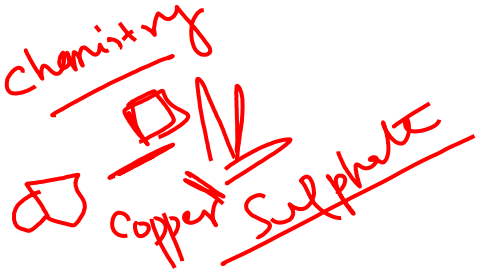
Class/Object Relationships

Polymorphism and Run Time Binding

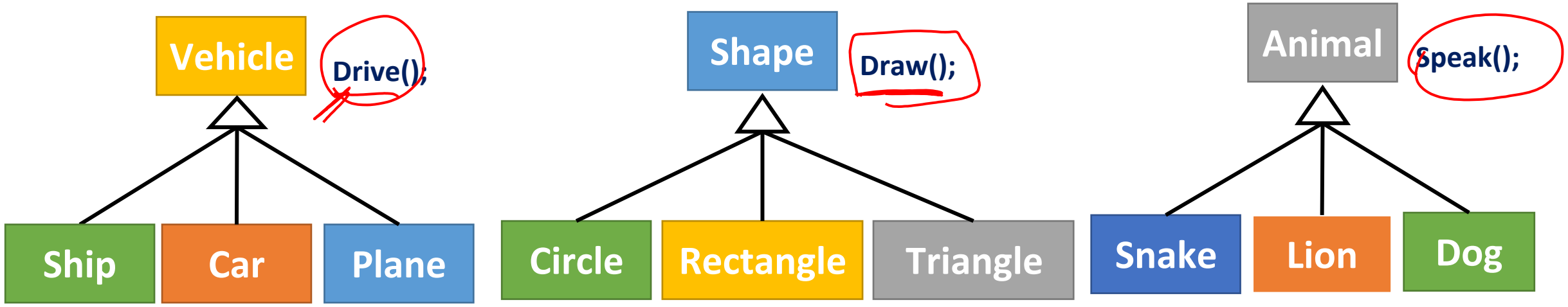
CS(217) Object Oriented Programming

Abeeda Akram

Inheritance (is-a) Polymorphism

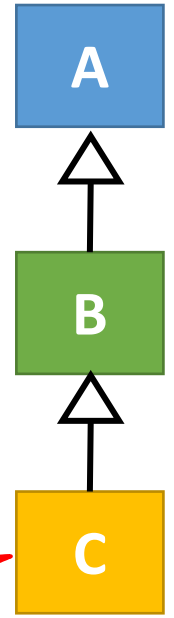


- **Poly** means "Many" **Morphism** mean "Forms"
  1. Same base type behavior will be changed according to object of derived class.
  2. Command and use objects without knowing their types explicitly.
  3. Extend the program with more functionalities through derive classes.
  4. Need one single array of base class to collect all different objects of derive.
  5. Base class represent a larger set for all objects (base and derived)

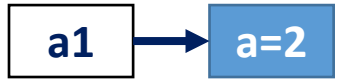


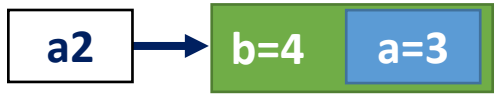
Inheritance (is-a) Polymorphism

- **One object can show different behaviors.**
  - Same message, “print”, given to many objects all through a base pointer.
  - Message takes on “many forms”

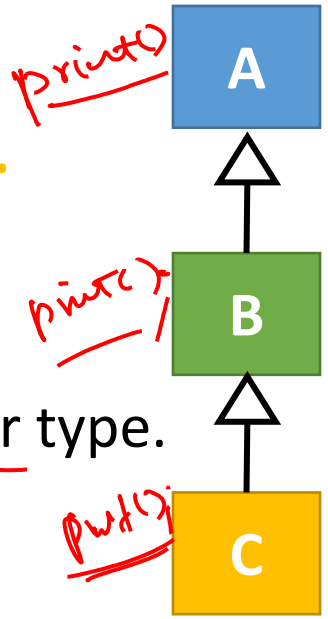



```
void main(){
   A * a1 = new A(2); //A's pointer to A's object
   a1->print(); //A's print called.

   A * a2 = new B(3, 4); //A's pointer to B's object
   a2->print(); //Should call B's print

   A * a3 = new C(5, 6, 7); //A's pointer to C's object
   a3->print(); //Should call C's print
}
```





Inheritance (is-a) Polymorphism



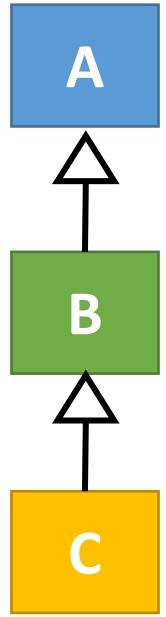
1. **Only base class inherited functions can be called through base pointer.**
2. **Override base class function in derived classes.**
3. Change Compile time binding of functions to Run time binding,
    1. **Run time binding:** Call functions according to object type not pointer type.
    2. Make functions **virtual** in base class.
        - Inherited as virtual in all derived classes, no need to make virtual again.
        - All virtual functions binding change to runtime.

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

Inheritance (is-a) Runtime Binding

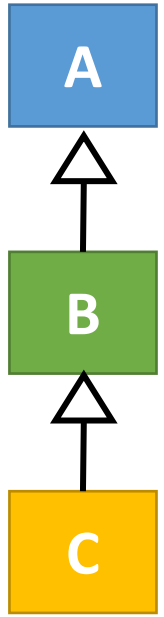
```
class A{
    int a;
public:
    A(int a=0){ this->a=a;}
    virtual void print(){ cout<<a;}
};
```

```
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() override{
        A::print();
        cout<<b;
    }
//overload print function inherited from A
    void print(int x){ cout<<x+b; }
    void funb() { cout<<"funb"<<endl};
};
```



Inheritance (is-a) Runtime Binding

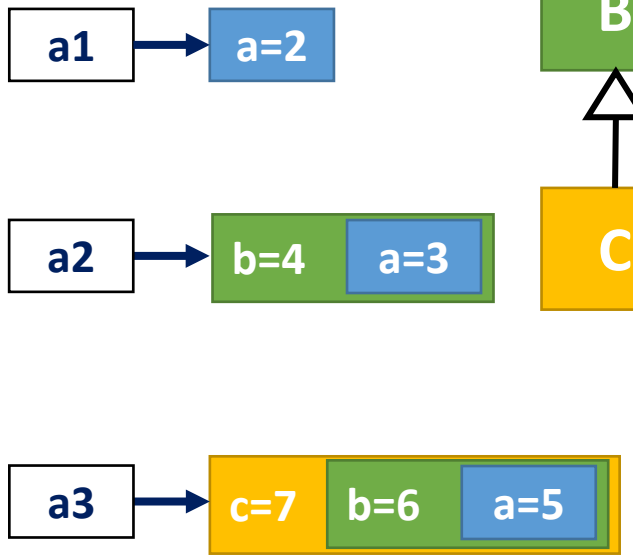
```
class C: public B{
    int c;
public:
    C(int a=0, int b=0, int c=0) :B(a,b)
    { this->c = c;}
//override print function inherited from B
 virtual void print() override{
        B::print();
        cout<<c;
    }
//overload print function inherited from B
    void print(int x, int y){
      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

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

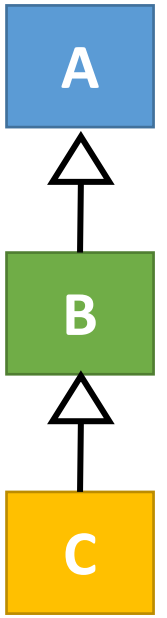


Inheritance (is-a) Runtime Binding

- 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
}
```

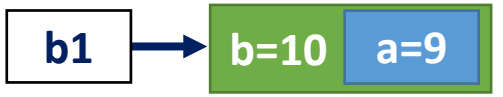


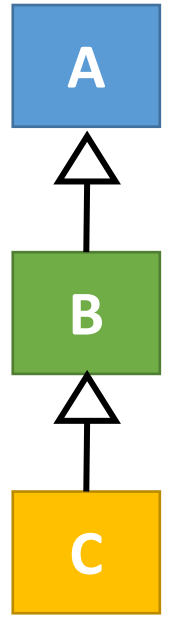


Inheritance (is-a) Runtime Binding

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

```
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
    b2->print(); //C's print called.
    b2->funb(); b2->print(3);
    b2->func(); b2->print(3,8); //Compile time Error

    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
}
```





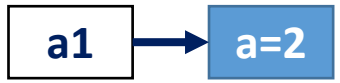
Inheritance (is-a) Runtime Binding

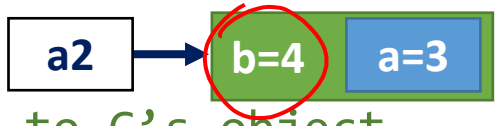
Base class static object		Call base class functions only
Derived class static object		Call derived class functions & inherited functions.
Base class static object	**Derived class static object**	Call base class functions only. **Slicing Issue** only copies base data in base object
Derived class static object	**Base class static object**	Error: Explicit cast required
Base class pointer or reference	**Base class object**	Call base class functions only
Base class pointer or reference	**Derived class object**	**Call derived class overridden functions that exist in base.**
Derived class pointer or reference	**Base class object**	Error: Explicit cast required
Derived class pointer or reference	**Derived class object**	Call derived class functions & inherited functions.

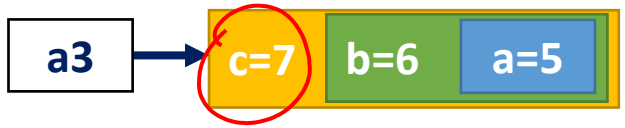
Inheritance (is-a) Virtual Destructor

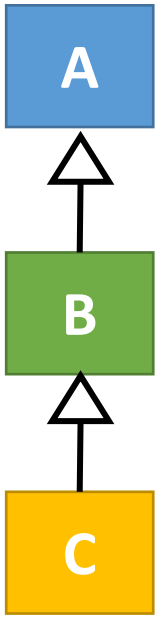
- Destructor should be called according to object type.
- Derived class dynamic members need to be deallocated.

```
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
}
```





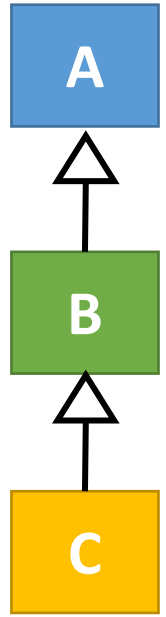




Inheritance (is-a) Virtual Destructor

```
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(){}
};
```

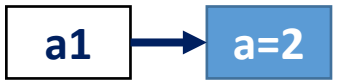


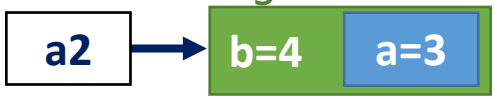
- **Make destructor virtual too.**
- **Destructor is not inherited so make is virtual in all classes.**

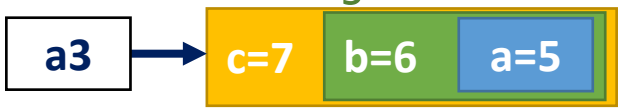
Inheritance (is-a) Virtual Destructor

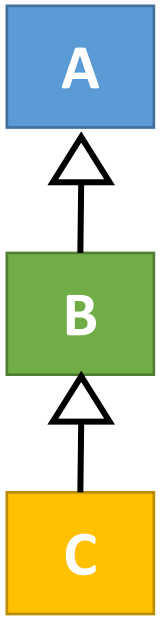
- Destructor should be called according to object type.

```
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
}
```





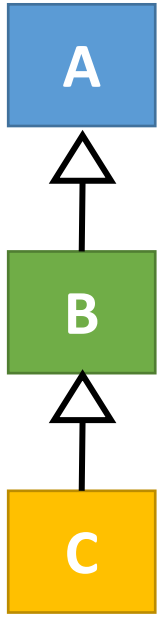




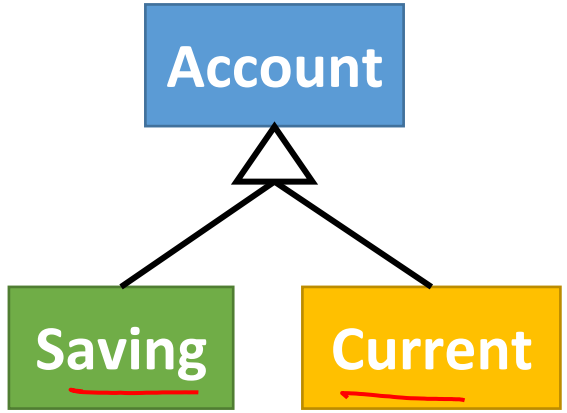
Inheritance (is-a) Virtual Destructor

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

```
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

}
```



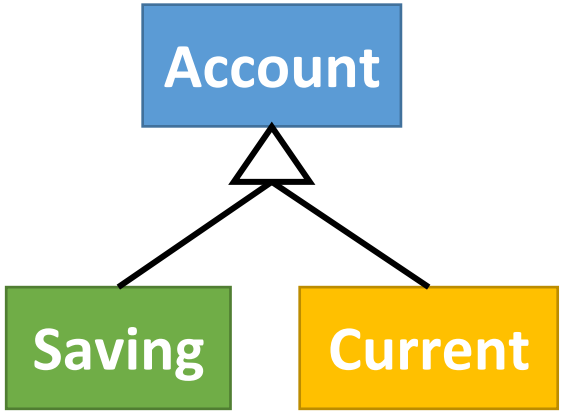
Inheritance (is-a) Polymorphism Example




```
class Account{
    int accountNo;
    float amount;
public:
//override Debit and Credit
functions according to derived
classes.
    virtual void debit(float);
    virtual void credit (float);
    virtual void print(){
        cout<< accountNo ;
        cout<< amount <<endl;
    }
    virtual ~Account(){}
};
```

```
class Current: public Account{
    float serviceCharges;
    float minBalance;
public:
    virtual void print() override{
        Account::print();
        cout<<serviceCharges;
    }
    virtual ~Current(){}
    //Use inherited Debit and Credit
};
class Saving: public Account{
    float interestRate;
public:
    virtual void print() override{
        Account::print();
        cout<<interestRate;
    }
    virtual ~Saving(){}
    //override Debit and Credit
    virtual void debit(float) override;
    virtual void credit (float) override;
};
```

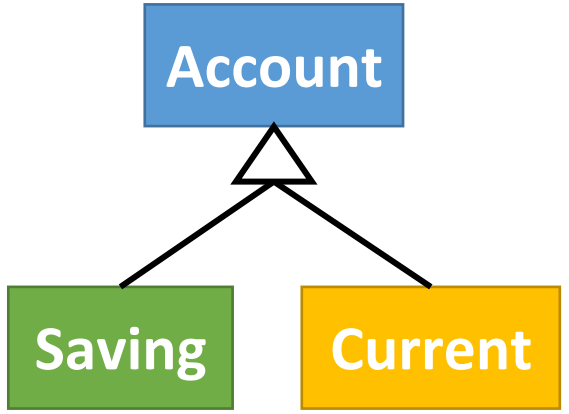
Inheritance (is-a) Polymorphism Example



- **Call the function according to the type of object not pointer.**
- Debit, credit and print function is different for different accounts.

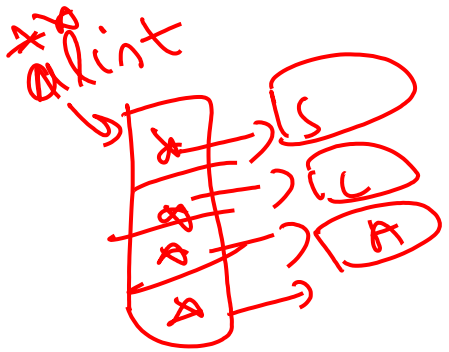
```
void main(){
    Account * a1 = new Saving; //Base pointer to derived object
    a1->print(); //Saving's print called.
    a1->debit(300); //Saving's Debit called.
    a1->credit(900); //Saving's credit called.

    Account * a2 = new Current; //Base pointer to derived object
    a2->print(); //Current's print called.
    a2->debit(500); //Current's Debit called.
    a2->credit(30085); // Current's credit called.

    delete a1; //Saving destructor called, then Account
    delete a2; //Current destructor called, then Account
}
```

Inheritance (is-a) **Polymorphism Example**

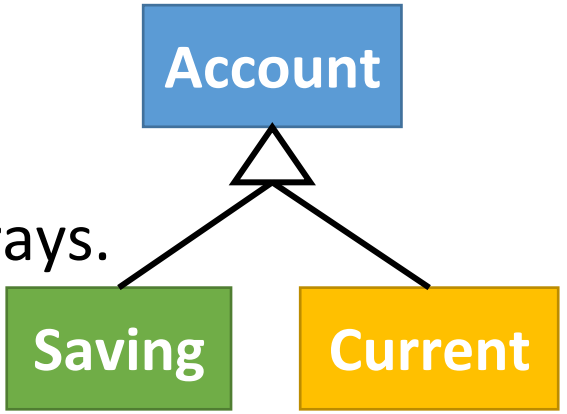


- Maintain a single array of Account instead of two separate arrays.

```
void main(){
//Array of base pointers
    Account ** alist = new Account*[10];
    alist[0] = new Saving;
    alist[1] = new Current;
    alist[2] = new Account;
    ….
//Print data of all accounts polymorphic behavior
    for(int i=0; i<10 ;i++)
        alist[i]->print();

//credit and debit polymorphic behavior
    alist[0]->credit(50);
    alist[2]->debit(333);
}
```



Inheritance (is-a) **Polymorphism Example**



- Maintain a single array of Account instead of two separate arrays.

```
//Destructors show polymorphic behavior
//Destructors are called according to object type
//Destroy all accounts
    for(int i=0; i<10 ;i++)
        delete alist[i];

//Deallocate array of pointers
    delete [] alist;
```

Inheritance (is-a) **Polymorphism Example**

- Create a payroll program for 3 types of employees, paid monthly
  - Salaried (fixed salary, no matter the hours)
  - Hourly (overtime [>40 hours] pays time and a half)
  - Commission (paid percentage of sales)

1. Each employee's pay will be calculated in different way.
2. Override calculatePay function in all employees accordingly.
3. Convert the binding to Runtime in employee class.

```
virtual float calculatePay();
```