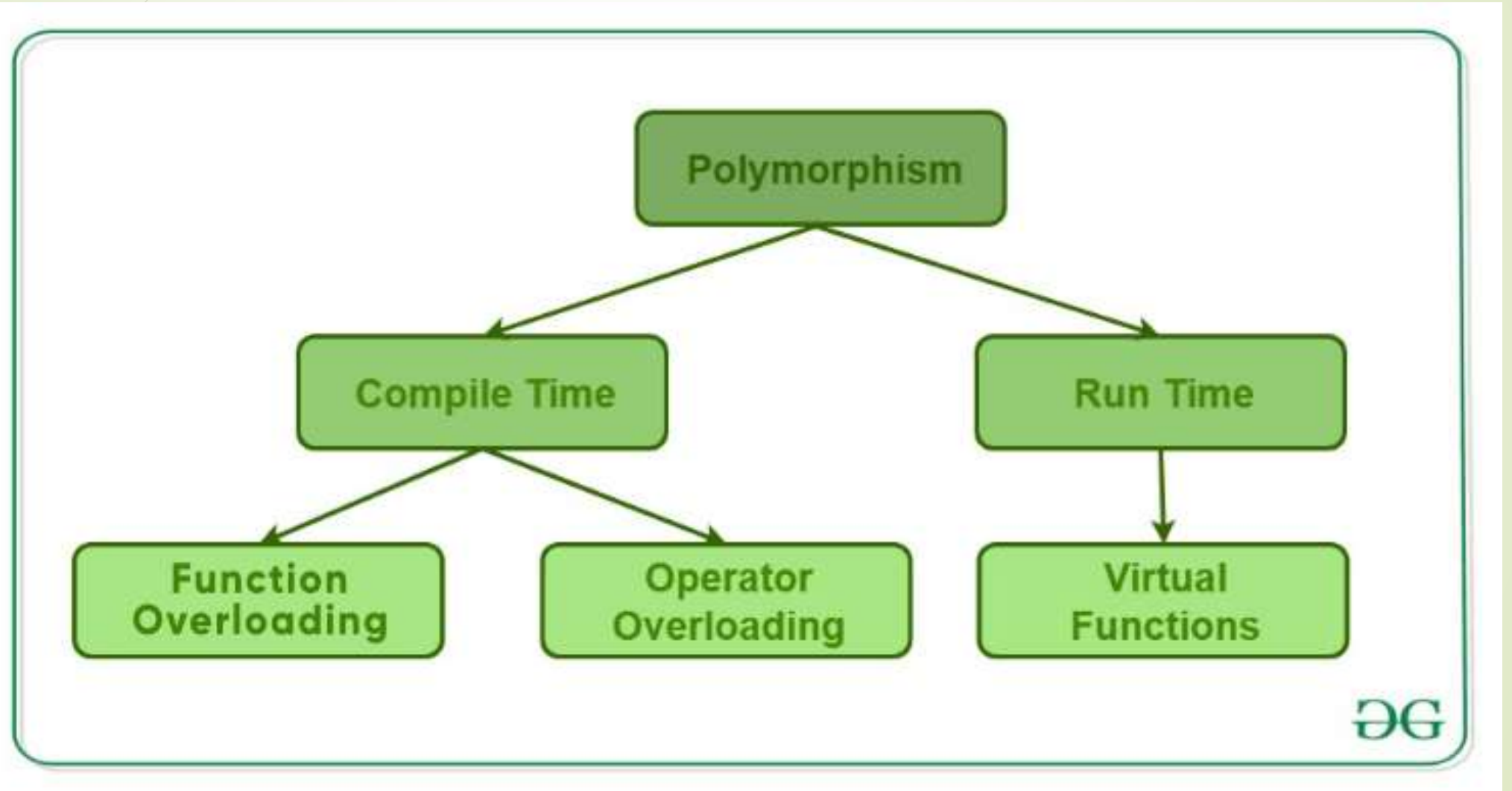




Polymorphism

- ▶ polymorphism refers to existence of **different forms** of a single entity. For example, both Diamond and Coal are different forms of Carbon.
- ▶ The word polymorphism means having many forms.
- ▶ A real-life example of polymorphism, a person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person possesses different behavior in different situations

Polymorphism



Compile time polymorphism:


This type of polymorphism is achieved by function overloading or operator overloading

Function Overloading: When there are multiple functions with same name but different parameters then these functions are said to be **overloaded**. Functions can be overloaded by **change in number of arguments** or/and **change in type of arguments**.

```
using namespace std;  
class Geeks
```

```
{  
    public:  
  
    // function with 1 int parameter  
    void func(int x)  
    {  
        cout << "value of x is " << x << endl;  
    }  
  
    // function with same name but 1 double  
    parameter  
    void func(double x)  
    {  
        cout << "value of x is " << x << endl;  
    }  
  
    // function with same name and 2 int parameters  
    void func(int x, int y)  
    {  
        cout << "value of x and y is " << x << ", " << y <<  
endl;  
    }  
};
```


```
int main() {  
  
    Geeks obj1;  
  
    // Which function is called will depend on  
    the parameters passed  
    // The first 'func' is called  
    obj1.func(7);  
  
    // The second 'func' is called  
    obj1.func(9.132);  
  
    // The third 'func' is called  
    obj1.func(85,64);  
    return 0;  
}
```



Write a program that reads a group of numbers from the user and places them in an array of type `float`. Once the numbers are stored in the array, the program should average them and print the result. Use pointer notation wherever possible.

Start with the `String` class from the `Str` example. Add a member function called `upit()` that converts the string to all uppercase. You can use the `toupper()` library function, which takes a single character as an argument and returns a character that has been converted (if necessary) to uppercase. This function uses the `CCTYPE` header file. Write some code in `main()` to test `upit()`.

Suppose you have a `main()` with three local arrays, all the same size and type (say `float`). The first two are already initialized to values. Write a function called `addarrays()` that accepts the addresses of the three arrays as arguments; adds the contents of the first two arrays together, element by element; and places the results in the third array before returning. A fourth argument to this function can carry the size of the arrays. Use pointer notation throughout; the only place you need brackets is in defining the arrays.

- 
- Make your own version of the library function `strcmp(s1, s2)`, which compares two strings and returns `-1` if `s1` comes first alphabetically, `0` if `s1` and `s2` are the same, and `1` if `s2` comes first alphabetically. Call your function `compstr()`. It should take two `char*` strings as arguments, compare them character by character, and return an `int`. Write a `main()` program to test the function with different combinations of strings. Use pointer notation throughout.



Virtual Function

- A **virtual** function is a *member* function which is declared in the *base* class using the keyword `virtual` and is re-defined (Overriden) by the *derived* class.
- The term **Polymorphism** means the ability to take many forms. It occurs if there is a hierarchy of classes which are all related to each other by *inheritance*.
- When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.
- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- Functions are declared with a **virtual** keyword in base class.
- The resolving of function call is done at Run-time.



Rules for Virtual Functions

- Virtual functions cannot be static and also cannot be a friend function of another class.
- Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism
- The prototype of virtual functions should be same in base as well as derived class.
- They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override (or re-define the virtual function), in that case base class version of function is used.


```

class Shape
{
public:
    Shape(int l=0, int w=0)
    {
        length = l;
        width = w;
    } // default constructor
    virtual int get_Area()
    {
        cout << "This is call to parent class area" << endl;
        return 0;
    }
protected:
    int length, width;
};
// Derived class
class Square : public Shape
{
public:
    Square(int l = 0, int w = 0): Shape(l, w)
    {
    } // declaring and initializing derived class
    // constructor
    int get_Area()
    {
        cout << "Square area: " << length * width << endl;
        return (length * width);
    }
};

```

```

// Derived class
class Rectangle : public Shape
{
public:
    Rectangle(int l = 0, int w = 0): Shape(l, w)
    {
    } // declaring and initializing derived class
    // constructor
    int get_Area()
    {
        cout << "Rectangle area: " << length * width <<
endl;
        return (length * width);
    }
};
int main(void)
{
    Shape* s;
    Square sq(5, 5); // making object of child class Sqaure
    Rectangle rec(4, 5); // making object of child class
    Rectangle

    s = &sq;
    s->get_Area();
    s = &rec;
    s->get_Area();

    return 0;
}

```



In the above function:


- we store the address of each child class **Rectangle** and **Square** object in **s** and
- then we call the **get_Area()** function on it,
- ideally, it should have called the respective **get_Area()** functions of the child classes but
- instead it calls the **get_Area()** defined in the base class.
- This happens due static linkage which means the call to **get_Area()** is getting set only once by the compiler which is in the base class.

What is the use?

Virtual functions allow us to create a list of base class pointers and call methods of any of the derived classes without even knowing kind of derived class object.

Consider an employee management software for an organization. Let the code has a simple base class *Employee* , the class contains virtual functions like *raiseSalary()*, *transfer()*, *promote()*, etc. Different types of employees like *Manager*, *Engineer*, etc. may have their own implementations of the virtual functions present in base class *Employee*. In our complete software, we just need to pass a list of employees everywhere and call appropriate functions without even knowing the type of employee. For example, we can easily raise the salary of all employees by iterating through the list of employees. Every type of employee may have its own logic in its class, but we don't need to worry about them because if *raiseSalary()* is present for a specific employee type, only that function would be called.

- 
- 
- **Explanation:** Runtime polymorphism is achieved only through a pointer (or reference) of base class type. Also, a base class pointer can point to the objects of base class as well as to the objects of derived class. In above code, base class pointer 'bptr' contains the address of object 'd' of derived class.



```
#include <iostream>
using namespace std;
```

```
class base {
public:
    void fun_1() { cout << "base-1\n"; }
    virtual void fun_2() { cout << "base-2\n"; }
    virtual void fun_3() { cout << "base-3\n"; }
    virtual void fun_4() { cout << "base-4\n"; }
};

class derived : public base {
public:
    void fun_1() { cout << "derived-1\n"; }
    void fun_2() { cout << "derived-2\n"; }
    void fun_4(int x) { cout << "derived-4\n"; }
};
```

Output:

```
base-1
derived-2
base-3
base-4
```

```
int main()
{
    base* p;
    derived obj1;
    p = &obj1;

    // Early binding because fun1 () is non-virtual
    // in base
    p->fun_1();

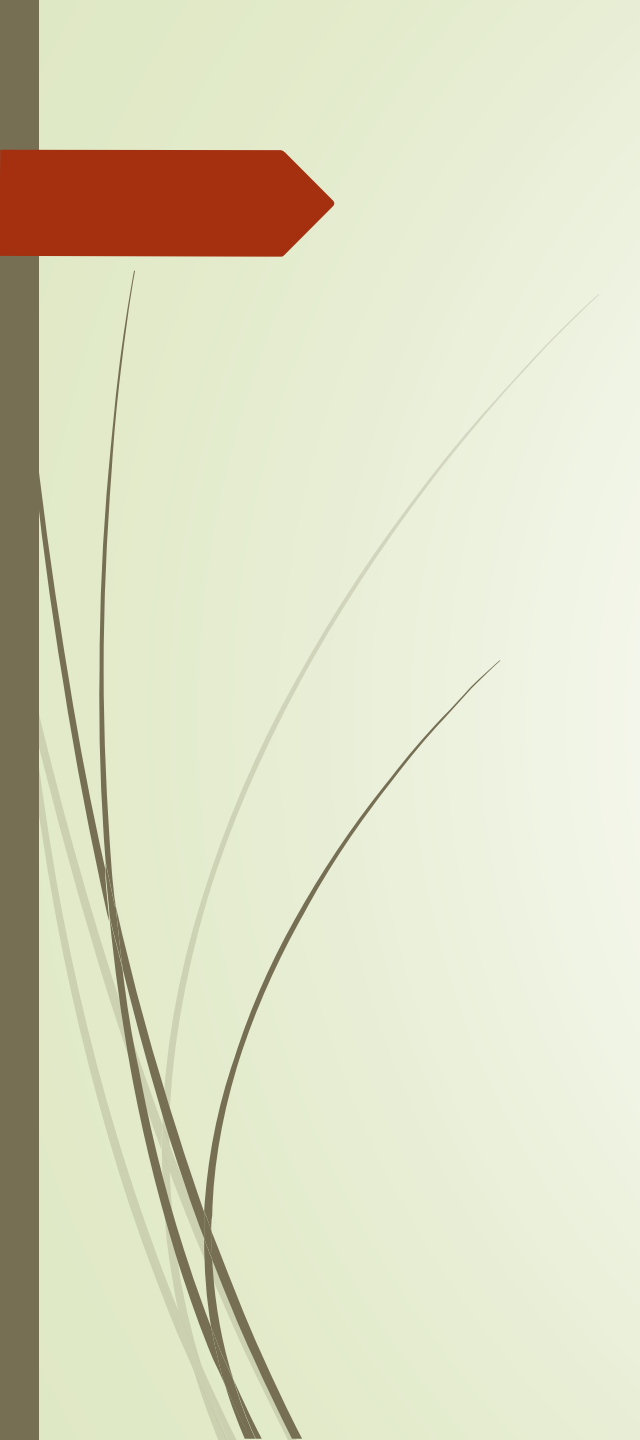
    // Late binding (RTP)
    p->fun_2();

    // Late binding (RTP)
    p->fun_3();

    // Late binding (RTP)
    p->fun_4();

    // Early binding but this function call is
    // illegal(produces error) because pointer
    // is of base type and function is of
    // derived class
    // p->fun_4(5);
}
```

NOTE: fun_4(int) in derived class is different from virtual function fun_4() in base class as prototype of both the function is different

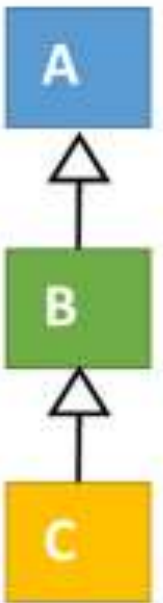


```
#include <iostream>
using namespace std;
class Base
{
public:
    ~Base() //non-virtual destructor
    //virtual ~Base() //virtual destructor
    {
        cout << "Base destroyed\n";
    }
};
////////////////////////////////////
class Derv : public Base
{
public:
    ~Derv()
    {
        cout << "Derv destroyed\n";
    }
};
int main()
{
    Base* pBase = new Derv;
    delete pBase;
    return 0;
}
```


Runtime Binding

```
class A{
    int a;
public:
    A(int a=0){ this->a=a;}
    virtual void print(){ cout<<a;}
};
```

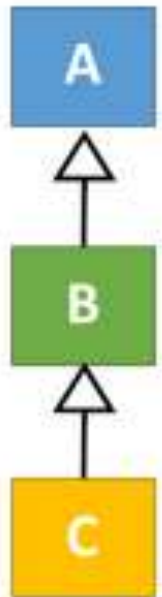
```
class B: public A{
    int b;
public:
    B(int a=0, int b=0):A(a)
    { this->b = b;}
    //override print function inherited from A
    virtual void print(){
        A::print();
        cout<<b;
    }
    //overload print function inherited from A
    void print(int x){ cout<<x+b; }
    void funb() { cout<<"funb"<<endl};
};
```



Runtime Binding...

```
class C: public B{
    int c;
public:
    C(int a=0, int b=0, int c=0) :B(a,b)
    { this->c = c;}
    //override print function inherited from B
    virtual void print() {
        B::print();
        cout<<c;
    }
    //overload print function inherited from B
    void print(int x, int y){
        cout<<x+y+c;
    }

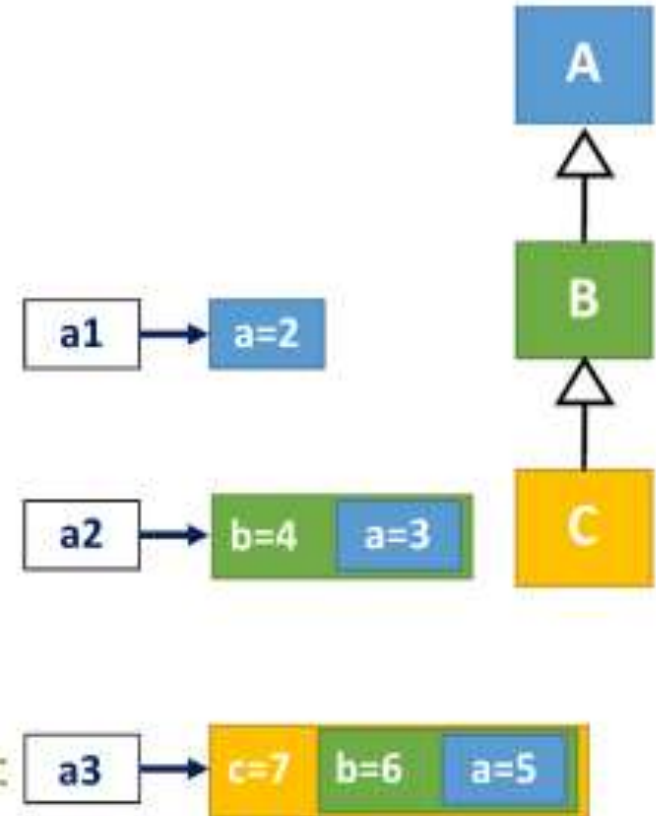
    void func(){ cout<< "func" <<endl; }
};
```



Runtime Binding...

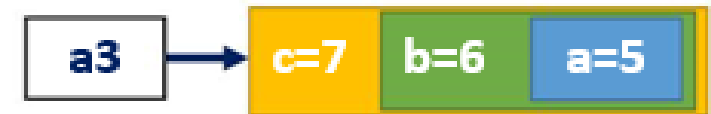
- Call the function according to the type of object not pointer.

```
void main(){  
    A * a1 = new A(2); //A's pointer to A's object  
    a1->print(); //A's print called.  
  
    A * a2 = new B(3, 4); //A's pointer to B's object  
    a2->print(); //B's print called  
    a2->funb(); a2->print(3); //Compile time Error  
  
    A * a3 = new C(5, 6, 7); //A's pointer to C's object  
    a3->print(); //C's print called  
    a3->funb(); a3->print(3); //Compile time Error  
    a3->func(); a3->print(3,8); //Compile time Error  
}
```



Virtual Destructor

```
void main(){  
    A * a1 = new A(2); //A's pointer to A's object  
    a1->print(); //A's print called.  
  
    A * a2 = new B(3, 4); //A's pointer to B's object  
    a2->print(); //B's print called  
  
    A * a3 = new C(5, 6, 7); //A's pointer to C's object  
    a3->print(); //C's print called  
  
    delete a1; //A's destructor called  
    delete a2; //A's destructor called, should call b's  
    delete a3; //A's destructor called, should call c's  
}
```



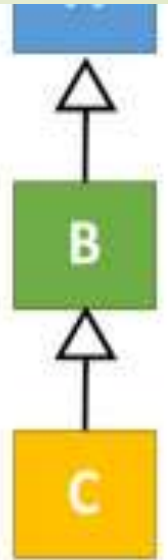
Virtual Destructor

```
class A{
    int a;
public:
    A(int a=0){ this->a=a;}
    virtual void print(){ cout<<a;}
    virtual ~A(){}
};

class B: public A{
    int b;
public:
    B(int a=0, int b=0):A(a)
    { this->b = b;}
    void print(){
        A::print();
        cout<<b;
    }
    virtual ~B(){}
};
```

```
class C: public B{
    int c;
public:
    C(int a=0, int b=0, int c=0) :B(a,b)
    { this->c = c;}
    //override print function inherited from B
    void print(){
        B::print();
        cout<<c;
    }
    virtual ~C(){}
};
```

- Make destructor virtual too.
- Destructor is not inherited so make it virtual in all classes.



Virtual Destructor

- Destructor should be called according to object type.

```
void main(){
```

```
    A * a1 = new A(2); //A's pointer to A's object
```

```
    a1->print(); //A's print called.
```



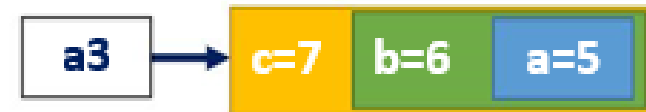
```
    A * a2 = new B(3, 4); //A's pointer to B's object
```

```
    a2->print(); //B's print called
```



```
    A * a3 = new C(5, 6, 7); //A's pointer to C's object
```

```
    a3->print(); //C's print called
```



```
    delete a1; //A's destructor called
```

```
    delete a2; //B's destructor called, which also calls A's
```

```
    delete a3; //C's destructor called, which also calls B's then A's
```

```
}
```

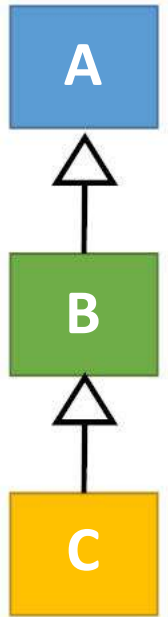
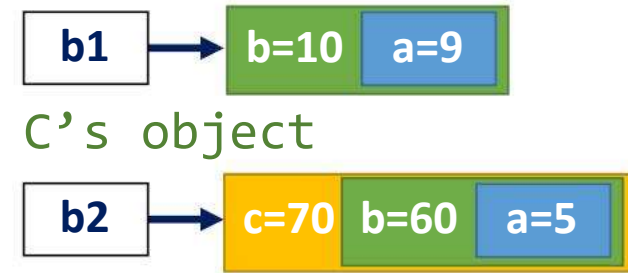
Virtual Destructor

```
void main(){
    B * b1 = new B(9, 10); //B's pointer to B's object
    b1->print(); //B's print called.

    B * b2 = new C(5, 60, 70); //B's pointer to C's object
    b1->print(); //C's print called.

    B * b3 = new A(2); //Error: B's pointer to A's object
    //Every derived is a base but every base is not a derived.
    //Allowed if explicit cast made

    delete b1; //B's destructor called, which also calls A's
    delete b2; //C's destructor called, which also calls B's then A's
}
```





virtual table and vptr

The **virtual table** is a lookup table of functions used to resolve function calls in a dynamic/late binding manner.

First, every class that uses virtual functions (or is derived from a class that uses virtual functions) is given its own virtual table. This table is simply a static array that the compiler sets up at compile time. A virtual table contains one entry for each virtual function that can be called by objects of the class. Each entry in this table is simply a function pointer that points to function accessible by that class.

Second, the compiler also adds a hidden pointer that is a member of the base class, which we will call `*__vptr`. `*__vptr` is set (automatically) when a class instance is created so that it points to the virtual table for that class.



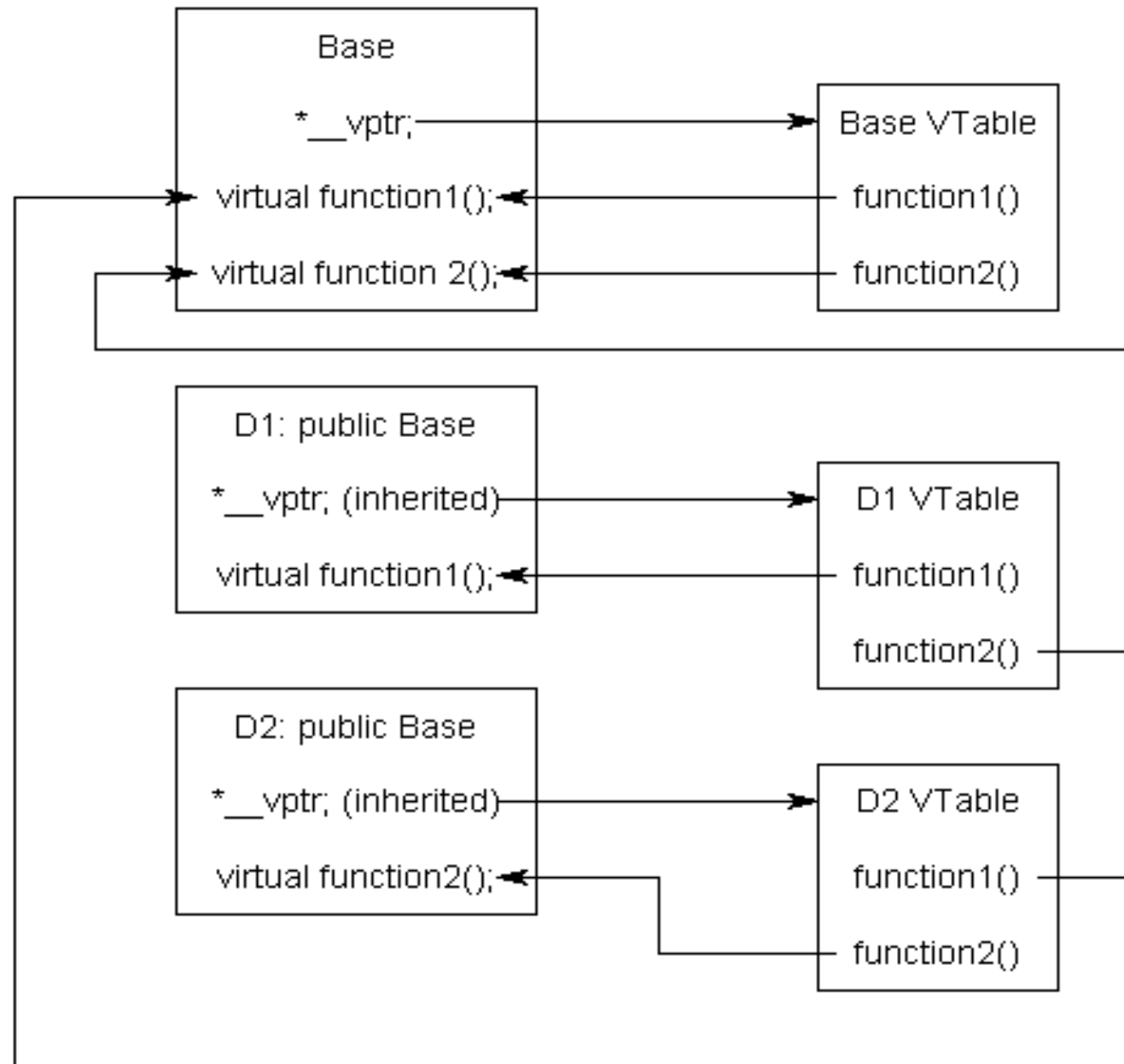
virtual table and vptr

```
class Base
{
public:
    VirtualTable* __vptr;
    virtual void function1 () {};
    virtual void function2() {};
};
```

```
class D1: public Base
{
public:
    virtual void function1 () {};
};
```

```
class D2: public Base
{
public:
    virtual void function2() {};
};
```

virtual table and vptr





Upcasting

Simple assigning base class pointers to the child object or base class as an alias to child class object.

Downcasting

It manually cast the base class's object to the derived class's object, so we must specify the explicit typecast.



Downcasting

It manually cast the base class's object to the derived class's object, so we must specify the explicit typecast

```
class Parent
{
    public:
        void base()
        {
            cout << " It is the function of the Parent class
" << endl;
        }
};

class Child : public Parent
{
    public:
        void derive()
        {
            cout << " it is the function of the Child class "
<< endl;
        }
};
```

```
int main ()
{
    Parent pobj; // create Parent's object
    Child *cobj; // create Child's object

    // explicit type cast is required in
    // downcasting
    cobj = (Child *) &pobj;
    cobj -> derive();

    return 0;
}
```



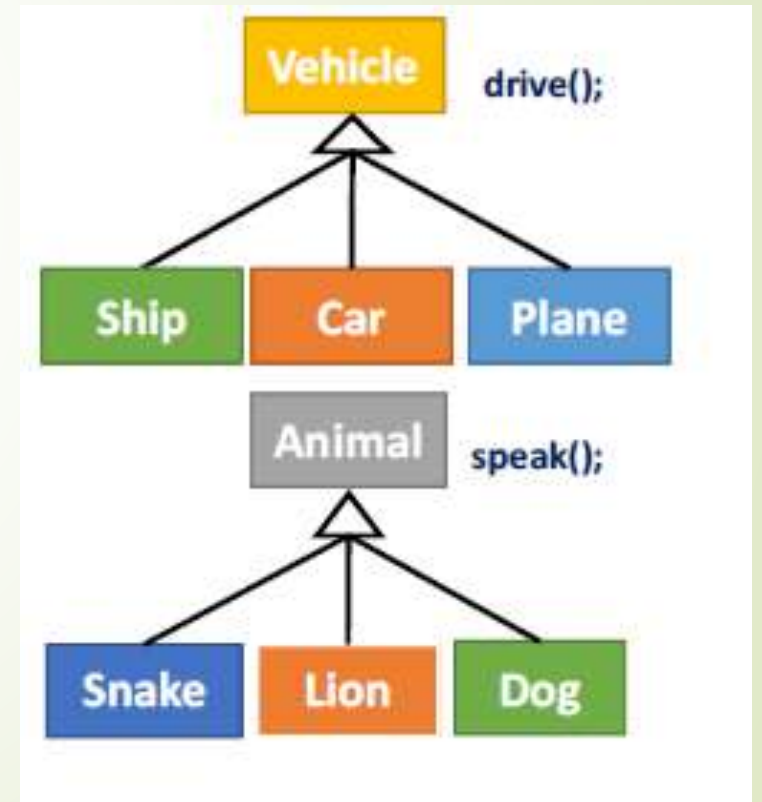
Classes we have worked on till now are concrete classes.



Abstract classes

Abstract class is a concept not a real object,

```
Class Shape{
    Point p;
    public:
        Shape(int x=0,int y=0):p(x,y){}
        virtual void draw(){//What to draw?}
};
Class Vehicle{
    public:
        virtual void drive(){// What to drive?}
};
Class Animal{
    public:
        Virtual void speak(){// What to speak?}
};
```



Abstract classes

Think about a playing card. It doesn't matter what game we're playing, all cards have a few things that are the same. The suit & the face. But, cards also have some sort of the value to the game they're being used in. But for that, you need to know the game.

```
class Card {  
    protected:  
        int face;  
        String suit;  
  
    public :  
        Card() { /* Implementation not shown */ }  
        int getFace() { return face; }  
        String getSuit() { return suit; }  
  
        virtual int getValue()=0;  
}
```

```
class BlackJackCard : public Card {  
    public: int getValue() {  
        return getFace() >= 10 ? 10 : getFace();  
    }  
}
```


```
class CrazyEightsCard : Public Card {  
    public: int getValue() {  
        // I haven't played crazy eights in years and have  
        // no idea what would really go here :)  
    }  
}
```

Abstract classes

- Sole purpose of abstract class is to be a base class
 - Place common attributes and functions in abstract base class.
- Implementation is Incomplete for some functions
 - Derived classes should fill in "missing pieces" according to their type.
- Cannot create objects (dynamic or static) of abstract class
 - Compiler will show error message, if create object of abstract class.
 - However, can create pointers and references of abstract classes.
 - Abstract class pointers are still useful for polymorphism.
 - We can still make pointer arrays of abstract base classes to store different type of derived objects.

Abstract classes

- How to make a class abstract?
 - Add one or more "pure" virtual functions in the class.
 - Declare function with initializer of 0
`virtual void draw() = 0;`
 - **Pure virtual functions**, have no implementation, just add prototype in base class.
 - Must be overridden in derived classes according to their type.
 - If a derive class do not override a pure virtual function, it also becomes an abstract class.
 - Normal virtual functions have implementations, overriding is optional
 - Abstract classes can still have data and concrete functions
 - Just required to have one or more pure virtual functions
 - Must add virtual destructor in abstract class.
 - Abstract class can has constructors



```
class Shape{
```

```
    Int x;
```

```
    Int y;
```

```
    public:
```

```
        Shape(int x=0, int y=0) :p(x,y){}
```

```
        virtual void draw() = 0; // Pure virtual function
```

```
};
```

```
class Vehicle{
```

```
    public:
```

```
        virtual void drive() = 0; // Pure virtual function
```

```
};
```

```
class Animal{
```

```
    public:
```

```
        virtual void speak() = 0; // Pure virtual function
```

```
};
```



```
void main(){
```

```
    Shape * s = new Shape(3, 2); //Error: cannot create object of abstract class.
```

```
    // Can still point to derived class objects.
```

```
    Shape * s2 = new Circle(3, 2, 5.5);
```

```
    s2->draw(); // Draw a circle
```

```
    Shape * s3 = new Rectangle(3, 2, 5.5, 6);
```

```
    s3->draw(); // Draw a Rectangle
```

```
    Animal * a = new Animal; //Error
```

```
    Animal * a2 = new Lion;
```

```
    a2->speak(); // Lion Roars
```

```
    Vehicle * v = new Vehicle; //Error
```

```
}
```



(is-a) Identifier **override**

- A derived class can override, inherited virtual functions.
 - but the return type, name and parameters should same.
- If by mistake the programmer change return type, name or parameters the program may generate logical errors.
 - To avoid this issue the identifier **override** is added at end of virtual overridden function header.
 - Compiler will generate an error message, if function is not properly overridden in derived class.
- Programmer can visualize the overridden virtual functions directly by looking at derived class implementation.

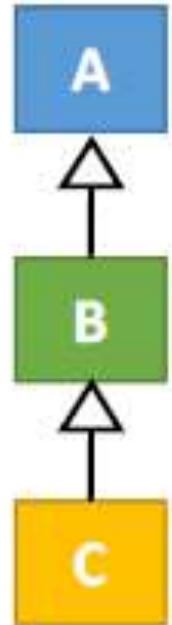
(is-a) Identifier **override**

```
class A{
    int a;
public:
    A(int a=0){ this->a=a;}
    virtual void print(){ cout<<a;}
    virtual ~A(){}
};

class B: public A{
    int b;
public:
    B(int a=0, int b=0):A(a)
    { this->b = b;}
    void print() override{
        A::print();
        cout<<b;
    }
    virtual ~B(){}
};
```

```
class C: public B{
    int c;
public:
    C(int a=0, int b=0, int c=0) :B(a,b)
    { this->c = c;}

    // Compile Time Error: change return type
    int print() override{
        B::print();
        cout<<c;
    }
    virtual ~C(){}
};
```



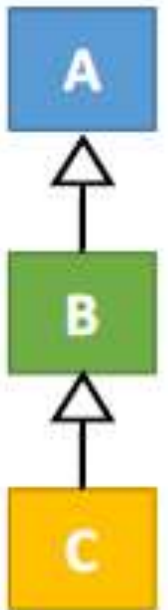
(is-a) Identifier **override**

```
class A{
    int a;
public:
    A(int a=0){ this->a=a;}
    virtual void print(){ cout<<a;}
    virtual ~A(){}
};

class B: public A{
    int b;
public:
    B(int a=0, int b=0):A(a)
    { this->b = b;}
    void print() override{
        A::print();
        cout<<b;
    }
    virtual ~B(){}
};
```

```
class C: public B{
    int c;
public:
    C(int a=0, int b=0, int c=0) :B(a,b)
    { this->c = c;}

    // Compile Time Error: Not override
    void print(int x) override{
        B::print();
        cout<<c;
    }
    virtual ~C(){}
};
```



(is-a) Identifier **final**

- We can stop a derive class to override an inherited function.
 - Add final keyword at end of the function header.
 - Compiler will generate an error and will not allow to override a final function.
- We can stop inheritance of a class.
 - Define the class as final
 - Compiler will generate an error and will not allow to derive a class from final class.

```
class A{
    int a;
public:
    A(int a=0){ this->a=a;}
    virtual void print(){ cout<<a;}
    virtual ~A(){}
};

class B: public A{
    int b;
public:
    B(int a=0, int b=0):A(a)
    { this->b = b;}
    void print() override final{
        A::print();
        cout<<b;
    }
    virtual ~B(){}
};
```

```
class C: public B{
    int c;
public:
    C(int a=0, int b=0, int c=0) :B(a,b)
    { this->c = c;}

    // Compile Time Error: Cannot override print
    // function inherited from class B as declared
    // final in class B

    void print(){
        B::print();
        cout<<c;
    }
    virtual ~C(){}
};
```

(is-a) Identifier **final** Class

```
class A final{
    int a;
public:
    A(int a=0){ this->a=a;}
    virtual void print(){
        cout<<a;
    }
    virtual ~A(){}
};
```

// Compile Time Error: Cannot
derive from final class A

```
class B: public A{
    int b;
public:
    B(int a=0, int b=0):A(a)
    { this->b = b;}
    void print() override {
        A::print();
        cout<<b;
    }
    virtual ~B(){}
};
```




(is-a) Down casting Pointers

- Down casting converts base class pointer to derived class pointer,
 - if base class is pointing to derived class object.
 - `dynamic_cast` operator is used for down casting pointers
 - Determine object's type at runtime
 - Returns 0 or Null, if not of proper type (cannot be cast)
 - `dynamic_cast` will not work
 - With protected and private inheritance
 - With classes, which not have any virtual functions.
- Down casting is helpful
 - For accessing explicitly derived class data and functions that does not exist in base class.

```
class A{
    int a;
public:
    A(int a=0){ this->a=a;}
    virtual void print(){ cout<<a;}
    virtual ~A(){}
};

class B: public A{
    int b;
public:
    B(int a=0, int b=0):A(a)
    { this->b = b;}
    void print() override{
        A::print();
        cout<<b;
    }
    virtual ~B(){}
};
```

```
class C: public B{
    int c;
public:
    C(int a=0, int b=0, int c=0) :B(a,b)
    { this->c = c;}
    void print() override{
        B::print();
        cout<<c;
    }
    virtual ~C(){}
};
```

```
void main(){  
    A * a1 = new A(2); //A's pointer to A's object  
    a1->print(); //A's print called.
```

```
    B *ptr = dynamic_cast<B*>(a1);  
    if (ptr != NULL) //return null when failed  
        ptr->print();
```

```
    // Type Casting failed as A's pointer is pointing  
    // to A's object
```

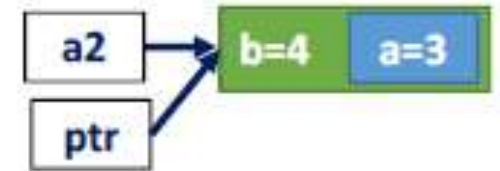
```
    // Through Null check we can avoid run time error
```

```
}
```



Downcasting Pointers

```
void main(){  
    A * a2 = new B(3, 4); //A's pointer to B's object  
    a2->print(); //B's print called.  
  
    B *ptr = dynamic_cast<B*>(a2);  
    if (ptr != NULL) //return null when failed  
        ptr->print();  
  
    // Type Casting is successful because A's pointer  
    // is pointing to B's object  
    // Not create new object just perform down casting  
    // of same object for derived class pointer  
}
```



Down casting References

```
void main(){  
    A & a = B(3, 4);  
    a.print(); //B's print called.
```

```
    B b1 = dynamic_cast<B &> (a);  
    b1.print();
```

```
// Type Casting is successful because A's reference to  
B's object  
// Create new object by calling copy constructor
```

```
}
```





Interface

- A special type of class, which has no data members.
- All functions are pure virtual.
- Derived class must implement all pure virtual functions.
 - If it skip implementation of any function, then it will become an abstract class, its object cannot be created.
- Must add virtual destructor in interface class.
- Name of Interface classes usually begin with letter “I”.
 - Just to differentiate it from other classes.

Interface

```
class IExample{
public:
    // Pure virtual functions
    virtual void Function1() = 0;
    virtual void Function2() = 0;
    virtual void Function3() = 0;
    virtual ~IExample();
};

class A: public IExample{
    int a;
public:
    A(int a=0){ this->a=a;}
    virtual void print(){ cout<<a;}
    virtual ~A(){}
    // Must implement all Pure virtual functions.
};
```

Interface with diamond problem

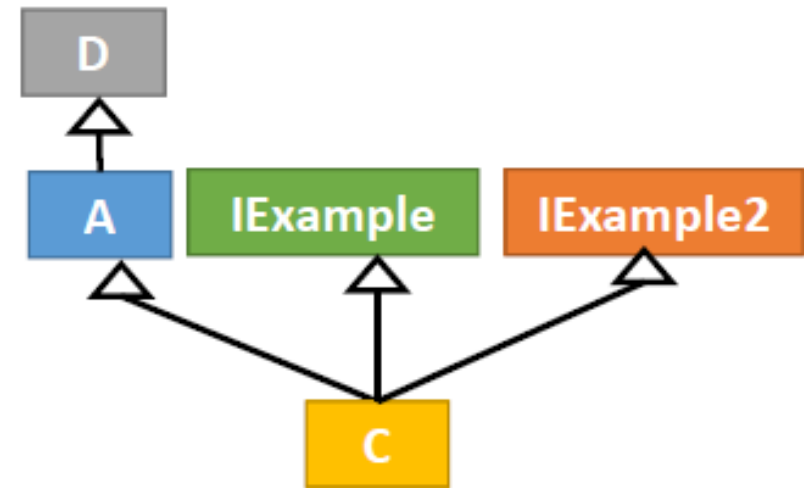
Some Object oriented languages such as java and c# does not allow multiple inheritance of concrete and abstract classes.

A derived class can inherit data from only single base class (Abstract or Concrete).


But, they allow to inherit multiple Interfaces.

- Interface classes has no data members, and have all pure virtual functions.
- Therefore issue of diamond problem is avoided completely by using them.

```
class IExample{  
public:  
    // Pure virtual functions  
    virtual void Function1() = 0;  
    virtual void Function2() = 0;  
    virtual void Function3() = 0;  
    virtual ~IExample();  
};
```



Forward declarations



```
class A;
class B;
class B {
    int x;
public:
    void getdata(int n)
    {
        x = n; }
    friend int sum(A, B);
};
class A {
    int y;
public:
    void getdata(int m)
    {
        y = m;
    }
    friend int sum(A, B);
};
int sum(A m, B n)
{
    int result;
    result = m.y + n.x;
    return result;
}
```

```
int main()
{
    B b;
    A a;
    a.getdata(5);
    b.getdata(4);
    cout << "The sum is : " << sum(a, b);
    return 0;
}
```