

CS100

Introduction to Programming

Lecture 15. Object-Oriented Programming: Polymorphism

Refresher :Object Relationships

- Two types of object relationships
- The “is-a” relationship
 - inheritance
- The “has-a” relationship
 - composition
 - aggregation

} both are forms of association

Refresher: Inheritance Relationship

a Car *is-a* Vehicle

- this is called *inheritance*


Refresher: 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 inherits from Vehicle

Inheritance Relationship Code

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



all Vehicles have
axles, wheels, a
max speed, and a
weight

Inheritance Relationship Code

```
class Car {
```

```
} ;
```

Inheritance Relationship Code

```
class Car: public Vehicle {
```



Car inherits from
the Vehicle class

```
} ;
```

Inheritance Relationship Code

```
class Car: public Vehicle {
```



Car inherits from
the Vehicle class


The diagram consists of a blue bracket under the text 'public Vehicle' in the code snippet above. A vertical blue arrow points from the text 'don't forget the colon here!' below to the colon in 'Car:' in the code snippet above.

don't forget the
colon here!

```
} ;
```


Inheritance Relationship Code

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



all Cars have a
number of seats, a
MPG value, a color,
and a fuel type

Today's Learning objectives

- Learn and understand different ways of polymorphism
- Get the concepts behind
 - Virtual functions
 - Abstract classes
- Learn how to use them

Outline

- Review of Inheritance
- Polymorphism
 - Limitations
 - Virtual Functions
 - Abstract Classes & Function Types
 - Virtual Function Tables
 - Virtual Destructors/Constructors

Review of Inheritance

- specialization through sub classes
- child class has direct access to
 - parent member functions and variables that are
 - ???

Review of Inheritance

- specialization through sub classes
- child class has direct access to
 - parent member functions and variables that are
 - public
 - protected

Review of Inheritance

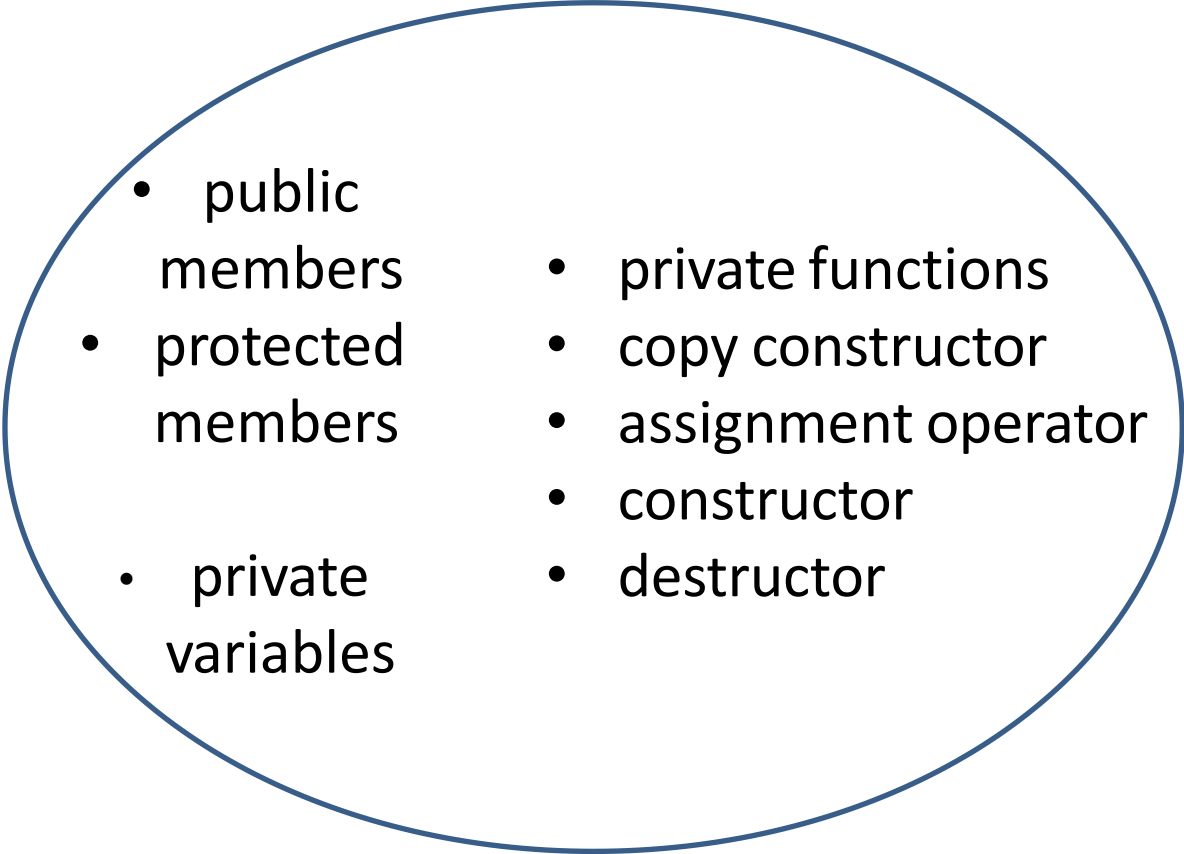
- specialization through sub classes
- child class has direct access to
 - parent member functions and variables that are:
 - public
 - protected
- parent class has direct access to:
 - ??? in the child class

Review of Inheritance

- specialization through sub classes
- child class has direct access to
 - parent member functions and variables that are:
 - public
 - protected
- parent class has direct access to:
 - **nothing** in the child class

What is Inherited

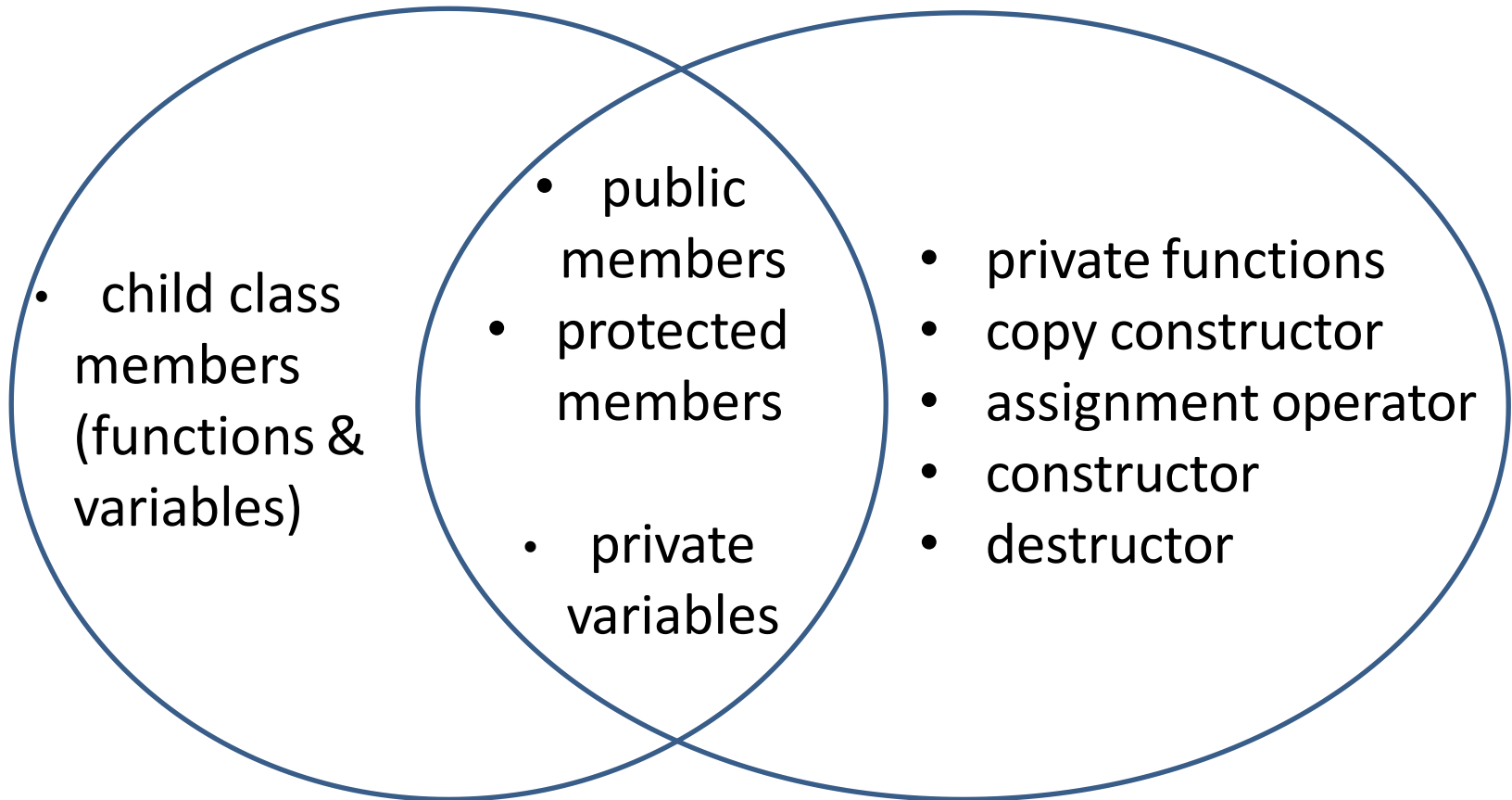
Parent Class

- 
- public members
 - protected members
 - private variables
 - private functions
 - copy constructor
 - assignment operator
 - constructor
 - destructor

What is Inherited

Child Class

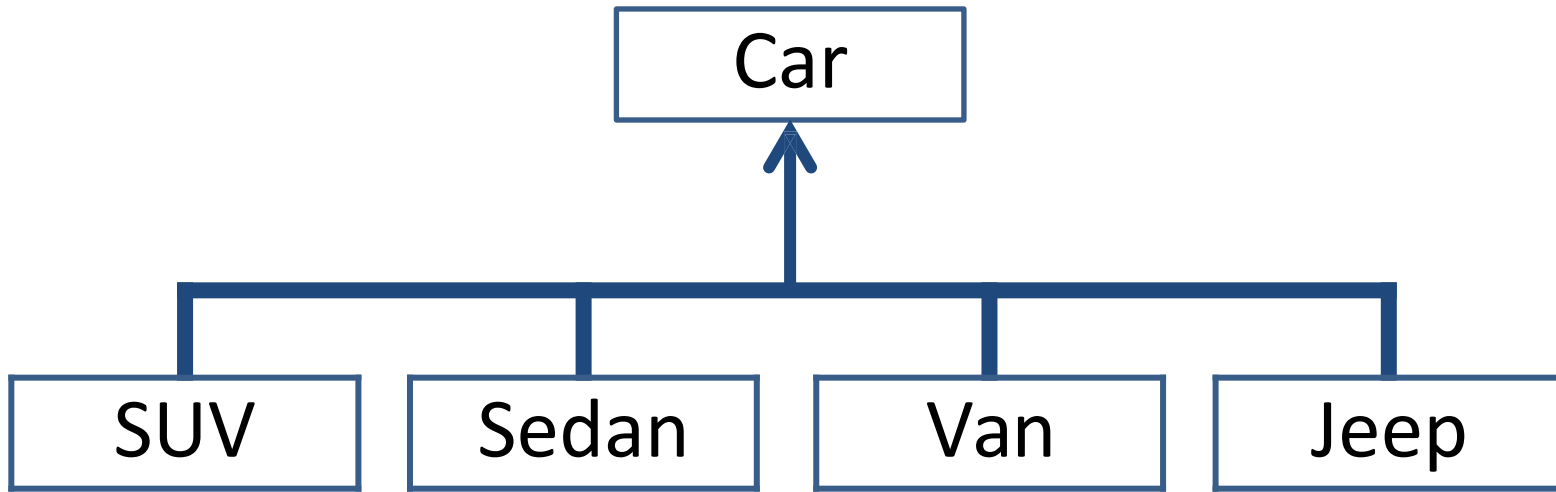
Parent Class



Outline

- Review of Inheritance
- Polymorphism
 - Limitations
 - Virtual Functions
 - Abstract Classes & Function Types
 - Virtual Function Tables
 - Virtual Destructors/Constructors

Car Example



```
class SUV:      public Car { /*etc*/ };
class Sedan:    public Car { /*etc*/ };
class Van:      public Car { /*etc*/ };
class Jeep:     public Car { /*etc*/ };
```

Car Rental Example

- we want to implement a catalog of different types of cars available for rental
- how could we do this?

Car Rental Example

- we want to implement a catalog of different types of cars available for rental
- how could we do this?
- can accomplish this with a single vector
 - using *polymorphism*

What is Polymorphism?

- ability to manipulate objects in a **type-independent** way

What is Polymorphism?

- ability to manipulate objects in a **type-independent** way
- already done to an extent via ***overriding***
 - child class overrides a parent class function

What is Polymorphism?

- ability to manipulate objects in a **type-independent** way
- already done to an extent via ***overriding***
 - child class overrides a parent class function
- can take it further using subtyping,
AKA inclusion polymorphism

Using Polymorphism

- a pointer of a parent class type can point to an object of a child class type

```
Vehicle *vehiclePtr = &myCar;
```

- why is this valid?

Using Polymorphism

- a pointer of a parent class type can point to an object of a child class type

```
Vehicle *vehiclePtr = &myCar;
```

- why is this valid?
 - because **myCar** is-a **Vehicle**

Polymorphism: Car Rental

```
vector <Car*> rentalList;
```

vector of **Car*** objects

--	--	--	--	--	--	--	--

Polymorphism: Car Rental

```
vector <Car*> rentalList;
```

vector of **Car*** objects

SUV	SUV	Jeep	Van	Jeep	Sedan	Sedan	SUV
-----	-----	------	-----	------	-------	-------	-----

- can populate the vector with any of **Car**'s child classes

Outline

- Review of Inheritance
- Polymorphism
 - Limitations
 - Virtual Functions
 - Abstract Classes & Function Types
 - Virtual Function Tables
 - Virtual Destructors/Constructors

Recall

- Class Vehicle is an example class which contains the public functions:

```
void Upgrade () ;
```

```
void PrintSpecs () ;
```

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

- Class Car overrides these functions and also extends the class by:

```
void RepaintCar () ;
```

Limitations of Polymorphism

- parent classes **do not** inherit from child classes
 - what about public member variables and functions?

Limitations of Polymorphism

- parent classes **do not** inherit from child classes
 - **not even** public member variables and functions

Limitations of Polymorphism

- parent classes **do not** inherit from child classes
 - **not even** public member variables and functions

```
Vehicle *vehiclePtr = &myCar;
```

Limitations of Polymorphism

- parent classes **do not** inherit from child classes
 - **not even** public member variables and functions

```
Vehicle *vehiclePtr = &myCar;
```

- which version of **PrintSpecs ()** does this call?
`vehiclePtr->PrintSpecs () ;`

Limitations of Polymorphism

- parent classes **do not** inherit from child classes
 - not even** public member variables and functions

```
Vehicle *vehiclePtr = &myCar;
```

- which version of **PrintSpecs ()** does this call?
`vehiclePtr->PrintSpecs () ;`

```
Vehicle::PrintSpecs ()
```

Limitations of Polymorphism

- parent classes **do not** inherit from child classes
 - **not even** public member variables and functions

```
Vehicle *vehiclePtr = &myCar;
```

- will this work?

```
vehiclePtr->RepaintCar();
```

Limitations of Polymorphism

- parent classes **do not** inherit from child classes
 - **not even** public member variables and functions

```
Vehicle *vehiclePtr = &myCar;
```

- will this work?

```
vehiclePtr->RepaintCar();
```

- NO! **RepaintCar()** is a function of the Car child class, not the Vehicle class

Outline

- Review of Inheritance
- Polymorphism
 - Limitations
 - Virtual Functions
 - Abstract Classes & Function Types
 - Virtual Function Tables
 - Virtual Destructors/Constructors

Virtual Functions

- can grant access to child methods by using *virtual functions*
- virtual functions are how C++ implements *late binding*
 - used when the child class implementation is unknown or variable at parent class creation time

Late Binding

- simply put, binding is determined at run time
 - as opposed to at compile time
- in the context of polymorphism, you're saying

I don't know for sure how this function is going to be implemented, so wait until it's used and then get the implementation from the object instance.

Using Virtual Functions

- declare the function in the parent class with the keyword **virtual** in front

```
virtual void Drive ();
```

Using Virtual Functions

- declare the function in the parent class with the keyword **virtual** in front

```
virtual void Drive ();
```

- only use **virtual** with the prototype

```
// don't do this
```

```
virtual void Vehicle::Drive ();
```

Using Virtual Functions

- the corresponding child class function does not require the **virtual** keyword
- but...

Using Virtual Functions

- the corresponding child class function does not require the **virtual** keyword
- should still include it, for clarity's sake
 - makes it obvious the function is virtual, even without looking at the parent class

```
// inside the Car class  
virtual void Drive();
```

Outline

- Review of Inheritance
- Polymorphism
 - Limitations
 - Virtual Functions
 - Abstract Classes & Function Types
 - Virtual Function Tables
 - Virtual Destructors/Constructors

Function Types – Virtual

```
virtual void Drive();
```

- parent class **must** have an implementation
 - even if it's trivial or empty
- child classes may override if they choose to
 - if not overridden, parent class definition used

Function Types – Pure Virtual

```
virtual void Drive() = 0;
```

- denote pure virtual by the “ = 0” at the end

Function Types – Pure Virtual

```
virtual void Drive () = 0;
```

- denote pure virtual by the “ = 0” at the end
- the parent class has **no implementation** of this function
 - child classes **must** have an implementation

Function Types – Pure Virtual

```
virtual void Drive () = 0;
```

- denote pure virtual by the “ = 0” at the end
- the parent class has **no implementation** of this function
 - child classes **must** have an implementation
 - parent class is now an ***abstract class***

Abstract Classes

- an ***abstract class*** is one that contains a function that is ***pure virtual***

Abstract Classes

- an ***abstract class*** is one that contains a function that is ***pure virtual***
- cannot declare abstract class objects
 - why?

Abstract Classes

- an ***abstract class*** is one that contains a function that is ***pure virtual***
- cannot declare abstract class objects
 - why?
- this means abstract classes can only be used as base classes

Applying Virtual

- Imagine a class Shape. How should we label the following functions?
(virtual, pure virtual, or leave alone)

CalculateArea () ;

CalculatePerimeter () ;

Print () ;

SetColor () ;

Outline

- Review of Inheritance
- Polymorphism
 - Limitations
 - Virtual Functions
 - Abstract Classes & Function Types
 - Virtual Function Tables
 - Virtual Destructors/Constructors

Behind the Scenes

- if our **Drive** () function is virtual, how does the compiler know which child class's version of the function to call?

vector of Car* objects

SUV	SUV	Jeep	Van	Jeep	Sedan	Sedan	SUV
-----	-----	------	-----	------	-------	-------	-----

Virtual Function Tables

- the compiler uses *virtual function tables* whenever we use polymorphism
- virtual function tables are created for:
 - what types of classes?

Virtual Function Tables

- the compiler uses *virtual function tables* whenever we use polymorphism
- virtual function tables are created for:
 - classes with virtual functions
 - child classes of those classes

Virtual Table Pointer

SUV	SUV	Jeep	Van	Jeep	Sedan	Sedan	Van
------------	------------	-------------	------------	-------------	--------------	--------------	------------

Virtual Table Pointer

- the compiler adds a hidden variable

[illegible]

Virtual Table Pointer

- the compiler also adds a virtual table of functions for each class

SUV	SUV	Jeep	Van	Jeep	Sedan	Sedan	Van
*__vptr	*__vptr	*__vptr	*__vptr	*__vptr	*__vptr	*__vptr	*__vptr

SUV virtual table

Jeep virtual table

Van virtual table

Sedan virtual table

Virtual Table Pointer

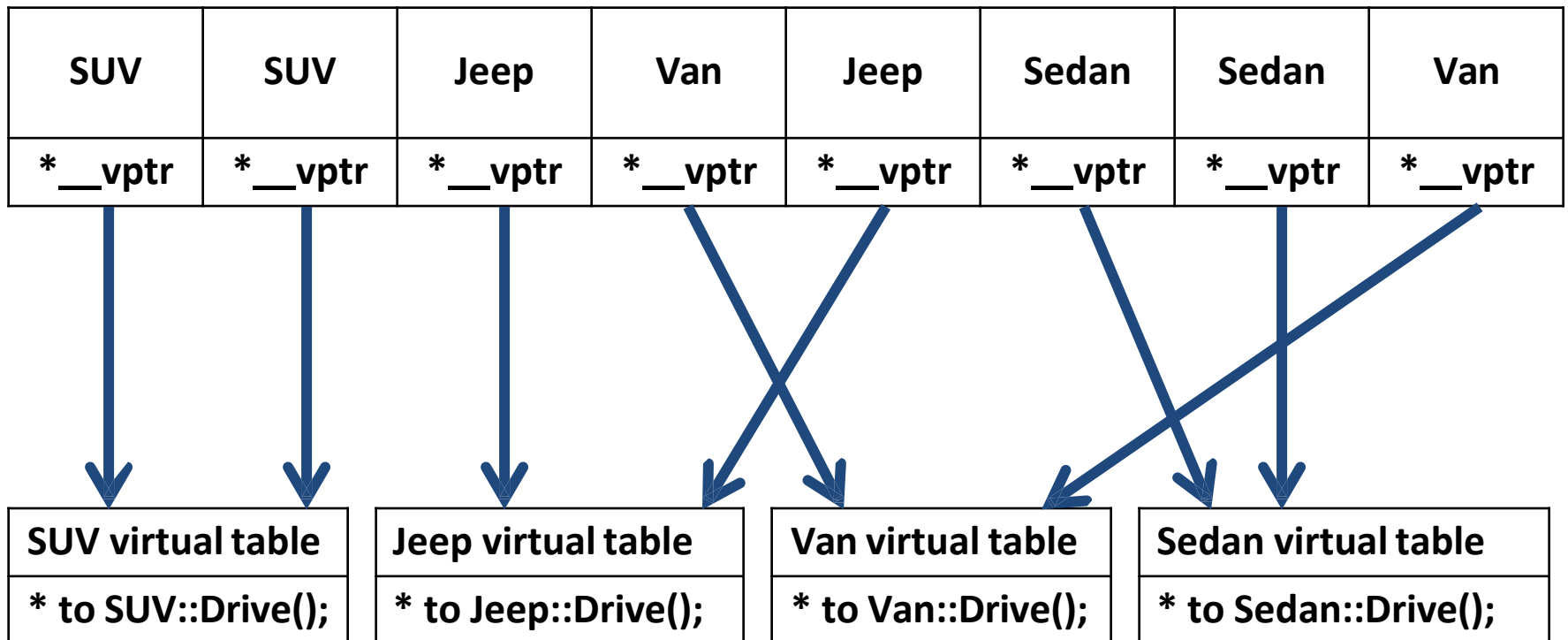
- each virtual table has pointers to each of the virtual functions of that class

SUV	SUV	Jeep	Van	Jeep	Sedan	Sedan	Van
*__vptr	*__vptr	*__vptr	*__vptr	*__vptr	*__vptr	*__vptr	*__vptr

SUV virtual table	Jeep virtual table	Van virtual table	Sedan virtual table
* to SUV::Drive();	* to Jeep::Drive();	* to Van::Drive();	* to Sedan::Drive();

Virtual Table Pointer

- the hidden variable points to the appropriate virtual table of functions



Virtual Everything!

- in Java, all functions are virtual by default
 - everything seems to work fine for Java
- why don't we make all our functions virtual in C++ classes?
 - ???

Virtual Everything!

- in Java, all functions are virtual by default
 - everything seems to work fine for Java
- why don't we make all our functions virtual in C++ classes?
 - non-virtual functions can't be overridden (in the context of parent class pointers)
 - creates unnecessary overhead

Outline

- Review of Inheritance
- Polymorphism
 - Limitations
 - Virtual Functions
 - Abstract Classes & Function Types
 - Virtual Function Tables
 - Virtual Destructors/Constructors

Virtual Destructors

```
Vehicle *vehicPtr = new Car;  
delete vehicPtr;
```

- for any class with virtual functions, you must declare a virtual destructor as well
- why?

Virtual Destructors

```
Vehicle *vehicPtr = new Car;  
delete vehicPtr;
```

- for any class with virtual functions, you must declare a virtual destructor as well
- non-virtual destructors will only invoke the base class's destructor

Virtual Constructors

- not a thing... why?

Virtual Constructors

- not a thing... why?
- we use polymorphism and virtual functions to manipulate objects **without** knowing type or having complete information about the object
- when we construct an object, we **have** complete information
 - there's no reason to have a virtual constructor