Unit - I st

OOT ⇒ OOT means object oriented technology. It is a s/w design model in which objects contain both data & instruction that work on the data. It is increasingly deployed in distributed computing.

OOP ⇒

OOP means object oriented programming. It is a solution to programming problems which offers a powerful model to develop computer s/w. It is a programming style that captures the ~~behavi~~ behavior of real world. OOP approach hides implementation details.

OOT Vs OOP ⇒

OOT is a s/w design model in which objects contain both data & instruction that work on the data. OOP is a programming paradigm based on the concept of "objects" which may contain data (in the form of field known as attributes) and code (in the form of procedures known as method).

Language ⇒ Language is a medium which communicate to each other.

Computer language :- A language communicate to user with computer is called computer language.

Programming language :- The language used in the communication of instructions to the computer is known as. computer or programming language. There are many types of

Programming language available
Ex:- c/c++/Java/dot net etc.

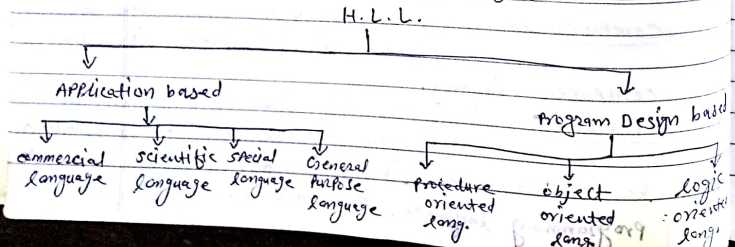
The process of writting of instructions using a computer language is known as Programming or coding.

High level language :- It is designed to be used by the human operator or programmer. It is easy ~~under stands~~ understandable ~~by~~ for user.
Every single program written in a high level language must be interpreted into machine language.

Machine language (low level language) :-

Machine language is the lowest (low)-level programming language. It is very difficult for user understandable. It is not required for compiler & interpreter. It is written in binary or hexadecimal instruction code. ~~con~~ It is direct communicate with machine or computer.

Classification of High level language (H.L.L) ⇒



* Application based :-
Commercial language :- It is mainly used in business related problems. It is related to payroll, accounts payable & tax handling.
Ex:- COBOL, PL/C (Programming lang. for business)
Scientific language :- It is mainly used in scientific & mathematical problems. It provide complex calculation based programming. Ex:- FORTRAN.
Special-Purpose language :- It is mainly used for performing some specific functions Ex:- SQL
General-purpose language :- It is mainly used for developing different application slws Ex:- C/c++/Java.

* Design based :-
Procedure oriented language :- It is also called imperative programming languages (instruction base).
In these languages program is written in sequence of procedure. Each procedure contains of series of instructions for performing specific task.
Ex:- BASIC, ALGOL, FORTRAN, C
Object oriented language :- It is mainly based on object oriented for solving given problems. It is divided a problem into number of objects.
Ex:- c++/Java/c#
Logic oriented lang:- It is mainly logic based approach for solving various computational problems. It is mainly used predicates, logic, rules & facts.
Ex:- LISP, PROLOG.

Object :-

Real world entity is called object. Means Object having these characteristics identity (identification), behaviour & state.
Identity means uniqueness, behaviour means task/work & state means solid/liquid/gess.

(OR)

objects are the basic run time entities in an object oriented system.

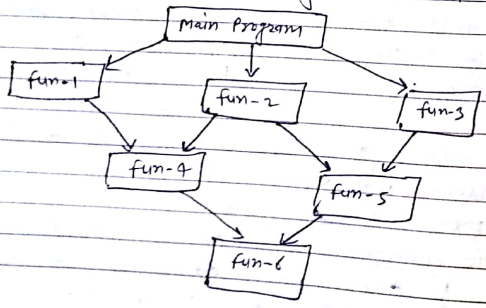
Example of object :- person, car, etc

Object orientation :-

Object orientation is about tring to represent the objects that we find in the real world in s/w.

Procedure Oriented Programming ⇒

It is basically consists of writting a list of instructions for the computer to follow & organising these instructions into groups known as functions.

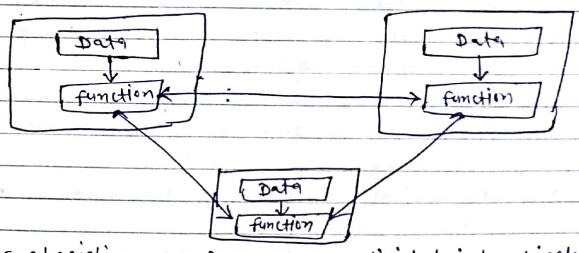


characteristics ⇒

* large programs are divided into smaller programs known as function.
* Most of the functions share global data.
* Data move openly arround the system from function to function.
* functions transfer data from one to another form.
* It is Top-Down approach.
* new data & function cannot be easily added.

Object Oriented Programming ⇒

It remove the some problems in procedure oriented programming approach. These programming not allow the data movement is freely arround the system. These allows decomposition of a problem into a number of entities is called object & then builts data & functions arround these objects.



characteristics :- * programs are divided into objects
* Data is hidden & can not be accessed by external func^{n}
* object may communicate with each other through functions
* new function & data can be easily added.
* It is bottom-up approach
* most of functions shared local data.

Need of OOP ⇒ There are mainly two reason that why we need oop.

* Duplicate code is bad.
* code will always be changed.

OOPs is provides code reusability which reduce the duplicate code. you have make changes every where which leads to performance. code can be changed anytime or requirement of application changed anytime so when you want to make changes in your application. OOPs makes it easier.

Merits (Advantages) of OOPs ⇒

* Through inheritence, we can eliminate redundant code & extend the use of existing classes.
* It is possible to have multiple instances of object to co-exist without any interface.
* object oriented system can be easily upgrated from small to large systems.
* S/w complexity can be easily managed.
* Message passing techniques for communication b/w objects make the interface.
* Data hidding helps the programmer to build secure programs.

Applications of OOPs ⇒

* Real time systems
* simulation & modeling.
* object oriented data base
* AI & expert system
* Hyper text, Hyper media & expert text.
* Neural N/w & Parallel programming
* CAD/CAM systems.
* Decision support & office automation system.

Benefits of OOPs / features of OOPs ⇒

These are include following features.

* class
* object
* Data abstractions & encapsulations
* inheritence
* Polymorphism
* Dynamic binding
* Message passing

Difference b/w POP Vs OOP

POP (Procedure oriented programming)	OOP (object oriented progr)
* TOP-Down Programming approach	* Bottom-up programming approach
* Large programs are divided into the functions	* Programs are divided into the objects.
* Data move openly arround the system.	* Data are tied more closely to the functions
* functions transforms data from one form to another	* Data structures are designed to characterise to objects
* Emphasis is on doing things (algorithms)	* Emphasis is an data rather than procedure.
* Many important data items are placed as global	* Data is hidden & cannot be accessed by external functio
* Ex:- C/FORTRAN/COBOL	* Ex:- C++/Java/small talk

Difference	object based approach	object oriented approach
*	Major features required. Data encapsulation, data hiding, operator overloading, Automatic initialization, Access mechanisms, clear up of object, object identity	* object-based features + inheritance + Dynamic binding
*	Doesnot support reusability	* supports reusability
*	Concurrency difficult	* Concurrency poor.

Limitation (Drawback) आगे Page No. 14 पर है

8

* Garbage collection does not find	* It may have garbage collection
* support abstraction data but does not support classes	* supports both abstract data & classes.
* object library does not match	* It may contain object library
* Ex:- Ada, Modular-2	Ex:- Java, small talk, simula

OOPs features:-

Object :-

Object are the instance of a class. These are the variable of type class. Object is the basic run time entity in an object oriented system.

class:-

A class is a collection of objects of similar type.

Data abstraction & encapsulation ⇒

Abstraction refers to the act of representing essential features without including the background details or explanations.

The process of wrapping data & functions into a single unit is called encapsulation.

Inheritance ⇒

Inheritance is the process by which object of one class aquire the properties of object of another class.

Concept of Inheritance provide the idea of reusability.

9

Polymorphism ⇒

Polymorphism means one name, multiple forms means the ability to take more than one form.

It allows us to have more than one function with the same name in a program.

Dynamic binding ⇒

Dynamic binding means that the code associated with a given procedure is not known until the time of the call at run time.

Message passing ⇒

Message passing involves specifying the name of the object, the message & information to be sent.

Message passing is the process of object oriented programming through which objects communicate with one another by sending & recieving information.

Abstract data types (ADT) ⇒

Abstract is the process of extracting the relevant properties of an object while ignoring non essential details.

Since the classes use the concept of data abstraction they are known as abstract data types.

c++ use abstract data types are implemented as classes.

Properties:-

* It provides an interface without exposing the implementation details.
* It exports a type with set of operations.
* It defines the application domain of the data type for the data.

* It defines attributes & methods which implement operations.

Tokens:-

The smallest individual units in a program are known as tokens. There are following types:-

- Keywords
- Identifiers
- Constants
- strings
- operations

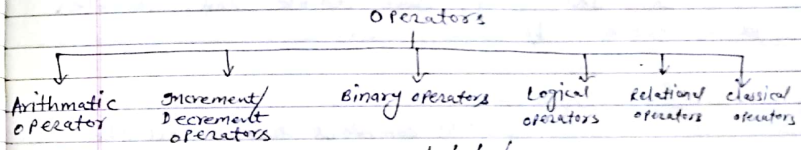
* Keywords:- The keywords implemented specific in C++ language features. It is explicitly reserved identifiers & can not be used as names for the program variables or user defined program elements.
Ex:- char, int, float, for, DO, while etc.

* identifiers:- Identifiers use the name of variable functions, arrays, classes, created by programmer.
Rules for naming identifiers:-
⇒ only alphabetic characters, digits & under scores are permitted.
⇒ It name can not be start with a digit.
⇒ Upper & lower case letters are distinct.
⇒ The declared keyword can not be used as a variable name (identifier name).

* Constant:- A constant is a value that can not be change during Holl program execution.
(or)
The value of a constant cannot be changed during program execution.

* String:- A string is a sequence of character
(or)
A string is a continous sequence of symbols or value such as character string or binary digit strings. Ex:- "RAM" or "10101"

* Operators:- A symbol that represents a specific action is called operator. An operator acts on different data items/entities called operands.
Ex:- a+b where
a & b are operand & + is a operator
These are following types of operators



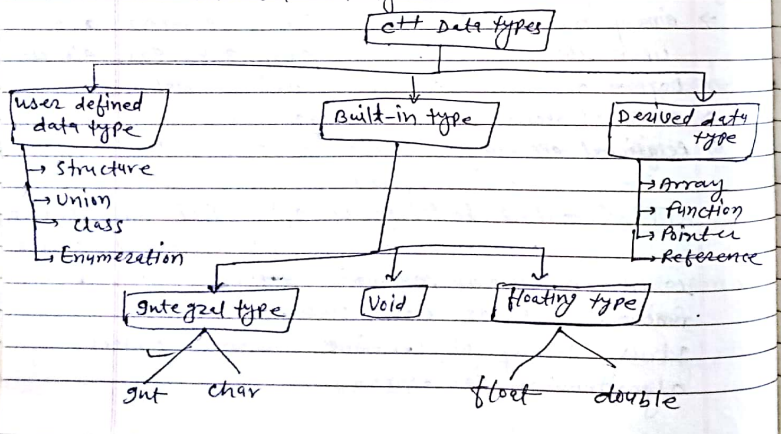
⇒ Arithmatic operators:- +/-/*/1/%.
⇒ Increment/Decrement operators:- ++/--
⇒ Binary operators:- The Binary operators act upon two operands. Ex:- a+b, a-b, a*b, a/b etc
⇒ Logical operators:- And(&&) / OR(||) / Not (!)
⇒ classical operators:- True/false
⇒ Relational operators:- </>/<=/>=/==/!= or <>

STL (Standard template Library):- (मानकीकृत formate) STL is a set of c++ template classes to provide common programming data structures & functions such as lists, stacks, array etc.
It is a library of containers container classes, algorithms & iterators.

components of STL
- Algorithms
- containers
- functions
- Iterators :- Iterators are used for working upon a sequence of values.

Variables :- Variable is a memory area in the RAM which is allocated memory for our used.
Each variable consists of it's type & name,
EX:- int, float, char, double, long double,
Ex:- → int a , float a, char a

Data types :-
c++ compilers support all the built in data types known as basic or fundamental data types.



Data type Range & memory size :-

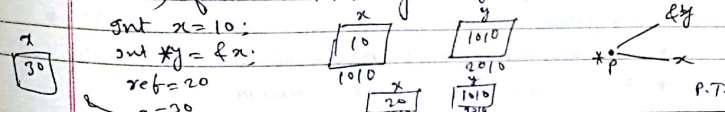
data type	Memory size	Range
int	2 byte	-32766 to 32767
char	1 byte	-128 to 127
float	4 byte	3.4×10^{-38} to 3.4×10^{38}
Double	8 byte	1.7×10^{-308} to 1.7×10^{308}
long double	10 byte	-2,147,483,648 to 2,147,483,647

Manipulators :-
Manipulators are operators that are used to format the data display. The most commonly used manipulators are Endl & setw.

User Defined data types :-
We have used user defined data types such as struct & union in C But in c++ new user defined data types like class.

Reference Variables :-
c++ introduces a new kind of variable known as the reference variables. A reference variable provides an alias (alternative name) for a priviously defined variables.

```
int x=10;
int *y = &x;
ref=20
```



Enumerated Data type :-

An Enumerated data type is another user defined type data type which provides a way for attaching a names to numbers.

(or)

An Enumeration is a method in which a small group of related objects can be put in a set and given a desired name.

"Enum" is a keywords that is used for enumerated data types declaration & it is automatically enumerates a list of words (elements) by assigning the value such as 0, 1, 2, 3, - - -

EX:-

```
Enum weekday {Mon, Tues, wednes, Thurs, Fri, Sat, sun};
enum colour {Black, Red, white, yellow, blue};
enum shape {circle, Triangle, rectangle};
```

Limitations (Drawbacks) of Structure Programming :-

"structured programming :- In structured programming program has a fixed well defined structured"

* Structured programming has reduced data security & integrity.
* structure programming does not support inheritance because it does not have the concept of class & object.
* structure programming approach does not support data encapsulation due to lack of class.
* It does not support Polymorphism.
* It does not support data abstraction.

Limitation (Drawback/difficulties) of procedure oriented prog. (POP)

* Global data are more weak (poor).
* Data moves openly arround the program.
* It is very difficult to varify what data is used by which function, in a large program.
* functions are oriented to their action only.
* It does not model real world problems very well.
* Portion of code is interdependent.
* code in one application can not be used in another program.
* Data are transformed by function from one form to another.

Limitation overcome by OOPs :-

* Decomposition, information Hiding, Reusability, maintenance, Evolutionary life cycle.

Objects as a S/w (Software) Units/modules :-

* In OOP problem is divided into objects.
* It allows us to decompose a problem into number of entities called objects.
* objects are the units of code that are eventually derived from the process.
* object represents real-world entities.
* object occupies memory space & has an associated address.
* Object communicates by sending message to one-another.
* Object consists of both attributes & operations.
* In Problem domain, object plays a well defined role.
* The set of methods, defines the behavior of an object.
* The state of these basic run time entities can be change according to the methods.

Data encapsulation & information (data) Hiding ⇒

Encapsulation is a mechanism that associates data & function (code) into a single unit & keeps them safe from misuse & external interference (OR)
In other words:-
wrapping up of data & function into a single unit is known as data encapsulation. This unit may be class. This important feature of class.

Data hiding means hiding the data from parts of the program that do not need to access it. Insulation of the data from direct access by the program is called data hiding or information hiding.

⇒ Thus use to encapsulation to protect the data & function of a class from unauthorised access is called data or information hiding.

Significance:-
* Data hiding helps to achieve encapsulation
* Data hiding protects
* Data hiding puts data in a class & makes it private.
* It makes hidden data of one class from the other class.

Object lifetime (Live cycle):- In OOP, The object lifetime of an object is the time between an object's creation & it's destruction (distroy).

Object Instantiation & Interactions:-

creation (function) message passing

The creation of an instance is called instantiation. In class-based programming, objects are created from classes by subroutines called constructors.
An object is an instance of a class and may be called a class instance or class object.
Instantiation is then also known as construction.

Objects of two different classes in object oriented programming interact with one another by sending & receiving messages.
In object oriented programming concept, objects communicate with each other through creating classes & objects & establishing communication among them.

Object:-

Object are real world entity which contains three properties are identification, behavior & state. Identity means that data is quantised into discrete, distinguishable to other data.
Types of objects:-
- External object (global object)
- Automatic or local object
- Static object
- Dynamic object

* External object:-
Existence:- Through out the life time of the whole program.
Visible visibility:- Globally available to all modules.

* Automatic or local objects :-
  ⇒ within the local scope.
  ⇒ Local scope in which objects are created.
* Static object :-
  ⇒ Through out the whole program
  ⇒ Local scope in which objects are created.
* Dynamic objects :-
  ⇒ Life time may be controlled within a particular scope.
  ⇒ Visibility within a particular scope.

State of object :-

Types of object based on their state.
- Active object
- Passive object

* Active object :- An object which has it's own thread of control is known as active object.
* Passive object :- An object which does not encompass it's own thread of control is known as passive object

Difference b/w Active object & Passive object

Active object	Passive object
* Active objects can exhibit some behavior without being operand upon by another object.	Passive objects can only undergo a state change when explicitly acted upon. It is only operated upon by other objects.
* Active object serves as the root of control	Passive object never operates on other objects.
* If our system involves multiple threads of control, then we will have multiple active objects.	In sequential architecture system, all other objects are passive & their behavior is triggered by one active object.

Unit - II

Modeling :-

A model is an abstraction of some things for the purpose of understanding it before building it because a model omits non-essential details. It is easier to manipulate than the original entity

importance of modeling :-
* testing a physical entity before building it.
* communication with customer.
* visualization
* Reduction of complexity.

OOM (Object oriented methodology) :-

We present a methodology for object-oriented development & a graphical notation for representing object oriented concepts.

The methodology consists of building a model of an application domain & then adding implementation details to it during the system design. We call this approach OMT (object modeling technique)

The methodology has the following stages :-
* Analysis, system design, object design, Implementation

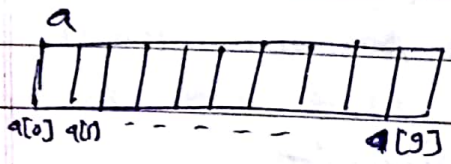
* Analysis :- Starting from a statement of the problem The analyst builds a model of the real world situations showing it's important properties.
* System design :- The system designer makes high-level decisions about overall architecture. During system design the target system is organised into subsystems based on both the analysis structure and proposed architecture.

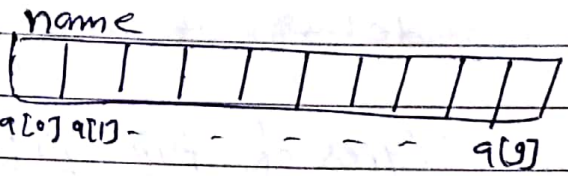
Unit-III

Array :-

Array is the collection of similar types of data.

Ex:- int a[10]; char name[10];



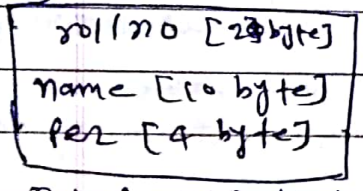


Structure & union :-

Structure & union both are the collection of different types of data.

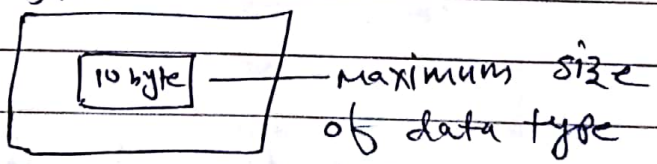
Ex:.

```
Struct gnfo
  {
  int rollno;
  char name[10];
  float per;
  };
```



Total = 16 byte memory space

```
union gnfo
  {
  int rollno;
  char name[10];
  float per;
  };
```



* Structure contain one/more data items together as a single unit but not a function.
* Structure elements are default as a public
* It is not provide security

Function :-

A complex problem may be decomposed into small or easily manageable parts or modules are called function.

function contain the set of instructions or code.

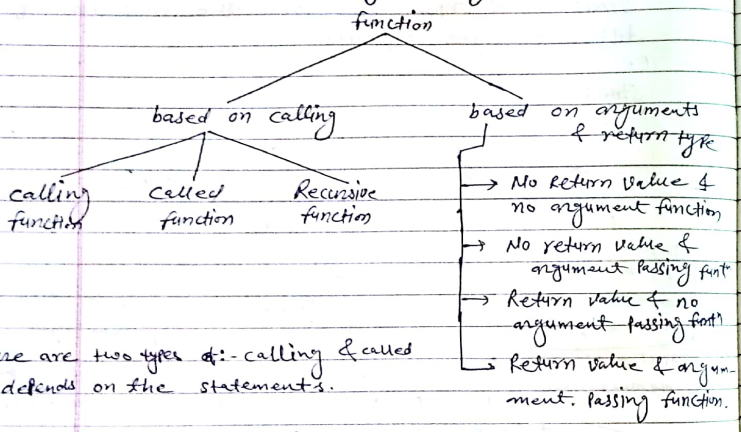
P.T.O

Need:-

Functions are useful when we need to divide large & complex programs into small parts. Functions are very useful to read, write & debug & modify complex problems.

Types of function (-Classification) :-

There are mainly two types of function

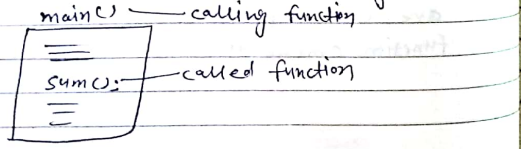


There are two types of:- calling & called depends on the statements.

Based on calling:-

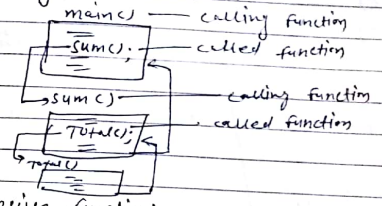
Calling function:-

If the function have calling statements that it is known as calling function.

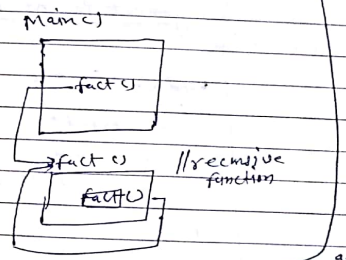


Called function:-

Any function called by other other function known as called function. Any called function again become a calling function function if it contain the call the statements of any other function



Recursive function:-




```
Void main()
{
  int num, sum;
cout<<"Enter number";
cin>>num;
int fact(int);
sum=fact(num);
cout<<" factorial "<<sum;
}

int fact (int n)
{
if (n==1)
return 1;
else
{
n=n* fact(n-1);
return n;
}
}
```

P.T.O.

Based on arguments & return type:-

* **No return value & No argument passing function:-**

Ex:- Void sum()

```
Void main()
{
Void msg();
int a, b, c;
msg();
cin>>a;
msg();
cin>>b;
c=a+b;
cout<<"sum"<<c;
}
```

```
Void msg()
{
cout<<"enter the number";
}
```

* **No return value & Argument passing function ⇒**

Ex:- Void sum(int, int);

```
Void main()
{
Void sum(int, int);
int a, b;
cout<<"enter Ist no.";
cin>>a;
cout<<"enter IInd no.";
cin>>b;
sum(a, b);
}
```

```
Void sum(int x, int y)
{
int total;
total = x+y;
cout<<"total"<<total;
}
```

* **Return value & no argument passing function ⇒**

Ex:- int sum();


```
Void main()
{
int a, b, c;
int value();
a= value();
b= value();
c= a+b;
cout<<c;
}
```

```
int value()
{
int N;
cout<<"enter number";
cin>>N;
return (N);
}
```

* **Return value & Argument passing function:-**

Ex:- int sum(int, int);

```
Void main()
{
int a, b, c;
cout<<"enter no";
cin>>a>>b;
int sum(int, int);
c= sum(a, b);
cout<<"sum"<<c;
}
```

```
int sum(int x, int y)
{
int s;
s= x+y;
return (s);
}
```

Function Prototyping:-

Function prototype is a declarative statement in the calling function.

Ex:- <return_type> <functionname> (Parameter list);

int sum(int a, int b);

P.T.O.

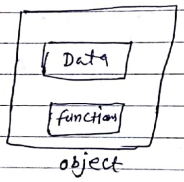
Class & object :-

Object is the instance of the class. Object is a bundle of s/w containing data & code to operate on those data (function). object are the basic run-time entities of the programming based on the object orientation.

Class is the collection of objects with similar type.

(or)

class is the collection of data & functions.



object

Aggregation components of object :-

There are following components :-

* Data members
* Member functions
* Static data members
* static member functions } आगे describe
* Constant member functions

* Data member :-
The data member of class describes the class data.

* Member function :-
The member function describes the class behaviour. It also defines the operations that can be performed on the class data members.

The functions & variables are collectively called class members.

Class specification (definition) :-

A class definition is a process of naming a class data variables & functions. It has two parts.

- class declaration
- class function definition

The class declaration describes the type & scope of data members & member functions. function of a class describes how the class functions are implemented.

Class specification consists of the following steps.

- class definition
- internal representation of memory
- internal operations to implementation the interface
- External operations for accessing & manipulating the objects of the class.

Class declaration :-

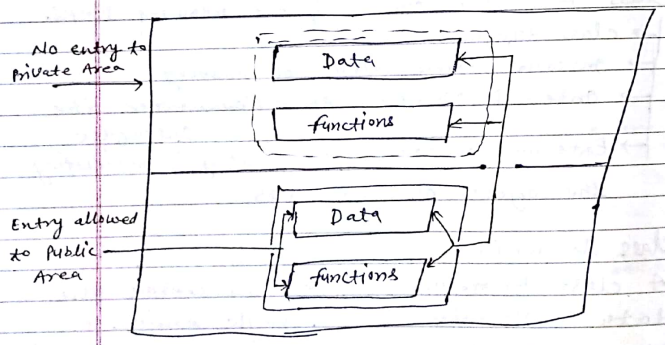
A class declaration is used to combine the data & operation into a single entity. It specifies the representation of objects of the class & the set of the operations that can be applied on these objects.

It consists of two sections

- Private or protected section
- public section

Private or Protected section holds data & Public section holds the interface operations. Private, Protected & Public are keywords. These are known as access specifiers.

```
class <class_name>
{
Private:
        Variable declaration;      — data member
        function declaration;      — function member

Public:
        Variable declaration;      — data member
        function declaration;      — member function
};  — termination of class (class ending)
```



Types of classes ⇒

There are following types of classes.

* Abstract class:-
  A class that is not used to create object.
* Generic class:-
  A class that serves as a template for other classes.
* Container class:-
  A class whose instances are collection of other objects.
* Object class:-
  A class which do not explicitly extend any other class.
* Super class:-
  A class which has one or more members which are classes themselves.
* Base class:-
  A class from which other classes are derived.
* Derived class:-
  A class that inherits properties of the base class.

~~Attributes & methods~~ ⇒

Static Data Member ⇒

The static data members access control to some shared resources. They normally used to maintain values common to the entire class. Each object has it's separate set of data member in memory. The memory functions are created only once & all objects share these functions. No separate copy of functions of each object is created in memory.

It's effect are same in Private / Public section

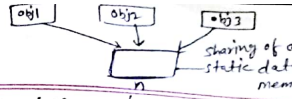
⇒ "The main concept of static data members are data objects of a class. They exist only once in all objects of this class".

⇒ The main advantage of static data member may be helpful to declare the global data which should be updated while the program exists in the memory.

Ex:-
```
class CS                                   Private
{
 static int n; //static data member by default
 public:
  void increment()
    {
    n++;
    cout<< "value of n"<<n;
    cout<<" Address of n"<<&n;
    }
};
int CS::n;   // static member variable is initialized to zero
                or
             int CS:: n=20;
             //initialized with 20 not zero
void main()
{
CS obj1, obj2, obj3;
obj1.increment();
obj2.increment();
obj3.increment();
}
```

O/P

Value	Address
n=1	0X16C70076
n=2	—11—
n=3	—11—



Characteristics of static data member ⇒

* Static member variables are normally used to maintain values common to the entire class.
* static member variables are visible within the class but it's life time is the entire program.
* Since they are associated with the class itself rather than with any class object, they are also known as class variables.
* Type & scope of each static member variable must be defined outside the class definition because the static data members are stored separately rather than as a part of an object.
* only one copy of that member is created for the entire class & is shared by all the objects of that class.
* static member variable is initialised to zero when the first object of it's class is created, no other initialization is permitted.

Static member function ⇒

The static member functions used to pre-initialize private static data members before any object is actually created. Like data members.

functions can also be declared as static.

static member functions can access only static data members & functions of the same class. The non static data members are not available to these functions.

P.T.O.

Features:-

* The static member function declared in public section can be invoked using it's class name without using it's objects.
* Just one copy of static member function is created in the ~~memb~~ memory for entire class. All objects of the class share the same copy of static member function.

Ex:-

```
class cs
 {
  static int n;   //static data member
public:
   static void increment()
     {
      n++;
     }
  static void show()
    {
     cout<<" value of n = "<<n;
    }
};
int cs::n;  //static data member automatic initialized with zero
void main()
 {
  cs obj1, obj2, obj3;
  cs::show();  // static function can be called using ^ class name
  cs:: increment();
  cs:: increment();
  cs:: show();
 }
```

O/P → value of n = 0
value of n = 2

Constant (const) member function :-

* The constant objects can access only constant member functions.
* when we declare a member function as a const. it can not alter any data value in the class.
* we can also make an object as constant using const keyword.
* The data member of constant object can only be read & act as a read only data member.
* Only constructor can initialize data member variables of constant ~~object~~ object.
* member function does not alter any value inside the class & it can be declare as follows:-
  <return type> <function_name() const>

Ex:-

```
class cs
 {
 int a;
 Public:
  cs (int m)  // Parameterised constructor
   {
    a=m;
   }
 Void show() const // constant member function
   {
    a++;
   cout<<" value of a = "<<a;
   }
};
Void main()
 {
 const cs obj(5);  // constant object
  obj.show();
 }
```

O/P value of a = 6

Class access modifiers :- → Private
→ Public
→ Protected

Private access modifiers :-
Private access mode specifies the members which are declared inside the private section which are not accessable of outside function.

Public access modifiers :-
The public section is specifies that all the variable & functions are accessable outside function.

Protected access modifiers :-
This section is used in inheritance concept.

Member functions :-

Class member functions defining in two ways :-
→ Outside the class definition
→ Inside the class definition

* Inside the class definition :-
when a member function is declared & define inside a class is called inside the class definition.

program page 51 पर है

class xyz
{
void ... // member function
};

* Outside the class definition ⇒
when we have declare a class member function or data declare outside the class.

Ex
class xyz
{
...

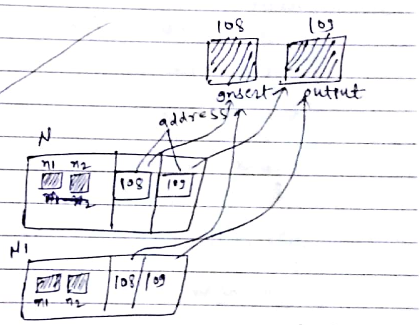
---

more class बनी है तो cursor को confusion ना हो यही class के बाहर कोई member function को declare करेंगे तो resolution operator का use करेंगे

Scope Resolution Operator (::) ⇒

Scope resolution operator is used when we have to declare class declare data member or member function outside the class.

Ex :-
```
class num
{
 int n1, n2;
public :
 void insert ();
 void output ();
};

void num :: insert ()   // member function
{
 cin >> n1 >> n2;
}
void num :: output ()
{
 cout << n1 + n2;
}

void main ()
{
 Num N;
 N.insert ();
 N.output ();
 Num N1;
 N1.insert ();
 N1.output ();
}
```



Scope resolution operator use करने से member function (insert & output) के लिए memory space class के object N & N1 में space allocate नहीं होगा only member function का address store होगा

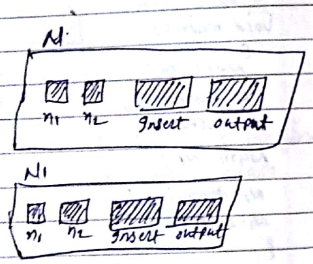
* Scope resolution (::) operator is a binary operator to excepts two arguments from user,
Ist argument is given at the LHS & it's a class name
IInd argument is a function or data member name.

* Scope resolution operator के LHS side की तरफ class का नाम लिखते हैं ताकी more class वाली हो तो cursor को confuse न हो यदि class के बाहर कोई member function को declare करेंगे तो scope resolution operator का use करेंगे।

Note:- Declaration हमेशा address store करता है। scope resolution class का address रखता है की membership function किस class का है।

Without scope resolution operator (inside member function)

```
class num
{
int n1, n2, sum; //data member
public:
 void insert()  //member function
  {
  cin>>n1>>n2;
  }
 void output()
  {
  cout<<"sum"<<(n1+n2);
  }
};
void main()
{
num N, N1;
N.insert();
N.output();
N1.insert();
N1.output();
}
```



इसमें insert & output member function का memory allocation object के अन्दर (inside) ही होता है दोनो object में कि अलग memory allocation होता है।

Characteristics of ~~scope~~ function scope resolution operator (outside)

* These functions do not return any value their return type is void.
* several different classes can use the same function name. The membership label will resolve their scope.
* Member functions can access the private data of the class. A non member function can not do so.
* A member function can call another member function directly without using dot (.) operator.

Nesting of member function :-

A member function of a class can be called only by an object of that class using a dot (.) operator.

⇒ When a member function can be called by using it's name inside another member function of the same class is called nesting of member function.

[इससे benefit ये है कि इसमे एक function से को main function में नहीं कॉल (call) करना पड़ता मानो एक calling कम करना पड़ती है।]

Ex:-

```
class set
{ int m, n;
public:
 void input();
 void display();
 int largest();
};
int set::largest()
{
 if(m>=n)
  return(m);
 else
  return(n);
}
void set::input()
 {
 cout<<"input values of m&n";
 cin>>m>>n;
}
```

```
Void set :: display ()
{
cout << "largest value" << largest ();
}
int main()
 {set A;
 A.input();
 A.display();
 return 0;
}
```

calling member function

Inline Function ⇒

Inline functions are those function which copies its contains in calling statements.
[जहाँ पर call होते हैं वहाँ पर सारे contain information copy कर देता है।]

The member function of class are by default are known as inline function.

Inline function से advantage यह है की cursor को बार-बार member function में नहीं भागना है यह member function के data को copy कर लेता है यह execution time कर देता है लेकिन memory space बढ़ा देता है

Ex:
```
inline void msg ()
{
cout << "15|10|2018";
cout << "Thursday";
}

Void main()
{
msg();
msg();
msg();
}
```


Friend function [Alternative for scope resolution operator]

Friend functions are those function which are able to recieve the object of class & able to work with private section.
To declare any outside function as a friend, we need to provide it's friend declaration inside the class using friend keywords.
After becoming a friend function is able to work with a private section.
TO call a friend function to did not it to specifies the object name with dot (.) operator but we need to provide object as an arguments.

```
class ABC
{
Public:
friend Void XYZ();
};
```

friend is a keyword.

Example

```
class employee
{
char name[10];
char address[40];
int code, salary;
Public:
Void insert()
{
cout << "Enter name";
gets(name);
cout << "Enter address";
gets(address);
cout << "Enter code";
cin >> code;
cout << "Enter salary";
cin >> salary;
}
friend void display (employee);
};
void display (Employee E)
{
cout << E.name;
cout << E.address;
cout << E.code;
cout << E.salary;
}
void main()
{
Employee E;
E.insert();
display(E);
}
```

Ex:-
```
class sample
{
int a,b;
public:
void setValue()
 {
  a=25;
  b=40;
 }
friend float mean (sample s);
};
float mean (sample s)
{
return float (s.a + s.b)/2;
}
```

```
void main()
{
sample X;
X.setValue();
cout<<" mean ";
cout<<mean(X);
}
```

Characteristics ⇒

* It is not in the scope of the class to which it has been declared as friend.
* Since it is not in the scope of the class, it can not be called using the object of that class.
* It can be invoked like a normal function without the help of any object. (नाम/उपयोग नहीं)
* It can declared either in the public or private section of a class, without affecting it's meaning.
* Usually it has the objects as arguments.
* Unlike member function, it can not access the member name directly & had to use an object name & dot membership operator with each member name (Ex:- A.x)

Friend class:-

We can declare all the member functions of one class as the friend function of another class these class is called friend class.

```
class ABC; // forward declaration
class XYZ.
{
int x;
public:
void setValue(int i)
 {
  x=i;
 }
friend void max(XYZ, ABC);
};
class ABC
 {
  int a;
public:
void setValue(int j)
 {
  a=j;
 }
friend void max(XYZ, ABC);
};
void max(XYZ m, ABC n)
 {
 if (m.x >= n.a)
  {
  cout<<m.x;
 else
 cout<<n.a;
}
```

```
int main()
 {
  ABC abc;
abc.setValue(10);
XYZ xyz;
xyz.setValue(20);
  max(xyz, abc);
 }
```

O/P 20

P.T.O.

⇒ Use common friend function to exchange the private value of two classes.

Ex

```
class CS-2;
class CS-1
{
int value;
public:
void gndata(int a)
 {value=a;
 }
void display()
 {
cout<<"value"<<value;
 }
friend void exchange(CS-1 &, CS-2 &);
};

class CS-2
 {
 int value2;
 Public:
 void gndata(int a)
 {
 value2 = a;
 }
 void display()
 {
 cout<<"value2"<<value2;
 }
friend void exchange(CS-1 &, CS-2 &);
};
void exchange(CS-1 &x, CS-2 &y)
 {
 int temp = x.value;
 x.value = y.value2;
 y.value = temp;
 }
```

```
void main
{
CS-1  C1;
CS-2  C2;
C1.gndata(100);
C2.gndata(200);
cout<<" value before
      exchange";
C1.display();
C2.display();
exchange(C1, C2);
cout<<"value after
      exchange";
C1.display();
C2.display();
}
```

O/P

value before exchange
100
200

value after exchange
200
100

⇒ Object returning program ⇒

Function can not recieve object as argument but also can return them.

Ex:-

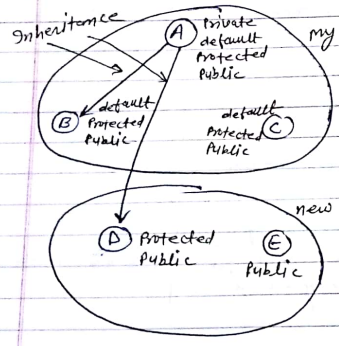
```
class complex
 {
 float x, y;
 public:
 void gnput(float r, float i)
  {
  x=r;  //real
  y=i;  //Imaginary
  }
friend complex sum(complex, complex);
 void show(complex);
 };

complex sum(complex C1, complex C2)
 {
 complex C3;
 C3.x = C1.x + C2.x;
 C3.y = C1.y + C2.y;
 return(C3); //object return
 }                value
Void complex :: show(complex C)
 {
cout<<C.x<<"+"<<C.y<<"i";
 }

Void main()
 {
complex A, B, C;
A.gnput(3,5);
B.gnput(4,7);
C = A sum(A,B);
```

```
cout<<" A = "<<
 A.show(A);
cout<<" B = "<<
 B.show(B);
cout<<" C = "<<
 C.show();
}
```

C may or not

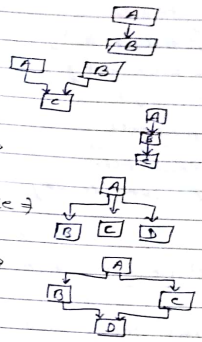
Inheritance (वंशागति) :-

Inheritance is an ability of reuse when we use the class data member & member function in another class. These concept is known as inheritance.
when any class is inherited that is known as with base class. & the class which inherites the properties of base class is known as derived class.
Derived class is able to access the public & protected section of base class. Protected section is that section which is exeeble by derived class not any outside class.



Types of inheritance :-

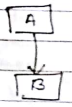
(1) Single inheritance ⇒
(2) multiple inheritance ⇒
(3) multilevel inheritance ⇒
(4) Hierarchical inheritance ⇒
(5) Hybrid inheritance ⇒



* Single inheritance ⇒

It is used when we have a single base class & single derived class

Ex:-
```
class info
{
Protected: int code;
           char name[20];
Public:
Void insert()
{
cout<<"Enter the code"; cin>>code;
cout<<"Enter the name"; gets(name);
}
Void output()
{ cout<<" code "<<code;
cout<<" name "<<; puts(name);
}
};
```



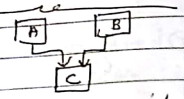
use for inheritance

```
class employee: Public info
{
float salary;
```

```
Public:
Void getdata()
{
Info :: insert();
Cout<<" enter salary ";
Cin>> salary;
}
Void putdata()
{
Info :: output();
Cout<< "salary "<<salary;
}
};
```

```
Void main()
{
Employee E;
E.getdata();
E.putdata();
}
```

* Multiple Inheritance ⇒



In this inheritance, we have more than one base classes for single derived class.

Ex:

```
Class Info
{
Protected:
Int code;
Char name[20];
Public:
Void Insert()
{
Cout<<" Enter code ";
Cin>> code;
Cout<<" Enter name ";
gets(name);
}
Void output()
{
Cout<<" code "<<code;
Cout<<" name ";
Puts(name);
}
};
```

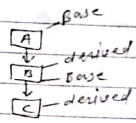
```
Class Employee
{
Protected:
float salary;
Public:
Void getdata()
{
Cout<<" enter salary "; Cin>>salary;
}
Void Putdata()
{
Cout<<" salary "<<salary;
}
};
```



```
Class report : Public Info, Public Employee
{
Public:
Void getfunction()
{
Info :: Insert();
Employee :: getdata();
}
Void Putfunction()
{
Info :: output();
Employee :: Putdata();
}
};
```

```
Void main()
{
Report r;
r.getfunction();
r.Putfunction();
}
```

* Multilevel Inheritance ⇒



The multilevel inheritance, we have total three classes. we have two base class & two derived class.

a = b & b = c then a = c

Ex:

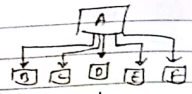
```
Class Info
{
Protected: Int code;
           char name[20];
Public:
Void Insert()
{
Cout<<" enter code ";
Cin>> code;
Cout<<" enter name ";
gets(name);
}
Void output()
{
Cout<<" code = "<<code;
Cout<<" name = "<<name;
}
};
```

```
Class Employee : Public Info
{
Protected: float salary;
Public:
Void getdata()
{
Info :: Insert();
Cout<<" enter salary ";
Cin>>salary;
}
Void Putdata()
{
Info :: Outdata();
Cout<<" salary "<<salary;
}
};
```

```
class report : public employee
{
public:
void getfunction()
{
employee :: getdata();
}
void putfunction()
{
employee :: putdata();
}
};
```

```
void main()
{
 Report r;
r.getfunction();
r.putfunction();
}
```

* Hierarchical inheritance ⇒



In this inheritance we have single base class with multiple derived class.

Ex:-

```
class info
{
protected:
int code;
char name[20];
public:
void insert()
{
cout<<"Enter code";
cin>>code;
cout<<"Enter name";
gets(name);
}
void output()
{
cout<<"code"<<code;
cout<<"name";
puts(name);
}
};
```

```
class employee : public info
{
float salary;
public:
void getdata()
{
info::insert();
cout<<"Enter salary"; cin>>salary;
}
void putdata()
{
info::output();
cout<<"salary"<<salary;
}
};
```



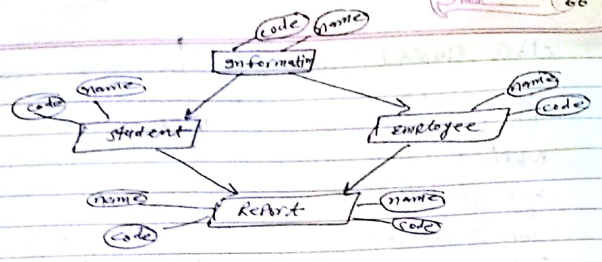
```
class student : public info
{
int total;
float per;
public:
void getdata()
{
info::insert();
cout<<"Enter marks";
cin>>total;
per= total/5;
}
void putdata()
{
info::output();
cout<<"total"<<total;
cout<<"per"<<per;
}
};
```

```
void main()
{
 Employee E;
 student S;
cout<<"employee information";
E.getdata();
E.putdata();
cout<<"student information";
S.getdata();
S.putdata();
}
```

* Hybride inheritance ⇒

In hybride inheritance, The combination of at least two different types of inheritance. In such a case we have a combination of single & multiple / single & Hierarchical multiple to ~~hierarchical~~ hierarchical / Hierarchical to multilevel some times.
we ~~have~~ have a combination of hierarchical & multiple.
Such type of combination required virtual base class.

⇒ Virtual base class is that class whose single copy is exist in the last derived class.



from diagram:-
जब information class के code & name student & employee class में आ जाते हैं। inheritance का use करते पर
फिर again student & employee class inherite होते हैं Report class में तो Report class में student & employee class से name & code आ जाते हैं। जिससे दो बार code & name आ जायेंगे जबकी दोनों name & code एक से हैं। क्योंकी दोनों information class से ही inherite होके आये हैं।
अतः ये दो बार का आने इसलिये Virtual base class का concept use करते हैं।

```
Class info
{
Protected:
int code;
char name[20];
Public:
Void getdata()
{
cout<<"enter code";
cin>> code;
cout<<"enter name";
gets(name);
}
Void putdata()
{
cout<<"code "<< code;
cout<<"name";
puts(name);
}
};
```



```
Class student : Virtual Public info
{
int total;
Public:
Void getdata()
{
cout<<"enter total marks";
cin>> total;
}
Void Putdata()
{
cout<<"total"<< total;
float per = total/5;
cout<<"percentage"<<per;
}
};
Class employee : Virtual Public info
{
int salary;
Public:
Void getfunction()
{
cout<<"enter salary";
cin>> salary;
}
Void Putfunction()
{
cout<<"salary"<< salary;
}
};
Class report: Public student, Public employee
{
Public:
Void output()
{
info:: Putdata();
student:: Putdata();
Employee:: Putdata();
}
};
Void main()
{
report r;
r.input();
r.output();
}
```

Difference b/w Specialization Vs Generalization

Specialization	Generalization
* Top-Down approach	* Bottom-up approach
* we start with a base class & deriving other classes from it	* we start with separate classes & generalize a common base class from them.

similarity:-
Both are iterative process of analysis & design.

Benefits (or need) of inheritance ⇒

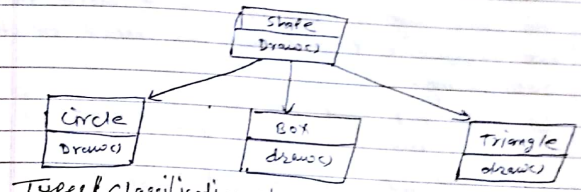
* Inheritance is a way to provide natural model for a particular domain.
  Ex:- A.I. domain consists of
  → object oriented programming
  ⇒ object oriented design
  → object oriented analysis
* Inheritance provides generalized concepts by reducing real-world's inherent complexity.
* It provides reusable structures & code.
* It explains a kind-of relationship.
* It provides increased ~~magni~~ magnitude of productivity between applications.
* Code redundancy can be avoided.
* It provides dynamic object ~~mainten~~ maintenance.

Dynamic Inheritance :-

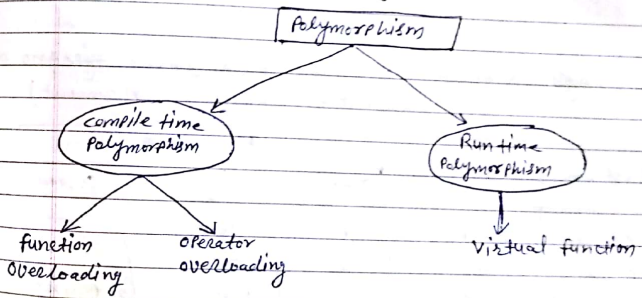
It is the ability to change parent class attributes at run time through object of the class. It allows object to change properties & attributes of the base class over time.
Ex:- Delegation Dynamic inheritance is achieved in the form of delegation by small talk, CLOS & actors.

Polymorphism ⇒ Polymorphism is one of the crucial features of OOPs. It simply "one name multiple forms". We have already seen how the concept of polymorphism is implemented using the overloaded concept.

~~Ex: class~~ Polymorphism is a greek word means "ability to take more than one form".



Types / Classification of Polymorphism ⇒



⇒ Compile time polymorphism / static linking or static binding or early binding or static polymorphism) ⇒ It can be achieved through overloaded concept.

→ function overloading
→ operator overloading

* Function overloading ⇒

We known that in C language there are no two function having same name but in c++ with it's possible to have two function name same.

In function overloading: we have the same logical name but different physical name.

(or)

The same logical name of two or more function but different physical name is called function overloading.

Ex:-
```
void swap(int, int);      } possible
void swap(int, float);

void swap(int, int)       } not possible
void swap(int, int)
```

Physical name = logical name + (argument type + no. of argument + sequence of argument)

Ex:

physical name	Logical name	arguments		
		no.	Type	Sequence
swap(int)	swap	1	int	—
swap()	swap	0	—	—
swap(int,int)	swap	2	int	int, int
swap(int,float)	swap	2	int, float	int, float
swap(float, int)	swap	2	float, int	float, int


```
int area(int x);
int area(int x, int y);
float area(float x, float y);

void main()
{
int r, l, w;
float b, h;
cout<<"Enter radius";
cin>>r;
cout<<"Area "<<area(r);
cout<<"enter length & width of rectangle";
cin>>l>>w;
cout<<"Area "<<area(l, w);
cout<<" enter base & hieght of triangle";
cin>>b>>h;
cout<<"Area "<<area(b, h);
}
int area(int r)
 {
 return (3.14*r*r);
 }
int area(int x, int y)
 {
 return (x*y);
 }
float area(float x, float y)
 {
 return (0.5*x*y);
 }
```

Advantage of function overloading ⇒

* Eliminating the use of different function name for same operation.
* Helps to debug
* Easy maintainability of the code
* Better understanding of the relation b/w the program & the outside world.

* Operator overloading ⇒

As we know that overloading a concept in which we have some logical name but different physical name. In the case of operator overloading we have to provide a new data types to ~~other~~ our arithmetic operator.

An overloaded operator is called an operator function. You declare an operator function with the keyword operator preceding the operator. Overloaded operators are distinct from overloaded functions but like overloaded function.

Ex:- + operator à addition of two no. &
addition of two strings (name & surname)

```
class OPover
{
private: float n;
public:
void insert()
{
cout<<"enter no";
cin>>n;
}
OPover operator+ (OPover A)
{
A.n = A.n+n;
return(A);
}
void output()
{
cout<<"n "<<n;
}
};
```

return type {class type}

O_1 [n]
O_2 [n]
O_3 [A.n]

```
void main()
{
OPover O1, O2, O3;
O1.insert();
O2.insert();
O3 = O1+O2;
O3.output();
}
```

internally $O_3 + O_1 . + O_2$

Q. Write a program to overload all the mathematic operation (operators) +, -, *, / using a single class.

Sol^n

```
class OPover
{
int n;
public:
void insert()
{
cout<<"enter no";
cin>>n;
}
OPover operator+ (OPover A)
{
A.n = A.n+n;
return(A);
}
OPover operator- (OPover B)
{
B.n = B.n - n;
return(B);
}
OPover operator*(OPover C)
{
C.n = C.n * n;
return(C);
}
OPover operator/ (OPover D)
{
D.n = Dn / n;
return(D);
}
void output()
{
cout<<"n "<<n;
}
};
```

```
void main()
{
OPover O1, O2, O3, O4, O5, O6;
O1.insert();
O2.insert();
O3 = O1+O2;
O3.output();
O4 = O1-O2;
O4.output();
O5 = O1 * O2
O5.output();
O6 = O1/O2;
O6.output();
}
```

Q.1
```
class complex
{
double real, img;
public:
complex(double real=0.0, double img=0.0);
complex operator + (const complex &) const; // operator + ()
};
complex:: complex(double r, double i)
{
real=r;
img=i;
}
complex complex:: operator + (const complex &c) const // overloaded + operator
{
complex result;
result.real= (This->real+c.real);
result.img = (This->img+c.img);
return result;
}
void main ()
{
complex (4,4);
complex (6,6);
complex z=x+y; // call complex + operator overloaded function
}
```

Overloading unary operators:-

```
Class cs
{
Private: int a,b,c,d;
Public:
```



```
cs(int i, int j, int k, int l) // Parameterised constructor
{
a=i; b=j; c=k; d=l;
}
Void show();  // member function
Void operator ++();  // operator function declaration
};
Void cs::show()
{
cout<< "value of a="<<a<<"b="<<b<<"c="<<c<<"d="<<d;
}
Void cs:: operator ++()
{
a++; b++; c++; d++;
}
Void main()
{
cs obj(2,3,4,5);
cout<<" Before increment";
obj.show();
obj++;
cout<<" After increment";
obj.show();
}
```

o/p
a=2, b=3, c=4, d=5
a=3, b=4, c=5, d=6

Overloading Binary operators:-
using member function

```
class cs
{
Private: int a,b,c,d;
Public:
Void getinfo();
Void Show();
cs operator + (cs);  // declaration of operator function using member function.
};
```

```
Void cs:: getinfo()
{
cout<<"enter value a,b,c,d";
cin>>a>>b>>c>>d;
}
void cs:: show()
{
cout<<"a"<<a<<"b"<<b<<"c"<<c<<"d"<<d;
}
cs cs:: operator +(cs t)
{
cs temp;
temp.a = a+t.a;
temp.b = b+t.b;
temp.c = c+t.c;
temp.d = d+t.d;
return temp;
}
```

```
void main()
{
cs obj1, obj2, obj3;
cout<<"enter value for obj1"
obj1.getinfo();
cout<<"enter value for obj2";
obj2.getinfo();
obj3 = obj1+obj2;
cout<<"value";
obj3.show();
}
```

Using friend function ⇒

all same code

```
Void getinfo();
void show();
friend cs operator *(int, cs);
};
```

```
cs operator *(int a, cs t)
{
cs temp;
temp.a = a*t.a;
return temp;
}
```

Advantage :-

* It is easy to read & debug.
* It helps programmer to use operators with the objects of classes.

```
void main()
{
cs obj1, obj2;
cout<<"value for obj1";
obj1.getinfo();
obj2 = 3*obj1;
cout<<"value";
obj2.show();
}
```


Advantage of compile-time Polymorphism ⇒

* Early binding is helpful to achieve greater efficiency
* function calls are faster because all the information necessary to call the function are hard code.

Overloaded operators in c++ are :-

+, +=, -, -=, *, *=, /, /=, !=, ==, <, <=, >, >= (Total = 14)

Operators common to unary & binary form ⇒

+ , - , * , &

+ = Addition or unary increment
- = subtraction or unary decrement
* = multiplication & Pointer reference
& = Bitwise AND & address of

c++ operators which can not be overloaded ⇒

(. & .*) ⇒ member access operators.
(?:) ⇒ conditional operator
(::) ⇒ Scope resolution operator
(sizeof) ⇒ size operator

Is operator overloading good ? ⇒

Since operator overloading provides a flexible option for the creation of new definition for the most of the c++ operators, so it is very useful. With the help of operator overloading we can create new data types for use in our program, we can perform different operations on our own data types so it is good.

Other example for Polymorphism

```
class num
{
int n1, n2;
public:
void insert()
{
cout<<"enter no";
cin>>n1>>n2;
}
void output()
{
cout<<"n1"<<n1;
cout<<"n2"<<n2;
}
};
```

Polymorphism

```
void main()
{
num *N;
num N1;
N=&N1;
N->insert();
N->output();
}
```

This pointer

normal

```
void main()
{
num N;
N.insert();
N.output();
}
```

⇒ Runtime Polymorphism [Dynamic binding / Late binding / Dynamic polymorphism]

implement using virtual function & object pointer

Virtual function ⇒ Virtual means existing in effect but not in reality. It is declared as virtual in base class & re-defined in a derived class.

When both the base & derived classes use the same function name, the function in base class must be declared as virtual using the keyword "virtual"

When pointer of base class is made to contain the address of a derived class, always executes the function in the base class.

Why:- Because compiler chooses the member function that matches the types of the pointer & ignores the contents of the pointer.]

⇒ Run time Polymorphism is achieved only when a virtual function is accessed through a pointer to the base class.

⇒ Without virtual function ⇒

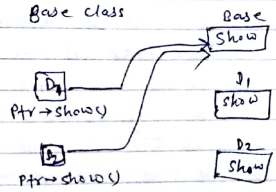
```
class Base
{
public:
void show()
{
cout<<"Base class";
}
};
class Derived1 : public Base
{
public:
void show()
{
cout<<"Derived1 class";
}
};
class Derived2 : public Base
{
public:
void show()
{
cout<<"Derived2 class";
}
};
```

```
void main()
{
Derived1 D1;
Derived2 D2;
Base *ptr;
ptr = &D1;
ptr->show();
ptr = &D2;
ptr->show();
}
```

o/p

Base class

Base class



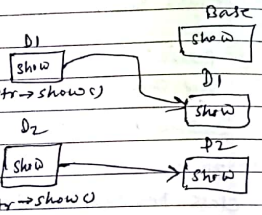
⇒ With Virtual function ⇒

```
class Base
{
public:
virtual void show()
{
cout<<"Base class";
}
};
class Derived1 : public Base
{
public:
void show()
{
cout<<"Derived1 class";
}
};
class Derived2 : public Base
{
public:
void show()
{
cout<<"Derived2 class";
}
};
```

```
void main()
{
Derived1 D1;
Derived2 D2;
Base *ptr;
ptr = &D1;
ptr->show();
ptr = &D2;
ptr->show();
}
```

O/P

Derived1 class
Derived2 class



Pure Virtual function ⇒

A pure virtual function is a virtual function with no body. you may have in virtual program that Base::show(); is never executed.
The body of the virtual function is in the base class can be removed & the notation = 0 added to the function declaration.


```
class Base
{
public:
virtual void show() = 0;
};
```

आगे सारा same है

```
void main()
{
Base* B1, B2;
```

or `Base B[2];`

```
Derived1 d1;
Derived2 d2;
B1 = &d1;        or  B1[0] = &d1;
B2 = &d2;            B2[1] = &d2;
B1->show();      or  B1[0]->show();
B2->show();          B2[1]->show();
```

O/P

Derived d1

derived d2

Virtual function

Advantages ⇒

* Powerful features of c++.
* Flexibility & increased power.
* It allows us to events that occur at run time without having to create a large amount of contingency-code.

Virtual function

Disadvantage:-

* Sometimes slower execution
* It requires some overhead.

Abstract class:-

These class, not need to create a object. It is designed only as a base class (inherited by other class). It is a designed concept in programming development.

Ex:- In inheritance
Info class is an abstract class in previous example because it is not used to create any objects.

Ex: Any inheritance program

Constructor ⇒

Constructor is a member function of a class having same name as class name. Generally we define a separate member function for reading input values for data members using object, member function is invoked & data members are initialized.
Constructor constructs the object & destructor destroys them.

Syntax rules for constructor ⇒

* The name of constructor function must be same as that of it's class name.
* constructor's are declared with no. return type & not even void.
* It should have public or protected access.

* It can be invoked for const object. constant
* It can not be declared as static virtual & ~

Syntax:-

```
class <classname>
{
Private :
==
public :
<classname> ()
  {
  ==
  }
};
```

Example

```
class Info
{
==
public :
==
Info()
{
==
}
};
```

Features (characteristics) of constructor ⇒

* whenever an object is created, the constructor will be executed automatically.
* It is called before main() function execution.
* we can not refer to their addresses
* constructors can have default arguments
* They can not be inherited
* Constructor without arguments is called a default constructor.
* An object with a constructor can not be used as member function.
* Initialization of an object of the class becomes mandatory when a constructor is declared for a class.
* constructor can be overloaded to accommodate many different forms of initialization.

P.T.O.

```
class cs
{
private: int i;
public:
cs()
{
cout<<"enter value of i";
cin>>i;
cout<<" value of i="<<i;
}
};
```

```
Void main()
{
cs obj;
}
```

Types of constructor ⇒

- Default constructor
- Parameterised constructor
- Multiple constructor
- copy constructor
- Dynamic constructor

* Default constructor (constructor without arguments)

A default constructor function initializes the data members with ~~not argu~~ no argument & performs no processing other than the reservation of memory.
It is a special member function which is invoked by the c++ compiler without ~~any~~ any parameter for initialising the object of the class

```
Class cs
{
Private: int a,b,c,X;
Public:
```

"That constructor which recieves any arguments is called Parameterised constructor"


```
cs()
{
cout<<"constructor called";
X=5;
cout<<"Enter a,b,c";
cin>>a>>b>>c;
}
Void show()
{
cout<<" a "<<a<<" b "<<b<<" c "<<c<<"X"<<X;
}
};
Void main()
{
cs obj;
obj.show();
}
```

o/p ⇒ constructor called
Enter a,b,c
10, 20, 30
a=10, b=20, c=30 & X=5

* Parameterised constructor (constructor with arguments)

In these constructor, when we pass the argument (Parameter) through constructor is called Parameterised constructors.

```
class integer
{
int m,n;
public:
integer(int, int);
};
integer::integer(int x, int y)
{
m=x;
n=y;
cout<<"m "<<m<<"n "<<n;
}
```

```
Void main()
{
integer obj(2,3);
}
```

* Multiple constructor (overloaded constructor) ⇒
As we know that overload ~~ed~~ means same logical name but different physical name.

Multiple constructor provides more flexibility to the ways in which an object may be initialized.

```
class cs
{
int a;
float b;
char c;
public:
cs(int i, float j, char k)
{
cout<<"constructor called with three argument";
a=i; b=j; c=k;
}
cs(int i, float j)
{
cout<<"constructor called with two arguments";
a=i; b=j;
}
cs()
{
cout<<"constructor called without argument";
a=0; b=0; c=0;
{cout<<"a"<<a<<"b"<<b<<"c"<<c;
}
Void show()
{
cout<<"a"<<a<<"b"<<b<<"c"<<c;
```

```
Void main()
{
cs obj1(1, 2.5, 'A');
obj1.show();
~~cs obj2(2, 3.5);~~
~~obj2.show();~~
cs obj2(2, 3.5);
obj2.show();
cs obj3;
obj3.show(); //May not used
}
```


* Constructor with default argument ⇒
It is ~~similar~~
Similar to function with default argument
Ex:- Power(int n, int p=3); //default argument is 3

```
class Power
{
int num, power, ans;
public:
Power(int n=4, int P=3)
{
num=n;
Power=P;
ans = Pow(n, P);
}
Void show()
{
cout<<num<<"Power"<<Power<<"is"<<ans;
}
};
```

```
Void main()
{
Power obj;
obj.show();
Power obj1(5);
obj1.show();
}
```

O/P
4 Power 3 is 64
5 Power 3 is 125

* Copy constructor ⇒
Copy constructor is that constructor which recieve the arguments of it's our class object & initialize own (स्वयं) data member as the member of recieve objects.
If in ~~our~~ our class we have copy constructor then must have ~~default~~ ~~constr~~ default constructor.

Features of copy constructor ⇒

* copy constructor is used when the compiler has to create a temporary object of a class object.

[copy constructor में एक class के more than two objects हो तो एक object के data को दूसरे object में copy करने के लिए copy constructor का use करते हैं]

* It allows programmer to instantiate an object as an exact copy of another object in terms of its attribute values.
* These may be used when programmer has to return object as a function value.
* It is one of the more important form of overloaded constructor.
* It may be used when a copy of an object is made to be passed to a function by value.
* It applies only to initialization.
* A reference variable can be used as an argument to the copy constructor.
* Default copy constructor can be used on any objects without being explicity defined.
* We can not pass the argument by value to the copy constructor.

```
class copy
{
int a,b,c;
Public:
Void insert()
{
cout<<"Enter no.";
cin>>a>>b>>c;
}
copy (copy &c1)        // reference value
{
a=c1.a;
b=c1.b;
c=c1.c;
}
```

```
copy ()
{
a=0;
b=0;
c=0;
}
Void output()
{
cout<<"(a+b+c);
}
};
Void main()
{
copy c1;
c1.insert();
copy c2(c1);
c2.output();
}
```


* Dynamic Constructor ⇒

The constructor can also be used to allocate memory while creating objects. This will enable the system to allocate the right amount of memory for each object when the objects are not of the same size. Allocating of memory to objects at the time of their construction is known as dynamic constructor.

```
class cs
{
int *i;
Public:
cs ()                    // Dynamic constructor
{
i=new int (5); //  ↑
}                        // new operator used for memory allocation
Void show()              // member function
{
cout<<"Value of i = "<<*i;
}
};
Void main()
{
cs obj;
cs *ptr;
ptr= &obj
or ptr= new cs;
ptr=>show();
}
```

↑ This pointer

Destructor ⇒

It is used to destroy the objects that have been created by a constructor. The destructor is a member function of a class. it's name is the same as the class name but is preceded by a ~~tilde~~ tilde (~) sign. Destructor ~~never~~ never takes any argument nor does it return any value.

```
#include <iostream.h>
int count = 0;
class cs
{
public:
cs()
{
count++;
cout<<"no. of object created";
cout<< count;
}
~cs()
{
cout<<"object destroyed";
cout<< count;
count--;
}
};
void main()
{
cs c1, c2, c3, c4;
{
cout<<"Enter in block";
cs c5;
}
{
cout<<"Enter again in block";
cs c6;
}
cout<<"return in main";
}
```

o/p

1
2
3
4
5
5
5
5
4
3
2

Features (characteristics) ⇒

* Destructor functions can not be overloaded
* A constructor can not be virtual but a destructor can be virtual
* only one destructor can be defined inside a destructor.
* Destructor can not be inherited through a derived class.
* we can not refer to their address.
* They do not have any argument.
* Destructor is the last member function of a class.
* It is called before main() function execution
* Destructor destroy the object.
* Destructor function ~~execute~~ Execute automatically.

⇒ Difference b/w Inheritance Vs Polymorphism

Inheritance	Polymorphism
* Inheritance is the process by which objects of one class aquire the properties of objects of another class.	* Polymorphism is the ability to take more than one form.
* It provides the idea of reusability.	* It Provides reusable structures of code with generalized concepts by reducing real-world's ~~inhere~~ inherent complexity.
* It provides reusable structures & code with generalised concepts of reducing realworld inherent complexity	* It plays an important role to allow objects having different internal structure with same external interface.
* types:- single, multiple, multilevel, Hierarchical, Hybrid	* types: compile time & run time polymorphism,
* It uses based classes & derived classes	* It uses function & operators to be overloaded.

Inherited Methods ⇒

The methods of the base class which are redefined in the derived class to specialise the behavior through overriding are known as inherited methods.

When inheritance is used for specialization, a class extends its super class in order to redefine some of the super class methods, while leaving unchanged the rest of the method it inherits.

Inherited methods may be extended by the derived class.

Overriding ⇒

Ex:

```
class Base
{
public:
void getinfo()
{
cout << "Base class function";
}
};
class derived: public Base
{
public:
void getinfo()
{
cout << "derived class function";
}
};
```

```
void main()
{
class derived d;
cout << "overriding of member function";
d.getinfo();
}
```

o/p

```
overriding of member function
Derived class function
```

Features of overriding ⇒

* It creates a class that is a variation of it's parent class.
* It creates a class which extends it's base class to redefine their methods.
* It provides a feature to derived class to create more specialized behaviour than the base class.
* To simplify the implementation, derived class uses the concept of reusability
* It creates a derived class that is more efficient than its base class in terms of less execution time & memory ~~non~~ requirement

Redefined Method ⇒

If we redefine a function in a derived class, it overrides the base class declaration with same function name & hides all base class methods of the same name, regardless of the parameters.

There is an important difference b/w redefined methods & overloading of functions.
Redefined methods provide a new way to change capability & performance of method by redefining an existing base class method.
Redefined method is not a variation of overloading.

Rules for redefined methods :-

* Redefines all the base class methods in derived class.
* Redefined method in the derived class must be match exactly with base class function prototype.
* If a derived class fails to redefine a virtual function, the base class function will be used by the derived class.

```
class Base
{
public:
virtual void show (char ch)
{
cout << "OOPs using Java";
}
};
class derived : public Base
{
public:
void show ()  // redefined method
{cout<<"OOPs using c++";
};
void main ()
{Derived d;
d.show();
}
O/P OOPs using c++
```


Disinheritance (Virtual inheritance) ⇒

Disinheritance is a totally theoretical concept which is used when a true inheritance relationship does not occur.
Inheritance can be private, public or virtual. In the c++ programming language, virtual inheritance is a kind of inheritance that solve some of the problems caused by multiple inheritance (particular the diamond problem) by clarifying ambiguity

With virtual inheritance (Disinheritance) there is only one copy of each object even if (because of multiple inheritance) the object appears more than once in the hierarchy. A multiple inheritance based class is denoted as virtual with the virtual keyword.

Ex:-

```
class Base
{
Public:
Virtual void show()
{
cout << "Base class function";
}
};
class Base1 : Virtual Public Base
{
Public:
Virtual void show()
{
cout << "Base1 class function";
}
};
class Base2 : Virtual Public Base
{
public:
Virtual void show()
{
cout << "Base2 class function";
}
};
class derived : Public Base1, Public Base2
{
public:
void show()
{
Base1::show();
Base2::show();
cout<<"derived class";
}
};
```