VIRTUAL FUNCTIONS ~ C++

* Virtual functions are declared using the "virtual" keyword. If a function is declared virtual in a base class, it automatically becomes virtual in the derived classes. However sometimes, virtual is explicitely written even in derived classes.

*
```
class Employee {
    public:
    void print() {
        cout << "Employee";
    }
    virtual void raiseSalary() {
    }
};
class Manager : public Employee {
    public:
    void print() {
        cout << "Manager";
    }
    void raiseSalary() {
    // some code
    }
};
class Worker : public Employee {
    public:
    void print() {
        cout << "Worker";
    }
    void raiseSalary() {
    // some code
    }
};
```

Date:

* Now, for :-
- Objects of Employee / Reference of Employee holding Employee / Pointer of Employee holding Employee address :- Calling print() will print "Employee" and calling raise salary will call Employee :: raiseSalary().
- Objects of Manager / Reference of Manager holding Manager / Pointer of Manager holding Manager address :- Calling print() will print "Manager" and calling raise salary will call Manager :: raiseSalary(). It should be noted that this is ~~ope~~ method overriding and has nothing to with virtual.
- A reference / pointer of Employee holding a Manager object :

```
Employee* emptr;
emptr = new Manager();
emptr->print();
emptr -> raiseSalary();
```

The print() method is statically bound i.e. it gets chosen during compile time and will follow the type of pointer i.e. print "Employee".
The raiseSalary() method is dynamically bound i.e. it gets chosen during runtime depending upon the type of object being pointed to. So it will call Manager :: raiseSalary()

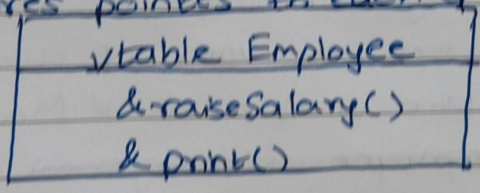
* The advantages of dynamic binding a.k.a. run-time polymorphism are that during writing the program, the coder doesn't have to remember / may not know which pointer points to what type of Object.
For example :-

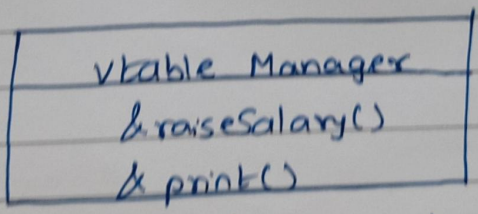
```
int main(){
    Employee* emplist[4];
```

```
// code which depending on user input initializes
// each emplist pointer to either a Manager
// or a Worker.

// now the coder wishes to raise everyone's salaries.
// one way to do it is to check if each element
// of emplist[] points to Worker or Manager
// and then call appropriate functions.
// But this is tedious.
// Instead we make use of virtual :

for(int i=0 ; i<4 ; i++){
    emplist[i] -> raiseSalary();
}
}
```

* How C++ implements this :-

The compiler adds to each class a table called vtable that stores pointers to each of its functions. So





Also, the compiler adds to each object of each class, a pointer called vptr that points to this table.

So,

Employee emp; —vptr→ vtable Employee

Manager man; —vptr→ vtable Manager

It should be noted that normal function calls donot make use of the vptr and vtable. That is, when static binding is present, the compiler has already chosen the base class function to be executed. However, when the base class function is virtual and dynamic binding occurs, the vptr of the object (and not of the pointer, pointer doesn't have vptr its just a hexadecimal address) is used as so vtable of derived class is accessed. So :-

```
Employee* emptr;
emptr = new Manager();
emptr -> print();    // line 1
emptr -> raiseSalary(); //line 2.
```

line 1 has static binding since print() isn't virtual. So it is statically / fixedly bound to Employee::print() with no use of vptr.
However line 2 has dynamic binding as raiseSalary() is virtual. So it uses the vptr of the object and hence uses the vtable of Manager, hence calling ~~Employee::Manager()~~ Manager::raiseSalary()