Scientific Programming with C++

OBJECTS AND CLASSES

Learning objectives

- After this lecture and related assignments, you will…
  - understand what classes are
  - understand how classes differ from objects
  - have a basic understanding of how inheritance works
  - have a basic understanding of overriding and overloading methods
  - be introduced to the principle of encapsulation

Classes

- Define how a programmatical entity should behave
  - Properties: variables and their visibility, member functions and their visibility
  - Models of real-world entities
  - Vehicles
    - Variables: engine horsepower, number of tires, mass, width, height, length, price
    - Member functions: accelerate, brake, turn
  - Animal cells
    - Variables: diameter, salinity, temperature, structural integrity, organelles (member classes)
    - Member functions: grow, synthesize, duplicate, endocytosis, apoptosis

Objects

- Instances of a class
  - The practical implementation to the blueprints provided by a class
- Examples
  - “Vehicle” class
    - Audi A6 car
    - Harley Davidson Road King motorcycle
  - “AnimalCell” class
    - Neurons
    - Red blood cells
    - Alpha cells

Objects

```
class Vehicle {
    public:
        int numberOfTires = 0;
        void brake() { };
};
```

- Members are accessed with the direct member access operator (.)
- Members of pointers are accessed with the arrow operator (->)

```
int main()
{
    Vehicle audiA6; // create a new vehicle object
    audiA6.numberOfTires = 4; // set its number of tires
    audiA6.brake(); // call the brake function

    Vehicle* anotherCar; // create a vehicle pointer
    anotherCar = &audiA6; // have it point to the audiA6 object
    int nTires = anotherCar->numberOfTires; // get its number of tires
    anotherCar->brake(); // call the brake function
    (*anotherCar).brake(); // same as above line
}
```

Classes: an example

```cpp
class Rectangle {
    public:
        Rectangle(double height, double width) {
            height_ = height;
            width_ = width;
            updateArea();
        }
        Rectangle(double side) {
            height_ = side;
            width_ = side;
            updateArea();
        }
        double getArea() { return area_; }
    protected:
        // not used in this example
    private:
        void updateArea() { area_ = height_ * width_; }
        double area_ = 0;
        double height_ = 0;
        double width_ = 0;
};
```

```cpp
int main()
{
    Rectangle rect(2.5, 1.0);
    Rectangle square(5);
    std::cout << rect.getArea() << std::endl;
    std::cout << square.getArea() << std::endl;
}
```

```
2.5
25
```

Classes: ingredients

- Unique name and namespace
- Constructors and destructors
  - Define what happens when the object is created and destroyed
- Member variables (called “attributes”) and their visibility
- Member functions (called “methods”) and their visibility
- Inheritance
- Function overloading and overriding
- Operator overloading and overriding

Writing classes

- Usually in separate files
  - Header (ClassName.h)
    - Defines the class, attributes and their visibility, inheritance, declares methods and their visibility
  - Method definitions (ClassName.cpp)
- Begin with the header
- Conventions
  - UpperCamelCase for class names (“AnimalCell”)
  - Private variables end with an underscore (“salinity_”)
  - Header doesn’t contain complicated method definitions

Visibility

- Access specifier keywords
  - Public
    - Accessible outside the class
    - Can be accessed by derived classes
  - Protected
    - Inaccessible outside the class
    - Can be accessed by derived classes
  - Private
    - Inaccessible outside the class
      - Use getter and setter functions (example on next slide)
    - Cannot be accessed by derived classes
- Class attributes and methods are private by default

Visibility example

```
class Meal {
    public:
        Meal(double priceIn, int caloriesIn)
        {
            price = priceIn;
            calories = caloriesIn;
        }
        void setCalories(int newCalories) { calories = newCalories; }
        int getCalories() { return calories; }
        double price = 0;
    protected:
    private:
        int calories = 0;
};
```

```
Meal tortillaChips(1.50, 773);
tortillaChips.price = 1.80; // valid
tortillaChips.calories = 800; // error
tortillaChips.getCalories(); // valid
```

Why should you care about visibility?

- In theory, you could just make everything public
- Visibility helps to manage larger projects
  - When debugging, you know private members will only have been accessed from one or two files
    - Public members could have been accessed anywhere
  - Simplicity: variables and other data are only available where they are used
- Encapsulation: one part of the code can be changed without affecting the other
- If you make your own library, you can let others use your classes without exposing the implementation details

Inheritance

- Classes can derive properties from other classes
  - Attributes
  - Methods
- The class inherited from is called the base or parent class
- The class that inherits is called the derived or child class
  - Can have its own attributes and methods in addition to inherited ones
  - Can change the behaviour of its inherited properties without changing the base class

Inheritance: conceptual example

- If your code has classes “Car” and “Motorcycle”, they have common properties that work similarly for both
  - Attributes: number of tires, engine horsepower
  - Methods: accelerate, brake
  - Note: they also have some distinct properties, otherwise they shouldn’t be different classes
- Instead of writing the same common properties for each class, define them in a parent class “Vehicle”
  - Then have Car and Motorcycle inherit from Vehicle

Inheritance: code example

Car is made to inherit from Vehicle with access specifier "public"

Constructor of Car assigns a value to inherited attribute "numberOfTires"

In addition to inherited members, Car has method "openDoors" that the parent class doesn't

```
class Vehicle {
    public:
        void brake();
        void checkTires() {
            std::cout << "Number of tires: " << numberOfTires << std::endl;
        }
        int numberOfTires = 0;
};

class Car : public Vehicle {
    public:
        Car() { numberOfTires = 4; }
        void openDoors();
};

class Motorcycle : public Vehicle {
    public:
        Motorcycle() { numberOfTires = 2; }
        void deployKickstand();
};
```

Although checkTires() is called through Car and Motorcycle, its behaviour comes from the parent class

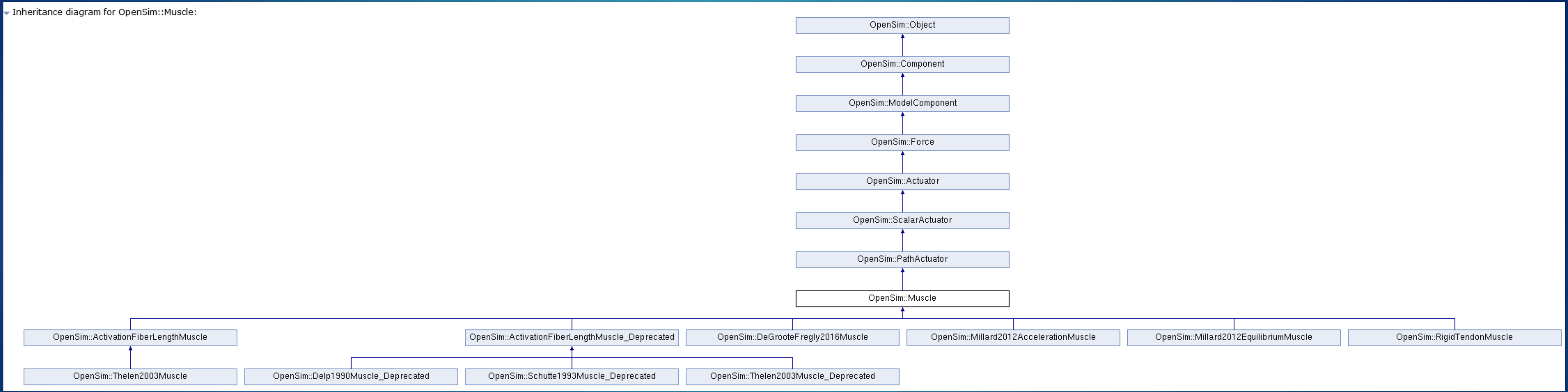
```
int main()
{
    Car someCar;
    someCar.checkTires();

    Motorcycle someMotorcycle;
    someMotorcycle.checkTires();
}
```

```
Number of tires: 4
Number of tires: 2
```

Inheritance: access specifiers

- Same keywords as in member visibility
  - public, protected, private
- Private inheritance (default)
  - Public and protected members of base class become private members of the derived class
- Protected inheritance
  - Public and protected members of the base class become protected members of the derived class
- Public inheritance
  - Public members of the base class become public members of the derived class
  - Protected members of the base class become protected members of the derived class

Example: inheritance spans multiple levels



Inheritance diagram for OpenSim::Muscle:
OpenSim::Object
OpenSim::Component
OpenSim::ModelComponent
OpenSim::Force
OpenSim::Actuator
OpenSim::ScalarActuator
OpenSim::PathActuator
OpenSim::Muscle
OpenSim::ActivationFiberLengthMuscle
OpenSim::ActivationFiberLengthMuscle_Deprecated
OpenSim::DeGrooteFregly2016Muscle
OpenSim::Millard2012AccelerationMuscle
OpenSim::Millard2012EquilibriumMuscle
OpenSim::RigidTendonMuscle
OpenSim::Thelen2003Muscle
OpenSim::Delp1990Muscle_Deprecated
OpenSim::Schutte1993Muscle_Deprecated
OpenSim::Thelen2003Muscle_Deprecated

Image taken from OpenSim API documentation (https://simtk.org/api_docs/opensim/api_docs/classOpenSim_1_1Muscle.html)

Virtual functions

- Use the “virtual” keyword in the base class when declaring methods that child classes should re-define
  - Omitting “virtual” will not result in a compiler error
    - But can cause unwanted behaviour when accessing classes through pointers
      - See next two slides
  - If no redefinition exists in the derived class, the method will default to the parent’s version
  - You may use the identifier “override” when declaring the method in the derived class
    - The compiler will then tell you if it cannot find the function in the base class
    - Additional safety

Example: overriding without virtual

```cpp
class Vehicle {
    public:
        void brake() {
            std::cout << "Generic braking noises!" << std::endl;
        }
        int numberOfTires = 0;
};

class Car : public Vehicle {
    public:
        Car() { numberOfTires = 4; }
};

class Motorcycle : public Vehicle {
    public:
        Motorcycle() { numberOfTires = 2; }
        void brake() {
            std::cout << "Motorcycle braking noises!" << std::endl;
        }

};
```

```cpp
void hitTheBrakes(Vehicle* vehicle) {
    vehicle->brake();
}

int main()
{
    Car someCar;
    someCar.brake();
    hitTheBrakes(&someCar);

    Motorcycle someMotorcycle;
    someMotorcycle.brake();
    hitTheBrakes(&someMotorcycle);
}
```

```
Generic braking noises!
Generic braking noises!
Motorcycle braking noises!
Generic braking noises!
```

Example: overriding with virtual

```cpp
class Vehicle {
    public:
        virtual void brake() {
            std::cout << "Generic braking noises!" << std::endl;
        }
        int numberOfTires = 0;
};

class Car : public Vehicle {
    public:
        Car() { numberOfTires = 4; }
};

class Motorcycle : public Vehicle {
    public:
        Motorcycle() { numberOfTires = 2; }
        void brake() {
            std::cout << "Motorcycle braking noises!" << std::endl;
        }

};
```

```cpp
void hitTheBrakes(Vehicle* vehicle) {
    vehicle->brake();
}

int main()
{
    Car someCar;
    someCar.brake();
    hitTheBrakes(&someCar);

    Motorcycle someMotorcycle;
    someMotorcycle.brake();
    hitTheBrakes(&someMotorcycle);
}
```

```
Generic braking noises!
Generic braking noises!
Motorcycle braking noises!
Motorcycle braking noises!
```

Operator overloading example

```cpp
class Population {
public:
    Population() {};
    Population(unsigned int indiv, double health) {
        numIndividuals = indiv;
        overallHealth = health;
    }
    // define + operator as something that combines the individuals and recalculates the mean (overall) health
    Population operator+(const Population& popul) {
        Population combinedPopulation;
        combinedPopulation.numIndividuals = numIndividuals + popul.numIndividuals;
        combinedPopulation.overallHealth = (numIndividuals * overallHealth + popul.numIndividuals * popul.overallHealth) /
(combinedPopulation.numIndividuals);
        return combinedPopulation;
    }
    unsigned int numIndividuals = 0;
    double overallHealth = 0;
};

int main()
{
    // a population of 15 individuals with an overall health of 100
    Population healthy(15, 100);
    // a population of 4 individuals with an overall health of 23
    Population unhealthy(4, 23);
    // let the population come together
    Population combined = healthy + unhealthy;
    std::cout << "Combined population, individuals: " << combined.numIndividuals << ", health: " << combined.overallHealth;
}
```

```
Combined population, individuals: 19, health: 83.7895
```

Data structures

- Class-like entities, declared with “struct” instead of “class”

```
class Animal{
    int legs = 6;
};
```

```
struct Animal {
    int legs = 6;
};
```

- All members are public by default
  - Class members are private by default
  - Otherwise used identically to classes
- Convention: used to store records of variables
  - Example: records of books, records of customers, records of data points from a measurement

Summary

- Classes are instructions, objects are the instances created from them
- Most characteristics of C++ revolve around classes and their inheritance
  - Polymorphism
    - The same entity has different behaviour in different situations
      - overloading = giving the same function or operator identifier several different definitions
        - Compile-time polymorphism
      - overriding = redefining a method from parent
        - Run-time polymorphism
  - Encapsulation
    - Wrapping data in a single unit
      - Protects private data
- Classes can be used to model real-world entities