- C++ began as an expanded version of C. The C++ extensions were first invented by "Bjarne Stroustrup" in 1979 in Bell laboratories in Murray Hills, New Jersey. In 1983, the name was changed to C++. It supports both procedural and OOPs.
- All the programs written in C can be compiled by C++ compiler and run but converse is not true.

✱ Structure of C++ program ⇒

1) Documentation (comment section): It consists of comment lines which indicate program title, author name and other details of program.
   e.g. // program to find area of circle.
2) Include files (header file section): They are to specify the files, the contents of which are to be included as part of the program. C++ compilers are provided with "standard library" which consists of definitions of a large number of commonly used in-built functions. These files are called as "header files" since they are included in the beginning of a program.
   e.g. # include <iostream>
3) Class declaration and defining section: They are to declare and define the classes that are used by the program.
4) Macros definition: It is to assign symbolic names to constants and define macros.
   e.g. # define max 50.
   The "preprossesor directive" i.e. # define is used for the purpose.
5) Global variables declaration: These are those which are required to be accessed by all the functions defined after their definitions.

✯ Simple C++ program ⇒

①
```
#include <iostream.h>
int main()
{
int i;
cout << " this is output\n";
cout << " Enter a number";
cin >> i;
cout << " square is = " << i*i << "\n";
return 0;
}
```

②
```
#include <iostream>
int main()
{
std :: cout << " Hello world";
return 0;
}
```

③
```
#include <iostream>
using namespace std;
```

→ The first character is the # symbol which is the signal to a program called "pre processor".

→ "using namespace std" tells the compiler to use the std namespace. namespaces are a recent addition to C++.

→ A namespace creates a declaration region in which various program elements can be placed.

→ The purpose of a namespace is to localise the names of identifiers to avoid name collisions.

→ Traditionally, the names of library functions and other such items were simply placed into the global name space.

→ The "using" statement informs the compiler that we want to use the std namespace, cout, cin etc are all present in the std namespace.

→ The "std" namespace is the area in which the entire C++ standard library is declared.

- The "cin" is the input statement. The statement is used to accept the value of variable.

  Syntax → cin >> variable ;

  The symbol ">>" is called insertion symbol.

- The "cout" is the output statement.

  Syntax → cout << variable;

  The symbol "<<" is called extraction operator and displays the value of variable.

Ques) # Program for finding the size of datatypes.

```
#include <iostream>
using namespace std;
int main()
{
cout << "size of int is " << sizeof(int);
cout << "size of char is " << size of (char);
cout << "size of double is " << size of (double);
return 0;
}
```

VARIABLE

- A variable is a temporary container to store information. It is a named location in container memory where various data like numbers and characters can be stored and manipulated during the execution of the program.

- When using a variable you refer to memory address of computer.
- C requires all the variables to be defined in the beginning of the scope. C++ allows the declaration of a variable anywhere in the scope. This means that a variable can be declared right at the place of its first use.
- Basically, there are 2 types of variables –
  1) local variables : within the scope.
  2) Global variables : for the entire program.

IDENTIFIERS

- Identifiers refers to the name of variables, functions, arrays, classes created by the user. They are the fundamental requirements of any programming language.

CONSTANTS

- The name constants are given to such variables or values which cannot be modified once they are defined.
- They are two simple ways to define a constant :

1) Using #define preprocessor.

```
#include <iostream>
using namespace std;
#define length 10
#define width 15
#define newline '\n'
int main ()
{ int area;
area = length * width;
cout << area;
cout << newline;
return 0; }
```

2) Using Keyword "const"

```
# include <iostream>
using namespace std;
int main ()
{
const int length = 10;
const int width = 15;
const char newline = '\n';
int area;
area = length * width;
cout << area;
cout << newline;
return 0;
}
```

→ Declaring a variable reserves memory. You can use the address operator to learn the address of this reserved memory.

Syntax → & [variable name];

Ques) Find the memory address for test score on your computer.

→
```
#include <iostream>
using namespace std;
int main ()
{
int a;
cout << " address of variable is " << &a;
return 0;
}
```

CONDITIONAL STATEMENTS

- Conditional statements, also known as selection statements, are used to make decisions based on a given condition. If the condition evaluates to True, a set of statements is executed otherwise another set of statements is executed.

→ **if statement**

```
if (test expression)
{
statements;
}
```

→ **if else statement**

```
if (test expression)
{
statements;
}
else
{
statements;
}
```

→ **nested if-else statements**

```
if (condition 1)
{
if (condition 2)
{
Statements;
}
else
{
Statements;
}
}
else
{
Statements;
}
```

→ **if - else - if ladder**

```
if (condition 1)
{
statements;
}
else if (condition 2)
{
statements;
}
else
{
statements;
}
```

→ **switch statement**

```
switch (expression)
{
case V1:
statement 1;
break;
case V2:
statement 2;
break;
default:
statements;
break;
}
```

= The keyword default specifies the set of statements to be executed in case no match is found

★ **Note** → There can be multiple case labels but there can be only one default label.

→ Fall through is a situation the causes execution of the remaining cases even after a match has been found. In order to prevent this, break statements are used at the end of each case.

OPERATORS

1) Arithematic operators : (+ , - , * , / , %) these are binary operators , as they work on two operands. (++ , --) these are unary operators as they work on one operand.

2) Assignment operators : These are used to assign value to a variable . " = " is a assignment operator. More operators are (+= , -= , *= , /=)

3) Relational operators : These are used for the comparison of the values of two operands (== , != , > , < , >= , <=) are relational operators.

4) Logical operators : Logical operators are used to combine two or more conditions or to complement the evaluation of the original condition in consideration.
(&& , || , !) are logical operators.

5) Bitwise operators : & (bitwise AND)
| (bitwise OR)
^ (bitwise XOR)
<< (left shift)
>> (right shift)
~ (bitwise NOT)

6) Ternary operator : (? :)

7) Scope (::) operator

8) new and delete operators

9) Address of operator (&)

10) sizeof ()

OPERATOR PRECEDENCE

Precedence	Operator	Description	Associativity
1	()	Parentheses (function call)	left-to-right
	[]	Brackets (array subscript)	
	.	Member selection via object name	
	->	Member selection via a pointer	
	++/–	Postfix increment/decrement	
2	++/–	Prefix increment/decrement	right-to-left
	+/-	Unary plus/minus	
	!~	Logical negation/bitwise complement	
	(type)	Cast (convert value to temporary value of type)	
	*	Dereference	
	&	Address (of operand)	
	sizeof	Determine size in bytes on this implementation	
3	*,/,%	Multiplication/division/modulus	left-to-right
4	+/-	Addition/subtraction	left-to-right
5	<< , >>	Bitwise shift left, Bitwise shift right	left-to-right
6	< , <=	Relational less than/less than or equal to	left-to-right
	> , >=	Relational greater than/greater than or equal to	left-to-right
7	== , !=	Relational is equal to/is not equal to	left-to-right
8	&	Bitwise AND	left-to-right
9	^	Bitwise exclusive OR	left-to-right
10	|	Bitwise inclusive OR	left-to-right
11	&&	Logical AND	left-to-right
12	||	Logical OR	left-to-right
13	?:	Ternary conditional	right-to-left
14	=	Assignment	right-to-left

	+=, -=	Addition/subtraction assignment	
	*=, /=	Multiplication/division assignment	
	%=, &=	Modulus/bitwise AND assignment	
	^=, |=	Bitwise exclusive/inclusive OR assignment	
	<>=	Bitwise shift left/right assignment	
15	,	expression separator	left-to-right

CONTROL STATEMENTS

C++ provides 3 looping structures

1) for loop -

```
for (initialization ; condition ; increment)
 {
  Statements ;
 }
```

2) while loop -

```
while (test_expression)
 {
  Statements;
  }
```

3) do-while loop -

```
do
{
 Statements;
 }
while (test_expression);
```

Ques) WAP for reading any 2 nos. from Keyboard and display the largest value of them.

→
```
#include <iostream>
using namespace std;
int main ()
{
float x, y;
cout << "Enter any 2 numbers : ";
cin >> x >> y;
if (x > y)
cout << "Largest value is =" << x << endl;
else
cout << "Largest value is =" << y << endl;
return 0;
}
```

Ques) WAP for finding largest values of any 4 numbers.

→
```
#include <iostream>
using namespace std;
int main ()
{
float x, y, a, b;
cout << "Enter any 4 numbers =" << endl;
cin >> x >> y >> a >> b;
if (x > y && x > a && x > b)
{ cout << "Largest number" << x << endl;
else if ( y > a && y > b)
cout << "Largest number :" << y << endl;
else if (a > b)
cout << "Largest number :" << a << endl;
else
cout << "Largest number :" << b << endl;
return 0;
}
```

TYPE CASTING

- C++ is rich in datatypes. It allows an expression to have data ~~types~~ items of different types.
- Conversion takes place at two instances :-

1) At the time of evaluation of an expression :

→ Wherever an expression has 2 data items which are of diff. data types. lower type gets converted into higher type. The result of expression is will be in higher data type.

2) At the time of assignment of a value of an expression :

OR

Source variable to the target variable :

→ The value of the expression on right hand side of an assignment statement gets converted into the type of the variable collecting it.

- Type casting refers to the conversion of data from one basic type to another by applying external use of data type Keywords.
- It is essential when the values of variables are to be converted from one type to another.

Syntax → data type (expression)

or

(data type) expression

e.g. float x;

x = (float) 5/2;

or

float x;

x = 5 | float (2);

Ques) WAP to convert a float datatype into integer.

→
```
#include <iostream>
using namespace std;
int main()
{
int a, b, c;
float x, y;
a = 19;
b = 10;
c = 3;
x = float (b)/c;
cout << " x =" << x;
y = (float) a/c;
cout << "\ny =" << y;
return 0;
}
```

Ques) Type casting example.

→
```
#include <iostream>
using namespace std;
int main()
{
int a;
double b = 2.55;
a = b;
cout << a << endl;
a = (int)b;
cout << a << endl;
a = int(b);
cout << a << endl;
return 0;
}
```

• Enumeration :-

An enumerated datatype is another user defined datatype which provides a way for attaching names to numbers, thereby increasing comprehensive ability of the code.

→ The enum Keyword automatically enumerates a list of words by assigning them values 0, 1, 2, ... This facility provides an alternative means for creating symbolic constants.

→ The enumerated type also known as "Enumeration" is a datatype where every possible value is defined as a symbolic constant (enumerator)

Syntax → enum enumtype {List of names} var-list ;

e.g. → enum color {red, green, blue}c ;

c = blue ;

Ex. 1) Program on enum (switch case).

→
```
#include <iostream>
using namespace std;
int main()
{
enum Fruits {orange, guava, apple};
Fruits myfruit;
int i;
cout << "Enter your choice :(0-2)";
cin >> i;
switch (i)
{
case orange: cout << "your fruit is orange";
             break;
case guava: cout << "your fruit is guava";
            break;
```

```
case apple : "Your fruit is apple";
             break;
}
return 0;
}
```

Ex. 2) Program on enum.

→
```
#include <iostream>
using namespace std;
int main()
{
enum week { Sunday, Monday, Tuesday, Wednesday,
            Thursday, Friday, Saturday };
week today;
today = wednesday;
cout << "Day is" << today + 1;
return 0;
}
```

Ex. 3) Program on enum. (using sizeof()).

→
```
#include <iostream>
using namespace std;
enum suit { club = 0, diamonds = 10, hearts = 20,
            spades = 3 } card;
int main()
{
card = club;
cout << "size of enum variable " << sizeof(card)
                                  << " bytes";
return 0;
}
```

FUNCTIONS

- A function is a block of code that perform some operations. In programming, "function" refers to a segment that groups code to perform a specific task.
- There are 2 types of functions :-

1) Library functions → These are built-in functions in C++.

e.g. :
```
#include <cmath.h>
#include<iostream>
using namespace std;
int main()
{
double number, squareroot;
cout << "Enter a number";
cin >> number;
squareroot = sqrt(number);
cout << "square root of" << number << " = " <<
        squareroot;
return 0;
}
```

2) User - defined functions → C++ allows programmers to define their own function.

→ A user defined function groups code to perform a specific task and that group of code is given a name.

→ When the function is invoked from any part of program, it executes all the code defined in the body of function.

→ The function itself is referred as "function definition"

→ When the function is called, control is transferred to the first statement to the function body.

e.g.) // Function definition

```
int add (int a , int b)
{
 int add1;
 add1 = a + b;
 return add1;
 }
```

★ FUNCTION PROTOTYPE ⇒

→ It is one of the major improvement added to C++ functions. The prototype describes the function interface through the compiler by giving details such as the numbers and the type of arguments of the type of return value. It's syntax is :-

type function_name (argument list);

Ques) Sum of 2 numbers using functions :-

```
#include <iostream>
using namespace std;
int sum (int x , int y); ——→ [Formal Parameters]
int main ()
{
int a = 10 , b = 20;
int c = sum (a, b); ——→ [Actual Parameters]
cout << c;
}
int sum (int x, int y)
{
return (x + y);
}
```

☆ Call by value & Call by reference ⇒

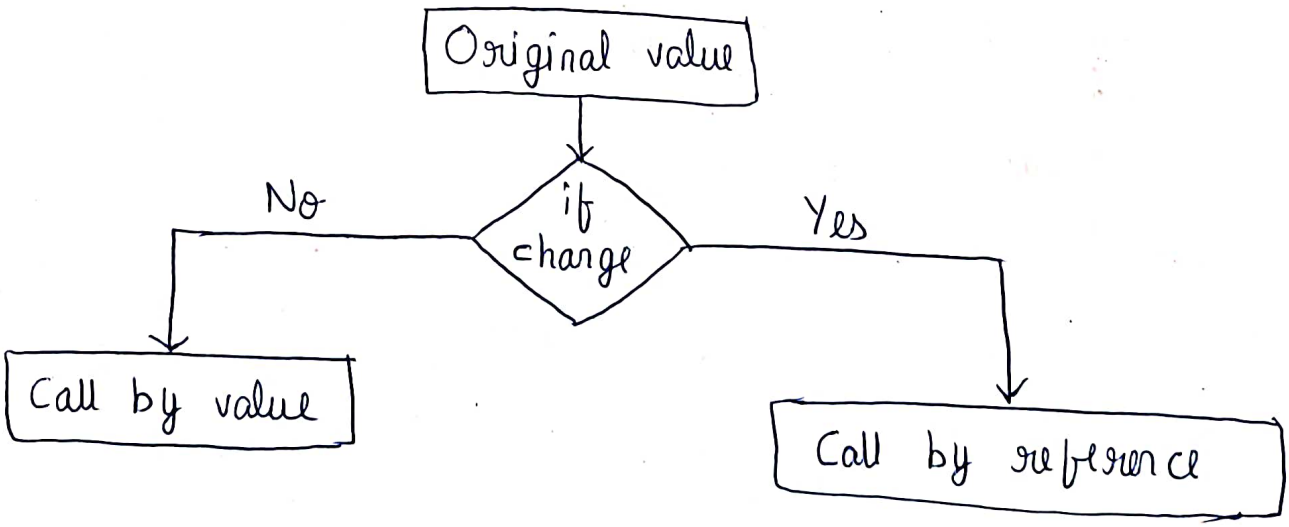
On the basis of arguments, there are two types of functions available in C++. They are,

1) with arguments
2) without arguments

with arguments	without arguments
• Declared and defined with parameter list.	• No parameter list.
• Values for parameter passed during function call.	• No value passed during function call.
• int sum (int x, int y); sum (10, 20);	• int show (); show ();

- If a function take any argument, it must declare variables that accept the value as an argument. These variables are called "Formal parameters" of the function.
- There are 2 ways to pass value of data to function in C++.

1) Call by value
2) Call by reference.



Call by Value :-

1) This method of passing arguments to functions copies the actual value of an argument into the formal parameter of the function.
2) In this method, original value cannot be changed (or) modified. When you pass values to the functions, it is locally stored by the function parameter in stack memory location.

Ex →

```
#include <iostream>
#include <conio.h>
using namespace std;
void swap(int a, int b)
{
int tmp; tmp = a;
a = b;
b = tmp;
cout << "Value of a" << a;
cout << "Value of b" << b;
}
void main()
{
int a = 100, b = 200;
swap(a, b);
}
```

Call by reference :-

1) This method of passing arguments to a function copies the reference of an argument into the formal argument.
2) In this method, original value is changed (or) modified bcoz we pass reference. Here, address of the value is passed in the function, so actual and formal arguments shares the same address space.

3) Any value changed inside the function, is reflected inside and outside the function.

Ex.

```
#include <iostream>
using namespace std;
#include <conio.h>
void swap (int *a, int *b)
{
int tmp;
tmp = *a;
*a = *b;
*b = tmp;
}
void main()
{
int a=100, b=200;
swap (&a, &b)
cout <<"Value of a" << a;
cout << " Value of b"<<b;
getch()
}
```

Basics of Classes & Objects :-

- Class structure :-

```
class Test ——————> starts with capital letter
{
private:
        int data1;
        int data2;
public:
void function1()
{ data1 = 2;
}
```

```
Float function2()
{
data2 = 3-5;
return data2;
}
};
```

→ A class is a blueprint for the object

→ A class is a user defined datatype which holds its own data members and member functions which can be accessed and used by creating instance of that class.

→ When class is defined, only the specification for that object is defined, no memory (or) storage is allocated.

→ A class is defined in C++ using Keyword "class" followed by the name of class.

Syntax :-

```
class classname
{
// some data
// some functions
};
```

e.g -

```
class Box
{
public:
    double length;
    double breadth;
    double height;
};
```

→ A class name must start with an Uppercase letter. But it isn't mandatory.

→ Classes contains data members and member functions and the access of these data members and variables depends on the access specifiers.

e.g.

```
#include <iostream>
using namespace std;
class Test
{
private:
        int data1;
        float data2;
public:
      void insertIntegerData (int d)
      {
      data1 = d;
      cout << "Number :" << data1;
      }
      float insertFloatData ()
      {
      cout << "\n Enter data";
      cin >> data2;
      return data2;
      }
      };
int main ()
{
Test O1, O2;
O1. insertIntegerData (12);
return 0;
}
```

- Defining a member function of a class outside its scope :-

→ A member function of a class is defined using the :: (double colon) and is called scope resolution operator.

Syntax :- return_type classname :: member function (arguments)
{

}

- Accessing data member (or) member function of a class:-

→ A data member or member function of a class is accessed using the "." operator. It is also called as period operator.

→ Syntax - class_object . data member / member function

Ques) WAP to assign data to the data members of the class such as day, month, year and display contents of class.

→
```
#include <iostream>
using namespace std;
void main()
{
class date
{
public:
        int day;
        int month;
        int year;
};
class date today;
today.day = 10;
today.month = 3;
today.year = 1977;
cout << "Todays day is" << today.day;
cout << "Todays month is" << today.month;
cout << "Todays year is" << today.year;
}
```

Ques) WAP to define both data members and member function of a class within the scope of class definition.

→
```
#include <iostream>
using namespace std;
class date
{ private :
        int day;
        int month;
        int year;
 public :
 void getdata (int d, int m, int y)
 {
  day = d;
  month = m;
  year = y;
 }
 void display ()
 {
 cout << " Today date is = " << day << " / " << month << " / " <<
          year << endl;

 } };
 void main ()
 {
 class date today;
 int d1 = 10;
 int m1 = 10;
 int y1 = 2003;
 today . getdata (d1, m1, y1);
 today . display ();
 }
```

not mandatory

Access Specifiers

→ Access specifiers in C++ defines how the members of the class can be accessed.

→ There are 3 access specifiers :-

1) public
2) private
3) protected

→ ✯✯ By default, all members and functions of a class is private, i.e., if no access specifier is specified.

→ Syntax for declaring Access specifier :

```
class classname
{
 private :
        // private members & functions
 public :
        // public members & functions
 protected :
        // protected members & functions
};
```

- public :- public class members are accessible outside the class and is availabel for everyone.
- private :- private class members are accessible within the class and it is not accessible outside the class. If someone try to access outside the class, it gives compile time error.
- protected :- protected access specifier is similar to private access specifier. It makes class members unaccessible outside the class. Byut they can be accessed by any subclass of that class.

	Our class	(sub class) Derived class	Outside the class
public	✓	✓	✓
private	✓	X	X
protected	✓	✓	X

Ques) private and public specifier program.

→
```
#include <iostream>
#include <conio.h>
class A
{
private :
        int a;
public :
        int b;
public :
        void show ()
        {
        a = 10 ; b = 20;
        cout << " Accessing variables within class " ;
       cout << " Value of a = " << a << endl;
       cout << " Value of b = " << b << endl;
       }
};
void main ()
{
A obj;
obj. show ();
getch ();
}
```

Constructor

→ It is a special member function whose task is to initialise the objects of its class.

→ It is special because its name is same as the class name and declared only in public.

→ The constructor is invoked whenever an object of its associated class is created.

→ Constructors are of 3 types :-

1) Default constructor : It is the constructor which does not take any argument, It has no parameter.

e.g.
```
#include <iostream>
using namespace std;
class Sample
{
public :
        int x, y;
Sample ()
{
x = y = 0;
}
};
int main ()
{
Sample A;
cout << " Default constructor values x, y << A.x
        << " , " << A.y << endl;
return 0;
}
```
Default Constructor.

2) Parameterised constructor : The constructor that can take arguments are called parameterised constructor.

e.g.
```
#include <iostream>
using namespace std;
class Cuble
{
public :
        int side;
        Cuble (int x)
        {
        side = x;
        }
};
int main ()
{
Cuble C1 (10);
Cuble C2 (20);
Cuble C3 (30);
cout << C1.side ;
cout << C2.side;
cout << C3.side;
return 0;
}
```

Parameterised constructor

3) copy constructor : One object member variable values assigned to another object member variables is called copy constructor.

→ It is used in foll situations :-

1) The initialisation of an object by another object of the same class.

2) Stating the object as by value parameter of a function.

e.g.→

```
#include <iostream>
#include <conio.h>
using namespace std;
class Example
{
int a, b;
public :
        Example (int x, int y)
        {
        a = x;
        b = y;
        cout << " I am in constructor " ;
        }
       void Display ()
       {
      cout << " Variables : " << a << " \t " << b ;
       }
};
int main ()
{
Example obj (10, 20);
Example obj2 = obj; ] Copy constructor
obj. Display ();
obj2. Display ();
getch ();
return 0;
}
```

Destructor

→ A destructor, as the name implies is used to destroy the objects that have been created by the constructor.

→ Like a constructor, destructor is a member function, whose name is same as the class name, but is preceded by a tild (~).

→ e.g. → ~Integer();

→ A destructor never takes any argument nor does it return any value.

→ e.g.

```
#include <iostream>
using namespace std;
int count = 0;
class Alpha {
 public:
        Alpha()
        {
        count++;
        cout << "No. of objects created " << count;
        }
        ~Alpha()
        {
        cout << "No. of objects destroyed " << count;
        count--;
        }
};
int main()
{
cout << "Enter main";
Alpha A1, A2, A3, A4;
return 0;
}
```

Destructor

Inline Functions

→ C++ provides an Inline function to reduce the function call overload.

→ It is a function that is expanded in line when it is called.

→ When the inline function is called, whole code of the inline function gets inserted (or) substituted at the point of inline function. This substitution is performed by the compiler at compile time.

→ Inline function may increase efficiency if it is small.

→ Syntax :-
```
inline return_type function_name (parameter)
{
// function code
}
```

→ Compiler may not inform inline in some situations :-

① If a function contains a loop.

② If a function contains static variables.

③ If a function is recursive.

④ If a function contains switch or goto statements.

e.g.
```
#include <iostream>
using namespace std;
inline float mul (float x, float y)
{
return (x*y);
}
inline double div (double p, double q)
{
return (p/q);
}
int main()
{ float a = 12.3, b = 9.8;
cout << mul(a,b) << "\n" << div(a,b); return 0; }
```

Friend function

→ If a function is defined as friend function than the private and protected data of a class can be accessed using the functions.

→ A compiler knows a given function is a friend function by the use of the keyword friend.

→ It can be declared either in public (or) private part of a class without effecting its meaning.

→ It has arguments as objects.

→ These are often used in operator overloading.

→ Syntax:

```
class classname
{
 friend return_type function_name (arguments);
};
return_type function_name (arguments)
{
 - - -
 - - -
}
```

→ A friend function of a class is defined outside that class's scope but it has the right to access all private and protected members of the class.

→ A "friend" can be a function (or) function template (or) class (or) member function (or) class template, in which case, the entire class and all of its members are friends.

→ Example :-

```
#include <iostream>
using namespace std;
class Sample
{
int a,b;
public:
      void setvalue()
      { a=25; b=40; }
```

```
friend float mean (Sample s);
};
float mean (Sample s)
{
return float (s.a + s.b)/2.0;
}
int main ()
{
Sample x;
x. setvalue ()
{ cout << " Mean value = " << mean (x);
return 0;
}
```

Static data members & Member Functions

★ **Data members (static) :-**

→ When a data member is declared as static, only one copy of data is maintained for all objects of class.

→ We define class members as static using static keyword. A static member is shared by all objects of the class.

→ Static members are only declared in class declaration, not defined. They must be explicitly defined outside the class using scope resolution operator (::).

→ Example :

```
#include < iostream >
using namespace std;
class item
{
static int count;
static int number;
public:
      void getdata (int a)
      { number = a ; count ++ ;
       }
```

```
    void getcount ()
    {
    cout << "Count =" << count ;
    }
};
int item :: count;
int main ()
{
item a, b, c;
a.. getcount ();
b . getcount ();
c. getcount ();
a. getdata (100);
b.getdata (200);
c. getdata (300);
cout << " After reading data"<< endl;
a. getcount ();
b. getcount ();
c. getcount ();
return 0;
}
```

★ Static member functions :-

→ A static member function can be called even if no object of class exist and the static function are accessed using only the class name and the scope resolution operator.

→ A member function declared as static has following properties :-

1) A static function can have access to only other static members declared in same class.

2) A static member function can be called using the class name as :-

class_name :: function_name ;

→ Example :

```
#include <iostream>
using namespace std;
class test
{
int code;
static int count;
public:
      void getcode()
      {
      code = ++ count;
      }
      void showcode()
      {
      cout << "object number:" << code;
      }
static void showcount()
{
cout << "Count" << count;
}
};
int test :: count;
int main()
{
test t1, t2;
t1.getcode();
t2.getcode();
test :: showcount();
test t3;
t3.getcode();
test :: showcount();
t1.getcode();
t2.showcode();
return 0;
}
```

OOPs

→ OOPs stands for Object Oriented Programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

Class

→ A class is a logical entity used to define a new data type. A class is a user defined type that describes what a particular kind of object will look like. A class contains data members, methods and constructors.

→ Class determines how an object will behave and what the object will contain. The data in a class is called member, while functions in the class are called methods.

→ Syntax:

```
class class_name
{
//class body
//data members
//methods
};
```

→ Example of smartphone class:

```
class smartphone
{
string model;
int year_of_manufacture;
bool 5g_support;
//constructor
smartphone(string mod, int manu, bool 5g_supp)
{
```

```
model = mod;
year_of_manufacture = manu;
5g_support = 5g_supp;
}
void print()
{
cout << "Model =" << model << endl;
cout << "Year of Manufacture =" << year_of_manufacture << endl
cout << "5g Supported =" << 5g_support << endl;
}
};
```

Object

→ An object is an instance of a Class. It is an identifiable entity with some characterstics and behavior. To access the members defined inside the class, we need to create the object of the class. Objects are the basic units of OOPs.

→ Syntax to create an object:

```
class_name objectName;
```

→ Syntax to create an object dynamically:

```
class_name * ObjectName = new class_name();
```

→ The class's default constructor is called, and it dynamically allocates memory for one object of the class. The address of the memory allocated is assigned to the pointer. i.e., objectName.

```
#include <iostream>
using namespace std;
class smartphone{
string model;
int year_of_manufacture;
bool _5g_supported;
smartphone(string model_string, int manufacture, bool _5g_){
model = model_string;
year_of_manufacture = manufacture;
_5g_supported = _5g_;
}
void print_details(){
cout << "Model : " << model << endl;
cout << "Year of Manufacture : " << year_of_manufacture << endl;
cout << "5g Supported : " << _5g_supported << endl;
}
};
int main(){
smartphone iphone("iphone 11", 2019, false );
smartphone redmi("redmi note 11 t", 2021, true );
smartphone oneplus("oneplus nord", 2020, true );
int iphone_manufacture_date = iphone.year_of_manufacture;
bool redmi_support_5g = redmi._5g_supported;
string oneplus_model = oneplus.model;
iphone.print_details();
redmi.print_details();
oneplus.print_details();
}
```

Interview Questions

1. **Why do we use OOPs?**
   - ❖ It gives clarity in programming and allows simplicity in solving complex problems.
   - ❖ Data and code are bound together by encapsulation.
   - ❖ Code can be reused, and it reduces redundancy.
   - ❖ It also helps to hide unnecessary details with the help of Data Abstraction.
   - ❖ Problems can be divided into subparts.
   - ❖ It increases the readability, understandability, and maintainability of the code.
2. **What are the differences between the constructor and the method?**

Constructor	Method
It is a block of code that initializes a newly created object.	It is a group of statements that can be called at any point in the program using its name to perform a specific task.
It has the same name as the class name.	It should have a different name than the class name.
It has no return type.	It needs a valid return type if it returns a value; otherwise void.
It is called implicitly at the time of object creation	It is called explicitly by the programmer by making a method call
If a constructor is not present, a default constructor is provided by Java	In the case of a method, no default method is provided.

3. **What are the main features of OOPs?**
   - Inheritance
   - Encapsulation
   - Polymorphism
   - Data Abstraction

4. **The disadvantage of OOPs?**
   - Requires pre-work and proper planning.
   - In certain scenarios, programs can consume a large amount of memory.
   - Not suitable for a small problem.
   - Proper documentation is required for later use.

5. **What is the difference between class and structure?**

Class: User-defined blueprint from which objects are created. It consists of methods or sets of instructions that are to be performed on the objects.
Structure: A structure is basically a user-defined collection of variables of different data types.

6. **What is the difference between a class and an object?**

Class	Object
Class is the blueprint of an object. It is used to create objects.	An object is an instance of the class.
No memory is allocated when a class is declared.	Memory is allocated as soon as an object is created.
A class is a group of similar objects.	An object is a real-world entity such as a book, car, etc.
Class is a logical entity.	An object is a physical entity.
A class can only be declared once.	Objects can be created many times as per requirement.
An example of class can be a car.	Objects of the class car can be BMW, Mercedes, Ferrari, etc.

ENCAPSULATION

→ Encapsulation is about wrapping data and methods into a single class and protecting it from outside intervention.

→ It can be used to hide both data members and data functions or methods associated with an instantiated class or object.

→ Example :

```
#include <iostream>
using namespace std;
class Student {
private:
        string studentName;
        int studentRollNo;
        int studentAge;
public:
        string getStudentName()
        {
        return studentName;
        }
void setStudentName (string studentName)
{
this -> studentName = studentName;
}
int getStudentRollNo()
{
return studentRollNo;
}
void setStudentRollNo()
{
this -> studentRollNo = studentRollNo;
}
int getStudentAge()
{
return studentAge;
}
```

```
void setStudentAge(int studentAge)
{
this -> studentAge = studentAge;
}
};
int main()
{
Student obj;
obj.setStudentName("Sharad");
obj.setStudentRollNo(101);
obj.setStudentAge(18);
cout << "Student Name:" << obj.getStudentName() << endl;
cout << "Student Roll No:" << obj.getStudentRollNo() << endl;
cout << "Student Age:" << obj.getStudentAge();
}
```

Abstraction

→ Abstraction means providing only some of the information to the user by hiding its internal implementation details. We just need to know about the methods of the objects that we need to call and the input parameters needed to trigger a specific operation, excluding the details of implementation and type of action performed to get the result.

→ Abstraction is selecting data from a larger pool to show only relevant details of the object to the user. It helps in reducing programming complexity and efforts. It is one of the most important concept of OOPs.

→ Advantages of Abstraction :-

- Only you can make changes to your data and function, and no one else can.
- Increases the reusability of the code.

- It makes the application secure by not allowing anyone else to see the background details.
- Avoids duplication of your code.

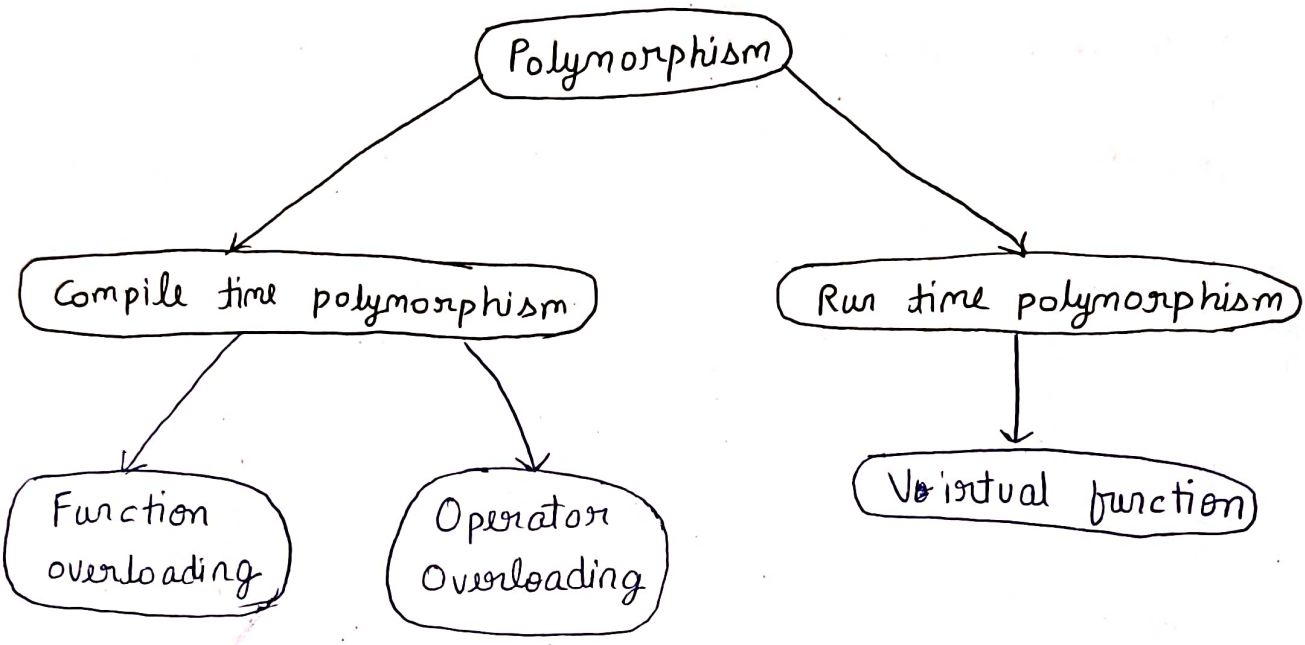
→ Example:

```
#include <iostream>
using namespace std;
class abstraction
{
private:
        int a, b;
public:
        void set (int x, int y)
        {
        a = x;
        b = y;
        }
  void display()
  {
  cout << " a = " << a << endl;
  cout << " b = " << b << endl;
  }
};
int main()
{
implementAbstraction obj;
obj. set (10, 20);
obj. display ();
return 0;
}
```

Polymorphism

→ Polymorphism is the combination of two Greek words, The poly means many, and morphs means forms.

→ It is a concept that allows you to perform a single action in different ways.



☆ Compile Time Polymorphism :-

→ Compile time polymorphism is also known as static poly---. This type of polymorphism can be achieved through function overloading and operator overloading.

(a) Function overloading : When there are multiple functions in a class with the same name but different parameters, these functions are overloaded. The main advantage of this is that it increases the program's readability. Functions can be overloaded by using different numbers of arguments or by using different types of arguments.

(b) Operator overloading : It is a type of polymorphism in which an operator is overloaded to give user defined meaning to it.

→ Overloaded operator is used to perform operation on user defined datatypes.

→ for example, '+' operator can be overloaded to perform addition on various datatypes like integer, string etc.

→ few operators which cannot be overloaded are -

1) Scope operator (::)
2) member selector (.) or dot operator (.).
3) size of
4) Ternary operator (?:).

→ Operator overloading can be done by implementing a function which can be a :-

1) member function.
2) non - member function.
3) friend function.

→ Operator Overloading using unary minus (-) :

```
#include <iostream>
using namespace std;
class space
{
int x ; int y ; int z ;
public :
        void getdata (int a , int b , int c);
        void display ();
        void operator - ();
};
void space :: getdata (int a , int b , int c)
{
x = a ; y = b ; z = c ;
}
void space :: display ()
{
cout << x << " " ;
cout << y << " " ;
cout << z << " " ;
}
```

```
void space :: operator - ()
{
x = -x;
y = -y;
z = -z;
}
int main ()
{
space s;
s.getdata (10, -20, 30);
cout << " S is " ;
s. display ();
- s;
cout << " s is " ;
s. display ();
return 0;
}
```

→ Operator Overloading using Binary operator :

```
#include <iostream>
using namespace std;
class loc
{
int longitude , latitude;
public :
loc ()
{ }
loc (int lg , int lt)
{
longitude = lg;
latitude = lt;
}
void show ()
{
cout << longitude;
cout << latitude;
}
```

```
loc operator + (loc op2);
};
loc :: operator + (loc op2)
{
loc temp;
temp.longitude = op2.longitude + longitude;
temp.latitude = op2.latitude + latitude;
return temp;
}
int main ()
{
 loc ob1 (10, 20);
loc ob2 (5, 30);
ob1.show ();
ob2.show ();
ob1 = ob1 + ob2;
ob1.show();
 return 0;
 }
```

★ Run time polymorphism :-

Run time polymorphism is also known as dynamic polymorphism. Method overriding is a way to implement runtime polymor...

Method overriding: Method overriding is a feature that allows you to redefine the parent class method in the child class based on its requirement. The child class is allowed to redifine that method based on its requirement. This process is called method overriding. It is possible through inheritance only.

Rules for method overriding:

→ The parent class method and the method of the child class must have the same name.

→ The parent class method and the method of the child class must have the same parameters.

→ Example:

```
#include <iostream>
using namespace std;
class Parent
{
public:
        void show()
        {
        cout << "Inside parent class" << endl;
        }
};
class subclass1 : public Parent
{
public:
        void show()
        {
        cout << "Inside subclass1" << endl;
        }
};
class subclass2 : public Parent
{
public:
        void show()
        {
        cout << "Inside subclass2" << endl;
        }
};
int main()
{
subclass1 o1;
subclass2 o2;
o1.show();
o2.show();
return 0;
}
```

Inheritance

→ The mechanism of deriving a new class from an old class is called Inheritance.

→ The old class is referred as Base class and new class is called derived or sub class.

→ The derived class inherits some or all of the behaviour from base class.

→ A class can also inherit property from more than one class or more than one level.

- **Defining derived class :-**

```
class derived-classname : visibility-mode base-class
{
members of derived class;
};
```

e.g. :-

```
class ABC : private xyz
{
 // members of ABC
};
class ABC : public xyz
{
// members of ABC
};
class ABC : xyz
{
// members of ABC
};
```

Q) WAP to create student details.

⇒

```
#include <iostream>
using namespace std;
class student
{
```

```
public :
    int roll;
    class har char name;
    int marks;
void getdata ()
{
cout << " Enter name = " ;
cin >> name;
cout << " Enter roll no. = " ;
cin >> roll;
cout << " Enter marks = " ;
cin >> marks;
 }
};
void display ()
{
cout << " Name is = " << name << endl;
cout << " Roll no. is = " << roll << endl;
cout << " Marks are = " << marks << endl;
}
int main ()
{
student obj;
obj. display ();
}
```

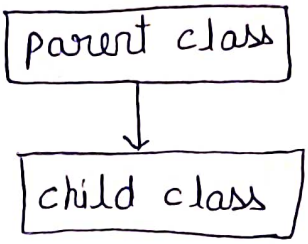
Q) WAP on "Virtual Function".

=>
```
#include <iostream>
#include <conio.h>
using namespace std;
class base
{
```

```
public :
    void display ()
    {
    cout << "\n display";
    }
  virtual void show ()
  {
  show cout << "\n show base";
  }
};
class derived : public base
{
public :
     void display ()
     {
     cout << "\n display derived";
     }
     void show ()
     {
     cout << "\n show derived";
     }
 };
int main ()
{
base b;
derived d;
base * bptr;
cout << "\n bptr points to base";
bptr = &b;
bptr → display ();
bptr → show ();
cout << "\n bptr points to derived";
bptr = &d;
bptr → display ();    bptr → show ();
return 0;
}
```

• Types of Inheritance :-

(1) Single Inheritance : When a single class is derived from a single parent class, it is called single inheritance.



Syntax -

```
class baseclass_name
{
 //methods;
};
class derivedclass_name : visibilitymode baseclass_name
{
methods;
};
```

Q) WAP to create and display properties of the student from the person class.

⇒

```
#include <iostream>
#include <conio.h>
using namespace std;
class person
{
private:
    char fname[10], lname[10], gender[10];
protected:
    int age;
public:
    void input_person();
    void display_person();
};
```

```
class student : public person
{
private :
    char clg_name [10];
    char level[10];
public :
    void input_student();
    void display_student();
};
void person :: input_person()
{
cout << " First name = ";
cin >> fname;
cout << " Last name = ";
cin >> lname;
cout << "Gender and age = ";
cin >> gender >> age;
}
void person :: display_person()
{
cout << " First name = " << fname << endl;
cout << " Last name = " << lname << endl;
cout << " Gender = " << gender << endl;
cout << " Age = " << age << endl;
}
void student :: input_student()
{
person :: input_person();
cout << " College = ";
cin >> clg_name;
cout << " Level = ";
cin >> level;
}
```

```
void student :: display_student()
{
person :: display_person();
cout << "College =" << clg_name << endl;
cout << "Level =" << level << endl;
}
int main()
{
student s;
cout << "!Input Data!" << endl;
s.input_student();
cout << "!Display Data!" << endl;
s.display_student();
return 0;
}
```

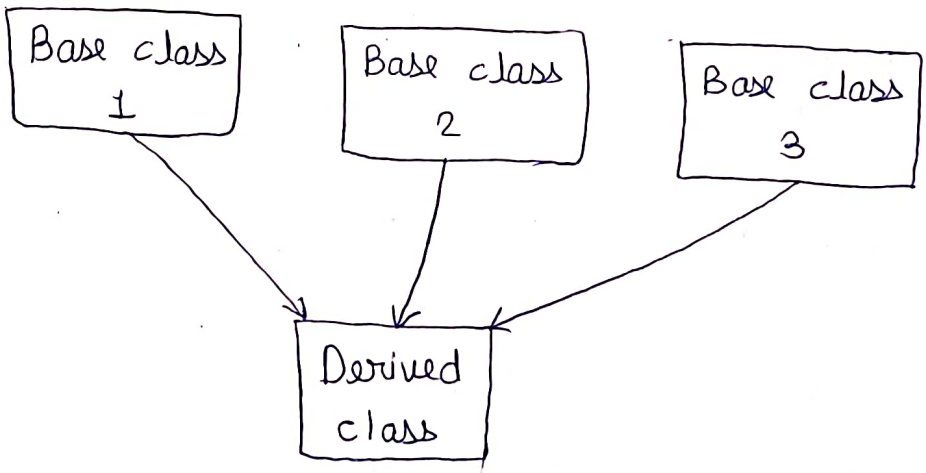
Q) Adding 2 numbers using single inheritance.

⇒
```
#include <iostream>
using namespace std;
class AddData
{
protected:
    int num1, num2;
public:
    void accept();
    {
    cout << "Enter first number =";
    cin >> num1;
    cout << "Enter second number =";
    cin >> num2;
    }
};
```

```
class addition : public AddData
{
int sum;
public :
    void add ()
    {
    sum = num1 + num2;
    }
    void display ()
    {
    cout << "Addition is =" << sum;
    }
};
int main ()
{
addition a;
a. accept ();
a. add ();
a. display ();
return 0;
}
```

---

(2) Multiple Inheritance : When a class is derived from two or more base classes, such inheritance is called "Multiple Inheritance".



→ It allows us to combine the features of several existing classes into a single class.

e.g. : A child has the characteristics of both mother and father.

Syntax -
```
class Baseclass_1
{
//methods
};
class Baseclass_2
{
//methods
};
class Baseclass_N
{
//methods
};
class derivedclass_name : visibilitymode Baseclass_1,
visibility_mode Baseclass_2, visibilitymode Baseclass_N
{
//methods
}
```

e.g.
```
#include <iostream>
using namespace std;
class student
{
protected:
    int rno, m1, m2;
public:
    void get()
    {
    cout << "Enter roll no:";
    cin >> rno;
    cout << "Enter 2 marks";
    cin >> m1 >> m2;
    } };
```

```
class Sports
{
protected:
    int sm;
public:
   void getsm()
   {
   cout << "Enter sports marks = ";
   cin >> sm;
   }
};
class statement : public student , public Sports
{
int tot , avg;
public: void. display()
        {
        total = m1 + m2 + sm;
        avg = tot /3;   cout << "Roll no = " << rno << endl;
        cout << "Total marks =" << tot << endl;
        cout << " Average marks =" << avg << endl;
        }
   };
  int main()
  {
  statement obj.
  obj. get();
  obj. getsm();
  obj. display();
  return 0;
  }
```

Q) WAP for calculating area and perimeter of a rectangle using multiple inheritance.

⇒
```
#include <iostream>
using namespace std;
class area
{
public:
        int l, b;
        void rarea ()
        {
        cout << " Enter length and breadth " ;
        cin >> l >> b;
        cout << " Area of rectangle is = " << l * b << endl;
        }
  };
class perimeter
{
 void repri ()
 {
 cout << " Enter length and breadth = " ;
 cin >> l >> b;
 cout << " perimeter of rectangle is = " << 2 * (l+b) << endl;
 }
 };
class rectangle : public rarea , public repri
{
void display ()
 {  int Area , perimeter;
 Area = l * b;
 perimeter = 2 * (l + b);
 cout << "Area = " << Area << endl;
 cout << " Perimeter = " << perimeter << endl;
  }
};
```

```
int main()
{
rectangle obj;
obj.rarea();
obj.reperi();
obj.display();
return 0;
}
```

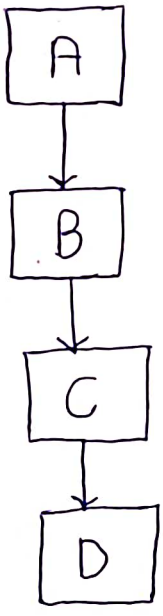
* The constructors of inherited class are called in the same order in which they are inherited.
* The destructors are called in the reverse order of constructors.

---

(3) Multi-level Inheritance - When a class is derived from a class which is also derived from another class i.e. a class having more than one parent classes, such inheritance is called Multi-level inheritance.

→ The level of inheritance can be extended to any number of level depending upon the relation.

→ It is similar to relation b/w grandfather, father & child.



Syntax -

```
class baseclassname
{
//methods;
};
class intermediateclassname : v.mode baseclassname
{
//methods;
};
class childclassname : v.mode intermediateclassname
{
//methods;
};
```

Q) WAP to create a programmer derived from employee which is itself derived from person using multi level inheritance.

→

```
#include <iostream>
using namespace std;
class Person
{
char name[100], gender[10];
int age;
public:
    void getdata()
    {
    cout << "Name = ";
    cin >> name;
    cout << "Gender =";
    cin >> gender;
    cout << "Age =";
    cin >> age;
    }
    void display()
    {
    cout << "Name =" << name << endl;
    cout << "Age =" << age << endl;
    cout << "Gender =" << gender << endl;
    }
};
```

```
class Employee : public Person
{
char company[100];
float salary;
public:
  void getdata()
  {
   person :: getdata();
   cout << "Name of company = ";
   cin >> company;
   cout << " Salary in Rs. = ";
   cin >> salary;
  }
 void display()
 {
  person :: display();
  cout << "Name of company = " << company << endl;
 cout << " Salary = Rs. " << salary << endl;
    }
};
class Programmer : public Employee
{
int number;
public:
 void getdata()
 {
  Employee : getdata();
  cout << " Number of programming languages Known = ";
  cin >> number;
 }
 void display()
 {
 Employee :: display();
cout << " Number of programming languages Known = " <<
          number << endl;
 } };
```

```
int main ()
{
 Programmer P;
 cout << "Enter data" << endl;
 P.getdata ();
 cout << " Display data" << endl;
 P. display ();
  return 0;
}
```

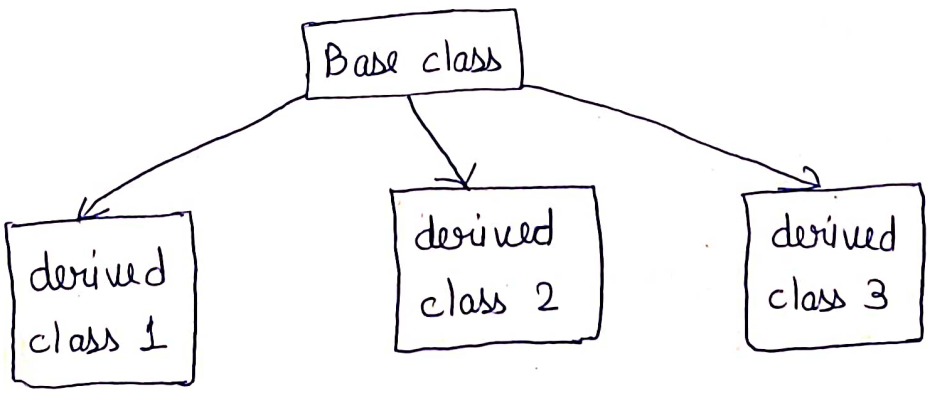
---

(4) Hierarchical Inheritance - When more than one classes are derived from a single base class, such inheritance is known as "hierarchical inheritance", where, features that are common in lower level are included in parent class.

e.g. civil, computer, IT, electrical are derived from engineer.

Syntax -

```
class baseclassname
{
 //methods;
 };
class derivedclass1 : v.mode baseclassname
 {
//methods;
 };
class derivedclass2 : v. mode baseclassname
{
//methods;
};
```



Q) WAP to create student and employee inheriting from person using hierarchical inheritance.

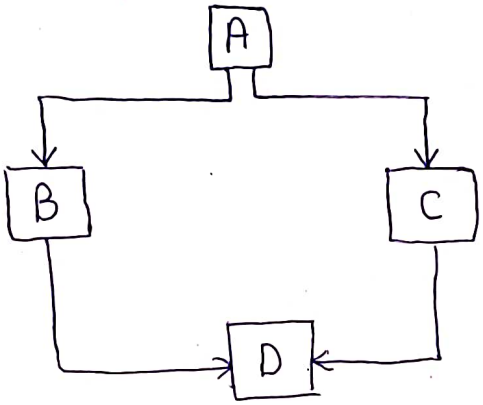
⇒
```
#include <iostream>
using namespace std;
class Person
{
char name[100], gender[10];
int age;
public:
    void getdata()
    {
    cout << "Name =";
    cin >> name;
    cout << "Age =";
    cin >> age;
    cout << "Gender=";
    cin >> gender;
    }
    void display()
    {
    cout << "Name is " << name << endl;
    cout << "Age is " << age << endl;
    cout << "Gender is" << gender << endl;
    }
};
class Student : public Person
{
char institute[100], level[20];
public:
    void getdata()
    {
    Person : getdata();
    cout << "Name of college and level =";
    cin >> institute >> level;
    }
```

```
void display ()
{
 person :: display ();
 cout << " Name of college =" << institute << endl;
 cout << " level =" << level << endl;
  }
};
class Employee : public Person
{
char company [100];
float salary;
public :
     void getdata ()
     {
     Person :: getdata();
     cout << " Name of company = ";
     cin >> company;
     cout << " Salary in Rs. =";
     cin >> salary;
     }
     void display ()
     {
     Person :: display ();
     Cout << " Name of company " << company << endl;
     cout << " Salary in Rs. =" << salary << endl;
      }
   };
     int main ()
     {
     Student S;
     Employee E;
     cout << " student";
     cout << " Enter data";
     s. getdata ();
```

```
cout << " display data";
E. display ();
return 0;
}
```

---

(5) Hybrid Inheritance - It is a combination of two or more inheritances such as single, multiple, multi-level or hierarchical inheritance.



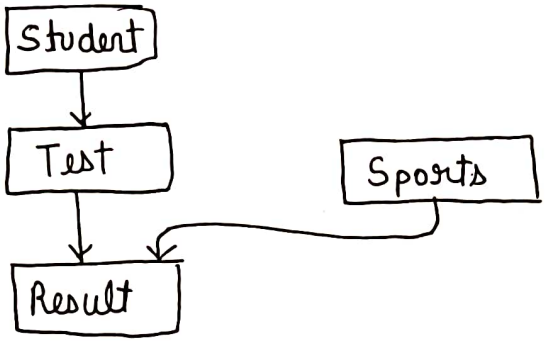
e.g.

```
#include <iostream>
using namespace std;
class arithematic
{
protected:
    int num1, num2;
public:
    void getdata()
    {
    cout << " for addition";
    cout <<" Enter the first number";
    cin >> num1;
    cout << " Enter the second number";
    cin >> num2;
    }
};
```

```
class plus : public arithematic
{
protected:
        int sum;
public:
      void add()
      {
      sum = num1 + num2;
      }
};
class minus
{
 protected:
        int n1, n2, diff;
 public:
      void sub()
      {
      cout << "\n For subtraction";
      cout << "Enter first number";
      cin >> n1;
      cout << "Enter second number";
      cin >> n2;
      diff = n1 - n2;
      }
    };
class result : public plus, public minus
{
 public:
      void display()
      {
      cout << "\n sum of" << num1 << " and " << num2 << " ="
                          << sum;
      cout << "\n difference of" << num1 << " and " << num2 << " ="
                          << diff;
      }
    };
```

```
int main()
{
result z;
z.getdata();
z.add();
z.sub();
z.display();
}
```

Q) WAP for hybrid inheritance.



⇒
```
#include <iostream>
using namespace std;
class student
{
protected:
    char name[100], clg_name[100];
public:
    void getdata()
    {
    cout << " Enter name of student = ";
    cin >> name;
    cout << " Enter name of college = ";
    cin >> clg_name;
    }
    void display()
    {
    cout << " Name of student is = " << name << endl;
```

```
    cout << " Name of College =" << clg_name << endl;
    }
};
class test : public student
{
 public :
     int m1, m2;
     void getdata()
     {
     student ::getdata();
     cout << " Enter marks of 2 subjects = " ;
     cin >> m1,>>m2;
     }
     void display ()
     {
     student :: display ();
     cout << " 1st marks = " << m1 << endl;
     cout << " 2nd marks =" << m2 << endl;
     }
  };
  class sports
  {
   public :
       int sm;
       void getdata()
       {
       cout << " Enter sports marks=";
       cin >> sm;
       }
       void display()
       {
       cout << " sports marks = " << sm << endl;
       }
    };
```

```
class result : public test , public sports
{
public :
    int total , avg;
    void getdata ()
    {
    test ::getdata ();
    sports :: getdata ();
    total = m1 + m2 + sm;
    avg = total /3;
    }
    void display ()
    {
    test :: display ();
    sports :: display ();
    cout << " Total marks =" << total << endl;
    cout << " Average marks =" << avg << endl;
    }
    };
int main ()
{
result r;
cout << " GET DATA" << endl;
r. getdata ();
cout << " DISPLAY DATA" << endl;
r. display ();
return 0;
}
```

Abstract Class

Abstract Class-

Abstract classes can't be instantiated, i.e., we cannot create an object of this class. However, we can derive a class from it and instantiate the object of the derived class. An Abstract class has at least one pure virtual function.

Properties of the abstract classes:

- It can have normal functions and variables along with pure virtual functions.
- Prominently used for upcasting(converting a derived-class reference or pointer to a base-class. In other words, upcasting allows us to treat a derived type as a base type), so its derived classes can use its interface.
- If an abstract class has a derived class, they must implement all pure virtual functions, or they will become abstract.

Example:

```
#include<iostream>
using namespace std;
class Base {
   public:
      virtual void s() = 0; // Pure Virtual Function
};

class Derived: public Base {
   public:
      void s() {
         cout << "Virtual Function in Derived_class";
      }
};

int main() {
   Base *b;
   Derived d_obj;
   b = &d_obj;
   b->s();
}
Output
Virtual Function in Derived_class
```

If we do not override the pure virtual function in the derived class, then the derived class also becomes an abstract class.

We cannot create objects of an abstract class. However, we can derive classes from them and use their data members and member functions (except pure virtual functions).

Interview Questions

1. **What is Encapsulation in C++? Why is it called Data hiding?**

   The process of binding data and corresponding methods (behavior) into a single unit is called encapsulation in C++.

   In other words, encapsulation is a programming technique that binds the class members (variables and methods) together and prevents them from accessing other classes. Thereby we can keep variables and methods safes from outside interference and misuse.

   If a field is declared private in the class, it cannot be accessed by anyone outside the class and hides the fields. Therefore, Encapsulation is also called data hiding.

2. **What is the difference between Abstraction and Encapsulation?**

Abstraction	Encapsulation
Abstraction is the method of hiding unnecessary details from the necessary ones.	Encapsulation is the process of binding data members and methods of a program together to do a specific job without revealing unnecessary details.
Achieved through encapsulation.	You can implement encapsulation using Access Modifiers (Public, Protected & Private.)
Abstraction allows you to focus on what the object does instead of how it does it.	Encapsulation enables you to hide the code and data into a single unit to secure the data from the outside world.
In abstraction, problems are solved at the design or interface level.	While in encapsulation, problems are solved at the implementation level.

3. **How much memory does a class occupy?**

   Classes do not consume any memory. They are just a blueprint based on which objects are created. When objects are created, they initialize the class members and methods and therefore consume memory.

4. **Are there any limitations of Inheritance?**
   Yes, with more powers comes more complications. Inheritance is a very powerful feature in OOPs, but it also has limitations. Inheritance needs more time to process, as it needs to navigate through multiple classes for its implementation. Also, the classes involved in Inheritance - the base class and the child class, are very tightly coupled together. So if one needs to make some changes, they might need to do nested changes in both classes. Inheritance might be complex for implementation, as well. So if not correctly implemented, this might lead to unexpected errors or incorrect outputs.

5. **What is the difference between overloading and overriding?**
   Overloading is a compile-time polymorphism feature in which an entity has multiple implementations with the same name—for example, Method overloading and Operator overloading.
   Whereas Overriding is a runtime polymorphism feature in which an entity has the same name, but its implementation changes during execution. For example, Method overriding.

6. **What are the various types of inheritance?**
   The various types of inheritance include:
   - Single inheritance
   - Multiple inheritances
   - Multi-level inheritance
   - Hierarchical inheritance
   - Hybrid inheritance

7. **What are the advantages of Polymorphism?**
   There are the following advantages of polymorphism in C++:
   a. Using polymorphism, we can achieve flexibility in our code because we can perform various operations by using methods with the same names according to requirements.
   b. The main benefit of using polymorphism is when we can provide implementation to an abstract base class or an interface.

8. **What are the differences between Polymorphism and Inheritance in C++?**
   The differences between polymorphism and inheritance in C++ are as follows:

a. Inheritance represents the parent-child relationship between two classes. On the other hand, polymorphism takes advantage of that relationship to make the program more dynamic.

b. Inheritance helps in code reusability in child class by inheriting behavior from the parent class. On the other hand, polymorphism enables child class to redefine already defined behavior inside parent class.
Without polymorphism, a child class can't execute its own behavior.

① What do you think are the major issues facing the software industry today?..

Ans -

- Security : We in daily life deal with internet so frequently in this deal we transact with our private information, our bank account info etc. So if we are going to develop a software, industry have to keep in mind the security should not lack in any case.
- Portability : Nowadays there are variety of devices are available for the communication like smart phone, tablets, laptops etc. So the portability is required in the software so that it can run smoothly on each device.
- Re - usability : In software development it is not easy to write a complex software code. It is not enough to put the chain of code together for software development. We need sound structure techniques of code writing, so that code written can be used in other software development also.
- Integrity : When we talk about integrity it mean compact, implies everything should be at one place. The software which you are going to develop it should be integrated with the technology so far we are using, and should fit in the future environment also.
- Friendliness : The software developed should be easy to use and friendliness so that user can interact with it without any essential training or wastage of time.

② What is procedure - oriented programming ? What are its main characteristics.

Ans- When we use Procedure language, we give instructions directly to our computer and tell it how to reach its goal through processes. Procedural programming focusses on the process rather than data and function.

The characteristics of procedural programming are :-

- It follows a top - down approach.
- The program is divided into blocks of codes called functions, where each function performs a specific task.
- The data and functions are detached from each other.
- The data moves freely in a program.
- It is easy to follow the logic of a program.

③ Discuss an approach to the development of procedural programming.

Ans- Program is broken down into procedures, or blocks of code that perform one task each. All procedures taken together from the whole program. It is suitable only for small programs that have low level of complexity.

e.g → For a calculator program that does simple mathematical calculations, each of the operations can be developed as seperate procedures. In the main program each procedure would be invoked on the basis of user's choice.

④ Describe how data are shared by functions in a procedure-oriented programming ?

Ans- They are two ways to share data in a program →

- Declaring global variables : Once we declare variables globally at the beginning of our program, the data of those variables can be accessed in each function and each part of that program.

- Passing values by function arguments: In C++, we use actual and formal parameters to share data & while calling a function. Actual parameters are the variables by which we pass the data and formal parameters are the variables which store that data for further execution.

⑤ ###What is Object Oriented Programming? How is it different from the procedure-oriented programming?

Ans- OOPs stands for Object Oriented Programming. It is the programming paradigm where everything is represented as an object. It is a methodology to design program using classes and objects. It simplifies the software development by providing some concepts such as:

→ Classes
→ Objects
→ Inheritance
→ Polymorphism
→ Abstraction
→ Encapsulation

- Procedural programming is about writing procedures or functions that perform operations on the data, while object-oriented programming is about creating objects that contain both data and functions.

⑥ How are data and functions organized in an object-oriented program.

Ans- Data and corresponding functions are organized using classes in object-oriented programming.

⑦ What are the unique advantages of an OOPs paradigm?

Ans- • OOP is faster and easier to execute.
• It provides a clear structure of the programs.

- It makes possible to create full reusable applications with less code and shorter development time.
- It helps to keep the code DRY → "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug.

(8) Distinguish between the following terms:

a)

Objects	Classes
• Object is an entity that has state and behaviour.	• Class is a group of similar objects. It is a templelate from which objects are created.
• Here, state means data and behaviour means functionality.	• It can have fields, methods, constructors etc.
• An object is created many times as per requirement.	• The class has to be declared only once.
• Objects are allocated memory space wherever they are created.	• When a class is created, no memory is allocated.
• It is created with a class name.	• It is declared with the class keyword.

b)

Data Abstraction	Data Encapsulation
• It hides the unnecessary details but shows the essential information.	• It wraps the data and code into a single unit so that the data can be protected from outside world.
• It focuses on the external lookout.	• It focuses on internal working.
• It can be implemented using abstract classes and interfaces.	• It can be implemented by using the access modifiers.

- It is the process of gaining information.

- It is the process of containing the information.

c) Inheritance

- It's basic motto is creating a new class inheriting the properties | functionalities of an already existing class.
- It is used to support the concept of reusability in OOP and reduces the length of code.
- There are 5 types of inheritance.
  → Single inheritance
  → Multiple inheritance
  → Multilevel inheritance.
  → Hierarchical inheritance.
  → Hybrid inheritance.

Polymorphism

- It defines the way in which a single task can be performed in multiple ways by the user.
- It allows objects to decide which form of the function to be invoked when, at compile time as well as run time.
- These are of 2 types.
  → compile time polymorphism.
  → run time polymorphism.

d) Dynamic Binding

- Binding refers to the linking of a procedure call to the code to be executed in response to the call.
- It allows execution of different codes using the same object at run time.

Message Passing

- The process of programming in which communication is involved is known as message passing.
- It allows developing communicati. between objects.
  It involves 3 basic steps:
  → Creating classes
  → Creating objects
  → Establishing communication among objects.

• It is also known as dynamic dispatch, late binding or run-time timing.	• It is also known as message exchanging.
• Object based programming does not support dynamic binding.	• Object based programming support message passing.
• OOPS support dynamic binding.	• OOPs also support message passing.

(9) What Kinds of things can become objects in OOP.

Ans - An object in OOP contains data members and member functions. And these data members and member functions solely depends on the design and the requirements.

(10) Describe inheritance as applied to OOP.

Ans - The capability of a class to inherit properties and characterstics from another class is called inheritance. When creating a class, instead of writing completely new data members and member functions, we can designate that the new class should inherit the members of an existing class. The existing class is called the base class and the new class is called derived class.

(11) What do you mean by dynamic binding? How is it useful in OOP?

Ans - The general meaning of binding is linking something to a thing. Here, linking of objects is done. In a programming sense, we can describe binding as linking function definition with the function call.

Dynamic binding helps us to handle different objects using a single function name. It also reduces the complexity and helps the developer to debug the code and errors.

(12) How does object oriented approach differ from object based approach?

Ans - • Object based approach :

→ It supports the usage of object and encapsulation.

→ It does not support inheritance and polymorphism.

→ It does not support built-in objects.

→ e.g. JavaScript, VB.

• Object oriented approach :

→ It supports all the features of OOPs including inheritance and polymorphism.

→ It supports built-in objects.

→ e.g. Java, C++ etc.

(13) List a few areas of application of OOP technology.

Ans - • Object Oriented databases.
- Hypertext and Hypermedia
- Office Automation Systems
- AI expert systems.
- Real time systems.

Review Questions

2.1 (a) True. All C++ programs will run under C++ compilers.

(b) True. A function contained within a class is called a member function.

(c) Yes, by looking at the input and output statements, we can easily recognize whether a program is written in C or C++.

(d)

(e) True. The concept of using one operator for different purposes is known as operator overloading.

(f) False. The output function printf() can be used in C++ but for that we need to include the header <cstdio>.

2.2 Why do we need the preprocessor directive #include <iostream>?

Ans- It is a C++ standard library header file for input output streams. It contains functions for input/output operations i.e., cin and cout.

2.3 How does a main() function in C++ differ from main() in C?

Ans- In C++, if we simply write main then it automatically assumes a return type int whereas in C, if we simply write main() than it assumes it as void main().

2.4 What do you think is the main advantage of the comment // in C++ as compared to the old C type comment.

Ans- In C language there is only a multiline comment i.e., /**/. But in C++, // represents a single line comment which can be used to provide information about a particular line of code in our program.

2.5 Describe the major parts of a C++ program.

Ans - Refer "Structure of C++ program"

Debugging Exercise

2.1
```
#include <iostream.h>
void main()
{
int i = 0;
i = i + 1;
cout << i << " ";
/* comment\ */ i = i + 1;
cout << i;
}
```

1) main cannot have void, should use return type int.
2) namespace std not used.
3) extra slash after a multi-line comment.

2.2
```
#include <iostream.h>
void main()
{
short i = 2500, j = 3000;
cout >> "i + j =" >> -(i + j);
}
```

1) main cannot have void, must use return type int.
2) namespace std not used.
3) insertion symbol used in cout.

2.3
```
#include <iostream.h>
void main()
{
int i = 10, j = 5;
int modResult = 0;
int divResult = 0;
modResult = i % j;
cout << modResult << " ";
divResult = i / modResult;
cout << divResult;
}
```

1) main cannot have void, must use return type int.
2) namespace std not used.

2.4 (a) `cout << "x = " x;`

missing extraction operator

(b) m = 5 ; || n = 10 ; || s = m + n ;

↳ int data type missing.

(c) cin >> x ; >> y ;

↳ semicolon not required

(d) cout << \n" Name : " << name;

↳ should be inside quotes

(e) cout << "Enter value : " ; cin >> x ; (No error)

(f) /* Addition */ z = x + y ; (no error)

Programming Exercise

2.1 WAP to display the following output using a single code statement.

Maths = 90
Physics = 77
Chemistry = 69

Ans →

```
#include <iostream>
using namespace std;
int main()
{
int m = 90, P = 77, c = 69;
cout << "Maths = " << m << "\n" << "Physics = " << p << "\n"
     << "Chemistry = " << c;

return 0;
}
```

2.2 WAP to read two numbers from the Keyboard and display the larger value.

Ans →

```
#include <iostream>
using namespace std;
int main()
{
 int a, b;
```

```
cout << " Enter 2 numbers = " << endl;
cin >> a >> b;
if (a>b)
cout << " larger number = " << a;
else
cout << " Larger number = " << b;
return 0;
}
```

2.3 WAP that inputs a character from Keyboard and displays its corresponding ASCII value.

Ans →

```
#include <iostream>
using namespace std;
int main()
{
char ch; int i;
cout << " Enter a character = " << endl;
cin >> ch;
i = ch;
cout << " Ascii value = " << i;
return 0;
}
```

2.4 WAP to read the values of a, b and c and display value of x,

$$x = a/b - c$$

Ans-

```
#include <iostream>
using namespace std;
int main()
{
int a, b, c, x;
cout << " Enter values of a, b and c = " << endl;
cin >> a >> b >> c;
x = a/b - c;
cout << "value of x is " << x;
return 0;
}
```

(a) a = 250, b = 85, c = 25

x = -23

(b) a = 300, b = 70, c = 70

x = -66

2.5 WAP that will ask for a temperature in Fahrenheit and display it in Celsius.

Ans →

```
#include <iostream>
using namespace std;
int main()
{
int f, c;
cout << "Enter temperature in fahrenheit = " << endl;
cin >> f;
c = (f - 32) * 0.555;
cout << "Temperature in celsius = " << c;
return 0;
}
```

2.6 Redo ques 5 using a class called temp and member functions.

Ans →

```
#include <iostream>
using namespace std;
class temp
{
public:
int c;
void calculate (int f)
{
c = (f - 32) * 0.555;
}
void print()
{
cout << "Temperature in celsius is = " << c;
}
};
```

```
int main()
{
int fa;
cout << "Enter temperature in fahrenheit =";
cin>>fa;
temp obj;
 obj.calculate(fa);
 obj.print();
 return 0;
 }
```

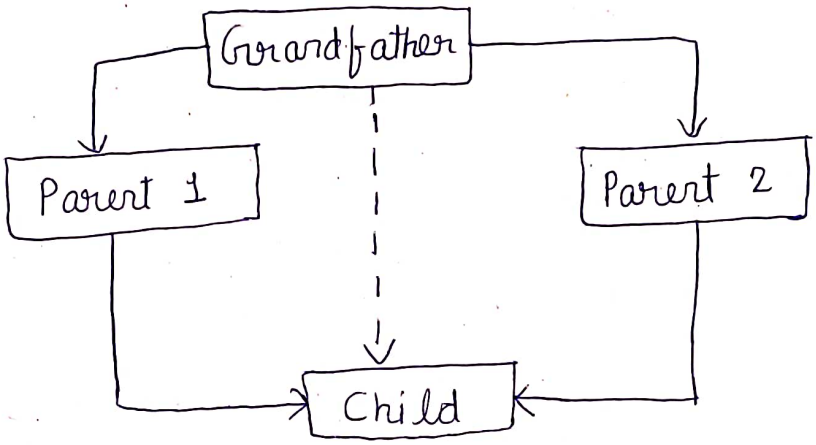
Virtual Base Class

→ An element of ambiguity can be introduced into a C++ program when multiple base classes are inherited.

→ The duplication of inherited members due to multiple path can be avoided by making the common base class as virtual base class.

→ The only difference b/w a normal base class and a virtual one is what occurs when an object inherits the base more than once.

→ If virtual base classes are used, then only one base class is present in the object. Otherwise, multiple copies will be found.



e.g. →

```
#include <iostream>
using namespace std;
class Base
{
public:
    int i;
};
class derived1 : virtual public base
{
 public:
     int j;
 };
```

```
class derived2 : virtual public base
{
Public :
    int K;
};
class derived3: public derived1 , public derived2
{
public :
    int sum;
};
int main()
{
derived3 ob;
ob.i = 10;
ob.j = 20;
ob.K = 30;
ob.sum = ob.i + ob.j + ob.K ;
cout << ob.i << " " << ob.j << " " << ob.K << " ";
cout << ob.sum;
return 0;
}
```

- Syntax :-

```
class A   //grandfather
{
- - - -
};
class B1 : virtual public A //parent 1
{
- - - -
};
class B2 : virtual public A //parent 2
{
- - - -
};
class C : public B1, public B2
{
- - - -        //only 1 copy of A will be inherited
- - - -
};
```

e.g.

```
#include <iostream>
using namespace std;
class student
{
protected:
    int roll_number;
public:
void get_number (int a)
{
roll_number = a;
}
void put_number (void)
{
cout << "roll no :" << roll_number;
}
};
class test : virtual public student
{
protected:
    float part1, part2;
public:
void get_marks (float x, float y)
{
part1 = x;
part2 = y;
}
void put_marks ()
{
cout << "Marks obtained" << endl;
cout << "part 1 :" << part1 << endl;
cout << "part 2 :" << part2 << endl;
}
};
```

```
class sports : virtual public student
{
protected:
    float score;
public:
 void get_score (float s)
 {
 score = s;
 }
 void put_score ()
 {
 cout << "Sports score = " << score << endl;
 }
};
class result : public test, public sports
 {
 float total;
 public:
    void display ();
 };
 void result :: display ()
 {
 total = part1 + part2 + score;
 put_number ();
 put_marks ();
 put_score ();
 cout << "Total marks = " << total;
 }
 int main ()
 {
 Result S1;
 S1. get_number (678);
 S1. get_marks (17, 27);
 S1. get_score (6.0);
 S1. display ();
 return 0;
 }
```

Method Overriding

→ Redefining a base class function in the derived class to have our own implementation is referred as "overriding".

→ Often confused with overloading which refers to using the same function name but with a diff signature. However, in overriding the function signature is the same.

→ It is a language feature that allow a sub-class (or) child class to provide a specific implementation of a method that is already provided by one of its super class (or) parent class.

→ The implementation in the class by providing a method that has same name, same parameters, same return types has a method in the parent class.

→ The version of the method that is executed will be determined by the object that is used to invoke it.

→ If an object of parent class is used to invoke the method, than the version in the parent class will be executed. But if the object of sub-class is used to invoke the method, than the version in the child class will be executed.

e.g.

```
#include <iostream>
using namespace std;
class Baseclass
{
public:
    void display()
    {
    cout << "\n This is display method of base class";
    }
    Void show()
    {
    cout << "This is show method of Base class" << endl;
    }
};
```

```
class derivedclass : public Baseclass
{
public:
    void display()
    {
    cout << "This is display method of derived class" << endl;
    }
};
int main()
{
derivedclass dr;
dr.display();
dr.show();
return 0;
}
```

Protected Members

- When the inheritance is protected, then :-

1) private members of base class are not accessible to derive the class.
2) protected members of base class remain protected in derived class.
3) public members of base class become protected in derived class.

e.g.

```
#include <iostream>
using namespace std;
class Base
{
protected:
    int i, j;
public:
void setij(int a, int b)
{
i = a;
j = b;
}
```

```
void showij()
{
cout <<i<<j;
}
};
class Derived : protected Base
{
int K;
public:
   void setK()
   {
    setij();
     K = i+j;
   }
  void showall()
  {
  cout << K<< show();
  }
  };
  int main()
  {
   Derived ob;
  ob. setij(3, 5);
  ob. setK();
  }
```

POINTERS

→ A variable which can store address of another variable is called pointer.

→ A pointer is a variable that holds a memory address. Pointers provide an alternative approach to access other data objects.

→ A pointer variable defines where to go and get the value of a specific data variable instead of defining actual data.

→ The declaration of the pointer variable takes the following form :-

Syntax - data type *pointer_variable.

→ C++ provides 2 special operators known as "&", "*".

= "&" stands for address and it is used to retrieve the address of a variable.

= "*" stands for value at address and it is used to access the value and location by means of its address.

e.g.

```
#include <iostream>
using namespace std;
int main()
{
int i=10;
int *ip;
float f=3.4, *fp;
char c='a', *cp;
cout << "i =" << i << endl;
cout << "f=" << f << endl;
cout << "c =" << c << endl;
ip = &i;
```

```
cout << " Address of i = " << ip << endl;
cout << " Value of i = " << *ip << endl;
fp = &f;
cout << " Address of f = " << fp << endl;
cout << " Value of f = " << *fp << endl;
cp = &c;
cout << " Address of c = " << cp << endl;
cout << " Value of c = " << *cp << endl;
return 0;
}
```

This pointer

→ This pointer is a pointer that is accessible only inside the member functions of a class and points to the object who has called "this" member function.

→ When a member function is called using an object, the address of the object is implicitly passed to the member function as the first argument. The member function collects this address in a pointer named "this".

→ When a member function is called, it is automatically passes an implicit argument. i.e. a pointer to the invoking object.

→ Friend functions do not have "this" pointer, as friends are not members of a class. Only member functions have a "this" pointer.

→ We can put "this" pointer to use in 2 situations:-

1) To distinguish b/w members of a class and the formal arguments when they have the same names.

2) To make a member function return the object which invokes it.

→ The member data of the object can be accessed through the pointer by the use of the operator " → " with the general form as.

```
this → memberdata;
```

e.g. ①

```
#include <iostream>
using namespace std;
class student
{
 int roll;
 char name[25];
 float marks;
 public:
student (int R, float mks, char nm[])
 {
roll = R;
 marks = mks;
 strcpy (Name, nm);
 }
 student (char name[], float marks; int roll)
 {
 roll = roll;
 strcpy (name, name);
 marks = marks;
 }
 student (int roll, char name[], float marks)
 {
 this → roll = roll;
 strcpy (this → name, name);
 this → marks = marks;
 }
```

```
void display()
{
 cout << " Roll =" << roll << endl;
 cout << " Name =" << name << endl;
 cout << " Marks =" << marks << endl;
 }
};
int main()
{
student s1(1, 89.6, "CVSR");
student s2("Kumar", 78.5, 2);
 s1. display();
 s2. display();
}
```

e.g. ②

```
#include <iostream>
using namespace std;
class sample
{
 int m_value;
 public:
 sample()
 {
 m_value = 10;
 }
 void display()
 {
 cout << m_value << endl;
 cout << this -> m_value << endl;
 }
};
int main()
{
sample obj;   obj.display();
return 0;
}
```

Pure Virtual function →

Syntax :

```
class Foo
{
virtual void bar() = 0;
};
```

Pure Virtual Function

```
class Foo
{
virtual void bar()
{
- - - -
}
};
```

Virtual Function

e.g. →

```
#include <iostream>
using namespace std;
class Account
{
 protected:
      int accno;
      char name[20];
 public:
     virtual void get() = 0;
     virtual void display() = 0;
 };
class sb_ac : public Account
{
private :
     float roi;
public:
     void get()
     {
     cout << "Enter acc no., name and roi = " <<endl;
     cin >> accno >> name >> roi;
     }
```

```
void display()
{
cout << "Account number = " << accno << endl;
cout << " Name =" << name << endl;
cout << " Rate of interest" << roi << endl;
}
};
class cur_acc : public Account
{
private :
    float odlimit;
public :
    void get()
    {
    cout << " Enter acc no, name and odlimit" << endl;
    cin >> accno >> name >> odlimit ;
    }
    void display()
    {
    cout <<" Account number = " << accno << endl;
    cout << " Name =" << name << endl;
    cout << " Od Limit =" << odlimit << endl;
    }
};
int main()
{
Account *ap;
sab_ac s;
cur_acc c;
ap = &s;
ap -> get();   ap -> display();
ap -> &c;
ap -> get();   ap -> display();
return 0;
}
```

STRUCTS

→ Structures in C++ are user defined data types which are used to store group of items of non-similar data types.

→ Syntax :-

```
struct structure_name
{
// member declaration
};
```

→ Structure variable can be defined as :-

```
Student s;
```

Here s is a structure variable of type Student. When the structure variable is created, the memory will be allocated.

→ The variable of the structure can be accessed by using the instance of the structure followed by the dot (.) operator and then the field of the structure.

e.g. `s.id = 4;`

e.g. →

```
#include <iostream>
using namespace std;
struct Rectangle
{
int width, height;
};
int main ()
{
struct Rectangle rec;
rec.width = 8;
rec.height = 5;
cout << "Area of rectangle = " << (rec.width * rec.
                                    height);
return 0;
}
```

e.g. →

```
#include <iostream>
using namespace std;
struct Rectangle
{
int weight, height;
Rectangle (int w, int h)
 {
 width = w;
 height = h;
 }
void area_of_rectangle ()
 {
cout << "Area of rectangle = " << (width * height);
 }
};
int main ()
{
struct Rectangle rec = Rectangle (4, 6);
rec.area_of_rectangle ();
return 0;
}
```

Q) Difference between structure and class?

Ans-

Structure	Class
= If access specifier is not declared explicitly, then by default it will be public.	= If access specifier is not declared explicitly, then by default it will be private.
= The instance of the structure is known as "structure variable."	= The instance of the class is known as "object of class".
= It doesn't support inheritance.	= It supports inheritance.
= The values allocated to structure are stored in stack memory.	= The reference type (before creating an object) is allocated on heap memory.

FILE I/O streams

- **Concepts of streams :-**

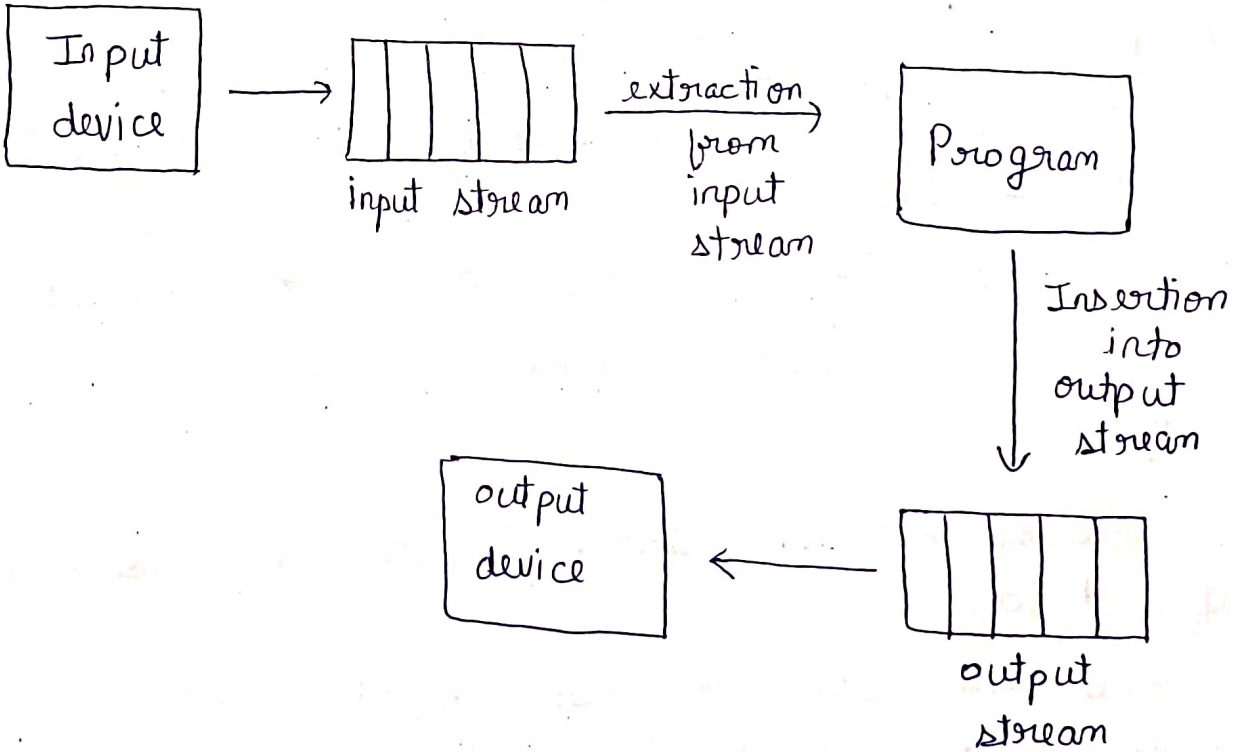
→ C++ supports a rich set of "io" functions and operations.

→ C++ supports all "C" rich set of "io" functions but we cannot use them due to 2 reasons -

1) "IO" methods in C++ supports the concept of OOPs.
2) "IO" methods in "C" cannot handle the user defined data types such as class objects.

→ The I/O system in C++ is designed to work with a wide variety of devices including terminals, disks and tape drives. Although each device is very diff., the I/O system supplies an interface to the programmer that is independent of the actual device being accessed. This interface is known as stream.

→ C++ uses the concept of stream & stream classes to implement its IO operations with the console and disk files.



- **Stream** : A stream is a sequence of bytes. It acts either as a source from which the input data can be obtained or as a destination to which the output data can be sent.

→ The data in the input stream can come from the Keyboard or any other storage device.

→ Similarly, the data in the output stream can go to the screen (or) any other storage device.

→ A stream acts as an interface b/w the program and the I/o device. Therefore, C++ handles data independent of the devices used.

→ C++ contains several pre-defined streams that are automatically opened when a program begins its execution.

→ **cin** :-

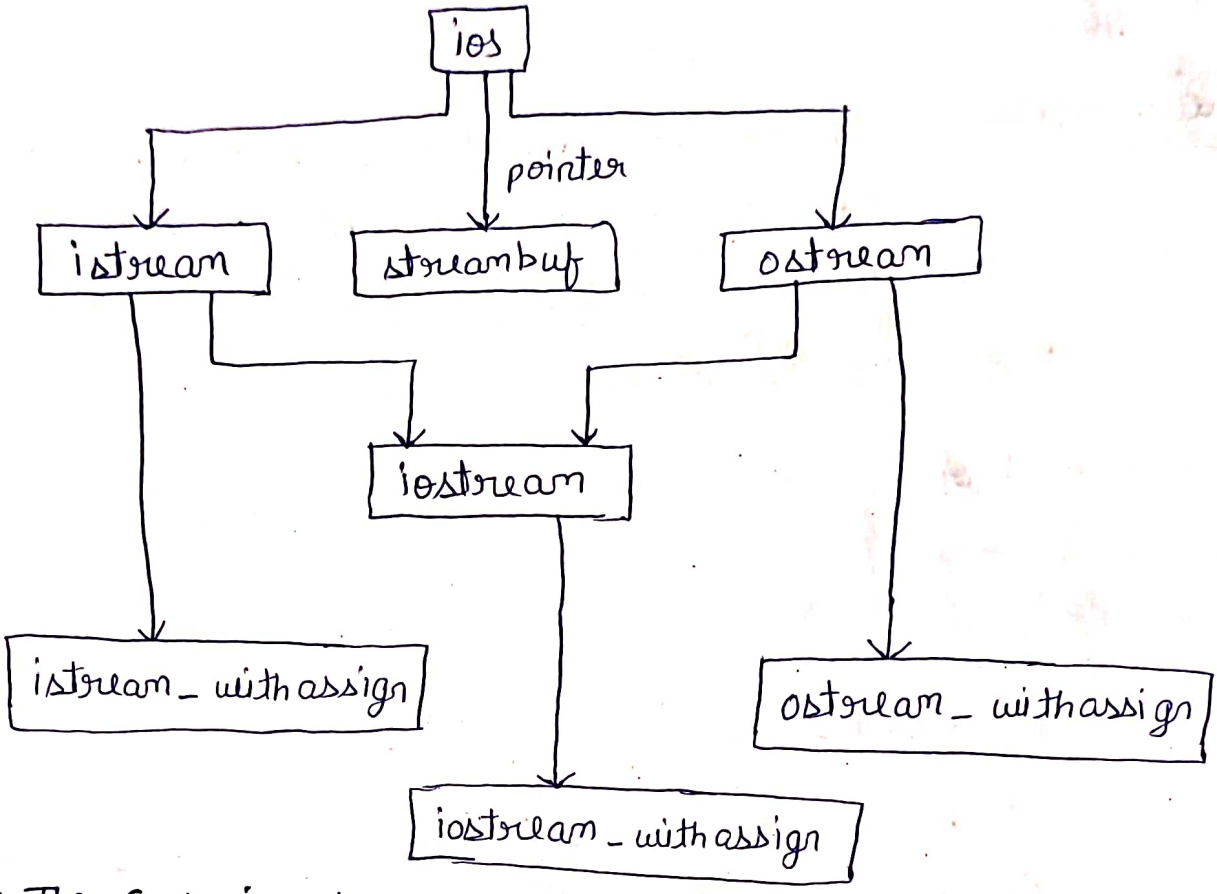
- It represents the input stream connected to the standard input device.
- The cin object which is responsible for input is made available to the program when you include the iostream header file.
- The overloaded ~~extraction~~ insertion operator [>>] to enter data into program variables.
- The general syntax of reading data from Keyboard.

```
cin >> variable
```

→ **cout** :-

- It represents the output stream connected to the standard output device.
- Use cout along with overloaded extraction operator [<<] to write strings, integer and other numeric data to the screen.

C++ stream classes



→ The C++ iosystem consists a hierarchy of class that are used to define various streams to deal with both the console and disc files. These classes are called stream classes.

→ These classes are declared in headerfile iostream.

→ **ios**:- It is the base class for istream and ostream which are inter base classes of iostream. The class ios is declared as the virtual base class.

→ **istream** :- It inherits the properties of ios. It declares input functions such as get, getline and read functions which contains overloaded insertion operator.

→ **ostream** :- It inherits the properties of ~~both istream~~ ios. It declares output functions such as put and write which contains overloaded extraction operator.

- **iostream** :- It inherits the properties of both istream and ostream by multiple inheritance and contains all the input and output functions.
- **Stream buffer** :- It provides an interface to physical devices through buffers which acts as base for file buffer class used in ios file.

☆ **Unformatted I/O operations** →

1) Overloaded operators :-
   a) >>
   b) <<

2) put and get functions :-

→ The classes istream and ostream define two member fn. get() & put(), to handle the single character I/P or O/P.

→ There are two types of get() functions :-
   a) get (void)
   b) get (char*)

→ To fetch a character including the blank space or tab and new line, get (char*) assigns the I/P character to its argument and get(void) returns the input character.

e.g. -
```
char c;
cin.get(c);
while (c != "\n")
{
cout << c;
cin.get(c);
}
```

→ The function put(), a member of ostream class can be used to output a line of text, character by character.

eg. -

```
cout.put('x');
cout.put(ch);
```

Q) Program to illustrate put() and get().

⇒

```
#include <iostream>
using namespace std;
int main()
{
int count = 0;
char c;
cout << "Input text";
cin.get(c);
while (c != '\n')
{
cout.put(c);
count++;
cin.get(c);
}
cout << "\n number of characters =" << count;
return 0;
}
```

3) getline() and write functions :-

→ The getline() reads a whole line of text that ends with a newline character. This function can be invoked by object cin.

syntax :

```
cin.getline(line, size);
```

→ The reading is terminated as soon as either the newline character is encountered or size of characters are readed.

e.g: char name [20];
cin.getline (name, 20);

→ The write() function displays an entire line and has the syntax:-

cout.write (line, size);

Q Program to illustrate getline() :-

⇒
```
#include <iostream>
using namespace std;
int main()
{
 int size = 20;
 char city [20];
 cout << " Enter city name =";
 cin >> city;
 cout << "City =" << city << endl;  cin.ignore();
 cout << "Enter city name again =";
 cin.getline (city, size);
 cout << "New city =" << city << endl;
 return 0;
}
```

Q Program to display the string using write().

⇒
```
#include <iostream>
#include <string>
using namespace std;
int main()
 {
char* string1 = "c++";
char* string2 = " programming";
```

```
int m = strlen (string1);
int n = strlen (string2);
for (int i=1; i<n; i++)
{
 cout.write (string2, i);
 cout << endl;
}
for (int i=n ; i>0; i--)
{
cout.write (string2, i);
cout << endl;
}
// concatenate strings
cout.write (string1, m).write (string2, n);
return 0;
}
```

☆ Formatted I/O operations :-

→ C++ supports a number of features that could be used for formatting the output.

→ These features include -

a) IO class functions & flags

b) Manipulators

c) user-defined O/P functions.

1) IO class functions & flags :-

→ IOS class contains a large number of member functions that could help us to format the output in a number of ways.

→ The most important member functions are :-

- width()
- precision()
- fill()
- setf()
- unsetf()

* width () :

→ we can use the width() to define the width of a field necessary for the O/P of an item.

→ Since it is a member function, we need to use an object to invoke it.

```
cout.width(w);
```

where, w = field width

e.g. cout.width(5);
cout << 543 << 12 << endl;

output →

5	4	3	1	2

Q. Program to illustrate width() :-

⇒
```
#include <iostream>
using namespace std;
int main()
{
int items[4] = [10, 8, 12, 15];
int cost[4] = [75, 100, 60, 99];
cout.width(5);
cout << "ITEMS";
Cout.width(8);
cout << "costs";
cout.width(15);
cout << "Total value =" << endl;
int sum = 0;
for (int i=0; i<4; i++)
{
 cout.width(5);
cout << items[i];
cout.width(8);
cout << cost[i];
```

```
int value = items[i] * cost[i];
cout.width(15);
cout << value << endl;
sum = sum + value;
}
cout << "\n Grand Total =";
cout.width(2);
cout << sum << "\n";
return 0;
}
```

---

* Precision():

→ By default, the floating nos. are printed 6 digits after the decimal point.

→ We can specify the no. of ~~objects~~ digits to be displayed after the decimal point while printing the floating point nos.

→ This can be done using the precision() member function.

→ Syntax:-

```
cout.precision(d);
```

d → no. of digits to the right of the decimal point.

Q. Program to illustrate for setting the precision:-

⇒
```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
cout << "precision set to 3 digits \n\n";
cout.precision(3);
cout.width(10);
```

```
cout << "Value";
cout.width(15);
cout << "sqrt of value" << "\n";
for (int n=1; n<=5; n++)
{
    cout.width(8);
    cout << n;
    cout.width(13);
    cout << sqrt(n) << "\n";
}
cout << "\n Precision set to 5 digits\n\n";
cout.precision(5);
cout << "sqrt(10) = " << sqrt(10) << "\n\n";
cout.precision(0);
cout << "sqrt(10) =" << sqrt(10) << "(default setting)\n";
return 0;
}
```

---

* filling and padding :- fill()

→ We have been printing the values using much larger field widths than required by the values. The unused positions of the field are filled with white spaces, by default.

→ We can use the fill function to fill the unused positions by any desired character.

```
cout.fill(ch);
```

Q Program to illustrate the fill():-

⇒
```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
cout.fill('<');
cout.precision(3);
for(int n=1; n<=6; n++)
{
 cout.width(5);
 cout<<n;
 cout.width(10);
 cout<<1.0/float(n)<<"\n";
 if(n==3)
 cout.fill('>');
}
cout<<"\n padding changed\n\n");
cout.fill('#');
cout.width(15);
cout<<12.345678<<"\n";
return 0;
}
```

---

* Setting the formatted flags :- setf()

→ To set a flag, use the setf() function.

→ This function is a member of ios.

→ Each stream is associated with a set of formatted flags that control the way in which information is formatted.

→ Syntax:-

```
stream.setf(ios::showpos);
        or
cout.setf(arg1, arg2);
```

where, arg1 → one of the formatted flag defined in the ios class. It specifies the format action required for the output.

arg2 → specifies the group to which the formatting flag belongs.

Format required	Flag (arg 1)	Bit-field (arg 2)
Left justified output	ios::left	ios::adjustfield
Right justified output	ios::right	ios::adjustfield
Padding after sign or base indicator	ios::internal	ios::adjustfield
Scientific notation	ios::scientific	ios::floatfield
Fixed point notation	ios::fixed	ios::floatfield
Decimal Base	ios::dec	ios::basefield
Octal Base	ios::oct	ios::basefield
Hexadecimal Base	ios::hex	ios::basefield

e.g.

→ cout.setf (ios::left , ios::adjustfield);

→ cout.setf (ios::scientific , ios::floatfield);

→ cout.setf (ios::showpoint);

→ cout.setf (ios::showpos);

* **Clearing the format flags :-**

→ The complement of setf() function is unsetf() function.

→ This member function of ios is used to clear 1 or more format flags.

→ Syntax :-

```
void unsetf (fmtflag flags);
```

Q Formatting flags using setf() function.

⇒

```
#include <iostream>
#include <cmath>
using namespace std;
int main ()
{
cout.fill ('*');
cout.setf (ios::left , ios::adjustfield);
cout.width(10);
cout << "value";
cout.setf (ios::right ; ios::adjustfield);
cout.width (15);
cout << "sqrt of value" << "\n";
cout.fill ('.');
cout.precision (4);
cout.setf (ios::showpoint);
cout.setf (ios::showpos);
cout.setf (ios::fixed , ios::floatfield);
for (int n=1; n<=10; n++)
{
    cout.setf (ios::internal , ios::adjustfield);
    cout.width (5);
    cout << n;
    cout.setf (ios::right , ios::adjustfield);
    cout.width (20);
    cout << sqrt(n) << "\n";
}
```

```
cout.setf(ios::scientific, ios::floatfield);
cout << "\n sqrt(100) = " << sqrt(100) << "\n";
return 0;
}
```

Q Program to illustrate unsetf() :-

⇒
```
#include <iostream>
using namespace std;
int main()
{
cout.setf(ios::uppercase | ios::scientific);
cout << 100.12;
cout.unsetf(ios::uppercase);
cout << "\n" << 100.12;
return 0;
}
```

☆ MANIPULATORS :-

→ The header file "iomanip" provides a set of functions called Manipulators, which can be used to manipulate the output formats.

→ They provide the same features as that of the ios member functions & flags.

→ setw(int w) ——— set field width to w.

→ setprecision(int d) ——— set floating point precision to d.

→ setfill(int c) ——— set fill characters to c.

→ setiosflags(long f) ——— set format flag

→ resetiosflags(long f) ——— reset format flag (clear)

→ endl ——— new line

Examples =

①
```
#include <iostream>
#include <iomanip>
void main()
{
int x1 = 123, x2 = 234, x3 = 789;
cout << setw(8) << "Exforsys" << setw(20) << " Values" << endl;
cout << setw(8) << " test123" << setw(20) << x1 << endl;
cout << setw(8) << " exam234" << setw(20) << x2 << endl;
cout << setw(8) << " result789" << setw(20) << x3 << endl;
}
```

②
```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
cout << setw(15) << setfill('*') << 99 << 97 << endl;
return 0;
}
```

③
```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
float x = 0.1;
cout << fixed << setprecision(3) << x << endl;
cout << scientific << x << endl;
return 0;
}
```

FILE HANDLING

→ File handling concept in C++ language is used to store a data permanently in computer.

→ Using file handling, we can store our data in secondary memory.

- **File** : The information or data stored under a specific name on a storage device is called a file.

- **Text file** : It is a file that stores information in ASCII characters. In text files, each line of text is terminated with a special character known as "EOL" (end of line) character or de-limiter character.

- **Binary file** : It is a file that contains information in the same format as it is held in memory. In Binary files, no de-limiters are used for a line. A program involves either or both types of data communication.

① Data transfer b/w the console unit and program.

② Data transfer b/w the program and disk file.

→ The File IO system of C++ contains a set of classes that define the file handling method.

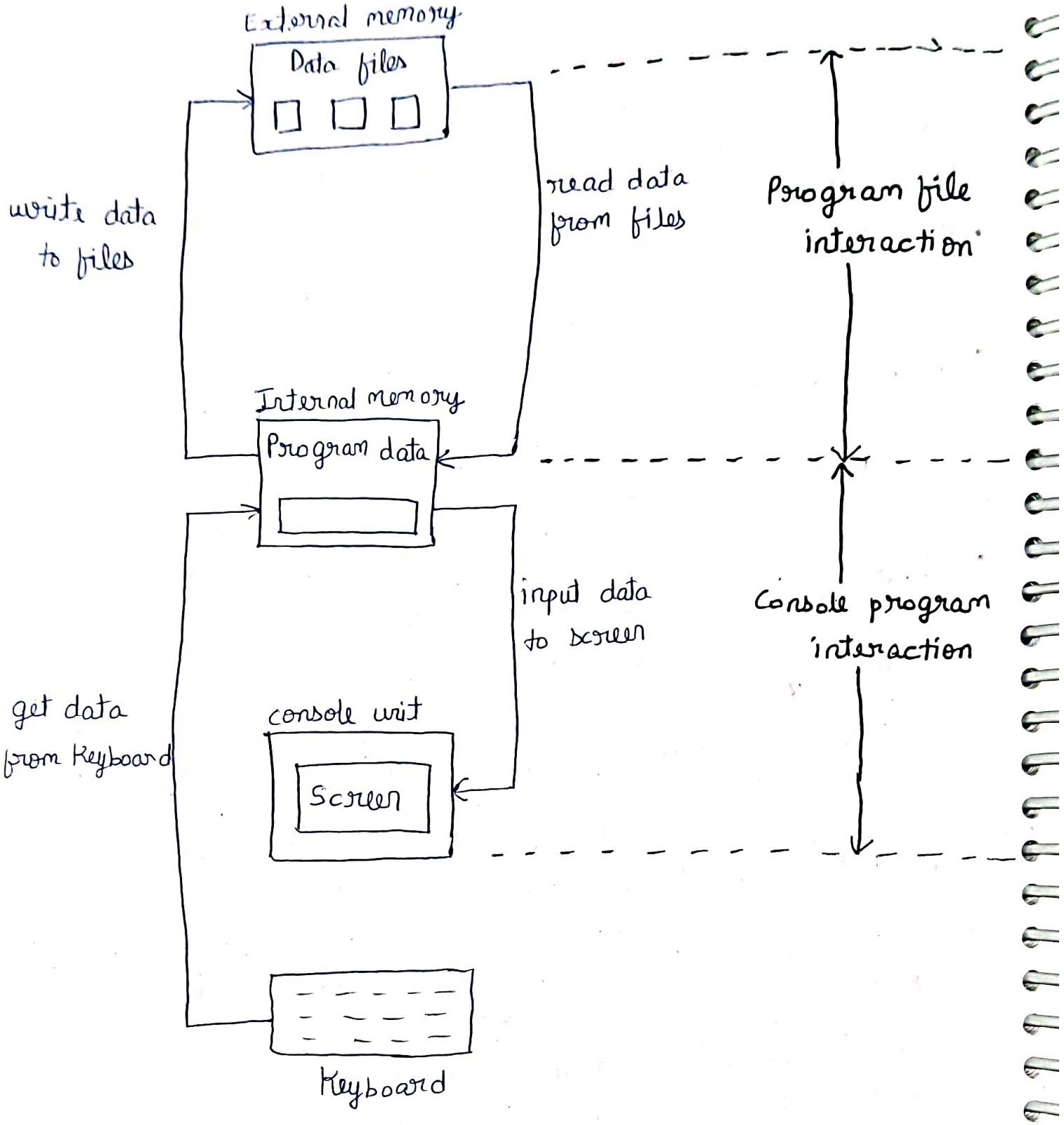
→ These classes are designed to manage the disk files (or) declared in header file called "fstream".

→ The operations on files are performed by using streams too. For this purpose, 3 classes exists :-

1) ofstream

2) ifstream

3) fstream



⇒ Console - program - file interaction

- ofstream :- Stream used for output to files. It inherits the functions put() and write() along with the functions supporting random access from ostream class defined inside the header file <iostream.h>.
- ifstream :- Stream used for input from files. It inherits the functions get(), getline() and read().
- filebuf :- It sets the file buffer to read and write. It contains close() and open() member functions.
- fstream :- Streams for both input and output operations. It provides support for simultaneous I/P or O/P operation.
- fstream base :- It is a base class for fstream, ifstream and ofstream classes. It provides operations common to these file streams. It also contains open() and close().

Q Program for opening and closing of a file.

⇒
```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
ofstream outf("ITEM");
cout << "enter name";
char name[30];
cin >> name;
outf << name;
cout << "Enter item cost:";
float cost;
cin >> cost;
```

```
outf << cost << "\n";
outf.close();
ifstream inf("ITEM");
inf >> name;
inf >> cost;
cout << "\n";
cout << "Item name = " << name << endl;
cout << "Item cost = " << cost << endl;
inf.close();
return 0;
}
```

☆ Opening and closing of File :-

→ For opening a file, we must first create a file stream and link it to the file name.

→ A file stream can be defined using the classes ifstream, ofstream and fstream that are contained in header file fstream.

→ A file can be opened in 2 ways :-

1) Using constructor function of class.
2) Using member function open() of the class.

The first method is useful when we use only one file in the stream.

The second method is used when we want to manage multiple files using one stream.

1) opening file using constructor –

This method involves the following steps:

a) Create a file stream object to manage the stream using the appropriate class.

b) Initialize the file object with the desired file name.

e.g. 1

```
ofstream outfile ("result");
```

ofstream ← file stream; outfile ↓ object; "result" → file name.

→ This creates outfile as an ofstream object that manages the o/p stream.

e.g. 2

```
ifstream in_file ("data");
```

ifstream ← file stream; in_file ↓ object; "data" → file name.

→ Declares the in_file as an ifstream object and attach it to the file data for reading.

Q Program to create files with constructor function.

⇒
```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
ofstream outf ("ITEM");
cout << "Enter item name:";
char name[30];
cin >> name;
outf << name;
cout << "Enter item cost";
float cost;
cin >> cost;
outf << cost << endl;
outf.close();
```

```
ifstream inf (" ITEM");
inf >> name;
inf >> cost;
cout << endl;
cout << " Item name =" << name << endl;
cout <<" Item cost =" << cost << endl;
inf. close ();
return 0;
}
```

☆ Opening files using open() :-

→ The function open() can be used to open multiple files that use the same stream object.

→ syntax -

```
file_stream_class stream_object;
stream_object. open ("filename");
```

example -

```
ofstream outfile;
outfile . open (" Data");
```

(ofstream → stream, outfile → object.)

Q. Program to illustrate working with multiple files.

Or

Program to create or open files using open().

⇒

```
#include <iostream>
#include <fstream>
using namespace std;
int main ()
{
 ofstream fout;
 fout . open (" country");
```

```
fout << "India";
fout << "USA";
fout << "South Korea";
fout.close;
fout.open ("Capital");
fout << "Delhi";
fout << "Washington";
fout << "Seoul";
fout.close();
const int N = 80;
char line [N];
ifstream fin;
fin.open ("Country");
cout << "contents of country file \n";
while (fin)
{
fin.getline (line, N);
cout << line;
}
fin.close();
fin.open ("Capital");
cout << "contents of capital file \n";
while (fin)
{
fin.getline (line, N);
cout << line;
}
fin.close();
return 0;
}
```

☆ File modes :-

→ We have used ifstream and ofstream constructors and function open() to create new files as well as to open existing files.

→ In both these methods, we used only one argument, that was the filename. These functions can take 2 arguments. :

1) filename.
2) specifying the file modes.

→ The general form of function open() with 2 arguments is :-

stream_object.open ("filename", mode);

→ The purpose of 2nd argument is for which the file is opened.

→ Filemode Parameters :-

ios :: app ⟶ append to the end of file.
ios :: ate ⟶ go to end of file on opening.
ios :: binary ⟶ binary file
ios :: in ⟶ open file for read only.
ios :: nocreate ⟶ open file if file doesn't exist.
ios :: noreplace ⟶ open fails if file already exist.
ios :: out ⟶ open file for writing only.
ios :: trunc ⟶ delete contents of file if it exists.

☆ Sequential and Random access files -

→ Sequential access files are simplest way of organising a file.

→ In sequential access files, we write the variables continuously one after other.

→ The length of each record is not fixed and can vary.

→ The advantage of this is that we do not waste any memory.

→ The disadvantage is that if you want to access the 3rd record stored in file, you have to read the first 2 records before accessing 3rd record. The reason for this is because in sequential access files the record length is not fixed and you cannot predict as to where the 3rd record might be stored.

→ The fstream class provides functions like:

1) get()
2) put()
3) write()
4) read() for writing & reading data to a file.

→ The general syntax of read() and write() methods is:

```
read((char*) &var, sizeof(var));
write((char*) &var, sizeof(var));
```

→ Reading data in any order is known as random access.

→ Random access files overcome sequential access problems since they have fixed length records.

→ The problem here is that, even if we want to store a small sized record, we still have to ~~store a small sized~~ occupy the entire fixed record length. This leads to wastage of some memory space.

→ To effectively use random access file, we make use of seekp() and seekg() functions and also tellp(), tellg(), write() and read() functions.

1) Seekp() → locate a point in the file where output will be done.

2) Seekg() → locate a point in the file from where data will be input.

3) Tellp() → get current file location where output pointer is pointing.

4) Tellg() → get current file location where input pointer is pointing.

5) write() → lower order unformatted byte write.

6) read() → lower order unformatted byte read.

Q. Program to illustrate C++ file streams.

⇒
```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
   char inform[80];
   char fname[20];
   char ch;
   cout << "Enter file name:";
   cin.get(fname, 20);
   ofstream fout(fname, ios::out);
   if(!fout)
   {
      cout << "error in creating the file!!\n";
      cout << "press any key to exit...\n";
      getch();
      exit(1);
   }
```

```
cin.get(ch);
cout << " enter a line to store in the file :\n";
cin.get (inform, 80);
fout << inform << endl;
cout << "\n entered information successfully stored..\n";
fout.close();
cout << " Press any key to exit";
return 0;
}
```

Q Program on sequential files using put() and get().

⇒

```
#include <iostream>
#include <fstream>
#include <string.h>
using namespace std;
int main()
{
char data[50], ch;
cout << " Enter a line of text ";
cin.getline (data, 50);
fstream file (" data.txt ", ios::in | ios::out);
int len = strlen (data);
for (int i=0; i< len; i++)
{
file.put (data[i]);
}
file.seekg (0);
for (int i=0; i<len; i++)
{ file.get(ch);
  cout << ch; }
file.close ();
return 0;
}
```

Q Program to perform read() and write() operations with objects using read() and write methods.

⇒

```
#include<iostream>
#include<fstream>
using namespace std;
class boys
{
    char name[20];
    int age;
    float height;
    public:
        void get()
        {
            cout << "Name";
            cin >> name;
            cout << "Age";
            cin >> age;
            cout << "Height";
            cin >> height;
        }
        void show()
        {
        cout << "name \t age \t height";
        }
};
int main()
{
    boys b[3];
    fstream out;
    out.open("boys.doc", ios::in|ios::out);
    cout << "Enter following information:";
```

```
for (int i=0; i<3; i++)
{
    b[i].get();
    out.write((char *) &b[i], sizeof(b[i]));
}
out.seekg(0);
cout << "entered information";
cout << "name age height";
for (int i=0; i<3; i++)
{
  out.read((char *) &b[i], sizeof(b[i]));
  b[i].show();
}
out.close();
return 0;
}
```

EXCEPTION HANDLING

→ C++ exception handling can automatically invoke an error handling routine when an error occurs.

→ The principle advantage of error handling is that it automates much of the error handling codes that previously had to be coded by and in any large program.

→ Exceptions are of 2 Kinds. :-

1) Synchronous exceptions
2) Asynchronous exception.

→ Synchronous : Errors such as out of range index and overflow belong to the synchronous type exception.

→ Asynchronous : The errors that are caused by events beyond the control of the program (such as Keyword interrupts).

→ The proposed exception handling mechanism in C++ is designed to handle only synchronous exceptions.

→ We often come across some peculiar (unusual or strange) problems other than logic or syntax errors. They are known as exceptions.

→ "Exceptions" are run-time anomalies (bugs or errors) or unusual conditions that a program may encounter while executing.

→ The error handling code consists of 2 segments:
1) to detect errors
2) to throw exceptions.
and other to catch the exception and to take appropriate actions.

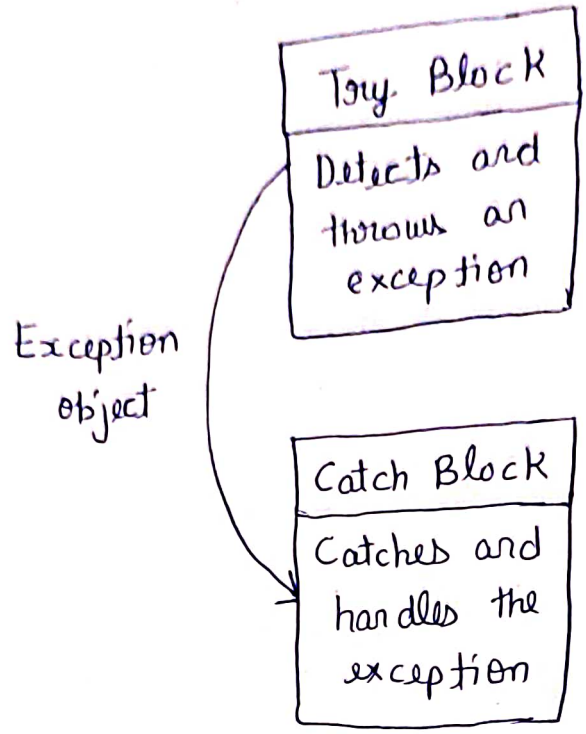


fig:- The block throwing exception

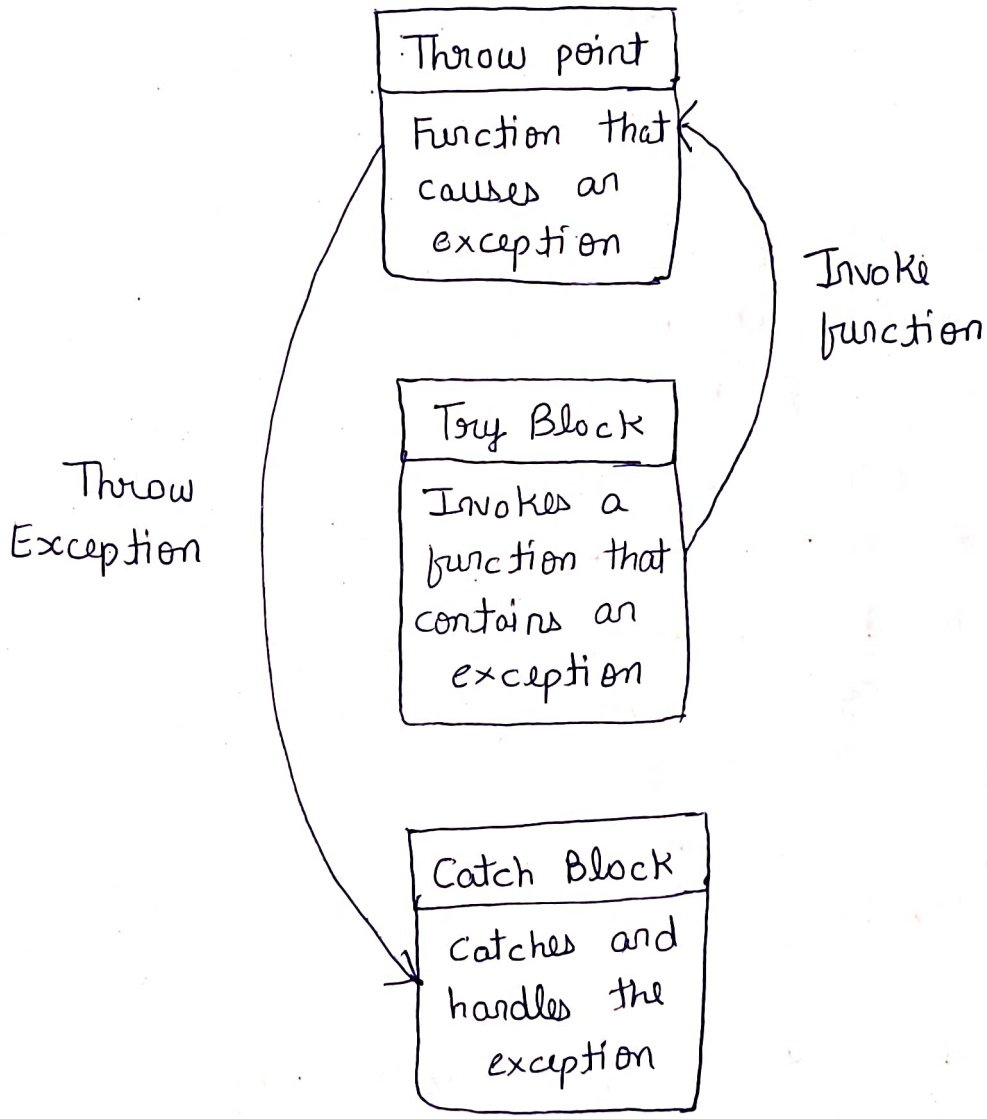


fig:- function invoked by try block throwing exception.

→ Syntax -

```
= = = = =
try
{
- - - - - -
throw exception; // Block of statements which detects
- - - - - -                & throws an exception.
}
catch (type arg) // catches exception
{
- - - - - -
= = = = = =  // Block of statements that handles the
- - - - - -                exception.
}
- - - - - - -
```

→ C++ exception handling mechanism is basically built upon 3 Keywords namely :-

1) try 2) throw 3) catch

→ The Keyword try is used to preface a block of statements which may generate exceptions. This block of statements is known as try block.

→ When an exception is detected, it is thrown using a "throw" statement in a try block.

→ A "catch block" defined by the Keyword catch, catches the exception thrown by the throw statement in the try block and handles it appropriately.

→ When a try block throws an exception, the program control leaves the try block and enters the catch statement of catch block.

→ It is possible that a program has more than one condition. In such case, we can associate more than one catch statement with a try block.

→ When an exception is thrown, the exception handlers are searched in order for an appropriate match.

☆ Throwing mechanism :-

→ When an exception that is desired to be handled is detected and is thrown using throw statement in one of the following ways -

1) throw (exception);

2) throw exception;

3) throw; // used for rethrowing an exception

→ The operand object exception may be of any type, including constants.

→ When an exception is thrown, it will be caught by the catch statement associated with the try block i.e. the control exit the current try block and is transferred to the catch block after the try block.

☆ Catching mechanism :-

→ Code for handling exception is included in catch block.

→ A catch block looks like a function definition and is of the following form :-

```
catch (type arg)
{
- - - - - -
- - - - - -
}
```

→ The type indicates the type of exception that catch block handles.

→ The parameter arg is an optional parameter name.

→ The catch statements catches an exception whose type matches with the type of catch argument.

Q. Program to illustrate and implement multiple catch statements.

⇒
```
#include <iostream>
using namespace std;
Void Test (int x)
{
  try
  {
     if (x==1) throw x;
     else
     if (x==0) throw 'x';
     else
      if (x==-1) throw 1.0;
     cout << " end of try block";
  }
 catch (char c)
{
cout << " caught a character ";
}
catch (int m)
{
cout << " caught an integer ";
}
 catch (double d)
 {
 cout << " caught a double ";
}
cout << " end of try-catch system ";
}
int main ()
{
cout << " Testing multiple catches ";
cout << " x==1 ";
Test (1);
 cout << " x==0 ";
Test (0);
```

```
cout << " x == -1 ";
Test(-1);
cout << " x == 2 ";
Test(2); // Here abnormal termination takes place
return 0;
}
```

Q Program to illustrate range of values of marks using the exceptional handling mechanism.

⇒
```
#include <iostream>
using namespace std;
int main()
{
int p, c, m, err = 0;
string name;
do
{
try {
 cout << " enter student name - "};
 cin >> name;
 cout << " enter physics marks - ";
 cin >> p;
 if (!(p >= 0 && p <= 100))
 marks entered is valid or not;
 {
 throw (p);
 transfer error to catch block;
 }
cout << " enter chemistry marks - ";
cin >> c;
if (!(c >= 0 && c <= 100))
marks entered is valid or not;
{
```

Q. Program 1 ...

```
throw (c);
transfer error to catch block;
}
cout << "enter maths marks - ";
cin >> m;
if (!(m>=0 && m<=100))
marks entered is valid or not;
{
throw (m);
transfer error to catch block;
}
err = 0;
}
catch (int e)
{
cout << "invalid marks\n";
Error;
err = 1;
}
} while (err);
}
```

☆ Rethrowing an exception -

→ If you wish to rethrow an exception from within an exception handler, you may do so by calling throw, by itself with no exception.

→ This causes the current exception to be passed on to an outer try / catch statement.

Q. Program to illustrate re-throwing :-

⇒
```
#include <iostream>
using namespace std;
void xhandler()
{
 try
 {
    throw "hello"; // throws a character
 }
 catch (const char*) // catch a character
 {
    cout << "caught character inside xhandler\n";
    throw; // rethrow character out of function.
 }
}
int main()
{
   cout << "start\n";
   try
   {
    xhandler();
   }
   catch (const char*)
   {
     cout << "caught character inside main\n";
   }
   cout << "end";
  return 0;
}
```

STRINGS

→ String is an object of std :: string class that represent sequence of characters. We can perform many operations of string.

✯ Input String :-

1) String s1 = "Hello";

2) getline() → This function is used to store a stream of characters as entered by the user.

⇒ string str;
getline (cin, str);

3) push_back() → This function is used to input a character at the end of the string.

⇒ str.push_back('s');

4) pop_back() → This function is used to delete the last character from the string.

⇒ str.pop_back();

✯ Capacity functions :-

1) capacity() → This function returns the capacity allocated to the string, which can be equal to or more than the size of the string. Additional space is allocated so that when the new characters are added to the string, the operations can be done efficiently.

2) resize() → This function changes the size of the string. It can be increased or decreased.

⇒ str.resize(13);

3) length() → This function finds the length of the string.

⇒ str.length();

4) shrink_to_fit() → This function decreases the capacity of the string and makes it equal to the minimum capacity of the string. This operation is useful to save additional memory if we are sure that no further addition of characters has to be made.

☆ Iterator Functions :-

1) begin() → This function returns an iterator to the beginning of the string.

2) end() → This function return an iterator to the end of the string.

3) rbegin() → This function returns a reverse iterator pointing at the end of the string.

4) rend() → This function returns a reverse iterator pointing at beginning of the string.

☆ Manipulating functions :-

1) copy ("char array", len, pos) → This function copies the substring in the target character array mentioned in its arguments. It takes 3 arguments, target char array, length to be copied, and starting position in the string to start copying.

2) swap() → This function swaps one string with other.

⇒ str1.swap(str2);

Function	Description
int compare(const string& str)	It is used to compare two string objects.
int length()	It is used to find the length of the string.
void swap(string& str)	It is used to swap the values of two string objects.
string substr(int pos,int n)	It creates a new string object of n characters.
int size()	It returns the length of the string in terms of bytes.
void resize(int n)	It is used to resize the length of the string up to n characters.
string& replace(int pos,int len,string& str)	It replaces portion of the string that begins at character position pos and spans len characters.
string& append(const string& str)	It adds new characters at the end of another string object.
char& at(int pos)	It is used to access an individual character at specified position pos.
int find(string& str,int pos,int n)	It is used to find the string specified in the parameter.
int find_first_of(string& str,int pos,int n)	It is used to find the first occurrence of the specified sequence.
int find_first_not_of(string& str,int pos,int n)	It is used to search the string for the first character that does not match with any of the characters specified in the string.
int find_last_of(string& str,int pos,int n)	It is used to search the string for the last character of specified sequence.
int find_last_not_of(string& str,int pos)	It searches for the last character that does not match with the specified sequence.
string& insert()	It inserts a new character before the character indicated by the position pos.
int max_size()	It finds the maximum length of the string.
void push_back(char ch)	It adds a new character ch at the end of the string.
void pop_back()	It removes a last character of the string.
string& assign()	It assigns new value to the string.
int copy(string& str)	It copies the contents of string into another.

char& back()	It returns the reference of last character.
Iterator begin()	It returns the reference of first character.
int capacity()	It returns the allocated space for the string.
const_iterator cbegin()	It points to the first element of the string.
const_iterator cend()	It points to the last element of the string.
void clear()	It removes all the elements from the string.
const_reverse_iterator crbegin()	It points to the last character of the string.
const_char* data()	It copies the characters of string into an array.
bool empty()	It checks whether the string is empty or not.
string& erase()	It removes the characters as specified.
char& front()	It returns a reference of the first character.
string& operator+=()	It appends a new character at the end of the string.
string& operator=()	It assigns a new value to the string.
char operator[](pos)	It retrieves a character at specified position pos.
int rfind()	It searches for the last occurrence of the string.
iterator end()	It references the last character of the string.
reverse_iterator rend()	It points to the first character of the string.
void shrink_to_fit()	It reduces the capacity and makes it equal to the size of the string.
char* c_str()	It returns pointer to an array that contains null terminated sequence of characters.
const_reverse_iterator crend()	It references the first character of the string.
reverse_iterator rbegin()	It reference the last character of the string.
void reserve(inr len)	It requests a change in capacity.
allocator_type get_allocator();	It returns the allocated object associated with the string.

String stream

→ A stringstream associates a string object with a stream allowing you to read from the string as if it were a stream (like cin).

→ To use stringstream, we need to include sstream header file. The stringstream class is extremely useful in parsing inputs.

→ Basic methods are :

1. clear() - to clear the stream.
2. str() – to get and set string object whose content is present in the stream.
3. operator << - Add a string to a stringstream object.
4. operator >> - Read something from the stringstream object.

e.g. ① Count numbers of words in a string.

⇒
```
#include <iostream>
#include <string>
#include <sstream>
using namespace std;
int countwords (string str)
{
 stringstream s(str);
 string word;
 int count = 0;
 while (s > word)
     count ++;
 return count;
}
int main()
{
string s = "Sharad Jha";
cout << "Number of words = " << countwords(s);
return 0;
}
```

e.g. ② Print frequencies of individual words in a string.

⇒
```
#include <iostream>
#include <string>
#include <sstream>
using namespace std;
void printFrequency (string st)
{
map < string , int > FW;
stringstream ss (st);
string word;
while (ss > word)
 FW [word] ++;
map < string , int > :: iterator m;
for (m = FW.begin (); m != FW.end (); m++)
   cout << m -> first << " -> " << m -> second << endl;
}
int main ()
{
string s = "sharad jha sharad";
printFrequency (s);
return 0;
}
```

e.g. ③ Remove spaces from a string.

⇒
```
#include <iostream>
#include <string>
#include <sstream>
using namespace std;
string removespaces (string str)
{
stringstream ss (str);
string temp;
```

```
    str = "  ";
    while (getline (ss, temp, ' '))
    {
    str = str + temp;
    }
    return str;
    }
    int main()
    {
    string s;
    getline (cin, s);
    cout << removespaces (s) << endl;
    return 0;
    }
```

e.g. ④ Converting string to numbers.

⇒

```
#include <iostream>
#include <sstream>
using namespace std;
int main()
{
string s = "12345";
stringstream obj (s);
int x = 0;
obj >> x;
cout << " Value of x = " << x;
return 0;
}
```

ARRAY

→ An array is a collection of similar data elements. These data elements have the same data type.

→ The elements of the array are stored in consecutive memory locations and are referenced by an index.

→ The index is an ordinal number which is used to identify an element of the array.

→ Syntax to declare an array :-

```
data_type name [size];
```

→ For array of size n, index ranges from 0 to n-1. index number always start with 0.

Q) How to find address of data element?

⇒ $$A[K] = BA(A) + w(K - lower\_bound)$$

e.g. Here, A is the array, K is the index of element, BA is the base address of the array, w is the size of data type in memory in bites.

- Storing Values in Array -

There are 3 ways to store values in an array.

1) Initialize the elements during declaration.
2) Input values for the elements from the Keyboard
3) Assign values to individual elements.

e.g. to input values from Keyboard.

⇒

```
int i, marks[10];
for (i = 0; i < 10, i++)
 {
 cin >> marks[i];
 }
```

e.g. code for filling an array with even numbers.

⇒
```
int i, arr[10];
for (i = 0; i < 10; i++)
arr[i] = i * 2;
```

✯ Advantages of Array :

→ Random access of elements using array index.

→ Use of fewer line of code as it creates a single array of multiple elements.

→ Traversal through the array becomes easy using a single loop.

→ Sorting becomes easy.

✯ Disadvantages of Array :

→ Allows a fixed number of elements to be entered which is decided at the time of declaration.

→ Insertion and deletion of elements can be costly since the elements are needed to be managed in accordance with the new memory allocation.

✯ Operations on Array :

→ Traversing an array.

→ Inserting an element in an array.

→ Searching an element in an array.

→ Deleting an element from an array.

→ Merging two arrays.

→ Sorting an array in ascending and descending order.

✯ Continued in data structures and algorithms