UNIT-1

classmate
Date
Page 1

Introduction to object oriented Programming

Object oriented Programming paradigm:-

-> Object oriented Programming paradigm 2 bits shape from the initial concept of a new programming approach. While the intrest in design and analysis methods came much later.

-> The first oop language was simula (simulation of real system) that was developed on 1960 by researchers at "Norwegian computing center".

-> In 1970 Alan kay and his research group at ~~Xerox~~ XEROX PARC created a personal computer named "DYNA BOOK" and the first pure oop language "small talk" used for Programming "DYNA BOOK"

-> In 1980's Grady Booch publish a paper titeled object oriented design. that mainly presented a designed for "ADA" (Programming language) In the resulting addition he extened his ideas to a complete object oriented design method.

-> In the 1990's Coad incorporated behavioral ideas to oop method.

-> The major motivating factor in the invention of object oriented approach is to remove some of the flaws encounted in the procedural approach.

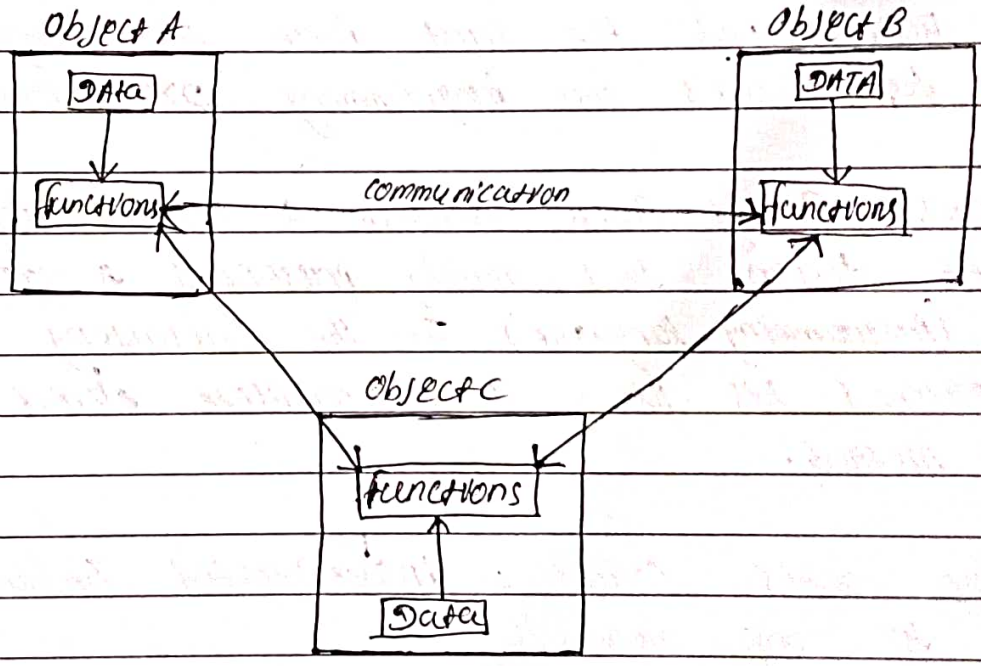
Paradigm :-

-> OOP treat data as a critical element in the program development and does not allows it flow freely around the system.

-> It ties data more closely to the functions that operates on it, and protects it from accident modification from outside functions.

-> OOP allows the composition a problem into a number of entities called objects and then builds data and functions around these objects.

-> The organization of data and function in OOP shown in the figure below.



organization of data and function in OOP

-> The data of an object can be accessed by only by the function associated with that object. However, functions of an object can accessed the functions of another object.

→ Object oriented Programming language is a Programming paradigm based open objects (having both data and method) that aims to incorporate the advantage of modularity and reusability.

→ Objects, which are usally instanies of classes are used to interact with one another to design applications and computer programs.

→ some examples of OOP language are c++, Java, smalltalk, Python, delphi, c#, Perl, ruby, PhP.

→ We define object oriented programming as an approach that provides a way of modularising Programs by creating partition memory area for both data and functions that can be used as templates for creating copies of such module on demand.

→ Thus, object is considere to be a partition area of computer memory that stores data and set of operations that can access that data.

→ since the memory partition are independent the objects can be use in a variety of different program without any modification.

→ The object oriented Programming is based on real world entities like inheritance, Polymorphism, data hidding. etc.

Benifits of oop's :

-> Programs written in oop language are easy to understand.
-> since every thing is treated as object so we can model a real world concept using oop.
-> oop provides the feature of reusability we can reuse the class that are already created without writting them again.
*-> since the parallel development of class is possible in oop concept, it results in the quick development of the complete program.
-> Programs written in oop language are easy to test, manage and maintain.
-> It is secure to development technique for program designing since data is hidden and cannot be accessed by external functions.
-> oop system can be easily upgraded from small to large systems.
-> It is easy to partition the work in project based on objects.
*-> we can build program from standard working module rather than having to start writting the code from beginning.
-> This leads to saving of development time. Because of this we can develope more projects in lesser time.

Characterstics of oop :-

-> oop is a type of program which uses objects, classes it functioning.
-> The oop is based on real world entities like inheritance, polymorphism, data hiding etc.

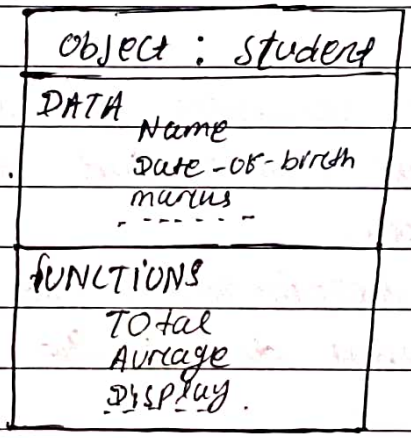
→ It aims at binding together data and functions that operate on them in a class, so that no other part (class) of the code can access this data except that function.
→ Object oriented programming focuses on both data and functions and provides an easier data accessing facility.

Basic concepts of object-oriented Programming :-

Class, object, Inheritance, Encapsulation, polymorphism, abstraction

Object :-

→ Objects are the basic runtime entities in OOP system.
→ object may be a person, a bank account, a plate, a table of data or any item that person handel.
→ object represent the real world objects closely.
→ During program execution one object communicate with other object by message passing.
→ Each object contains data and functions to manipulate those data.



→ objects are instance of class.

Class :-

→ A class is a collection of object of similar type.
→ A class is a datatype that has it's own members that is data members and member function.
→ It is the blueprint of an object oriented programming language.

→ It is the basic building blocks of object oriented programming in c++.

→ The members of a class are accessed in programming language by creating an instance of the class.

→ once a class has been defined, we can create any number of objects belonging to that class.

→ The syntax used to create an object is no different than that syntax used to create an intergers in c.

ex
If fruit has been defined as a class then the statement

fruit mango;

will create an object 'mango' belonging to the class fruit

some important properties of class :-

→ class is a user defined datatype.

→ A class contains members like data member & member function.

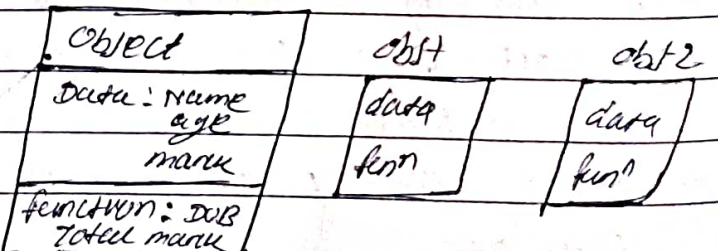
→ Data members are the variables of the class.

→ Member function are the method that are used to manipulate data members.

→ Data members define the properties of the class where as the member function define the behaviour of the class.

→ A class can have multiple object which have properties & behaviours that is common for all of them.

Representation of class :- class



Data encapsulation :-

→ The wrapping of data and functions into a single unit is known as encapsulation.

→ In c++ data encapsulation can be achive by using class.

→ By data encapsulation data is not accessable to the outside class. only those functions which are wrapped in that class can access it.

→ The function of the class provide the interface bet^n the object's data and outside object (data function) This insulation of the data is called data hiding or information hiding.

→ Data encapsulation is an extemely important aspect of OOPS. This is because it prevents hackers from modifying data that is stored in the data base.

→ If hackers are able to modify the data in the database, it could have disastrous effect of another organization.

→ encapsulation is the process by which Programmers isolate data within the object.

→ It prevents hackers from viewing the details of the data and the data structure itself.

→ Encapsulation and abstraction are two main component which improve security.

Access specifier :-

→ Access specifier define how the members of a class can be accessed.

→ In c++ there are 3 access specifier.

Public :-

→ Members are accesive from outside of the class.

Private :-

→ Members can not be accesse from outside of the class

Protected :-

Members can not be accessed from outside the class, however they can be accessed in inharitate class.

Data abstraction :-

→ It refers to the act of representing essential information to the outside world without including the background details or explanations.

→ That is to represent the needed information in Program without presenting the details.

→ Let's take one real use example :- TV
we can turen it on & off, change the chanel, adjust the volume add external component such as speakers, signal, connection of satalight, and antinas. But we don't know it's internal details, i.e we don't know how it recive signal over the air or through a cable & how it translate them and finally display them on the screen.

-> C++ classes provides great level of data abstraction.
-> They provide public method to the outside world to play with the functionality of the object and to manipulate object data.

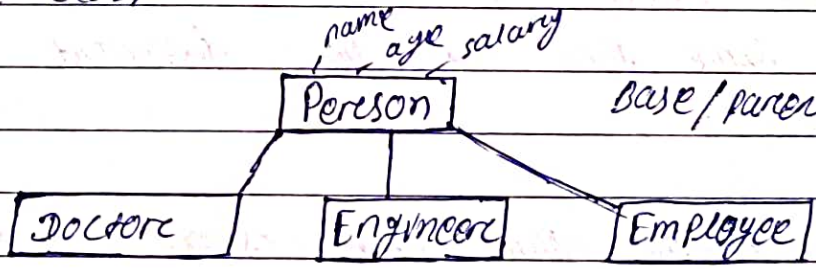
Inheritance :-

-> In OOP, the concept of inheritance provides the idea of reusability.

-> The capability of a class to derive properties & characteristics from another class is called inheritance.

-> Inheritance is the process by which objects of one class inherit the properties of objects of another class. completely or partially is known as inheritance.
-> Inheritence is a feature or a process in which objects new classes are created from the existing classes.

-> The new class created is called "derived class" or 'child class' & the existing class is known as the 'base class' or "Parent class".
-> The derived class now is said to be inherited from the base class.



Eg:-

Base / parent class / super class

child class / derived class / sub class

→ When we say derived class inherits the base class, it means, the derived class inherits all the properties of the base class without changing the properties of base class & may add new features to its own.

→ The new/derived class will have the combined feature of both classes.

eg→ [x] - 5 features
↓
[y] - 10 new features
↳ total 15 features

Types of Inheritance:

(1) Single inheritance
(2) multiple inheritance
(3) Multilevel inheritance
(4) Hierarchical inheritance
(5) Hybrid inheritance.

Polymorphism:

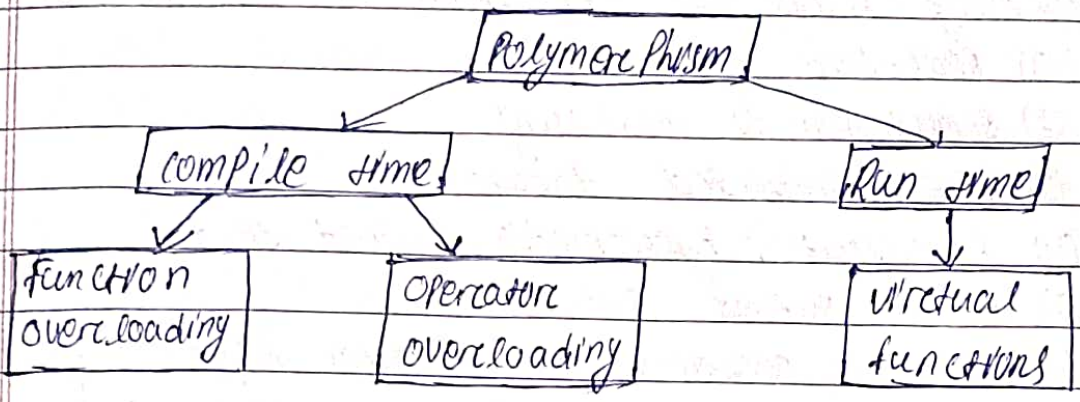
→ The word 'polymorphism' means having many forms.
→ In simple words, we can define 'polymorphism' as the ability of a message to be displayed in more than one form.

→ A real-like example of polymorphism is a person who at the same time can have different different characteristics.

→ A man at the same time is a a father, a husband, and an employee.
So the same person exhibits different behavior in different situations.
→ It is implemented using function overloading and operation overloading and virtual function.

-> This is called Polymorphism & Polymorphism is considered one of the important features of object-oriented Programming.

Types of Polymorphism



example : function overloading.

OOP c++

```
void sum (int, int);
void sum (int, float);
void sum (float, float);
void sum (int, int, int) ; 3 arguments.
```

compiler differentiate these functions by their argument data type & number of arrangement

POP-C

functions

```
void sum1 (int, int);
void sum2 (int, float);
void sum3 (float, float);
void sum4 (int, int, int);
```

Applications of OOP in c++ :

-> The most popular application of object-oriented Programming is user interface design such as windows.

-> Hundreds of windowing systems have been developed, using the OOP techniques.

→ OOP is useful for Real-business systems because these systems are much more complex & contain many more objects with complicated attributes & methods and oop can simplify these kind of complex problems.

→ The promising areas for application of oop includes:

(1) Real-time systems
(2) simulation & modelling
(3) object-oriented database
(4) Hypertext, hypermedia, expert text
(5) AI & expert systems
(6) Neural networks and parallel programming
(7) Decision support & office automation systems
(8) CIM/CAD/CAM system

Object oriented languages:-

→ Programming becomes critical & may generate confusion when the program grow large.

→ A language that is specially designed to support the oop concepts makes it easier to implement them.

→ The languages should support several of the oop concepts to claim that they are object-oriented.

→ Depending upon the features they support, they can be classified into the following two categories.

(i) Object based programming language

(ii) object oriented programming language

→ Object-based Programming is the style of programming that supports encapsulation & object-identity.
It also supports: Data hiding & access mechanisms, Automatic initialization and clear-up of objects & operator overloading.

→ object-oriented Programming is the style of Programming that supports (i) encapsulation
(ii) Object identity
(iii) Data hiding
(iv) Automatic initialization of objects
(v) Operator overloading.

→ Languages that supports Programming with objects are said to be object-based Programming languages.
→ They do not support inheritance and dynamic binding.
→ Ada is a typical object-based Programming language.

→ Object-oriented Programming incorporates all of object-based Programming features along with two additional features namely, inheritance & dynamic binding.
i.e
Object-based Programming + inheritance + dynamic binding
= object oriented Programming.

Benifits of OOPs :-

→ Abstraction technique are used to hide the unnecessary details and focus is only on the relevent part.

→ OOP provides the feature of reusability. We can reuse the class that are already created without writting them again.

→ Since everything is treated as object so we can model a real world concept using oop.

→ Programs written in oop language are easy to understand, maintain, modify, and testing.

→ OOP is more suitable for large project. Because it is easy to partition the work in project based object.

→ software complexity can be easily managed.

→ OOP system can be easily upgraded from small to large system.

→ OOP techniques are more reliable.

→ It provides features like, inheritance, polymorphism, Data abstraction, etc.

Introduction to C++

classmate
Date
Page 15

-> C++ is the extension of C.
-> It developed by Bjarne Stroustroup in 1979 AT&T Bell Laboratories. AT&T -> American Telephone and telegraph.
-> C++ is an OOP language & it also known as structural Programming language.
-> C++ runs the variety of Platforems, such as windows, Mac os, & the various version of unix.
-> C++ is most widely used Programmicy language in application & system Programmiry.
-> It is a middle-level language, as it encapsulates both high and low level language feature.
-> Allmost all c progroams are also C++ progroams.
-> The most important facilities that c++ adds on to C are classes, Inheritance, function overloadiny and operatore overloadiny
-> The object oriented features in C++ allow programmers to build large progroams with clarity, extensibility and easy of maintenance.
-> By using Turebo c++ software, we can woride and emeude a C++ Program easily. we can also use different types of IDE to start a C++ Program like, Eclipse, visual studio.

Example of a simple C++ Progroam :-

```
#include<iostream>
void main()
{
 cout<<"Hello world! \n";
 cout<<"I am subhasis";
 return 0;
}
```

-> Output :-

Hello world!
I am subhasis.

NOTE:- → Two \n characters after each other will create a blank line.

→ Another way to insert a new line, is with the endl manipulator. cout << "Hello world!" << endl;

→ Both \n and endl are used to breake lines.

Difference between C & C++ :-

C	C++
① C is a POP language.	① C++ is a OOP language.
② In 'C' language we use scanf() for input and printf() for output.	② In C++ we use cin>> for input and cout<< for output.
③ In 'C' Polymorphism is not possible.	③ In C++ Polymorphism is possible & it is an important feature.
④ Operator & function is not possible in 'C'.	④ Operator and function are possible in C++.
⑤ Mapping between data & functions difficult and complicated.	⑤ Mapping between data & function is possible using objects.
⑥ Inheritance is not possible in C	⑥ C++ supports inheritance.
⑦ file extention in C is .c.	⑦ file extention in C++ is .cpp.
⑧ For the development of code C supports Procedural Programming language	⑧ C++ is known as hybrid language because C++ supports both POP & OOP language.
⑨ Data and function are separated in C because it is a Procedural oriented language.	⑨ Data and function are encapsulated in form of an object.
→ It is less secure.	→ It is highly secure.
→ It takes less memory.	→ It takes more memory than pop.

→ C contains 32 keywords	C++ contains 63 keywords
→ POP follows top down approach	→ OOP follows bottom up approach.
(10) Build in data types are supported in C.	(10) Build in & user defined datatypes are supported in C++.
(11) Instead of focous on data 'C' focous on methods & process.	(11) C++ focuses on data instead of focousing on method or procedure.
(12) In C. we use standard input output header file i.e <stdio.h>	(12) In C++ we use input output stream header file in C++ i.e <iostream>
(13) There is no any acess specifier.	(13) It have access specifiers like Public, Private, Protect

TOKENS :-

→ The smallest individual units in a program are known as tokens. C++ has the following tokens :-

→ (1) keywords (2) Identifiers (3) constants (4) strings (5) operators

⇒ keywords :-

→ keywords are explicitly reserved identifiers and can not be used as names for the program variables or other user defined program elements.

→ keywords has a predefined meaning which can not be changed. eg→ int, float, char, auto, case, break, new,

⇒ Identifieres :-

→ Identifiers are the fundamental requirement of any language.

→ Each language has it's own rules for

→ Identifiers are created to give a unique name to an entity to identify it during the execution of the program.

→ Identifi__ are the names given to variables, constants, functions and user-defined data types.

→ Identifiers must be unique.

⇒ Rules for naming Identifiers :-

⇒ A valid identifier can have letters (both uppercase and lowercase letteres), dygits and underscores.

→ The first letter of an identifier should be either a letter or an underscore. It cannot start with a dygit.

-> keywords are not use as a variables name.
-> Identifiers are case sensitive (myVar and myvar are different variables).
-> commas or blank spaces are not allowed within an identifier..

Constants / Literals

-> Constants refers to fixed value that the program modify during its execution.
-> These fixed values are also called as literals.
-> constants can be of any of the basic data types like an integere constant, a floating-point constant, a charecter constant or a strung literal.
-> Constants are treated just like regular variables except that there values cannot be modified after there declaration.

Types of constants :-

1- Integere constants

2- floating- point constant

3- charecter constant

4- strung constant

5- Backslash charecter constants
Eg-> \n \t \b

How to define constants in c++

→ There is one simple way in c++ to define constants: using const keywords.

Syntax :- const <data type> <variable> = value;

Eg :- const int LENGTH = 10;

Program :-

```
#include<iostream> using namespace std;
void main()
{
const int LENGTH = 10;
const int WIDTH = 7;
int area;
area = LENGTH * WIDTH;
cout << "Area of rectangle is" << area << "cm2";
}
```

(11) Strings :-

→ strings are used for storing text.

→ A string variable contains a collection of characters surrounded by double quotes.

→ String is a sequence of characters that is treated as a single data item and terminated by null character '\0'.

→ c++ language does not support strings as a data type.

→ A string is actually one-dimensional array of character in c++ language

→ Character array is a datatype that is used to store multiple character values by using a single variable name.

Declaration & Initialization of string variable.

→ There are different ways to declare and initialize a string variable.

```
char name[5] = {'B', 'i', 's', 'u', '\0'};
char name[] = {'M', 'U', 'N', 'U', '\0'};
char name[5] = "MUNU";
char name[] = "MUNU";
```

→ If we want to declare a string variable using "string" keyword, then we have to use an additional header file, i.e <string> library in program.
"TURBO C++ does not support this."

String input & output :-

→ We can use cin statement to read a string.

→ The cin statement reads the sequence of character until it encounters white space (space, newline, tab, etc)

→ To read a string with white space or to read a single line of text we can use ~~getline()~~ getline() function with cin statement and string variable as argument.

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

→ To print a string we can use cout statement.

String concatenation :-

→ The '+' operatore can be used between strings to add them togethere to make a new string. This is caued concatenation.

eg:- string firstName = "subhasis";
string lastName = "sahoo";
string fullname = firstName + lastName;
cout << fullname;

Program :-

```
#include <iostream> using namespace std;
void main()
{
char str[1000];
cout << "Enter a string value :";
cin.getline (str, 1000);
cout << " string value is :" << str;
}
```

output :- Enter a string value : subhasis sahoo
string value is : subhasis sahoo

Operators in c++

Operators in c++ :-

→ Operators are used to perform operations on variables and values.

→ Operator are the symbols that are used to perform some mathematical and logical operations.

→ The values or variables on which the operation performs the special task are known as operands.

Types of operators :-

1- Arithmatic operator
2- Relational operator
3- Logical operator
4- Bitwise operator
5- Assignment operator
6- Conditional operator
7- Increment & decrement operator.

1. Arithmatic operator :-

→ Arithmatic operator are used to perform common mathematical operations.

→ The arithmatic operators are +, -, *, / and %.

Operator	name	Description	Eg
+	Addition	Add two values	x+y
-	Subtraction	Subtract one value from another	x-y
*	Multiplication	Multiplies two values	x*y
/	Division	Divide one value from another	x/y
%	Modulus	Return the division remainder	x%y

2. Relational operators :-

→ Relational operators are also known as comparison operators.

→ These operators are used to compare two operands to one another.

→ The relational operators are ==, !=, <, >, <=, >=.

Operator	Name	Description	Example
==	Equal to	check if the values of two operand are equal or not. If yes, then the condition becomes true	x == y
!=	Not equal to	check if the values of two operands are equal or not if the values are not equal, then the condition becomes true.	x != y
>	Greater than	checks if the value of left operand is greater than the value of right operand. if yes, then condition becomes true.	x > y
<	Less than	checks if the value of left operand is less than the value of right operand. if yes, then the condition becomes true.	x < y

Operator	Name	Description	example
>=	Greater than or equal to	checks if the value of left operand is greater than or equal to the value of right operand. if yes, then condition becomes true.	x >= y
<=	Less than or equal to	checks if the value of left operand is less than or equal to the value of right operand. if yes, then the condition becomes true	x <= y

3. Logical operator :-

→ These operators are used to make a decision on two conditions.

→ Logical operators are typically used with boolean values and they also return boolean values.

→ logical operators are used to perform logical operation. The operation which is used to represent logical values i.e, true or false.

→ Logical operators sometimes are also called as boolean operator.

→ The logical operators are &&, ||, !

Operator	Name	Description	example
&&	Logical AND	Return true if both the statement are true.	X>5 && X<=10
||	Logical OR	Return true if any one or the statement is true or both the statements are true.	(x>5 || x<=10)
!	Logical NOT	Reverse the result. Return true if the result is false	!(x>5 && x<=10)

4. Bitwise operators :-

-> A bitwise operator works on bits and perform bit-by-bit operation.

-> These operators manipulate the values of the data at bit level.

-> for example, if A = 60 and B = 13.
Now in binary format A = 0011 1100
B = 0000 1101

operator	Name	example	Result
&	Bitwise AND	A&B	0000 1100 (i.e 12)
|	Bitwise OR	A|B	0011 1101 (i.e 61)
~	Bitwise NOT Complement	~A	1100 0011 (i.e -61)
^	Bitwise XOR	A^B	0011 0001 (i.e 49)

It returns the value 1 when odd number of input is 1.

<<	Left shift	A<<2	1111 0000	(i.e 240)
>>	Right shift	A>>2	0000 1111	(i.e 15)

5. Assignment operator :-

→ Assignment operators are used to assign values to variables.

operator	name	
=	Assignment operator	C = A+B Assign value of A+B to C.
+=	Add & assignment operator	C+=A is equivalent to C = C+A
-=	Subtract & assignment operator	C-=A is equivalent to C = C-A
*=	Multiply & assignment operator	C*=A is equivalent to C = C*A
/=	Divide & assignment operator	C/=A is equivalent to C = C/A
%=	Modulus & assignment operator	C%=A is equivalent to C = C%A

6. Conditional operators :-

→ The conditional operator is also known as the ternary operator.

→ This operator consist of three operands (one expression and two statements).

→ The syntax of this operator is :-

(expression) ? statement 1 : statement 2 ;

firstly 'expression' is tested, if it is true the statement 1 will be returned. otherwise, statement 2 will be returned.

example :- a = 3;
b = 4;
c = (a<b) ? a : b ;
= (3<4) ? 3 : 4 ;
or c = 3

7. Increment & Decrement operator :-

→ The increment operator is used to increase some value by 1.

→ The decrement operator is used to decrease some value by 1.

→ This operator is known as unary operator.

→ Increment operator is represented as '++' and decrement operator is represented as '--'.

Expression	Description	Example	Result
++A	Add 1 to variable before use	int A=2, B; B= ++A	A = 3 B = 3
A++	Add 1 to variable after use	int A=2, B B=A++;	A = 3 B = 2
--A	substract 1 from a variable before use	int A=2, B; B= --A;	A = 1 B = 1
A--	substract 1 from a variable after use	int A=2, B; B= A--;	A = 1 B = 2

Note

1. Unary operator :-

→ These are the operators which work on single operands.
→ Increment and decrement operators are the examples of unary operator.

2. Binary operator :-

→ These are the operators which work on two operands.
→ Arithmatic operators, Relational operators, etc. are the examples of binary operator.

3. Ternary operator :-

→ These are the operators which work on three operands.
→ conditional operator (?:) is an example of ternary operator.

(5)

Variables in c++ :

→ A variable is a container that is used to hold some value in memory location, which can be used during the time of execution of a program.

→ In Programming, a variable is a container (storage area) that is used to hold data.

→ ~~A variable is a container that is used to hold some value in memory location, which can be used during the time of execution of a program.~~

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

→ eg:- int length = 10;
Here, length is a variable of int type.
Here, the variable is assigned an integer value 10.

→ The value of a variable can be changed, hence the name variable.

```
int length = 10;
// some code
length = 20;  // hence the value changed.
```

In c++, there are different types of variables (defined with different keywords), for example.

→ int - stories integers (whole numbers), without decimals, such as 123, -123.

→ double - stores floating point numbers, with decimals, such as 19.99 or -19.99

-> char :- Stores single characters, such as 'a' or 'B'. char values are surrounded by single quotes.
-> string :- Stores text, such as "Hello world". String values are surrounded by double quets.
-> bool :- Stores values with two states ; true or false.

Rules fore Naming a variable :-

-> A variable name can only have letter (both upper-case and lowercase letters), digits and underscore.

-> The first letter of a variable should be either a letter or an underscore.

-> Variable name must be meaningful, short and easy to read.

-> C++ is a strongly typed language. This means that the variable type cannot be change once it is declared.
eg-> int length = 10;
length = 10.5 ; // error
because int cannot store decimal number
double length ; // error
because length is already declared as int.

Declaration of variable
-> A variable is declared by using its data type and variable name.
syntax :- <data type> <variable name> ;
eg->
int length ;

Initialization of a variable :-

1. Static Initialization :-

Here, the variable is assigned a value in advance in the program. This variable then acts as a constant.

eg:
```
int a;
a = 5;
```

2. Dynamic Initialization :-

Here, the variable is assigned a value at the run-time. The value of the variable can be altered every time the program is being run.

We can initilize a variable in different ways :-

1- Declaring the variable and then initializing it.
```
int a;
a = 5;
```
2- Declaring and initializing the variable together.
```
int a = 5;
```
3- We can declare more than one variable of same data type at a time by using a separator (comma ,).
```
int a, b, c;
OR
int a = 10, b = 20, c = 30;
```
How to display the value of a variable.

→ The cout object is used together with the << operator to display value of the variables.

→ To combine both text and a variable, separate them with the << operator.

Data types in c++

Program :-

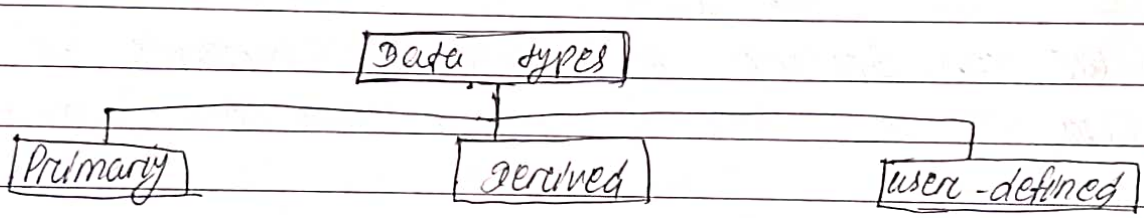
```
#include <iostream>
void main()
{
int my-Age =23;
cout << "I am" << my-Age << "Years old";
}
```

Data types in c++ :-

→ Data types are the keywords that describe the type of data that a variable can store, the type of data that a function can return.

→ There are three types of data type :

1. Pre-defined Data types] Build in data type
2. Derived Datatypes
4. User-defined Data types.] Reference data type.



Integer	function	class
character	Array	structure
Boolean	Pointer	Union
floating point	Reference	Enum
Doubble floating point		Typedef
void		
wide character		

1. Primitive datatype :

-> These are the basic data types that are not derived from any other data types.

-> These are pre-defined and are also known as built-in-data types.

Types of Primitive datatype

1. Integer data type :

-> The keyword used for integer data types is int. Integer typically require 4 bytes of memory space and range from -2147483648 to 2147483647.

-> It stores all positive, negative integer number without decimal.

ex-> `int a = 2333;`

2. Character data type :-

-> Character data type is used for storing characters. The keyword used for the character data type is char. Characters typically require 1 byte of memory space and range from -128 to 127 or 0 to 255.

ex-> `char ch = 'a';`

3. Boolean data type :-

-> Boolean data type is used for storing boolean or logical values. A Boolean variable can store either true or false. The keyword used for the boolean data type is **bool**.

4. floating point data type:

floating point data type is used for storing decimal values. The keyword used for the floating point data type is float. float variables typically require 4 bytes of memory space. It has 6 digit of precision.

5. Double floating point:

Double floating point data type is used for storing double-precision floating-point values or decimal values. The keyword used for double floating-point data type is double. Double variable typically require 8 bytes of memory space. It has 15 digit of precision.

6. void data type:

→ void means without any value. void data type represents a valueless entity. A void datatype is used for those function which does not return a value. The keyword use for void datatype is void.

→ The only object that can be declared with the type specifier void is a pointer.

7. wide character:

It is also a character data type but this data type has a size greater than the normal 1 byte /8-bit data type. It is generally 2 or 4 bytes long. It is represented by wchar_t.

Note: sizeof() operatore:

It is used to find the numbere of bytes occupied by a variable / data type in computere memory.

enample:

```
int m, n[50];
cout << sizeof(m); // returen 4 which is the numbere of
                      bytes occupied by the integere variable "m".
cout<< sizeof(n); // returns 200 which is the numbere of
                      bytes occupied by the integere array "n".
```

Program:

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

output:

```
size of chare : 1
size of int : 4
size of long : 8
size of float : 4
size of double : 8
```

Referenced data type:

→ These are the data types which are constructed by using primitive data types, are called as referenced data types.

→ These are also called as non-primitive data types.

→ These data types are defined by the user.

There are two types of referenced data types.

1. Derived data type.
2. User-defined data type.

1. Derived data type:

→ Those data types whose variables allow us to store multiple values of same type are known as derived data types.

→ But they never allow storing multiple values of different data types.

→ Eg→ Array, function, pointer, reference

function:-

A function is a block of code or program that is defined to perform a specific well-defined task. A function is generally defined to save the user from writing the same line again and again for the same input. All the lines of code are put together inside a single function and this can be called anywhere required.

Program :-

```
#include <iostream>
using namespace std;
// declare function
void print()
{
cout<< "Hello world!";
}
int main() {
    print(); // calling the function
return 0;
}
```

Output :
Hello world!

Array :

-> Array is a derived data type.
-> An array is the collection of elements having same datatype and grouped together under a single name.
-> It stores elements in continuous memory location.
-> We can access any element using index number.

Program :-

```
#include <iostream>
using namespace std;
int main() {
    int arr[10];
    arr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0}
    cout << "5th element = " << arr[5] << endl;
}
```

O/P : 5th element = 6

Pointer :

-> The Pointer is a variable which stores the address of another variable.
-> The pointer can be declared using * (asterisk symbol)

Syntax - data type *variable name;

```
int *a;
```

4. Reference:

→ When a variable is declared as reference, it becomes an alternative name for an existing variable.

→ A variable can be declared as reference by putting '&' in the declaration.

Program:

```
#include<iostream>
using namespace std;
int main(){
    int n=10;
    int& ref=n;
    // Reference derived type
    //ref is a reference to n.
    ref=20;
    cout<<"n="<<n<<endl;
    // value of n is now change to 20
    n=30;
    cout<<"ref="<<ref<<endl;
    return 0;
}
```

O/p n=20
ref=30

User defined data type -

→ Those data types whose variable allow us to store multiple values of same or different data type or both, are known as user-defined data type.

→ User defined data types are developed by programmers.

→ eg→ class, structure, union, Enumeration, Typedef etc.

class:

-> A class is a collection of object. which holds its own data members and member functions. which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

-> The keyword used to declare a class is class.

Syntax:
```
class <class name>
{
    Access specifier : // can be private, public, protected
    Data members ;  // variables to be used
    Member functions() {  // method to acess data members
    statement
    }
};                          // class ends with a semicolon
```

Structure:

-> structure is the collection of variable having same or different datatypes grouped together under a single name. To define a structure the struct keyword is used.

syntax :-
```
struct <structure name>
{ data type member 1;
  data type member 2;
  .....
};
```

example:
```
struct person
{
char name[50];
int Roll no. ;
float salary ;
};
```

Union:

→ Like structure union is a used defined data type. In union all members share the same memory location.

→ for example: both x and y share the same location. if we change x, we can see the changes being reflected in y.

→ we use union keyword to declare an union.

Enumeration:

→ It is mainly used to assign name to integral constants, the names make a program easy to read and maintain.

→ By default, the enumerators are assigned integral value starting with 0 for the first enumerator, 1 for the 2nd enumurator and so on.

→ The keyword use for enumerated data type is enum.

→ In c++ an enum defined within a class or structure is local to that class or structure only.

Program:

```
#include <iostream>
using namespace std;
enum week { mon,
            tue,
            wed,
            thur,
            fri,
            sat,
            sun };
```

```
int main() {
    enum week day;
    day = wed;                    output : 2
    cout << day;
    return 0;
}
```

Typedef :

→ C++ allows to define explicitly new data type names by using the keyword typedef. Using typedef does not actually create a new data class, rather it defines a name for an existing type.

→ This can increase the portability (the ability of a program to be used across different types of machines; i.e, mini, mainframe, micro, etc ; without much changes into the code) of a program as only the typedef statements would have to be changed.

Syntax :

```
typedef  data type <name>;
```

Program :

```
#include <iostream>
using namespace std;
typedef unsigned char BYTE;
int main()
{
  BYTE b1, b2;
      b2 = 'c';
  cout << "b1" << b1;
  return 0;
}
```

output : c

Structure of C++ Program

Basic structure of c++ program is described below:-

1. Documentation
2. Link section
3. Defination section
4. Global declaration section
5. Main function
6. Sub Program section

1. Documentation:

The documentation section is the part of the program where the programmer gives the details associated with the program. By using comment.

2. Link section:

→ This part of the code is used to declare all the header files that will be used in the program.

→ These header files provides instructions to the compiler to link function from the library function.

3: Defination section:

In this section, we define different constants. The keyword define, is used in this part.

4. Global declaration section:

In this section, all the global variable, user-defined data types, class, etc. that we will use in a program are declared as well as the user-defined functions are also declared in this section of the program.

5. Main function:

Every c++ program must have a main() function which is the starting point of the program execution.

6: Sub-Program section:

All the user-defined functions are defined in this section of the program.

```
//Documentation
/* write a program to find the area of a circle */
//Link section
#include <iostream.h>
Using namespace std;
//Defination section
#define PI 3.14
//Global declaration section
 void circle();
 float rc;
//Main function
 void main()
 {
  cout<<"Enter the radius of circle:";
  cin>>rc;
  circle();
 }
// sub program section
 void circle()
 {
    float area;
    area = PI*rc*rc;
    cout<<"Area of a circle is:"<<area;
 }
```

Note:-

-> We use cin>> for input and cout<< for output.
-> Like c prog. all statement ends with a ';'.
-> Namespace is a new concept in c++. This defines a scope for the identifiers that are used in a program. For using the identifiers defined in the namespace scope we must include the

using directive like

using namespace std;

Here std is the namespace where c++ standard class libraries are defined.

→ c++ implementations use extensions such as .c, .C, .cc .cpp and .cxx

Turbo c++ use .cpp, Zortech c++ uses .cxx while UNIX AT&T versions uses .c (capital c) and .cc (capital cc)

Comments in c++

→ Comments can be used to explain the code, and to make it more readable.

→ It can also be used to prevent execution when testing alternative code.

→ comment can be singled-lined or multi-lined.

→ single-line comments start with two forwaard slashes (//).

→ Any text between // and the end of the line is ignored by the compiler will (not be executed)

⇒ Multi-line comments start with /* and ends with */.

→ Any text between /* and */ will be ignored by the compiler.

Control structure in c++

→ The statement in c++, that is used to control the flow of execution in a program is known as control structure.

→ The control structure in c++ are used to combine individual instruction into a single logical unit.

→ The logical unit has one entry point & one exit point.

→ A control structure is used to determine that a statement in a program do be executed once, many times or not at all.

Types of control structure :

(I) sequence structure
(II) selection / decision making structure.
(III) Loop structure (Iteration or Repetition)

(I) sequence structure :

→ Executes the statements in the same order in which they are written in the Basic Program.

Program :-

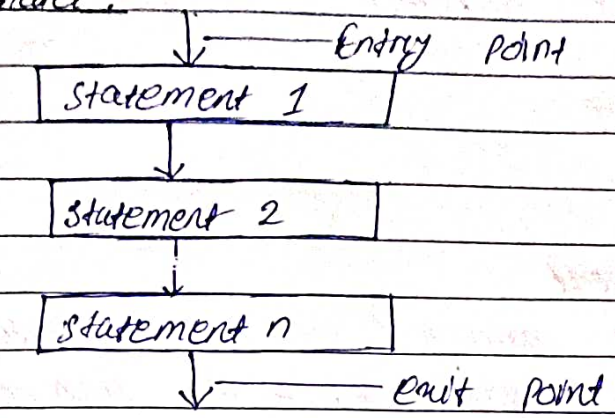
```
#include<iostream>
using namespace std;
void main()
{
statement 1;
statement 2;
.....
statement n;
}
```

```
#include<iostream>
using namespace std;
void main()
{
int a, b, result;
a=10;
b=20;
result = a+b;
cout<< result;
return 0;
}
```

output

result = 30

flow chart :



(II) Selection control structure :

→ Decision making structures have one or more conditions to be tested by the program, along with a statement or group of statements that are to be executed if the condition is true, otherwise other statement to be executed if the condition is false.

→ It is also called decision-making or conditional structure.

Different types of selection structure in C++

1. If Statement
2. If..... Else statement
3. Nested If... Else statement.
4. Else if ladder.
5. switch statement.

(III) Looping Structure :

→ In general, statements are executed sequentially in a Program.

→ When we need to execute a block of statements several number of times. At that time, we use Looping structures to execute a statement (block of statement) multiple number of times.

Types of looping statement

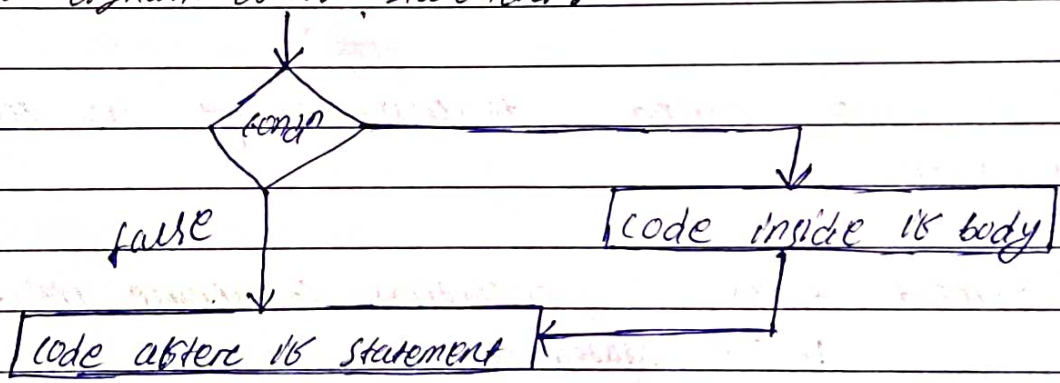
1. while loop
2. DO-- while loop
3. for loop

i) Simple If Statement

It is a decision making / conditional statement which executes a single block of statement if the specified condition is true, otherwise the given set of statements are ignored.

Syntax:- if (condition)
{
// statement will execute if the condition is true
}

flow digram of if statement:-



example:- #include <iostream>
using namespace std;
int main() {
int x=45, y=34;
if (x>y)
cout << x << "is greater than" << y;
}

output:- 45 is greater than 34

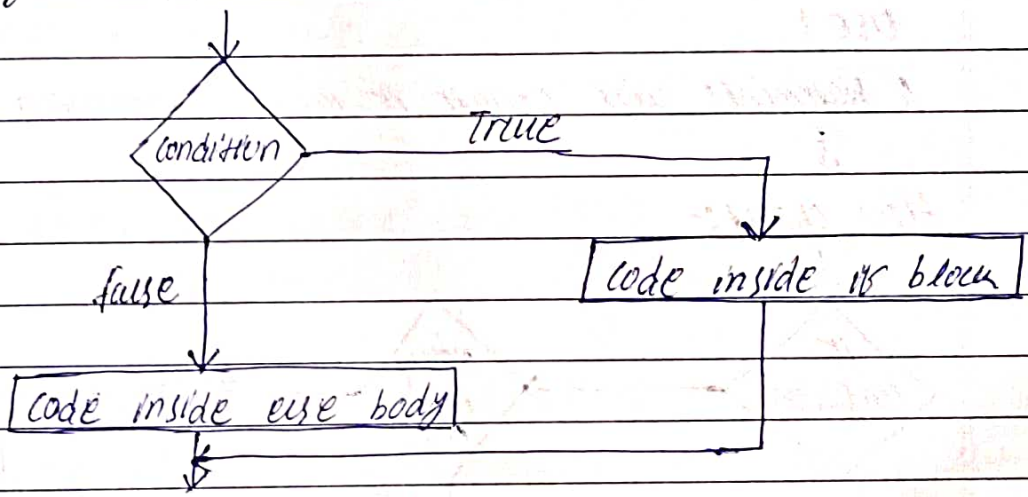
(11) if_else statement:-

-> It executes a single block of statement if the specified condition is true, otherwise the else block is executed.

Syntax:-

```
if (condition)
{
 // statements will execute if the condition is true.
}
else
{
 // statements will execute if the condition is false.
}
```

flow diagram of if_else statement:-



example:-

```
#include <iostream>
using namespace std;
void main()
{
 int x=45, y=34;
 if (x>y)
 {
 cout<<x<<" is greatere than"<<y;
 }
else{
 cout<<y<<" is greatere than"<<x;
}
```

output:-

45 is greatere than 34

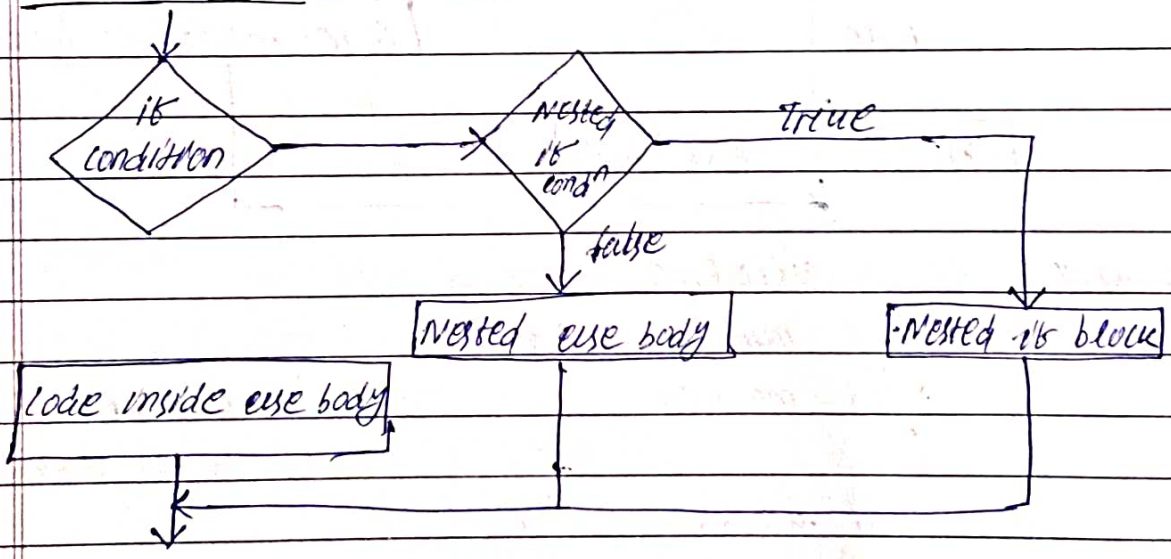
(iii) Nested if-else statement :-

when an if-else statement is present inside the body of another "if-else" statement, then this is called nested if---else statement.

syntax :-

```
if (condition)
{
  if (condition){
  // statement will execute if both the condition are true.
  }
  else{
  // statement will execute if the nested if condition is false
    }
}
else{
// statements will execute if first if condition if false.
  }
```

flow chart :-



example :-

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

```
int n;
cout<<"Enter the value of n:";
cin>>n;
if(n%2==0)
 {
  if(n%3==0)
    {
     cout<<"Divisible by 6";
    }
else{
   cout<<"Not divisible by 6";
  }
}
else{
    cout<<"Not divisible by 6";
   }
 }
```

Output:- Enter the value of n: 12
Divisible by 6

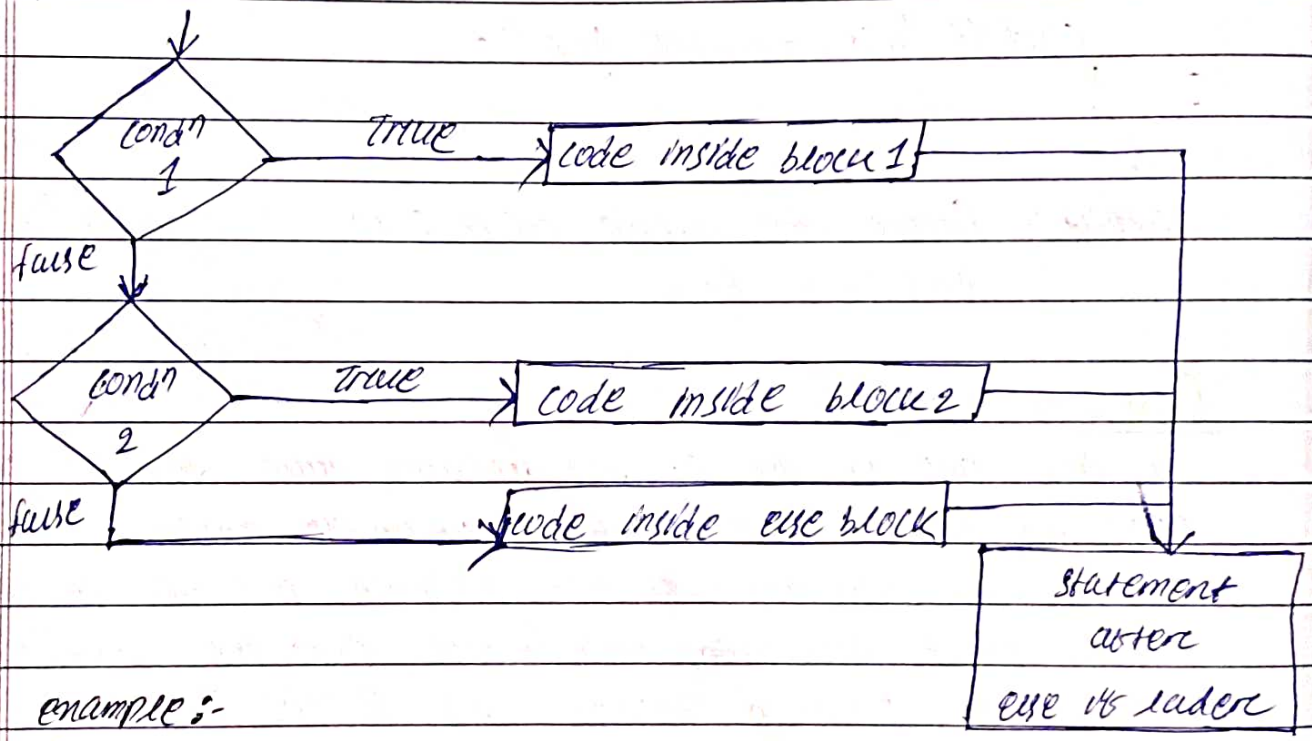
Else_if ladder:

In this case if the condition returns true than the statements associated with it are executed otherwise another if condition is checked. If this condition returns true then the statement associated with it are executed otherwise another if condition checked and so on. If none of the condition are true than the statements under last else statement are executed.

Syntax :-

```
if (condition 1)
 {
  //block 1 ;
 }
else if (condition 2)
 {
  //block 2;
 }
 :
else {
  //default block
  }
```

flow chart :-



example :-

```
#include<iostream>
using namespace std;
int main()
{
 int n1, n2;
 cout<< " Enter the value of n1 :";
```

```
cin>>n1;
cout<< "Enter the value of n2:";
cin>> n2;
if (n1<n2){
   cout<<n1<<" is the smallest no.";
        }
else if (n2>n1){
        cout<<n2<<" is the smallest no.";
             }
else{
    cout<<"n1 is same as n2";
   }
  }
```

Switch Statement :-

-> It is a selection control statement that is used to select a block of statement to be executed from multiple blocks based on the values of a variable which will be inserted by the user during the time of execution of a program.

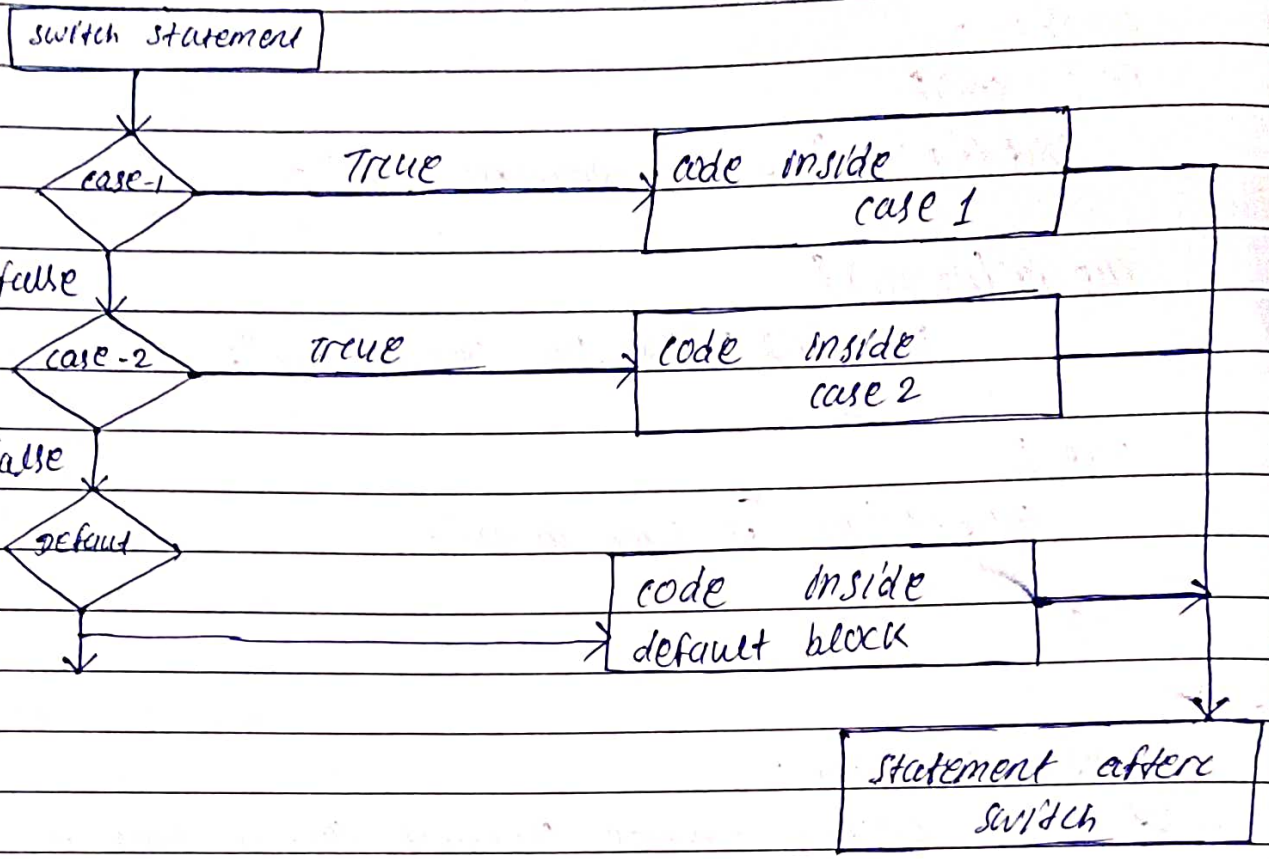
-> A switch statements allows a variable to be tested for equality against a list of values.

-> Each value is called a case, and the variable used on switch is check for each case.

Syntax:

```
switch (variable)
{
   case <value1>:
      <statement>;
      break;
   case <value2>:
      <statement>;
   case <value n>:
      <statement>;
      break;
   default:
      <statement>;
}
```

flow chart:



Note:

-> The variable used in a switch statement can only be of integer and char data type.

-> we can have any number of case statements within a switch: Each case followed by the value to be compared to and a colon.

-> The value for a case must be the same data type as the variable in the switch and it must be a constant or a literal.

-> When the value of the variable (which is used in switch) is equal to a case, the statement inside that case will execute until a break statement is reached.

-> When a break statement is reached, the switch terminates, and the flow of control jumps to the next line after the switch statement.

-> Not every case needs to contain a break.

-> A switch ~~case~~ statement can have an optional default case, which must appear at the end of the switch. If no case matches then the default part is executed.

example:-

```
#include <iostream>
using namespace std;
int main()
{
int day;
cout<<"Enter the day numbere (1-7):";
cin>>day;
switch (day)
    {
 case 1:
       cout<<"sunday";
         break;
 case 2:
       cout<<"Monday";
        break;
 case 3:
       cout<<"Tues day";
        break;
 case 4:
       cout<<"wednesday";
        break;
 case 5:
      cout<<"Thursday";
       break;
 case 6:
      cout<<"friday";
         break;
```

```
default:
 cout<<"Enter day no. between 1-7";
 }
}
```

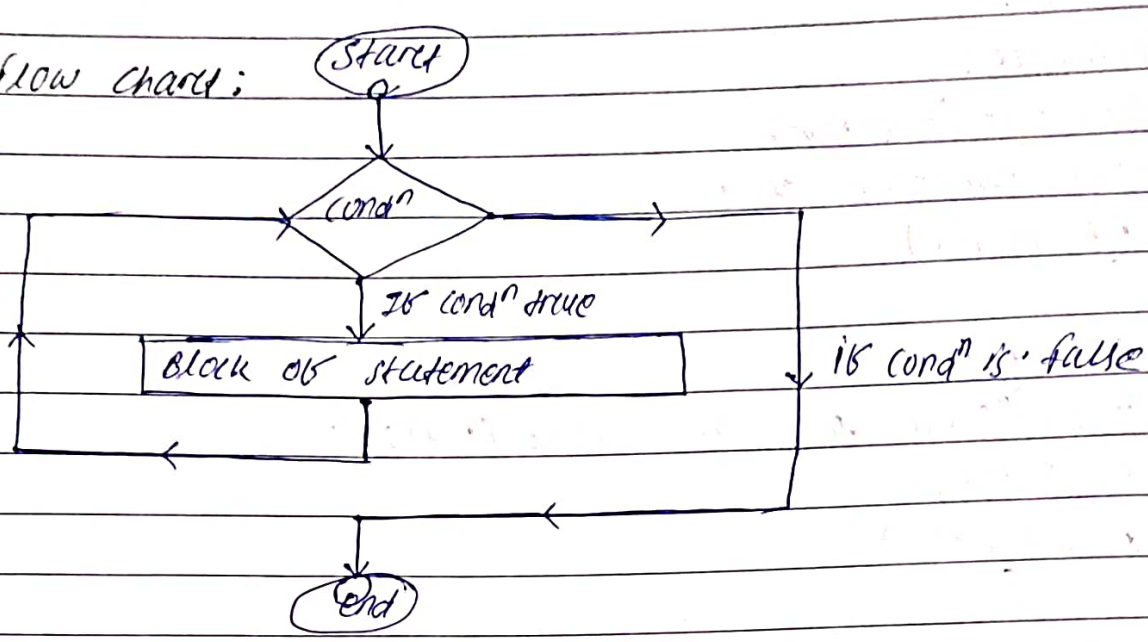
Output:

Enter the day number(1-7): 6
friday

Looping statement:-

While loop:-
It is a looping statement that is used to execute a block of statement as long as a given condition is true.

flow chart:



Syntax:

```
while (condition)
 {
  block of statements;
  update_expression;
 }
```

example:-

```
#include<iostream>
using namespace std;
void main()
{
 int n=1;
 while (n<=10)
 {
  cout<<"value of n:"<<n<<endl;
   n++;
 }
}
```

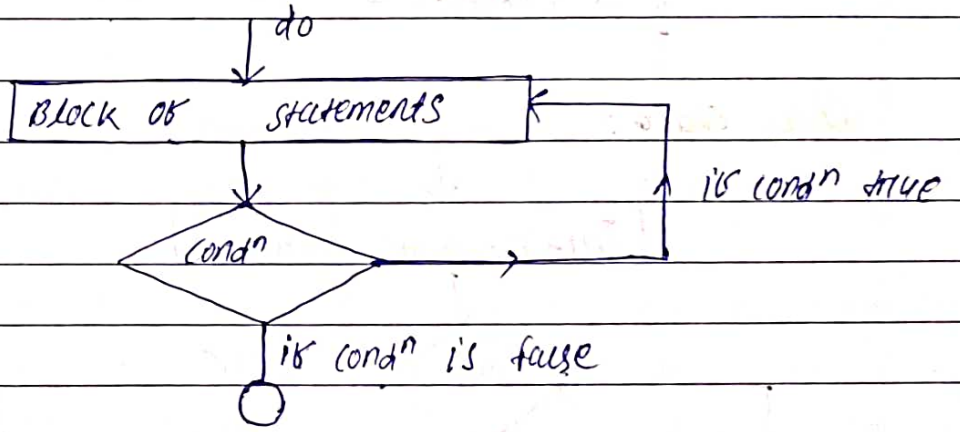
Output: value of n: 1
value of n: 2
⋮
value of n: 10

do---- while loop :

A do... while loop statement is similar to a while loop statement, except that a do---while loop statement will execute the statement at least once, even if the condition is not true.

syntax:
```
do
{
 // statements;
 update_ expression;
}
while (condition);
```

flow chart:



Program:
```
#include <iostream>
using namespace std;
int main()
{
 int n=1;
 do
 {
  cout << "value of n:" << n << endl;
  n++;
 }
 while (n<=10);
}
```

output:
```
value of n:1
value of n:2
value of n:3
:
:
value of n:10
```

for loop statement :

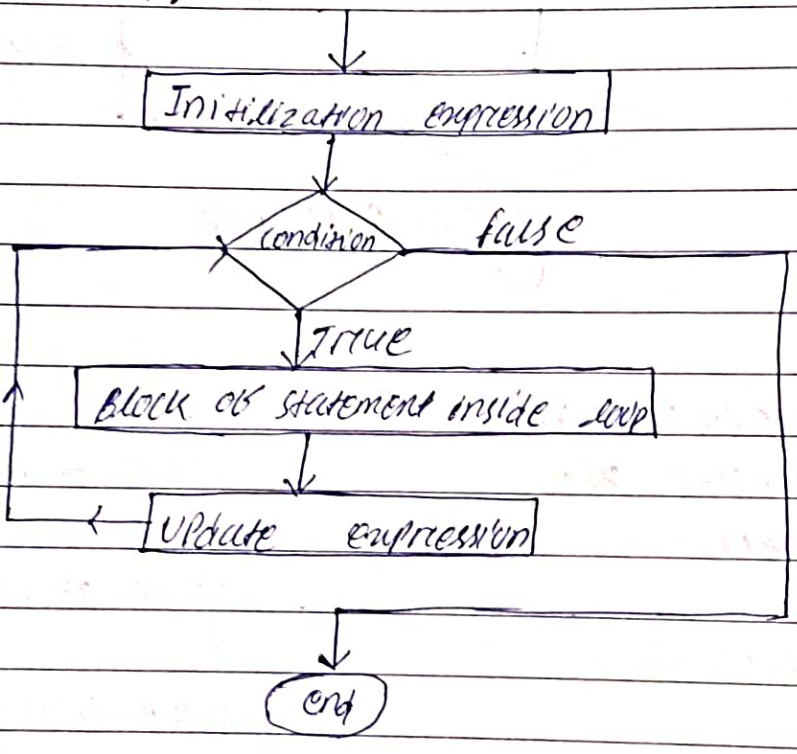
→ A for loop statement is used to repeatedly executes a block of statements(s) as long as a given condition is true.

→ A for loop statement is useful when we know how many times a task (block of statement) is to be repeated.

Syntax:

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

flow chart:



example:

```
#include <iostream>
using namespace std;
int main()
{
int n;
for (n=1; n<=10; n++)
  {
  cout<<"value of n:"<<n<<endl;
  }
}
```

output:

```
value of n:1
value of n:2
value of n:3
   :
value of n:10
```

Jumps in loops

-> As we know, looping statements are used to execute a block or statement(s) as long as the given condition is true.

-> But, in some special cases ~~use~~ we may need to terminate the loop before the condition becomes false or we may need to skip some statements during the execution of a program.

-> At that time, which type of statements we use to perform such type of operation inside the loop statement are called jump statements or jumps in loops.

-> Different types of jump statements in c++ language.

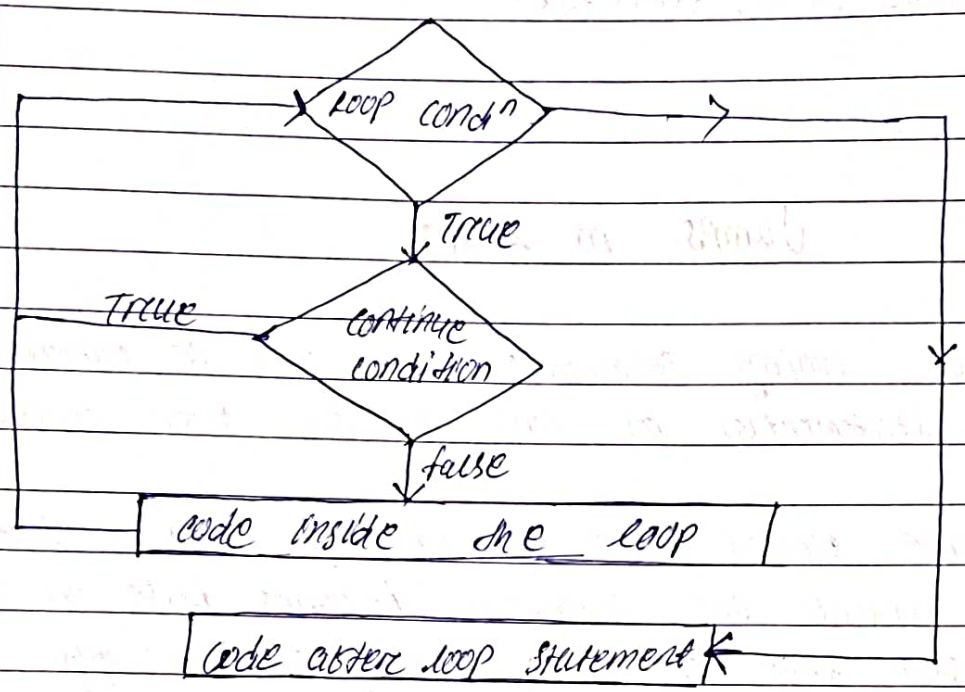
(i) Break

(ii) continue

Continue statement:-

→ When a continue statement is encountered inside a loop, the flow of control jumps to the beginning of the loop for next iteration, skipping the execution of statement inside the body of loop for the current iteration.

→ flow diagram:-



Program:-

```
#include <iostream>
using namespace std;
int main(){
    int n;
    for (n=1; n<=10; n++)
    {
      if (n==7)
          continue;
    cout<<"value of n:"<<n<<endl;
    }
}
```

Output:-

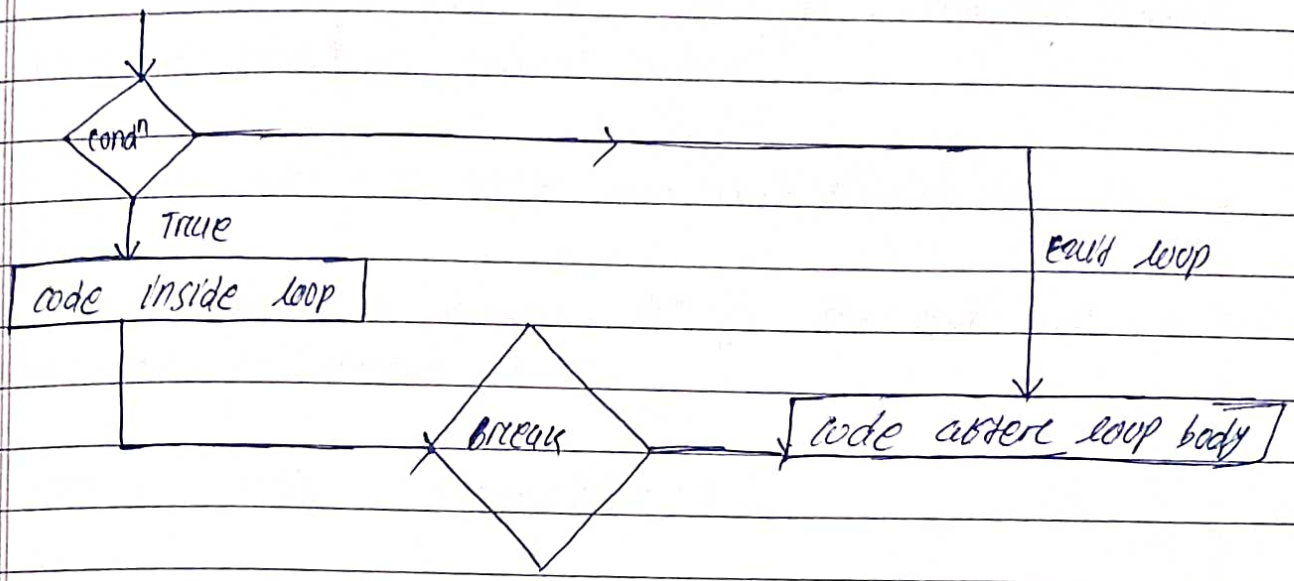
```
value of n:1
value of n:2
value of n:3
value of n:4
value of n:6
value of n:8
value of n:9
value of n:10
```

Break statement :-

-> When a Break statement is encountered inside a loop statement, the control directly comes out of loop and loop gets terminated.

-> Break statement is used in loops and switch case.

Flow digram:



Program:

```
#include <io.stream>
using namespace std;
int main()
{
  int n;
  for (n=1; n<=10; n++)
    {
      if (n==7)
          break;
    cout<<"value of n:"<<n<<endl;
    }
}
```

Output:

value of n: 1
value of n: 2
value of n: 3
value of n: 4
value of n: 5
value of n: 6

function in c++ :-

-> In c++, we can divide a large program into basic building blocks known as function.

-> function is a block having some set of statement that are used to perform some specific task and return a value.

-> c++ provides some pre-defined functions, such as main(), which is used to execute code.

-> But we can also create our own function to perform certain actions.

To create a function, specify the name of the function, followed by parentheses ().

Syntax:
```
void functionName()
{
  // code to be executed;
}
```

-> The function contains the set of programming statements enclosed by {}.

-> A function can be called multiple times to provide reusability and modularity to the c++ program.

Advantages of function:

-> By using function, we can avoid rewritting of same code again and again in a program.

-> Reusability is the main achievement of functions.

function aspects:

There are three aspects/features of a c++ function.

1. function declaration
2. function definition
3. function call

① function declaration:

A function must be declared globally in a c++ program in the global declaration section so tell the compiler about the function name, function parameteres, and return type.

Syntax :- return_type function_name (argument list);

② function definition:

→ It contains the actual statement which are to be executed.

→ It is the most important aspect to which the control comes when the function called.

→ Here, we must notice that only one value can be returned from the function.

Syntax: return_type function_name (argument list)
{
function body;
}

③ function call:

→ function can be called from anywhere in the program.

→ The parameter list must not differ in function calling and function declaration.

→ We must pass the same number of parameters as it is declared in the function declaration.

→ Syntax: function_name (argument_list);

Types of functions :-

→ There are two types of functions in C++ program.
1. Library function
2. User-defined function

1. Library functions :-

→ The functions which are declared in the C++ header files (size(), getline(), clrscr(), getch(), etc), are called Library function.

→ These functions are also called Build-in functions.

2. User-defined function :-

→ The functions which are created by the C++ Programmer, are called user-defined functions.

→ We can define the functions anywhere in the Program and then call these functions from any part of the code.

→ Just like variables, it should be declared before using, functions also need to be declared before they are called.

Return value in function :-

-> A function may or may not return a value from the function.

-> If we don't want to return any value from the function, use void as return type.

-> If we want to return a value from the function, we need to use any type (int, long, char, etc.) in the return type.

-> The return type depends on the value to be returned from the function.

eg :-

```
int sum()
{
  int a, b, sum=0;
  a=5;
  b=10;
  sum = a+b;
  return sum;
}
```

function Arguments :-

-> function arguments are also known as parameters.

-> These are the variables that we use inside the parenthesis of a function to receive the data sent by the calling program.

-> These arguments are used to provide data as input to the function to perform some specific task.

example:

```
#include<iostream>
using namespace std;
return-type function-name (argument); // function prototype
                                  └→ formal argument
void main()
{
  ------

    function_name (argument_value); // function call
                                  └→ Actual Argument
}
return_type function-name (argument) // function definition
                                  └→ formal argument
{
  ----
  -----
}
```

Types of function Arguments :-

-> There are two types of function arguments in c++.

1. formal argument
2. Actual argument

formal argument :-

The argument provided during the function declaration or function definition are called formal argument.

Actual argument :-

During function call, the values that are pass as arguments to the called functions from main function, these values are called Actual argument.

→ The Actual argument and formal argument must match in numbers, type and order.

Example :-

```
#include <iostream>
using namespace std;
void sum(int a, int b); // formal Argument.
void main()
  {
    int a, b;
    a = 50;
    b = 60;
    sum (a,b);      // Actual Argument.
  }
void sum(int a, int b)  // called function
  {
    int s=0;
    s= a+b;
    cout<< "sum" = " << s;
  }
```

Output :- sum = 110.

Types of user-defined function:-

→ There are 4 different type of user-defined functions.

1. function with no arguments and no return value.
2. function with no arguments and a return value.
3. function with arguments and no return value.
4. function with arguments and a return value.

1. function with no argument and no return value:

→ function with no argument means the called function does not receive any data from the calling function.

→ function with no return value means the calling function does not receive any data from the called function.

example:

```
#include <iostream>
using namespace std;
void sum();
void main()
{
   sum();   // calling function
}
void sum() // called fun^n
{
   int a=5, s;
   int b=b;
   s= a+b;
   cout<< "sum=" <<s;
}
```

output: sum = 15

CLASSMATE
Date
Page 20

2. function with no argument & a return value :-

→ function with no argument means the called function does not receive any data from the calling function.

→ function with return value means the calling function will receive one data (result) from the called function.

example :

```
#include <iostream>
using namespace std;
int sum();
void main()
{
    int s1 = 0
    s1 = sum();                    // calling function
    cout << "Sum = " << s1;
}
int sum()                          // called function
{
    int a, b, s = 0;
    a = 5;                         Output = Sum = 15
    b = 10;
    s = a + b;
    return s;
}
```

3. function with argument & no return value :-

→ function with argument means the called function will ~~receive~~ receive some data from the calling function.

→ function with no return value means the calling function does not recieve any data from the called function.

example :- #include <iostream>

```
using namespace std;
void sum (int a, int b);
void main()
  {
    int a,b;
     a=5, b=10;
     sum (a,b);
  }
void sum (int a, int b)
  {
    int s=0;
     s= a+b;
    cout<<" sum = "<<s;
  }
```

Output : sum = 15

4. function with argument & a return value :

-> function with argument means the called function will receive some data from the calling function.

-> function with a return value means the calling function will receive one data (result) from the called function.

example :

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

```
int a, b, s=0;
    a=5;
    b=10;
  s= sum (a, b);
  cout << "sum =" << s << endl;
int sum (int a, int b)
   {
     int s1 =0;
     s1 = a+b;
     return s;
```

User-defined function examples :-

using user-defined function WAP in c++ to find addition, substraction, multiplication & division of two numbers.

```
#include <iostream>
using namespace std;
void sum (int a, int b);
void sub (int a, int b);
void mul (int a, int b);
void div (int a, int b);
void main()
{
  int a, b;
  cout << "Enter the first no." << endl;
  cin >> a;
  cout << "Enter the second no." << endl;
  cin >> b;
  sum (a, b);
  sub (a, b);
  mul (a, b);
  div (a, b);
}
```

```
void sum (int a , int b)
  {
   int s=0;
   s= a+b;
   cout << " sum = "<<s ;
  }
void sub( int a, int b)
  {
   int s1=0;
   s1 =a-b;
   cout <<" sub = "<<s1;
  }
void mul (int a, int b)
   {
     int m=1;
     m= a*b;
   cout << "mul=" <<m;
  }
void div ( int a, int b)
   {
   int d=1;
   d= a/b;
   cout << " div=" <<d;
   }
```

functions call by value & call by reference :-

Call by value :-

-> During function call, if we are passing the value of the variables as an argument to the called function from the calling function, then it is called call by value method.

-> We can say in call by value method, the value of the actual parameters is copied into the formal parameters.

Example :-

```
#include <iostream>
using namespace std;
void sum(int a, int b); // foremal parameters
void main()
  {
  int m, n;
   m=5;
   n= 10;
   sum(m,n);  // Actual parameters
  }
void sum (int a, int b)  // called function
     {
       int sum =0;
       sum = a+b;
       cout<< "sum"<<"="<<sum;
     }
```

→ In call by value, different memory is allocated for actual and foremal parameters, since the value of the actual parameter is copied into the foremal parameter.

Call by reference :

→ During the function call, if we are passing the address of the variables as an argument to the called function from the calling function, then this is called call by reference method.

→ In call by reference, same memory is allocated for actual and foremal parameters / arguments.

example:

```
#include <iostream>
using namespace std;
void sum (int *Ptr1, int *Ptr2)
int main()
  {
    int a,b;
    a=5;
    b=10;
    sum (&a, &b);
  }
  void sum (int *Ptr1, int *Ptr2)
    {
      int sum=0;
      Sum = *Ptr1 + *Ptr2;
      cout<< "sum =" << sum;
    }
```

→ In call by reference method, we can modify the value of the actual parameter by the formal parameter.

example:

```
#include <iostream>
using namespace std;
void sum (int *Ptr1, int *Ptr2)
void main()
  {
    int a=5, b=10;
    sum (&a, &b);
  }
  void sum (int *Ptr1, int *Ptr2)
    {
      *Ptr1 =10;
      *Ptr2 = 5;
    }
```

Default argument :

→ A default argument is a value provided in function declaration that is automatically assigned by the compiler if caller of the function doesn't provide a value for the argument with default value.

→ Default argument are only allowed in the parameter lists of function declaration.

example :

```
#include<iostream>
using namespace std;
int sum(int x, int y, int z=0, int w=0);
void main()
{
int sum(int x, int y, int z=10, int w=20);
int main()
{
  cout << sum(10,20) << endl;
  cout<< sum(10, 15, 25)<< endl;
  cout<< sum(10,15, 25, 30)<< endl;
}
int sum(int x, int y, int z, int w)
 {
  return (x+y+z+w);
 }
```

Output :

50

60

80

Constant Arguments :

-> If we are declaring any argument in a function as constant, then this type of argument is called as constant argument.

-> If we want to declare any constant argument, then we must declare during function declaration.

-> An argument can be declared as constant in a function by using "const" keyword.

-> The value of a constant argument cannot be modified and any such attempt to modify that values will generates a compile time error.

Syntax: type function_name (const data_type variable_name);

Eg: int sum (int a, const int b);

Program:-
```
#include<iostream>
using namespace std;
int sum( int x, const int y);
int main()
{
 cout<<sum (10, 15)<<endl;
}
Int sum (int x, const int y)
 {
 // If we try to change the value of y
   y=x+y; // It will show error cannot modify a constant object.
   return (x+y);
}
```

Inline function:

→ The inline function is the advance feature of c++ that is used to increase the performance speed and reduce the execution time of a program.

→ By using 'inline' keyword functions can instructed to compiler to make them inline. so that compiler can replace these function definition whenever those are being called.

→ If a compiler treates a function as an inline function, then it substitute the code of function defination in a single line.

→ Instead, the executable statements of the function are copied at the place of each function call. Note:- This is just a request to compiler to make the function inline, if function is big in term of executable instruction etc.) then, compiler can ignore the "inline" request and treat the funcation as normal function.

Syntax:

```
inline return_type function_name (parameters)
{
// statements;
}
```

Example :-

```
#include <iostream>
using namespace std;
inline int cube (int s)
  {
  return s*s*s;
  }
int main ()
  {
  cout << " The cube of 3 is : " << cube(3) << "\n";
  }
```

output :

The cube of 3 is : 27

Example : 2

```
inline int add (int a, int b)
  {
  return (a+b);
  }
int main ()
  {
cout << " Addition of a & b is " << add (2,3);
 return 0;
}
```

output =

Addition of a & b is 5

friend function :

→ friend function is a special function which can access the private member of a class.
→ By using 'friend' keyword we can declare any member function as 'friend function'.
→ friend function is declared inside a class but defind outside of the class.
→ To call a friend function any object reference is not used.
→ To define a friend function class name specification is not used.
→ The syntax to declareing friend function is :
⟨friend⟩ ⟨return type⟩ ⟨function name⟩ (argument) ;
→ friend function can be declared also in private section.

ex:- WAP to display "Hello world!" using friend function.

```
#include<iostream>
using namespace std;
class demo
    {
    public:
    friend void show();  // friend function.
    };
void show()              // friend function definition.
    {
    cout<< "Hello world!";
    }
void main()
{
show();
}
```

Scope resolution operator :

Scope resolution operator is use
-> To define a member function outside of a class.
-> The syntax of using :: operator for defining member function outside

Return type <class name> : : <function name>

example:- WAP to input 2 no and find the greater one using scope resolution operator (::).

```
#include <iostream>
using namespace std;
class check
{
Private :
  int a, b;
Public :
void input ();
void show ();
};
void check :: input () // member function called by class
{
cout<<"Enter 2 no.";
cin>>a>>b;
}
void check :: show ()
{
if (a>b)
cout <<a;
else
  cout<<b;
}
```

```
void main ()
{
check obj ;
obj . input ();
obj . show ();
}
```

WAP to input 2 no. and find out the sum using friend function.

```
#include <iostream>
using namespace std;
class demo
  {
  int a,b,c;
  public:
  void input();
  friend void sum (demo);
  };
void demo :: input()
  {
  cout <<" Enter 2 no ";
  cin>> a >>b;
  }
void sum (demo d1)
    {
    d1.c = d1.a+d1.b;
    cout <<d1.c;
    }
void main()
{
 demo obj;
 obj. input();
 sum (obj);
}
```

WAP to input 2 no.s and find out the sum using friend().

```
#include<iostream>
using namespace std;
class add
  {
    int a,b,c;
    public:
    void input()    // member function
    {
    cout<<"Enter 2 no.s";
    cin>>a>>b;
    }
friend void sum(Add); // Non member or friend function
  };
void sum(add ob)
  {
    int c=ob.a+ob.b;
    cout<<c;
    }
void main()
  {
    add obj;
    obj.input();
    sum(obj);
  }
```

→ In the above program we have used friend() sum, passing (which) object as argument. Inside the sum() we are adding a&b which are private data members of the class.

NOTE:-

The difference between a friend() and a non member() is:
A friend function can access private and protected members where as a non-member function cannot access.

Classes and Object

Object:

-> Object are the basic runtime entities in oop system.
-> Object may be a Person, a bank account, a place, a table of data or any item that person handel.
-> object are the instance of class.
-> Each object contains data and function to manipulate those data.
-> During Program execution one object communicate with other object by message passing.

Class:

-> A class is an user-defined data type.
-> It has it's own member i.e data member and member function.
-> Data members are the variable of the class. & member function are the method that are used to manipulate data members.
-> once a class has been defined we can create any number of objects belonging to that class.

ex
If fruit has been defined as a class then the statement
fruit mango; will create an object 'mango' belonging to the class fruit.

-> We can create a class by using the keyword "class".

Syntax:

```
class <class-name>
{
  Data members;
  method or member function;
}
```

Example: class A // class
{
Public: // access specifier
int mynum; // data member (int type)
string mystring; // data member (string type)
void show(); // member fun^n
};

How to create a class;

→ The class keyword is used to define a class

→ Then we can use any type of access specifier.
→ Then we declare the data members.
→ Then we declare member functions.
→ At last, end the class destination with semicolon ; .

How to create an object:

→ An object is created from a class.
→ To create an object of a class, specify the class name, followed by the object name.
ex→ A obj;
→ To access the class data member or member fun^n use the . dot operator on the object.
→ we can create multiple objects of a class.
ex:- #include<iostream>
using namespace std;
class demo
{
public;
int num;
string s;
};

```
void main()
    {
      demo obj;
      obj.num = 7;
      obj.s = "subhasis";
cout << obj.num << endl;
cout << obj.s << endl;
}
```

53 Data members & methods / member functions :

-> Everything in c++ is associated with classes and objects, along with its attributes and methods.

-> for example : in real life, a car is an object. The car has attributes, such as weight and colour, and methods, such as drive and brake.

-> Attributes and methods are basically variables and functions that belongs to the class. These are also called as "class members".

-> variables declared inside the class are called as Data members.

-> functions declared inside the class are called as member function / methods.

Declaration & Definition of class methods (member function)

-> Methods are functions that belongs to the class.
-> There are two types to define member function
1. inside class definition
2. outside class definition

1. Define function inside class definition :-

-> Here we define a function inside the class.

example:

```
#include <iostream>
using namespace std;
class employee                    // class declaration
  {
    Private:
    int empid;
    float salary;
    Public:
    void display()    // member function defined inside class
    {
     empid = 001;
     salary = 10000;
     cout<<" employee id : "<< empid << endl;
     cout<<" salary : "<< salary<< endl;
    }
};
int main()          // main function
{
 employee obj;      // create an object of employee
 obj . display();   // call the method
 return 0;
}
```

2. Define function outside class definition :-

→ To define a function outside the class definition, we have to declare it inside the class and then define it outside of the class.

→ This type of function definition is done by specifying the name of the class, followed the scope resolution(::) operator, followed by the name of the function.

example:

```
#include<iostream>
using namespace std;
class employee                    // class declaration
 {
   private :
      int empid;
      float salary;
   Public:
      void display()              // member function declaration inside class
 };
void employee :: display() // member function definition outside the class
 {
   empid =001;
   salary= 10000;
cout<<" Employee id : "<< empid << endl;
cout<<" salary : "<< salary << endl;
 }
int main()                        // main function
 {
   employee obj ;                 // create an object of employee
   obj . display();               // call the function
   return 0;
 }
```

we can also add parameters to the class methods:

```
#Include<iostream>
Using namespace std;
class car {
    Pubelic:
    int speed (int maxspeed);
        };
int carc :: speed (int maxspeed) {
    return maxspeed;
    }
void main() {
        carc obj;
  Cout<< obj. speed (200) << "kmps";
        }
```

55

Making outside function inline:

We can define a member function outside the class definition and still make it inline by just using the keyword "inline" in the header line of the function defination.

example:-

```
#Include<iostream>
Using namespace std;
class S
  {
    Public:
      int square (int a);
  };
```

```
inline int S :: square (int a)
 {
    return a*a;
 }
void main()
    {
    S obj
    obj . square (4);
cout<< " square of a number is :" << obj . square (4);
    }
```

56

Nestead member functions :

→ A member function can call another member function of the same class without using the '.' operator this is called as nestead member function.

→ for calling the nestead member function object is not required.

example :-

```
#include <iostream>
using namespace std;
 class nest
    {
    int a ;
    int square ( )
    {
     return a*a;
    }
    Public :
         void input ( )
         {
```

```
cout<<"Enter a number";
cin>>a;
}
void display()
{
int sq = square(); // nested member function
cout<<" The square of "<<a<<"is"<<sq;
}
};
int main()
{
nest obj;
obj.input();
obj.display();
return 0;
}
```

57 Private member function:

→ A function declared inside the class's private section is known as "Private member function".

→ A private member is accessible only through the public member function of that particular class.

example:-

```
#include<iostream>
using namespace std;
class employee
{
private:
    int empid;
```

```
float salary;
void input()
  {
    empid = 001;
    salary = 100000;
  }
  Public:
      void display()
        {
          input();
          Cout<<" Employee id : " << empid << endl;
          cout<<" employee salary : "<<salary<< endl;
        }
};
int main()
  {
    employee obj;
    obj . display();
    obj . input();

  }
```

58 Arrays within a class :-

→ Array can be declared as the members of a class.

→ The array can be declared as private, Public or Protected members of the class.

example :- WAP to input student details with the rollno. and 5 subject mark then display total marks.

```
#include<iostream>
using namespace std;
class student
  {
    int mark[5], i;
    int roll no;
    Public:
      void input_array()
        {
          cout<<"Enter rollno."<<":";
          cin>>rollno;
          cout<<"Enter subject mark"<<(i+1)<<":";
          for (i=0;i<5;i++)
            {
              cin>>mark[i];
            }
        }
      void display_array()
        {
          int s=0;
          cout<<" Roll no. = "<<rollno<<endl;
```

```
for (i=0; i<5 ; i++)
  {
   s= s+ marks[i];
  }
cout<<" Total marks = "<<s ;
  }
};
int main()
  {
   student obj;
   obj . input_array();
   obj . display_array();
   return 0;
  }
```

Output: Enter rollno.
21028
Enter subject mark
10
20
30
40
50
Rollno. = 21028
Total mark = 150

59

Memory allocation for objects :-

-> Before using a member of a class, it is necessary to allocate the required memory space to that member.

-> The way the memory space for data members and member function is allocated is different. Whenever both data members and member functions belongs to the same class.

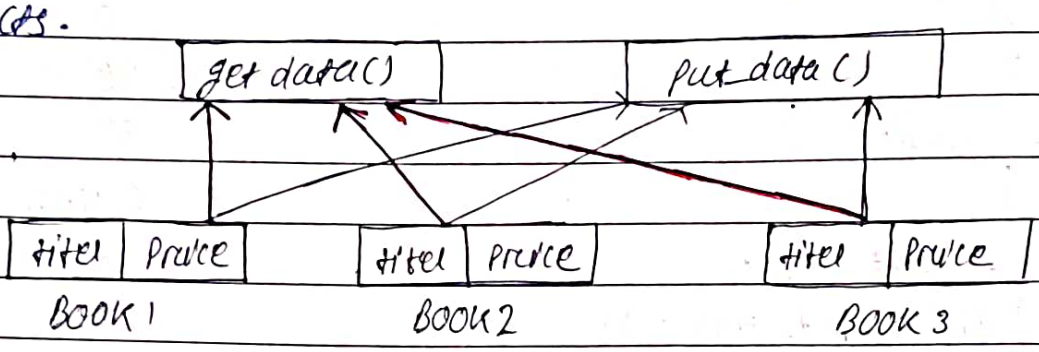
-> The memory space is allocated to the data members of a class only when an object of the class is declared, and not when the data members are declared inside the class.

-> Since a single data member can have different values for different objects at the same time, every object has an individual copy of all data members.

→ The memory space for the member functions is allocated only once when the class is defined.

→ There is only one single copy of each member function, which is shared among all the objects.

ex →



→ The three objects, namely book1, book2, book3 of the class BOOK have individual copies of the data member title and price, however there is only one copy of the member functions getdata() and putdata() that is shared by all the three objects.

60 Static data members:

→ Static data members are class members that are declared using the 'static' keyword.

→ The static data member gets memory only once, which is shared by all the objects of its class.

→ The static data member is always initialized to zero when the first class object is created.

→ The static data member is associated with class not with the object.

-> The static data members must be defined outside of the class ; otherwise, the linker will generate an error.

-> The syntax of the static data members is given as

static <data-type> <data-member_name>;

eg-> static int a;

Program: WAP to find area of circle.

```
#include <iostream>
using namespace std;
class circle
  {
    float r;
    static float Pi;
  Public:
     void getdata();
     void area();
  };
  float circle ::Pi = 3.147;
  void circle ::getdata()
   {
    cout<<" In input radious : ";
    cin>>r;
   }
  void circle ::area()
   {
    float area = Pi*r*r
    cout<<" In Area of circle is = "<<area;
   }
void main()
   {
    circle obj;
    Obj.getdata();
  { Obj.area();
```

output:

input radious : 4

Area of circle is = 50.24

(61) Static member functions :

→ A static member function is a special member function, which is used to access only static data members and other static member functions.

→ Any other normal data member or member function cannot be accessed through static member function.

→ Just like static data member, static member function is also a class function ; it is not associated with any class object.

→ A static member function ~~we don't~~ can be called even if no objects of the class exist and the static functions are accessed using only the class name and the scope resolution operator (::).

→ This member function is declared using "static" ~~member~~ keyword.

Program :

```
#include <iostream>
using namespace std;
class notebook
    {
      static int page_number; // static data member
public:
      static int values() // static member function
      {
       return page_number;
      }
    };
```

```
int notebook :: Page_number = 1;
void main()
{
  cout<< "Number = "<<notebook :: values()<<endl;
}
```

Output:
Number = 1

ex→2

```
class note
  {
  static int num;
  Public:
  static int fun()
   {
    cout<< "num = "<<num <<endl;
   }
  };
int note :: num = 5;
Int main()
  {
  note obj;
  obj.fun();
  return 0;
  }
```

Output:
num = 5

(62) Objects of Array :-

→ Like array of other user-defined data types, an array of class can also be created.

→ The array of class contains the objects of the class as its individual element.

→ Array of a class type is also known as an array of objects.

Syntax fore declaring an array of object it's

class_name array-name [size];

Program:- Will do enter Book details and show them.

```
#include <iostream>
using namespace std;
class books
  {
    char title [30];
    float price;
    Public:
      void getdata();
      void showdata();
  };
  void books ::getdata()
  {
   cout<<"Title : ";
   cin>> title;
   cout<< "Price :";
   cin>> Price;
  }
  Void books ::showdata()
  {
   cout<<" Title : "<< title <<"\n";
   cout<<" Price : "<< price <<"\n";
  }
  void main()
    {
      books book[3]; // array of object
      fore (int i=0; i<3; i++)
      {
      cout<<"Enter details of Book"<<(i+1)<<"\n";
       book[i]. getdata();
      }
```

```
for (int i=0; i<3; i++)
    {
    cout<<"In Book:"<<(i+1)<<"\n";
    book[i].showdata();
    }
}
```

Output:

```
Enter details of book 1
Title : C++
Price : 300
Enter details of book 2
Title : DBMS
Price : 400
Enter details of book 3
Title : Java
Price : 500
```

```
Book 1
Title : C++
Price : 300
Book 2
Title : DBMS
Price : 400
Book 3
Title : Java
Price : 500
```

Objects as function Arguments :-

→ The objects of a class can be passed as arguments to member functions either by value or by reference.

→ When an object is passed by value, a copy of the actual object is created inside the function. This copy is destroyed when the function terminates.

→ In pass by value, any changes made to the copy of the object inside the function are not reflected in the actual object.

→ In pass by reference, only a reference (address) to that object is passed to the function. Thus, the changes made to the object within the function are also reflected in the actual object.

-> Whenever an object of a class is passed to a member function of the same class, its data members can be accessed inside the function using the object name and the dot operator.

-> However, the data members of the calling object can be directly accessed inside the function without using the object name and the dot operator.

Program :-

```
#include<iostream>
using namespace std;
class demo
    {
      int a;
      Public:
        void set(int x)
          {
            a=x;
          }
        void sum(demo ob1, demo ob2)
          {
            a= ob.a + ob2.a;
          }
        void print()
          {
          cout<<"value of A:"<<a<<endl;
          }
void main()
 {
 demo d1;
 demo d2;
 demo d3;
```

```
d1.set(10);
d2.set(20);
d3.sum(d1,d2); // object as function argument.
d1.Print();
d2.Print();
d3.Print();
}
```

output :-

value of A : 10
value of A : 20
value of A : 30

Constructors & Destructors :

(69)

Constructors :

-> Constructors in c++ are special member function which are created when the object of a class is created or defined.

-> Its task is to initialize the object of its class.

-> A constructor has the same name as the name of the class and it don't have any return type.

-> It is called automatically whenever an object of its associated class is created.

-> It don't have any return type, it can't return values.

-> To create a constructor, use the same name as the class, followed by parentheses ().

Program:

```
#include<iostream>
using namespace std;
class myclass                    // my class
{
public:                          // Access specifier
    myclass()                    // Constructor
    {
    cout<<" Hello world!";
    }
};
void main()
  {
    myclass obj;  // create an object of my class
    return 0;                    // This will call the constructor
  }
```

Output:-
Hello world!

Types of constructores:-

1. Default constructore
2. Parameterized constructore
3. Copy constructore

NOTE: constructore must be declared or defined in Public section of the class.

70

1. Default constructore:

→ A default constructore is a constructore, which does not have any parameteres.

→ It is called once you declare an object.

→ compiler supplies a default constructore if no such constructore is defined.

Program:

```
#include <iostream>
using namespace std;
class student
    {
    public:
    int roll-no;
    char name [10];
    double fee;
    public:
        student()
        {
        cout << "Enter rollno. ";
        cin >> roll-no;
```

```
cout<<" Enter student name : ";
cin>> name;
cout<<" Enter fee : ";
cin>> fee;
void display()
    {
    cout<<" endl << roll-no <<"\t" <<name <<"\t" << fee;
    }
};
int main()
  {
   student s;
   s. display();
   return 0;
  }
```

output :

Parameterized constructor :

-> The constructors that can take arguments are called parameterized constructor.

-> Sometimes, it may be necessary to initialize the data members of different objects with different values when they are created. Hence parameterized constructors are used.

Program :
```
#include <string.h>
#include <iostream>
using namespace std;
class student
 {
  int roll-no;
  char name [10];
```

```
double fee;
Public:
    Student (int no, char n[10], double f)
        {
            roll_no = no;
            strcpy (name, n);
            fee = f;
        }
        void display()
        {
cout<< "\nName is = "<<name<< "\nRoll no = "<< roll_no << "\n fees = "<<fee
                                                                <<endl;
        }
    };
int main()
 {
   Student obj (1, "manu", 1000);
   Student obji (2, "Puja", 2000);
   Obj . display();
   Obji . display();
   Return 0;
 }
```

Output:

Name is = manu
Roll no = 1
fees = 1000

Name is = Puja
Roll no = 2
fees = 2000

Constructor with default argument:

→ A default argument is a value provided in function declaration that is automatically assigned by the compiler if caller of the function doesn't provided a value for the argument with default value.

→ It is possible to have a constructor with default argument.

Program:

```
#include<iostream>
using namespace std;
class Point
 {
    int a, b;
public:
      Point (int x1, int y1 = 15)
        {
          a=x1;
          b=y1;
        }
      int getA()
        {
          return a;
        }
      int getB()
        {
          return b;
        }
void main () {
          Point obj(5); // constructor called.
cout<< "A= " << obj.getA() << "B= " << obj.getB();
        }
```

Output:

A=5 , B=15

Note: Default arguments are assigned from right to left.
-> Default arguments are different from constant arguments as constant arguments value can't be changed where as default arguments value can be overwritten if required.

Copy constructor:

-> A copy constructor is a type of constructor, that initializes an object using another object of the same class.
-> copy constructor takes a reference to an object of the same class as an argument.
-> Syntax of declaring a copy constructor :-

```
ClassName (const className & old_object);
```

Program:

```
#include<iostream>
using namespace std;
class Student
{
public:
int x;
Student(int a);
{
x=a;
}
Student (Student &a1) // copy constructor
{
x= a1.x;
}
};
int main() {
Student a1 (20);
Student a2 = a1;
cout<< "copy constructor is :" << a2.x;
return 0;
}
```

output:

copy constructor is : 20

Dynamic initialization of object :-

→ Dynamic initialization of object refers to initializing the objects at run time i.e the initial value of an object provided during run time.

→ This type of initialization is required to initialize the class variables during run time.

→ It can be achived using constructors and passing Parameters values to the constructore (parameterized constructore).

Program:

```
#include <iostream>
using namespace std;
class Point
  {
    int a, b;
    Public:
      Point (int x1, int y1)
        {
          a=x1;
          b=y1;
        }
      void display ()
        {
          cout<< " values after dynamic initialization "<< endl;
          cout<< "\nA = "<<a<<"\nB = "<<b<<endl;
        }
  };
int main()
 {
```

```
int x1,y1;
cout<<"Enter firest no A = ";
cin>>x1;
cout<<"Enter second no B = ";
cin>>y1;
Point obj(x1,y1); // Dynamic initialization
obj.display();
return 0;
}
```

output:

```
Enter firest no A = 10
Enter second no B = 20
Values after dynamic initialization
A=10
B=20
```

note :- If we return values inside the parameter then that time it is called as static initialization. → Point obj(10,20)

Dynamic initialization of object by using Pointer.

```
class A {
        int *ptr;
        public:
        A()
          {
           ptr= new int ;
           *ptr = 10;
           }
           void display()
           {
            cout<< *ptr << endl;
            }
          };
          int main() {
                      A obj ; // or A obj = new A();
                      obj.display();
                      return 0;
                      }
```

→ we can allocate memory at runtime by using 'new' and deallocate by using 'delete'.

Dynamic constructors :-

→ The constructor that is used to allocate the memory to the objects at the run time, is known as Dynamic constructor.

→ Memory is allocated at run time with the help of 'new' operator.

Program :-

```
#include<iostream>
using namespace std;
class example
    {
      int *p;
      Public:
        example()
      //dynamic constructor
          {
            p= new int;
            *p= 10;
          }
        void display(){
cout<<"value of p = "<<*p;
                }
        };
int main()
   {
    example obj;
    obj. display();
    return 0;
   }
```

O/P value of p=10

```
#include<iostream>
using namespace std;
class example
    {
      Int *p;
      Public:
        example (int a)
      //Dynamic parameterized constructure
          {
            p= new int;
            *p= a;
          }
        void display(){
cout<<"value of p= "<<*p;
                }
        };
int main()
   {
    example obj(20);
    obj . display();
    return 0;
   }
```

value of p = 20

```
int main(){                                output:-
    example obj (100);                     value of p = 100
    obj.display();                         value of p = 10
    example obj2;
    obj2.display();
}
```

17 Constructor overloading :-

→ If a class having more than one constructor with different parameters, then this is called constructor overloading.

→ Each constructor is used to perform different task.

Program:-

```
#include <iostream>
using namespace std;
class example {
        int a;
    public:
        example(){          // constructor with no parameters.
            a=0;
        }
        example (int b){    // constructor with one parameters.
            a=b;
        }
        void display(){
            cout<<"a = "<<a<<endl;
        }
};
void main(){                               output:
    example obj;                           a=10
    example obj1(20);                      a=20
    obj.display();
    obj1.display();
    return 0;
}
```

Destructors :-

-> A destructore is a speical member function that work just opposite to constructor.

-> constructores are used fore initializing an object, destructors are used to destroy or delete the object.

-> Similare to constructor, the destructore name should exactly match with the class name.

-> A destructore declaration should always begin with the tilde (~) symbol.

-> A destructore don't return a value and it can not take any Parameteres.

-> A destructor is automatically called when the program finished execution.

Syntax :-

```
~class_name()
  {
  // code
  }
```

Program :-

```
#include<iostream>
using namespace std;
class CD
  {
  public:
~CD().
  {
cout<<" constructore invoked "<<endl;
}
```

```
~CD ()
  {
  cout<<" Destructor invoked "<<endl;
  }
};
int main()
  {
  CD obj;
  return 0;
  }
```

Output:

Constructor invoked
Destructor invoked

unit-3

Inheritance

classmate
Date
Page 117

→ The process of creating new class from an existing class is called as inheritance.

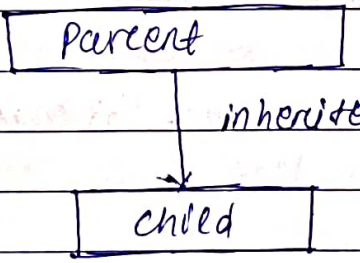
→ In oops the concept of inheritence gives the idea of reusability.

→ In this process object of one class inherit the properties of object of another class completely or partially.

→ The new class is called as child class or sub class or derived class.

→ The existing class is called as parent class or super class or base class.

→ Inheritance defines parent-child relationship between classes.



→ An objects of a child class can access or inherit the data members and methods of the parent class. By using colon (:).

Syntax:
```
class <Parent_class>
{
    // code;
};
class <child_class>:<access_mode><Parent_class>
{
    // body of child_class
};
```

Program: #include <iostream>
using namespace std;
class A // parent class
{
int a;
Public:
void input()
{
cout<<"Enter a no."<<endl;
cin>>a;
}
void show(){
cout<<a;
}
};
class B : Public A // child class
{
int b;
Public:
void input_1()
{
cout<<"Enter a no."<<endl;
cin>>b;
}
void show_1()
{
cout<<b;
}
};
int main(){
B obj;
obj.input();
obj.show();
obj.input_1();
obj.show_1();
}

Advantages of inheritance :

-> Reusability of existing code.
-> Enhancement of base class.
-> Reduction of time and effort.
-> Increment of program reliability.

Access mode of inheritance :-

-> A derived class can access all the non-private members of its base class.
-> Thus base class members that should not be accessible to the member functions of derived classes should be declared private in the base class.

1- Public mode :

-> If we derive a sub class from a public base class.
-> Then the public members of the base class will become public in the derived class and protected members of the ~~class~~ base class will become protected in the derived class.

2- Protected mode :

-> If we derive a sub class from a protected base class.

-> Then both public and protected members of the base class will become protected in derived class.

3- Private mode :

-> If we derive a sub class from a private base class.
-> Then both public member and protected members of the base class will become private in derived class.

Base class member access specifier	Type of Inheritance		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible	Not accessible	Not accessible

Note :-

The Private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed.

Accessibility of the members of base class :-

Access	Public	Protected	Private
same class	yes	yes	yes
Derived class	yes	yes	NO
outside the class	yes	NO	NO

example :-

```
class A
  {
  public:
     int x;
  Protected:
     int y;
  Private:
     int z;
  };
class B : public A
  {
  // x is public
```

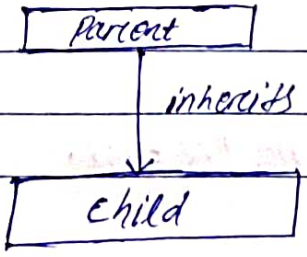
```
// y is protected
// z is not accessible from B
};
class C : Protected A
{
// x is protected
// y is protected
// z is not accessible from C
};
class D : Private A
{
// x is private
// y is private
// z is not accessible from D
};
```

Types of inheritance :-

-> Single inheritance
-> Multiple inheritance
-> Multilevel inheritance
-> Heercarchical inheritance
-> Multipath inheritance
-> Hybrid inheritance

1. Single inheritance :

-> When a child class inherits properties of an only one parent class, then it is known as single inheritance.



Syntax: class <Parent_class>
{
// body of Parent_class
};
class <child_class> :<access_mode> <Parent_class>
{
// body of child_class
};

Example of single inheritance:

```
#include<iostream>
using namespace std;
class A                    // Parent class
  {
   int a;
   Public:
     void input()
      {
    cout<<" Enter a no. "<< endl;
    cin>> a;
      }
     void show()
      {
       cout<< a;
      }
  };
class B : Public A     // child class
  {
   int b;
   Public:
     void input1()
      {
    cout << " Enter a no. "<< endl;
    cin >> b;
    }
```

```
void showl()
 {
  cout<<b;
 }
};
void main()
 {
  B obj;
  obj.input();
  obj.inputl();
  obj.showc();
  obj.showl();
  return 0;
 }
```

-> In the above example class B object - obj consist of both, a Parent and b Parent so we can call both class A function and class B function.

2. Multiple Inheritance :

-> When a child class inherits properties of more than one Parent classes then it is called as multiple inheritance.

Syntax :- class <Parent_class1>
{
// body of Parent class1;
};

```
class <Parent_class2>
  {
  // body of Parent_class 2
  };
class <child_class> : <access_mode> <Parent_class1>, <access_mode>
                                    <Parent_class2>
    {
      // body of child_class
    };
```

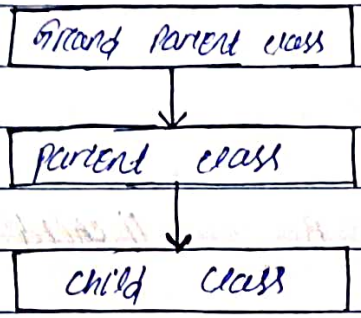
Program:-

```
#include<iostream>
using namespace std;
class A                          // Parent class 1
  {
  int a;
  Public:
    void input()
    {
    cout<<"Enter a no."<<endl;
    cin>>a;
    }
    void show()
    {
     cout<<a;
    }
  };
class B                          // Parent class 2
  {
  int b;
  Public:
    void inputB()
    {
    cout<<"Enter a no."<<endl;
```

```
cin>>b;
}
void showc()
{
cout<<b;
}
};
class c : public A , public B        // child class
    {
      int c;
      public:
        void input2()
        {
        cout<<"Enter a no."<<endl;
        cin>>c;
        }
        void show2()
          {
          cout<<c;
          }
      };
void main()
  {
   C obj;
  obj.input();
  obj.inputc();
  obj.input2();
  obj.show();
  obj.showc();
  obj.show2();
  }
```

3 Multilevel Inheritance :

The inheritance in which a class can be derived from another derived class is known as multilevel inheritance.



Syntax :

```
class <grand Parent>
  {
   // body of grandparent class;
  }
class <Parent_class> :<access_mode> <grandparent_class>
  {
   // body of Parent_class
  };
class <child_class> : <access_mode> < parent class>
  {
   // body of child_class
  };
```

Program : #include <iostream>

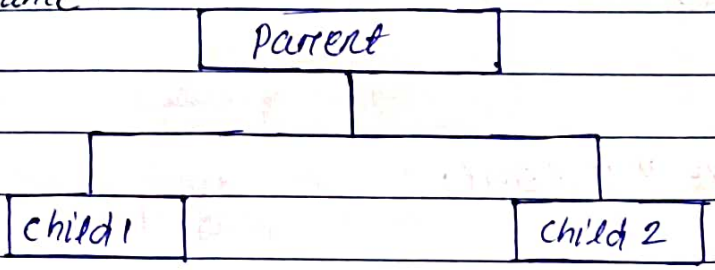
```
using namespace std;
class A
  {
   int n;
   public:
    void display()
      {
        n=50;
       cout<< "n = "<<n <endl;
      }
  };
```

```
class B : public A
    {
    int y;
    public:
    void show()
    {
     y= 60;
     cout<< "y= " y<< endl;
    }
   };
class C : public B
    {
    int z;
    public:
       void result()
       {
       z=70;
       cout<< "z= "<<z;
       }
    };
void main()
  {
    C obj;
   obj. display();
   obj. show();
   Obj. result();
   return 0;
  }
```

Output : x = 50
y = 60
z = 70

4. Hierarchical inheritance :

-> When more than one child classes inherits properties of a single parent class, then it is called as hierchical inheritance.



Syntax :-

```
class <Parent_class>
    {
      // body of Parent_class
    };
class <child_child1> : <access_mode> <Parent_class>
    {
      // body of child_class1
    };
class <child_class2> : <access_mode> <Parent_class>
    {
      // body of child_class2
    };
```

Program:

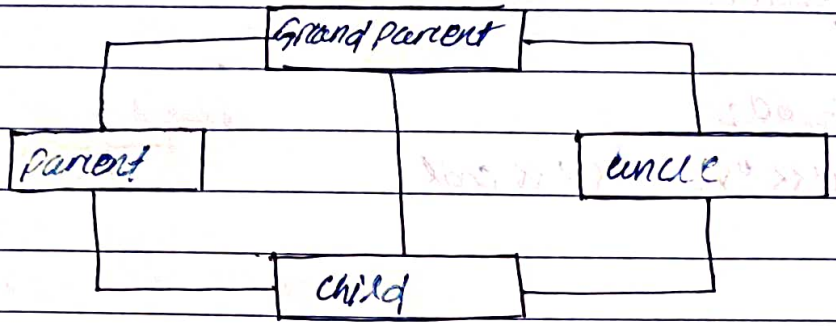
```
#include <iostream>
using namespace std;
class Parent
      {
      int x;
      Public:
      void display()
      {
       x=50;
       cout<< "x = "<<x<<endl;
      }
```

```
};
class child1 : public parent
{
    int y;
    public:
    void show()
    {
      y= 60;
      cout<< "y= " << y << endl;
    }
};
class child2 : public Parent
{
    int z;
    public:
      void result()
      {
        z=70;
        cout<< "z = " <<z;
      }
};
void main()
{
    child1 obj;
    child2 obj1;
    obj . display();
    obj1 . display();
    return 0;
}
```

Output - x=50
x=50

5. Multipath Inheritance :

→ When a child class inherits properties of a parent class from more than one path then it is called as multipath inheritance.



Syntax :

```
class <grandparent_class>
    {
    // code
    };
class <parent_class> : <access mode> <grand parent_class>
    {
    // code
    };
class <uncle_class> : <access mode> <grantparent_class>
    {
    // code
    };
class <child_class> : <access mode> <Parent_class>
    {
    //code
    };
```

Program :-

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

```
int x;
public:
    void display()
    {
     x=50;
    cout<<" x= "<<x<<endl;
    }
};
class parent : public grandparent
    {
     int y;
    };
class uncle : public grandparent
    {
     int p;
    };
class child : public parent      // Here the child calass have 3 classes
    {                            // to inherit grandparent, parent, uncle
    int z;
    public:
        void result()
        {
        z=70;
        cout<<"z= "<<z;
        }
    };
void main()
{
child obj;
obj.display();
obj.result();
}
```

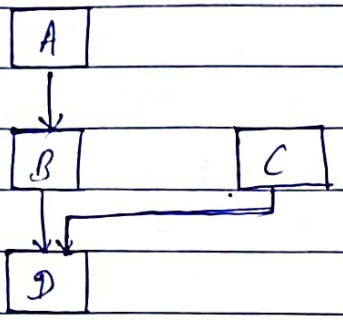
Output:

x = 50

z = 70

Hybrid inheritance :

→ When two or more inheritance are used together in a single program, then it is known as Hybrid inheritance.



Syntax :

```
class <A>
  {
  //code
  };
class <B> :<access mode> <A>
  {
  //code
  };
class <C>
  {
  //code
  };
class <D> :<access mode> <B> , <access mode> <C>
  {
  //code
  };
```

Program :-

```
#include <iostream>
using namespace std;
class A.
{
 Public :
   int a;
 void input_1 ()
    {
```

```
cout<<"\n Enter value at A: ";
cin>>a;
}
void display_1()
   {
   cout<<"value at A = "<<a<<endl;
   }
};
class B : Public A
  {
   Public :
    int b;
   void input_2()
     {
     cout<<"\n Enter value at B : ";
     cin>>b;
     }
   void display_2()
     {
     cout<<"value at B = "<<b<<endl;
     }
  };
class C
  {
  Public :
   int c;
     void input_3()
       {
       cout<<"\n Enter value at C : ";
       cin>>c;
       }
     void display_3()
```

```
{
cout<<"value at c = "<<c<<endl;
}
};
class D : public B, public C
    {
    public:
     int d;
     void input_4()
        {
        cout<<"\n Enter value at D : ";
        cin>>d;
        }
      void display_4()
        {
        cout<<"value at D="<<d<<endl;
        }
    };
int main()
  {
    D obj;
   obj. input_1();
   obj. display_1();
   obj. input_2();
   obj. display_2();
   obj. input_3();
   obj. display_3();
   obj. input_4();
   obj. display_4();
   return 0;
  }
```

Virtual base classes :-

-> An ambiguity can occur when several paths exist to a class from the same base class.

-> This means that a child class could have duplicate sets of members inherited from a single base class.

-> At that time compiler will generate an error. C++ solves this issue by introducing a virtual base class.

-> When a class is made virtual, necessary care is taken so that the duplication is avoided regardless of the number of paths that exist to the child class.

-> The virtual class is used when a derived class has multiple copies of the base class.

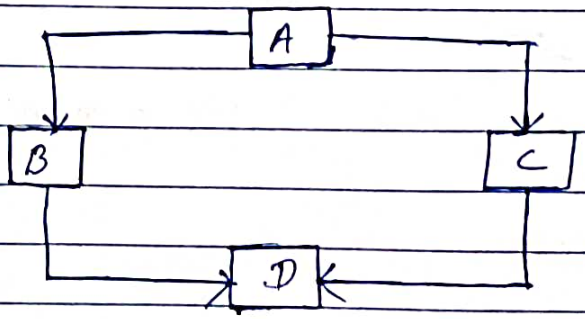
Syntax :

```
class <class-name> : virtual <access mode> Parent_class>
  {
  };
```

```
class B : virtual public A
  {
  };
```

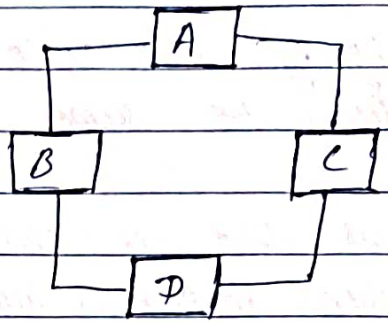
```
class <class_name> : public virtual <Parent class>
  {
  };
```

```
class B : public virtual A
  {
  };
```



Program :- #include <iostream>

```
class A
  {
public :
  int a;
  void display()
   {
   a = 10;
   cout<< "a = " <<a << endl;
   }
};
class B : public virtual A
   {
   };
class C : public virtual A
   {
   };
class D : public B, public C
   {
   };
void main()
   {
   D obj;
   obj. display();
   }
```



Output :

a = 10

Virtual function:

→ A virtual function is a member function that is declared within a base class and redefined by a derived class.

→ To create a virtual function, we use the keyword virtual to the function name during function declaration.

→ Base class pointer can point to derived class object.

→ In this case, using base class pointer if we call some function which is in both classes, then the base class function is invoked.

→ But if we want to invoke derived class function using base class pointer, it can be achived by defining the function as virtual in base class.

Program:

```
#include<iostream>
using namespace std;
class A
  {
  public:
  virtual void display()          // virtual function.
  {
cout<<"Base class is invoked"<<endl;
 }
};
class B : public A
  {
 public:
void display()
```

```
{
cout<<"Derived class is invoked"<<endl;
}
};
void main()
 {
  A *p;
  B obj;
  P=& obj;
  P-> display ();
}
```

output :- Derived class is invoked

Pure virtual function :-

→ If a virtual function is not used for performing any task, then this virtual function is known as pure virtual function.

→ Pure virtual function can be defined as :-

virtual void display () = 0;

virtual <return_type> <function_name> = 0;

Program :-

```
#include <iostream>
using namespace std;
class Base
   {
    public:
    virtual void display()=0; //Pure virtual function.
    };
class B : public Base
  {
```

```
Public:
void display()
 {
cout<<" Derived class is invoked "<< endl;
 }
};
void main()
 {
  B obj;
 obj.display();
 }
```

Output:
Derived class is invoked.

Abstract class:-

→ A class containing the pure virtual function cannot be used to declare its object. Such class are known as abstract classes.

→ In the above example, the base class contains the pure virtual function. Therefore, the base class is an abstract base class. We cannot create the object of the base class.

Nesting of classes:-

→ The process of declaring a class inside the body of another class, is called as Nesting of classes.

→ A Nested class is declared inside another class.

→ The nested class is also a member of the enclosing class and has the same access rights as the other members.

→ How ever the member function of the enclosing class have no special access to the member of a nested class.

Program:-

```
#include <iostream>
using namespace std;
class A                    //enclosing class
  {
public:
    class B                //Nested class
      {
        int num;
        public:
          void getdata (int n)
            {
              num=n;
            }
          void putdata()
            {
              cout<<" The number is "<< num;
            }
        };
    };
int main()
{
  A :: B obj;
  obj.getdata(9);
  obj.putdata();
}
```

Output = The number is 9

Polymorphism:

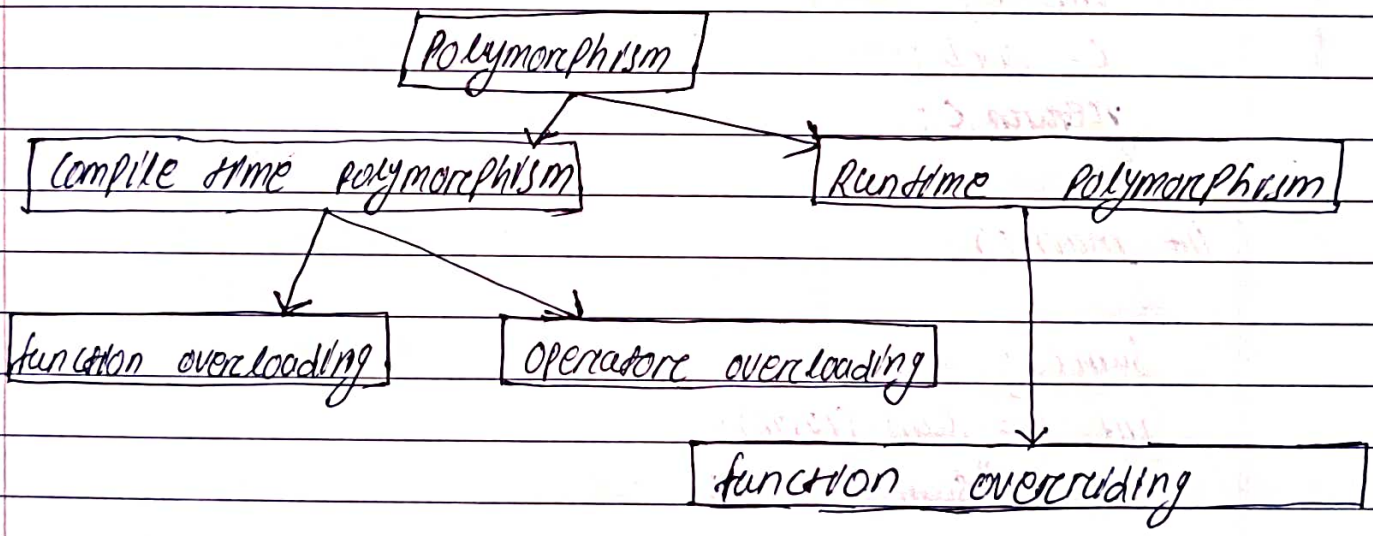
classmate
Date
Page 141

84

-> The word polymorphism means having many forms.
-> In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.
-> A real life example of polymorph is a man who have different characteristics at a same time like, son, father, brother etc.

-> Polymorphism is the most important feature of oops.

-> In c++ polymorphism is mainly divided into two types:
1- compile time polymorphism
2- Runtime polymorphism.



function overloading:-

-> function overloading is a compile time polymorphism.

-> When multiple functions can be declared with the same name but different prototype.

-> In function overloading "function" name should be the same and the arguments should be different.

Program: WAP to input 2 no. and find out the sum using function overloading:- (without class).

```
#include <iostream>
using namespace std;
void sum()
 {
  int a,b,c;
  cout << " Enter 2 no. ";
  cin>> a >>b;
  cout<< "sum = "<< c;
 }
int sum (int a, int b)
   {
    int c;
    c= a+b;
    return c;
   }
int main()
   {
   sum();
   int c = sum (10,20);
   cout<<"Sum = "<< c;
   }
```

Using class:-

```
#include <iostream>
using namespace std;
class Add
   {
   int a,b,c;
   Public:
      void sum()
        {
```

```
cout<<"Enter 2 no.s";
cin>> a>>b;
c= a+b;
cout<< "sum ="<< c;
}
int sum (int x, int y)
   {
   c= x+y;
        cout<<" sum = "<<c<<endl;
   }
};
void main()
   {
   Add obj;
   obj. sum();
        obj. sum (10,20);
   }
```

Output:

```
Enter 2 numbers = 15
20
sum = 35
sum = 30
```

Operator overloading:

→ It is one type of compile time polymorphism.

→ c++ allows us to specify more than one defination for an operator in the same scope, which is called as operator overloading.

→ We can overload most of the build-in operators available in c++.

→ operator overloading can be implemented either using member function or friend function.

Syntax:

<return type> operator <operator symbol> (Argument)

Rules for operator overloading :-

1- Operator function is used.
2- Here we have to use class object.
3- In case of unary operator overloading one class object is used.
4- In case of binary operator overloading either 2 class objects are used or one class object with one variable (predefined)

→ Some special type of operator cannot be overloaded, ::, sizeof, ., *, ?: etc

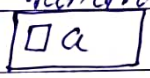
Program:

```
#include<iostream>
using namespace std;
class A
  {
  int a;
  public:
  void input()
   {
   cout<<"Enter a no."<<endl;
   cin>>a;
   }
void operator ++()
     {
      a++;
     }
void show()
    {
     cout<<"Number = "<<a;
    }
};
```

```
void main()
{
  A obj;
  Obj. input();
  ++ obj;
  obj. show();
}
```

Description :-

-> In the above example we have used a class A.
-> Inside the class A we have use the data members a.
-> Inside the class we have use 3 member function.
   input(), operator++(), show().
-> Inside the main function we have use an object -> obj



-> In the input function let we have entered 10

-> The statement obj++ will execute the operator++() function and the value of a is incremented to 11.

-> By using the show() we will get output 11.

* The statement obj++ proves operator overloading.

Binary operator overloading.

WAP to overload + operator.

```
#include <iostream>
using namespace std;
class A
  {
    int a;
    public:
```

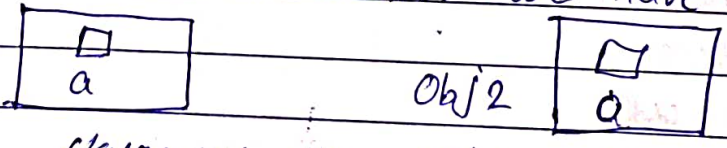
```
void input()
{
cout << "Enter a no.";
cin >> a;
}
int operator + (A temp)
    {
     int c;
     c = a + temp.a;
     cout << c;
    }
};
void main()
{
A obj1, obj2;
obj1.input();
obj2.input();
int c = obj1 + obj2;
}
```

Description:

→ The above example we have implemented binary + operator overloading.

→ In the class A we have used one input function and one operator + function.

→ Inside the function main() we have used 2 object



→ Now the statement c = obj1 + obj2 will execute operator + function of object obj1.

→ Inside the operator +() we have used 'temp' which will contain obj2 object.

→ Inside the operator+() the statement a+temp.a specifies → obj1.a + obj2.a ;

function overridding :-

→ suppose the same function is defined in both the derived class and base class. Now if we call the function using the object of the derived class, the function of the derived class is executed.

→ This is known as function overridding in c++

→ The function in derived class overrides the function in base class.

example :-
```
#include<iostream>
using namespace std;
class Base
    {
    public:
        void print()
            {
                cout<<" Base function "<<endl;
            }
    };
class Derived : public Base
    {
    public:
        void print()
            {
                cout<<" Derived function "<<endl;
            };
void main() {
            Derived d1;
            d1.print();
}
```

Output :
Derived function

→ To access the overridden function of the base class, we use the scope resolution operator (::).

→ Program → same as the ~~old~~ previous.

```
void main()
{
  Derived d1, d2;
  d1.Print();
  d2.Base::Print();
}
```

Output :-
Derived function.
Base function.

→ We can also access the overridden function of the base class by using a pointer of the base class to point to an object of the derived class and then calling the function from that pointer.

```
#include <iostream>
using namespace std;
class Base
    {
    Public:
        void Print()
            {
                cout<< "Base function" << endl;
            }
    };
class Derived : Public Base
    {
    Public:
        void Print()
        {
```

```
cout<< "Derived function" << endl;
}
void main()
   {
   Base *P;
   Derived d1;
   P= &d1;
   P->Print();
   }
```

Output:

Base function

→ Description:-

1- Here we create a Pointer of base class.
2- Then we creat an object of derived class.
3- Then we assign the derived class object to base class Pointer. Now the P= &d1; point to base class function
4- Now we call the Print() using base class Pointer

Pointer

→ In c++ language, the pointer is a variable which is used to store the address of another variable.

→ The variable can be of type int, float, char etc.

→ The pointer in c++ language can be declared using * (dereference operator).

→ Syntax to declare pointer:-

<data-type> * <variable name>;

int *mark;

→ We can initialize a pointer by

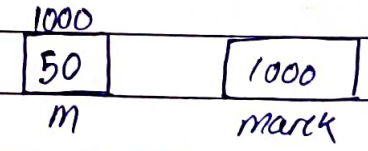
<pointer variable name> = &<variable-name>

p = &a

→ & - Reference operator

→ * - Dereference operator

```
#include<iostream>
using namespace std;
void main()
 {
  int *mark, m;
    m=50;
    mark=&m;
cout<<" Address of m is : "<<mark;
cout<<" value of m is : "<<*mark;
}
```



Output :-

Address of m is : 1000

value of m is : 50

Pointers to objects:-

→ By using a pointer we can store the address of an object of a class.

→ If we are using a pointer to point or store the address of an object of a class, then this is called. Pointer to object.

Syntax:-

```
<class_name> *<Pointer_name>;
<class_name> <object_name>;
<Pointer_name> = & <object_name>;
```

example:

```
class *Ptr;    // creating Pointer object
class obj;     // creating object
Ptr = &obj;    // Initializing pointer to an object.
```

Program:-

```
#include <iostream>
using namespace std;
class B
 {
  int a, b;
  Public:
      void getdata()
        {
          a=10;
          b=20;
        }
      void Putdata()
        {
    cout<<" a= "<<a<<endl<<"b="<<b<<endl;
        };
void main()
```

```
{
    B obj;
    B *P;
    P = &obj;          // Pointer to an object.
    P->getdata();      // when we use to call function by
    P->putdata();      //Pointer we have to write like this.
};
```

Output : a=10
b=20

This Pointer :-

→ The this Pointer holds the address of current object, in simple words we can say that this pointer points to the current object of the class.

→ we can pass this Pointer as an argument to the member function of a class implicitly, to access the address of the object of the same class.

→ By using this Pointer compiler can easily differentiate between local variable and data member of a class if both have same name.

Program :-

```
#include<iostream>
using namespace std;
class test
   {
    int x;
    Public
     void setx(int x)
     {
```

```
this -> x = x;
}
void print ()
    {
    cout<< "x = " <<x<< endl;
    }
};
void main()
  {
   test obj;
   obj. setx (20);
   obj. print();
  }
```

Pointers to derived class :

→ A pointer to derived class is a pointer of the base class that is pointing to the object of a derived class.

→ This pointer of base class will be able to access the functions and variables of its own class and can still point to derived class object.

→ By using base class pointer we can only access the overridden member function of derived class by making that function virtual.

Program :-

```
#include<iostream>
using namespace std;
class base
        {
        public:
```

```
int n1 = 10;
virtual void show()
    {
    cout<< "N1 = " <<n1<<endl;
    }
};
class derived : public base
{
  public:
  int n2 = 40;
  void show();
    {
    cout<<" N1 = "<< n1 <<endl;
    cout<< " N2 = "<< n2 <<endl;
    }
};
int main()
  {
  base b;
  base *bptr;  // base pointer
  bptr = &b;  // address of base class
  bptr -> show(); // access base class member by base class
                  //pointer
  derived d;
  bptr = &d;  // pointer to derived class.
  bptr -> show();
  }
```