And, depending on the operands, different operator functions are executed.

This is also a compile-time polymorphism.

C++ Function Overriding

In C++ inheritance, we can have the same function in the base class as well as its derived classes.

When we call the function using an object of the derived class, the function of the derived class is executed instead of the one in the base class.

So, different function are executed depending on the object calling the function.

This is known as function overriding in C++.

Its a runtime polymorphism because the function call is not resolved by the compiler, but it is resolved in the runtime instead.

C++ Virtual functions

In C++, we may not be able to override function if we use a pointer base class to point to an object of the derived class.

Using virtual functions in the base class ensures that the function can be overridden in these cases.

Thus, virtual functions actually fall under function overriding.

Virtual functions are runtime polymorphism.

Why Polymorphism

Polymorphism allows us to create consistent code. For example,

Suppose we need to calculate the area of a circle and a square. To do so, we can create a shape class and derive two classes circle and square from it.

In this case, it makes sense to create a function having the same name calculateArea() in both the derived classes rather than creating functions with different names, thus making our code more consistent.

TEMPLATES

Templates are powerful features of C++ which allows us to write generic programs.

There are two ways we can implement templates:

- Function Templates
- Class Templates

Function template

We can create a single function to work with different data types by using a template.

Defining a function Template

A function template starts with the keyword "template" followed by template parameter(s) inside <> which is followed by the function definition.

```
template <typename T>
T function name (T parameter1 , T parameter2 ...){
        //code
}
```

In the above code, T is a template argument that accepts different data types (int, float, etc), and typename is keyword.

When an argument of a data type is passed to functionName(), the compiler generates a new version of function Name() for the given data type.

Calling a function Template

Once we've declared and defined a function template, we can call it in other functions or templates (such as the main() function) with the following syntax.

functionName<dataType>(parameter1, parameter2, ...);

For Example, let us consider a template that adds two numbers:

```
Template <typename T>
T add (T num1, T num2) {
    return (num1 + num2);
}.
```

we can then call it in the main() function to add int and double numbers.

```
int main() {

    int result1;
    double result2;
    //Calling with int parameters.
    result1 = add<int>(2,3);
    Cout << result1 << endL;

    // Calling with double parameters
    result2 = add<double>(2.2, 3.3);
    Cout << result2 << endl;
    return 0;
}.
```

Output

```
5
5.5
```

Class Template

Similar to function templates, we can use class templates to create a single class to work with different data types.

Class template come in handy as they can make our code shorter and more manageable.

Class template Declaration

A class template starts with the keyword template followed by template parameter(s) inside <> which is followed by the class declaration.

```
template <class T>
Class className{
  private:
    T var;
    ... .. ...
  public:
    T functionName(T arg);
    ... .. ...
};
```

Creating a class template object

For Example:- ClassName <data type> class object;

ClassName<int> class object
ClassName<float> Class object
Class Name<String> Class object.

Defining a class Member Outside the Class Template

Suppose we need to define a function outside of the Class template. we can do this with the following code.

```
template<class T>
class ClassName {
    ... .. ...
    // function prototype.
    return type functionName();
};

// function definition
template <class T>
returntype ClassName<T> :: functionName(){
    //code.
}
```

Notice that the code template <class T> is repeated while defining the function outside of the Class, This is necessary and is part of the syntax.

we can define getNum() outside of Number with the following code:

```
template <Class T>
Class Number {
    ... .. ...
    // function prototype.
    T getnum();
};
// function definition.
template <class T>
T Number<T> :: getNum() { return num; }
```

C++ Class templates with Multiple Parameters.

In C++, we can use multiple template parameters and even use default arguments for to parameters. for Example,

```
template <class T, class U, class V = int>
class ClassName {
  private:
    T member1;
    U member2;
    V member3;
    ... .. ...
  public:
    ... ... ...
};
```

Notice the code class V = char. This means that V is a default parameter whose default type is Char.

In main(), we can objects of classTemplate with the code.

```
// create object with int, double and Char types
ClassTemplate <int, double> obj1(7, 7.7, 'c');

// create object with double, Char and bool types
ClassTemplate <double, char, bool> obj2(8, 8.8, false);
```

for obj1, T = int, U = double and V = char.
for obj2, T = double, U = Char and V = bool.

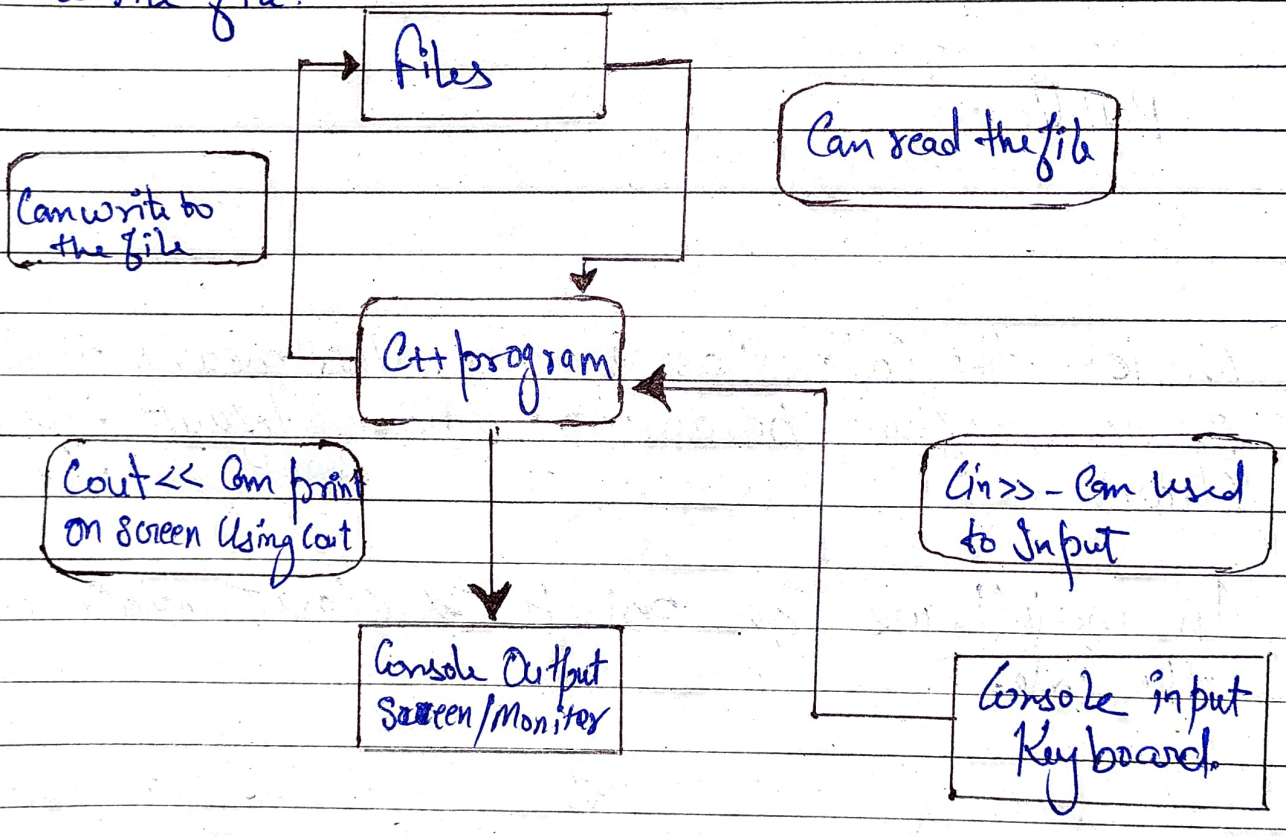
FILE HANDLING

File Introduction

A file is a named location that can be used to store information. for example, main.cpp is file where we write C++ code.

There are two main operations which can performed on files:-

- Read file
- Write file.

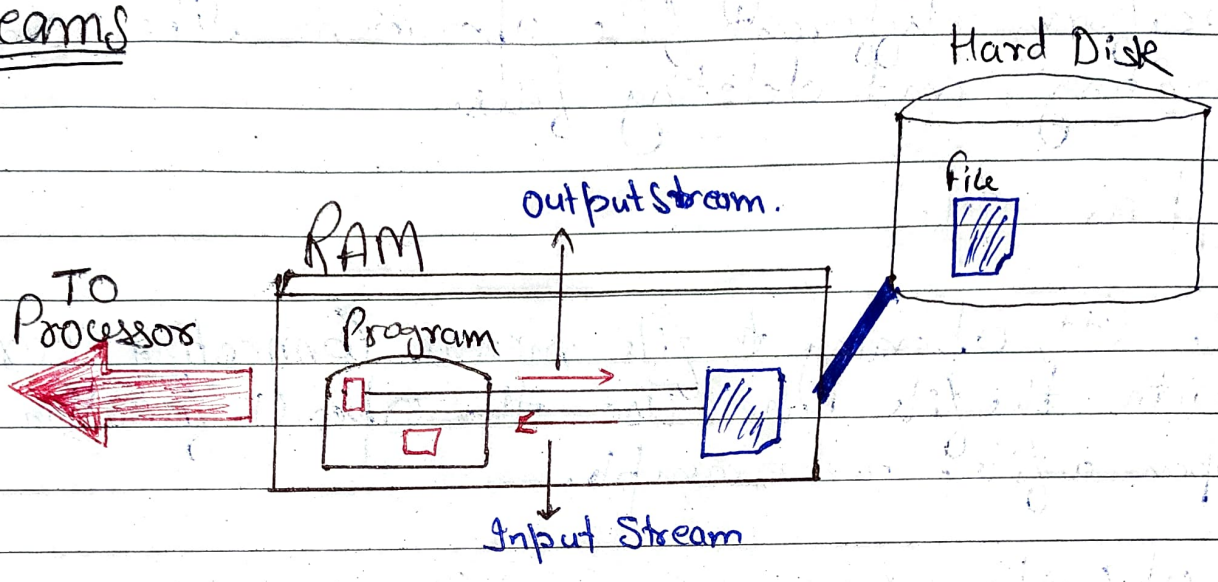


FILE Read and Write diagram.

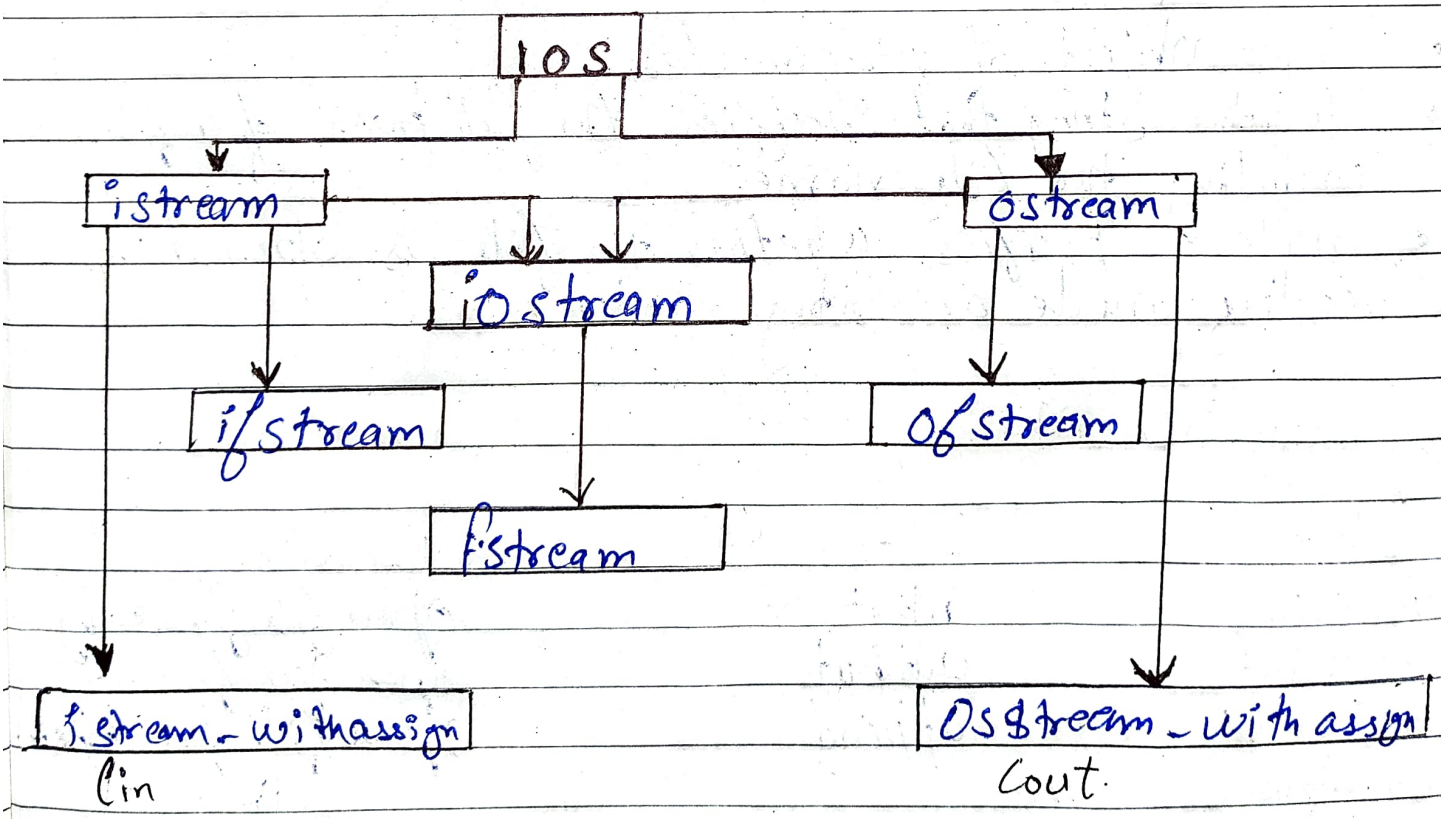
1) The User Can provide input to the C++ program by using Keyboard through Using "Cin>>" Keyword.
2) The User Can get output from the C++ program on the monitor through "Cout<<" Keyword.

3) The User Can write on the file.
4) The User Can read on the file.

Streams



Flow of Data in Files Diagram.



In C++, we Use the fstream Library to work with files. TO Use this Library we must include the fstream header file to our program.
#include <fstream>.

fstream library

The fstream library provides various classes to perform different file operations like reading, writing, and deleting files.

Open a file

In C++, opening a file means connecting our program with the file. We use the open() function to open a file. For example;

```
obj.open(file_path, mode);
```

Here,

- obj - object of fstream class.
- file path - String that specifies the full path of a file including the file name.
- mode - Specifies whether the file is opened in reading mode or writing mode.

File opening Modes

Mode	Meaning	Description
ios::in	input	Open file for reading
ios::out	output	Open file for writing
ios::app	append	Open file for appending.

Example:-

```
fstream my_file;
//open the file in writing mode
myfile.open("Sample.txt", ios::out);
//open file in reading mode
myfile.open("Sample.txt", ios::in);
```

The Open modes are members of the ios class. Hence, we need to use the scope resolution operator :: when specifying the modes.

Check the file status.

Once we've opened our file, it is good practice to check if the file has been opened properly. This will prevent errors from occuring in our program.

We use fail() function to check the file status.

for example:-

```
fstream my_file;
// Open file.
my_file.open("Sample.txt", ios::out);
// Check file status.
cout << my_file.fail();
```

Here, we are checking if the Sample.txt file is opened correctly. It returns.

- true - if the file has errors or has not been opened
- false - if the file opening is successful.

Closing a file.

After we perform operations on a file we need to close it. We use the close() function to close a file.

For Example :-

```
fstream my_file;
// open file
my_file.open("Sample.txt", ios::in);
// perform file operation.
// close the file
my_file.close();
```

There are some useful classes for working with files in c++.

- fstreambase
- istream --> derived from fstreambase
- ostream --> derived from fstreambase.

Primarily, there are 2 ways to open a file:

- Using the Constructor
- Using the member function open() of the class.

Example to demonstrate Read operation through Constructor. (Input file.)

```
string st;
// opening file Using Constructor and reading it.
ifstream in("this.txt"); // Read operation.
in >> st;
```

Example writing file operation output:

```
String st = "Harry bhai";
// Opening files using Constructor and writing it.
ofstream out("this.txt"); // Write Operations.
out << st;
```

Opening file through open function using ifstream or ofstream class is same as we use Ofstream class object just we donot add the mode in a function because it is defined in ifstream or ofstream.

FILE Operations

- Read a file
- Write a file
- Delete a file.

Read from a file

To perform read operation, we should open a file in read mode (ios::in). Then we can use the getline() function to read the file content.

Second step is to check if the connection is successful.

```
if (my_file.fail()) {
    cout << "Error! File not found!";
    exit(0);
}
```

Next step is to read the file.

```
while (!my_file.eof()) {
    getline(my_file, Content);
    Cout << Content << endl;
}
```

The getline() function reads the file and stores it in the content string. The while loop runs until the end of the file is encountered, which is given by eof() function.

Then we should close the file.

```
my_file.close();
```

its a good practice to close the file.

Write a file

To perform the write operation, we need to open a file in write mode (ios::out). Then, we can use the insertion operator << to write to a file. For example.

```
//write to the file
my_file << "This is a Sample file";
```

Things to remember:

- If the file doesn't exist, the program will create a file with the same name and write content to it.

- If the file exist, the program will replace the contents of the file with the new data.

Delete a file

In C++, we use the remove() function to delete a file. For example,

```
#include <iostream>
using namespace std;

int main() {
    char* file_path = (char*) "D:/Sample.txt";
    int check = remove(file_path);
    if (check == 0) {
        cout << "File deleted Successfully";
    }
    else {
    cout << "Couldn't delete file";
    }

    return 0;
}
```

Since remove() doesnot accept C++ string objects, we have stored the file path to the char pointer. Also we have converted the string literal "D:/Sample.txt" to char pointer using (char*).

The remove() function returns 0 if the file is deleted Successfully. otherwise, it returns -1.

Note: we can simply use a string literal as the file path for remove() function.

```
int check = Remove("D:/Sample.txt");
```

EXCEPTION HANDLING

C++ Errors

When writing a program, we can't avoid making errors.

Errors caused by not following the proper structure (Syntax) of the language is called Syntax error.
For example;

```
#include <iostream>
using namespace std;
int main(){
  cout<< "Test Code"
  return 0;
}
```

When we run this program, we will get an error message.

This error occured because of a missing semicolon. we can easily fix these errors by fixing the syntax.

C++ Exceptions

An exception is an unexpected event that causes the program to terminate abnormally. It occurs during the runtime (execution of the program)
For example;

- division by zero - Occurs if we try to divide a number by zero.
- bad - alloc exception - Occurs if we try to dynamically allocate an extremely large amount of memory to an array.

Example: Divide a Number by zero.

```
#include <iostream>
using namespace std;

int main() {
    cout << 5/0;
    return 0;
}

// Output: Floating point exception.
```

Here, we get floating point exception. This is because we are trying to divide a number by 0 which is impossible in C++.

Example: Bad Allocation Exception:

```
#include <iostream>
using namespace std;

int main() {
    int* ptr = new int [1000000000000];
    delete [] ptr;
    return 0;
}
```

In the above example, we are trying to allocate 1000000000000 memory block for the int array.

Here, the allocated memory exceeds the memory that the program can allocate.
Hence, we get the bad allocation exception.

C++ Exception Handling.

In C++ exceptions abnormally terminate the execution of a program.

This is why it is important to handle exceptions.

In C++, we use the try, throw, catch statements to handle exceptions.

- try: It includes the block of code that can cause exceptions.
- throw: It throws an exception when an error is encountered.
- catch: It catches the exception and executes a block of code when exception occurs.

Note:- The throw statements is not compulsory, especially if we use standard C++ exceptions.

Output 2

```
Enter first number: 72
Enter second number: 5
72/5 = 14.4
```

Example: C++ try.. throw... catch Block.

```
#include <iostream>
using namespace std;

int main() {
    double numerator, denominator, result;
    Cout<< "Enter first number: ";
    Cin >> numerator;
    Cout<< "Enter Second number: ";
    Cin >> denominator;

try {
    if (denominator == 0)
        Throw 0;
    // These statements are not executed if denominator
is 0.
    result = numerator / denominator;
    Cout<< numerator << "/" << denominator << "=" <<
    result << endl;

Catch (int num_exception) {
    cout << "Exception: Cannot divide by" << num_Except;
    }
return 0;
}
```

Output 1

```
Enter first number: 72
Enter Second number: 0
Error: Cannot divide by zero.
```

Explanation:
Here, if denominator is 0, the program will cause an Exception. to handle the exception, we have enclosed the division code inside the try block

Inside the try block, we throw an exception if denominator is 0;

```
if (denominator==0)
    throw 0;
```

Here, the code throw 0; throws an Integer value 0 to the Catch block.

The Catch block catches the exception, where the thrown value 0 is stored in the num_exception parameter.

Here, we have printed a message to make users aware about the exception that occurs if the denominator is 0.

Catch Any exception

Sometimes, we do not know what kind of exception we might encounter from piece of code.

Or, our code might cause multiple exceptions which typically requires multiple Catch statements. But we might only want to use a single Catch statement to handle all those exceptions.

In such cases we use the ellipsis symbol (...) as the catch parameter.

```
try {
    // Code that might cause exceptions
}
Catch (...) {
    // Code
}
```

this allows us to catch all types of exceptions.

Example: Bad Allocation Exception.

```
#include <iostream>
Using namespace std;

int main() {
    try {
        int *ptr = new
    int[10000000000000];
        delete [] ptr;
    }
    Catch (...) {
        Cout<<"Error!";
    }
    return 0;
}
```

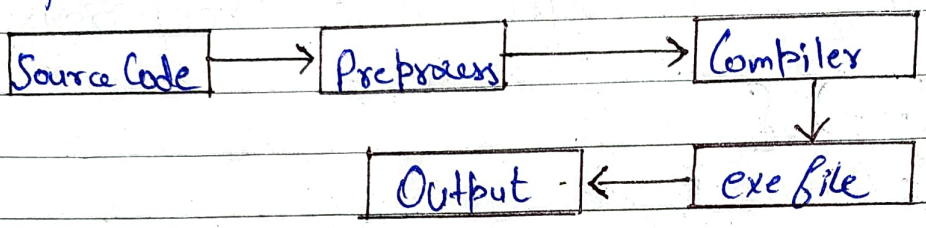
Output

Error!

Here, our code in the try block creates a bad Alloc exception. Suppose we don't know that it is a bad allocation exception.
So, we have caught his exception using ... as the catch parameters and printed an error message.

Preprocessors

Preprocessors are used to preprocess our program before it is passed to the Compiler.



So far we have been using

```
#include <iostream>
```

in our program. Here, #include is a preprocessor. It includes the header file iostream to our program before it is compiled.

In C++, all preprocessing directives begin with the # symbol.

#include preprocessor

The #include preprocessor is used to include header files to our program. For example,

```
#include <cmath>
```

Here, the cmath is a header file that is readily available when we install the C++ Compiler.

The #include preprocessor replaces the above line with the contents of the cmath header file.

Custom header files

We can also include custom header files in our program using the #include directive.

For Example;

```
#include "path-to-file/my-header.h"
```

Here, we have included the header file named my header.h in our program before our program is compiled.

This helps us to divide a larger program into multiple files.

#define preprocessor

We use the #define preprocessor to define variables and functions that can be used in our programs. For example,

```
#define PI 3.1415
```

Here, we have defined the PI variable. Now, when we use PI in our program, it is replaced by 3.1415.

Example:- Define Variables.

```
#include <iostream>
#define PI 3.1415
using namespace std;
int main() {
    double radius, area;
    cout<<"Enter the radius: ";
    cin >> radius;
```

// Notice the use of the PI
area=PI * radius * radius;
Cout << "Area = " << area;
return 0;
}

Output

Enter the radius: 4
Area = 50.264

In this case, the value 3.1415 is used in place of PI.

Undefine Preprocessor

The Undefine preprocessor is used to Undefine variables that were defined using the #define preprocessor. For Example:-

```
#include <iostream>
using namespace std;
# define PI 3.1415
# define Circle_area(r) (PI*r*r)
int main(){
Cout<< "Using PI (4 decimal places): "2<endl;
Cout << "Area =" << circle_area(5) << endl;
#Undef PI
# define PI 3.14
```

Cout << "Using PI (2 decimal places): " << endl;
Cout << "Area" << Circle_area(5) << endl;
return 0;
} → Here we have defined the variable PI and then redefined it with new value 3.14

Namespaces

Naming Conflicts in C++

Consider the following code:

```
int main() {
  int number;
  //error: Conflicting declaration
  double number;
}
```

This code cannot be compiled successfully because we have declared two variables of same name number with in the scope of the main function. In C++, this is known as a naming conflict

We can resolve naming conflicts inside the same scope through the use of namespaces. This becomes especially important in large projects, where naming conflicts are more likely to occur.

Creating a Namespace:

In C++, a Namespace allow to store variables, fuctns, classes and other programming features inside a named scope to prevent naming conflicts.

We use the namespace keyword to create a named scope. For Example,

```
namespace double_data {
  double numbers;
}
```

Here, we have defined the number variable inside the double_data scope. Now, if we use the same variable name in global scope, there won't be any naming conflict.

Multiple namespaces

We can also create and use multiple namespaces in a single program. For example,

```
#include <iostream>
using namespace std;

namespace double_data {
    double number;
}
namespace float_data {
    float number;
}

int main() {
    int number = 25;
    cout << "Integer Data: " << number << endl;
// access members of double_data namespace.
    double_data :: number = 39.45;
    cout << "Double Data: " << double_data::number << endl;

// access members of float_data namespace.
    float_data :: number = 24.8f;
    cout << "Float Data: " << float_data::number;
    return 0;
}
```

Output

Integer Data: 25
Double Data: 39.45
Float Data: 248

Here, these namespaces allows us to create 3 variables with the same name number.

Note that we have used the name of the namespaces and :: operator to access data from the namespaces.

The Using directive

The Using directive allows us to directly access members of namespaces without using the :: operator.

For Example:

```
namespace one{
    void Display(){
        Cout<<"namespace one"<<endl;
    }
}
namespace two{
    void Display(){
        Cout<<"namespace two"<<endl;
    }
}
int main(){
    using namespace one;
    // Call one::display()
    display();
    two::display();
    return 0;
}
```

Output

Namespace One
Namespace two.

Here, the Using directive allows us to directly Use the display() function inside the main() function.

Std Namespace

We have Used the Code.
Using Namespace Std;

in most of Our programs. In C++, Std is a predefined namespace that provides different entities Such as Cout, Cin, endl, etc. For example,

```
//Using entities of std namespace.
Std:: Cout << "Std is a namespace" <<
Std :: endl;
```

To make Our Cleaner and more readable, we have removed the std:: part with the Code Using namespace std;

However, it is preferable to use :: with namespace because Sometimes the Using directive Can Create naming Conflicts if we include two or more namespaces with the same identifiers.