CMSC202
Computer Science II for Majors

Lecture 14 –
Polymorphism

Dr. Katherine Gibson

Last Class We Covered

- Miscellaneous topics:
  - Friends
  - Destructors
    - Freeing memory in a structure
  - Copy Constructors
  - Assignment Operators

Any Questions from Last Time?

Today's Objectives

- To review inheritance
- To learn about overriding
- To begin to understand polymorphism
  - Limitations of Inheritance
  - Virtual Functions
  - Abstract Classes & Function Types
  - Virtual Function Tables
  - Virtual Destructors/Constructors

Review of Inheritance

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

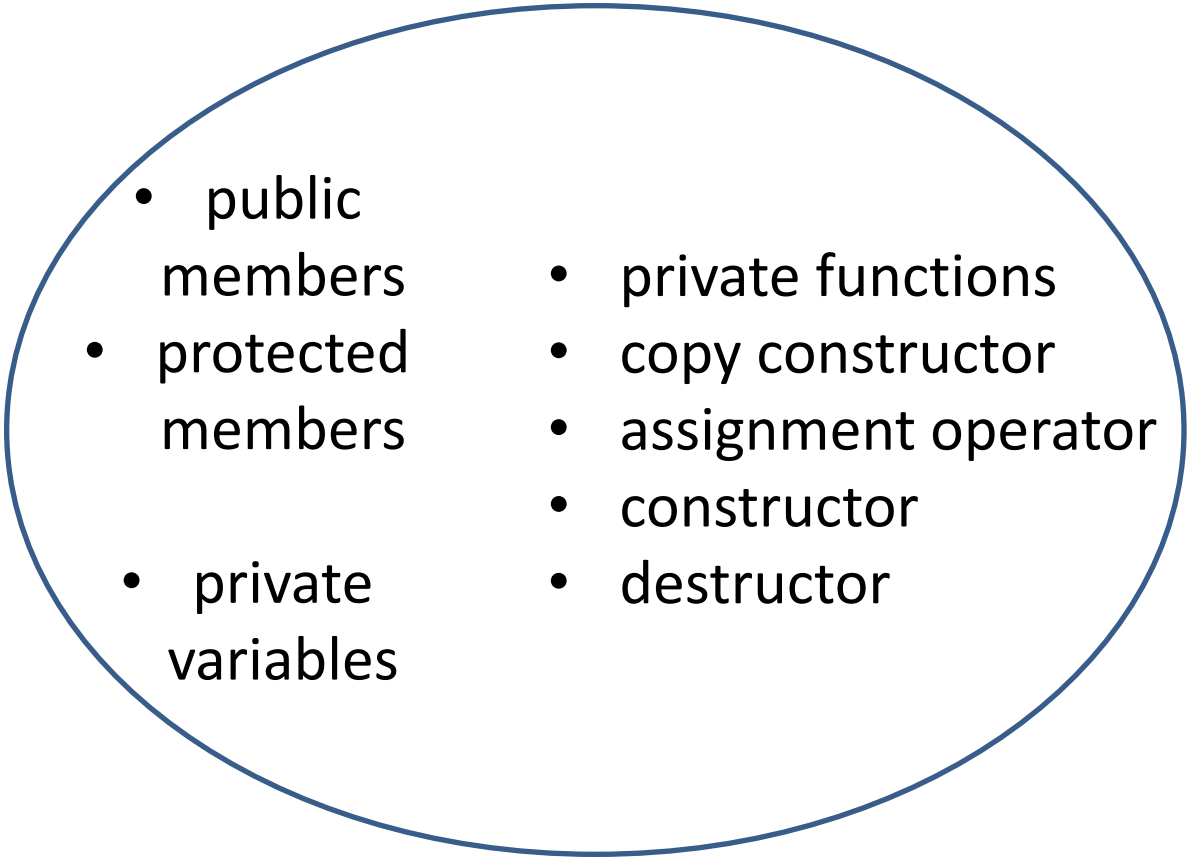
- Child class has direct access to
  - Parent member functions and variables that are:
    - Public
    - Protected
- Parent class has direct access to:
  - **???????** in the child class

Review of Inheritance

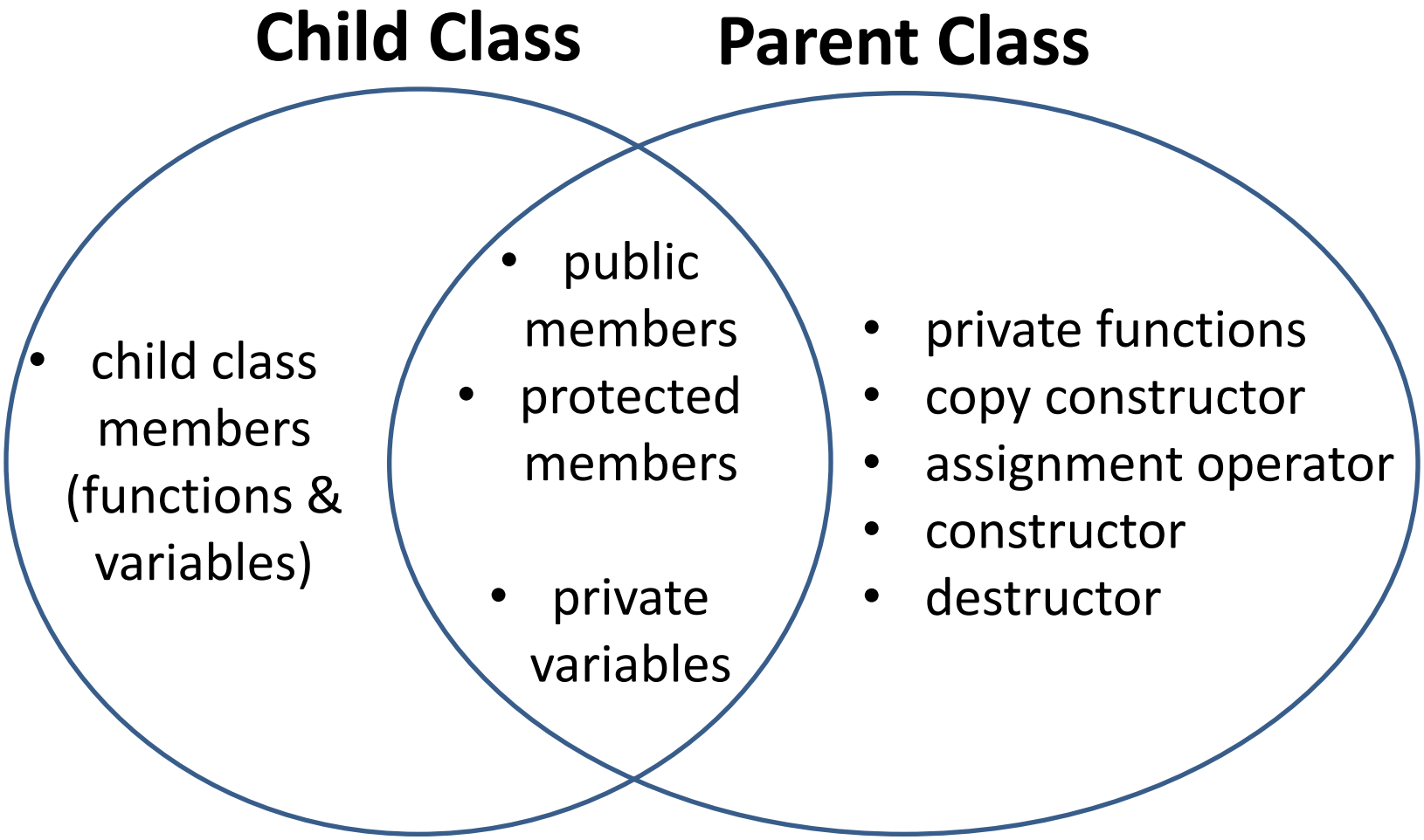
- Specialization through sub classes

- 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



What is Inherited



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**

Overloading vs Overriding

- ***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 Examples

- For these examples, the Vehicle class now contains these public functions:

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

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

Basic Overriding Example

- Car class overrides Upgrade()

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

- When Upgrade() is called on a object of type Car, what happens?
  - 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 original 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 does something we weren't expecting!

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

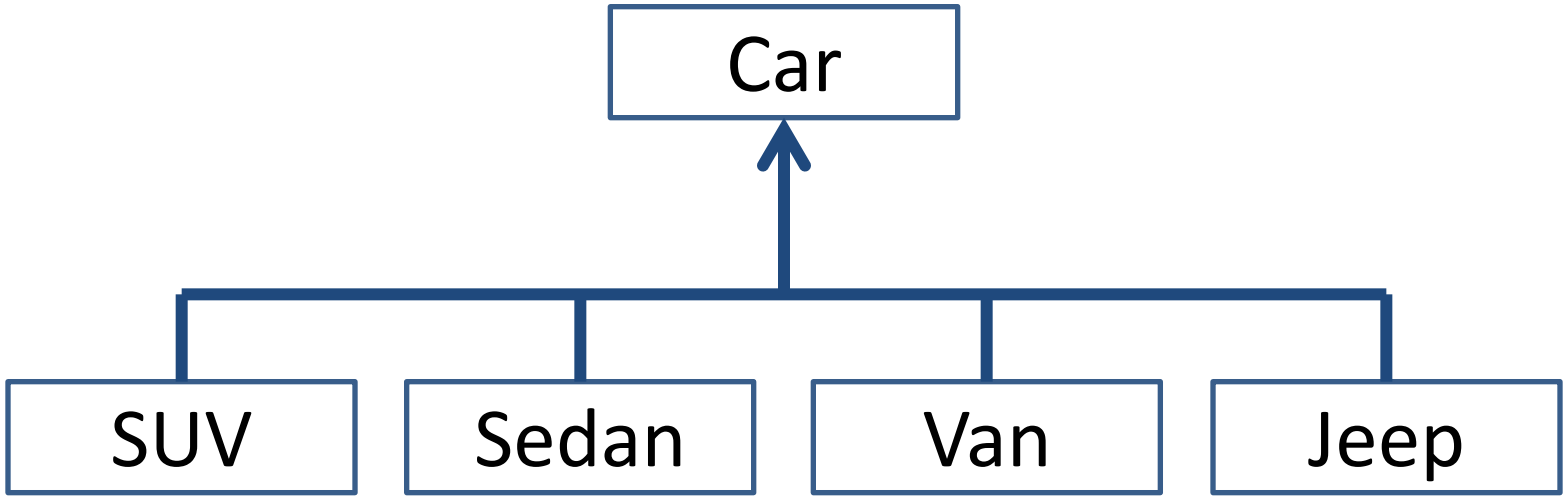
- To overload, we must have both original and overloaded functions in child class

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

- The “original” one parameter function can then explicitly call the parent function

Limitations of Inheritance

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?
  - Multiple vectors, one for each type (boo!)
  - Combine all the child classes into one giant class with info for every kind of car (yuck! don't do this!)
- We can accomplish this with a single vector
  - Using ***polymorphism***

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

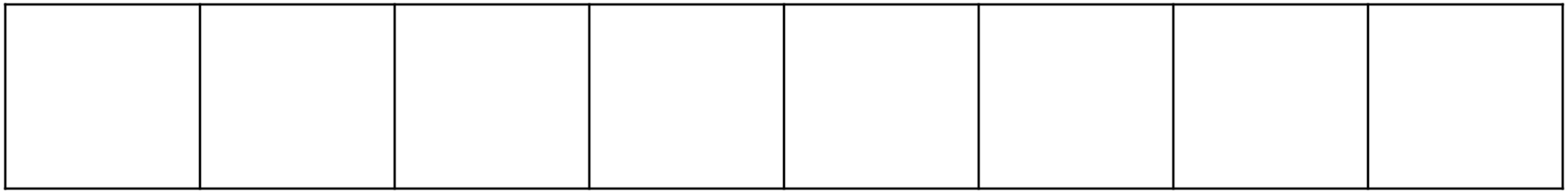
```
Vehicle *vehiclePtr = &myCar;
```

- Why is this valid?
  - 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

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

```
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();
```

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

Virtual Functions

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();
```

- 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

- 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();
```

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

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 • Calls `Car::Drive`
`vi`	lement functi te Vehicle	tion
`virtual void Drive() = 0`	• **Cannot** implement function • **Cannot** create Vehicle	• **Must** implement function • Can create Car • Calls `Car::Drive`

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

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

Announcements

- Project 3 is out – get started now!
  - Due Thursday, March 31st
- Exam 2 is in 1.5 weeks
  - Will focus heavily on:
    - Classes
    - Inheritance
    - Linked Lists
    - Dynamic Memory