22/Oct/2015

PAGE:
DATE: / /

★ Difference between Inline function & function like macro in C++ :-

- 1) Definition gets replaced at the place of call in case of inline fn as well as fn like macro.
- 2) Consider the below program which explains the concept of fn like macro's as,

```
# define MULT (A, B) A*B

int main()
{
    int Ans;
    Ans = MULT(5,3);
    Ans = MULT(3,5);
    Ans = MULT(2,1);
    return 0;
}
```

- In this program there is an fn like macro which accepts 2 arguments & it performs multiplication on that arguments. After performing preprocessing our main() fn becomes,

```
int main()
{
    int Ans;
    Ans = 5*3;
    Ans = 3*5;
    Ans = 2*1;
    return 0;
}
```

Scanned by CamScanner

- According to this converted main() fn we can conclude that def^n gets replaced at every place wherever we call fn like macro.

- There are some differences bet^n inline fn & Macros as,

inline fn	Function like Macro
1) Concept of inline is used in c++ only.	1) This concept can be used in C as well as c++.
2) It is handled by compiler.	2) It is handled by preprocessor.
3) Inline is considered as request.	3) It is not considered as request.
4) We can write large no. of lines in inline fn	4) We can't write large no. of code in Macro.

★ Some IMP points about inline

1) We can't check whether our fn is considered as inline fn or not by the compiler.
2) If our fn is considered as inline fn we can not apply fn pointer to it

Scanned by CamScanner

PAGE:
DATE: / /

2/Oct/2015

☆ Constant in C++ :-

- 1) Concept of constant in 'C' is similar as concept of constant in 'C++'.
- 2) Constant is considered as datatype qualifier in 'C' & 'C++'
- 3) In 'C' there is only one concept of constant i.e. constant variable. but, in 'C++' there are multiple concepts which are considered under constant concept. such as,
  - (A) Constant variable of program.
  - (B) Constant characteristics of class.
  - (C) Constant behavior of class.
  - (D) Constant object of class.

✱(A) Constant Variable of Program :-

Constant variable of C++ is exactly same as constant variable in 'C'.
In C++ if we are declaring the constant variable it should be initialized at the time of declaration itself otherwise it generates compile time error.

```
const int i;    // error
const int i=11; // allowed ✓
```

Scanned by CamScanner

If our variable is considered as constant variable then we can't change value of that variable throughout, exep of our program.

```
const int i=21;
i=10;    // error
i++;     // error
```

* [B] Constant Characteristics of class:-

- According to data qualifier there are 2 types of characteristics such as,
  1) Non-constant characteristics-
  2) Constant characteristics-

- If characteristics of class is constant then we can't change contents of that characteristics but in case of non-constant characteristics we can change its contents

- Consider the below syntax as ~~class demo~~

```
class demo{

    public:
        int i;          // Non-Const characteristics
        const int j;    // Constant characteristics
    };
```

- According to the rule of c++ we have to ~~initize~~ initidize the constant at the ~~to~~ fine of declaration.
  Due to which we have initialize that constant member properly.

Scanned by CamScanner

- There are multiple scenario's in which we can Initialize const characteristics, such as, -

DATE: / /

- (i)
```
class demo
{   public:
        const int j=11; //error
};
```

- (i) This syntax generates compile time error bcz at that point memory is not allocated to our characteristics & we can't initialize the characteristics inside the class.
- (ii) If we can't initialise the constant inside the const^{r} bcz, control comes inside the const^{r} after allocating the memory
- (ii)
```
class demo
{
    public: int i;
            const int j;
            demo()
            {
                j=11; //error
            }
};
```

* (iii) member initialization list is the proper sol^{n} to initialize constant characteristics of class.

```
class demo
{   public:
        const int j;
        demo():j(11)
        {
        }
```

Scanned by CamScanner

- (iii) This is the only one way in which we can initialize constant characteristics of class.

- This concept is allowed to initialize the constant members, bcz, control comes in member initilization list immidiately at the time of memory allocation.

Member Initialization List in C++ :-

Concept of member initialization list in C++ is used in diff scenarios such as,

1. To initialize constant characteristics of class.
2. To initialise reference variable of class.
3. If base class contains only parameterized constr & if we want to create object of derived class then compiler generates error due to the default constr.

   And we can avoid that problem by using member initialization list.

1. To initialize const characteristics of class :-

- In below program i is considered as constant characteristics & we can initialize it by using member initialization list.

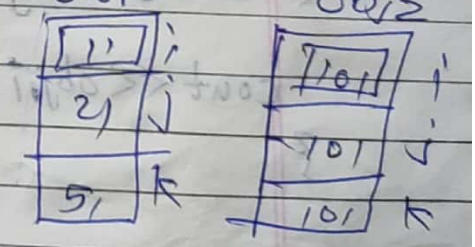
Scanned by CamScanner

22/oct/2015

```
class demo
{
    public;
        const int i;
        demo(-) : i(11)
        {
        }
};
```

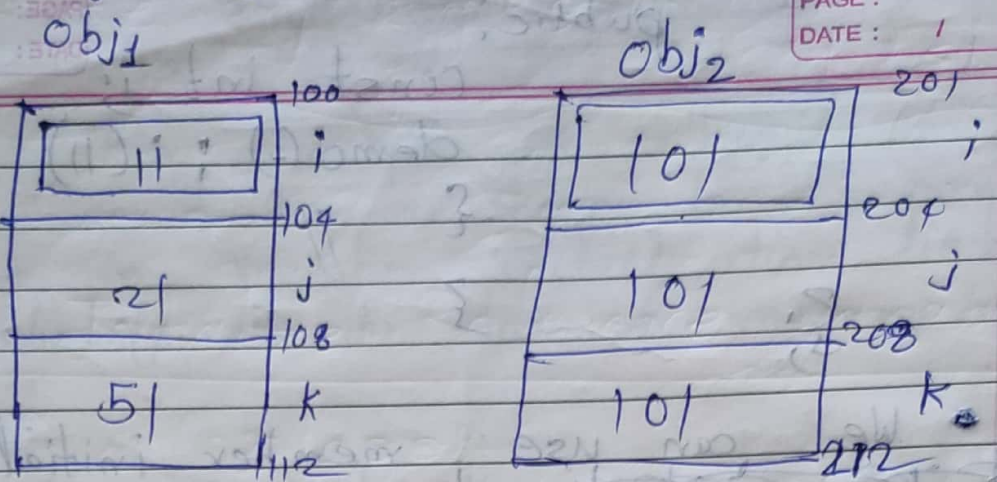
- We can use member initialization list to initialise constant as well as non constant data member also.

```
class demo
{
    public:
        const int i;
        int j, k;
    demo(): i(11), j(21), k(51)
    {
    }
    demo(int x): i(x), j(x), k(x)
    {
    }
};
int main()
{
    demo obj1;
    demo obj2(101);
    cout << obj1.i << obj1.j << obj1.k;
                                    // 11 21 51
    cout << obj2.i << obj2.j << obj2.k;
    return 0;
}                                   // 101 101 101
```



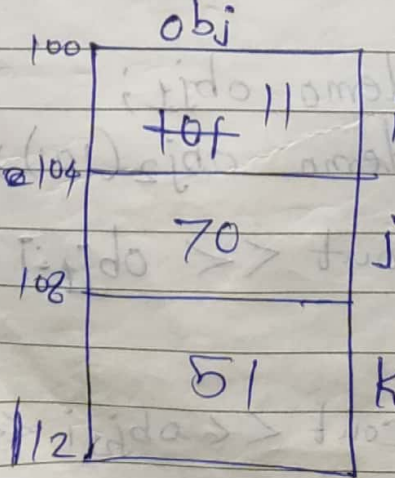
Scanned by CamScanner

Memory layout of obj_1 & obj_2 As,



- Consider the below example as,

```
class demo
{
    public :
        int i, j, k;
        demo(int a, int b): i(a), j(b)
        {
            k = 51;
            i = 11;
        }
};
int main()
{ demo obj(101, 70); // object creation in main() Cnstr call.
  cout << obj.i << obj.j << obj.k; // o/p  11 70 51
}
```



Scanned by CamScanner

- According to above syntax we can observe that we can initialize partial characteristics of class as well as we can re-initialize that member inside constructor also.

- As member initialization list get invoked first then constructor get invoked due to which the value which gets initialized inside constr becomes the final value.

Imp

- Member initialization list get invoked in order in which characteristics are declared in class.

Exa:-

```
class demo
{
    public:
        int i,j,k;
    demo(int x): j(++x), i(x), k(++x)
    {
    }
};
int main()
{
    demo obj(10);
    cout << obj.i << obj.j << obj.k; // 10 11 12
    return 0;
}
```

// initialization sequence

// In this constr our i variable gets initialized 1st. then j variable gets initialized & then k variable gets initialized.

Scanned by CamScanner

- According to above example we can use any expression in initialization list.

2 To Initialize Reference variable of a class:-

Member Initialize list use

- Like constant variable our reference variable should be initialised at the time of declaration itself.

But we can not initialise our reference variable inside the class according to c++ rule.

To avoid that problem in case of constant & reference member initialization list is the only one solution.

Exa:-① //const

```
const int i;  } //error
X  i=11;
```

✓ `const int i=11;` // allowed

Exa-② //Ref

```
int &ref;
X  ref=x;  } //error
```

✓ `int &ref=x;` // allowed

Scanned by CamScanner

- consider the below class which contains one member as a reference which gets initialised by using member initialization list as:-

```
class demo
{
    public:
        int i;
        int &ref;

        demo(int x) : ref (x)
        {
            ref++;
            i = 11;
        }
};
int main()
{
    int a=101;  //demo dobj(100);
    demo obj(a);  // internal working → int &ref = a;
    cout<< obj.i << obj.ref;  // 11 102
    cout<< a;  //101
    return 0;
}
```

Try

In this syntax our reference variable ref referred to variable 'a' which is present in our main function.

In this example if we change value of 'a' that change is not gets reflected in variable ref.

Scanned by CamScanner

• 3 member initialization List Uses

To avoid drawback of default constructor of base class :-

Consider below syntax as,

```
class base
{
    public:
        int i, j;
        base(int a, int b)
        {
            i = a; j = b;
        }
};
class derived : public base
{
    public:
        int x;
        derived();
        {
            x = 11;
        }
};
int main()
{
    derived dobj;
}
```

// problem or error case

// Correction by member initialization List.

Error code Problematic case

As we are going to create object of derived class which is derived from base class at this point compiler tries to invoke default constructor of base class & as there is no default constructor in base class it generates

compile time error.

To avoid that problem there are two solutions as,

① provide the default constructor in base class.

This soln is not useful if that class is designed some outsourced company.

② We can use member initialization technique at the time of writting constr of derived class.

By applying the concept of initialization list our modified constructor of derived class becomes-

```
derived ( ) : base (51, 101)
{
    x = 11;
}
```

Scanned by CamScanner

C Constant behavior of class:

There are 2 types of behavior in c++ →

1) Non Constant behavior.
2) Constant behavior.

① In case of nonconstant behavior we can modify the contents of caller object of that class. inside f^n.

② But if our behavior is constant behavior then we can't modify the contents of its caller object.

If we want avoid modification in our object inside the f^n then that f^n should be defined as constant member f^n.

Example:

```
class demo
{
    public:
        int i, j;
    demo()
    {
        i=11; j=21;
    }
    void fun(int x)   // Non Constant fn
    {
        int y=11;
        x++;
        y++;
        i++; j++;
    }
```

// allowed

Scanned by CamScanner

2/Oct/20

```
void gun (int x) const      // constant f^n
{
    int y = 11;
    x++;                    // allowed
    y++;
    // i++;                 // error
    // j++;
}
};
```

```
int main()
{ demo obj;
  obj.fun();
} obj.gun();
```

In above class fun() is considered as non constant member f^n & gun() is considered as constant member f^n.

In this gun() f^n we can't modify contents of its caller object due to which the statements i++ & j++ generates error but x & y are input argument & local variable due to which we can modify it.

* D Constant Object of class:-

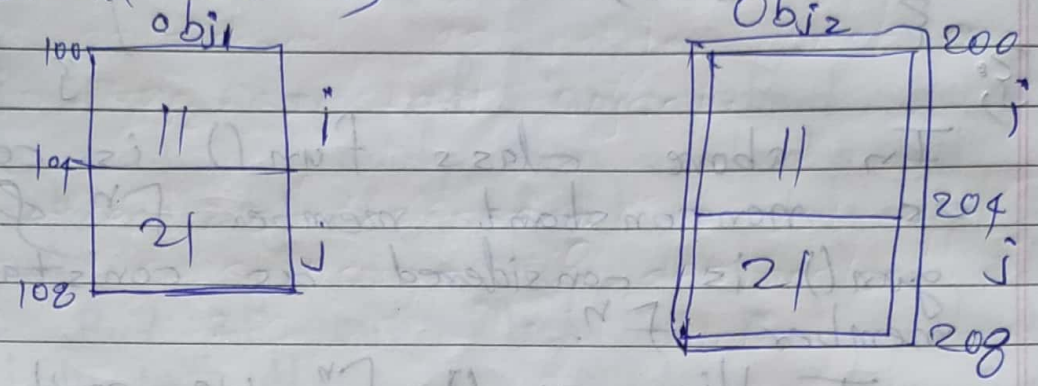
- If we create constant object of class then all the characteristics of that object becomes constant.
- Consider the class written in above point

```
int main()
{
    demo obj1;
    const demo obj2;
    return 0;
}
```

Scanned by CamScanner

```
int main()
{
    demo obj1;
    const demo obj2;
}
```

Memory Layouts of obj_1 & obj_2 should be as,




```
obj1.i++;
obj1.j++;
obj1.fun(11);
obj1.gun(21);
```
// Allowed

```
obj2.i++;   // Not allowed
obj2.j++;   // Not allowed.
obj2.fun(11); // Not allowed.
obj1.gun(21); // Allowed
```

Note: * Non constant object can call const member fn aswell as non constant member fn.

Note: * Constant object can call only constant member fn but that object can't ca

Rec | Imp | 10-5 Booting etc | Power of ptr

23/Oct/2015

PAGE:
DATE: / /

☆ Important Points About Constants:-

[1] If a class contains Atleast single characteristics as constant characteristics then that class should contain atleast default constr in it otherwise compiler generates an error.

[2] If class contains atleast one reference member then that class should contain parameterized constr with member initialization list.

#imp ☆→ [3] If our class contains constant behavior in it then internal layout of this pointer gets changed as,

```
class demo
{
    public:
        int i, j;
        void fun(int x)
        {

        }
        void gun(int y) const
        {
        }
};
```

After compilation fn prototype ~~becomes~~ gets modified as,

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

Non constant member fn

Scanned by CamScanner

void gun(const demo * const this, int y);

According to prototype of gun() f^n we can conclude that for the constant member f^n this pointer is considered as pointer which points to the constant object.

(IMP) * Concept of constant is not applicable for constructors & destructor.

(VIMP) ★ Consider the below example which contains all the concepts of constants in one class,

```
class demo
{
    public:
        const int i;        // const characteristics
        int j;              // NonConst. " "

        demo():i(10)        // constr
        {                   // member initialization list
            j=20;
        }

        void fun(int x)     // Non Constant behavior
        {                   // Error Always
            x++; i++; j++;  // Repeats on call
        }                   // Allowed always
        void gun(const int x, int y)   // Non constant behavior
        {
            x++;            // Error Always
            y++;            // Allowed Always
            i++; j++;       // depends on object call
        }
```

Scanned by CamScanner

// Constant behavior ⇒ void sun(const int x, demo *p) const

{

x++; // Not allowed

(p->i)++; // Always Error due to 'i' constant

(p->j)++; // Allowed Always due to pointer 'p'

(this->i)++; // Always Error due to 'i' constant

(this->j)++; // Always Error due to fn is constant

i++; } // Always Error

j++;

}

// Constant behavior ⇒ void run(const demo *p) const

{

i++; // Always Error due to 'i' is constant

j++; // Always Error due to this fn is constant

(p->i)++; // Always Error due to 'i' constant

(p->j)++; // Error due to constant pointer 'p'

(this->i)++; // Error

(this->j)++; // Error

}

};

demo dobj;

p = &dobj; // Allowed

int main()

{

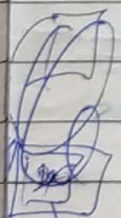
demo obj1; // Non Constant object

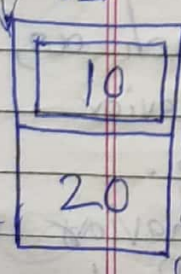
const demo obj2; // Constant object

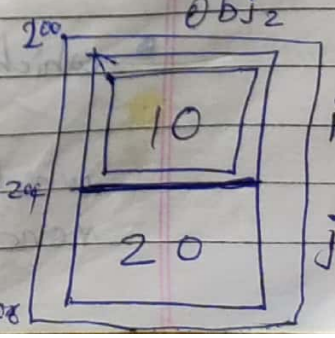
obj1.i++; obj1.j++;

obj2.i++; obj2.j++;

// Error Const (i) // Allowed // Error Const obj







Scanned by CamScanner

```
obj1.fun(11);
obj1.gun(11,21);
obj1.sun(11, &obj1);
obj1.run(&obj1);

obj2.fun(21);  //
obj2.gun(51,11); //   } Not Allowed
obj2.sun(101, &obj1);
obj2.sun(91, &obj2);
obj2.run(&obj1);
obj2.run(&obj2);
return 0;
}
```

Program Description:-

1) This class contains 2 characteristics as 'i' which is constant & 'j' which is nonconstant.

2) It contains 4 behaviors such as,
- fun() is nonconstant behavior which accept nonconstant value.
- gun() is nonconstant behavior which accept 1 constant & 1 nonconstant value.

Scanned by CamScanner

- sun() is constant behavior which accept 2 arguments 1st is constant integer & 2nd is address of demo class object.
- run() is constant behavior which accept one argument i.e. address of constant object of demo class.

3) Inside main fn 2 objects are created as obj_1 is non constant object & obj_2 as constant object.

fn calls

① obj_1 . fun (11)
// वही same in class
j++; // allowed

② obj_1 . gun (11, 21);
// वही same in class
j++; // allowed

③ obj_1 . sun (11, & obj_1);
// same in class defn

④ obj_1 . run (& obj_1);
// same as in class

⑤ obj_2 . fun(21); // fn call ला Error → bcz it is const. obj calling to non const behavior

⑥ obj_2 . gun (51,11); // fn call ला Error

⑦ obj_2 . sun(101, & obj_1); // same in class defn
(P->j)++; // Allowed

Scanned by CamScanner

(8) $obj_2.sum(\&obj_1, \&obj_2);$ // same as in class

~~$(p \rightarrow i)++;$~~ $(p \rightarrow j)++;$ // Allowed

(9) $obj_2.run(\&obj_1);$

// Same As in class def^n Explain^n

(10) $obj_2.run(\&obj_2);$

// Same As in class def^n Explain^n

Scanned by CamScanner

☆ Static In C++ ☆

PAGE:
DATE: / /

23/04/2015

- In C prog. language static is considered as storage class.
- As c++ is influenced from 'C' static is also considered as storage class.
- In c static concept is applicable for local static & global static variable.
  In c++ also we can use that same concept as local static & global static.
- As c++ object oriented programming lang. due to which it contains some extra points as,

1) characteristics.
2) behavior.
3) object.

- Due to the above point static concept used in c++ as,

* 1) Local static & global static variable
* 2) static characteristics of class.
* 3) static behavior of class
* 4) static object of class

Scanned by CamScanner

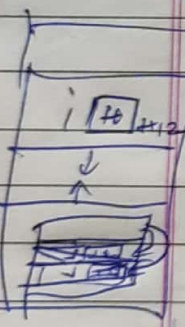
11 Local static & Global static Variables

PAGE:
DATE: / /

- In this case static is considered as storage class.
- If local variable is declared as a static then value of that variable gets preserved across the function calls.

Example:-

```
void fun()
{
    static int i=10; int j=10;
    cout << i;
    cout << j;
    i++;
    j++;
}
int main()
{
    fun();  // 10  10
    fun();  // 11  10
    fun();  // 12  10
    fun();  // 13  10
    return 0;
}
```



- If we declare variable as a global static variable then we can't access that variable by using extern keyword in other file.

According to the above explaination this concept of static is exactly similar as concept of static in 'c'.

Scanned by CamScanner

* [2] Static Characteristics of Class:-

- class contains 2 types of variables i.e. characteristics i.e.

1] Non-static variable:-

This type of variables are called as "Instance variables" bcoz every instance of class i.e. object has separate memory for non-static variables.

2] Static variables:-

This types of variables are called as "class variables" bcz, memory for that variables gets allocated only once per class. means that memory gets shared betn all the objects of same class.

- Size of our object is equal to summation of sizes of it's non-static characteristics.

- According to above point we can conclude that memory for static variables is not allocated inside the objects memory layout.

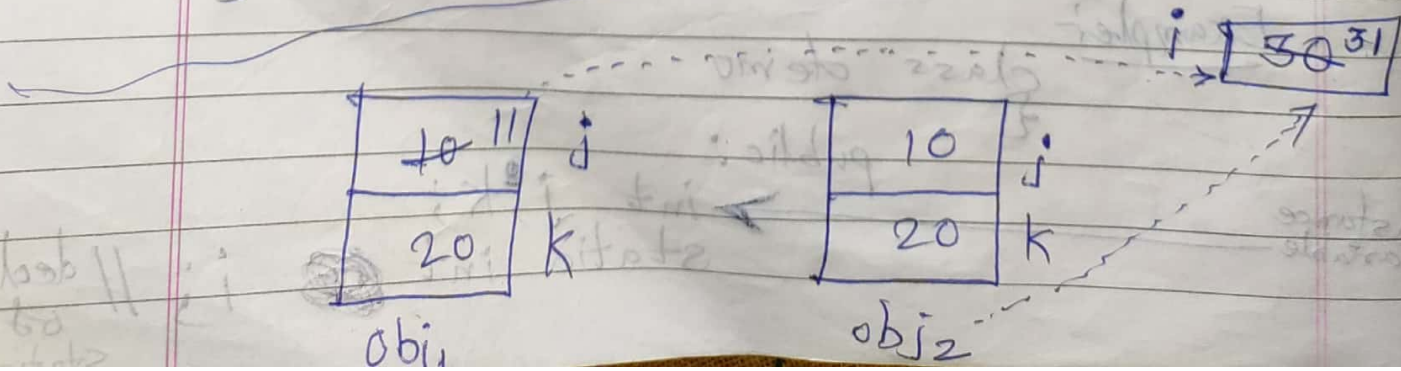
Example:-

```
class demo
{
public:
    int j,k;
    static int i; // decl" of static
```

Instance Variable

Scanned by CamScanner

```
demo ()
{
    j = 10;
    k = 20;
};
int demo::i = 30;   // definition of static variable

int main()
{
    demo obj1, obj2;
    cout << demo::i;          // 30
    cout << obj1.i;           // 30
    cout << obj2.i;           // 30
    cout << obj1.j << obj1.k;   // 10 20
    cout << obj2.j << obj2.k;   // 10 20

    obj1.i++;

    cout << obj1.i;           // 31
    cout << obj2.i;           // 31
    cout << demo::i;          // 31

    obj1.j++;
    cout << obj1.j << obj2.j;   // 11 10
    cout << sizeof(obj1) << sizeof(obj2);   // 8 8
    return 0;
}
```



Scanned by CamScanner

- In this class there are 2 types of characteristics as, static & non static

PAGE:
DATE: / /

- 'i' is considered as static characteristics 'j' & 'k' are considered as non-static characteristics.
- If class contains static variables in it, then that variable should be defined outside the class other wise compiler will generates an error.
- At the time of definition of static variable we can initialize it with some value.
- We can also initialize static variable inside const^{r}.
- We can access static variables of class by using class name as demo:: i
- We can also static variable by using object name as,
  $obj_1 . i$ or $obj_2 . i$
- Memory for static variables gets allocated when our executable file gets loaded into the memory due to which it is called as load time variables.
- Memory layouts of obj_1 & obj_2

Scanned by CamScanner

24/07/2015

*3 Static behavior of Class:-

PAGE :
DATE : / /

- There are 2 types of behavior in a class as static behavior & nonstatic behavior
  * Static behaviors are called as class behaviors.
  * Whereas Non-static behaviors are called as instance behavior
- If we want to define fn as static fn then we have to write static keyword before fn prototype.
- As static behaviors are called as class behavior we can access or we can call that behavior without creating object of that class
- We can access static behavior by using classname.
- We can call static behavior Without creating object of that class (means)

Scanned by CamScanner

24/Oct/2015

PAGE:
DATE: / /

```
class demo
{
    public:
        int i;
        static int j;
        demo()
        {
            i=10;
        }
        void fun()                    // demo *const this
        {
            cout << i;                // Allowed
            cout << j;                // Allowed
        }
        static void gun()
        {
            cout << i;                // Error Not Allowed
            cout << j;                // Allowed.
        }
};
int demo :: j = 20;
int main()
{
    demo obj;
    cout << sizeof(obj);              // 4
    cout << obj.i;                    // 10
    cout << obj.j;                    // 20
    cout << demo::j;                  // 20

    obj.fun();                        // call as fun(&obj);   // allowed
    obj.gun();                        // call as demo::gun()  // Allowed
    demo::gun();                      // Allowed
    return 0;
}
```

Scanned by CamScanner

every object oriented paradigm has a separate meaning.

According to which static behaviors are used to check resource availability.

- = If resource is available then we have to ~~check~~ create object of that class.
- - After creating the object constr gets implicitely called which is used to allocate resources.
- - After creating object we can call any member fn which internally to allocate that resource.
- - At the end we have to deallocate memory of the object & before deallocation destructor gets implicitely called which internally deallocate ~~reso~~ the resources which are allocated in constructor.
- - This should be the normal flow of any object oriented program.
- - As we call the static fn without creating the object 1st we can check whether the expected resources are available or not inside static fn.
- - If that resources are not available then we can not create object of that class otherwise we can create the object & allocate the resources from the constr.

Scanned by CamScanner

22/Oct/2015

Consider the below code snippet which explains all the above concepts (static) in our class,

```
type def int BOOL;
#define TRUE 1
#define FALSE 0

class fileWrap
{
    public:
        //characteristics
        fileWrap()                    // Constr
        {
            // open file as Hello.txt
        }
        ~fileWrap()                   // destr
        {
            // close file Hello.txt.
        }
        static BOOL checkfile()       // static Behavior.
        {
            //If Hello.txt is available
                return TRUE;
            //Otherwise
                return FALSE;
        }
        void readfile()               // Non Static Behavior
        {
            //read the content of
                Hello.txt.
        }
};
```

Scanned by CamScanner

```
int main()
{
    BOOL bRet;
    bRet = fileWrap::checkfile();
    if(bRet == FALSE)
    {
        // Resource are Not Available
        // Resource Allocation failed
        return (-1);   // Termination Due to failure
    }
    fileWrap obj;
    obj.readfile();
    // Use the other behaviors if necessary if declared &
    return 0;   // succesful termination
}
```

✱ Description:

- In this prog. fileWrap class is used to perform file handling oprations
- checkfile() is static behavior which is used to check whether Hello.txt file is available or Not.
- As this fn is static fn we can call that fn without creating the object
- If that fn returns TRUE means resources are available then we can create object of the class otherwise we can return from main() fn.
- This is practical example used to demonstrate concept of static behavior.

Scanned by CamScanner

- If we call static behavior by using the object compiler search the name of class whose object is used in that f^n call & it replace object name with the classname.
- To call static behavior object is not required due to which there is no such concept called as this pointer in case of static behavior
- As static behavior can be called before creating the object it can only access static characteristics of class
- on the other hand non static behavior can access static characteristics as well as non static characteristics
- We can't apply concept of static for const's & dest'.

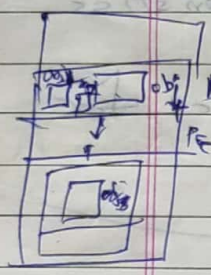
If base class contain static variable then memory for that static behavior is not inherited for the derived class but derived class can access static members of base class by using its name.

Scanned by CamScanner

(4) Static Object of class :- 24/04/2015

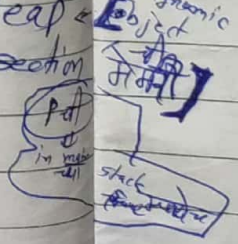
PAGE :
DATE : / /

- Static object is considered as normal object only difference between static object & non-static object is that the memory allocation for the static object is inside the static section or Data section.
- There is no logical difference beth static object & non-static object.
- Memory for the object gets allocated depend on the storage class of that object.



Example:

```
class demo
{
  // code
};
                                   Always BSS
demo obj1;          // Data Section
static demo obj2;   // static segment or Data section

int main()
{
  demo obj3;        // stack section (stack frame of main)
  demo *p = new demo();   // Heap section
  static demo obj4;       // static Segment or Data section
  return 0;
}
```



1) •
2) •
3) •
4) •
5) •
6) •

Scanned by CamScanner

24/Oct/2015

★ Polymorphism ★

in C++

1) • Polymorphism is one of the object oriented paradigm of c++.

2) • Polymorphism means single name multiple behaviors.

3) • Depend on biding of name with its actual behavior there are 2 types of polymorphism as,
   1) compile time polymorphism
   2) Run time polymorphism

4) • ① If which fn should be called is decided @ compile time then it is called as compile time polymorphism & that binding is called as early binding.

5) • ② If which fn ~~which~~ should be called is decided @ runtime then that polymorphism is called as runtime polymorphism & that binding is called as late binding.

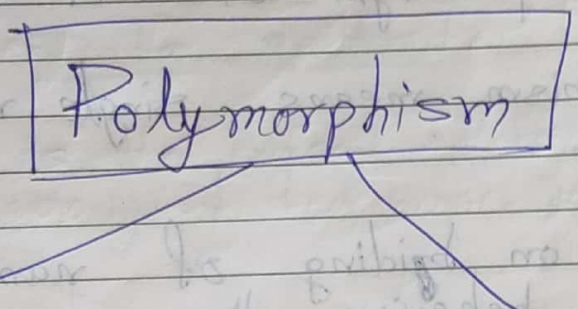
6) • Polymorphism is considered as single name & multiple behavior.

Scanned by CamScanner

7) • Compile time polymorphism is achieved by overloading.

PAGE :
DATE : / /

There are 2 types of overloading as,
1) fn overloading
2) Oprator overloading



IMP C++ paradigm which means single name multiple behavior.

Compile time

Runtime

Early Binding

Late Binding

Overloading

Virtual

fn overloading

Oprator Overloading

Virtual fn

Pure Virtual fn

obj

1000

2000

demo : fun

demo : fun

1) fn should be

2) every fn of overloading

3) concept of class.

4) the any

To Apply member

Function Overloading:-

PAGE:
DATE: / /

24/Oct/2015

1) f^n overloading is one of the type of compile time polymorphism.
Because which f^n should be called is decided by compiler at compile time.

2) In case of f^n overloading names of all the f^{ns} should be same.
To apply the logic of f^n overloading there should be at least 2 f^{ns} having same name.

3) In case of f^n overloading all the overloaded f^{ns} should be in same class, means we can't overload the f^n across the class.

4) We can overload the f^n by changing the no of arguments to f^n

Ex:-

```
class demo
{
    public:
        void fun(int i)          // 1000  Address of fun
        {
            cout<<"fun 1";
        }
        void fun(int i, int j)   // 2000  Address of fun
        {
            cout<<"fun 2";
        }
};
int main()
{
    demo obj;
    obj.fun(10);  // fun1
    // CALL 1000; // Assebly instn
```

Imp: Applying concept of overloading for every f^n should be a member of any class.

Scanned by CamScanner

```
obj.fun(10,20);  //fun 2
//CALL 2000      //Assembly Instruction
return 0;
}
```

5) • In the above class there are 2 fn having same name fun() but no. of argument for both the fns are different

6) • 1000 is the base address of 1st fn & 2000 is the base address of 2nd fn.

7) • When we call this fn from main() after compilation that call gets replaced with appropriate call instruction & its address.

8) • We can overload the fn by changing type of argument

Example:

```
class Demo
{
    public:
        void fun(int i)      // Base Address of fn 1000
        {
            cout<<"fun1\n";
        }
        void fun(double d)   // 2000
        {
            cout<<"fun2\n";
        }
        void fun(char c)     // 3000
        {
            cout<<"fun3\n";
        }
};
```

Implicit type Conversion

Scanned by CamScanner

```
int main()
{
    Demo obj;
    obj.fun(11);      // fun 1 // CALL 1000
    obj.fun(3.14);    // fun 2 // CALL 2000
    obj.fun('A');     // fun 3 // CALL 3000
}
```

PAGE:
DATE: / / Assembly Instrn

25/Oct/2015

9) ☆ While calling the function C++ compiler uses the concept of implicite type conversion if desired data type is not available in our i/p argument of the fn

Exa:-

```
class demo
{
    public:
        void fun(int i)          // 1000
        {
            //code
        }
        void fun(double d)       // 2000
        {
            //code
        }
};
int main()
{
    obj.fun(11);      // CALL 1000
    obj.fun('m');     // CALL 1000
    obj.fun(7.14f);   // CALL 2000
    obj.fun(7.12);    // CALL 2000
    return 0;
}
```

Base Address of fn

Assembly Instrn

Implicite Conversion

Scanned by CamScanner

25/Oct/2015

- In this example when we pass character as a argument it gets converted into integer & when we pass float it gets converted into doubly & this concept is called as Implicit type conversion by compiler

- If any datatype gets converted into other datatype then while converting there should not be any datal loss. Then implicit conversion is applicable.

10) - As like a implicit conversion we can also decide which fn should be called by using the concept of explicite type casting as.

Exa:-

```
class demo
{
public:
    void fun (int i)        // 1000
    {
    }
    void fun (char ch)      // 2000
    {
    }
};
int main()
{
    demo obj;
    obj.fun(11);
    obj.fun('m');
    obj.fun((int) 3.14f);
    obj.fun((int) 'm');
    return 0;
}
```

Base Address of fn

obj.fun(5)

Implicit conversion

11) • In this f^n call we are specifying datatype in circular bracket ~~which is called as explicit type conversion.~~

• Converting small datatype into large datatype is called "data winding"

• Converting large datatype into small datatype is called as "data narrowing"

• In case of data narrowing due to which it is not applicable for implicit type conversion & in that case compiler will generate warning or error.

---

12) • We can't ~~the~~ overload f^n by changing its Access Specifier Bcz at the time of f^n call access specifier is not considered.

Ex:-

```
class demo
{
  public:
        void fun (int i)
        {
        }
  private:
        void fun (int i)
        {
        }
};
```

// Not Allowed (Error)

// f^n call object चा
// Depend नसतो access specifier वर

कारण object call होताना माहित नसते access specifier

Scanned by CamScanner

We can overload the fn by using the concept of constant:

PAGE :
DATE : / /

Exa:-

```
class demo
{
    public:
        void fun(int i)
        {
            //pf("Inside fun");
        }
        void fun(int i) const
        {
            // pf("Inside fun2");
        }
};
int main()
{
    const demo obj1;
    demo obj2;
    obj1.fun(11);
    obj2.fun(21);
    return 0;
}
```

f" of Base Addr

//1000

//2000

Assembly instr^n

//CALL 2000

//CALL 1000

Scanned by CamScanner

14) • We can't overload the fn by changing datatype qualifier of input argument of fn bcz at the time of call compiler can't predict which fn should be called due to which it generates compile time error.

Exa:-

```
class demo
{
    public:
        void fun(int i)
        {
        }
        void fun(const int i)
        {
        }
};
```

// Not Allowed — Error @ Compile time

15) • We can overload the fn by applying the concept of datatype modifier for the i/p argument as signed & unsigned.

Exa:-

```
class demo
{
    public:
        void fun(signed int i) //1000
        {
        }
        void fun(unsigned int i) //2000
        {
        }
};
```

Allowed

Scanned by CamScanner

```
int main()
{   demo obj;                                   Assembly instrn
    obj.fun(-11);                   //CALL 1000
    obj.fun([redacted] 11);         //CALL 1000
    obj.fun((unsigned) 11);         //CALL 2000
    [redacted] unsigned int no=12;
    obj.fun(no);                    //CALL 2000
}
```

16) • We can overload the fn by changing datatype modifier of i/p argument.

Exa:
```
class demo
{   public:
        void fun(short int i)       //1000
        {
        }
        void fun(long int i)        //2000
        {
        }
        void fun(int i)             //3000
        {
        }
};
int main()
{   demo obj;
    obj.fun(11);                    // CALL 3000   (Assembly instns)
    obj.fun((short) 11);            //CALL 1000
    obj.fun((long) 11);             //CALL 2000
    return 0;
}
```

Scanned by CamScanner

17) • We can't use inline concept as overloading criteria.

Exa:-

```
class demo
{
    public:
        void fun(int i)
        {
        }
        inline void fun(int i)
        {
        }
};
// Not Allowed
```

---

18) • We can also overload class behaviors that is static member fns.

Exa:-

```
class demo
{
    public:
        static void fun(int i) //1000
        {
        }
        static void fun(int i, int j) //2000
        {
        }
};
int main()
{
    demo::fun(11);      //CALL 1000
    demo::Fun(11, 21);  //CALL 2000
    return 0;
}
```

Scanned by CamScanner

19) • We can overload the fn by changing sequence of argument.

PAGE :
DATE : / /

21)

Exa:

```
class demo
{
    public:
        void fun(int i, float f) //1000
        {
        }
        void fun(float f, int i) //2000
        {
        }
};
int main()
{
    demo obj;
    obj.fun(10, 5.14f);    // CALL 1000.
    obj.fun(5.14f, 10);    // CALL 2000
    return 0;
}
```

20) • We can not overload the fns on the basis of static.

Exa:

```
class demo
{
    static public:
        static void fun(int i)
        {
        }
        void fun(int i)
        {
        }
};
// Not allowed  Error
```

Scanned by CamScanner

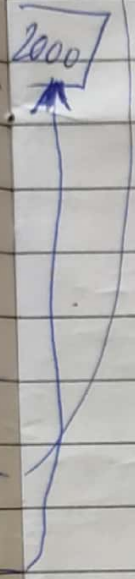
21) We can use concept of reference as an overloading criteria.

PAGE:
DATE: / /

But if we call the f^n by using the name of variable then it generates an biguity error Means there is no such scenario using which we can call the f^n which accept reference as argument.

Exa:

```
class demo
{
public:
    void fun(int i)          //1000
    {
    }
    Void fun(int &ref)       // 2000
    {
    }
};
int main()
{
    demo obj;
    obj.fun(11);             //Allowed
    int i = 10;
    obj.fun(i);              //Ambiguity Error.
    return 0;
}
```



Scanned by CamScanner

22) • We can't consider return value as overloading criteria

It is not applicable for overloading criteria.

Bcz, return value is not considered at the time of fn calling.

Due to which compiler is unable to bind the appropriate def fn of fn.

Exa:-

```
class demo
{
    public:
        int fun()
        {
        }
        char fun()
        {
        }
};
// Not Allowed [Error Redefn.]
```

Scanned by CamScanner

25) • We can use concept of pointer to overload the fn.

PAGE:
DATE: / /

2.5 @ 2015

Exa:

```
class demo
{
    public:
        void fun(int *p)        //1000
        {
            // code
        }
        void fun(char *p)       //2000
        {
            // code
        }
};
int main()
{
    demo obj;                   <-- int i=10; char ch='A';
    obj.fun(&i);    // CALL 1000
    obj.fun(&ch);   // CALL 2000
    obj.fun((int *) 7658);  // CALL 1000
    obj.fun((char *) 7568);  // CALL 2000
    return 0;
}
```

• We can also use generic pointer as the i/p argurgument. (void *)

Exa:

```
class demo
{
    public:
        void fun(int *p)        //1000
        {
        }
        void fun(char *ch)      //2000
        {
        }
```

Scanned by CamScanner

```
void fun( void *p)          // 3000
{
}
};

int main()
{
    demo obj;
    int i; char ch; float f;
    obj.fun(&i);            //CALL 1000
    obj.fun(&ch);           //CALL 2000
    obj.fun(&f);            //CALL 3000
    return 0;
}
```

Scanned by CamScanner

Name Mangling of Naming Decoration

25/Oct/2015

1) • Every c++ prog. gets converted into its approated c program

2) • But in c prog. language there is no such concept in which we can use the same name of fn.

3) • In C++ name of every fn gets changed by the compiler & that concept is called as Name Mangling or Naming Decoration.

4) • According to this point we can conclude that concept of name mangling is only applicable for user of that prog. but it is not applicable for internal use.

5) • Consider below programm As,

```
class demo
{
    public:
        void fun(int b);
        void fun(int b, char ch);
        void fun(int *p);
};
```

only ded^n आहेत

After compilation name of that fn becomes,

Scanned by CamScanner

fun(int b); ⇒ fun@1i()

fun(int b, char ch); ⇒ fun@2ic()

fun(int *ptr); ⇒ fun@1ip()

- While decorating the name compiler considers name of the fn, no. of arguments & type of arguments.
- Rules used for name manging may vary according to compiler.
- Programmatically we can't fetch mangled names.
  But there are some reverse engg. tools by using which we can get the mangled name of the fn.
  Tool ⇒
  1) cff explorer.
  2) WIN DGB.
- We can avoid name manging strategy by compiler by using extern "C" before fn prototype as,

Syntax:

```
extern "C" void fun()
{
}
```

// By using above syntax plz Compile this prog. using 'C' prog. strategy

This extern "C" keyword indicates tha
that fn should be compiled by using
"C" compiler strategies where there is
no concepts of name changing

30/Oct/20

☆ Runtime Polymorphism in C++

Scanned by CamScanner

★ Upcasting in C++:-

(30/Oct/20)

- To understand concept of upcasting there should be atleast single level inheritance in it.
- Consider there are 2 classes as Base class & derived class.
- If base class pointer can hold address of derived class then that concept is called as 'upcasting'.
- Derived class pointer holds the address of base class called as 'Down casting'.

Eg ★ Consider Below Classes to understand the concept of upcasting & runtime polymorphism.

```
class base
{
    public:
        int i, j;
        virtual void fun()      //1000
        {
            cout<<"fun of Base\n";
        }
        virtual void gun()      //2000
        {
            cout<<"Gun of Base\n";
        }
        void sun()              //3000
        {
            cout<<"Sun of Base\n";
        }
};
```

Add(int, int)

Scanned by CamScanner

30/08/2015

```
class derived : public base
{
public:
        int x, y;
        void fun()          // 4000
        {
            cout << "fun of Derived\n";
        }
        void sun()          // 5000
        {
            cout << "Sun of Derived";
        }
        virtual void run()  // 6000
        {
            cout << "run of Derived";
        }
};
int main()
```

Description:-

- In this prog. there are 2 classes which are related in single level inheritance.
- Base class contains 2 characteristics as i & j.
- Derived class contains its own 2 characteristics as x & y.
- Base class contains 2 fns which are defined as virtual [fun(), gun()] & one fn which is not virtual [sun()].
- Derived class contains 2 non virtual fns named as [fun() & sun()] & one virtual fn named as run().

Scanned by CamScanner

* Imp Defns in C++ :-

30/Oct/2015

PAGE :
DATE : / /

1) Redefinition :-

If same identifier name is used in the base class for characteristics or behavior & if that identifier is used in the derived class then it is called as "Redefinition".

For Redefition there should be single level inheritance

The can be 1) Characteristics Redefinition
or 2) Behavior Redefinition.

2) Overloading :-

If same identifier is used to define multiple behaviors in same class then that concept is called as "overloading"

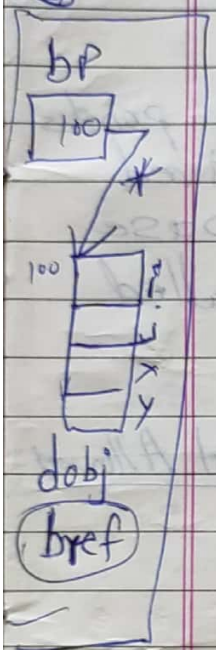
3) Overriding :-

If base class contains a behavior defined as virtual and same name & same prototype behavior is defined in the derived class then it is called as "overriding"

Scanned by CamScanner

30/Oct/2015

```
int main()
{
    base *bp = NULL;
    derived dobj;
    base &bref = dobj; } // Upcasting
    bp = &dobj;
    cout << "Function Calls by using Pointer\n";
    bp -> fun();          // Upcasting
    bp -> gun();
    bp -> sun();
    cout << "\nFunction Calls by using Reference\n";
    bref.fun();
    bref.gun();
    bref.sun();
    cout << sizeof(base) << "\n";
    cout << sizeof(derived) << endl;
    return 0;
}
```



O/P:-

```
Function Calls by Using Pointer
Fun of Derived
gun of Base
sun of Base
Function Calls by using Reference
Fun of Derived
Gun of Base
Sun of Base
12   <-- // Base ki size
20   <-- // Derived ki size.
```

Scanned by CamScanner

30/oct/18

Description:-

- In this prog. base class pointer holds address of derived class object of Base class reference is refering to derived class object. This concept is called as upcasting.

- If derived class pointer points to base class object or derived class reference refering to base class object then it is called as down casting.

- Concept of downcasting is Not Allowed

Ex:-

```
Base bobj;
derived dobj;

base *bp = NULL;
derived *dp = NULL;

bp = &dobj;   // Allowed (Upcasting)
bp = &bobj;   //Allowed
dp = &dobj;   //allowed
// dp = &bobj;   // Not Allowed (Downcasting)

base &bref1 = bobj;   //Allowed
base &bref2 = dobj;   //Allowed (Upcasting)
derived &dref1 = dobj;   //Allowed
derived &dref2 = bobj;   //Not Allowed
```

Scanned by CamScanner

Imp ⇒ * According to above Example downcasting is not allowed syntactically but, Upcasting is allowed by using pointer & reference.

* Need of Virtual Concept:

- 1) If our prog. contain upcasting by using pointer or reference, then at the time of accessing behavior. Behavior of pointer type gets involked automatically.
- 2) Logically behaviors of pointed type should be involked.
- 3) And to achieve this concept we have to use virtual

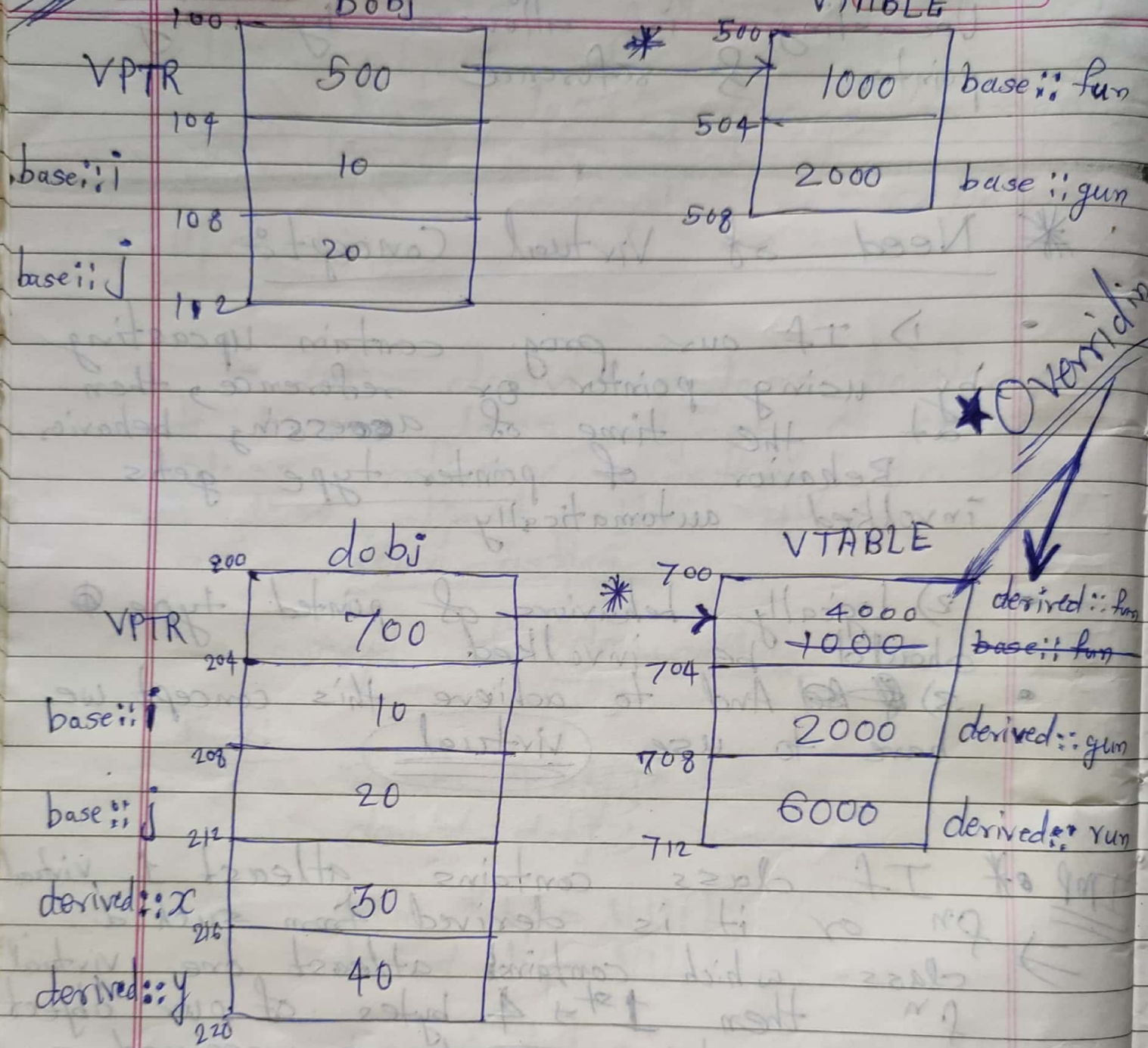
Imp ⇒ * If class contains atleast 1 virtual fn or it is derived from such a class which contains atleast one virtual fn then 1st 4 bytes of our object layout gets reserved as VPTR.

At the time of object creation one new data structure gets created by the compiler named as VTABLE.

VTABLE is considered as array of fn pointers which contains base address of every virtual fn of that class in order in which they are defined in class.

Scanned by CamScanner

30/Oct/2015



* Overriding

Scanned by CamScanner

1/Nov/2015

☆ Pure Virtual fⁿ in C++ :-

- This is one of the topic used in c++ to extend the concept of virtual in c++.
- Virtual fⁿ is such a fⁿ which is not accessible directly to the outside world but it is existing.
- If base class has to provide some behavior but that class known what to do but that class is unable to predict how to do.
- In such senarios that kind of behavior is considered as Pure virtual fⁿ in c++.
- Exa:

```
class base
{ public: int i, j;
// 1000→   void fun()                    // Concrete fⁿ
        {   cout << "fun of base\n";
        }
// 2000→   virtual void gun()            // virtual fⁿ
        {
            cout << "gun of base\n";
        }
        virtual void sun()=0;             // Pure Virtual fⁿ
};
class derived : public base
{
    public:
        int x, y;
// 3000→   void gun()                    // Overriding
        {   cout << "Gun of derived\n";
        }
```

Scanned by CamScanner

(4000) →

```
void sun()          // Def^n Not overriding
{
    cout << "sun of derived";
}
};
int main()
{
    base bobj;          // error
    derived dobj;       // Allowed
    base *bp=NULL;      // Allowed
    bp = &dobj;         // Upcasting
    bp->fun();          // fun of base
    bp->gun();          // gun of derived
    bp->sun();          // sun of derived
    cout << "size of (base)" << endl; // 12
    cout << "size of (derived)" << endl; // 20
    return 0;
}
```

1/Nov/2015

PAGE:
DATE: / /

Description:-

- In this syntax base class is considered as abstract class bcz, it contains pure virtual f^n in it. as sun().
- If class contains pure virtual f^n in it then compiler is unable to insert its address in VTABLE.
  Due to which that VTABLE is incomplete.
- As VTABLE is part of object our object is also incomplete.
  Due to which can not create object of such class which contains

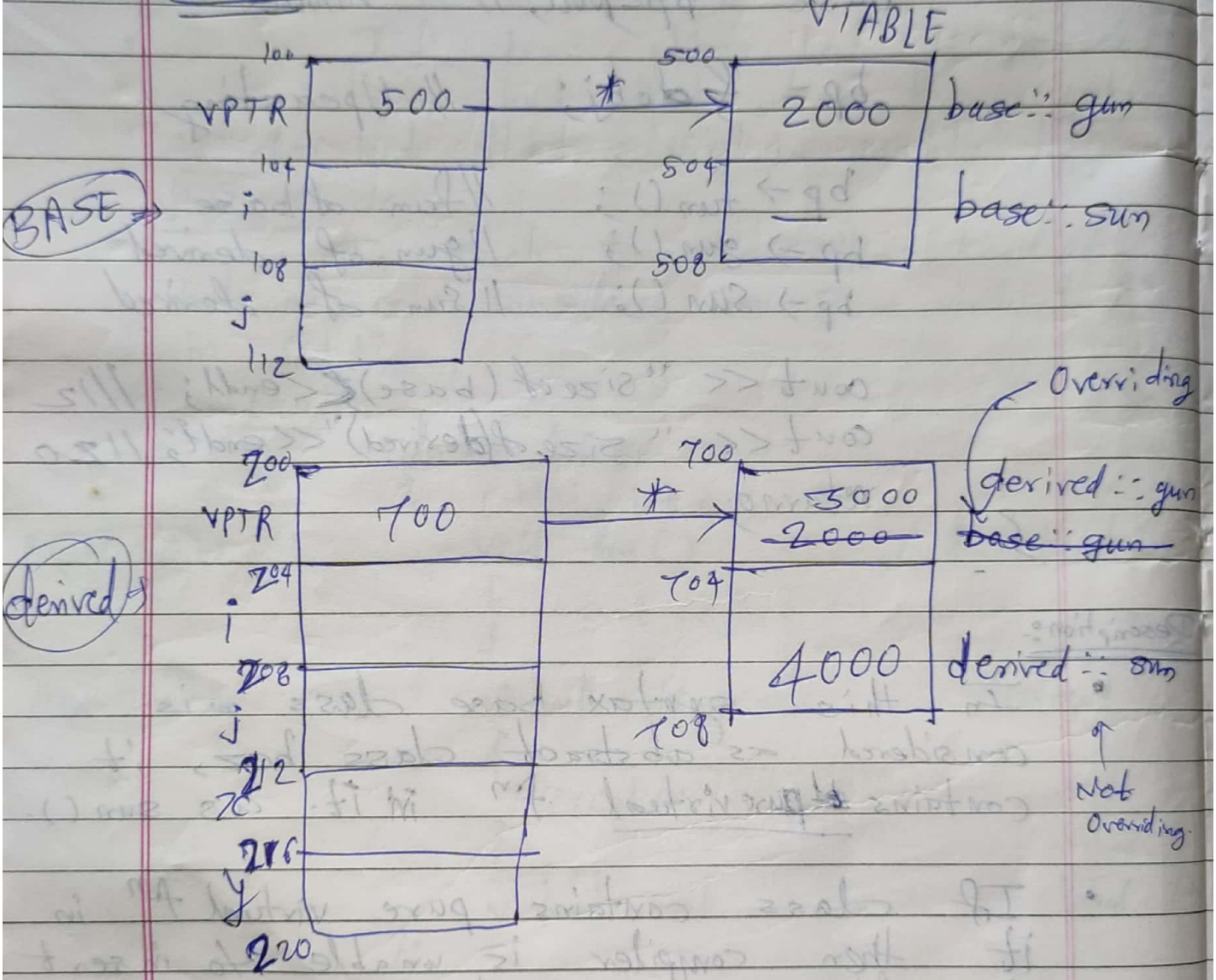
Scanned by CamScanner

at least a single pure virtual fn in it.

11/Nov/2015

- If class is derived from such a class which contains pure virtual fn in it. Then it is responsibility of that derived class to provide defn for that pure virtual fn in it.

- Layout :



- Due to incomplete VTABLE we can't create object but we can create pointer of that class.

- If derived class is unable to provide the defn of that pure virtual fn of that class then

Scanned by CamScanner

compiler generates an error. 4/Nov/2015

- To avoid that error there should at least empty def of that fn or derived class can also declare that fn as pure virtual fn.

  In that case that derived class is also considered as abstract class its entry of VTABLE is still incomplete.

- If our fn is pure virtual fn then we can also provide partial defn for that fn.

- That partial defn should be defined outside the class.

  Still the partial defn is provided by the class still VTABLE of that class is incomplete

- Address of that partial defn is stored in method table but it is not reflected in VTABLE

- That partial defn can be called from derived class.

- Generally that method is called having partial defn from the defn of pure virtual fn in derived class.

Exa:-

Scanned by CamScanner

1/Nov/2018

```
class base
{ public:
    virtual void fun() = 0;
};

void base :: fun()
{
    cout << " Partial Def" of fun
                in " base";
}

class derived : public base
{ public:
    void fun()
    {
        cout << " fun of derived";
        base :: fun();
    }
};

int main()
{
    base *bp = NULL;
    derived dobj;
    bp = &dobj;
    bp -> fun();   // fun of derived
                   // Partial def" of fun in base
    return 0;
}
```

Scanned by CamScanner

Virtual Destructor in C++:-

1/Nov/2015

PAGE:
DATE: / /

- Virtual concept is applicable for destructor in C++.
- But virtual concept is not applicable for the constructor.

- Concept of virtual destructor is similar as normal a virtual fn in C++.
- Consider below class as,

```
class base
{
    public:
        base()
        {
            cout << "Constr of base\n";
        }
        ~base()
        {
            cout << "Destr of base\n";
        }
};
class derived : public base
{
    public:
        derived()
        {
            cout << "Derived Constr\n";
        }
        ~derived()
        {
            cout << "Derived destr \n";
        }
};
```

Scanned by CamScanner

1/Nov/2015

```
int main()
{
    cout<<sizeof(base)<<sizeof(derived);
    //11
    base *bp=new derived();
    delete bp;
    return 0;
}
```

- In this class there is no characteristics still size of the object is displayed as 1 byte.

Imp - Size of empty class is considered as 1 byte bez,

- As there is no characteristics it requires 0 bytes of memory but if we create object of such a class it requires existance in memory to create its symbol table entry.
- Due to which minimum 1 memory gets allocated i.e. 1 byte.

- If we add any other characteristic in that one byte is not considered.

- When we run above prog constr of base class & derived class gets called as we are creating object of derived class.
  In this prog. delete opr is used to free that dynamically allocated memory.

Scanned by CamScanner

- for delete opr bp pointer is the argument & as it is pointer of base class it only calls destructor of base class i.e. pointer type.

1/Nov/2015

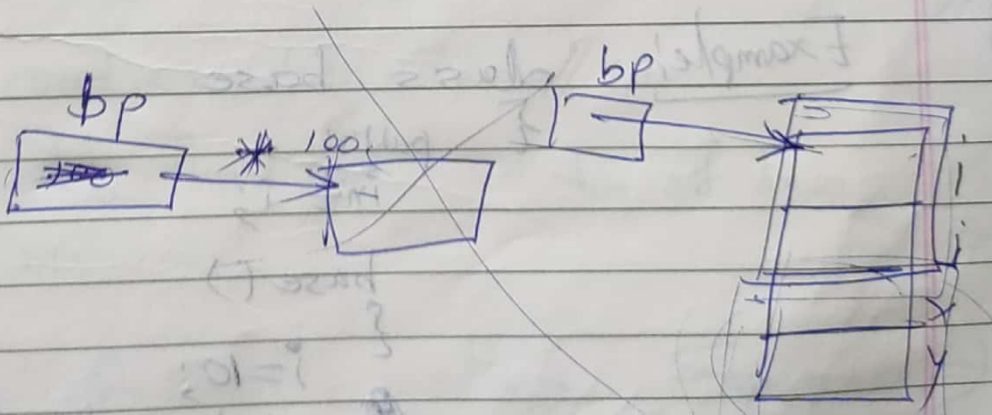
PAGE:
DATE: / /

- Due to this destr of derived class is not called & the resources of derived class are not gets freed.

- To avoid this problem we have to define destr of base class as virtual.

```
virtual ~base()
{
    cout << "destr of base";
}
```

- After writing this destr our delete opr first invoke destr of derived class & then it invokes destr of base class.



Scanned by CamScanner

★ Pure virtual destructor in C++:

PAGE :
DATE : / /

- Concept of pure virtual destructor is same as pure virtual fⁿ in C++.
- If you want to avoid object creation of class then that class should contain pure virtual destʳ in it.
- If class contains pure virtual destʳ then object contains VPTR & that VPTR points to incomplete VTABLE.
- If we want to inherit such a class which contains pure virtual destʳ in it, then we have to provide partial defⁿ for pure virtual destʳ of base class.
- From derived class we can access every characteristics & behavior of base class.

Example:

```
class base
{
public:
    int i;
    base()
    {
        i=10;
        cout << "base constr in ";
    }
    virtual ~base()=0;
    void fun()
    {
        cout << " fun in base\n";
    }
};
```

Scanned by CamScanner

11/Nov/2015

```
base :: ~base
{
    cout << "Partial destr of base\n";
}
class derived : public base
{ public:
    derived()
    {
        cout << "Derived Constr\n";
    }
    ~derived()
    {
        cout << "Destr of Derived\n";
    }
};
int main()
{
    base bobj;      // error
    derived dobj;   // Allowed
    cout << sizeof(base) << endl <<
            sizeof(derived) << endl;
    cout << dobj.i;  // 10
    dobj.fun();      // fun of base
    return;
}
```

// i + VPTR

// 8 8

⇒ // Constructor & destr calling Sequence ⇒

constructor of base
constructor of derived
destruction of derived
partial destructor of base.

Scanned by CamScanner

6/Nov/2015

Friend in C++

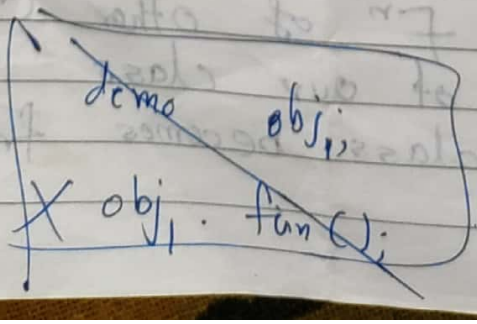
PAGE:
DATE: / /

- This concept is added in C++ which is not available in C.
- Concept of friend is related to access specifiers ~~rela~~ used in C++.
- This concept is only used in C++ which is not allowed in java due to some security problems.
- Every class can contain some characteristics & behavior under public or private or protected access specifier.
  Outsiders of class can access only public members of our class means it is not allowed to access non public members of class for some outsider entity.
- By using this concept of friend we can access non public members of class from some outsider entity.
- Concept of friend is used in C++ in 3 different ways.
  1. Naked f^n becomes friend of our class.
  2. Member f^n of other class becomes friend of our class
  3. Whole class becomes friend of our class

Scanned by CamScanner

* [1] Nacked f^n becomes friend of our class:-

```
class demo
{
    public:
        int pub;
        demo()
        {
            pub=10;
            pri=20;
            pro=30;
        }
    private:
        int pri;
    protected:
        int pro;
    friend void fun();
};
void fun()
{
    demo obj;
    cout << obj.pub<< obj.pro << obj.pri;
}                                   //10 30 20
int main()
{
    fun();
    return 0;
}
```



Scanned by CamScanner

Description:-

- In this program pro, pri are considered as non-public members of class.
- fun() is considered as nacked fn which is not a part of any class.
- As our fun() fn wants to access nonpublic members of demo class, it should be declared as friend of class.
- After declaring that fn as friend fn it can access all the members of that class.

[2] Member fn of one class becomes friend of other class:-

```
class hello
{
    public:
        void fun();
};
class demo
{
    public:
        int pub;
        demo()
        {
            pub = 10;
            pri = 20;
            pro = 30;
        }
    private:
        int pri;
    protected:
        int pro;
```

Scanned by CamScanner

```
friend void hello :: fun();
};

void hello :: fun()
{
    demo obj;
    cout << obj.pub << obj.pri << obj.pro;
}
//10 20 30

// In this
int main()
{
    hello hobj;
    hobj.fun();
    return 0;
}
```

demo dobj;
dobj.~~fun~~(); ✗

- **Description:** In this example fun() is the member fn of class hello, which is declared as friend in demo class. Due to the concept of friend our fun() fn can access non public members of demo class.

Scanned by CamScanner

3 Whole class becomes friend of our class:

```
class hello
{
    public:
        void fun();
        void gun();
};

class demo
{
    public:
        int pub;
        demo()
        {
            pub = 10;
            pri = 20;
            pro = 30;
        }
    private:
        int pri;
    protected:
        int pro;

    friend class hello;
};

void hello :: fun()
{
    demo obj;

    cout << obj.pub << obj.pri << obj.pro;
                                    // 10 20 30
}
```

Scanned by CamScanner

```
void hello :: gun ()
{
    demo obj;
    cout << obj.pub << obj.pri << obj.pro;
}
int main()
{
    hello hobj;
    hobj.fun();     //10 20 30
    hobj.gun();     //10 20 30
    return 0;
}
```

- **Description:-**

In this example our whole hello class becomes friend of demo class.

As whole class is friend both the f^ns of that hello class, becomes friend of demo class.

```
demo dobj;
dobj.fun();
cout << dobj.pub;
cout << dobj.pri;
```

Scanned by CamScanner

6/Nov/2015

IMPORTANT Points about friend :-

PAGE:
DATE: / /

- 1) There is only one concept in C++ which allows to access private members of class.
- 2) In inheritance also we can't access private member of class.
- 3) If any f^n is declared as friend f^n of class then it is not considered as its member f^n.
- 4) We can declare friend f^n under any access specifier.
- 5) If any of the f^n is friend of our class then that f^n is not considered as friend for its derived class.
- 6) Inside friend f^n to access the non-public member we have to create object of that class explicitely.
- 7) There is no such concept called as this pointer for the friend f^n.
- 8) If A class becomes friend of B class then B class is not considered as friend of A class.
- 9) We can use ~~of the~~ concept of friend with the concept of static.
- 10) We can use concept of friend with the concept of virtual f^ns.

In C++ development process if any of the member f^n wants to access non public members of other class to get the desired o/p, There is no need to inherit that class.

Scanned by CamScanner

6/Nov/2015

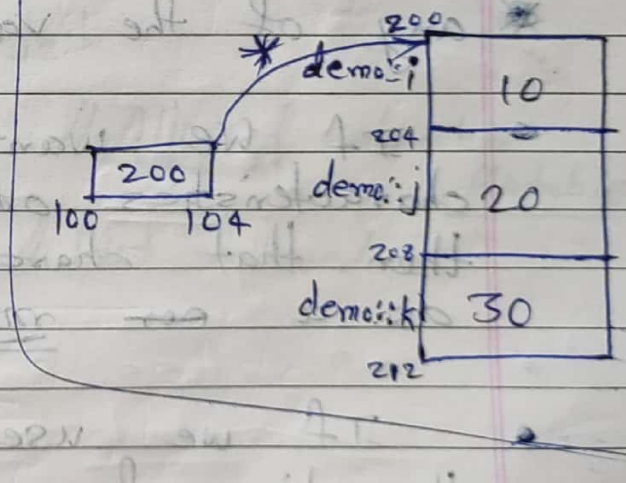
Instead of using concept of friend or removing the access specifiers from the class it is good programming practice to declare that class or fn as friend of other class.

11) • Our friend function can also access static members of class. The change performed by the friend fn is reflected in actually memory layout of class.

Scanned by CamScanner

Non Public member Access by using pointer :-

```
class demo
{
    public:
        int i;
    private:
        int j;
    protected:
        int k;
    public:
        demo()
        {
            i=10;
            j=20;
            k=50;
        }
};

int main()
{
    demo obj;
    int *p = NULL;
    p = (int *) &obj;
    cout << *(p+0);    // 10
    cout << *(p+1);    // 20
    cout << *(p+2);    // 30
    return 0;
}
```



Scanned by CamScanner

- According to above code snippet we can proove that by using concept of pointer we can access anything from objat irrespective of it's access specifier [but it is bad programming practice]

6/Nov/2015

★ Mutable in C++ :-

- Concept of mutable is added in C++ due to the concept of constant Object.
- If we create constant object of a class then all the characteristics of that class becomes constant.
  Due to which we can't change any of the value of that characteristics.
- If we want to change any of the characteristics of the constant object then that characteristics should be declared as a mutable characteristics.
- If we use mutable keyword at the time of declaration of that characteristics compiler allows to change that contents irrespective the constant object.

Example:-

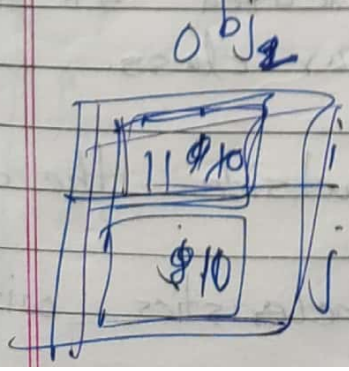
Scanned by CamScanner

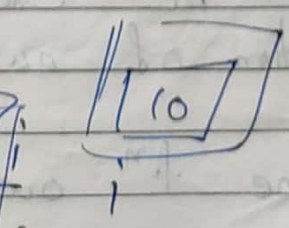
6th Nov 2015.

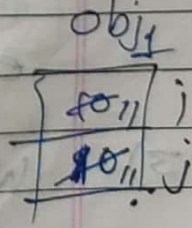
```
class demo
{
    public:
        mutable int i;
        int j;
        demo()
        {
            i = j = 10;
        }
};
int main()
{
    demo obj1;

    obj1.i++;      // Allowed.
    obj1.j++;      // Allowed

    const demo obj2;

    obj2.i++;      // Allowed due to Mutable
    obj2.j++;      // Not allowed.

    return 0;
}
```

const int i = 10;

i++;







Scanned by CamScanner

/Nov/2015

PAGE :
DATE :

Unnamed class in C++ :-

- There is concept called as unnamed structure, unnamed union, unnamed enum unnamed bitfield in C.
- The similar concept is used in C++ as unnamed class.

Exa:-

```
class ~~~~~~        // Unnamed class
{
        public:
                int i;
                static int j;
                void fun()
                {
                        cout << "Inside fun\n"
                                << this->i;
                }
} obj;
```

1) - If class is unnamed class there is no symbol table entry for that class.

2) - We can't create object of unnamed class after its declaration. Object should be created at the end of declaration of our class.

3) - We can't define f^n outside the class.

4) - We can create static characteristics without

Scanned by CamScanner

definining it outside the class but we can't access that characteristics. It generates runtime error.

PAGE:
DATE: / /

5). We can't write const^r or dest^r for that class.

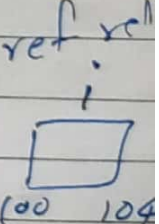
6). As there is no const^r & dest^r all the characteristics of that class gets initialized with its default value 0, 0.0, '\0'.

7). We can't inherit unnamed class bcz, there no name for that class.

8). We can use this pointer inside member fn of unnamed class.

9). If you want to restrict the user from object creation & reusability we can create that class as unnamed class.

IMP



int & ref = i;
& ref2 = i;

Symbol

Name	Visibility Attribute	From	To	Another
i		100	104	ref, ref1

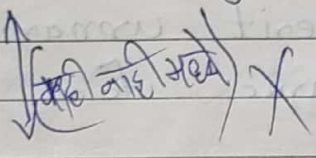
fun
demo

Scanned by CamScanner

7/Nov/2015

Generic Programming in C++:

- Generally programs are designed according to specific requirement.
- If requirement change then we have to change our program by maintaining same logic.
- Due to this approach it creates redundancy in programming.
- To avoid that redandancy we have to use concept of generic programming.



- Concept of generic programming is also available in C, which can be achieved by (Concept of) generic pointer (void pointer).
- This style of programming by using void pointer becomes complex, after certain level to use void pointer.

To avoid that complexity in C, C++ provide inbuilt programming lang. concept which can be used for generic programming.

That concept which is used for generic programming is called Templates in C++.

7/Nov/2015

PAGE:
DATE: / /

* There are 2 types of templates in C++:-

1. function Template:-
2. Class Template:-

X 1. Function Template:-

- 1) If you want to write a fn multiple times due to multiple input arguments then instead of writing defn multiple times we have to use the concept of fn template.

- 2) Consider the below prog as,

```
int Add1(int no1, int no2)
{
    int Ans = no1 + no2;
    return Ans;
}

float Add2(float no1, float no2)
{
    float Ans = no1 + no2;
    return Ans;
}

double Add3(double no1, double no2)
{
    double Ans = no1 + no2;
    return Ans;
}
```

Scanned by CamScanner

7/Nov/2015

In above functions the logic which is used to write that functions is exactly same but due to the changing datatypes we have to write the same fn multiple times.

Above application is example of specific programming in which different function is written for specific datatype.

In above programm if user pass the arguments as integer / float / double then our fn works.
But if user pass some new datatype as character. then our fn fail.

To avoid that failure & redundancy we have to use concept of generic programming by using function templates.

* Steps to convert specific program into generic program:

1) 1st write specific prog. depend on requirement
2) check whether basic logic which is used in that prog. is same or not.
3) Try to observe such entities which are changing in every fn

Scanned by CamScanner

7/Nov/2015

/* template < class T > */
↑ (or) classname (or) forname

PAGE:
DATE: / /

- 4) Write template header before any of the function.
- 5) Replace every ~~ent~~ changing entity with the template variable
- 6) Remove all the functions of specific programming.

---

* In our above application (Addition example) the underlined datatypes are going to be changed.

By following the above rules our template fn be comes,

Regular class सारखा नाही. म्हणजे class ला redefine करत

// Keyword // Syntax // Template Variable

```
template < class T >

T Add ( T No1 , T No2 )
{
    T Ans = No1 + No2;
    return Ans;
}

int main ()
{
    int A = 10, B = 20 , C;
    C = Add (A, B);   // C = 30
```

Scanned by CamScanner

float f = 3.14f, g = 6.10f, h;

h = Add(f, g); // h = 9.24

return 0;

}

Description:-

In above program there is only one fⁿ which accept template argument as input.

From main fⁿ we are calling Add fⁿ 2 times. one by passing integer parameter & other by passing float parameters

When we compile this prog. our compiler will create separate defⁿ of Add() fⁿ which accept integer as argument & it also create separate defⁿ which accept float as argument.

This process of creating the separate defⁿ depend on the fⁿ call & its datatype is called as "template instantiation"

If we call fⁿ N times, by passing diff datatypes to it compiler creates N diff. definitions for every separate datatype by using concept of template instantiation

Scanned by CamScanner

But if we call the fn 'N' times by using same datatype then only one defn gets created (for that N calls) by the compiler.

★ Important points about Template:

1) If in fn template there is no input argument as template then compiler is unable to perform instantiation process. due to which it generates compiletime error.

We can't only use template variable as local variable or return value

example:

```
template < class T>

void fun (int no)
{
    T num;
}
int main ()
{
    fun (10);    // Error
    fun (2.0);   // Error
}
```

In above syntax compiler generates error bcz, compiler is unable to predict what should be the datatype of template variable T. depend on the i/p argument.

Scanned by CamScanner

• 2) Same problem occurs if we use return value as template variable.

Exa:-

```
template< class T>
T fun (int No)
{
    // Code;
}
int main()
{
    fun(10);    // Error
    fun(20);    // Error
}
```

• 3) According to Above 2 points for successful template instanciation there should be atleast 1 i/p argument as template variable.

Imp

• 4) We can also use concept of template for creating array statically or dynamically as,

```
template < class T>
void fun(T value)
{
    T Arr[10];
    T* p = new T[10];
}
```

Scanned by CamScanner

In above syntax array Arr can be created by using input arguments datatype.

That generic array creation concept is applicable for dynamic memory allocation also.

5) • We can use multiple template arguments for same fn. By using this concept we can pass diff. datatype to that fn.

Exa:-

```
template< class T1, class T2>
void fun(T1 No1, T2 No2)
{
    // Code;
}
int main()
{
    fun(10, 3.14f); // Allowed
    fun(3.45f, 10); // Allowed
    fun(10, 10); // Allowed
    return 0;
}
```

★ [2] Class Template :-

7/Nov/2015

PAGE :
DATE : / /

Consider below example which implement stack datastructure in object oriented manner.

```
class stack
{
    public:
        int top;
        int *p;                    // characteristics
        int size;
        stack (stack &);           // Copy Constr
        stack (int);               // Parameterised Constr
        ~stack ();                 // Destructor
        void push (int);           // Behaviors
        int pop ();
};
// Parameterized constr
stack :: stack (int value)
{
        size = value;
        top = -1;
        p = new int [size];
}
// Copy Constr
stack :: stack (stack &ref)
{
        this -> top = ref.top;
        this -> size = ref.size;
        this -> p = new int [size];
```

Scanned by CamScanner

```
int i=0;
for(i=0; i<=top; i++)
{
    p[i] = ref.p[i];
}
}
// Destructor
stack::~stack()
{
    delete []p;
}

void stack::push(int value)
{
    if (top==(size-1)) // Stack full
    {
        return;
    }
    p[++top] = value;
}

int stack::pop()
{
    if(top==-1) // Stack Empty
    {
        return -1;
    }
    return p[top--];
}

int main()
{
    stack obj1(15); // Parameterised Const
    obj1.push(11);
    obj1.push(21);
    obj1.push(51);
```

// DeepCopy

Scanned by CamScanner

7/Nov/2015

```
stack obj3(obj1);          // Copy constr.
cout << obj3.size() << endl;          //15

cout << obj1.pop() << endl;          //5
cout << obj1.pop() << endl;          //2
cout << obj.pop() << endl;          //1

cout << obj3.pop() << endl;          //5
cout << obj3.pop() << endl;          //2
cout << obj3.pop() << endl          //1

return 0;
}
```

- Above class only works for stack of integers.

If we want to create stack of characters or float we have to design separate class.

To avoid this redundancy we can use the concept of class template.

- Our above class is a specific class.

To convert that specific class into generic class we have to replace every changing datatype into template variable.

Scanned by CamScanner

1/Nov/2015

PAGE:
DATE: /

Generic Program for Datastructure stack

```
template <class T>
class stack
{
    public:
        int top;
        T *p;
        int size;
        // characteristics

        stack(int);          // Parameterized Constr
        stack (stack &);     // Copy Constr

        ~stack();

        void pust(T);
        T pop();             // Behavior
};
// Parameterized constr
template <class T>
stack<T> :: stack(int value)
{
    size = value
    p = new T[size];
    top = -1;
}
// Copy Constr
template <class T>
stack<T> :: stack(stack &ref)
{
    this->top = ref.top;
    this->size = ref.size;
```

Scanned by CamScanner

```
        this -> p = new T [size];
        int i=0;
        for(i=0; i<=top; i++)
        {
            p[i] = ref.p[i];
        }
}

template <class T>
stack<T> :: ~stack()  //destr.
{
    delete []p;
}

template <class T>
void stack<T> :: push(T value)
{
    if(top == (size-1)) //Stack full
    {
        return;
    }
    p[++top] = value;
}

template <class T>
T stack<T> :: pop()
{
    if (top == -1)  //Stack empty
    {
        return -1;
    }
    return p[top--];
```

Scanned by CamScanner

```
int main()
{
    stack <int> obj1[10]; //stack of integer
    stack <char> obj2[5]; //stack of character

    obj1.push(11);
    obj1.push(21);

    obj2.puch('A');
    obj2.push('B');

    cout << obj1.pop() << endl;   21
    cout << obj1.pop() << endl;   11

    cout << obj2.pop() << endl;   B
    cout << obj2.pop() << endl;   A

    return 0;
}
```

- Every fn of class template is fn template.
- If we want to create object of class template we have to specify the datatype for that template argument.
- Same above concept is used in STL [standard template Library] in C++.

Scanned by CamScanner

- Same concept is used in collection framework of java.

PAGE :
DATE : / /

- Every class of STL & collection framework is implemented in generic manner.

Scanned by CamScanner

8/Nov/2015

PAGE:
DATE: / /

Namespaces in C++:-

- If we want to use some outsiders entity then every language has its own concept.
- 'C' language provides concept called as header files.
- C++ provides concept of namespace
- Java provides concept of packages.
- As C++ is used for team development due to the naming strategy used by the developer there are some chances of name clashes.
- To avoid this problem of name clashes we have to use concept of name-spaces in C++.
- There are 2 types of namescapes in C++ as,
  1) Predefined Namespaces
  2) Userdefined Namespaces.

- **I] Predefined namespace:**

If name space is provided by lang designer then that namespace is called as predefined namespace

Std is frequently used namespace from C++ to perform input output activity.

Scanned by CamScanner

8/Nov/2015

2] User defined namespaces:-

As a programmer we can create our own namspace & can use that namespace.

Exa:-

```
namespace demo
{
    class base
    { public:
        void fun()
        {
            cout<<"fun of base"<<endl;
        }
    };
}

int main()
{
    demo :: base bobj1;
    bobj1.fun();

    using namespace demo;

    base bobj2;
    bobj2.fun();
    return 0;
}
```

* Description:-

In this example demo is our user defined namespace in which contain base class as its member.

We can use contents of namespace using :-

1) using keyword
2) :: operator.

Scanned by CamScanner

- To access the members of that namespace we have to specify name of that namespace & name of that member.
- Inside namespace we can write anything [syntactically correct].
- Inside one namespace we can write multiple classes.
- If we write the syntax as
  using namespace namespacename

  There is no need to provide name of namespace after that line.

Nested Namespaces:

Like nested structures or nested union or nested class we can also create nested namespace.

There is no restriction on level of nesting on namespace.

Eg:-

```
namespace outer
{
    class base
    {
        //code
    };
    namespace inner
    {
        class hello
        {
            //code
        };
    }
}
```

Scanned by CamScanner

8/Nov/2

```
int main()
{
    outer::base bobj1;
    outer::inner::hello hobj1;

    using namespace outer;
    using namespace outer::inner;

    base bobj2;
    hello hobj2;

    return 0;
}
```

// In this example there are 2 namespaces named as inner & outer which are internally nested.

We can also access members of inner namespace by using the concept of scope resolution opr or using keyword.

★ Unnamed Namespace:

Like unnamed structure, unamed union, unamed class, unamed enum we can also create unnamed namespace

(Imp) ★ If we want to restrict outside to access the contents of our namespace or file then we can write our content inside unnamed namespace.

The contents of unnamed namespace is only accessible inside the file in which it is written.

Scanned by CamScanner

- We can't use using keyword or scope resolution opr to access the contents of unnamed namespace.

- Generally unnamed namespace is used for security perpose.

Exa:-

```
namespace
{
    class base
    {
        //code
    };
}
int main()
{
    base obj;    // Allowed
    return 0;
}
```

Scanned by CamScanner