Classification of Languages

1. ***Procedure Oriented Programming Language***

FORTRON is 1st POP Language

e.g.: C, FORTRON and PASCAL

2. ***Object Oriented Programming Language***

Simula is 1st OOP Language in 1960

e.g.: C++, Smalltalk, Java and C#

Smalltalk is only language which is purely OOP Language

C++ is not purely object oriented programming language. it is also called as partial object oriented programming language

3. ***Object Based Programming Language***

Ada is 1st object based language.

e.g.: visual basic, Ada and Modula-2

4. ***Rule Based Programming Language***

e.g.: PROLOG and LISP

Any New language is basically designed for two reasons

1. To overcome or avoid limitations of previous language
2. To provide new features

Advantages of C Language

1. C is portable
2. C is efficient
   - can interact with hardware efficiently
3. C is flexible
   - we can cerate application software or system software
4. C is freely available
   - wide variety of compliers are available

- C is said to be procedure oriented, structured programming language.
- When program becomes complex, understating and maintaining such programs is very difficult.

- ***Limitations of C Programming with respect to C++***
- Language don't provide security for data.
- Using functions we can achieve code reusability, but reusability is limited. The programs are not extendible.
- We can not write function inside structure

So "Bjarne Stroustrup" designed a new language c with classes in 1979 on DEC PDP11 machine.
Restructure by ANSI in 1983.

in C++ 63 Keywords are available.
(unmanaged c++)

Procedure oriented	*Object Oriented*
Emphasis on steps or algorithm	Emphasis on data of the program
Programs are divided into small code units i.e. functions	Programs are divide into small data units i.e. classes
Most functions share global data & can modify it	Data is hidden & not accessible outside class
Data move from function to function	Objects communicate with each other
Top-down approach	Bottom- up approach

Variable declaration

- In C, variable should be declared at the start of the block.
- This restriction is removed in C++. We can declare the variables anywhere in function.

Data types

- C++ supports all data types provided by C language. i.e. int, float, char, double, long int, unsigned int, etc.
- C++ add two more data types:
- 1. *bool* :- it can take *true* or *false* value. It takes one byte in memory.
- 2. wchar_t :- it can store 16 bit character. It takes 2 bytes in memory.

Comments in C++

- In C, comments are written as
- /*This is comment*/
- In C++, we can use above style. In addition C++ provides one more way for writing comments.
- //This is comment
- The second style is preferred for single line comments.

Structure

- Structure is a collection of similar or dissimilar data. It is used to bind logically related data into a single unit.
- This data can be modified by any function to which the structure is passed
- Thus there is no security provided for the data within a structure.
- This concept is modified by C++ to bind data as well as functions.

Diff Between struct in c & c++

- struct in c
- We can't write function inside structure
- At the time of creating variable of structure writing struct keyword is compulsory
  eg. struct time t;
- By default all the members are accessible outside structure. C lang does not have a concept of access specifies
- If we want to call any function on structure variable
  struct time t1;
  input(&t1), print(t1),

- struct in c++
- We can write function inside structure
- At the time of creating object of structure writing struct keyword is optional
  eg.time t;
- By default all members of struct in c++ are public (we can make them private)
- If we want to call member function on object.
  time t1;
  t1.input();
  t1.print();

```
struct time {
    int hr, min, sec;
};

void input( struct time *p)
{
    printf("Enter Hr Min Sec:");
    scanf("%d%d%d", &p->hr,
    &p->min, &p->sec);
}
struct time t;
input(&t);
```

```
struct time
{
    int hr, min, sec;
    void input()
    {
    printf("Enter Hr Min Sec::");
    scanf("%d%d%d",&this->hr,
    &this->min, &this->sec);
    }
};
time t;
t.input();
```

Access specifiers

- By default all members in structure are accessible everywhere in the program by dot(.) or arrow(→) operators.
- But such access can be restricted by applying access specifiers
  - private: Accessible only within the struct
  - public: Accessible within & outside struct

```
#include<stdio.h>
void f1(int a)
{
        printf("\n Inside int blcok");
}
void f1(float a)
{
        printf("\n Inside float blcok");
}
void f1(double a)
{
        printf("\n Inside double blcok");
}
int main()
{       f1(1);
        f1(1.2);
        f1(1.2f);
        f1((int)1.2);
        f1(1.2);
        return 0;
}
```

1

Function overloading :
Function having same name but differs either in different number of arguments or type of arguments or order of arguments such process of writing function is called function overloading .
Functions which is taking part in function overloading such functions are called as overloaded functions.

1. Function having same name but differs in number of arguments
   eg: void sum (int no1, int no2);
       void sum (int no1, int no2, int no3);

2. Function having same name and same number of arguments but differs in type of arguments
   eg: void sum (int no1, int no2);
       void sum (int no1, double no2);

3. Function having same name ,same number of arguments but order of arguments are different
   eg. void sum (int no1, float no2);
       void sum (float no1, int no2);

2

Name mangling: when we write function in c++ complier internally creates unique name for each and every function by looking towards name of the function and type of arguments pass to that function. Such process of creating unique name is called name mangling . That individual name is called mangled name.
Eg:

sum (int no1, int no2, it no3)
sum@ int, int , int
sum(int no1, int no2)
sum@int,int

cin and cout

- C++ provides an easier way for input and output.
- The output:
  - cout << "Hello C++";
- The input:
  - cin >> var;


```
#include<stdio.h>
int main()
{
   printf(" hellow world");
}
```

```
#include<iostream.h>
int main()
{
   cout<<"hellow world";
}
```

printf is a function & declared in stdio.h header file. So if we want to use printf it is necessary include stdio.h header file

cout is object of ostream class and ostream class is declared in iostream.h that's why if u want to use cout object is necessary to include iostream.h header file

operator which is used with cout is called as insertion operator <<

```
int res=30;
printf("res=%d", res);
```

```
int res=30;
cout<<"res="<<res;
```

5

```
   int a=10,b=5;
   printf("a=%d b=%d",a,b);
```

```
   int a=10, b=5;
   cout<<"a="<<a<<" b="<<b;
```

```
#include<stdio.h>
int main()
{ int num;
  scanf("%d",&num);
}
```

```
#include<iostream.h>
int main()
{ int num;
  cin>>num;
}
```

scanf is a function & declared in stdio.h header file. So if we want to use scanf it is necessary include stdio.h header file

cin is object of istream class and istream class is declared in iostream.hthat's why if u want to use cin object is necessary to include iostream.h header file

operator which is used with cin is called as operator extraction (>>)

```
int no1, no2;
scanf("%d%d",&no1, &no2);
```

```
int no1, no2;
cin>>no1>> no2;
```

What is class?

- **Building blocks that binds together data & code**
- **Program is divided into different classes**
- **Class has**
  - **Variables (data members)**
  - **Functions (member functions or methods)**

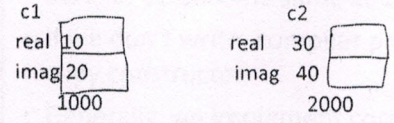
What is object?

- Object is an instance of class
- Entity that has physical existence, can store data, send and receive message to communicate with other objects
- Object has
  - Data members (***state*** of object)
  - Member function (***behavior*** of object)
  - Unique address(***identity*** of object)

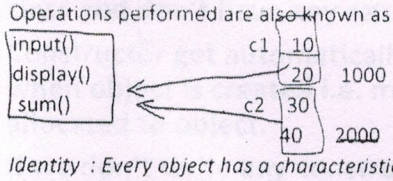
What is object?

- The values stored in data members of the object called as 'state' of object.
- Data members of class represent state of object.

complex c1, c2;



- Behavior is how object acts & reacts, when its state is changed & operations are done
- Behavior is decided by the member functions.
- Operations performed are also known as messages



- *Identity : Every object has a characteristics that makes object unique. Every object has unique address.*
- *Identity of c1 1000 & c2 is 2000*

Class

it is template or blue print for an object.

object is always created by looking towards class, that's why it is template for class.

class is a logical entity.

memory is not allocated to that class.

class is collection of data members and member functions.

Object

it is an instance of class.

object is physical entity.

memory is always allocated to object.

Data members

- **Data members of the class are generally made as private to provide the data security.**
- **The private members cannot be accessed outside the class.**
- **So these members are always accessed by the member functions.**

Member Functions

- Member functions are generally declared as public members of class.
- Constructor : Initialize Object
- Destructor : De-initialize Object
- Mutators : Modifies state of the object
- Inspectors : Don't Modify state of object
- Facilitator : Provide facility like IO

Constructor

- We can have constructors with
  - –No argument : initialize data member to default values
  - –One or more arguments : initialize data member to values passed to it
  - –Argument of type of object : initialize object by using the values of the data members of the passed object. It is called as copy constructor.

Copy Constructor

- Complex c1(c2);
- This statement gives call to copy constructor. State of c1 become same as that of c2.
- If we don't write, compiler provides default copy constructor.
- Generally, we implement copy constructor when we have pointer as data member which is pointing to dynamically allocated memory.

Constructor

- Constructor is a member function of class having same name as that of class and don't have any return type.
- Constructor get automatically called when object is created i.e. memory is allocated to object.
- If we don't write any constructor, compiler provides a default constructor.

Destructor

- Destructor is a member function of class having same name as that of class preceded with ~ sign and don't have any return type and arguments.
- Destructor get automatically called when object is going to destroy i.e. memory is to be de-allocated.

Destructor

- If we don't write, compiler provides default destructor.
- Generally, we implement destructor when constructor of the object is allocating any resource
- e.g. we have pointer as data member, which is pointing to dynamically allocated memory.

this pointer

- When we call member function by using object implicitly one argument is passed to that function such argument is called this pointer
- *this* is a keyword in C++.
- this pointer always stores address of current object or calling object.
- Thus every member function of the class receives *this* pointer which is first argument of that function.
- This pointer is constant pointer.

For complex class member function type of this pointer is
complex * const this
classname * const this

- Size of object of empty class is always 1 byte
- When you create object of an class it gets 3 characteristics
  - State
  - Behavior
  - Identity
- When you create object of empty class at that time state of object is nothing. Behavior of that object is also nothing.

  but that object have unique identity(address).

memory in computer is always organized in form of bytes.
Byte is unit of memory. Minimum memory at objects unique address is one byte that's why size of empty class object is one byte.

Default arguments

Assigning default values to the arguments of function is called default arguments.
Default arguments are always assigned from right to left direction.

Passing these arguments while calling a function is optional. If such argument is not passed, then its default value is considered. Otherwise arguments are treated as normal arguments


```
#include<iostream.h>
int sum (int a=0, int b=0, int c=0, int d=0)
{       return a+b+c+d;
}
int main()
{       int ans=sum();
        cout<<"sum="<<ans<<endl;
        ans=sum(10);
        cout<<"sum="<<ans<<endl;
        ans=sum(10, 20);
        cout<<"sum="<<ans<<endl;
        ans=sum(10, 20, 30);
        cout<<"sum="<<ans<<endl;
        ans=sum(10, 20, 30, 40);
        cout<<"sum="<<ans<<endl;
        return 0;
}
```


Inline functions

- C++ provides a keyword *inline* that makes the function as inline function.
- Inline functions get replaced by compiler at its call statement. It ensures faster execution of function just like macros.
- Advantage of inline functions over macros: inline functions are type-safe.
- Inline is a request made to compiler.
- Every function may not be replace by complier , rather it avoids replacement in certain cases like function containing switch , loop or recursion may not be replaced

Dynamic memory allocation

- C++ provides operators *new* and *delete* for allocating and de-allocating memory at run-time. The memory is allocated on heap.
- int *ptr; ptr=new int;
  delete ptr;
- char *str; str = new char[10];
  delete[] str;


```
#include<iostream.h>
void main()
{
        int no;
        cout<< "Enter How many Number You want ::";
        cin>> no;
        int *ptr= new int[no];
        for(int cnt=0;cnt<no;cnt++)
        {
                cout<<"Enter number :: ";
                cin>>ptr[cnt];
        }
        for( cnt=0;cnt<no;cnt++)
        {
                cout<<" "<<ptr[cnt] << endl;
        }
        delete [] ptr;
        ptr=NULL;
        cout<<"memory freed"<<endl;
}
```

Difference between malloc and new :

	malloc	new
1	malloc is a function	new is a operator
2	Allocate memory using malloc constructor is not called i.e.. malloc is not aware of ctor	Allocate memory using new constructor is called i.e.. new is aware of ctor
3	If malloc fails return NULL	if new fails it throws bad_alloc exception
4	Need to specify number of bytes and typecasting is required	Need to specify number of objects and typecasting is not required

Malloc

- sizeof operator is required
- When we deallocate memory by using free function which is allocated for an object by using malloc function at that time implicitly destructor will not be called .

New

- sizeof operator is not required
- When we deallocate memory by using delete operator which is allocated for an object by using new operator at that time implicitly destructor of that class will be called.

References

- References are treated as aliases to the variable.
- It can be used as another name for the same variable
- int a=10; int &r = a;
- Thus we can pass arguments to function, by value, by address or by reference.
- Reference is internally treated as constant pointer to the variable, which gets automatically de-referenced.


```
#include<iostream.h>
void swap(int &n1, int &n2)
{
        int temp;
        temp=n1;
        n1=n2;
        n2=temp;
        cout<< "\n &n1 ::"<< &n1<<" &n2 ::" << &n2<<endl;
}
void main()
{
        int no1=5, no2=10;
        cout<< "\n no1 ::"<< no1<<" no2 ::" << no2<<endl;
        cout<< " \n &no1 ::"<< &no2<<" no2 ::" << &no2<<endl;
        swap(no1, no2);
        cout<< "\n no1 ::"<< no1<<" no2 ::" <<no2<<endl;
        cout<< "\n &no1 ::"<< &no1<<" &no2 ::" << &no2<<endl;
}
```

Difference between pointer and reference:

	Pointers	References
1	pointers may not be (not compulsory) initialized at the point of declaration.	reference must be initialized at the point of declaration.
2	pointers must be derefeneced explicitly using value at (*) operator.	References are automatically dereference.
3	address stored in a pointer can be modified later.	reference keeps referring to the same variable till it goes out of scope.
4	we can have pointer arithmetic, null pointers, dangling pointers.	such concepts do not exist for references.
5	We can initialized pointer to NULL.	We can not initialized reference to NULL
6	We can create a array of pointer.	We can not create a array of reference.
7	We can create pointer to pointer	Can not create reference to reference.

Copy Constructor:
it is a special member function of a class which is having same of that class and which gets called implicitly.
1. when we pass an object to the function by value.
2. when we return an object from function by value.
3.when we assign already created object to the newly created object (object initialization)
such special member function is called copy constructor of that class. Job of copy constructor is to create new object from existing object.
When class do not contain copy constructor at that time complier provides one copy constructor for that class by default such copy constructor is called default copy constructor.
It is a special member function of a class having same name of a class and which is taking only one argument of same type but as a reference.

```
TComplex (const Complex& other)
{
        this->_real=other._real;
        this->_imag=other.imag;
}
int main()
{
        TComplex  c1(10, 20);
        TComplex  c2=c1; // in this case copy constructor will call for object c2;
}
```

Why c++ is not pure object oriented programming language?
Or why it is partial object oriented programming language ?

1. in c++ we can access private data members of class outside that class by using friend function.

2 . in c++ we can access private data members of class outside that class by using pointers.

3.in c++ we can write global functions and main itself a global function

4. in pure object oriented programming lang data types are also having a classes. Such type of class is called wrapper class. Such type of concept is not available in c++ that's why c++ is not pure object oriented programming language.

Friend Function
it is a non member function of a class which can access private members of class in which it is declared as friend.
Friend function do not have this pointer.
We can declared global function as friend.

```
#include<iostream.h>
class Test
{
 private:
        int _a;
        int _b;
 public :
        Test();
        Test(int a, int b);
        friend void sum();
};
```

```
Test::Test()
{
        this->_a =0;
        this->_b =0;
}
Test::Test(int a, int b)
{
        this->_a = a;
        this->_b = b;
}
void sum()
{
        Test t(10, 20);
        int ans=t._a +t._b;
        cout<<"ans::"<<ans<<endl;
}
int main()
{
        sum();
        return 0;
}
```

The global functions cannot access private data members of the class.
If such function is made as *friend* of the class, then it can access private members of the class.
If class is made as friend of other class, then all functions of friend class can access private members of that class.

Operator overloading:
giving extension to the meaning of operator is called operator overloading.
By using operator overloading we can change the meaning of operator but we should not change the meaning of operator.

```
class TComplex
{
        int _real;
        int _imag;
public:
        TComplex();
        TComplex(int real , int imag);
        TComplex sum(TComplex c);
        TComplex operator+(TComplex c);
        void Output();

};
```

```
TComplex TComplex::sum(TComplex c)
{
        TComplex temp;
        temp._real =this->_real +c._real ;
        temp._imag =this->_imag +c._imag ;
        return temp;
}
TComplex TComplex::operator+(TComplex c)
{
        TComplex temp;
        temp._real =this->_real +c._real ;
        temp._imag =this->_imag +c._imag ;
        return temp;
}
int main()
{
        TComplex c1(10, 20), c2(20, 10);
        TComplex c3=c1.sum (c2);
        c3.Output ();
        TComplex c4=c1+c2;
        c4.Output ();
        return 0;
}
```

When we write like c3=c1+c2 at that time compiler resolves call for this statement like c3=c1.operator+(c2);
meaning of this statement is that we have to write function by name of ' operator+'. Since this function has called by using object name it must be inside class. Operator+ function taking one argument means at the time of function definition it is necessary to pass argument to that function

function definition for statement c3=c1+c2 is

```
TComplex TComplex::operator+(TComplex c)
{       TComplex temp;
        temp._real =this->_real +c._real ;
        temp._imag =this->_imag +c._imag ;
        return temp;
}
```

```
c3=c1 – c2; // c3=c1.operator-(c2);
c3=c1 * c2; // c3=c1.operator*(c2);
c3=c1 / c2; // c3=c1.operator/(c2);
```

We can overload operator function by using member as well as friend function friend function.
When we write c3=c1+c2;
and operator+ function is member function at that time function call will be resolved.
like c3= c1.operator+(c2)
when we overload operator+ function by friend function implicitly call will be resolved like c3= operator+(c1, c2):

when we overload operator- function by friend function implicitly call will be resolved like c3= operator-(c1, c2):

```
TComplex operator-(TComplex c1, TComplex c2)
{
        TComplex temp;
        temp._real=c1._real – c2._real;
        temp._imag=c1._imag – c2._imag;
        return temp;
}
```

we can not overload operator function as member function as well as friend function at same time. In this case compiler will confuse and it will give you ambiguity error.

List of function that complier provides by default for any class if it is not available in that class
1. default parameter less constructor
2. default destructor
3. default copy constructor (shallow copy)
4. default assignment operator function

Limitations of operator Overloading:
in c++ there are some operator that we can not overload using member function as well as friend function.
1. . (dot) member selection operator
2. :: scope resolution operator
3. ? : conditional operator
4. sizeof size of operator
5. .* pointer to member selection
6. typeid

There are some operators we can not overload then as friend function
1 = assignement
2 [] subscript or index
3 () function call
4 -> arraow operator

can overload as member fn

7) Static_cast
8) reinterprit_cast
9) Dynamic_cast
10) const_cast

OOPS

Structure

- It is a programming methodology to organize complex program in to simple program in terms of classes and object such methodology is called oops.
- It is a programming methodology to organized complex program into simple program by using concept of abstraction , encapsulation , polymorphism and inheritance.
- Languages which support abstraction , encapsulation polymorphism and inheritance are called oop language.
- 4 major pillars of oops
  - Abstraction
  - Encapsulation
  - Modularity and Hierarchy

- Minor pillars of oops
  - Polymorphism
  - Concurrency (OS)
  - Persistence (file handling)
- Abstraction : getting only essential things and hiding unnecessary details is called as abstraction.
- Abstraction always describe outer behavior of object.
- In console application when we give call to function in to the main function it represents the abstraction

- Encapsulation :binding of data and code together is called as encapsulation. Implementation of abstraction is called encapsulation.
- Encapsulation always describe inner behavior of object.
- Function call is abstraction
- Function definition is encapsulation.
- Abstraction always changes from user to user.

- Information hiding
- Data : unprocessed raw material is called as data.
- Process data is called as information.
- Hiding information from user is called information hiding.
- In c++ we used access specifier to provide information hiding.

- Modularity
- Dividing programs into small modules for the purpose of simplicity is called modularity.
- There are two types of modularity
  - Physical modularity
  - Logical modularity

- Physical modularity (.h & .cpp files)
  - Dividing a classes into multiple files is nothing but physical modularity
- Logical modularity
  - Dividing classes into namespaces is called logical modularity

- Polymorphism (Typing)
- One interface having multiple forms is called as polymorphism.
- Polymorphism have two types
- Compile time polymorphism and runtime polymorphism.

Compile time	run time
Static polymorphism	Dynamic polymorphism
Static binding	dynamic binding
Early binding	late binding
Weak typing	strong typing
False polymorphism	true polymorphism

- Compile time polymorphism: when the call to the function resolved at compile time it is called as compile time polymorphism. And it is achieved by using function overloading and operator overloading (f° mangled name)
- Run time polymorphism : when the call to the function resolved at run time it is called as run time polymorphism. And it is achieved by using function overriding.

- Hierarchy
- Order/level of abstraction is called as hierarchy.
- Types of Hierarchy
  - has a Hierarchy (Composition) (has a relationship)
  - is a Hierarchy (inheritance) (is a ")
  - use a Hierarchy (dependency) (use a ")

Composition : when object is made from other small-small objects it is called as composition.
When object is composed of other objects it is called as composition
eg: room has a wall
room has a chair
system unit has motherboard
system unit has modem.
Types of composition :
1. Association 2. Aggregation 3. Containment

1. Association
Removal of small object do not affect big object it is called as associatio
eg. Room has chair . Association is having "loose coupling"

2. Aggregation :
Removal of small object affects big object it is called as Aggregation .
eg. Room has wall Aggregation is having "tight coupling"

3. Containment : stack , queue, linked list, array , vectors these are collectively called collections.
When class contain object of collection it is called as Containment.
Eg: room has number of chairs
company has number of employees.
In composition we always declared object of one class as a data member of another class.

```
class TDate
{       int _dd;        int _mm;        int _yy;
};
class TAddress
{       char _city[20];   char _month[20];        int _pincode;
};
class Tperson
{       char name[30];   TDate _birthdate;        Taddress _address;
};
```

```
#include<iostream.h>
#include<string.h>
class TDate
{
        int _dd; int _mm; int _yy;
public:
        TDate()
        {       this->_dd=1;
                this->_mm=1;
                this->_yy=1900;
        }
        TDate(int dd,int mm,int yy)
        {       this->_dd=dd;
                this->_mm=mm;
                this->_yy=yy;
        }
        void Display()
        {
           cout<<"Date :"<<this->_dd<<"/"<<this->_mm<<"/"<<this->_yy<<endl;
        }
};
```

```
class TAddress
{       char _city[20];
        char _state[20];
        int _pincode;
public:
        TAddress()
        {       strcpy(this->_city,"");
                strcpy(this->_state,"");
                this->_pincode=0;
        }
        TAddress(char* ct,char* st,int pin)
        {       strcpy(this->_city,ct);
                strcpy(this->_state,st);
                this->_pincode=pin;
        }
        void Display()
        {       cout<<"City :"<<this->_city<<endl;
                cout<<"State :"<<this->_state<<endl;
                cout<<"Pin Code :"<<this->_pincode<<endl;
        }
};
```

```
class TPerson
{       char _name[30];
        TDate _birthdate;
        TAddress _address;
public:
        TPerson()
        {       strcpy(this->_name,"");
        }
        TPerson(char* nm,int dd,int mm,int yy,char* ct,char* st,int pin)
                :_birthdate(dd,mm,yy),_address(ct,st,pin)
        {       strcpy(this->_name,nm);
        }
        void Display()
        {       cout<<"Name :"<<this->_name<<endl;
                _birthdate.Display();
                _address.Display();
        }
};
void main()
{       TPerson p("ABC",11,1,1975,"Pune","Maharastra",12345);
        p.Display();
}
```

Inheritance: acquiring all properties(all data members) and behavior of one class (base class) by another class (derived class) this concept is called as inheritance.
Ex: class TEmployee : TPerson (is-a repletionship)
Derived Class : Base Class
at the time of inheritance when name of members of base class and name of members of derived class are same at that time explicitly mention scope resolution operator is a job of programmer.

When you create a object of derived class at that time constructor of base class will call first and then constructor of derived class will called.

Destructor calling sequence is exactly opposite that is destructor of derived class will called first and then the destructor of base class will called.

At the time of inheritance all the data members and member functions of base class are inherited into derived class but there are some functions which are not inherited into derived class.
1. constructor 2. copy constructor 3. destructor 4. friend function
5. assignment operator function

When we create a object of derived class at that time size of that object is size of all non static data members declared in base class plus size of all non static data members of declared in derived class.

```
#include<iostream.h>
class A
{
        int a;
public:
        A()
        {
                this->a=10;
        }
        A(int a)
        {
                this->a=a;
        }
        void Print()
        {
                cout<< "a ::"<<this->a<<endl;
        }
};
```

```
class B: A
{       int b;
public: B(){
            this->b=20;
        }
        B(int b){
            this->b=b;
        }
        void Print()
        {       A::Print();
                cout<< "b ::"<<this->b<<endl;
        }
};
int main()
{       B obj;
        obj.Print();
        cout<<"size of b ::"<<sizeof(obj)<<endl;
        A obj1;
        obj1.Print();
        cout<<"size of a ::"<<sizeof(obj1)<<endl;
        return 0;
}
```

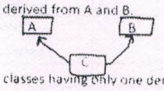
Types of inheritance:

1. Single Inheritance: B is derived from A

one base class having only one derived class. such inheritance is called as single inheritance. Eg Employee is a person

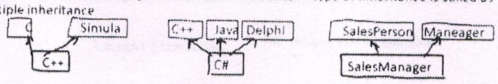
2. Multiple Inheritance:

C is derived from A and B.



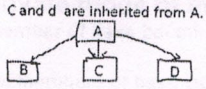
multiple base classes having only one derived class such type of inheritance is called as multiple inheritance

eg

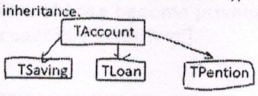


3. hierarchical inheritance:

B, C and d are inherited from A.



one Base class having multiple derived classes such type of inheritance is called hierarchical inheritance.



4. Multilevel inheritance :

B is derived from A , C is derived from B and D is derived from C.

When single inheritance has multiple levels is called as multilevel inheritance.

Hybrid Inheritance :

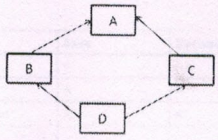
Combination of all types of inheritance is called as hybrid inheritance.

Diamond Problem:

What do u mean by diamond problem.?

What do u mean by virtual base class ?

What do u mean by virtual inheritance ?



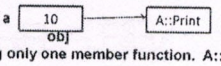
Constructor calling sequence A->B-> A->C->D

destructor calling sequence D->C->A->B->A

- **Class A is direct base class for class B and C.**
- **Class B and class C are direct base classes for class D**
- **Class A is indirect base class for class D**

• when we create object for class A it will get single copy of all the data members declared in class.

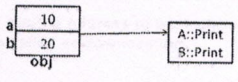
•A obj;



•A having only one member function. A:: Print()

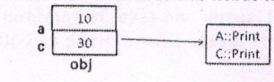
•Class B is derived from class A.

•When we cerate a object of class B. it will get single copy of all the non static data members declared in class B as well as class A

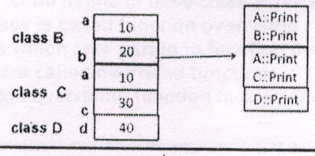


•Class C is derived from class A.

•When we cerate a object of class c. it will get single copy of all the non static data members declared in class C as well as class A



•Class D is derived class which is derived from class B and class C. This is called as multiple inheritance. When we crate a object of class D. it will get a single copy of all the data members of declared in class B as well as class C and class D.



In above diagram class A is indirect base class for class D. that why all the data members and member functions of class A are available twice in class D.

When we try to access these members of class A by using object of class D Compiler will confuse. (which members to be access members available from class B and class C)
Such problem created by hybrid inheritance is called diamond problem.

Solution for hybrid inheritance :

1. Explicitly mention name of the class which of which data member and member function do u want to access.

- D obj;
- cout<<"BA::"<< obj.B::a <<endl;
- cout<<"CA::"<< obj.C::a <<endl;
- obj.print();

- Declared base class as a virtual i.e. derived class B from class A virtually and derived class C from class A virtually
- class B : virtual public A
- class C : virtual public A
- Such type of inheritance is called as virtual inheritance.
- Now in this case (class D)will get single copy of all the data members of indirect base class .

Mode of inheritance
when we use private ,public , protected at the time of inheritance it is called as mode of inheritance.
class B : public A
here is mode inheritance is public.
In c++ by default mode of inheritance is private.

Mode of inheritance public:

	Base	Derived	Out side class
Private	A	NA	NA
Public	A	A	A
Protected	A	A	NA

A means Accessible NA means Not Accessible

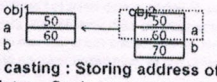
•In private mode of inheritance,
•public member of base becomes private members of derived.
•protected members of base become private members of derived.
•private members of base become private members of derived [not accessible in derived].

	Base	Derived	Out side class
Private	A	NA	NA
Public	A	NA	NA
Protected	A	NA	NA

In protected mode of inheritance,
public member of base becomes protected members of derived.
protected members of base become protected members of derived.
private members of base become private members of derived [not accessible in derived].

	Base	Derived	Out side class
Private	A	NA	NA
Public	A	A	NA
Protected	A	A	NA

Object slicing : when we assign derived class object to the base class object at that time base class portion which is available in derived class object is assign to the base class object. Such slicing (cutting) of base class portion from derived class object is called object slicing.
A obj1;
B obj2(50, 60, 70);



Up casting : Storing address of derived class object into bas class pointer . Such concept is called as up casting.
A * ptr ; B obj; ptr= &obj; or
A*ptr=new B();

Down casting : storing address of base class object into derived class pointer is called as downcasting.
B *ptr=new A();

•Virtual functions:
• function which gets called depending on type of object rather than type of pointer such type of function is called as virtual function.
•Class which contains at least one virtual function such type of class is called as polymorphic class. This is late binding.

•Late Binding :
•When call to the virtual function is given by either by using pointer or reference the it is late binding. In rest of the cases it is early binding
•Function overriding :
•Virtual function define in base class is once again redefine in derived class is called function overriding.
•Functions which take part in to function overriding such functions are called overrided functions.
•For function overriding function in base class must be virtual.

Function overloading	*Function overriding*
Function overloading is compile time polymorphism	Function overriding is run time polymorphism
Signature of the function must be different	Signature of the function must be same
For function overloading no keyword is required	For function overriding virtual keyword is required in base class
For function overloading functions must be in same scope . ie either function must be global or it must be inside the same class only.	Function must be in base class and derived class.
Function gets call by looking toward the mangled name	Function gets call by looking toward the vtable

- virtual function table
- When class contain at least one virtual function at that time complier internally creates one table which stores address of virtual function declared in side that class. Such table is called virtual function table or vftable or vtable

- Virtual function pointer
- When class contains virtual functions internally vtable is created by the compiler and to store address of the vtable compiler implicitly adds one hidden member inside a class which stores address of vtable such hidden member is called virtual function pointer / vfptr / vptr

- Vtable stores addresses of virtual function and vptr stores address of the vtable

- Pure virtual function
- Virtual function which is equated to zero such virtual function is called pure virtual function.

- Generally pure virtual functions do not have body.
- Class which contains at least one pure virtual function such type of class is called as called abstract class .

- If class is abstract we can not create object of that class. But we can create pointer or reference of that class.
- It is not compulsory to override virtual function but it is compulsory to override pure virtual function in derived class.
- If we not override pure virtual function in derived class at that time derived class can be treated as abstract class

- Abstract class can have non virtual member function, virtual functions as well as pure virtual functions

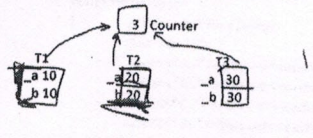
Static variables are same as global variable but limited scope .
int x=100;
static int y=200;
x can be accessible in any file but y will be accessible only in his scope.

Static is nothing but shared.
When you declared data member as static. It is compulsory to provide global definition for that static data member because static data member always gets memory before creation of object.

```
#include<iostream.h>
class Test
{       int _a;
        int _b;
        static int counter;
        public:
        Test()
        {       this->_a=0; this->_b=0; counter++;
        }
        void Print()
        {       cout<<"_a :: "<<this->_a<<" _b:: "<<this->_b<<endl;
                cout<<" counter" <<this->counter<<endl;
        }
};
```

```
int Test::counter =0; // global definition for static data member
int main()
{
        Test T1, T2, T3;
        T3.Print ();
        return 0;
}
```

Size of a object : size of object is size of all non static data members declared in that class. When we declared data member as static at that time instead of getting copy (memory) of static data member to each and every object all the object s share single copy of static data member available in data segment. That's why size of static data member is not consider in to size of object. If you want to shared value of any data member throughout all the objects we should declared data member as static.



Static member function: if we want to call any member function without object name we should declared that member function as static. Static member functions are mainly designed to call using class name only, but we can call static member function by using object name also.
Static member functions can access only static data. Static member functions do not have this pointer.
When we call member function of class by using object implicitly this pointer is pass to that function. When we call member function of class by using class name implicitly this pointer is not pass to that function. Static member functions designed to call by using class name only that's why this pointer is not pass to static members functions of class.

Members that we can call using object of class is called *instance members.*
Members that we can call using class is name these are called as class *members.*

```
#include<iostream.h>
class TChair
{
private:
        int _height;        int _width;        static int _price;
public:
        TChair()
        {  this->_height=20;
           this->_width=25;
        }
        TChair(int height, int width)
        {  this->_height=height ;
           this->_width=width ;
        }
        static void Setprice(int price)
        {    TChair::_price =price;
        }
        void Print()
        {          cout<<"Height::"<<this->_height<<endl;
                   cout<<"Width::"<<this->_width<<endl;
                   cout<<"Price::"<<this->_price<<endl;
        }
};
```

```
int TChair::_price  =125; // global defination for static data
member
int main()
{
        TChair ch1, ch2, ch3;
        ch1.Print ();
        ch2.Print ();
        ch3.Print ();

        TChair ::Setprice (200);

        ch1.Print ();
        ch2.Print ();
        ch3.Print ();

        return 0;
}
```

Constant Data Member:

in c language it is not compulsory to initialize constant variable at the time of declaration because we can modify value of const variable using pointer. In c++ it is compulsory to initialize const variable at a time of declaration other wise compile time error will occur.

We cannot initialize data member at the time of declaration. If you want to initialize it in constructor.

```
Test():a(10), b(20)
{
}
Test(int a, int b):a(a), b(b)  // constructor member initialize list
{
}
```

when we declare data member as constant it is compulsory to initialize that data member inside constructor initialiser list

Constant member function

we can not declared global function as const. we can declared member function as a const in c++. When we declared member function as const we can not modify the state of as object only with in that const member function.

We should declared member function as a const in which we are not modifying the state of object.

If you want to modify state of the object inside constant member function at that time declared data member as a mutable.

```
#include<iostream.h>
class Test
{       const int a;
        const int b;
        int c;
        mutable int d;
public:
        Test():a(10),b(20)
        {       c=30;  d=40;
        }
        Test(int a, int b, int c, int d):a(a), b(b)
        {       this->c=c; this->d=d;
        }
        void Print() const
        {       //this->c=199; //error
                this->d=199;
                cout<<"a::"<<this->a<<endl;
                cout<<"b::"<<this->b<<endl;
                cout<<"c::"<<this->c<<endl;
                cout<<"d::"<<this->d<<endl;
        }
};
```

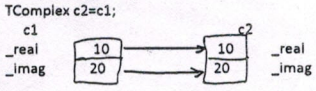
```
int main()
{
        Test t1;
        t1.Print ();
        Test t2(1,2,3, 4);
        t2.Print ();

        return  0;
}
```

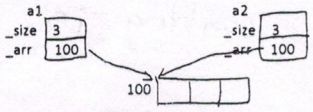
generic can catch any data type

Shallow copy :
when we assign one object to another object at that time copying all the contents from source object to destination object as it is. Such type of copy is called as shallow copy.
Complier by default create a shallow copy. Default copy constructor always create shallow copy.

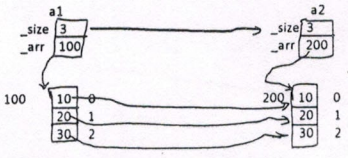
eg. TComplex c1(10, 20);
TComplex c2=c1;



eg;
TArray a1(3);
a1.Accept();
TArray a2= a1;
a1.Print();



Deep Copy : when class contains at least one data member of pointer type , when class contains user defined destructor and when we assign one object to another object at that time instead of copy base address allocate a new memory for each and every object and then copy contains form memory of source object into memory of destination object. Such type of copy is called as deep copy.



Condition for deep copy:
1. Class must contain at least one data member of pointer type.
2. class must contain user define destructor
3. one object must be assigned to another object.

Exception Handling:
runtime error which can be handled by programmer is called exception.
In c++ exceptions are handled using try, catch and throw.

```
#include<iostream.h>
int main()
{       int x=10,y=0;
        try
        {       if(y==0)
                {  throw 1;
                }
                else
                {       int res=x/y;
                        cout<<"res::"<<res<<endl;
                }
        }
        catch(int)
        {       cout<<"Enter Other Than 0"<<endl;
        }
        return 0;
}
```

Try block must have at least one catch handler.
One try block can have multiple catch blocks .
When exception is thrown but matching catch block is not available at that time compiler give call to library defined function terminate which internally give call to abort function.
Catch block which handles all kind of exceptions such type of catch is called generic catch block.

```
catch(...) (ellipse)
{
        cout<<"inside genric block"<<endl;
}
```

catch handler for ellipse must last handler in exception handling

Generic Programming- Template

Many times we need to write the code that is common for many data types. The simplest example is Swap() function. We can observe that logic of swapping remain same irrespective of the data type used.

We can write templates of function, instead writing separate function for each data type. The syntax is as follows:

```
template<class T>
void Swap(T& a, T& b)
{       T c = a;
        a = b;
        b = c;
}
```

Now the same function can be used to swap, two integer variables of float variables or even any user defined type's variables.

If function is called as:
Swap(a, b); // where a and b are int variable

Compiler converts above template function, to the function for swapping integer variables. To do this, compiler simply converts above function to a new function by replacing "T" with "int". Thus compiler creates the functions corresponding to every data type for which function has been called.

The way we can use template functions to represent generic algorithms, we can write template class to represent generic classes. The best examples are data structure classes like stack, queue, linked list, etc. Template classes are also called as "meta classes", because compiler generates real classes from template classes and tnen create the objects.