CS 32 Week 5 Discussion 2E

Srinath

Announcements

- Homework 3 is Up!
- Due : 11:00 PM Wednesday, February 15

Outline

- Inheritance
- Polymorphism
- Destruction
- Recursion
- Worksheet 5

Inheritance

Inheritance : What is it?

Inheritance is an is-a relationship. We use inheritance only if an is-a relationship is present between the two classes.

Examples?

- A Circle is a Shape
- A Car is a Vehicle
- A Bear is an Animal

Inheritance : Declaration

Say, Circle is inherited from Shape

Shape is called the Base class, Circle is called Derived class.

Say, Car is inherited from Vehicle

Base class : Vehicle
Derived class : Car

Inheritance : Declaration

Say, Circle is inherited from Shape

Shape is called the Base class, Circle is called Derived class.

Say, Car is inherited from Vehicle

Base class : Vehicle
Derived class : Car

CODE

```
class Circle : public Shape {
        ….
}

class Car : public Vehicle {
        ….
}

Shape* shape = new Shape(..)

Circle* circle  = new Circle(..)
```

Inheritance : Construction

CODE

```
class Vehicle {
      public:
         Vehicle(string name){
               m_owner=name;
          }
         string m_vehicleNo;
         string getOwner(){return m_owner;}
      private:
         string m_owner;
}

class Car : public Vehicle {
      public:
         Car(string name, string model);
         string m_company;
         string getModel(){return m_model;}
      private:
         string m_model;
}
```

Memory Structure

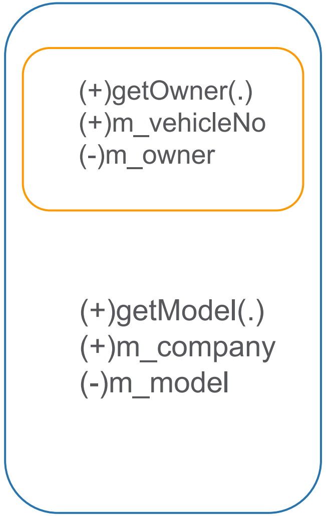
Inheritance : Construction

CODE

```
class Vehicle {
        public:
           Vehicle(string name){
                 m_owner=name;
            }
           string m_vehicleNo;
           string getOwner(){return m_owner;}
        private:
           string m_owner;
}

class Car : public Vehicle {
        public:
           Car(string name, string model);
           string m_company;
           string getModel(){return m_model;}
        private:
           string m_model;
}
```

Memory Structure



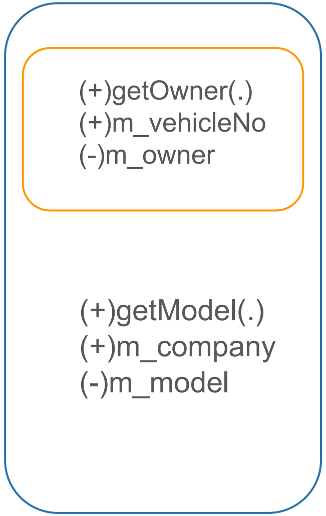
Inheritance : Construction

CODE

```
class Vehicle {
        public:
           Vehicle(string name){
                 m_owner=name;
            }
           string m_vehicleNo;
           string getOwner(){return m_owner;}
        private:
           string m_owner;
}

class Car : public Vehicle {
        public:
           Car(string name, string model);
           string m_company;
           string getModel(){return m_model;}
        private:
           string m_model;
}
```

Memory Structure



Constructing derived classes?

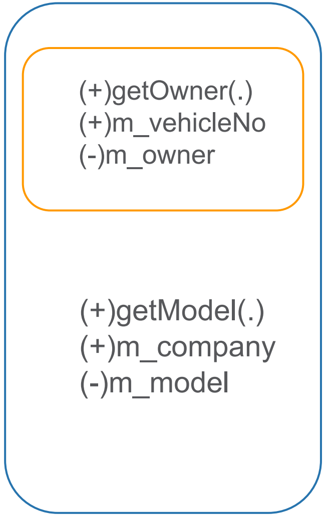
Inheritance : Construction

CODE

```
class Vehicle {
        public:
           Vehicle(string name){
                 m_owner=name;
            }
           string m_vehicleNo;
           string getOwner(){return m_owner;}
        private:
           string m_owner;
}

class Car : public Vehicle {
        public:
           Car(string name, string model);
           string m_company;
           string getModel(){return m_model;}
        private:
           string m_model;
}
```

Memory Structure



(+)getOwner(.)
(+)m_vehicleNo
(-)m_owner

(+)getModel(.)
(+)m_company
(-)m_model

Constructing derived classes?

Review: Order of Construction

1. …..?
2. Initialise data members…
3. Parse the body…

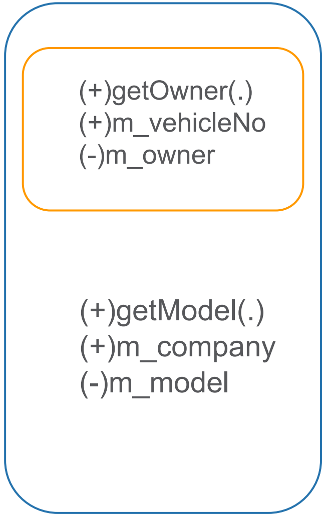
Inheritance : Construction

CODE

```
class Vehicle {
        public:
           Vehicle(string name){
                 m_owner=name;
            }
           string m_vehicleNo;
           string getOwner(){return m_owner;}
        private:
           string m_owner;
}

class Car : public Vehicle {
        public:
           Car(string name, string model);
           string m_company;
           string getModel(){return m_model;}
        private:
           string m_model;
}
```

Memory Structure



Constructing derived classes?

Review: Order of Construction

1. Construct the Base part
2. Initialise data members…
3. Parse the body…

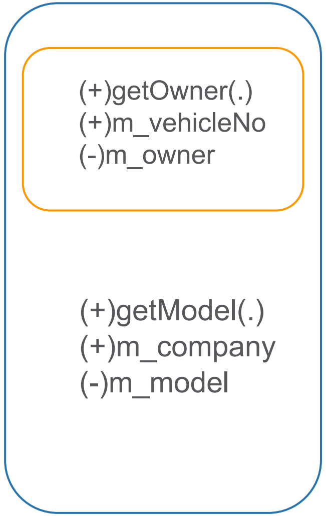
Inheritance : Construction

CODE

```
class Vehicle {
        public:
           Vehicle(string name){
                 m_owner=name;
            }
           string m_vehicleNo;
           string getOwner(){return m_owner;}
        private:
           string m_owner;
}

class Car : public Vehicle {
        public:
           Car(string name, string model);
           string m_company;
           string getModel(){return m_model;}
        private:
           string m_model;
}
```

Memory Structure



(+)getOwner(.)
(+)m_vehicleNo
(-)m_owner

(+)getModel(.)
(+)m_company
(-)m_model

Constructing derived classes?

Review: Order of Construction

1. Construct the Base part
2. Initialise data members…
3. Parse the body…

Constructor for derived class

```
public Car::Car(string name, string model): m_model(model)
{ }
```

Will the above construction work?

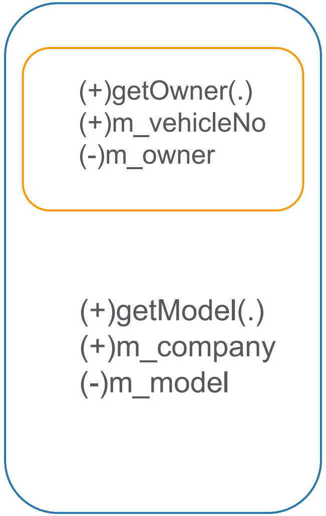
Inheritance : Construction

CODE

```
class Vehicle {
        public:
           Vehicle(string name){
                 m_owner=name;
            }
           string m_vehicleNo;
           string getOwner(){return m_owner;}
        private:
           string m_owner;
}

class Car : public Vehicle {
        public:
           Car(string name, string model);
           string m_company;
           string getModel(){return m_model;}
        private:
           string m_model;
}
```

Memory Structure



(+)getOwner(.)
(+)m_vehicleNo
(-)m_owner

(+)getModel(.)
(+)m_company
(-)m_model

Constructing derived classes?

Review: Order of Construction

1. Construct the Base part
2. Initialise data members…
3. Parse the body…

Constructor for derived class

```
public Car::Car(string name, string model): m_model(model)
{ }
```

Will the above construction work?

- No, as Vehicle do not have a default constructor.

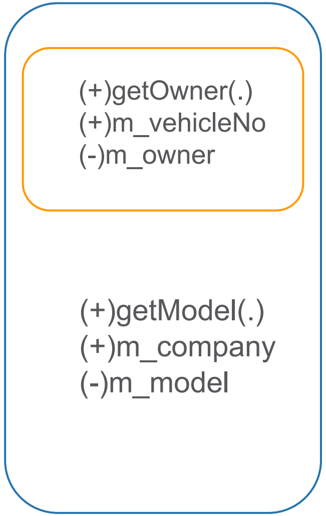
Inheritance : Construction

CODE

```
class Vehicle {
        public:
           Vehicle(string name){
                 m_owner=name;
            }
           string m_vehicleNo;
           string getOwner(){return m_owner;}
        private:
           string m_owner;
}

class Car : public Vehicle {
        public:
           Car(string name, string model);
           string m_company;
           string getModel(){return m_model;}
        private:
           string m_model;
}
```

Memory Structure



Constructing derived classes?

Review: Order of Construction

1. Construct the Base part
2. Initialise data members…
3. Parse the body…

Constructor for derived class

```
public Car::Car(string name, string model): m_model(model)
{ }
```

Will the above construction work?

- No, as Vehicle do not have a default constructor.

Correct construction

```
public Car::Car(string name, string model): Vehicle(name),
m_model(model) { }
```

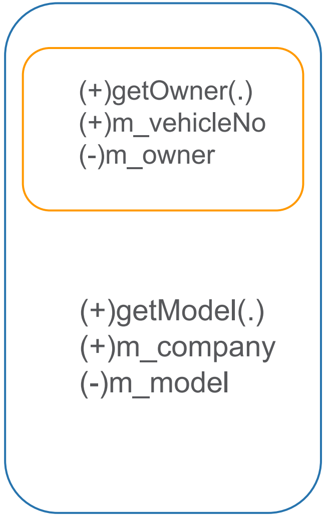
Inheritance : Construction

CODE

```
class Vehicle {
        public:
           Vehicle(string name){
                 m_owner=name;
            }
           string m_vehicleNo;
           string getOwner(){return m_owner;}
        private:
           string m_owner;
}

class Car : public Vehicle {
        public:
           Car(string name, string model);
           string m_company;
           string getModel(){return m_model;}
        private:
           string m_model;
}
```

Memory Structure



Access

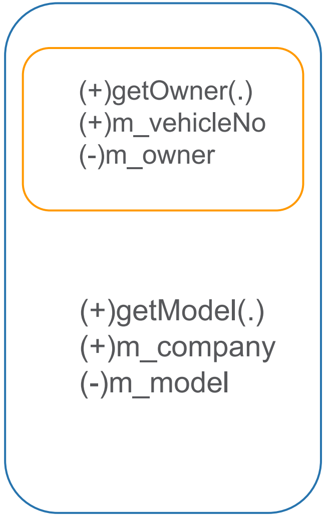
Inheritance : Construction

CODE

```
class Vehicle {
        public:
           Vehicle(string name){
                 m_owner=name;
            }
           string m_vehicleNo;
           string getOwner(){return m_owner;}
        private:
           string m_owner;
}

class Car : public Vehicle {
        public:
           Car(string name, string model);
           string m_company;
           string getModel(){return m_model;}
        private:
           string m_model;
}
```

Memory Structure



Access

Can Car directly access m_owner?

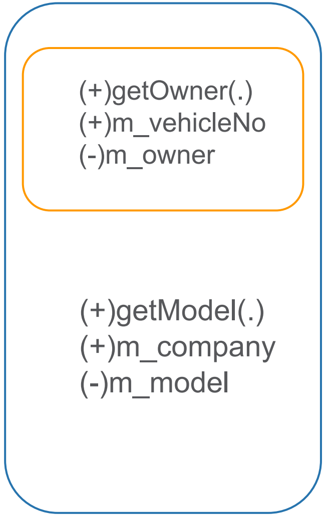
Inheritance : Construction

CODE

```
class Vehicle {
        public:
           Vehicle(string name){
                 m_owner=name;
            }
           string m_vehicleNo;
           string getOwner(){return m_owner;}
        private:
           string m_owner;
}

class Car : public Vehicle {
        public:
           Car(string name, string model);
           string m_company;
           string getModel(){return m_model;}
        private:
           string m_model;
}
```

Memory Structure



Access

Can Car directly access m_owner?

No.

Can Car directly access m_vehicleNo?

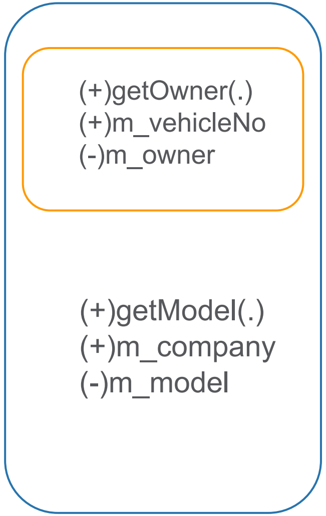
Inheritance : Construction

CODE

```
class Vehicle {
        public:
           Vehicle(string name){
                 m_owner=name;
            }
           string m_vehicleNo;
           string getOwner(){return m_owner;}
        private:
           string m_owner;
}

class Car : public Vehicle {
        public:
           Car(string name, string model);
           string m_company;
           string getModel(){return m_model;}
        private:
           string m_model;
}
```

Memory Structure



(+)getOwner(.)
(+)m_vehicleNo
(-)m_owner

(+)getModel(.)
(+)m_company
(-)m_model

Access

Can Car directly access m_owner?

No.

Can Car directly access m_vehicleNo?

Yes.

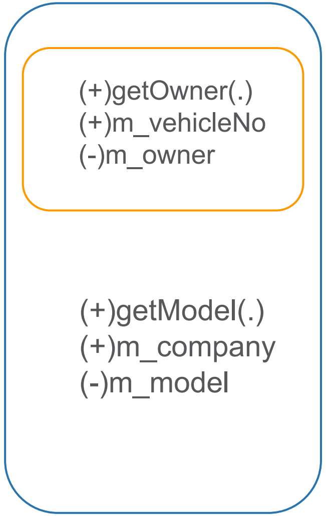
Inheritance : Construction

CODE

```
class Vehicle {
        public:
           Vehicle(string name){
                 m_owner=name;
            }
           string m_vehicleNo;
           string getOwner(){return m_owner;}
        private:
           string m_owner;
}

class Car : public Vehicle {
        public:
           Car(string name, string model);
           string m_company;
           string getModel(){return m_model;}
        private:
           string m_model;
}
```

Memory Structure



(+)getOwner(.)
(+)m_vehicleNo
(-)m_owner

(+)getModel(.)
(+)m_company
(-)m_model

Access

Can Car directly access m_owner?

No.

Can Car directly access m_vehicleNo?

Yes.

Derived class can only access **public member variables** of its base class, but **not private member variables** of its base class.

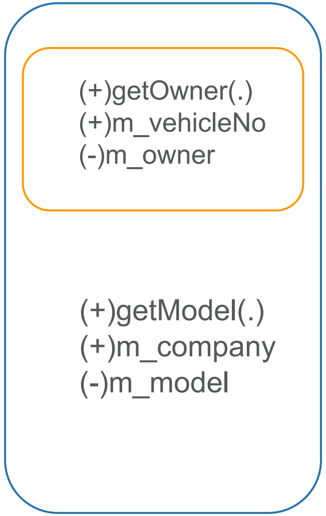
Inheritance : Construction

CODE

```
class Vehicle {
        public:
           Vehicle(string name){
                 m_owner=name;
            }
           string m_vehicleNo;
           string getOwner(){return m_owner;}
        private:
           string m_owner;
}

class Car : public Vehicle {
        public:
           Car(string name, string model);
           string m_company;
           string getModel(){return m_model;}
        private:
           string m_model;
}
```

Memory Structure



Assignment

Car * car = new Vehicle(..), is this valid?

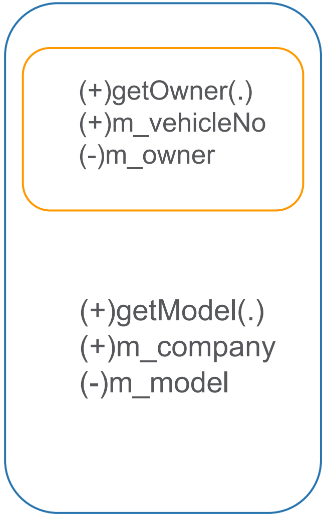
Inheritance : Construction

CODE

```
class Vehicle {
        public:
           Vehicle(string name){
                 m_owner=name;
            }
           string m_vehicleNo;
           string getOwner(){return m_owner;}
        private:
           string m_owner;
}

class Car : public Vehicle {
        public:
           Car(string name, string model);
           string m_company;
           string getModel(){return m_model;}
        private:
           string m_model;
}
```

Memory Structure



(+)getOwner(.)
(+)m_vehicleNo
(-)m_owner

(+)getModel(.)
(+)m_company
(-)m_model

Assignment

Car * car = new Vehicle(..), is this valid?

No.

Vehicle * v = new Car(..), is this valid?

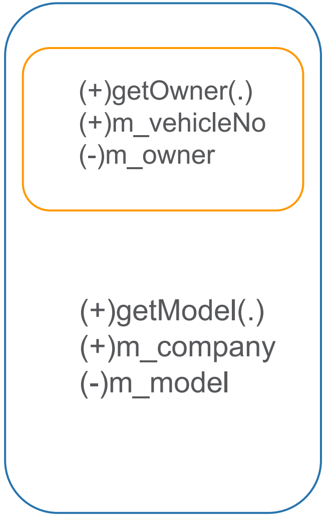
Inheritance : Construction

CODE

```
class Vehicle {
        public:
           Vehicle(string name){
                 m_owner=name;
            }
           string m_vehicleNo;
           string getOwner(){return m_owner;}
        private:
           string m_owner;
}

class Car : public Vehicle {
        public:
           Car(string name, string model);
           string m_company;
           string getModel(){return m_model;}
        private:
           string m_model;
}
```

Memory Structure


(+)getOwner(.)
(+)m_vehicleNo
(-)m_owner

(+)getModel(.)
(+)m_company
(-)m_model

Assignment

Car * car = new Vehicle(..), is this valid?

No.

Vehicle * v = new Car(..), is this valid?

Yes.

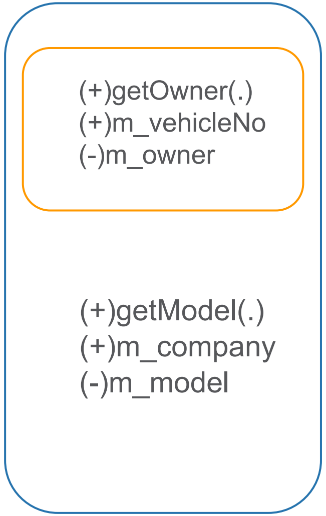
Inheritance : Construction

CODE

```
class Vehicle {
        public:
            Vehicle(string name){
                m_owner=name;
             }
            string m_vehicleNo;
            string getOwner(){return m_owner;}
        private:
            string m_owner;
}

class Car : public Vehicle {
        public:
            Car(string name, string model);
            string m_company;
            string getModel(){return m_model;}
        private:
            string m_model;
}
```

Memory Structure



Assignment

Car * car = new Vehicle(..), is this valid?

No.

Vehicle * v = new Car(..), is this valid?

Yes.

You can assign a **derived class pointer/reference** to the **base class pointer/reference.** The conversion is automatic.

In this case, v points to Vehicle part of the Car class(ref memory structure.)

Inheritance : Overriding

You can **override** a member function of your **base** class in the **derived** class

Inheritance : Overriding

You can **override** a member function of your **base** class in the **derived** class

```
Vehicle* vehicle = new Vehicle(“bruin”);
Car* car = new Car(“bruin”, “ferrari”);

vehicle->move();
car->move();
```

What is the output?

CODE

```
class Vehicle {
        public:
           Vehicle(string name){
                 m_owner=name;
            }
           string m_vehicleNo;
           string getOwner(){return m_owner;}
           void move(){
                 cout<< “Vehicle Moved” <<endl;
            }
        private:
           string m_owner;
}

class Car : public Vehicle {
        public:
           Car(string name, string model);
           string m_company;
           string getModel(){return m_model;}
           void move(){
                 cout<< “Car Moved” <<endl;
            }
        private:
           string m_model;
}
```

Inheritance : Overriding

You can **override** a member function of your **base** class in the **derived** class

```
Vehicle* vehicle = new Vehicle(“bruin”);
Car* car = new Car(“bruin”, “ferrari”);
```

```
vehicle->move();
car->move();
```

What is the output?

```
Vehicle Moved
Car Moved
```

CODE

```
class Vehicle {
        public:
           Vehicle(string name){
                 m_owner=name;
            }
           string m_vehicleNo;
           string getOwner(){return m_owner;}
           void move(){
                 cout<< “Vehicle Moved” <<endl;
            }
        private:
           string m_owner;
}

class Car : public Vehicle {
        public:
           Car(string name, string model);
           string m_company;
           string getModel(){return m_model;}
           void move(){
                 cout<< “Car Moved” <<endl;
            }
        private:
           string m_model;
}
```

Inheritance : Overriding

You can **override** a member function of your **base** class in the **derived** class

```
Vehicle* vehicle = new Vehicle(“bruin”);
Car* car = new Car(“bruin”, “ferrari”);

vehicle->move();
car->move();
```

What is the output?

```
Vehicle Moved
Car Moved
```

How to make car object print “Vehicle Moved”?

CODE

```
class Vehicle {
            public:
                Vehicle(string name){
                        m_owner=name;
                 }
                string m_vehicleNo;
                string getOwner(){return m_owner;}
                void move(){
                        cout<< “Vehicle Moved” <<endl;
                 }
            private:
                string m_owner;
}

class Car : public Vehicle {
            public:
                Car(string name, string model);
                string m_company;
                string getModel(){return m_model;}
                void move(){
                        cout<< “Car Moved” <<endl;
                 }
            private:
                string m_model;
}
```

Inheritance : Overriding

You can **override** a member function of your **base** class in the **derived** class

```
Vehicle* vehicle = new Vehicle("bruin");
Car* car = new Car("bruin", "ferrari");
```

```
vehicle->move();
car->move();
```

What is the output?

```
Vehicle Moved
Car Moved
```

How to make car object print "Vehicle Moved"?

It means calling **base class's** functions

```
car.Vehicle::move();
```

CODE

```
class Vehicle {
          public:
             Vehicle(string name){
                   m_owner=name;
              }
             string m_vehicleNo;
             string getOwner(){return m_owner;}
             void move(){
                   cout<< "Vehicle Moved" <<endl;
              }
          private:
             string m_owner;
}

class Car : public Vehicle {
          public:
             Car(string name, string model);
             string m_company;
             string getModel(){return m_model;}
             void move(){
                   cout<< "Car Moved" <<endl;
              }
          private:
             string m_model;
}
```

Inheritance : Overriding

Let’s try creating an array of different Vehicle’s

```
Vehicle* vehicles[3]
vehicles[0] = new Car(“bruin1”, “Ferrari”);
vehicles[1] = new Bus(“bruin2”, “BruinBus”);
vehicles[2] = new Truck(“bruin3”, “Tesla”);
```

CODE

```
class Vehicle {
        …
          void move(){
                cout<< “Vehicle Moved” <<endl;
           }
        …
}
class Car : public Vehicle {
        …
          void move(){
                cout<< “Car Moved” <<endl;
           }
        …
}
class Bus : public Vehicle {
        …
          void move(){
                cout<< “Bus Moved” <<endl;
           }
        …
}
class Truck : public Vehicle {
        …
          void move(){
                cout<< “Truck Moved” <<endl;
           }
        …
}
```

Inheritance : Overriding

Let’s try creating an array of different Vehicle’s

```
Vehicle* vehicles[3]
vehicles[0] = new Car(“bruin1”, “Ferrari”);
vehicles[1] = new Bus(“bruin2”, “BruinBus”);
vehicles[2] = new Truck(“bruin3”, “Tesla”);
```

```
vehicles[0]->move();
vehicles[1]->move();
vehicles[2]->move();
```

What is the output?

CODE

```
class Vehicle {
        …
          void move(){
                cout<< “Vehicle Moved” <<endl;
           }
        …
}
class Car : public Vehicle {
        …
          void move(){
                cout<< “Car Moved” <<endl;
           }
        …
}
class Bus : public Vehicle {
        …
          void move(){
                cout<< “Bus Moved” <<endl;
           }
        …
}
class Truck : public Vehicle {
        …
          void move(){
                cout<< “Truck Moved” <<endl;
           }
        …
}
```

Inheritance : Overriding

Let’s try creating an array of different Vehicle’s

```
Vehicle* vehicles[3]
vehicles[0] = new Car(“bruin1”, “Ferrari”);
vehicles[1] = new Bus(“bruin2”, “BruinBus”);
vehicles[2] = new Truck(“bruin3”, “Tesla”);
```

```
vehicles[0]->move();
vehicles[1]->move();
vehicles[2]->move();
```

What is the output?

```
Vehicle Moved
Vehicle Moved
Vehicle Moved
```

Why so?

CODE

```
class Vehicle {
        …
          void move(){
                cout<< “Vehicle Moved” <<endl;
           }
        …
}
class Car : public Vehicle {
        …
          void move(){
                cout<< “Car Moved” <<endl;
           }
        …
}
class Bus : public Vehicle {
        …
          void move(){
                cout<< “Bus Moved” <<endl;
           }
        …
}
class Truck : public Vehicle {
        …
          void move(){
                cout<< “Truck Moved” <<endl;
           }
        …
}
```

Inheritance : Overriding

Let’s try creating an array of different Vehicle’s

```
Vehicle* vehicles[3]
vehicles[0] = new Car(“bruin1”, “Ferrari”);
vehicles[1] = new Bus(“bruin2”, “BruinBus”);
vehicles[2] = new Truck(“bruin3”, “Tesla”);
```

```
vehicles[0]->move();
vehicles[1]->move();
vehicles[2]->move();
```

What is the output?

```
Vehicle Moved
Vehicle Moved
Vehicle Moved
```

Why so?

- because of static binding.

CODE

```
class Vehicle {
        …
          void move(){
                cout<< “Vehicle Moved” <<endl;
          }
        …
}
class Car : public Vehicle {
        …
          void move(){
                cout<< “Car Moved” <<endl;
          }
        …
}
class Bus : public Vehicle {
        …
          void move(){
                cout<< “Bus Moved” <<endl;
          }
        …
}
class Truck : public Vehicle {
        …
          void move(){
                cout<< “Truck Moved” <<endl;
          }
        …
}
```

Inheritance : Overriding

Let’s try creating an array of different Vehicle’s

```
Vehicle* vehicles[3]
vehicles[0] = new Car(“bruin1”, “Ferrari”);
vehicles[1] = new Bus(“bruin2”, “BruinBus”);
vehicles[2] = new Truck(“bruin3”, “Tesla”);
```

```
vehicles[0]->move();
vehicles[1]->move();
vehicles[2]->move();
```

What is the output?

What is the output we need?

```
Vehicle Moved
Vehicle Moved
Vehicle Moved
```

CODE

```
class Vehicle {
        …
          void move(){
                cout<< “Vehicle Moved” <<endl;
           }
        …
}
class Car : public Vehicle {
        …
          void move(){
                cout<< “Car Moved” <<endl;
           }
        …
}
class Bus : public Vehicle {
        …
          void move(){
                cout<< “Bus Moved” <<endl;
           }
        …
}
class Truck : public Vehicle {
        …
          void move(){
                cout<< “Truck Moved” <<endl;
           }
        …
}
```

Inheritance : Overriding

Let’s try creating an array of different Vehicle’s

```
Vehicle* vehicles[3]
vehicles[0] = new Car(“bruin1”, “Ferrari”);
vehicles[1] = new Bus(“bruin2”, “BruinBus”);
vehicles[2] = new Truck(“bruin3”, “Tesla”);
```

```
vehicles[0]->move();
vehicles[1]->move();
vehicles[2]->move();
```

What is the output?

```
Vehicle Moved
Vehicle Moved
Vehicle Moved
```

What is the output we need?

```
Car Moved
Bus Moved
Truck Moved
```

How to achieve this?

CODE

```
class Vehicle {
        …
          void move(){
                cout<< “Vehicle Moved” <<endl;
           }
        …
}
class Car : public Vehicle {
        …
          void move(){
                cout<< “Car Moved” <<endl;
           }
        …
}
class Bus : public Vehicle {
        …
          void move(){
                cout<< “Bus Moved” <<endl;
           }
        …
}
class Truck : public Vehicle {
        …
          void move(){
                cout<< “Truck Moved” <<endl;
           }
        …
}
```

Polymorphism

Polymorphism : What is it?

The same object taking **different forms**.

You can access objects of different types through the **same interface.** Each type can provide its **own independent implementation** of this interface**.**

Polymorphism : What is it?

The same object taking **different forms**.

You can access objects of different types through the **same interface**. Each type can provide its **own independent implementation** of this interface.

```
Vehicle* vehicles[3]
vehicles[0] = new Car(“bruin1”, “Ferrari”);
vehicles[1] = new Bus(“bruin2”, “BruinBus”);
vehicles[2] = new Truck(“bruin3”, “Tesla”);
```

The compiler knows that these are pointers of type Vehicle, and calls Vehicle’s respective functions instead of derived class’s functions.

What we need is a specification of which function to be called at run time

How do we tell the compiler to call the appropriate function at run time(dynamic binding)?

Polymorphism : Virtual functions

Name the function as **virtual,** Rest all will be a **magic.**

```
class Vehicle {
        …
          virtual void move(){
                cout<< “Vehicle Moved” <<endl;
           }
        …
}
class Car : public Vehicle {
        …
          virtual void move(){
                cout<< “Car Moved” <<endl;
           }
        …
}
class Bus : public Vehicle {
        …
          virtual void move(){
                cout<< “Bus Moved” <<endl;
           }
        …
}
class Truck : public Vehicle {
        …
          virtual void move(){
                cout<< “Truck Moved” <<endl;
           }
        …
}
```

Polymorphism : Virtual functions

Name the function as **virtual**, Rest all will be a **magic.**

```
Vehicle* vehicles[3]
vehicles[0] = new Car(“bruin1”, “Ferrari”);
vehicles[1] = new Bus(“bruin2”, “BruinBus”);
vehicles[2] = new Truck(“bruin3”, “Tesla”);
```

```
vehicles[0]->move();
vehicles[1]->move();
vehicles[2]->move();
```

What is the output?

```
class Vehicle {
        …
          virtual void move(){
                cout<< “Vehicle Moved” <<endl;
          }
        …
}
class Car : public Vehicle {
        …
          virtual void move(){
                cout<< “Car Moved” <<endl;
          }
        …
}
class Bus : public Vehicle {
        …
          virtual void move(){
                cout<< “Bus Moved” <<endl;
          }
        …
}
class Truck : public Vehicle {
        …
          virtual void move(){
                cout<< “Truck Moved” <<endl;
          }
        …
}
```

Polymorphism : Virtual functions

Name the function as **virtual**, Rest all will be a **magic.**

```
Vehicle* vehicles[3]
vehicles[0] = new Car(“bruin1”, “Ferrari”);
vehicles[1] = new Bus(“bruin2”, “BruinBus”);
vehicles[2] = new Truck(“bruin3”, “Tesla”);

vehicles[0]->move();
vehicles[1]->move();
vehicles[2]->move();
```

What is the output?

```
Car Moved
Bus Moved
Truck Moved
```

What if a derived class has no implementation for move()?

```
class Vehicle {
        …
          virtual void move(){
                  cout<< “Vehicle Moved” <<endl;
          }
        …
}
class Car : public Vehicle {
        …
          virtual void move(){
                  cout<< “Car Moved” <<endl;
          }
        …
}
class Bus : public Vehicle {
        …
          virtual void move(){
                  cout<< “Bus Moved” <<endl;
          }
        …
}
class Truck : public Vehicle {
        …
          virtual void move(){
                  cout<< “Truck Moved” <<endl;
          }
        …
}
```

Polymorphism : Virtual functions

Name the function as **virtual,** Rest all will be a **magic.**

```
Vehicle* vehicles[3]
vehicles[0] = new Car(“bruin1”, “Ferrari”);
vehicles[1] = new Bus(“bruin2”, “BruinBus”);
vehicles[2] = new Truck(“bruin3”, “Tesla”);

vehicles[0]->move();
vehicles[1]->move();
vehicles[2]->move();
```

What is the output?

```
Car Moved
Bus Moved
Truck Moved
```

What if a derived class has no implementation for move()?

- It uses the base class’s implementation.

If specified in base class, No need to specify **virtual** in derived class, they will be virtual by default as inherited from base class.

```
class Vehicle {
        …
          virtual void move(){
                    cout<< “Vehicle Moved” <<endl;
          }
        …
}
class Car : public Vehicle {
        …
          virtual void move(){
                    cout<< “Car Moved” <<endl;
          }
        …
}
class Bus : public Vehicle {
        …
          virtual void move(){
                    cout<< “Bus Moved” <<endl;
          }
        …
}
class Truck : public Vehicle {
        …
          virtual void move(){
                    cout<< “Truck Moved” <<endl;
          }
        …
}
```

Polymorphism : Virtual functions

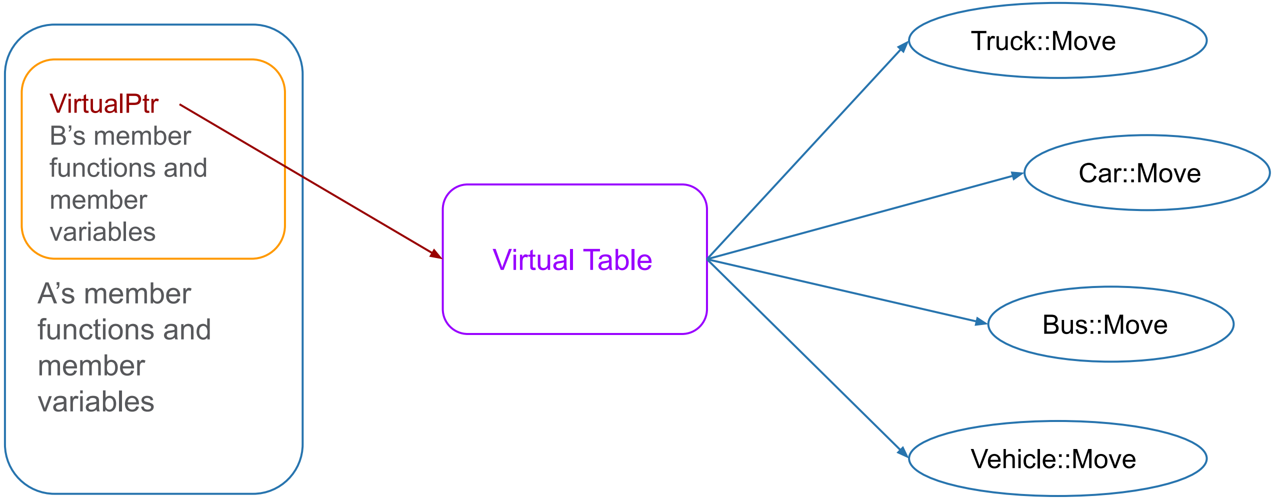
Rest all will be a **magic…** What is this magic?

A is derived from B → class A : public B {**...**}

Polymorphism : Virtual functions

Rest all will be a **magic…** What is this magic?

A is derived from B → class A : public B {**…**}


VirtualPtr
B's member functions and member variables
A's member functions and member variables
Virtual Table
Truck::Move
Car::Move
Bus::Move
Vehicle::Move

Each class in which virtual functions are present, has its own virtual ptr pointing to its virtual table.

Polymorphism : Virtual functions

What if we don’t want an implementation for move(.) in the **base** class, **Vehicle?**

How can we achieve it?

```
class Vehicle {
        …
            virtual void move(){
                  cout<< “Vehicle Moved” <<endl;
            }
        …
}
class Car : public Vehicle {
        …
            virtual void move(){
                  cout<< “Car Moved” <<endl;
            }
        …
}
class Bus : public Vehicle {
        …
            virtual void move(){
                  cout<< “Bus Moved” <<endl;
            }
        …
}
class Truck : public Vehicle {
        …
            virtual void move(){
                  cout<< “Truck Moved” <<endl;
            }
        …
}
```

Polymorphism : Pure virtual functions

What if we don’t want an implementation for move(.) in the **base** class, **Vehicle?**

How can we achieve it?

- function declaration = 0;

These are called Pure Virtual Functions.

```
class Vehicle {
          …
            virtual void move() = 0;
          …
}
class Car : public Vehicle {
          …
            virtual void move(){
                    cout<< “Car Moved” <<endl;
             }
          …
}
class Bus : public Vehicle {
          …
            virtual void move(){
                    cout<< “Bus Moved” <<endl;
             }
          …
}
class Truck : public Vehicle {
          …
            virtual void move(){
                    cout<< “Truck Moved” <<endl;
             }
          …
}
```

Polymorphism : Pure virtual functions

What if we don’t want an implementation for move(.) in the **base** class, **Vehicle?**

How can we achieve it?

- function declaration = 0;

These are called Pure Virtual Functions.

Sometimes, we do not need implementations where it doesn’t make much sense.

Example:

- **Shape’s** draw function from class

Base Classes with pure virtual functions are also called **Abstract Base Classes(ABC).**

```
class Vehicle {
            …
               virtual void move() = 0;
            …
}
class Car : public Vehicle {
            …
               virtual void move(){
                       cout<< “Car Moved” <<endl;
                }
            …
}
class Bus : public Vehicle {
            …
               virtual void move(){
                       cout<< “Bus Moved” <<endl;
                }
            …
}
class Truck : public Vehicle {
            …
               virtual void move(){
                       cout<< “Truck Moved” <<endl;
                }
            …
}
```

Polymorphism : Pure virtual functions

What if we don’t want an implementation for move(.) in the **base** class, **Vehicle?**

How can we achieve it?

- function declaration = 0;

These are called Pure Virtual Functions.

Sometimes, we do not need implementations where it doesn’t make much sense.

Example:

- **Shape’s** draw function from class

Base Classes with pure virtual functions are also called **Abstract Base Classes(ABC).**

Abstract classes cannot be instantiated

i,e **Vehicle* v1 = new Vehicle(); gives compile error.**

```
class Vehicle {
          …
            virtual void move() = 0;
          …
}
class Car : public Vehicle {
          …
            virtual void move(){
                  cout<< “Car Moved” <<endl;
             }
          …
}
class Bus : public Vehicle {
          …
            virtual void move(){
                  cout<< “Bus Moved” <<endl;
             }
          …
}
class Truck : public Vehicle {
          …
            virtual void move(){
                  cout<< “Truck Moved” <<endl;
             }
          …
}
```

Destruction

Inheritance : Destruction

Review: Order of Destruction

1. Parse the body…
2. Destroy data members…
3. …?

Inheritance : Destruction

Review: Order of Destruction

1. Parse the body…
2. Destroy data members…
3. Destroy the Base part (calls **base** destructor)

Inheritance : Destruction

Review: Order of Destruction

1. Parse the body…
2. Destroy data members…
3. Destroy the Base part (calls **base** destructor)

```
Vehicle* vehicles[3]
vehicles[0] = new Car("bruin1", "Ferrari");
vehicles[1] = new Bus("bruin2", "BruinBus");
vehicles[2] = new Truck("bruin3", "Tesla");

delete vehicles[0];
delete vehicles[1];
delete vehicles[2];
```

What is the output?

```
class Vehicle {
        …
          ~Vehicle(){
                cout<< "Vehicle Destroyed" <<endl;
          }
        …
}
class Car : public Vehicle {
        …
          ~Car(){
                cout<< "Car Destroyed" <<endl;
          }
        …
}
class Bus : public Vehicle {
        …
          ~Bus(){
                cout<< "Bus Destroyed" <<endl;
          }
        …
}
class Truck : public Vehicle {
        …
          ~Truck(){
                cout<< "Truck Destroyed" <<endl;
          }
        …
}
```

Inheritance : Destruction

Review: Order of Destruction

1. Parse the body…
2. Destroy data members…
3. Destroy the Base part (calls **base** destructor)

```
Vehicle* vehicles[3]
vehicles[0] = new Car("bruin1", "Ferrari");
vehicles[1] = new Bus("bruin2", "BruinBus");
vehicles[2] = new Truck("bruin3", "Tesla");

delete vehicles[0];
delete vehicles[1];
delete vehicles[2];
```

What is the output?

```
Vehicle Destroyed
Vehicle Destroyed
Vehicle Destroyed
```

Why so?

```
class Vehicle {
        …
          ~Vehicle(){
                cout<< "Vehicle Destroyed" <<endl;
          }
        …
}
class Car : public Vehicle {
        …
          ~Car(){
                cout<< "Car Destroyed" <<endl;
          }
        …
}
class Bus : public Vehicle {
        …
          ~Bus(){
                cout<< "Bus Destroyed" <<endl;
          }
        …
}
class Truck : public Vehicle {
        …
          ~Truck(){
                cout<< "Truck Destroyed" <<endl;
          }
        …
}
```

Inheritance : Destruction

Review: Order of Destruction

1. Parse the body…
2. Destroy data members…
3. Destroy the Base part (calls **base** destructor)

```
Vehicle* vehicles[3]
vehicles[0] = new Car("bruin1", "Ferrari");
vehicles[1] = new Bus("bruin2", "BruinBus");
vehicles[2] = new Truck("bruin3", "Tesla");

delete vehicles[0];
delete vehicles[1];
delete vehicles[2];
```

What is the output?

```
Vehicle Destroyed
Vehicle Destroyed
Vehicle Destroyed
```

Why so?

- because of static binding, even Destructors have to be virtual.

```
class Vehicle {
        …
          ~Vehicle(){
                cout<< "Vehicle Destroyed" <<endl;
          }
        …
}
class Car : public Vehicle {
        …
          ~Car(){
                cout<< "Car Destroyed" <<endl;
          }
        …
}
class Bus : public Vehicle {
        …
          ~Bus(){
                cout<< "Bus Destroyed" <<endl;
          }
        …
}
class Truck : public Vehicle {
        …
          ~Truck(){
                cout<< "Truck Destroyed" <<endl;
          }
        …
}
```

Inheritance : Destruction

Review: Order of Destruction

1. Parse the body…
2. Destroy data members…
3. Destroy the Base part (calls **base** destructor)

```
Vehicle* vehicles[3]
vehicles[0] = new Car(“bruin1”, “Ferrari”);
vehicles[1] = new Bus(“bruin2”, “BruinBus”);
vehicles[2] = new Truck(“bruin3”, “Tesla”);

delete vehicles[0];
delete vehicles[1];
delete vehicles[2];
```

What is the output?

```
class Vehicle {
        …
          virtual ~Vehicle(){
                cout<< “Vehicle Destroyed” <<endl;
          }
        …
}
class Car : public Vehicle {
        …
          virtual ~Car(){
                cout<< “Car Destroyed” <<endl;
          }
        …
}
class Bus : public Vehicle {
        …
          virtual ~Bus(){
                cout<< “Bus Destroyed” <<endl;
          }
        …
}
class Truck : public Vehicle {
        …
          virtual ~Truck(){
                cout<< “Truck Destroyed” <<endl;
          }
        …
}
```

Inheritance : Destruction

Review: Order of Destruction

1. Parse the body…
2. Destroy data members…
3. Destroy the Base part (calls **base** destructor)

```
Vehicle* vehicles[3]
vehicles[0] = new Car(“bruin1”, “Ferrari”);
vehicles[1] = new Bus(“bruin2”, “BruinBus”);
vehicles[2] = new Truck(“bruin3”, “Tesla”);

delete vehicles[0];
delete vehicles[1];
delete vehicles[2];
```

What is the output?

```
Car Destroyed
Vehicle Destroyed
Bus Destroyed
Vehicle Destroyed
Truck Destroyed
Vehicle Destroyed
```

```
class Vehicle {
        …
          virtual ~Vehicle(){
                cout<< “Vehicle Destroyed” <<endl;
          }
        …
}
class Car : public Vehicle {
        …
          virtual ~Car(){
                cout<< “Car Destroyed” <<endl;
          }
        …
}
class Bus : public Vehicle {
        …
          virtual ~Bus(){
                cout<< “Bus Destroyed” <<endl;
          }
        …
}
class Truck : public Vehicle {
        …
          virtual ~Truck(){
                cout<< “Truck Destroyed” <<endl;
          }
        …
}
```

Inheritance : Destruction

Review: Order of Destruction

1. Parse the body…
2. Destroy data members…
3. Destroy the Base part (calls **base** destructor)

```
Vehicle* vehicles[3]
vehicles[0] = new Car("bruin1", "Ferrari");
vehicles[1] = new Bus("bruin2", "BruinBus");
vehicles[2] = new Truck("bruin3", "Tesla");

delete vehicles[0];
delete vehicles[1];
delete vehicles[2];
```

What is the output?

```
Car Destroyed
Vehicle Destroyed
Bus Destroyed
Vehicle Destroyed
Truck Destroyed
Vehicle Destroyed
```

if a class is designed to be a base class, declare a destructor for it, make it virtual. (and implement it)

```
class Vehicle {
        …
          virtual ~Vehicle(){
                cout<< "Vehicle Destroyed" <<endl;
          }
        …
}
class Car : public Vehicle {
        …
          virtual ~Car(){
                cout<< "Car Destroyed" <<endl;
          }
        …
}
class Bus : public Vehicle {
        …
          virtual ~Bus(){
                cout<< "Bus Destroyed" <<endl;
          }
        …
}
class Truck : public Vehicle {
        …
          virtual ~Truck(){
                cout<< "Truck Destroyed" <<endl;
          }
        …
}
```

Recursion

Recursion

Recursion is solving a problem by breaking it into smaller problems, solving the subproblems and using their results to solve the original problem.

Recursion

Recursion is solving a problem by breaking it into smaller problems, solving the subproblems and using their results to solve the original problem.

You can compare it with mathematical functions
f(n) = 2*f(n-1), f(0)=1

Can you guess the closed form for **f(n)**?

Recursion

Recursion is solving a problem by breaking it into smaller problems, solving the subproblems and using their results to solve the original problem.

You can compare it with mathematical functions
f(n) = 2*f(n-1), f(0)=1

Can you guess the closed form for **f(n)**?

One of the simple, yet powerful concepts in computer science.

Recursion : Guess the next number?

1, 1, 2, 3, 5, 8, 13, 21, 34, ?

Recursion : Guess the next number?

1, 1, 2, 3, 5, 8, 13, 21, 34, ?

It’s 55

What’s the relation?

Recursion : Guess the next number?

1, 1, 2, 3, 5, 8, 13, 21, 34, ?

It’s 55

What’s the relation?
f(n) = f(n-1)+f(n-2), f(0)=1,f(1)=1

It’s called a Fibonacci sequence

Recursion : Reverse a string

Given a string s, create string s_rev representing reverse of the given string
(Just provide the pseudocode)

Example
s = “bruin”
s_rev = “niurb”

Recursion : Reverse a string

Given a string s, create string s_rev representing reverse of the given string
(Just provide the pseudocode)

Example
s = “bruin”
s_rev = “niurb”

Observe that
s_rev = “niur” + “b” = reverse(“ruin”) + “b”

Can you see the recursion??

Recursion : Reverse a string

Given a string s, create string s_rev representing reverse of the given string
(Just provide the pseudocode)

Example
s = “bruin”
s_rev = “niurb”

Observe that
s_rev = “niur” + “b” = reverse(“ruin”) + “b”

Can you see the recursion??

```
reverse(string s):
    // base case
    int N  = s.length()
    If N ==0 || N==1:
        return s;

    return reverse(s[1..N-1])+ “s[0]”
```

Recursion : Merge-Sort

Given two sorted arrays, seq1, seq2 How can we get a single sorted array containing all the elements?

Seq1 : 2, 4, 6, 8, 9, 11, 18, 20, 25
Seq2 : 1, 3, 5, 7, 10, 15, 100
Sorted Array : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 15, 18, 20, 25, 100

Recursion : Merge-Sort

Given two sorted arrays, seq1, seq2 How can we get a single sorted array containing all the elements?

Seq1 : 2, 4, 6, 8, 9, 11, 18, 20, 25
Seq2 : 1, 3, 5, 7, 10, 15, 100
Sorted Array : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 15, 18, 20, 25, 100

```
void MERGE(int * A, int p, int q, int r) {
        int n1 = q-p+1;
        int n2 = r-q; // not considering q
        int L[n1+1];
        int R[n2+1];

        for(int i=0; i<n1; i++){
                L[i] = A[p+i];
        }
        L[n1] = INT_MAX;

        for(int i=0; i<n2; i++){
                R[i] = A[q+i+1];
        }
        R[n2] = INT_MAX;

        int i = 0;
        int j = 0;
        for(int k=p; k<=r ; k++){
                if(L[i] <= R[j]){
                        A[k] = L[i];
                        i = i+1;
                }else {
                        A[k] = R[j];
                        j = j+1;
                }
        }
        return;
}
```

The Merge Step

Recursion : Merge-Sort

Given two sorted arrays, seq1, seq2 How can we get a single sorted array containing all the elements?

Seq1 : 2, 4, 6, 8, 9, 11, 18, 20, 25
Seq2 : 1, 3, 5, 7, 10, 15, 100
Sorted Array : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 15, 18, 20, 25, 100

Merge Sort

```
void MERGE_SORT(int * A, int p, int r) {
        if(p < r){
                int q = (p+r)/2;
                MERGE_SORT(A, p, q);
                MERGE_SORT(A, q+1, r);
                MERGE(A, p, q, r);
        }
}
```

```
void MERGE(int * A, int p, int q, int r) {
        int n1 = q-p+1;
        int n2 = r-q; // not considering q
        int L[n1+1];
        int R[n2+1];

        for(int i=0; i<n1; i++){
                L[i] = A[p+i];
        }
        L[n1] = INT_MAX;

        for(int i=0; i<n2; i++){
                R[i] = A[q+i+1];
        }
        R[n2] = INT_MAX;

        int i = 0;
        int j = 0;
        for(int k=p; k<=r ; k++){
                if(L[i] <= R[j]){
                        A[k] = L[i];
                        i = i+1;
                }else {
                        A[k] = R[j];
                        j = j+1;
                }
        }
        return;
}
```

Recursion : Guidelines

See if the problem can be solved by solving subproblems

Take care of base cases

Often, recursion is leap of faith. You just assume that the subproblems are solved :)

A little more than recursion…

A(0, n) = n+1
A(m+1, 0) = A(m, 1)
A(m+1, n+1) = A(m, A(m+1, n))

A recursion in two variables, calculate A(4,2)?
Feel free to use your computers and code it up…

A little more than recursion…

$A(0, n) = n+1$
$A(m+1, 0) = A(m, 1)$
$A(m+1, n+1) = A(m, A(m+1, n))$

A recursion in two variables, calculate A(4,2)?
Feel free to use your computers and code it up…

Called as **Ackermann function.** This function grows rapidly
$A(4,2) = 2^{65536}-3$

Mere recursion won't help, need to use technique called
Dynamic Programming