CS-202

C++ Classes – Inheritance (Pt.2)

C. Papachristos
Autonomous Robots Lab
University of Nevada, Reno

Course Week

Course , Projects , Labs:

Monday	Tuesday	Wednesday	Thursday	Friday
			Lab (4 Sections)	
	CLASS	RL – Session	CLASS	
PASS Session	PASS Session	**Project DEADLINE**	NEW Project	

Your 5th Project will be announced today Thursday 3/1.

4th Project Deadline was this Wednesday 2/28.

- NO Project accepted past the 24-hrs delayed extension (@ 20% grade penalty).
- Send what you have in time!
- Always check out **WebCampus** CS-202 Samples for some **help** !

Today's Topics

Method Overriding

- Overriding *vs* Overloading

Inheritance Rules

- Base–Derived Constructors
- Derived–Base Destructors
- Assignment Operator

Polymorphism Prelude

- Base class pointers to address Derived class Objects

Advanced Examples

- Derived–to–Base forwarding via `static_cast`-ing
- `protected` Interface: Constructors, Destructor, Assignment Operator

Code Reuse

Important to successful coding

- Efficiency: No need to reinvent the wheel.
- Error free: If code already used/tested (not guaranteed, but more likely).

Ways to reuse code?

- Functions
- Classes
- Aggregation:
  `RentalAgency` "*has-a*" `RentalCar`
- Inheritance!

Object Relationships

"*Uses a*" relationship:

- **`ObjectA`** "*uses an*" **`ObjectB`**

  `Car` refuels from a **`GasStation`**

"*Has a*" – Composition or Aggregation (more in this distinction later…)

- **`ObjectA`** "*has an*" **`ObjectB`**

  `Car` incorporates a **`Sensor`**

"*Is a*" or "*Is a kind of*" – Inheritance

- **`ObjectA`** "*is a*" **`ObjectB`**

  `Car` is a **`Vehicle`**

Inheritance Relationship

What is Inheritance?

- A **`Car`** *"is also a / is a kind of"* **`Vehicle`**

Code reuse by sharing related Set-Methods:

- Specific classes *"Inherit"* methods from general classes.

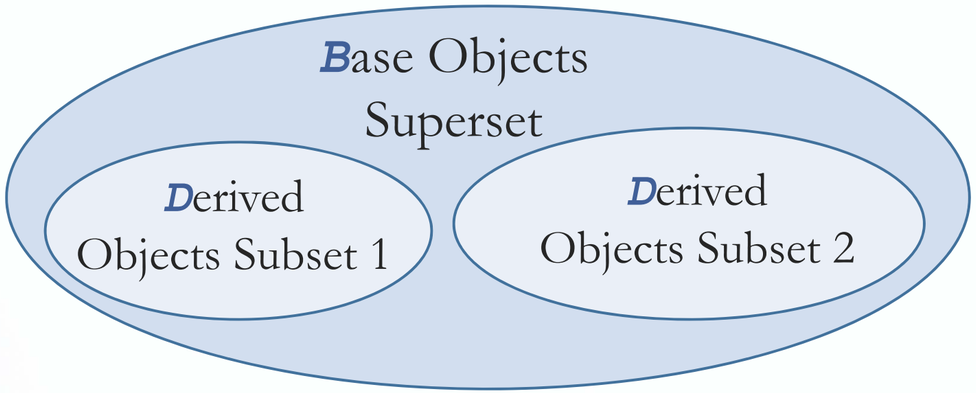
The **`Car`** Class Inherits from the **`Vehicle`** Class:

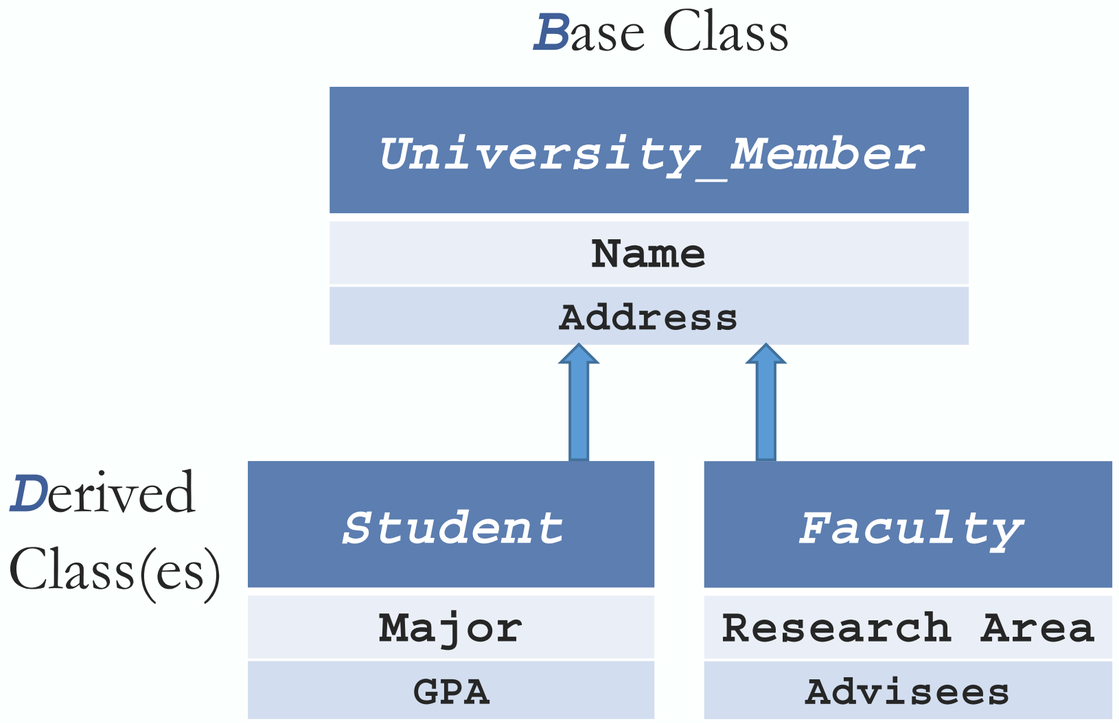
- **`Vehicle`** is the general class, or the *Base* Class.
- **`Car`** is the specialized class, or *Derived* Class, that Inherits from **`Vehicle`**.

Inheritance Relationship

Inheritance Example:

- Every ***D*** is also a ***B***
- Not every ***Di*** is a ***Dj***
- Some ***B***s are ***D***s





Inheritance Relationship

Inheritance Syntax:

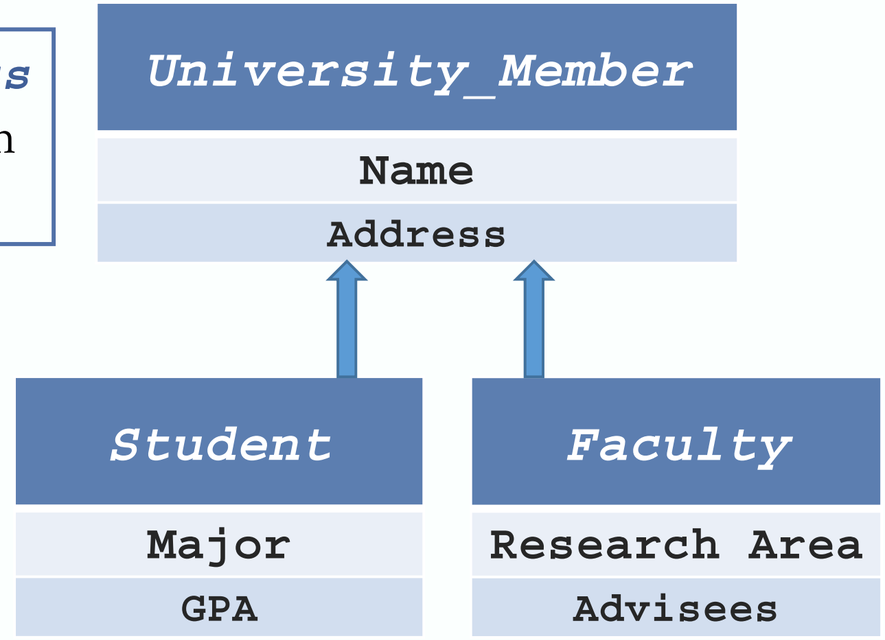
```
class BaseClass {
    public:
        //operations
    private:
        //data
};

class DerivedClass : public BaseClass {
    public:
        //operations
    private:
        //data
};
```

Indicates that this *DerivedClass* Inherits data and operations from this *BaseClass*

*B*ase Class

*D*erived Class(es)

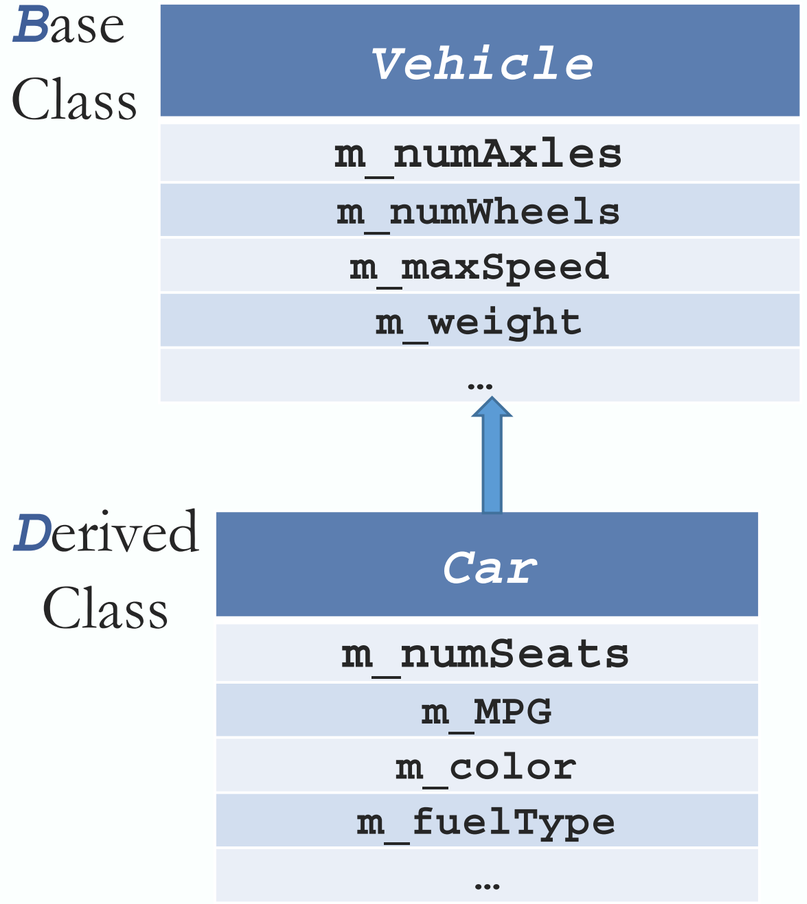


Inheritance Relationship

Indicative Code example:

```
class Vehicle {
  public:
    // functions
  private:
    // data
    int     m_numAxles;
    int     m_numWheels;
    int     m_maxSpeed;
    double  m_weight;
} ;
```

All *Vehicle*s have axles, wheels, a max speed, and a weight



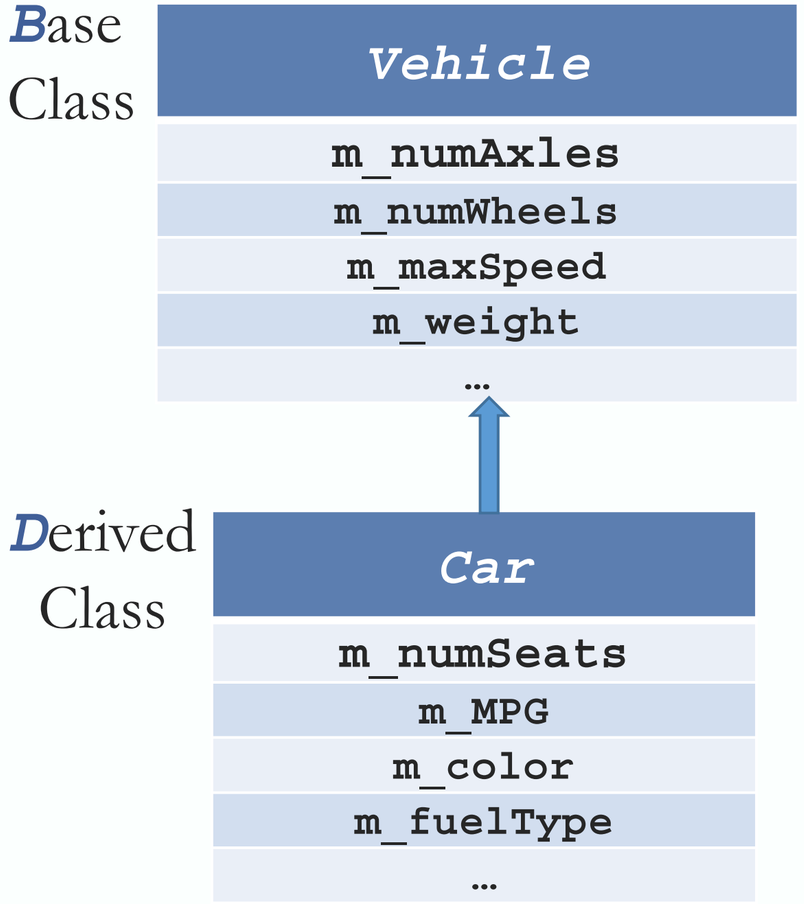
Inheritance Relationship

Indicative Inheritance Code example:

➢ Colon in Declaration indicates Inheritance.

```
class Car : public Vehicle {
  public:
    // functions
  private:
    // data
    int     m_numSeats;
    double  m_MPG;
    string  m_color;
    string  m_fuelType;
} ;
```

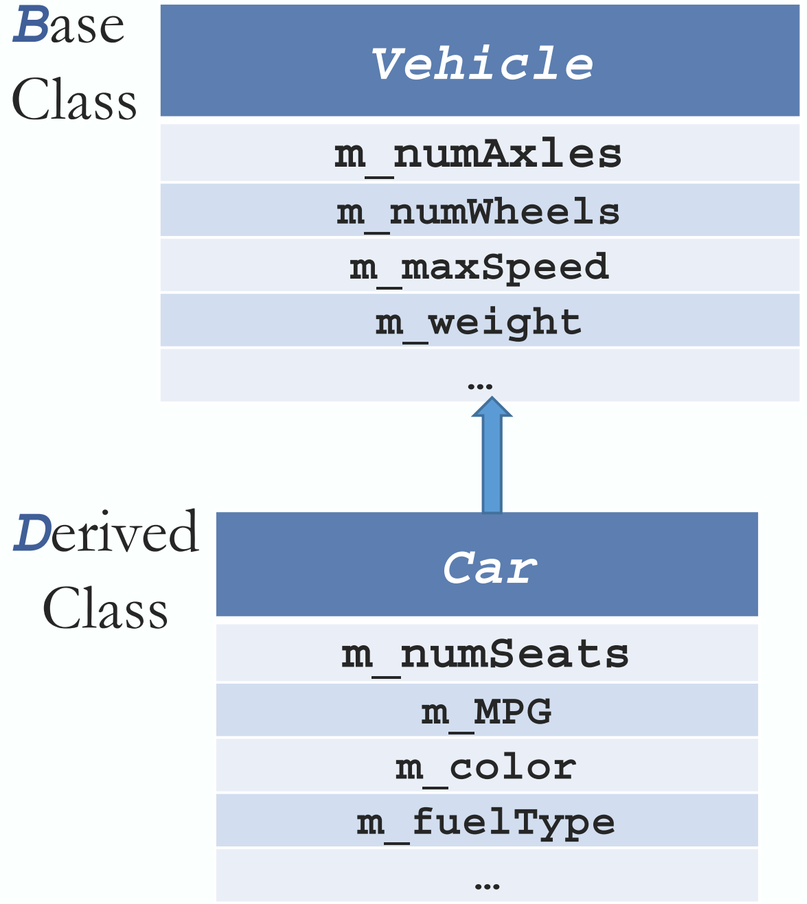
All *Car*s have a number of seats, a MPG value, a color, and a fuel type



Inheritance Relationship

Indicative Inheritance Code example:

```
class Car :
  public Vehicle { /*etc*/ };
class Plane :
  public Vehicle { /*etc*/ };
class SpaceShuttle :
  public Vehicle { /*etc*/ };
class BigRig :
  public Vehicle { /*etc*/ };
```



Composition Relationship

Defining Composition?

- A **`Car`** *"is made with a / incorporates a"* **`Chassis`**.

The **`Car`** class *"Owns"* a class object of type **`Chassis`** :

- **`Car`** object is composed by a **`Chassis`** object.

The **`Car`** class has the *"Lifetime*-responsibility" for its **`Chassis`** member object :

- The **`Chassis`** cannot "live" out of context of a **`Car`**.
- If the **`Car`** is destroyed, the **`Chassis`** is also destroyed!

Composition Relationship

Indicative Code example:

- No Inheritance for ***Chassis***:

```
class Chassis {
  public:
    // functions
  private:
    // data
    char m_material[MAT_LENGTH];
    double m_weight;
    double m_maxLoad;
} ;
```

```
class Car : public Vehicle {
  public:
    // functions
  private:
    // made-with (composition)
    Chassis m_chassis;
} ;
```

Aggregation Relationship

What is Aggregation?

- A **`Car`** "*can have a / use a*" **`Driver`**.

The **`Car`** Class can be "*Linked*-with" an object of type **`Driver`** :

- **`Car`** object can possibly have one **`Driver`** object, or another, or none at all.

The **`Driver`** class is only "*Associated*-to" the **`Car`** Class.

- A **`Driver`** can "live" out of context of a **`Car`**.
- A **`Driver`** must be linked with the **`Car`** object *via a Pointer* to a separately existing external **`Driver`** Object.

Aggregation Relationship

Indicative Code example:

- ***Driver*** Inherits from *Base* Class ***Person***:

```
class Driver: public Person {
  public:
    // functions
  private:
    // data
    Date m_licenseExpire;
    char m_licenseType[LIC_MAX];
} ;
```

```
class Car : public Vehicle {
  public:
    // functions
  private:
    // has-a (aggregation)
    Driver * m_driver;
} ;
```

Inheritance (detailed)

Why Inheritance?

Abstraction for sharing similarities while retaining differences.

Group classes into related families:

- Share common operations and data.

Multiple Inheritance(s) is possible:

- Inherit from multiple Base Classes

```
class Car : public Vehicle ,
            public DMVRegistrable { … };
```

Promotes code reuse

- Design general Class once.
- Extend implementation(s) through Inheritance.

Inheritance (detailed)

Access Specifier(s)

Inheritance can be `public`, `private`, or `protected`.

- Our focus will be `public` Inheritance.

`Public`

- Everything that is aware of Base(Parent) and Derived(Child) is also aware that Derived Inherits from Base.

`Protected`

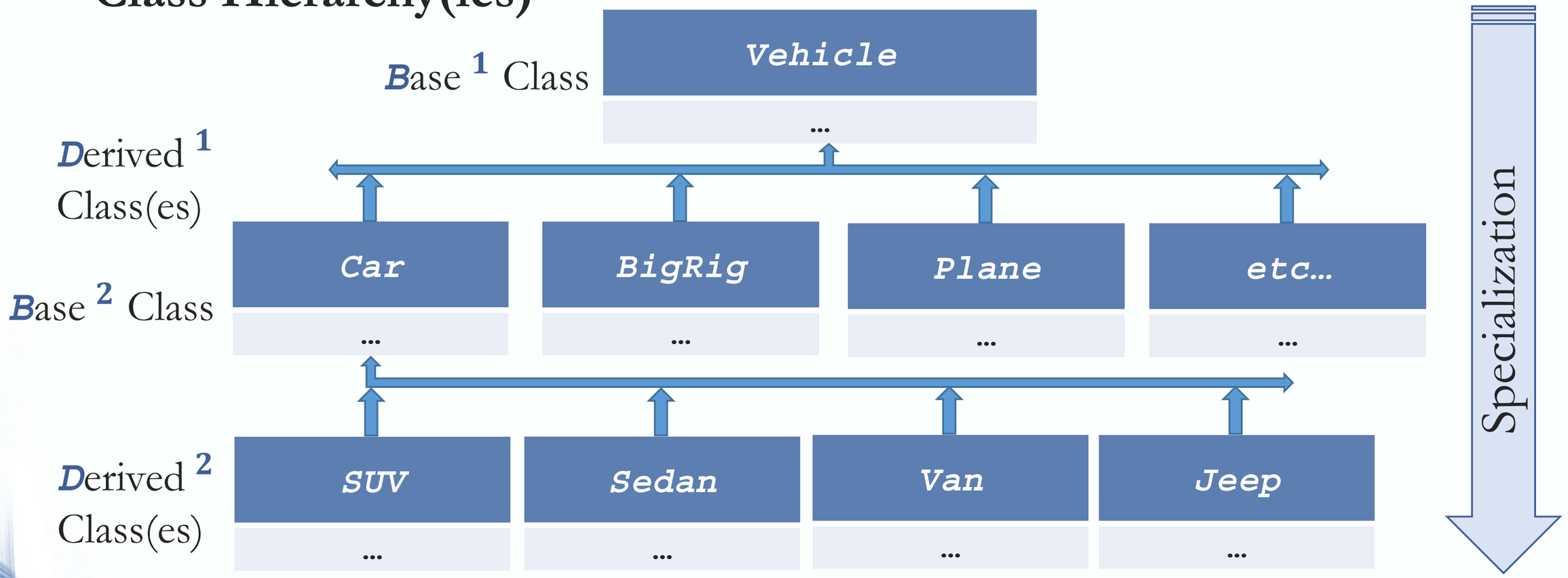
- Only Derived(Child) and its own Derived(Children), are aware that they Inherit from Base.

`Private`

- No one other than Derived(Child) is aware of the Inheritance.

Inheritance

Class Hierarchy(ies)



Class Hierarchy(ies)

More general Class (e.g. `Vehicle`) is called:

- Base Class
- Parent Class
- Super-Class

The more specialized Class (e.g. `Car`) is called:

- Derived Class
- Child Class
- Sub-Class

Class Hierarchy(ies)

Parent/Base Class:

- Contains all that is common among its child classes (less specialized).

Example:

A `Vehicle` has members like max speed, weight, etc. because all vehicles have these.

Member Variables and Functions of the Parent/Base Class are Inherited:

- By all its Child/Derived Classes (Inherited *doesn't necessarily* mean directly accessible!)

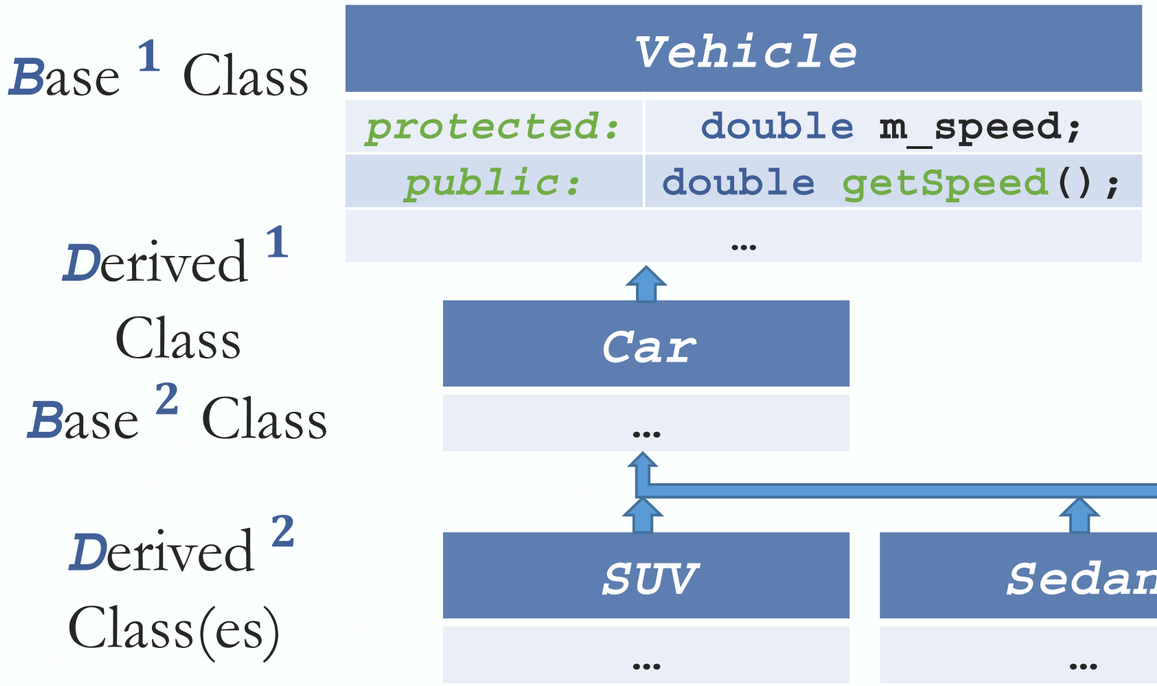
Note: Parent/Base Class `protected` (and of course any `public`) Member Variables:

- Directly accessible by Derived/Child Class.

Class Hierarchy(ies)

Derived/Child Class has access to all `public` Methods of Base/Parent Class.

- Can be used on Derived/Child Class Objects!




```
Car myCar;
SUV mySUV;
cout << myCar.m_speed;
cout << myCar.getSpeed();
cout << mySUV.m_speed;
cout << mySUV.getSpeed();
```

Class Hierarchy(ies)

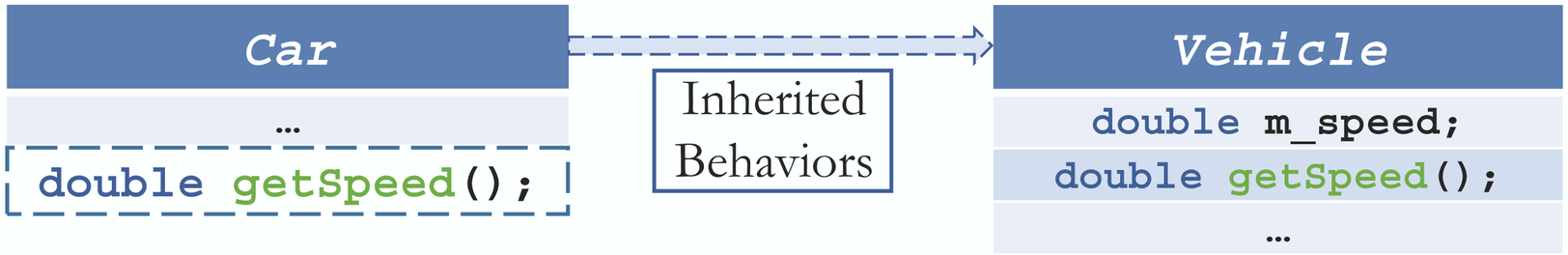
Derived/Child Class has access to all `public` Methods of Base/Parent Class.

- Can be used on Derived/Child Class Objects!
- Derived/Child Classes can *Use*, *Extend*, or *Replace* the Base/Parent Class behaviors.

Use

Derived/Child Class takes advantage of the Parent Class behaviors exactly as they are:

- E.g. Mutators and Accessors from the Parent Class.



Class Hierarchy(ies)

Derived/Child Class has access to all `public` Methods of Base/Parent Class.

- Can be used on Derived/Child Class Objects!
- Derived/Child Classes can *Use*, *Extend*, or *Replace* the Base/Parent Class behaviors.

Extend

Derived/Child Class creates entirely new behaviors:

- E.g. A `repaintCar()` function for the `Car` Child Class.
  Sets of Mutators & Accessors for new Member Variables.

`Car`
`double m_steeringWheelAngle;`
`double getSteeringWheelAngle();`
…

Own more specialized behaviors

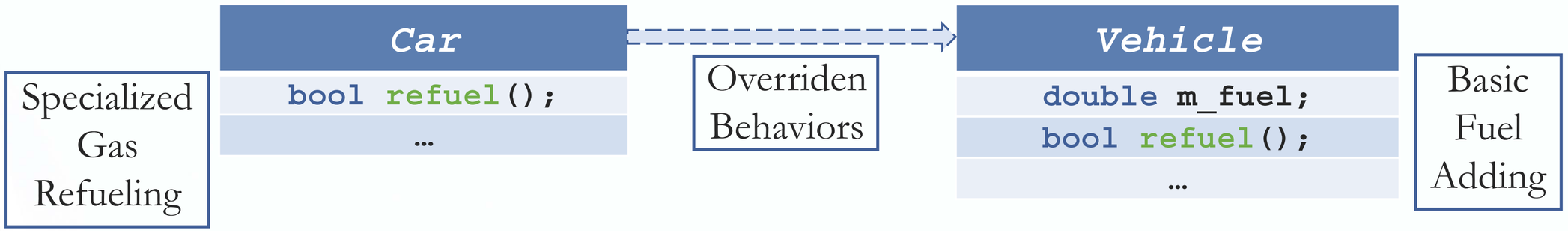
Class Hierarchy(ies)

Derived/Child Class has access to all `public` Methods of Base/Parent Class.

- Can be used on Derived/Child Class Objects!
- Derived/Child Classes can *Use*, *Extend*, or *Replace* the Base/Parent Class behaviors.

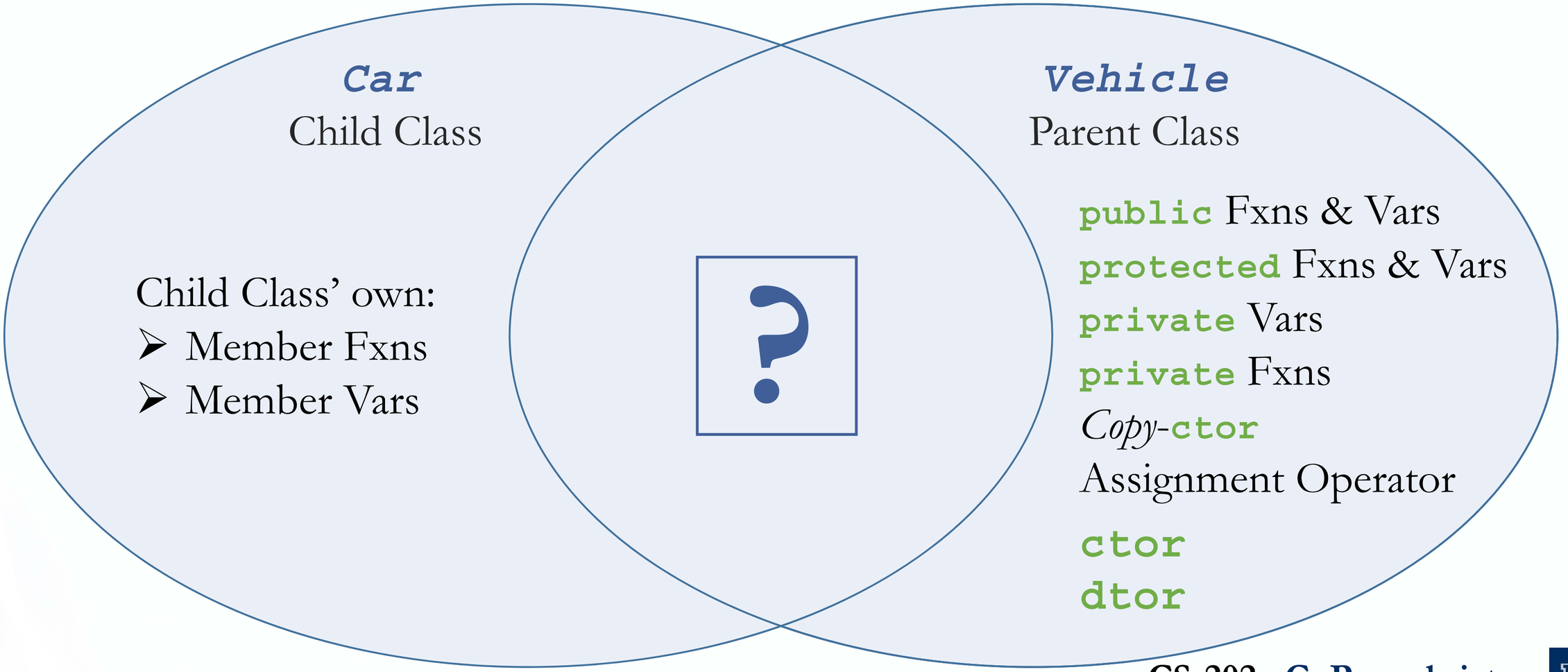
Replace

Derived/Child Class `override`s Base/Parent Class's behaviors.



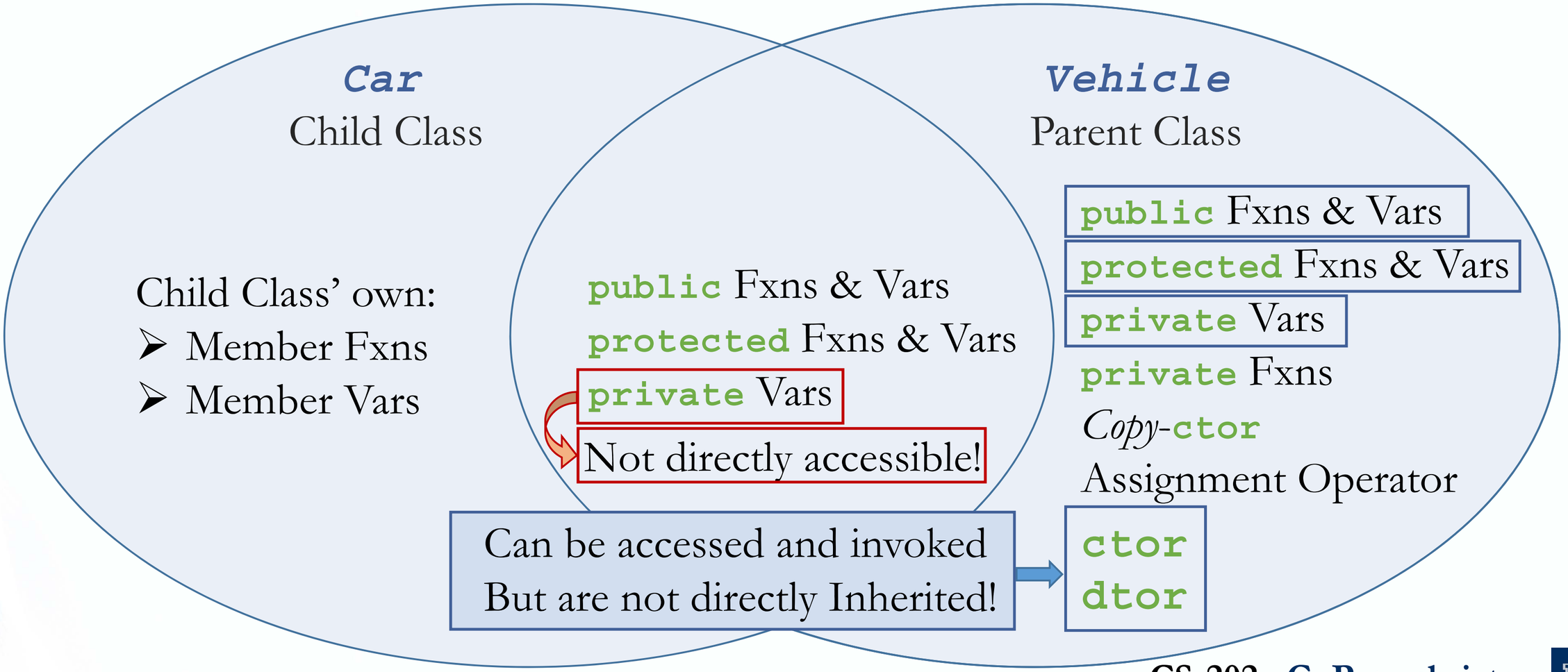
Inheritance

Inherited Member(s)



Inheritance

Inherited Member(s)



Handling Access

Derived/Child Class has access to Base/Parent Class's:

- `protected` Member Variables/Functions.
- `public` Member Variables/Functions (as everything else also does).

No access to Base/Parent Class's `private` Member Variables/Functions:

- Not even through Derived/Child Class' own Member Function.

Remember:

`private` Member Variables are only directly accessible ("by name") in Member Functions of their own Class (they one they are defined in).

Handling Access

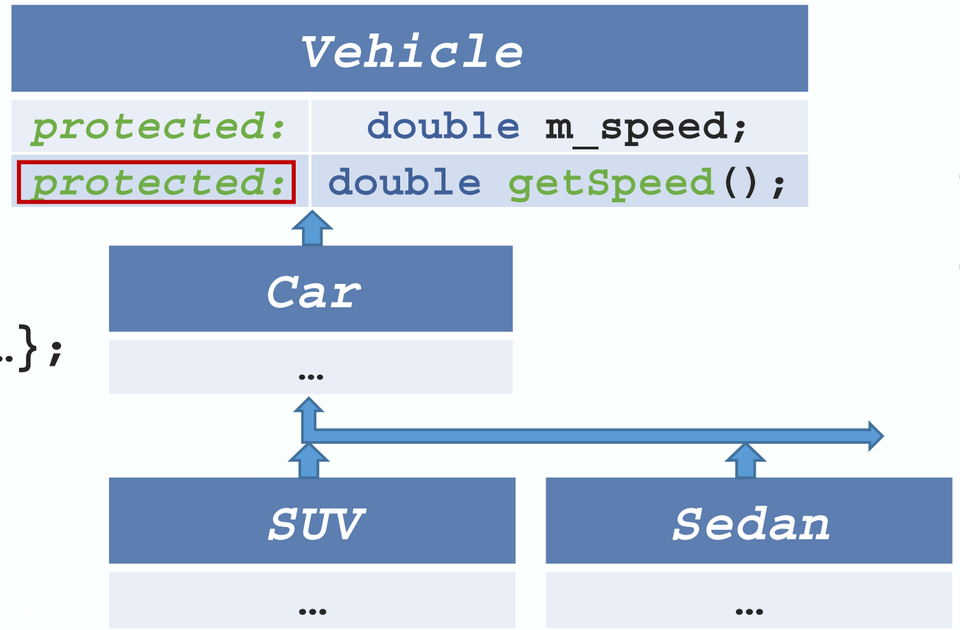
Only Derived/Child Class has access to Base/Parent Class's:

- `protected` Member Variables/Functions.

```
class Vehicle{…};

class Car :
  public Vehicle{…};

class SUV :
  public Car{…};
```




```
Car myCar;
SUV mySUV;

cout << myCar.getSpeed();

cout << mySUV.getSpeed();
```

`protected` specifier does not allow access from outside of Derived/Child Class Functions

Handling Access

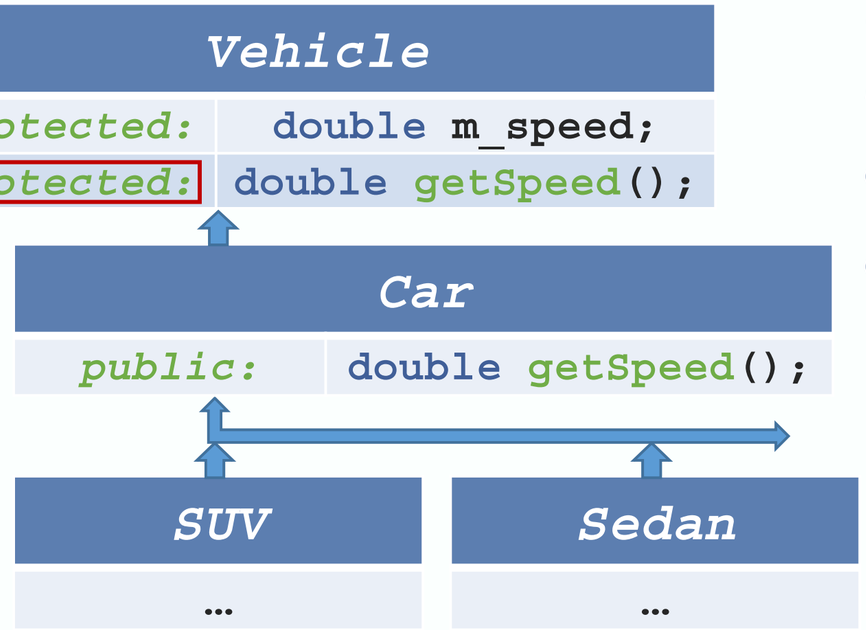
Derived/Child Class can override access specification(s) of Base/Parent Class's:

- `protected` Member Variables/Functions.

```
class Vehicle{…};
```

```
class Car :
  public Vehicle{…};
```

```
class SUV :
  public Car{…};
```




```
Car myCar;
SUV mySUV;
cout << myCar.getSpeed();
cout << mySUV.getSpeed();
```

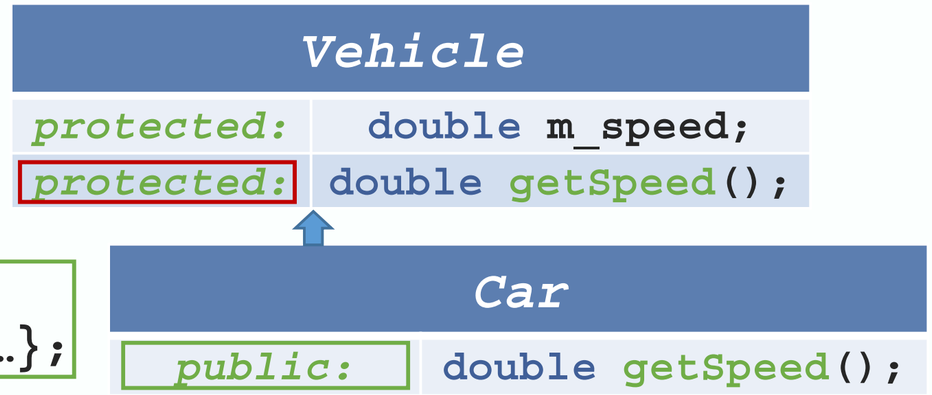
Child Class overrides `protected` access specifier to `public`, Derived Class(es) Inherit new behavior.

Handling Access

Derived/Child Class can override access specification(s) of Base/Parent Class's:

- `protected` Member Variables/Functions.

```
class Vehicle{…};
```




```
class Car :
  public Vehicle{…};
```

```
Car myCar;
SUV mySUV;
cout << myCar.getSpeed();
```

Child Class overrides `protected` access specifier to `public`, Derived Class(es) Inherit new behavior.

Note: You can even call the Base Class' method inside your Derived Class' one which overrides it (essentially override only access specification)

```
Vehicle::getSpeed(){ return m_speed; }
Car::getSpeed(){ return getSpeed(); }
```

Handling Access

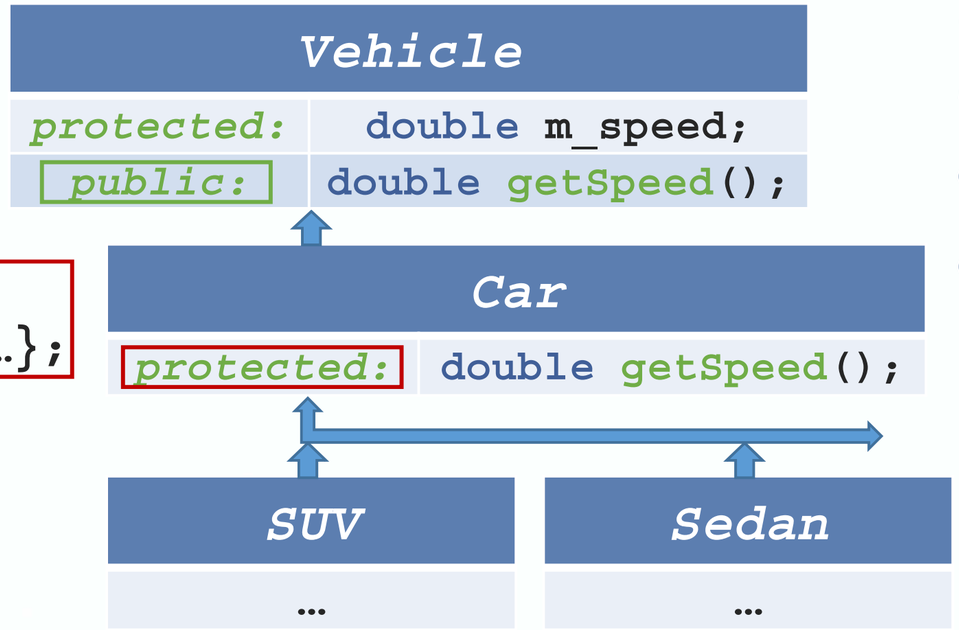
Derived/Child Class can override access specification(s) of Base/Parent Class's:

- `protected` Member Variables/Functions.

```
class Vehicle{…};
```

```
class Car :
  public Vehicle{…};
```

```
class SUV :
  public Car{…};
```




```
Car myCar;
SUV mySUV;
cout << myCar.getSpeed();
cout << mySUV.getSpeed();
```

Child Class overrides `public` access specifier to `protected`, Derived Class(es) Inherit new behavior.

Method Overriding

Overriding

Remember: Interface of a Derived/Child Class:

- *Extends*: Contains declarations for its own new Member Functions.
- *Overrides*: Contains declarations for Inherited Member Functions to be changed.

Implementation of a Derived/Child Class will:

- Define new Member Functions.
- Redefine Inherited Functions *when you Declare them*!

```
class Vehicle {
  public:
    int getMileage(){ return m_mileage; }
  private:
    int m_mileage;
} ;
```

```
class Car : public Vehicle {
  public:
    int getMileage();
} ;
```

Now that you re-Declared it, you have to Define it!

Method Overriding

Overriding *vs* **Overloading**

Overriding in a Derived/Child class means "*Redefining what it does*":

- The same parameters list.
- Essentially "crossing-out & re-writing" what the one-and-same function does!
- Overridden functions share the same signature (because they are one function)!

Overloading a Function means "*Reusing its name*":

- Using a different parameter(s) list.
- Essentially defining a "new version of" a function (that takes different parameters).
- Overloaded functions must have different signatures!

Method Overriding

Overriding *vs* **Overloading**

Overriding in a Derived/Child class means "*Redefining what it does*":

- Overridden functions share the same signature (because they are one function)!

Overloading a Function means "*Reusing its name*":

- Overloaded functions must have different signatures!

Function "*Signature*":

- The *unqualified* name of the function.
- The specific sequence of types (names are irrelevant) in parameters list (including order, number, types).
- Signature does NOT include: `return` type (not always – but it's a later encountered issue), `const` keyword or `&` for parameters.
- Signature DOES include: `cv`-qualifiers (e.g. `const` keyword at the end)

Method Overriding

Overriding *vs* Overloading

Method Overriding (uses exact *same signature*):

- Derived Class Method can modify, add to, or replace Base Class methods.
- Derived Method will be called for Derived Objects.
- Base Method will be called for Base Objects.

```
class Animal {
  public:
    void eat(){
      cout<<"I eat stuff"<<endl;
    }
};
class Lion : public Animal {
  void eat(){
    cout<<"I eat meat"<<endl;
  }
};
```

```
int main(){
  Animal animal;
  animal.eat();  // I eat stuff
  Lion lion;
  lion.eat();    // I eat meat
}
```

Method Overriding

Overriding *vs* Overloading

Method Overloading (uses exact *different signature*):

- A different function (which however carries the same name!)
- Derived/Child Class has access to both functions.

```
class Animal {
  public:
    void eat(){
      cout<<"I eat stuff"<<endl;
    }
};
class Lion : public Animal {
  public:
    void eat(const char* food)
      cout<<"I ate a "<<food<<endl;
    }
};
```

```
int main(){
  Lion lion;
  lion.eat();          // I eat stuff
  lion.eat("Steak"); // I ate a Steak
}
```

Inheritance Rules - Further

All "normal" functions in Base/Parent class are Inherited in Derived/Child Class.
Inheritance exceptions to the rules so far are:

- Constructor(s) - `ctor`
- Destructor(s) - `dtor`
- *Copy*-`ctor`
  If none is specified for Derived Class, compiler will still generate an "automatically synthesized" one.
- Assignment Operator (`=`)
  If none is specified for Derived Class, compiler will still generate an "automatically synthesized" one (which will also invoke the Base Class assignment `operator=`) .
  If one is implemented for the Derived Class that one will be called alone .

Inheritance Rules

Assignment `operator=` in Derived Class(es)

Assignment Operator (**=**)

- If none is specified for Derived Class, compiler will still generate an "automatically synthesized" one (which will also invoke the Base Class assignment `operator=`) .
- If one is implemented for the Derived Class that one will be called alone .

We can however also directly call the Base Class Assignment from the Derived Class one !

```
class Animal{
  Animal& operator=(const Animal& rhs_animal ){
    cout << "Animal assignment" << endl;
  }
};
class Lion : public Animal{
  Lion& operator=(const Lion& rhs_lion ){
    Animal::operator=( rhs_lion );
    cout << "Lion assignment" << endl;
  }
};
```

Inheritance Rules

Constructor & Destructor in Derived Class(es)

The Base/Parent Class `ctor`s are *not* Inherited in Derived(Child) Classes.

- They can however be invoked within Derived(Child) Class' `ctor`.

Nothing more is required:

- A Base(Parent) Class `ctor` must instantiate all Base Class Member Variables.
- These are Inherited by Derived(Child) Class!

The "First thing" that *any* Derived(Child) Class `ctor` does is to try and invoke the Base(Parent) Class *Default* `ctor` !

- Unless we otherwise explicitly specify to call another Base(Parent) Class `ctor` .
  (We will see how in a little while…)

Inheritance Rules

Constructor & Destructor in Derived Class(es)

Constructor(s)

- Base/Parent Class `ctor` is called *before* Derived/Child Class `ctor`.

```
DerivedClass dClass;
```

```
BaseClass();
DerivedClass();
```

DerivedClass	→	BaseClass
DerivedClass();		BaseClass();
…		…

Destructor

- Derived/Child Class `dtor` is called *before* Base/Parent Class `dtor`.

```
dClass.~DerivedClass();
```

```
~DerivedClass();
~BaseClass();
```

DerivedClass	→	BaseClass
~DerivedClass();		~BaseClass();
…		…

Inheritance Rules

Constructor & Destructor in Derived Class(es)

Sequence of Base-Derived `ctor` & Derived-Base `dtor` calls

Example:

```
class Animal {
  public:
    Animal(){cout << "Base constructor"<<endl;}
    ~Animal(){cout << "Base destructor"<<endl;}
};
class Lion : public Animal {
 public:
    Lion(){cout << "Derived constructor"<<endl;}
    ~Lion(){cout << "Derived destructor"<<endl;}
};
```

```
int main(){
  Lion lion;
  return 0;
}
```

Output:

```
Base constructor
Derived constructor
Derived destructor
Base destructor
```

Inheritance Rules

Parametrized Constructor in Derived Class(es)

Calling the Base `ctor` from a Derived `ctor` explicitly

Example:

```
class Animal {
  public:
    Animal(const char* name) {
       strcpy(m_name, name);
       cout<<m_name<<endl;
     }
  protected:
     char m_name[MAXNAME];
};
class Lion : public Animal {
  public:
    Lion(const char* name) : Animal(name) {
     }
};
```

```
int main(){
   Lion lion("King");
   return 0;
}
```

Output:

```
King
```

Note: Initializer-list is the only way that allows calling the Base `ctor` from a Derived `ctor`.

Inheritance Rules

Parametrized Constructor in Derived Class(es)

Calling the Base `ctor` from a Derived `ctor` explicitly

Example:

```
class Animal {
  public:
    Animal(const char* name) {
       strcpy(m_name, name); cout<<m_name<<endl;
    }
  protected:
    char m_name[MAXNAME];
};
class Lion : public Animal {
  public:
    Lion(const char* name): Animal(name) {
       strcat(m_name," Lion"); cout<<m_name<<endl;
    }
};
```

```
int main(){
  Lion lion("King");
  return 0;
}
```

Output:

```
King
King Lion
```

Note: Calls Parametrized Base `ctor` by passing down argument from the Derived `ctor`.

Polymorphism (prelude)

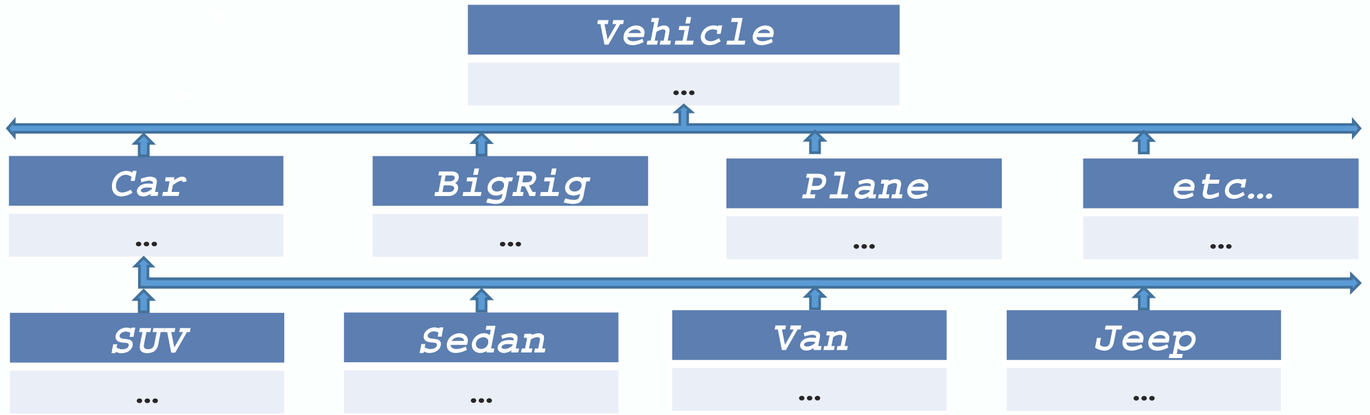
Inheritance & Polymorphism

Polymorphism means "the ability to take many forms".

- Allowing a single (*Remember*: Overriding *vs* Overloading) behavior to take on many type-dependent forms.
- Hence, grants the ability to manipulate Objects in a type-independent way.

Pointers of Base Class-type:

- They are used to address "*Common Ancestor*"-type Objects from a Class Hierarchy.

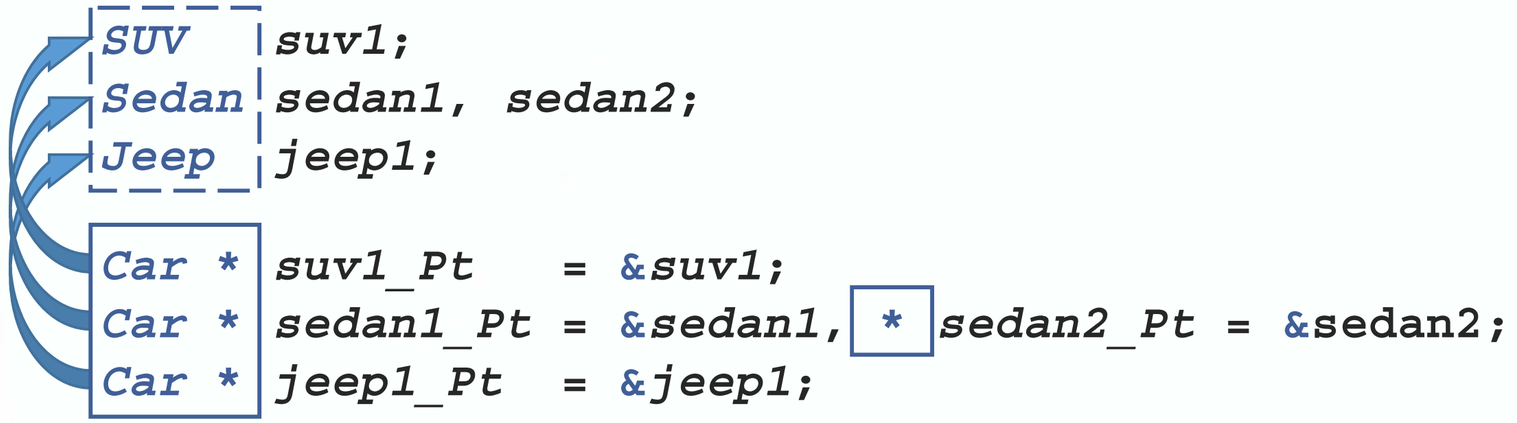


Polymorphism (prelude)

Inheritance & Polymorphism

Base Class-type Pointers :

- A Pointer of a Base Class type can point to an Object of a Derived Class type.



```
SUV   suv1;
Sedan sedan1, sedan2;
Jeep  jeep1;

Car * suv1_Pt   = &suv1;
Car * sedan1_Pt = &sedan1, * sedan2_Pt = &sedan2;
Car * jeep1_Pt  = &jeep1;
```

This is valid: A Derived Class (***SUV***, ***Sedan***, ***Van***, ***Jeep***) "is a type of" Base Class (***Car***).

- *Note*: A 1-way relationship:

  A Derived Class Pointer cannot point to a Base Class Object.

Advanced Examples

Derived to Base Class forwarding

Parent Class "hides" data even from Children (but provides an *Interface* to them!) :

```
class Animal {
  friend ostream& operator<<(ostream& os, const Animal& animal){
    os << "I have "  << aanimal.m_legs << " legs" << endl;
    return os;
  }
  private:
    int m_legs;
};
class Lion : public Animal {
  friend ostream& operator<<(ostream& os, const Lion& lion){
    os << "I am a "  << lion.m_color << " lion and\n"
       << "I have "  << lion.m_legs << " legs" << endl;
    return os;
  }
  private:
    char m_color[256];
};
```

Advanced Examples

Derived to Base Class forwarding

Parent Class "hides" data even from Children (but provides an *Interface* to them!) :

```
class Animal {
  friend ostream& operator<<(ostream& os, const Animal& animal){
    os << "I have "  << aanimal.m_legs << " legs" << endl;
    return os;
  }
  private:
    int m_legs;
};
class Lion : public Animal {
  friend ostream& operator<<(ostream& os, const Lion& lion){
    os << "I am a "  << lion.m_color << " lion and\n"
       << "I have "  << lion.m_legs << " legs" << endl;
    return os;
  }
  private:
    char m_color[256];
};
```

Not possible !
The function is a **friend** of ***Lion***,
not a **friend** of ***Animal*** !

Advanced Examples

Derived to Base Class forwarding

Parent Class "hides" data even from Children (but provides an *Interface* to them!) :

```
class Animal {
  friend ostream& operator<<(ostream& os, const Animal& animal){
    os << "I have "  << animal.m_legs << " legs" << endl;
    return os;
  }
  private:
    int m_legs;
};
class Lion : public Animal {
  friend ostream& operator<<(ostream& os, const Lion& lion){
    os << "I am a "  << lion.m_color << " lion and\n"
       << static_cast< const Animal& >( lion );
    return os;
  }
  private:
    char m_color[256];
};
```

Casting to a *Reference* of Base class type invokes the corresponding function overload for ***Animal*** !

CS-202

Time for Questions !