CIS 190: C/C++ Programming

Lecture 10

Inheritance

Outline

- Code Reuse
- Object Relationships
- Inheritance
  - What is Inherited
  - Handling Access
- Overriding

Code Reuse

- important to successful coding

- efficient
  - no need to reinvent the wheel
- error free (more likely to be)
  - code has been previously used/test

Code Reuse Methods

- functions
- classes
- inheritance
  - what we'll cover now

Outline

- Code Reuse
- Object Relationships
- Inheritance
  - What is Inherited
  - Handling Access
- Overriding

Object Relationships

- two types of object relationships

- is-a
  - inheritance

- has-a
  - composition
  - aggregation

} both are forms of association

Inheritance Relationship

a Car **is-a** Vehicle

- the Car class **inherits** from the Vehicle class

- Vehicle is the general class, or the **parent class**
- Car is the specialized class, or **child class**, that is a subclass of Vehicle

Inheritance Relationship Code

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

Inheritance Relationship Code

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

Inheritance Relationship Code

```
class Truck:
  public Vehicle { /*etc*/ };
class Plane:
  public Vehicle { /*etc*/ };
class UnmannedDrone:
  public Vehicle { /*etc*/ };
class SpaceShuttle:
  public Vehicle { /*etc*/ };
class Submarine:
  public Vehicle { /*etc*/ };
```

Composition Relationship

a Car **has-a** Chassis

- the Car class **contains** an object of type Chassis

- a Chassis object is part of the Car class
- a Chassis cannot “live” out of context of a Car
  - if the Car is destroyed, the Chassis is also destroyed

Composition Relationship Code

```
class Chassis {
  public:
    //functions
  private:
    string m_material;
    double m_weight;
    double m_maxLoad;
    // etc
} ;
```

Composition Relationship Code

```
class Car: public Vehicle {
  public:
    //functions
  private:
    // member variables, etc.

    // has-a (composition)
    Chassis m_chassis;
} ;
```

Aggregation Relationship

a Car **has-a** Person (driver)

- the Car class is **linked to** an object of type Person

- the Person class is not related to the Car class
- a Person **can** live out of context of a Car
- a Person must be "contained" in the Car object <u>via a pointer</u> to a Person object

Aggregation Relationship Code

```
class Person {
   public:
      // functions
   private:
      string m_firstName;
      string m_lastName;
      double m_height;
      double m_weight;
      // etc
} ;
```

Aggregation Relationship Code

```
class Car: public Vehicle {
  public:
    //functions
  private:
    // member variables, etc.

    // has-a (aggregation)
    Person *m_driver;
} ;
```

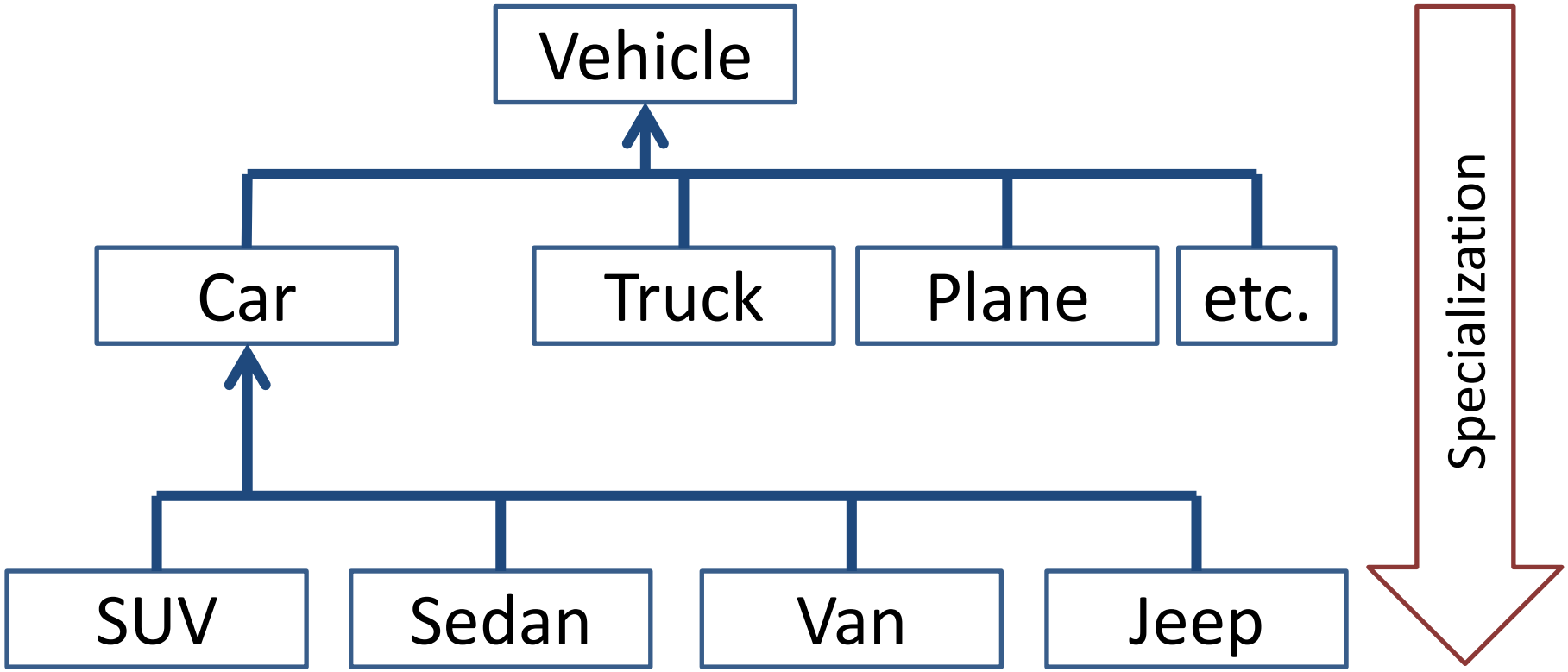
Outline

- Code Reuse
- Object Relationships
- Inheritance
  - What is Inherited
  - Handling Access
- Overriding

Inheritance Access

- inheritance can be done via public, private, or protected
  - like member functions and member variables
- we're going to focus exclusively on public inheritance
- you can also have multiple inheritance; we won't be covering it

Hierarchy Example



Hierarchy Vocabulary

- **more general class** (e.g., Vehicle) can be called:
  - parent class
  - base class
  - superclass
- **more specialized class** (e.g., Car) can be called:
  - child class
  - derived class
  - subclass

Hierarchy Details

- parent class contains all that is common among its child classes
  - Vehicle has a maximum speed, a weight, etc. because all vehicles have these
- member variables and functions of the parent class are inherited by all of its child classes
- child classes can use, extend, or replace the parent class behaviors

Hierarchy Details

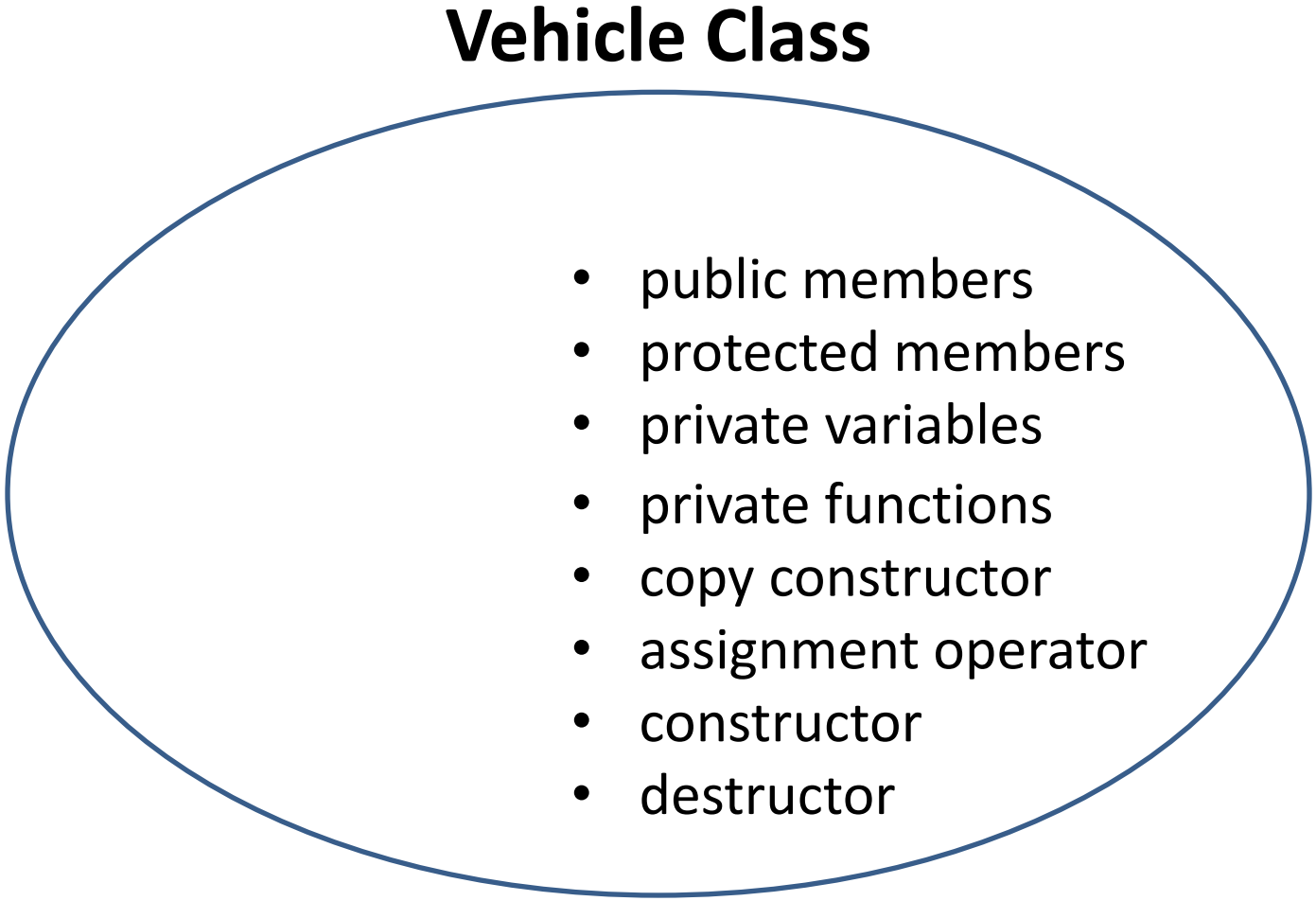
- use, extend, or replace base class behaviors

- use
  - entirely unchanged (e.g., mutators, accessors, etc.)
- extend
  - create entirely new behaviors (e.g., RepaintCar(), new mutators/accessors, etc.)
- replace
  - overriding functions (covered later)

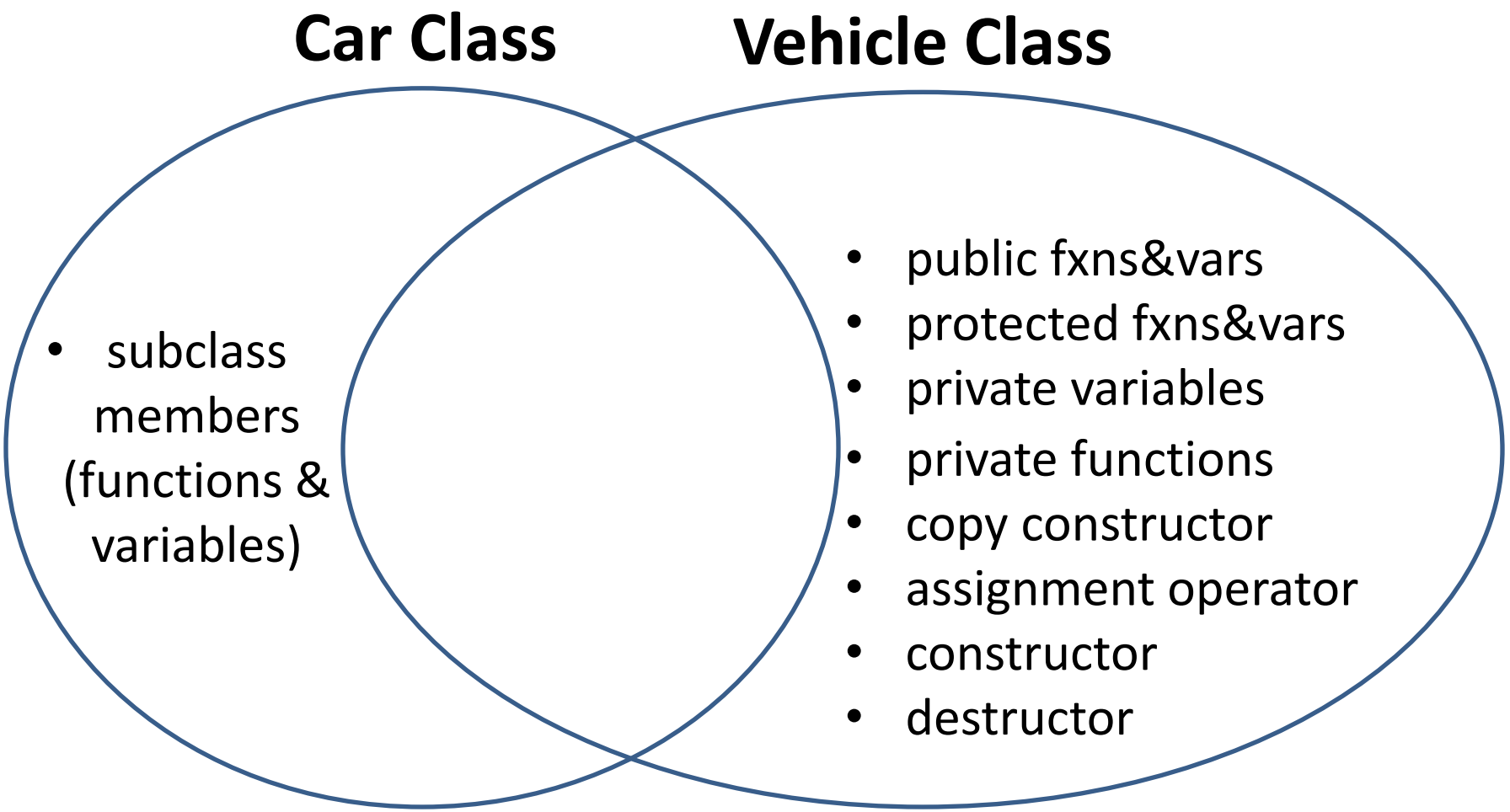
Outline

- Code Reuse
- Object Relationships
- Inheritance
  - What is Inherited
  - Handling Access
- Overriding

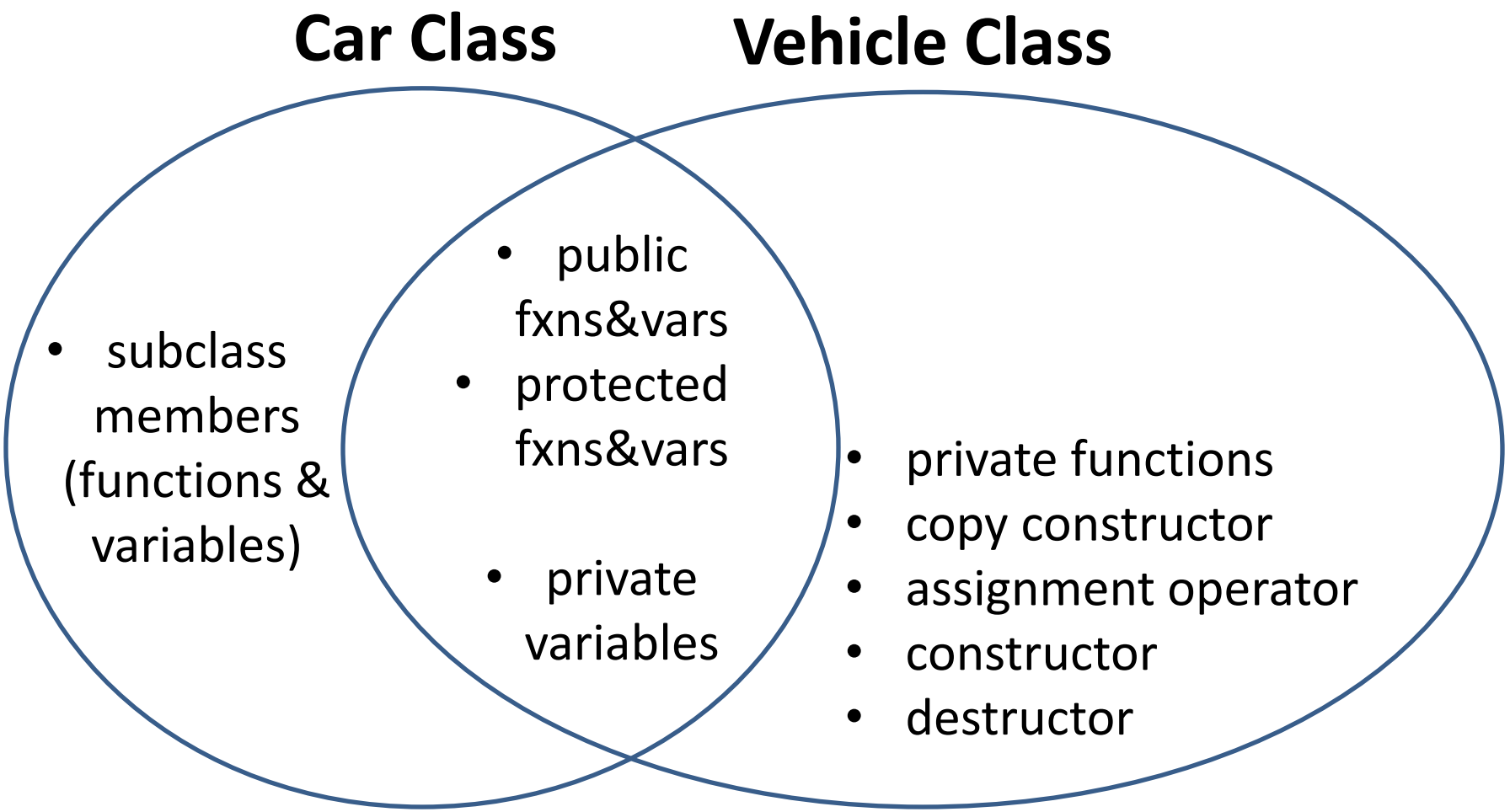
What is Inherited



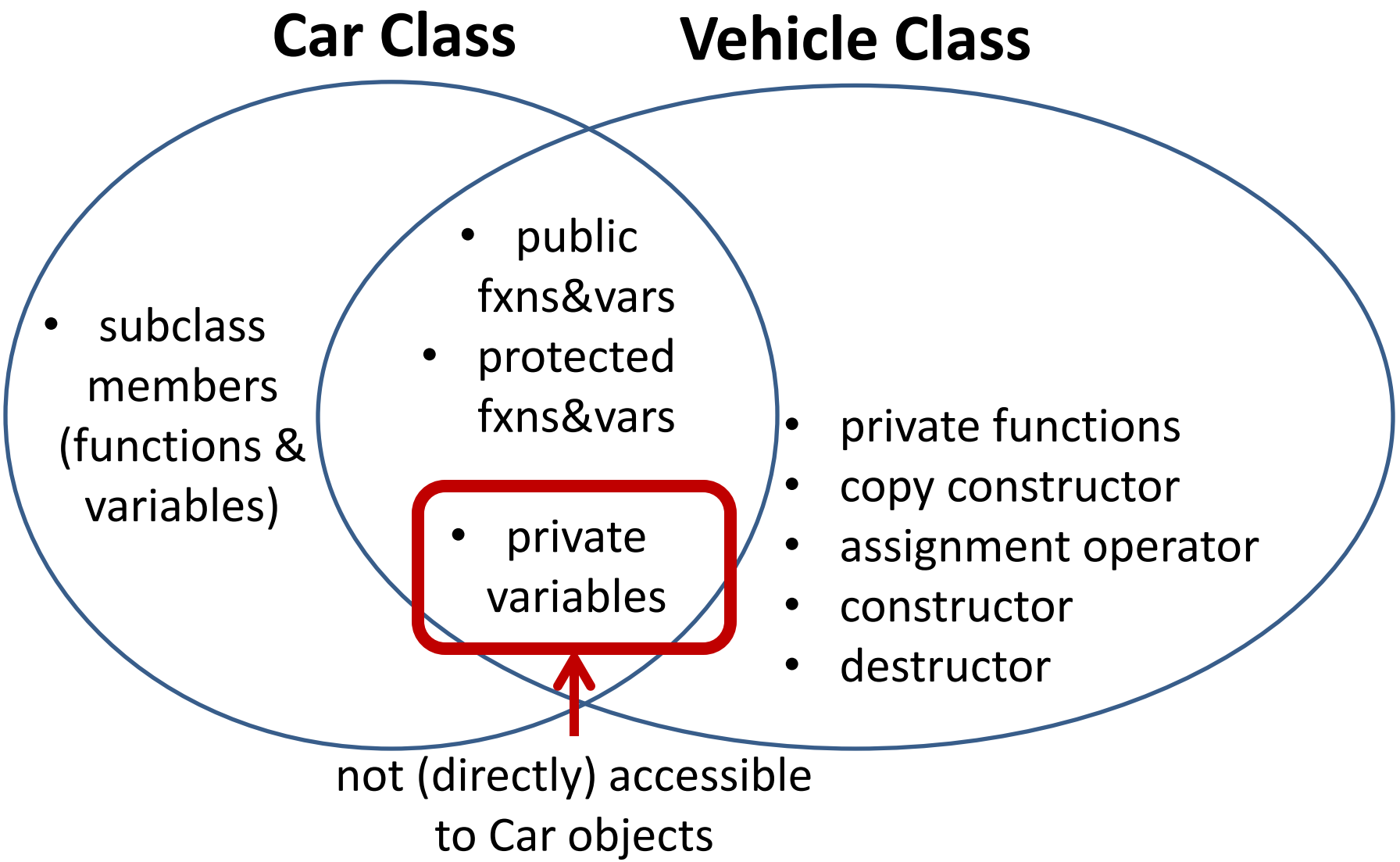
What is Inherited



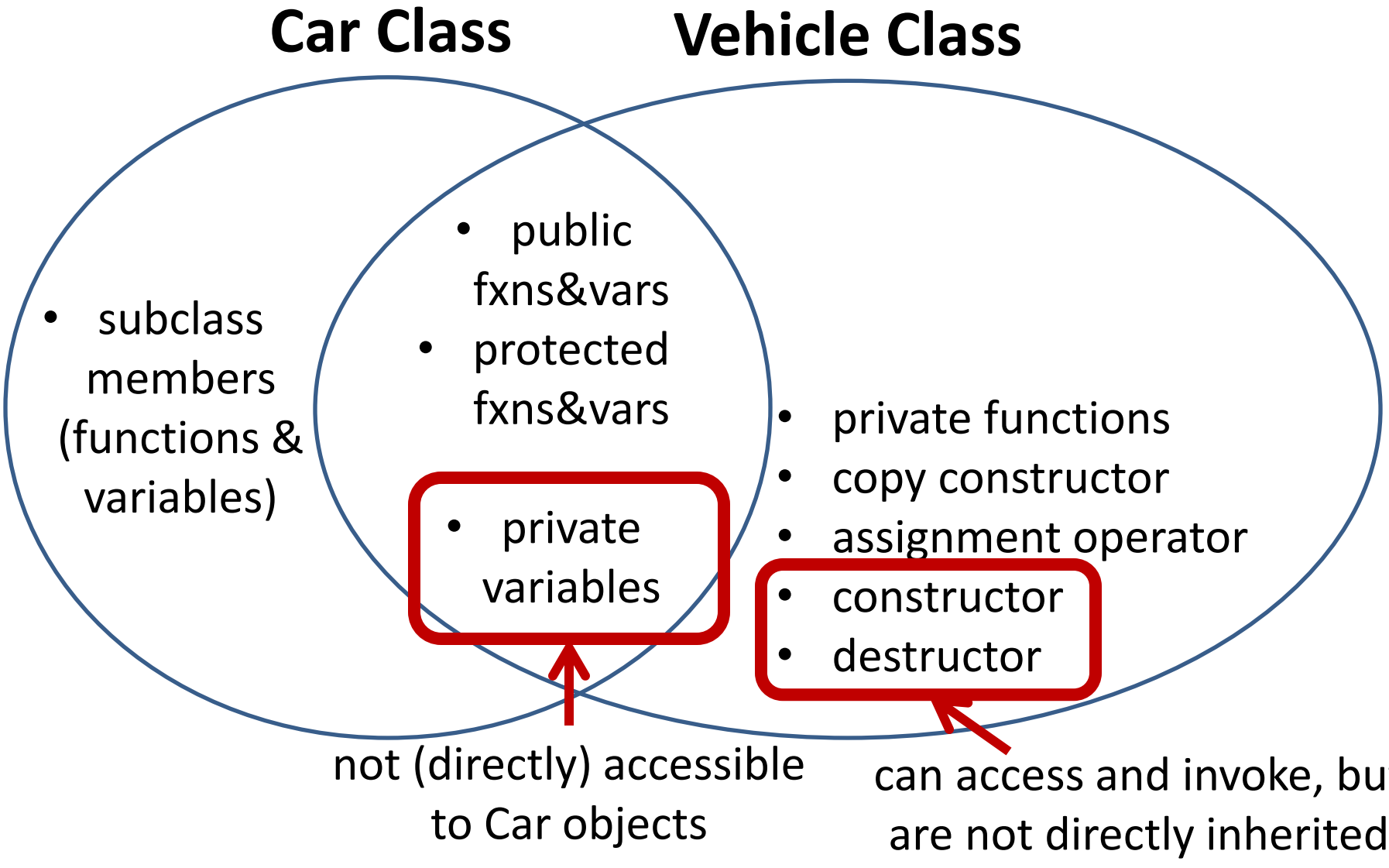
What is Inherited



What is Inherited



What is Inherited



Outline

- Code Reuse
- Object Relationships
- Inheritance
  - What is Inherited
  - Handling Access
- Overriding

Handling Access

- child class has access to parent class's:
  - public member variables
  - public member functions
  - protected member variables
  - protected member functions

- how should we set the access modifier for variables we want the child class to access?

Handling Access

- we should not make these variables protected!

- leave them private!
- instead, child class uses protected functions when interacting with parent variables
  - mutators
  - accessors

Outline

- Code Reuse
- Object Relationships
- Inheritance
  - What is Inherited
  - Handling Access
- Overriding

Specialization

- child classes are meant to be more specialized than parent classes
  - adding new member functions
  - adding new member variables

- child classes can also specialize by overriding parent class member functions
  - child class uses **exact same function signature**

Overriding vs Overloading

- overloading
  - use the same function name, but with different parameters for each overloaded implementation
- overriding
  - use the same function name and parameters, but with a different implementation
  - child class method “hides” parent class method
  - **only possible by using inheritance**

Overriding/Overloading Examples

- Vehicle class contains these public functions

```
void Upgrade();
void PrintSpecs();
void Move(double distance);
```

- Car class inherits all of these public functions
  - can therefore override them

Overriding Example

- Car class overrides Upgrade()

```
void Car::Upgrade()
{
   // entirely new Car-only code
}
```

- when Upgrade() is called on a object of type Car, the Car::Upgrade() function is invoked

Overriding (and Calling) Example

- Car class overrides **and calls** PrintSpecs()

```
void Car::PrintSpecs()
{
  Vehicle::PrintSpecs();
  // additional Car-only code
}
```

- can explicitly call a parent's function by using the scope resolution operator

Attempted Overloading Example

- Car class attempts to **overload** the function Move(double distance) with new parameters

```
void Car::Move(double distance,
               double avgSpeed)
{
   // new overloaded Car-only code
}
```

- but this won't work the way we expect!

Precedence

- **overriding takes precedence over overloading**
  - instead of overloading the Move() function, the compiler assumes we are trying to override it

- declaring `Car::Move(2 parameters)`
- overrides `Vehicle::Move(1 parameter)`

- we no longer have access to the original Move() function from the Vehicle class

Overloading in Child Class

- must have both original and overloaded functions in child class

```
void Car::Move(double distance);
void Car::Move(double distance,
               double avgSpeed);
```

- "original" one parameter function can then explicitly call parent function

Homework 6

- check validity of input values
- acceptable does not mean guaranteed!

- be extra careful with following coding standards, and making appropriate decisions

- any questions?

Project

- proposal due next week **in class**
- alphas due 1 ½ weeks after proposal

- **please don't turn in anything late!**

- will grade **last** submission from group members for alpha and project