CMSC202
Computer Science II

Lecture 17 -
Polymorphism (cont'd)

CMSC 202 Faculty

Based on slides by Chris Marron, Katie Gibson, and Walter Savitch

Last Class We Covered

- Review of inheritance
- Overriding (vs overloading)

- Understanding polymorphism
  - Limitations of inheritance
  - Virtual functions
  - Abstract classes & function types

Any Questions from Last Time?

Today’s Objectives

- Review of polymorphism
  - Limitations of inheritance
  - Virtual functions
  - Abstract classes & function types
- Finishing polymorphism
  - Virtual function Tables
  - Virtual destructors/constructors
- Livecoding application

Review of
Inheritance vs Polymorphism

Inheritance

- Using **non-virtual** functions a derived classes can:
  1. **Use** a base class's public and protected functions
     - The function will not exist in the child class. Child uses parent function.
  2. **Replace or Override** a base class's public and protected functions
     - The function has the same signature as the parent class.
  3. **Extend** a base class's public and protected functions
     - The function has a different signature as the parent class

Problem: If I replace the function, how can I still use the parent version of the function?

Scope Resolution

Polymorphism

- **Polymorphism** refers to the ability to associate many meanings with one function name by means of a special mechanism known as **virtual functions** or **late binding**.

Polymorphism

- Using **virtual** functions a derived classes can:
  1. **Override** a base class's public and protected functions
     - The function has the same signature as the parent class.
  2. **Overload** a base class's public and protected functions
     - The function has a different signature as the parent class

Abstract Classes & Function Types

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
- 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***

- Cannot declare abstract class objects
  - Why?
  - They have functions whose behavior is not defined!

- This means abstract classes can only be used as ***base classes***

Overview of Polymorphism

- Assume we have `Vehicle *vehiclePtr = &myCar;`
- And this method call: `vehiclePtr->Drive();`

prototype	Vehicle class	Car class
`void Drive()`		
`virtual void Drive()`	- Can implement function	- Can implement function
`virtual void Drive() = 0`	- **<u>Cannot</u>** implement function	- **<u>Must</u>** implement function

Overview of Polymorphism

- Assume we have `Vehicle *vehiclePtr = &myCar;`
- And this method call: `vehiclePtr->Drive();`

prototype	Vehicle class	Car class
`void Drive()`	• Can implement function • Can create Vehicle	• Can implement function • Can create Car • Calls `Vehicle::Drive`
`virtual void Drive()`	• Can implement function • Can create Vehicle	• Can implement function • Can create Car • Calls `Car::Drive`
`virtual void Drive() = 0`	• **Cannot** implement function • **Cannot** create Vehicle	• **Must** implement function • Can create Car • Calls `Car::Drive`

Overview of Polymorphism

- Assume we have `Vehicle *vehiclePtr = &myCar;`
- And this method call: `vehiclePtr->Drive();`

prototype	Vehicle class	Car class
`void Drive()`	- Can implement function - Can create Vehicle	- Can implement function **rive**
`vi`	lement functio te Vehicle	tion
`vi` `Dr`	mplement function create Vehicle	- Must implement function - Can create Car - Calls **`Car::Drive`**

This is a ***pure virtual*** function, and Vehicle is now an ***abstract*** class

If no **`Car::Drive`** implementation, calls **`Vehicle::Drive`**

Virtual Function Tables

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:
  - 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

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

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
*** to SUV::Drive();**

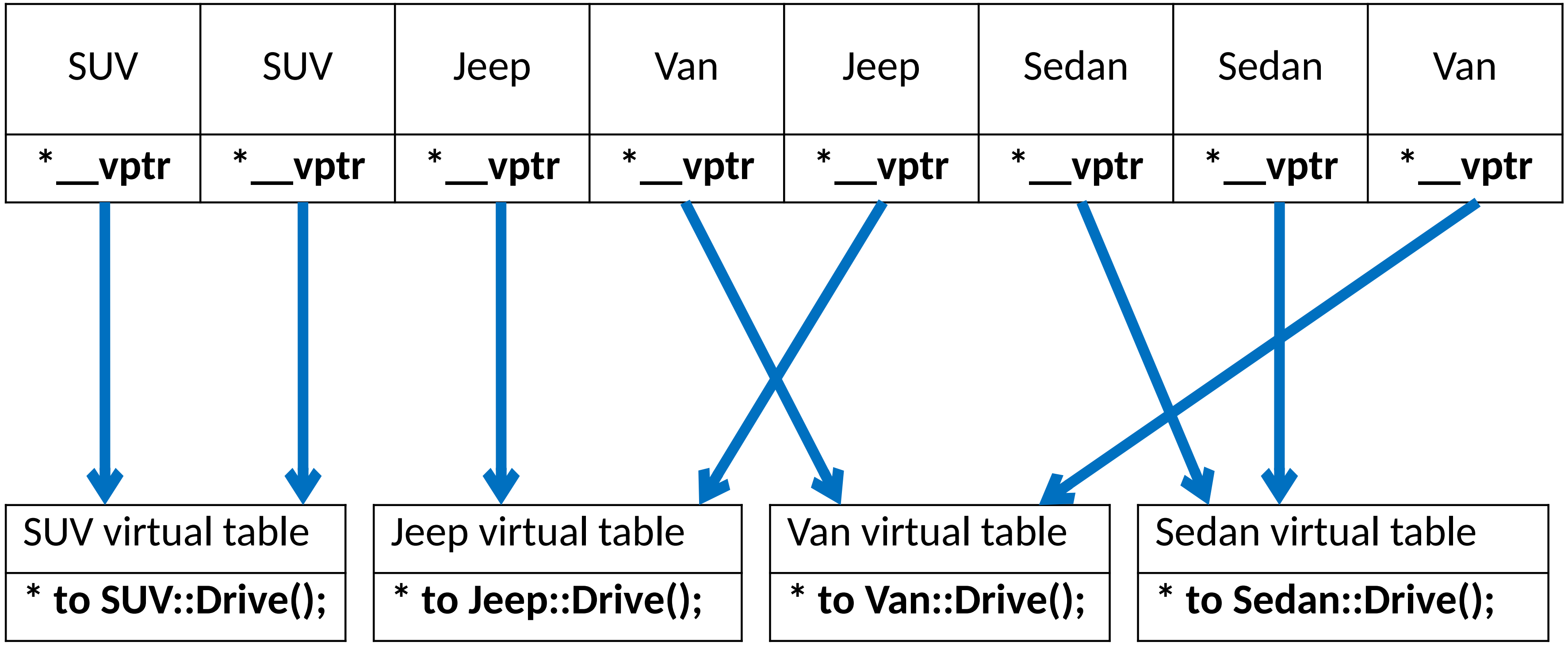
Jeep virtual table
*** to Jeep::Drive();**

Van virtual table
*** to Van::Drive();**

Sedan virtual table
*** to Sedan::Drive();**

Virtual Table Pointer

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



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

SUV virtual table
*** to SUV::Drive();**

Jeep virtual table
*** to Jeep::Drive();**

Van virtual table
*** to Van::Drive();**

Sedan virtual table
*** to Sedan::Drive();**

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?
    - Non-virtual destructors will only invoke the base class's destructor

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

Livecoding

- Pets (Bird, Cat, and Dog)
    - All Animals can: Eat(), Speak(), and Perform()
- Vector of Animal pointers – what happens?

LIVECODING!!!

Live Coding

Lec17–> pet.cpp

Announcements

- Prelab Quizzes (4 pts)
  - Released every Friday by 10am on Blackboard
  - Due every Monday by 10am on Blackboard
- Lab (6 pts)
  - In Engineering building during scheduled time!
- Project 4
  - Due on Tuesday, April 15th at 8:59pm on GL
- Exam 2 Review
  - On Friday, April 4th from 2-4pm in LH 1 (Movie Theater)
- Exam 2
  - In person during scheduled lecture on Wednesday, April 9th and Thursday, April 10th

Next Time: Templates