Imp points of C++

★ 90 to 92% Work is Done on Object oriented @ IT industry

→ C++ is object oriented prog. languages.

→ C++ is thinking process rather than prog. lang.

→ C++ is designed by Bjarne Stranstrup in 1981 at AT&T

→ C++ is Influenced from :-

1] Small Talk ⇒ Concept of classes is derived for SmallTalk

2] Simula 67

3] C ⇒ Structural & Syntactical approach derived from C

→ C++ is thinking process but it is used in Team development.

Multiple Peoples Working on same project

* Procedural is unmanagable @ some point i.e. 'C'

*So Object Oriented ~~it~~ concept ~~for~~ team Dev.

Imp

→ After getting problem statement ? Client सहायता

Calculator ~~चा~~ Exa.

Requirement Analysis

Ask 3 Question ⇒

(1) Whether C++ is Applicable or Not?

⇒ If Ans Yes ↓

(2) Which (All) things are required to solve these problem?

class { ② characteristics ③ Behaviors }

(3) How these things can be applicable to solve the problem statement? (After getting all these things)

P^n सोडवायला जाऊ शकतो

MVC ⇒ Module View Control

PAGE :
DATE : / /

Class :-

Class is user datatype data which almost similar as concept of structure in 'C'.

Class is collection of characteristics & behavior.

class = Characteristics + Behavior.

Exa :- Mobile

★ C++ Programming ★

10-Oct-2015

Platforms → Hardware, O.S.

★ There are multiple types of programming languages-

1. Procedural Language -
   C, Win32 SDK (C in Windows)
   (Superset meaning)
2. Object Oriented -
   C++, COM
   (Component Object Model)
3. Virtual Machine Based Lang -
   Java, .Net.
4. Scripting Languages :- Python,
   Php,
   Perl.

★ Important Point of C++ :-

"C++ is not a prog lang but it is thinking Process"

Live Examples ☺

PASSION

1978 to 1981

Bjarne Stroustrup

1) Small talk ⇒ 1st oops

2) Simula 67 ⇒ (Simulation)

(A) Classes ⇒ concept

3) 'C' prog lang ⇒

(B) Structural & Syntactical Approach

→ 1978-1981 ⇒ C with classes नावाने Design झाली

→ 1997 ला standardization ला गेली.

Prog Lang ला Multi Word नाव चालत नाही (Name) Discarded.

Some person sujested Name as '++C'

But this indicate 1 step Ahead & ++opr by C++

But strastup hasn't aggred on it

++ pre increment ⇒ so '++C'

C++-97
C++-03
C++-07
C++-11
C++-14 ★ Recent

If we are influenced from 'C'

so ⇒ 'C++' ⇐ i.e. Post increment

Name final झाले

★ 'C' front नावाचा Compiler Design केला

C++ Prog → To C front compiler → C Prog मिळेल → C Compiler → C-ft Toolchain

As it is

☆ Important Points About Class:-

11/Oct/24.

1) Class is considered as user defined datatype in c++.

2) Concept of class is similar as structure of c & c++.

3) If our prog. is object oriented prog then it should contain atleast single class.

4) After performing the requirement analysis we have to decide which things are required to fulfill that requirement.
Answer of this quetion becomes characteristics of our class.

5) After getting the appropriate characteristics means the data member. We have to think about the functions or methods which are required to fulfill that requirement.
This methods or fns are considered as behavior of our class.

6) After deciding the behavior we have to check whether that kind of similar behavior is available in any other class or not.
If it is available in any other class then we have to inherit [Reuse] that class

7) After finalizing characteristics & behavior of our class we have to apply appropriate access specifier to every characteristics & behavior.

- 8) After class declaration we have to write (;) semicolon. bcz, class is user defined datatype.
And after declaration of every userdefined datatype there should be (;)

- 9) Name of the class should be meaningful.

- 10) Identifier name of every characteristics & behavior of class should be distict to avoid ambiguity error.

- 11) We can't initialize any characteristics inside class declaration bcz, at that time memory is not allocated for that perticular characteristics.

Example:-

```
class demo
{
    int i = 11; // error.
};
```

- 12) To initialize the characteristics of class we have to write appropriate constructor in our class.

Que * Problem Statement - 11/Oct/2015

W.A.P. which is used to accept 2 integers from user & display its addition & substraction.

⇒ * Approach ① - [Procedural Approach] (C)

```
#include<stdio.h>

int Add(int no1, int no2)
{
    int iAns=0;

    iAns = no1+no2;
    return iAns;
}
int Sub(int no1, int no2)
{
    int iAns=0;
    iAns = no1-no2;
    return iAns;
}
int main()
{
    int iNum1, iNum2, iRes;

    printf("Enter 2 Nos:");
    scanf("%d,%d", &iNum1, &iNum2);

    iRes = Add(iNum1, iNum2);
    printf("Addition is=%d", iRes);

    iRes = Sub(iNum1, iNum2);
    printf("Substraction is = %d", iRes);

    return 0;
}
```

4/Oct/2015

In this procedural approach we are using diff. procedure such as,

1) Accept I/p from user.
2) Perform opr^n on that i/p.
3) Return result to the user.

As it is a procedural approach it should be invoked in sequence.

If we want to perform this tasks again we have to follow same set of procedures again & again.

* Approach 2 - [Object Oriented Approach (c++)]

```
using namespace std
# include<iostream>

class Maths
{
    public:
        int iNo1, iNo2, iAns; //characteristics  (Data members)

        void Add();
        void Sub();           // Behavior or Methods
};

void Maths :: Add()
{
    iAns = iNo1 + iNo2;
}
void Maths :: Sub()
{
    iAns = iNo1 + iNo2;
}
```

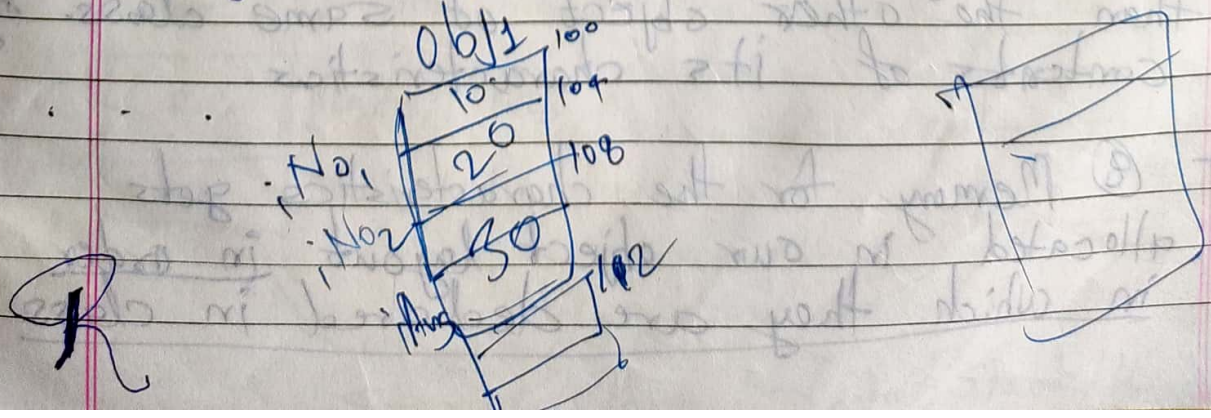
11/Oct/2015

```
int main()
{
    Maths obj1, obj2;

    obj1.iNo1 = 10;
    obj1.iNo2 = 20;     // Initialization of characteristics

    obj2.iNo1 = 20;
    obj2.iNo2 = 10;

    obj1.Add();

    cout << obj1.iAns;      // 30

    obj2.Sub();

    cout << obj2.iAns;      // 10

    return 0;
}
```

In this object oriented approach we are using Maths class which contains 3 characteristics & 2 behavior.

Inside main() we are creating 2 objects of that class as obj1 & obj2.

In this approach we are calling the member fn of Maths class by using the appropriate object.



☆ Important Points about class:-

11/Oct/2015

PAGE :
DATE : / /

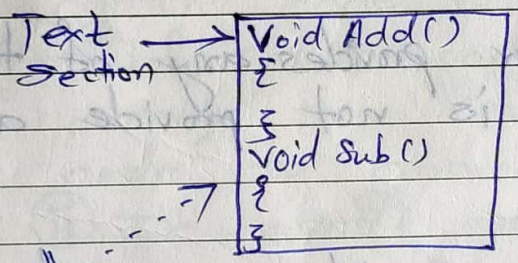
- ① When we declare a class memory is only allocated for its behavior inside text section.
- ② When we create object or instance of class then separate memory gets allocated for its characteristics.
- ③ If we create N objects then memory gets allocated N times for its characteristics only.
- ④ If we create 'N' objects then memory for behavior gets allocated only once, in text section.
- ⑤ Memory for behavior gets allocated only once bcz, the contents of behavior are non modifiable due to which it can be shared betn all the objects of same class.
- ⑥ If we change characteristics of any of the object then that change is not reflected inside characteristics of other object.
- ⑦ Every object of same class is different than the other object of same class. by contents of its characteristics.
- ⑧ Memory for the characteristics gets allocated in our object layout in order in which they are declared in class.

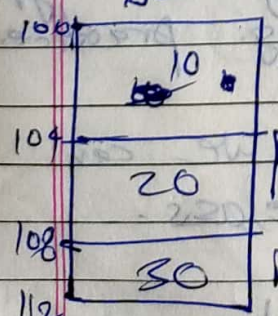
- (9) In C++ there are 2 types f^ns as,
  (A) Member f^n:- If it is declared inside the class.
  (B) Nested f^n:- If it is outside the class.

- (10) If we want to call any member f^n of class then object of that class is required to call that f^n.

- (11) If our f^n is member f^n then that f^n can access any of the characteristics of that class directly by using its name.

Consider the below Memory layouts of objects Obj_1 & Obj_2 ⇒





Depends on Storage class

Data/Stack/Heap

11 Oct 2014

Constructor in C++ :-

PAGE :
DATE : / /

1) - Constructor is user defined member f^n of class

2) - Constructor is such a f^n which gets called implecitely when we create object of that class.

3) - Call to the constructor is provided by the compiler.

4) - There are 2 types of constructors in c++

[A] Default Constructor -
[B] Parameterized constructor -

5) - If programmer is unable to provide the constructor then it gets provided by the compiler.

6) - If programmer provides any of the constructor then compiler is not provide any other constructor.

7) - If programmer provides parameterized constructor then compiler is not going to provide default constructor.

Due to which writting all types of constructor is good prog. practice

8) - Like other user defined f^n we can define our constructor outside class.

9) - Name of constructor should be same as class name,

bcz it gets implicitely called

by the compiler.

10) - If we use any other name to the constructor then our compiler is unable to predict the constructor.

11) - There should not be any return value to the constructor bcz, There is no explicite caller to the constructor.

12) - Constructor gets involked after allocating memory of our objects

13) - Constructor is used to allocate resources which are required in our class

exa - (i) If class uses database activity then constructor should contains code to connect our database
(ii) If class contains networking activity then constructor should contain code to connect our socket.
(iii) If class contains graphical activity then code which is used to get device context & used to get capabilities of our device should be written in constructor.
(iv) If any dynamic memory is required inside our class then memory allocation code should be written in our constructor.

14) All the opposite task should be performed in destructor as compared to constructor.

15) — If we create 'N' objects of same class then constructor gets invoked N times.

16) — Inside the constructor we can call any of the member fn of same class.

17) — Inside constructor we can access every characteristics of same class.

18) — Every constructor should be written under public access specifier bcz,
If it is written under private then compiler generates error bcz, that constructor is unaccessible to the compiler.

19) — We can overload the constructor & like normal fn overloading concept.

20) — We can't apply & pointer to our constructor.

21) — We can't use concept of virtual with the constructor (But it can be used with destructor.)

22) — Concept of constant is not applicable for constr as well as destructr.

23) — Concept of static is not applicable for constructr as well as destructr

24) — We can access this pointer from constructor also.

29) We can't apply concept of friend with constructor.

28) We can't apply concept of inline with constructor

27) Inside constr we can also create another object.

Lecture 3 13/Oct/2015

☆ Polymorphism

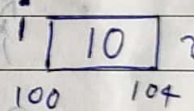
17/Oct/2015

★ Reference in C++ :-

- Concept of reference is introduced by c++.
- Concept of reference is only used in c++.
- Concept of reference is similar as, concept of pointer in c.
  Not same
- This concept of reference is as it is used in Java & .Net
- In c++ we can create reference to anything which we can access.
- Reference is just considered as a another name to an existing identifier.

- Syntax:

```
int i = 10;
int &ref = i;
pf("%d", ref);
```



In this syntax 'i' is considered as, a variable of type integer & which is initialised to value 10.

'ref' is considered as a reference which is used to refer an integral value & it is currently refering to an variable 'i'.

Reading ⇒

- According to above point we can conclude that there is no separate memory allocation for a reference variable

- When we create a reference variable its entry gets added in symbol table

17/Oct/2015

- Name of our reference variable gets added in another name column of symbol table as,

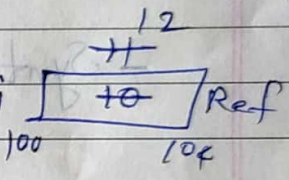
Name	Addr	Value	From (Scope / LineNo)	To	Another Name
i	100	10	7	15	Ref

Symbol Table

- As memory is shared bet^n our original variable & reference variable, If we modify the contents of original variable or reference variable then the actual contents gets modified in a memory

Exa:

```
int i=10;
int &Ref = i;
Ref++;
i++;
printf("%d %d", Ref, i); // 12 12
```



- When we create ref. variable the name of that referenced variable should be ~~declared~~ mentioned at the time of declaration.

```
int i=10;
int &Ref; // error
int Ref=i; // error
```

- According to above ~~syntax~~ it is necessary to initialize reference variable at the time of declaration as,

```
int i=10;
int &Ref = i;
```

- In c++ '&' operator is used in two scenarios ⇒

1) If & operator is used after assignment opr (i.e. =) then it is considered as address of operator.

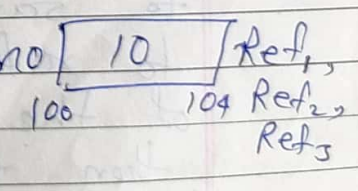
Exa- int no = 10;
int *p = &no;

2) If & oprator is used before assignment opr (i.e. =) then it is considered as reference opr.

Exa- float f = 5.14;
float &Ref = f;

⇒ Like pointer in 'C' we can create multiple references which are going to refer a single variable as,

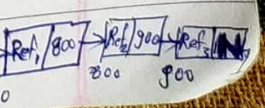
int no = 10;
int &Ref1 = no;
int &Ref2 = no;
int &Ref3 = Ref1;



- In this syntax Ref1, Ref2 &Ref3 are considered as reference variable which are refering to single variable i.e. no.

Name	Addr.	Value	Scope From	Scope To	Another Name
i	100	10	5	17	700

Symbol Table



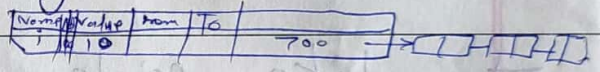
— In C++ 11 we can create any number of references to single variable.

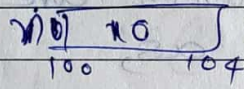
— Like pointer to pointer in C we can also create reference to reference in c++. as,

```
int no = 10;
int &Ref1 = no;
int &Ref2 = Ref1;
int &Ref3 = Ref2;
```

In this example Ref_3 is refering to Ref_2, Ref_2 is Refering to Ref_1 & Ref_1 is refering to no.

Diagram of symbol table for this example is exactly same as above diagram.

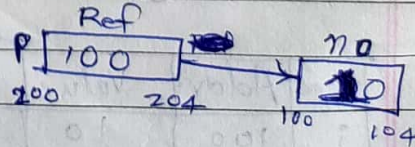




— If there are multiple references to same variable or there are reference to reference then we can change any of the value of the reference variable then that change gets reflected to the original variable.

— We can also create reference to a pointer as,

```
int no = 10;
int *p = &no;
int* (&Ref) = p;
```



11/Oct/2015

printf("%d %d %d", no, *p, *Ref); // o/p → 10 10 10

In this syntax Ref is considered as reference variable which refer to an integer pointer & currently it is refering to pointer 'p'.

- We can access the contents of no by using pointer p as well as by using Reference variable ref.

Exa:

— As there is no separate memory allocation for reference variable due to which address of our original variable & address of reference variable is same.

Exa:

```
double D = 7.67;
double &Ref = D;

printf("%u %u", &D, &Ref);
// 100 100
```

D [7.67] Ref
100 108

⁂ — Type of reference variable & refered variable must be same otherwise it generates compile time error.

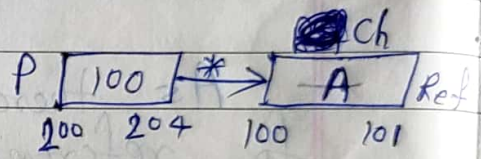
```
int no = 10;
float &Ref = no; // error
```

- We can't initialize constant value to reference variable direcly as

```
int no = 10;
int &Ref = 11;   // Error.
```

- As we can fetch address of reference variable we can also apply pointer to that reference variable

```
char ch = 'A';
char &Ref = ch;
char *p = &Ref;
```



- As there is no separate memory allocation for the reference variable we can't create array of reference variables as,

```
int &(Arr[5]);   // Error
```

If we try to read the above statement it indicates that Arr is one D Array which contains 5 elements & each element is reference to an integera.

As we can't create array of references,

But we can create reference to an Array as,

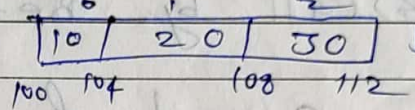
```
int A[3] = {10,20,30};
```

```
int [3] (*B) = A;
```

OR

```
int (*B)[] = A;
```

★ We can read the above syntax as, B is reference which refers to an 1-D array which contains 3 elements & each element is of type integer.




```
printf("%d%d%d", A[0], A[1], A[2]); //102030
printf("%d %d %d", B[0], B[1], B[2]);
//10 20 30
```

Function Calling Techniques In C++

* There are 3 way in ways in which we can call the f^n in C++, as,

1) Call by value -
2) Call by Address -
3) Call by Reference -

Concept of call by reference is only applicable in C++ but, concept of call value & call address is used in C & C++.

* If cally f^n is going to change the contents of i/p argument which are passed by the caller f^n & if our caller is interested in that change then we have to use either call by address or call by reference

Function Calling Techniques in C++

Caller

1) // Call By Value

```
int main()
{
    int no = 10;
    printf("%d", no); //10
    value(no);
    printf("%d", no); //10
    return 0;
}
```

no 10 (100 – 104) — 10

2) // Call By Address

```
int main()
{
    int no = 10;
    printf("%d", no); //10
    Address(&no);
    printf("%d", no) //11
    return 0;
}
```

100

no 11 (100 – 104)

3) // Call By Reference

```
int main()
{
    int no = 10;
    printf("%d", no); //10
    Reference(no);
    printf("%d", no); //11
    return 0;
}
```

10

no 11 Ref (100 – 104)

Cally

```
void value(int i)
{
    i++;
}
```

i 11 (200 – 204)

```
void Address(int *p)
{
    (*p)++;
}
```

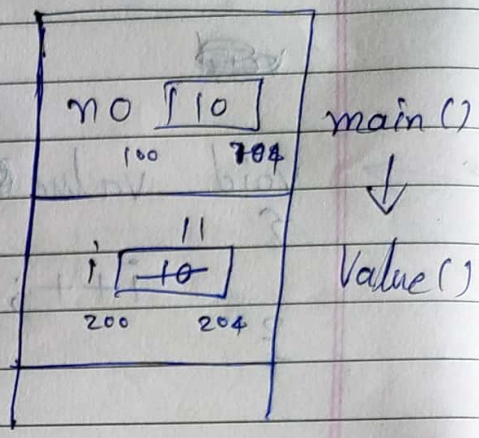
P 100 (200 – 204)

```
void Reference(int &Ref)
{
    Ref++;
}
```

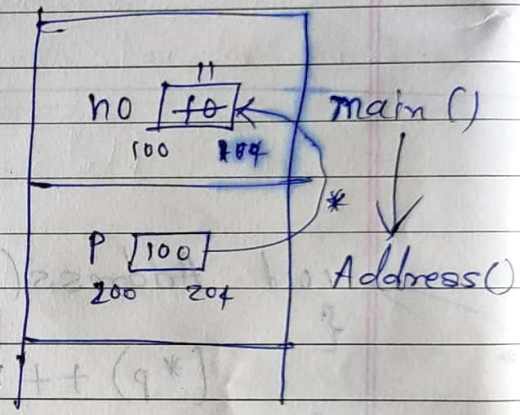
Stack layouts of fn calling techniques:-

Stack frames for Calling Techniques programs

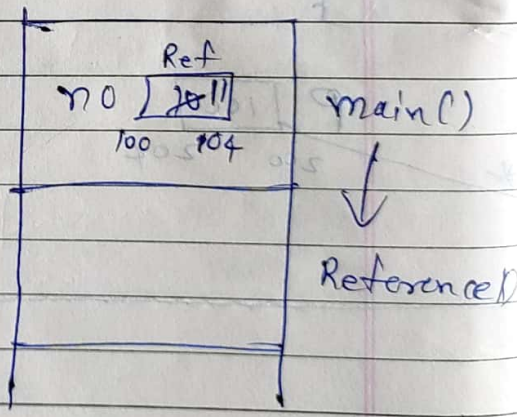
1) call By Value:-



2) Call By Address:



3) Call By Reference:



Constructor Continued:-

17/06/2015

PAGE:
DATE: / /

If object is already created if you want to create new object which is exactly same as an existing object then we have to provide one type of constructor called as copy constructor.

If we want to provide copy constructor then i/p argument of that copy constructor should be reference of the object of same class.

Consider the below syntax in which we are providing all the type of constructors as,

1) Default Constructor -
2) Parameterized Constructor -
3) Copy Constructor -

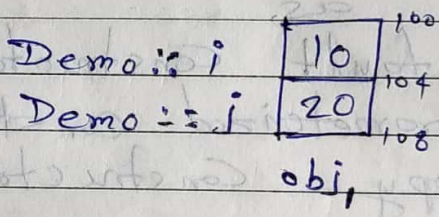
```
class Demo
{
    public:
        int i,j;    // characteristics

        Demo()
        {                    // Default Constr
            i=10;
            j=20;
        }
        Demo(int x, int y)
        {                    // Parameterized constr
            i=x;
            j=y;
        }
```

```
Demo(Demo &Robj)          // Copy Constructor
{
    i = Robj.i;
    j = Robj.j;
}
};
```

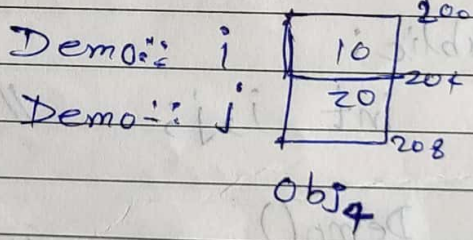
In above class there are all types of constructors.

* We are considering some object creation syntax & its appropriate constructor call as

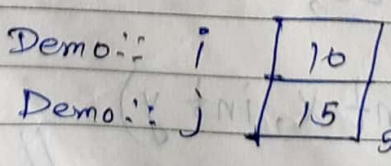
1) Demo Obj1; // default const^r call.



2) Demo Obj4 ( ); // Default const^r call



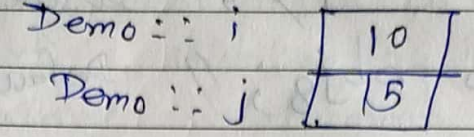
3) Dem Obj2 (10,15); // Parametrized const^r call



Obj2

4) Demo

Demo Obj3(Obj2); // Copy const^r call



obj3

When we use this syntax copy const^r gets invoked implicitely which is used to copy the contents of one object into other object.

In this syntax obj2 is already created object & we are passing that object while creating new object as obj3.

In this syntax of copy const^r obj2 is passed as a name which gets collected in a reference variable named as Robj.

In def^n of copy const^r we are copying the contents of Reference object into our newly created object.

There is another scenario in which the copy const^r gets invoked as,

Demo (Obj5); // default const^r call

Demo Obj6 = obj5; // copy const^r call

According this Syntax when we initialize the object at the time of creating that object copy const^r gets invoked implicitely.

In below syntax copy constr is not invoked as,

```
Demo obj7; // Default constr call
Demo obj8; // Default constr call
```

```
obj8 = obj7; // Equal to operator.
```

Not copy constr called.

In this syntax copy constr is not invoked bcz, both the objects are already created.

When we use '=' equal to opr at that point. Overloaded equal to opr's fn gets called implicitly.

Instead of calling copy constr.

★ Destructors In C++:-

18/Oct/2015

PAGE:
DATE: / /

- 1) Destructor is considered as a user defined f^n which gets implicitely called before memory deallocation of our object.
- 2) Destructor is used to free the resources which are allocated in constructor.
- 3) It is not compulsury to provide destructor but it is good programming practice to provide the destructor.
- 4) Like constructor we can write anything inside destructor.
- 5) There are no types of destructors.
- 6) We can not pass any parameters to the destructor.
- 7) Destructor can not return any value from its definition.
- 8) Name of destructor should be same as class_name.
- 9) We have to write telda (~) op^r before name of destructor.
- 10) (~) Telda op^r is selected for the destructor bez, it performs opposite tasks as compared to the constructor.
- 11) If destructor is not provided by the programmer then compiler is not providing its own destructor.
- 12) If we write destructor in non public section ie. private or protected compiler will generate error at that point, bcz that sections are not accessible to the compiler.
- 13) Generally destructor is used to perform reverse activity as compared to the constructor as,

18/Oct/2023

- 1) If database connection is performed in constructor then it should be disconnected in destructor.
- 2) If socket is created in the constructor then that socket should be released in destructor.
- 3) If device context is allocated in constructor then it should be deallocated in destructor.
- 4) If dynamic memory is allocated in constructor then that memory should be freed in destructor.

* We can predict the life cycle of our object as -
  - 1) When we create the object memory is allocated for that object.
  - 2) After allocating the memory constructor gets called immediately.
  - 3) At this stage we can access any characteristics or behavior of that class.
  - 4) When lifetime of our object gets finished according to the storage class destructor gets called implicitely.
  - 5) After calling destructor physical memory gets deallocated by the memory manager of O.S.

★ class design to explain the concept of constructors, Destructors & member fns

```
#include<iostream>
#include<fstream>
using namespace std;
#include<stdio.h>
#include<string.h>

class FileWrap
{
    public:
        FILE *fp;
        char fname[50];
        char *ptr; int len;
        FileWrap();
        FileWrap(char*, int);
        FileWrap(FileWrap &);
        void ReadFile(int);
        ~FileWrap();
};
```

// characteristics

// constructors

// Behavior (member fn)

// Destructor

// Definitions of all the behaviors of a class.

```
FileWrap::FileWrap()                    // Default Constructor
{   len = 256
    fp = fopen("Demo.txt", "r");
    strcpy(fname, "Demo.txt");
    ptr = (char *) malloc(256);
}

FileWrap::FileWrap(char *name, int size) // Parameterized Constructor
{   len = size;
    fp = fopen(name, "r");
    strcpy(fname, name);
    ptr = (char *) malloc(size);
}
```

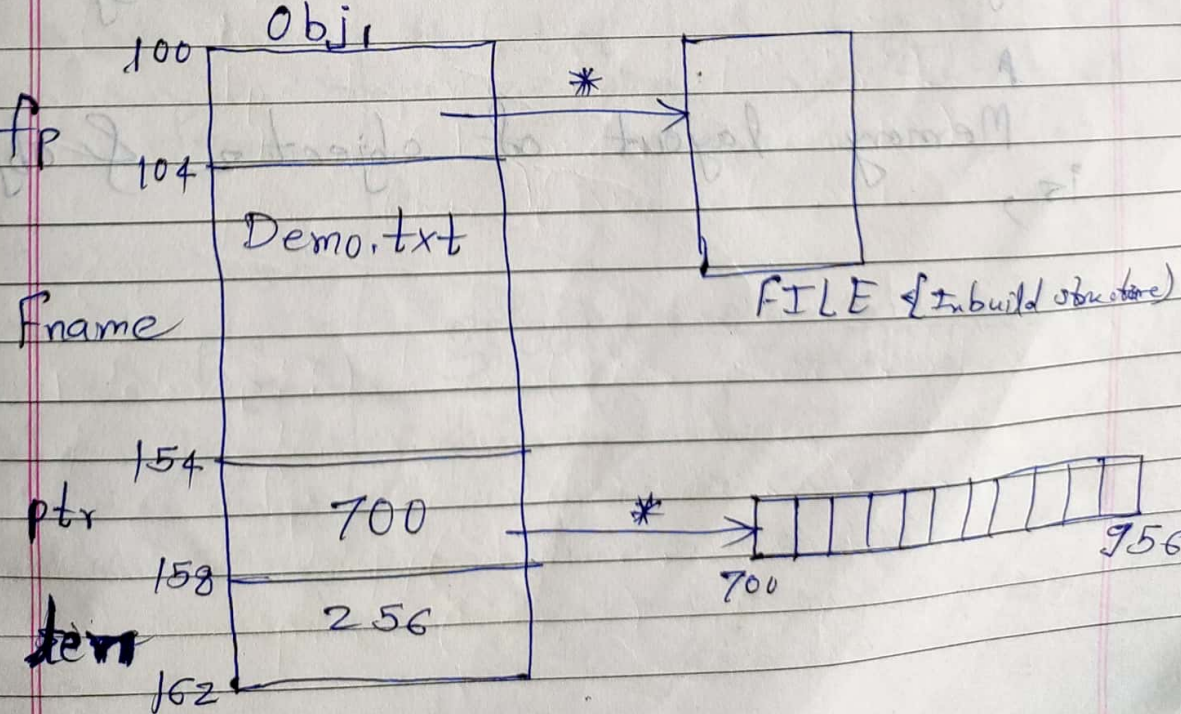
```
FileWrap :: FileWrap(FileWrap &Ref ) // Copy Constructor
{
    fp = fopen(Ref.fname, "r");
    strcpy(fname, Ref.fname);
    ptr = (char*) malloc(Ref.len);
    memcpy(ptr, Ref.ptr, len);
}

FileWrap :: ~FileWrap() // Destructor
{
    fclose(fp);
    free(ptr);
}

void FileWrap::ReadFile(int cnt) // Behavior (memfn)
{
    fread(fp, ptr, cnt);
}

int main()
{
    FileWrap obj1; // Default constr
    FileWrap obj2("hello.txt", 1024) // Parameterized Constr
    obj2.ptr = "India is my country";
    FileWrap obj3(obj2); // Copy constr
    printf("%s", obj3.ptr); // India is My Country
    obj2.ptr = "All Indians are My Brothers";
    printf("%s", obj2.ptr); // All Indians are My Brothers
```

```
    printf("%s", obj3.ptr); // India is My Country
    obj1.ReadFile(20);
    return 0;
}
```

** Important Points about above program :-

1) This class is the practical example of industrial object oriented language
2) This class is used to perform file handling activities.
3) This class contains multiple characteristics which are required to perform file handling operations.
4) This class contains all the types of constructors & the appropriate destructor.

* Consider object layout of obj1 which is created by invoking default constructor -



- Size of this object is 62 byte in which there are 2 pointers which are pointing to some external memory.

- This object obj_1 is created by invoking default constructor.

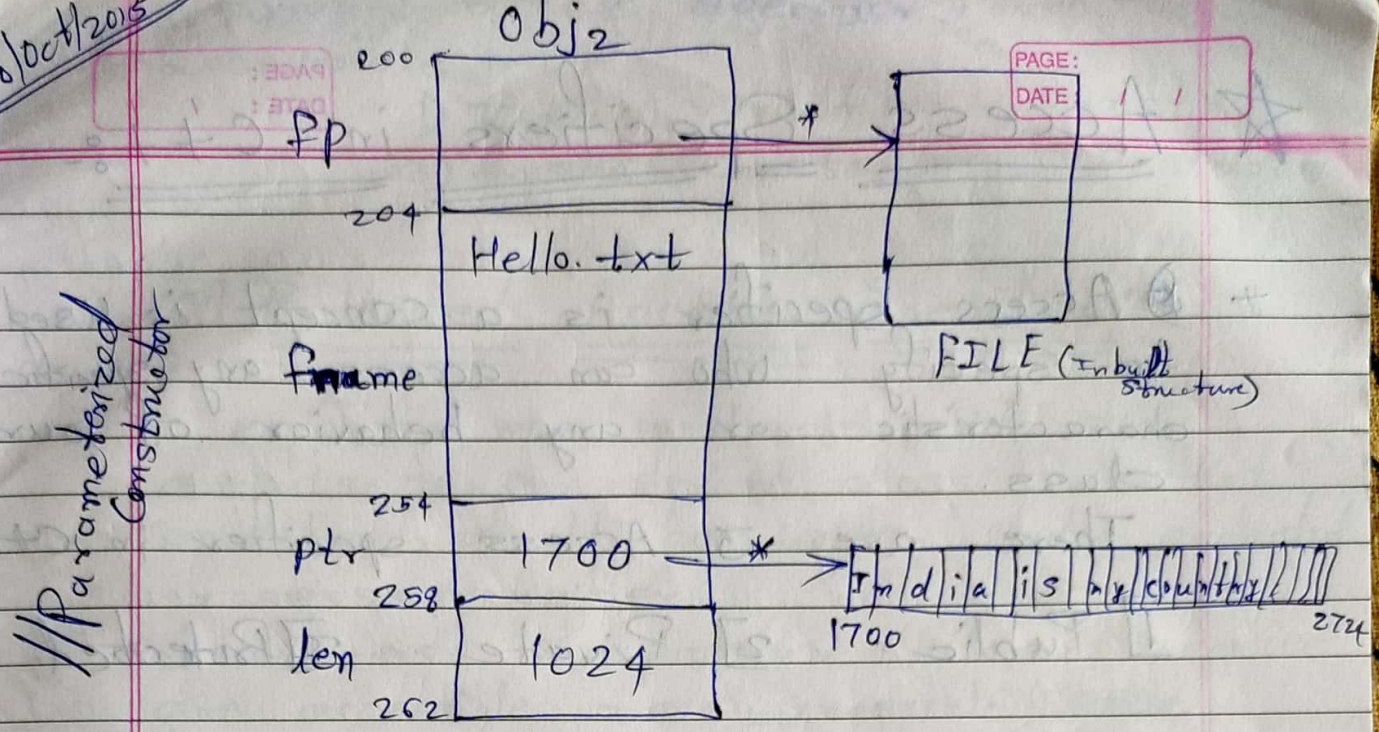
* obj_2 is created by invoking parameterized constructor by passing some specific arguments to it.
* obj_3 is created by invoking copy constructor. Due to copy constructor all contents gets copied from object2 to object 3.

In this copy constr we are using the concept of deep copy, due to which separate memory gets allocated by using malloc function & contents of obj_2 are copied inside dynamically allocated memory.

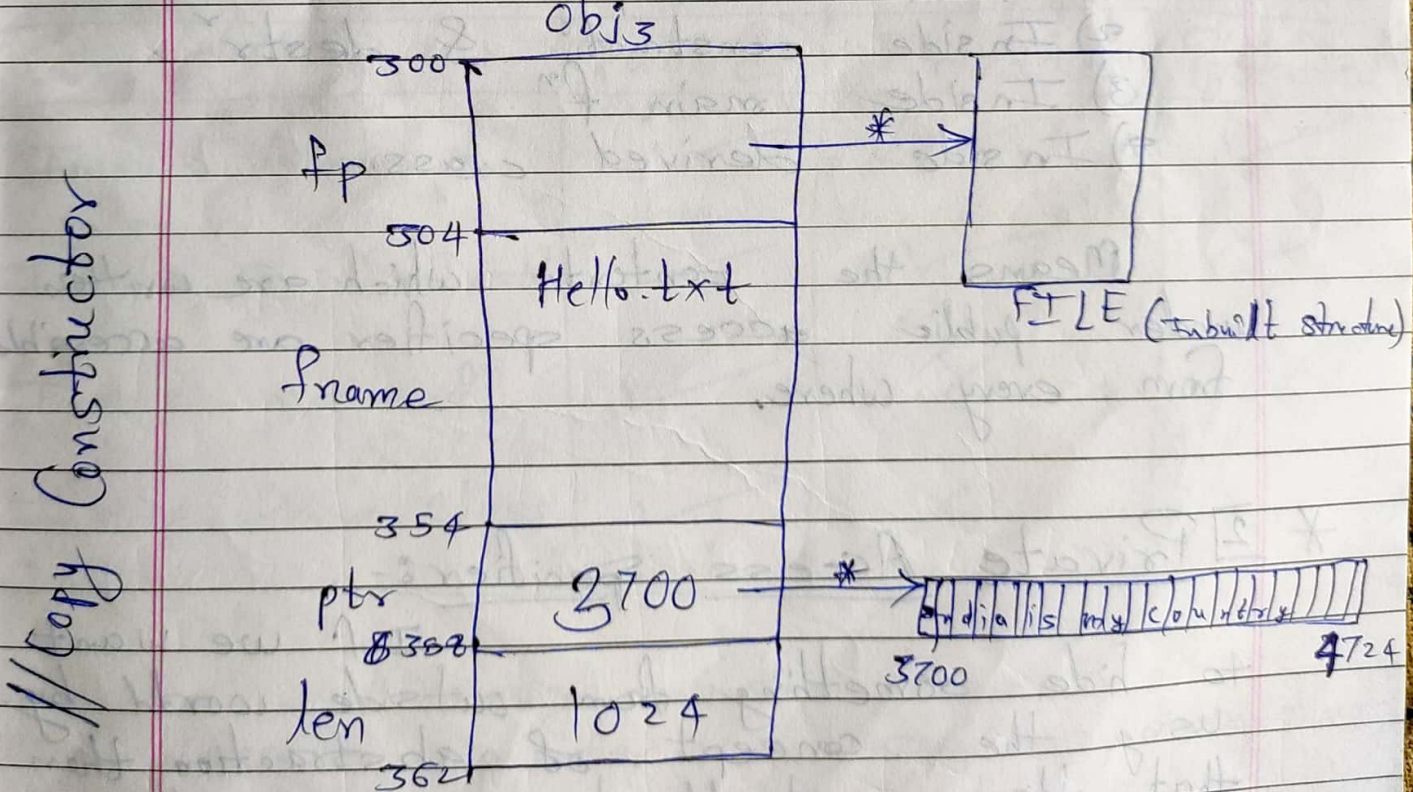
Due to concept of deep copy contents which are modified by object2 are reflected to object3.

Memory layout of object2 & object3 is,

18/Oct/2015



//Parameterized Constructor



//Copy Constructor

☆ Access Specifiers in C++:-

* Access specifier is a concept is used to specify who can access any specific characteristic or any behavior of our class.

There are 3 Access specifier in C++

1] Public 2] Private 3] Protected.

* 1] Public Access Specifier:-

The content which are written under public access specifier are accessible,

1) Inside member fn
2) Inside constructor & destr
3) Inside main fn.
4) Inside derived class.

Means the contents which are written under public access specifier are accessible from every where.

* 2] Private Access specifier:-

If we want to hide something from outside world by using the concept of abstraction then that things should be written under private access specifier.

Contents which are written under

private access specifier
accessible →
1) Inside member function
2) Inside Constructor & Destructor

* 3] Protected Access Specifier:-

If we want to provide some contents to our derived class then that contents should be written under protected access specifier.

Contents of protected Access specifier are accessible ~~are accessible 1) inside member~~

1) Inside member fn
2) Inside constructor & destructor.
3) Inside derived class.

	Access Specifiers		
	Public	Private	Protected
Inside Member fn	✓	✓	✓
Inside Constr n Destr	✓	✓	✓
Inside main fn	✓	X	X
Inside Derive class	✓	X	✓

According to the concept of access specifier we can conclude that rules of access specifiers are applicable for outsiders only.

But we can access everything inside the class irrespective of access specifier.

Inheritance

18/Oct/2015

PAGE :
DATE : / /

If we want to reuse something then we have to use concept of inheritance from c++.

Inheritance is considered as major object oriented paradigm in c++.

If any of the class is already designed then while designing the new class we can reuse the functionality of existing class.

Consider the below example which contains 2 classes as Base class & Derived class.

Derived class is derived from Base class.

Base class contains one method which is used to add 2 integers.

According to new requirement we have to provide such a f^n in derived class which is used to add 3 integers.

```
class Base
{
    public:
        int Add(int no1, int no2)
        {
            int Ans = 0;
            Ans = no1 + no2;
            return Ans;
        }
};
```

```
class Derived : public Base
{
    public:
        int myAdd(int no1, int no2, int no3)
        {
            int Res = 0;
            Res = Add(Add(no1, no2), no3);
            return Res;
        }
};
int main()
{
    Base bobj;
    bobj.Add(10, 20);          // o/p 30
    Derived dobj;
    dobj.MyAdd(10, 20, 30);    // o/p 60
    return 0;
}
```

Derived class is reusing the functionality of base class by using fn Add().

19/Oct/2015

★ Types of Inheritances:-

There are 2 types of inheritances
(A) According to the Hierarchy.
~~1) Hierarchy~~
~~2) According to~~
(B) According to Access Specifier.

★ [A] Type of Inheritance According to Hierarchy:

1. Single level Inheritance-
2. Multi level Inheritance-
3. Multiple Inheritance-

[1] Single level Inheritance:-

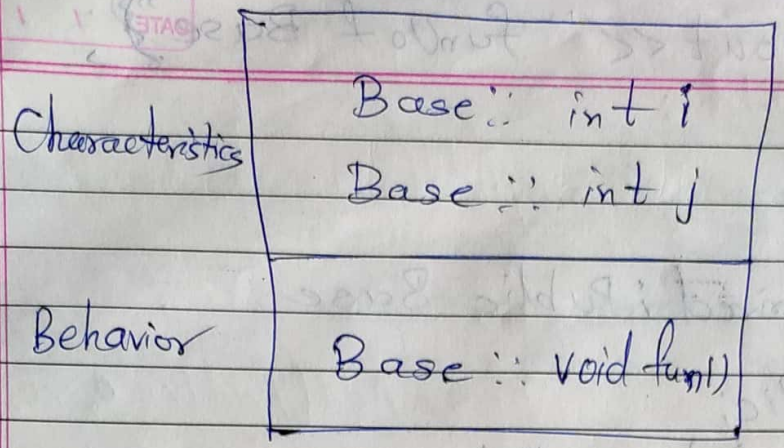
In this type of inheritance there are only 2 classess such as base class & derived class. All the characteristics & behaviors of base class are as it is available to the derived class.

```
class Base
{
    public:
        int i, j;
    Base ()
    {
        cout << "Base :: Constructor";
    }
    ~Base ()
    {
        cout << "Base :: Destructor";
    }
```

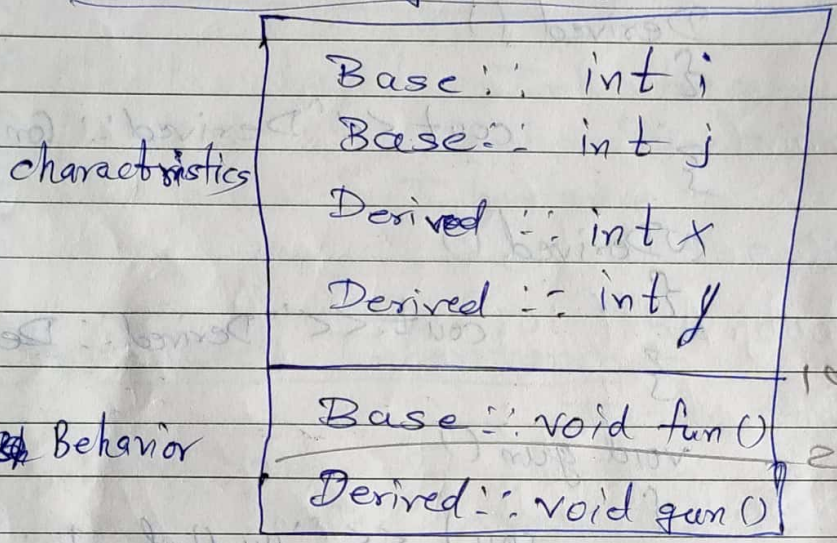
```
    void fun ()
    {
        cout << " fun() of Base ";
    }
};

class Derived : Public Base
{
    public:
        int x, y;

        Derived ()
        {
            cout << "Derived :: Constr";
        }
        ~Derived ()
        {
            cout << " Derived :: Destrctr";
        }
        void gun ()
        {
            cout << " gun() of derived";
        }
};
int main ()
{
    Base bobj;
    Derived dObj;

    cout << sizeof (dObj);      // 16
    cout << sizeof (bobj)       // 8

    return 0;
}
```

19/Oct/2015

* Class Diagram of Base Class

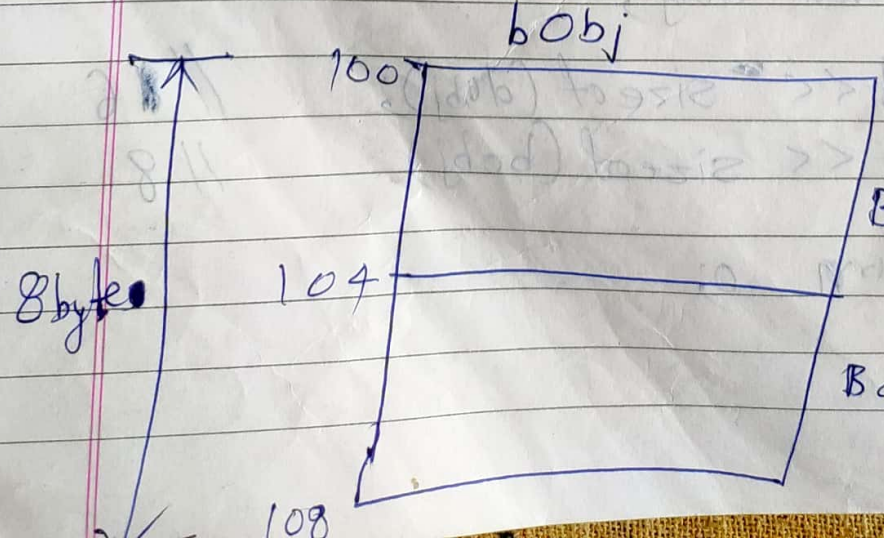


* Class Diagram of Derived class

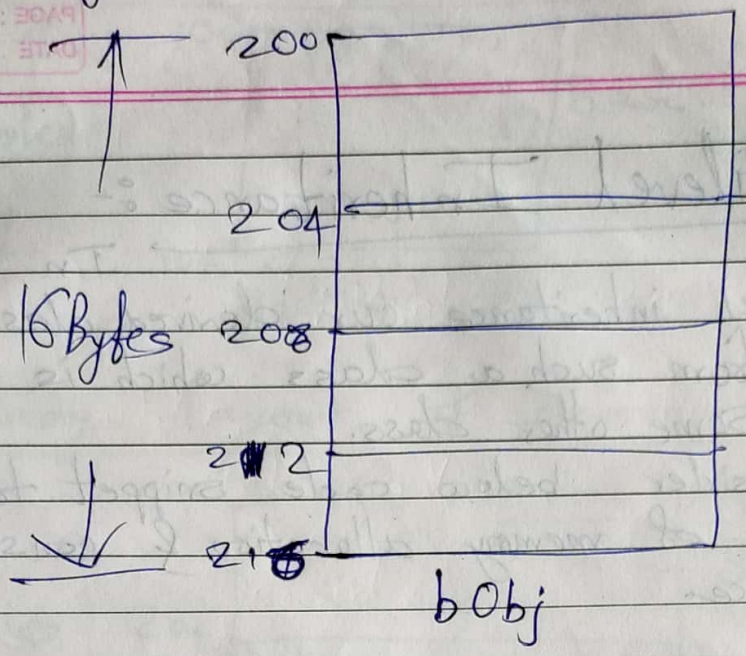


class diagram of class is divided into 2 parts first part contains characteristics & 2nd part contains behavior.

* Memory Layout of Base class :-



* Memory Layout of Derived class



* Constructor & Destructor Calling Sequence

When we create object of derived class in single level inheritance as,

```
Derived dObj;
```

Base :: Constructor
Derived :: Constructor
Derived :: Destructor
Base :: Destructor.

* According to the memory layout & constructor & destructor calling sequence

3.

19/oct/2015

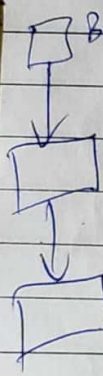
PAGE :
DATE . / /

[2] Multilevel Inheritance :-

In case of multilevel inheritance our derived class is derived from such a class which is already derived some other class.

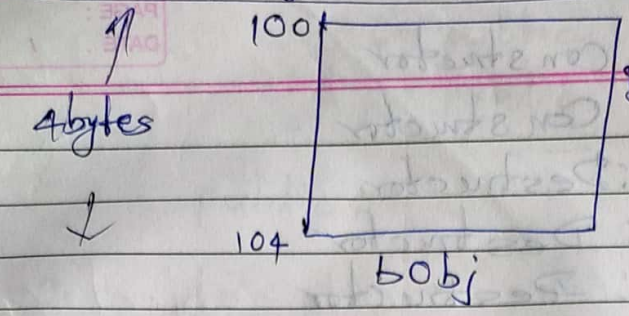
Consider below code snippet to understand concept of memory allocation & const^r calling sequence.

```
class Base
{
    public:
        int i;
};                    //Constructor & Destructor दिखेगा
```

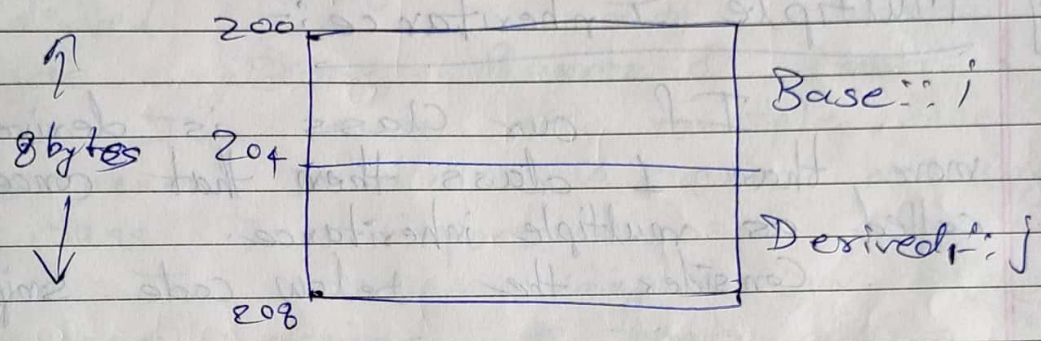



```
class Derived1 : Public Base
{
    public:
        int j;
};                    //const^r & Dest^r दिखेगा
class Derived2 : Public Derived1
{
    public:
        int k;        //const^r & Dest^r दिखेगा.
};
Base b obj;           // 4 byte
Derived1 d obj1;      // 8 byte
Derived2 d obj2;      // 12 bytes
```

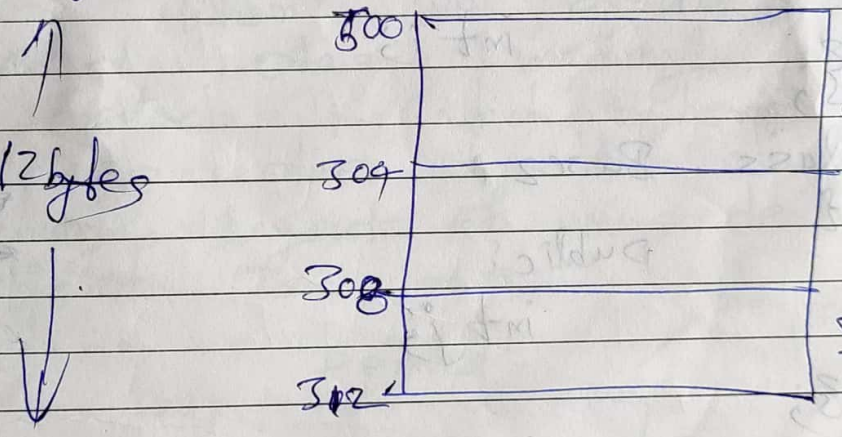
* Memory Layout of Base class :-



* Memory Layout of Derived1 :-



* Memory Layout of Derived2



* Constructor & Destructor Calling Sequence :-

When we create object of class Derived2 as,

Derived2 dobj2;

Sequence :-

Base :: constructor
Derived1 :: Constructor
Derived2 :: Constructor
Derived2 :: Destructor
Derived1 :: Destructor
Base :: Destructor

PAGE :
DATE : / /

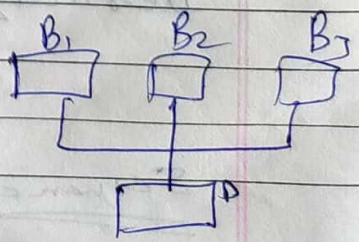
3 Multiple Inheritance :-

If our class is derived from more than 1 class then that concept is called as multiple inheritance.

Consider the below code snippet as,

```
class Base1
{
    public:
        int i;
};
class Base2
{
    public:
        int j;
};
class Base3
{
    public:
        int k;
};
```

/* Multiple class Constr & Destr ही होते. */



```
class Derived: public Base2,
               public Base1,
               public Base3
{
    public:
        int x;
};
```

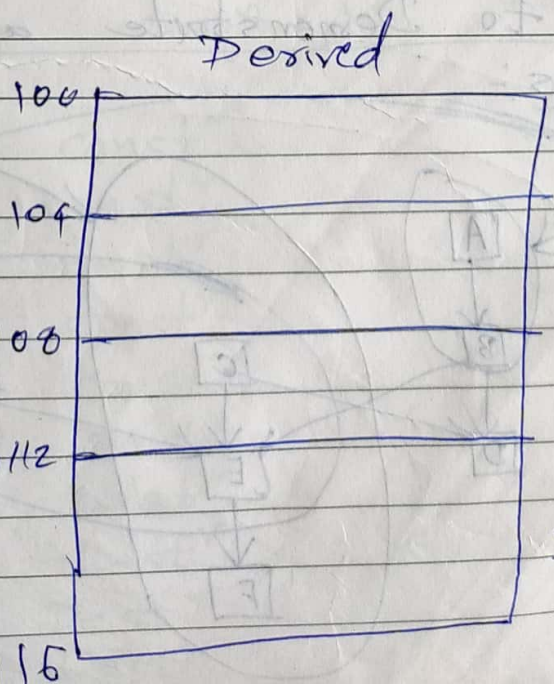
In our syntax Derived class is derived from 3 diff classes such as Base1, Base2 & Base3.

In case of multiple inheritance sequence in which we are deriving the class is important.

In our example our Derived class is derived from class Base2 then class Base1 then class Base3.

When we create object of Derived class memory gets allocated in order in which inherit that class due to this concept memory layout of our derived class object is as,

Derived dob1;



19/oct/2015

* Constructor & Destructor calling sequence in multiple inheritance is also depend on inheritance sequence.

If we create object of Derived class as,

```
Derived dObj;
```

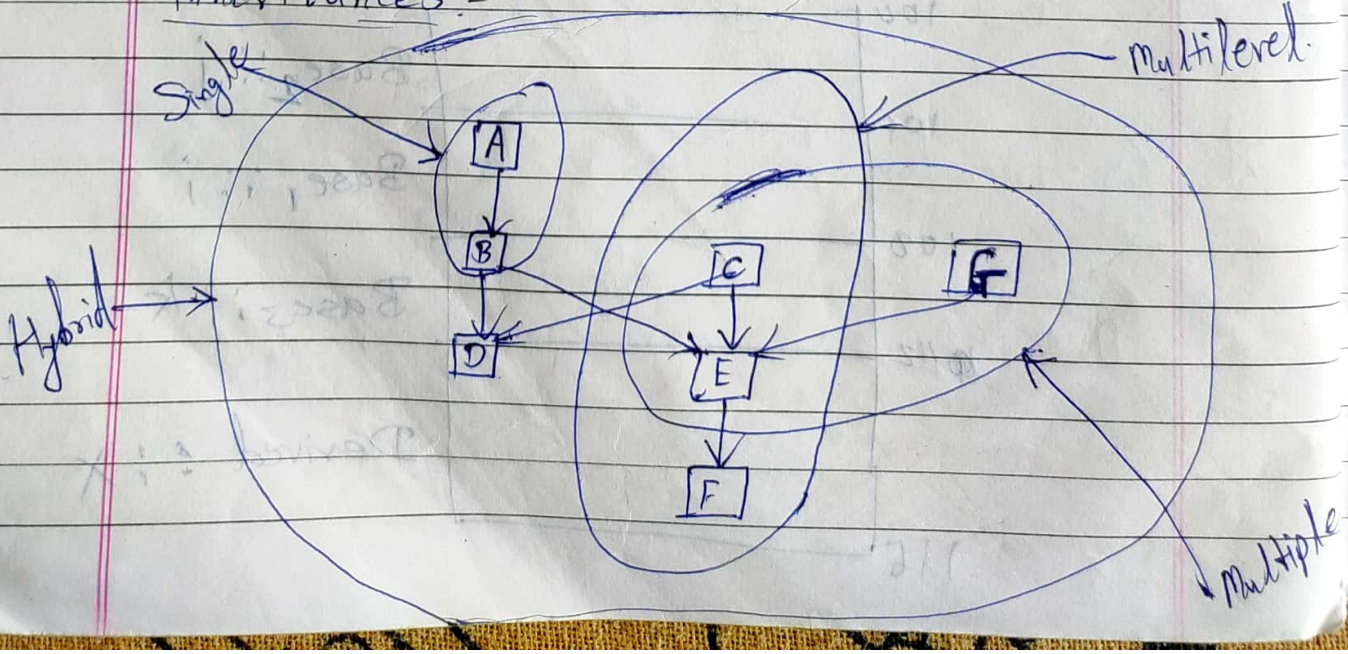
- then Constructor calling sequence is,

  Base2 :: Constructor
  Base1 :: Constructor
  Base3 :: Constructor
  Derived :: Constructor

- then Destructor calling sequence is

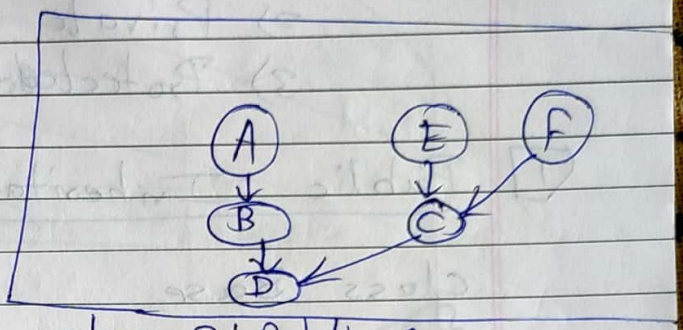
  Derived :: Destructor
  Base3 :: Destructor
  Base1 :: Destructor
  Base2 :: Destructor

* Diagram to Demonstrate all types of inheritances :-



★ Consider below syntax & Its Constr & Destr calling sequence:-

```
class A
{
};
class E
{
};
class F
{
};
class C : Public E,
          public F
{
};
class D : Public B,
          public C
{
};
```




```
class B : Public A
{
};
```

If we create of object of class D then constr & destr calling seqn is,

D dobj;

A constr
B constr
E constr
F constr
C constr
D constr
D Destr
C Destr
F Destr
E Destr
B Destr → A Destr

15/Oct/2015

Types of Inheritances According To Access Specifiers:

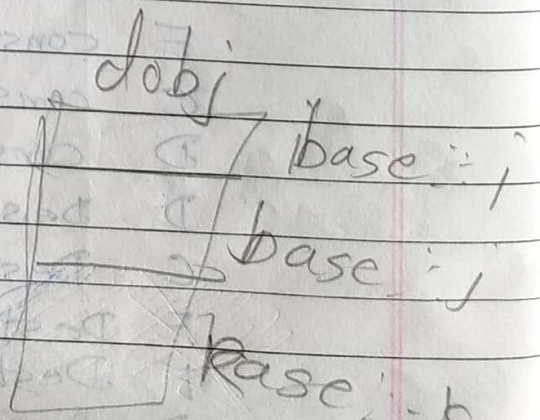
There are 3 types of inheritances —

1) Public Inheritance -
2) Private Inheritance -
3) Protected Inheritance -

1 Public Inheritance:

```
class Base
{
    public:
    Private:
    Protected:
};
class Derived : Public Base
{
  public:
  Private:
  Protected:
};
```

Derived dobj;



[2] Private Inheritance :-

```
class Base
{
    public:
    Private:
    protected:
};
class Derived : [Private] Base
{
    public:
    private:
    Protected:
};
```

[3] Protected Inheritance :

```
class Base
{
    Public:
    Private:
    Protected:
};
class Derived : [Protected] Base
{
    Public:
    Private:
    Protected:
};
```

* Contents of Base class are accessible to derived class is purely depend on the access specifiers which are used by a base class.
Means it is not depend on type of inheritance.

Become in Derived ~~it class then~~

Access specifier \ Inheritance	Public Inh. Type	Private Inh Type	Protected Inh. Type
Base → Public Access specifier	Public	Private	Protected
Base → Private Access specifier	Private	Private	Private
Base → Protected Access specifier	Protected	Private	Protected.

22/Oct/2015

* child class can access member of base class is not depend on type of inheritance used by the child class but it is depend on access specifier under which that member is defined in base class.

- child class can access the members of its grand parent class it is depend on 2 things (i) under which access specifier that member is coritten in grand parent class & (ii) which type of inheritance is used by the parent class.

- Size of object of any class is equal to summation of sizes of all its non-static data members

- Size of derived class = Summation of sizes of its all non-static characteristics
  + Summation of sizes of all non-static characteristics of base class.

- If the contents are written is base class under private access specifier then memory for that content is allocated in derived class, but we can't access that member.

22/Oct/2015

Member fn calling Mechanism in C++:-

PAGE :
DATE : / /

i) There are 2 types of fns in C++. As -

1) Member fn of class-
2) Nacked fn [defined outside the class]-

ii) If we want any member fn of class then we have to create object of that class.

Then by using that object we can call any member fn of class

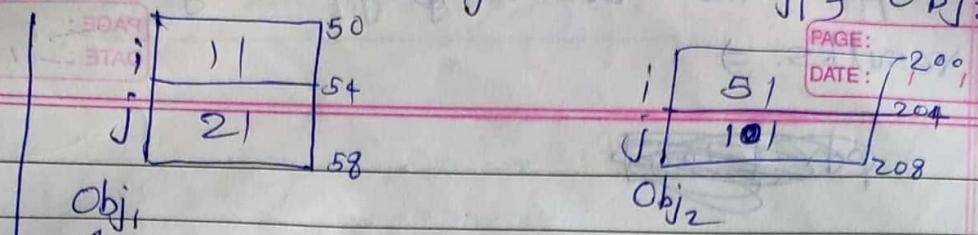
Consider below syntax which contains 2 characteristics as i, j & 1 member fn named as fun.

```
Class demo
{
    public:
        int i,j;
    demo()
    {
        i=11;
        j=21;
    }
    void fun()
    {
        gun();
        cout << i << j;
    }
};
int main()
{
    demo obj1, obj2;
    obj2.i = 51;
    obj2.j = 101;
```

```
private:
    gun()
    { i++; j--;
      cout << "gun
    }
```

22/08/2019

Consider memory layout of obj_1 & obj_2 as,




```
obj1.fun();   // 11 21
obj2.fun();   // 51 101
```

According to above syntax we can conclude that,

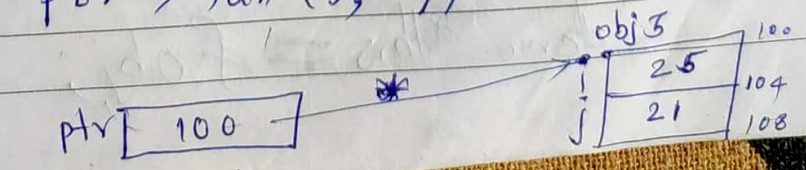
1) Any member fn can access member of its class by using its name.
Contents of that members are depend on the contents of caller object.

2) To call any member fn we have to create object of that class bcz to call any member fn object is required as a caller.

If we have object of class then we can use (.) dot operator to access its characteristics as well as behavior.

3) But if we have pointer which points to the object then we have to use -> arrow oprator to access any characteristics or behavior.

```
demo obj;
demo *ptr = NULL;
ptr = &obj3;
ptr->i = 25;
ptr->fun();   // 25 21
```



All the below syntax give the same results. ⇒

PAGE :
DATE : / /

```
ptr -> fun();      // 25 21
(&obj3) -> fun();  // 25 21

(*ptr).fun();      // 25 21
obj3.fun();        // 25 21
```

Fun(const demo *this)

void fun(int);

void demo(void *const krish, int A);

void fun(demo *krish, int);

*this;

krish A)

Fun(&obj, 50);

demo *this = &obj;

same

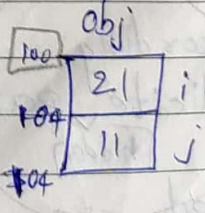
this Pointer in C++

22/Oct/2015

PAGE:
DATE: / /

- To call any member fn object of that class is required
- Use of . (Dot) op^{r} & object name is required to call any member f^{n}
- Every c++ program gets converted into its corresponding C prog then every f^{n} call gets converted according to 'C' syntax.
- Consider the below prog as-

```
class demo{

        public:
            int i,j;
            demo()
            {
                i=21; j= 11;
            }
            void fun (int x)
            {
                cout << i<<j <<x;
            }
        };
main()
{ demo.obj;

  obj. fun (51);
}
```



- According to above syntax obj is considered as a caller object.
- As our c++ prog. is converted into C program it is not allowed to write anythin

before fn call.

- To avoid this problem compiler converts this syntax into below syntax as,

obj.fun(51);
converted as
fun(&obj, 51);

- According to this syntax, when we call any member fn which is non static first hidden argument gets passed by address of caller object by the compiler ?
- If our member fn has 'N' arguments then after compilation it becomes 'N+1'.
- As no. of parameter going to be changed due to which prototype of member fn is also changed.
- Before compilation prototype of fn fun() is as,

  void fun(int x);
- After compilation that prototype becomes,

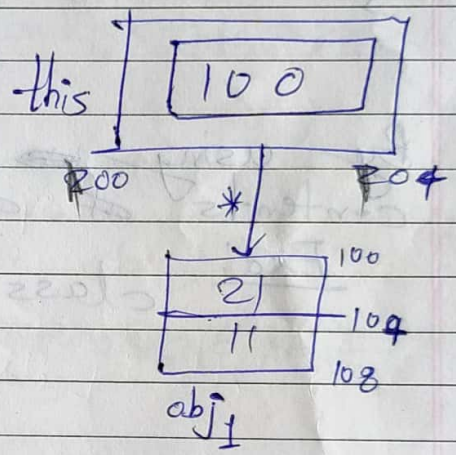
  void fun(demo * const this, int x);

- According above modified syntax first implicite [hidden] parameter gets added which is called as "this pointer"
- this is considered as constant pointer which holds address of caller object

- As it is a constant pointer we can't reinitialize that pointer by storing address of other object.

Exa:-
```
void fun(int x)
{
    demo obj1;        // Local object
    this = &obj1;     // Error
}
```

- According to the syntax of this pointer its memory layout becomes



- By using this pointer we can access characteristics as well as behavior of class

Exa:-
```
class demo{
    public: int i;
    demo()
    {   i=10;
    }
    void fun()
    {
        cout<<this->i;   // 10
        cout<<i;         // 10
        cout<<(*this).i; // 10
```

```
    this -> gun();    // Gun fn
    (*this).gun();    // Gun fn
}
void gun()
{
    cout << "In Gun fn";
}
};
int main()
{
    demo obj;
    obj.fun();        // 10 10 10
    return 0;         Gun fn
}                     Gun fn
```

- By using this pointer we can modify contents of caller object also.

Exa:-

```
class demo{
    public: int i;
    demo()
    {
        i = 10;
    }
    void fun()
    {
        this->i = 21;
    }
};
int main()
{
    demo obj;
    cout << obj.i;    // 10
    obj.fun();
    cout << obj.i;    // 21
    return 0;
}
```

⇒

- Generally this pointer is used if the name of characteristics of class is same as name of input argument of that fn.

NOTES

Example:-

```
class demo{
    public:
        int i,j;
    demo():
    {
        i=11; j=21;
    }
    void fun (int i, int j)
    {
        cout<<i<<j; // 51 121
        cout<<this->i<<this->j;
    }                // 11 21
};
int main()
{
    demo obj;
    obj.fun(51,121);   51 121
    return 0;          11 21
}
```

- We can use this pointer only inside any member fn of class.
  We can't use this pointer from nested fn.

Exa:-

```
class demo{
    public:
        void fun()
        { cout << "\nFun fn Call";
        }
        void gun():
        { cout << "\nGun fn call";
        }
        demo()
        {
            cout << "\nConstr";
            this -> fun();
        }
```

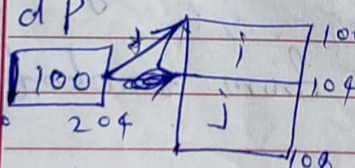
```
demo
{
    cout << "Destr";
    fun();
}
};
int main()
{
    demo obj;
    return 0;
}
```

O/P
Constr
fun fn call
Destr
Fun fn call

- According to above point we can use this pointer from constr to destr due to which we can conclude that, concept of this pointer is created from constr.
- By using this pointer we can deallocate memory of caller object from the member fn.

Example:

```
class demo{
    public: int i, j;
    demo()
    { i=11, j=21;
    }
    void fun()
    {
        delete this;
    }
};
int main()
{
    demo *dp = new demo();
    dp->fun(); return 0;
}
```



dp->fun();
↓
fun(dp);

- According to this syntax after calling fun fn memory of caller object gets freed by using delete operator.

Obj Dumb
Core Dumb

- To delete that memory of caller object from member fn it is necessary that memory should be allocated dynamically by using new opr.
- If we create 'N' diff. objects in our program then for every object this pointer gets created separately.
- Every this pointer holds address of its corresponding object.

demo dobj;

demo * dptr = &dobj;