Objective:

- To look into the technical and theoretical aspects of the Polymorphism.

Task 1:

A student is assigned to make a program. The program is supposed to play the voice of animal selected. For this purpose student comes up with two alternate designs. Which design do you think is better and WHY (give some logical reason(s))?

//Solution-A

```
class Animal
{
public:
   Animal();
   virtual void speak()
   { }
};
class Cat : public Animal
{
public:
     void speak()
     { cout<<"Meeeaaauuun"; }
};
class Dog : public Animal
{
public:
     void speak()
     { cout<<"Baao Baaaaaaoooo"; }
};
class Rat : public Animal
{
public:
     void speak()
     { cout<<"chi een eeeen"; }
};
```

//Solution-B

```
class Animal
{
public:
    enum EnumType { Cat, Dog, Rat };
    Animal( EnumType type );
    void speak();
private:
    EnumType animalType;
};
void Animal::speak()
{
    switch ( animalType )
    {
          case Cat:
                cout<<"Meeeaaauuun";
                break;
          case Dog:
                cout<<"Baao Baaaaaaoooo";
                break;
          case Rat:
                cout<<"chi een eeeen";
                break;
    }
}
```

Task 2: POLYMORPHISM

Polymorphism appears in C++ in two different forms

1. **Static/Compile Time/adhoc Polymorphism**
   1.1. Function/Operator Overloading
   1.2. Parameterized [Template] Polymorphism
2. **Dynamic/Runtime Polymorphism**

1.1 Function/Operator Overloading

In order to overload a function, a different list of parameters is used for each overloaded version. For example, a set of valid overloaded versions of a function
named f() might look similar to the following:

```
void f(char c, int i);
void f();
void f(int i);
```

So, a function named as " f " exists in more than one shapes(different parameter list). And which shape of " f " to call? The decision is made on compile time.

Similarly, if we observe the following + operations:

```
int a; float b; double d; char t;
a+b;
b+a;
a+d;
t+a;
```

The same " + " operator is performing "additions operation" on different types of data. And this concept can be extended to user defined classes.
So which overloaded operator function to call? The decision is made at compile time.

1.2Parameterized [Template] Polymorphism

A generic (template) function/class can be defined out of which actual function /class can be generated. The generation of actual function/class is dependent on the (in case of function: on function call parameter, in case of class: on object creation)
So, again the decision is made at compile time,

```
Template<class T>
Void f(T a)
{}
Template<class T>
class Test
{
        T data;
public:
        Test()
        { data=0; }
};
……..
f(12);              //compiler generates void f(int) considering the call parameter.
Test<float> fun; //Compiler generates class "Test<float>" replacing type T by float
                 //Wherever T is used in class template
Test<int> you; //this time: class Test<int> is generated.
```

2. Dynamic/Runtime Polymorphism

Object interaction take place by sending messages to them.
When a message (member function call) is sent to an object, it will react by executing an appropriate method (function body).
Let's see an example:

```
class base
{
public:
        void f() {}
};
class derive : public base
{
public:
        void f() {}
};
void main()
{
        derived d;
        base *pb = & d;
        pb->f();
}
```

In the above example "**static type**" of "pb" is "base*" and "**dynamic type**" of "pb" is the type of pb's referent (derive*).
If we put the duty of "appropriate method invocation (binding)" to compiler it will do silly things. Because compiler has knowledge of only "static type" of an identifier, It will invoke base::f() rather than derive::f() due to the "static type" of "pb" is "base*". We have to find another mechanism that has access to "dynamic type" of an identifier.
Let's update the above example:
Updated example:

```
class base
{
public:
        virtual void f() {}
};
class derive : public base
{
public:
        void f() {}
};
void main()
{
        derived d;
        base *pb = & d;
        pb->f();
}
```

Now pb->f() binds to derive::f();
Miracle: because of overriding

Overriding:

1. The method in base class is declared as virtual"
2. The derive class method is exactly **"same type function"** as the base class method. That means the derive class method has the same return type, the same formal arguments, and it is a const member function if and only if the base member function is also const.

Overriding is precisely what is necessary for runtime polymorphism/binding
So, in short runtime **polymorphism** means:

Polymorphism:

"**Objects of different classes related by inheritance respond differently to the same message.**"

Binding: means; which function body to invoke on a function call. If the decision is taken at compile time, it is called **compile/static binding** and if the decision is taken at runtime then it is called **runtime binding.**

```
class base
{
public:
        virtual void f() {}
};
class derive : public base
{
public:
        void f() {}
};
class Leaf : public derive
{
public:
        void f() {}
};
void main()
{
        derived d;
        Leaf f;
        base *pb = &d;
1……..pb->f();       //derive::f()
        pb = &f;
2……..pb->f();       //Leaf::f()
}
```

The same message "pb->f()" at line 1 & 2 but different responses(different method invocation/binding).

The following are synonyms:

- Early=Compile Time=Static binding
- Late=Dynamic=Runtime binding

Overloading vs Hiding vs Overriding:

- **Overloading**
  - Two functions that are overloaded must have the same scope but different signatures. Their formal argument lists are different or one of the is const and the other is not.
- **Hiding**
  - A function declared in derived class with the same as in base class hides the base class method and does not override it.
- **Overriding**
  - Already defined above….

Coding Example to differentiate among hiding, overriding and overloading.

```
class base
{
int a;
public:
        virtual void f(int)
        {
        }
        virtual void g(int)
        {
        }
        void h(float)
```

```
        {
        }
};

class derive : public base
{
int b;
public:
        void f(int) // overrides base::f(int)
        {}
        void f(double) //overloads derive::f(int) doesn't override
        {}              //base::f(int)

        char g(int)     //illegal: return must mach for overriding
        {}              //virtual functions.

        char g(char)   //Legal : but doesn't override base::g(int).
        {}              //signatures are different

        void h(float)  //just hides base::h(float)
        {}

        void h(int)     //Legal: since base::h is not virtual. Overloads
        {}              //derived::h(float)
};
```

Runtime Polymorphism:

Some points to ponder:

1. If the return type is a pointer or reference to the base class, then it may be a pointer or reference to the derived class in the derived class signature. Otherwise, the return type must match in any derived class.

```
class base
{
public:
        virtual base * f();
        virtual ~base();
};
class derive : public base
{
public:
        virtual derive * f();
        virtual ~derive();
};
```

2. Virtual global friend is illegal, but virtual functions in one class may be designated as friend in another class.

3. Virtual function may be inline and have default arguments.

4. Changing the access level of a virtual function in a derived class is a poor idea. For example:

```
Class base
{
        public:
                virtual void fun()
                {}
};
Class derive : public base
{
        private:
                virtual void fun()
                {}
};

void main()
{
        base *pb=new derive;
        pb->fun();
}
```

Note: The statement above "pb->fun()" is valid, because access specifiers are compile time check, so at compile time, compiler checks the type of pointer "pb" which is "base", so "pb" can access "base::fun()", But at runtime because of dynamic binding "pb->fun()" resolves to derive::fun().

5. **Using** declaration help derived class to avoid hiding of base class functions**.**

Example 1:

```
Class base
{
```

```
        public:
                void print(int , int)    {}
};
Class derive
{
        public:
                using base::print;          //bring into local scope
                void print(char)   {}       //overloads with derive::print(char)
};
```

Example 2:

```
class base
{
        public:
                virtual void fun(int , int)  {}
                virtual int fun(float)  {}
                virtual void fun(char *)  {}
                virtual void fun()  {}
};
class derive : public base
{
        public:
                using base::fun;  //bring rest of three functions into local scope
                void fun(int , int)  {}
};
```

NOTE:
Without making use of "using declaration", derive::fun(int,int) hides the other versions of fun().

6. The virtual function mechanism works only with public derivation.

7. The virtual function mechanism doesn't work with private derivation.

```
Class base
{
        public:
                virtual void fun() {}
};
class derive: private base
{
        public:
                void fun() {}
};

void main()
{
base *pb = new derive;  // illegal: conversion from 'class derive *' to 'class
                        //base *' does not exists
base *pb=(base*) new derive;  //Legal: now pb->fun() resolves to
pb->fun();                    //         derive::fun()
}
```

8. But what about this?

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

```
};
class derive: private base
{
 public:
     void fun()
     {
     base *pb = new derive; //Legal
     pb->fun();
     }
};
```

REASON:

The main function is just like a client which is using base and derive class. So, a client can't access private part of a class that is why client can't manipulate derive class object using its base class pointer (privately inherited).

9. The virtual function mechanism doesn't work with protected derivation.

```
Class base
{
        public:
                virtual void fun() {}
};
class derive: protected base
{
        public:
                void fun() {}
};
void main()
{
base *pb = new derive;   // illegal: conversion from 'class derive *' to 'class
                         //base *' exists, but is inaccessible
base *pb=(base*) new derive;  //Legal: now pb->fun() resolves to
pb->fun();                    // derive::fun()
}
```

10. But what about this?

```
class base
{
public:
       virtual void fun() {}
};
class derive: protected base
{
 public:
     void fun()
     {
     base *pb = new derive; //Legal
     pb->fun();
     }
};
```

REASON:

The same reason as in point 8, the difference is that the 'class base' can be further inherited in a class derived from 'class derive'.

11. What happens when we call virtual function inside constructor or destructor:

Example

```
class base
```

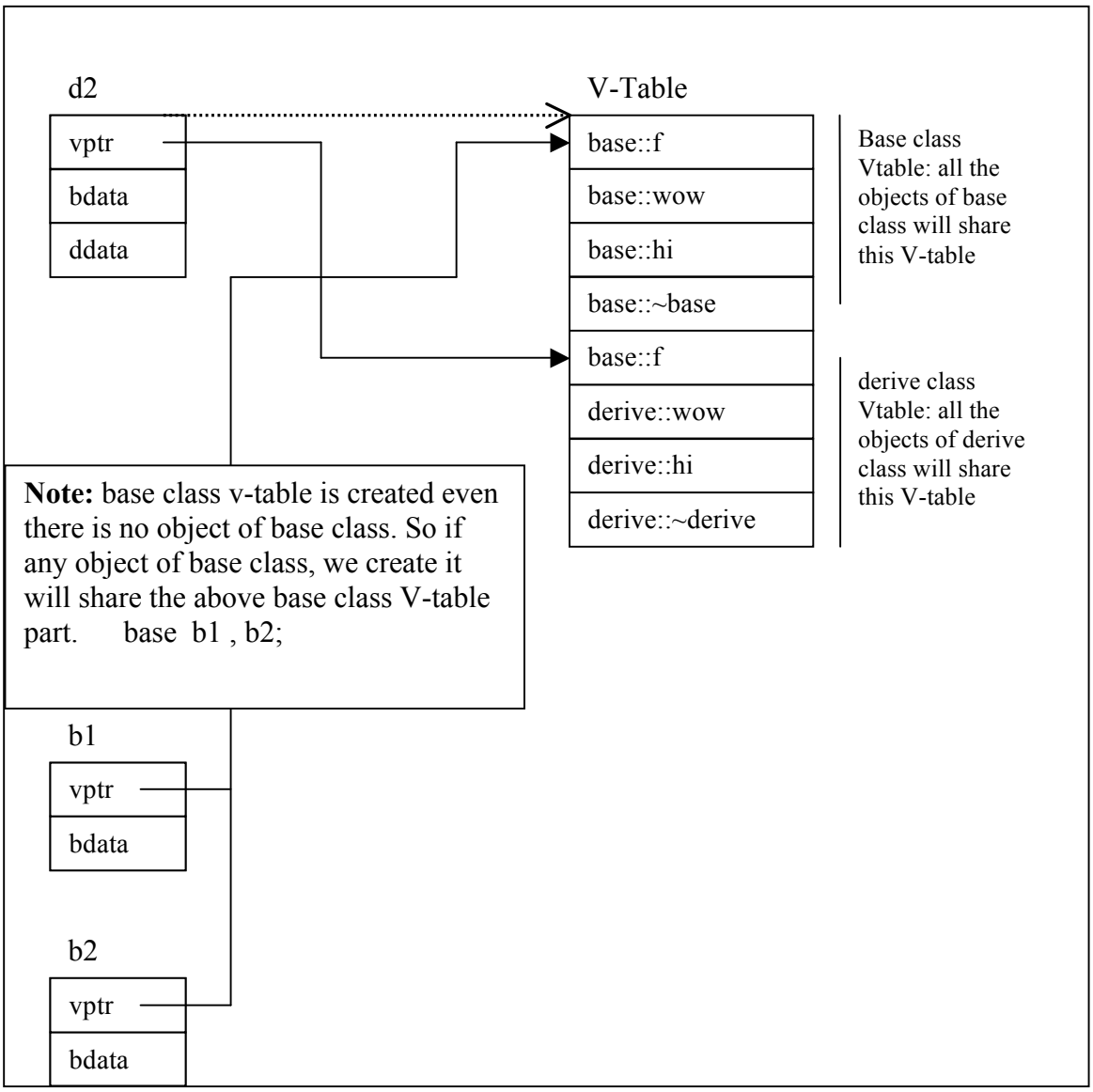
```
{
public:
        virtual void f()
        {
                wow();          //if class to f() is from constructor/destructor of
                                //base class then base::wow() Otherwise //derive::wow()
        }
        virtual void wow()    {}
        virtual void hi()     {}
        base()
        {
                hi();           //base::hi()
        }
        virtual ~base()
        {
                hi();           //base::hi()
        }
        int bdata;
};
class derive : public base
{
public:
        virtual void wow()    {}
        virtual void hi()     {}
        derive()
        {
                hi();           //derive::hi()
                f();            //base::f()
        }
        virtual ~derive()
        {
                hi();           //derive::hi()
                f();            //base::f()
        }
        int ddata;
};
```

NOTE Strange lets look at the layout of v-table construction

When the derive object d2 is created the following layout in memory was constructed (i.e; the base class v-table part along with derive class v-table part. So, the interesting thing about this figure is that the d2 vptr is pointing to two different location. Certainly not at a time. Actually when d2 object was being constructed, in constructor/destructor of base class the d2 vptr was pointing to base class v-table part and other than this vptr value of d2 object will remain same : pointing to its own derive class v-table part.)

……..

```
derive d;
```


d2
vptr
bdata
ddata
V-Table
base::f
base::wow
base::hi
base::~base
base::f
derive::wow
derive::hi
derive::~derive
Base class Vtable: all the objects of base class will share this V-table
derive class Vtable: all the objects of derive class will share this V-table
Note: base class v-table is created even there is no object of base class. So if any object of base class, we create it will share the above base class V-table part. base b1 , b2;
b1
vptr
bdata
b2
vptr
bdata

12. Pure Virtual functions can also have definition but out of line.

13. A destructor can also be pure virtual and we have to give its definition out of line otherwise all the derive classes will be abstract.
Important point over here is that if the based class defines a pure virtual dtor then all the derive classes have to define their dtors.

```
class base
{
public:
        virtual void f()=0;
        virtual ~base()=0;
};
void base::f()                  // always out of line
{
}
void base::~base()              // always out of line
{
}
class derive : public base
{
public:
        virtual void f()
        {
        }
        virtual ~derive()
        {
        }
};
void main()
{
        derive d;
        d.f();
        d.base::~base();   //Now there will be
        d.base::f();         //no linker error
}
```

14. What is a "**Virtual Constructor**"?
Technically speaking, there is no such thing. You can get the effect you desire by a virtual "clone()" member fn (for copy constructing), or a "fresh()" member fn (also virtual) which constructs/creates a new object of the same class but is "fresh" (like the "default" [zero parameter] ctor would do).
The reason ctors can't be virtual is simple: a ctor turns raw bits into a living object. Until there's a living respondent to a message, you can't expect a message to be handled "the right way".
You can think of ctors as "factories" which churn out objects.
Thinking of ctors as "methods" attached to an object is misleading.
Here is an example of how you could use "clone()" and "fresh()" methods:

```
class Set
{
  public:
        virtual void insert(int);     //Set of "int"
        virtual int  remove();
        //...
        virtual Set& clone() const = 0;       //pure virtual; Set is an ABC
        virtual Set& fresh() const = 0;
        virtual ~Set()
        { }
};
```

```
class SetHT : public Set
{
   public:
        //...
        Set& clone() const
        {
            return *new SetHT(*this);
         }
         Set& fresh() const
         {
            return *new SetHT();
          }
  };
```

"new SetHT(...)" returns a "SetHT*", so "*new" returns a SetHT&. A SetHT is-a Set, so the return value is correct. The invocation of "SetHT(*this)" is that of copy construction ("*this" has type "SetHT&"). Although "clone()" returns a new SetHT, the caller of clone() merely knows he has a Set, not a SetHT (which is desirable in the case of wanting a "virtual ctor"). "fresh()" is similar, but it constructs an "empty" SetHT.

Clients can use this as if they were "virtual constructors":

```
      void client_code(Set& s)
      {
        Set& s2 = s.clone();
        Set& s3 = s.fresh();
        //...
        delete &s2;   //relies on destructor being virtual!!
        delete &s3;   //
      }
```

Another Example

If the class "owns" the object pointed to by the (abstract) base class pointer, use the Virtual Constructor Idiom in the (abstract) base class. As usual with this idiom, we declare a pure virtual clone() method in the base class:

```
class Shape
{
 public:
        Shape()
   {
   }
  Shape(Shape &ref)
   {
   }
   virtual Shape* clone() const = 0;   // The Virtual (Copy) Constructor
};
```

Then we implement this clone() method in each derived class:

```
class Circle : public Shape
{
 public:
   Circle()
   {
   }
   Circle(Circle &ref):Shape(ref)
   {
```

```
  }
  virtual Shape* clone() const
  { return new Circle(*this); }
};

class Square : public Shape
{
 public:
  Square()
  {
  }
  Square(Square &ref):Shape(ref)
  {
  }
  virtual Shape* clone() const
  { return new Square(*this); }
};
```

Now suppose that each Fred object "has-a" Shape object. Naturally the Fred object doesn't know whether the Shape is Circle or a Square or ... Fred's copy constructor and assignment operator will invoke Shape's clone() method to copy the object:

```
class Fred
{
 public:
  Fred(Shape* p)
  {
        p_=p;
  }
  ~Fred() { delete p_; }
  Fred(const Fred& f) : p_(f.p_->clone())
  {
  }
  Fred& operator= (const Fred& f)
    {
     if (this != &f)
     {            // Check for self-assignment
        Shape* p2 = f.p_->clone();   // Create the new one FIRST...
        delete p_;                   // ...THEN delete the old one
        p_ = p2;
     }
     return *this;
    }
 private:
  Shape* p_;
};


void main()
{
        Fred f(new Circle);
        Fred f1(f);
        Fred f2(new Square);
        Fred f3(new Square);

        f1=f3;
        Fred f4=f;
}
```